E6156 – Topics in SW Engineering: Cloud Computing

Lecture 2: Microservices, REST, Containers, PaaS, S3

Contents

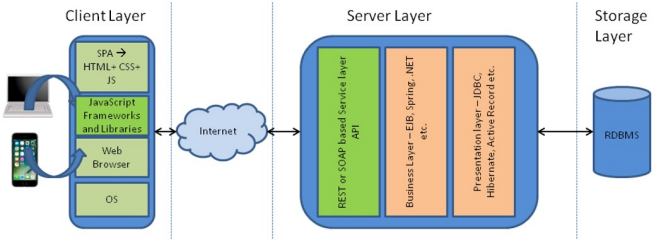
Contents

- Some considerations.
- Static Web Hosting – S3
- Microservices:
  - Concept and motivation.
  - SOLID, Twelve Factor Apps, XYZ-Scale Cube, ...
- REST, Part I
  - Core concepts.
  - URL patterns.
- Introduction to PaaS and Elastic Beanstalk
  - Concepts
  - Walkthrough
- Introduction to Containers, Docker, etc.
  - Concepts
  - Walkthrough
- Project – Sprint 1

Some Considerations

Simple Flask Application – Configuration

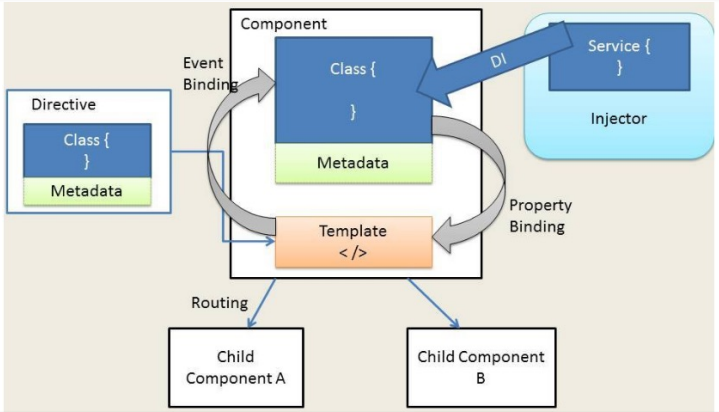
- Possible configurations:
  - There are three “things:” web server, application server, DB server.
  - A “thing” can be on your laptop or on AWS.
  - You want your code to be the same and get information from the configuration.
  - One simple approach is environment variables for web application.
  - The angular application can use the URL because. Why would the cloud deployed angular application access your local web application.

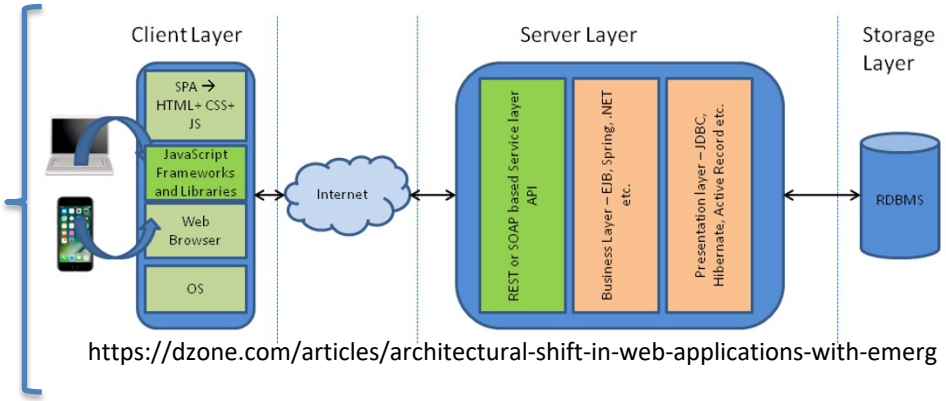


- Show code for environment variables and how to set environment variables.
  - Locally in PyCharm
  - On EC2 (.bash_profile)
  - Show use of URL in Typescript.
- These approaches are “OK” but there are more secure approaches.
- Also, remember to set security group rules properly.

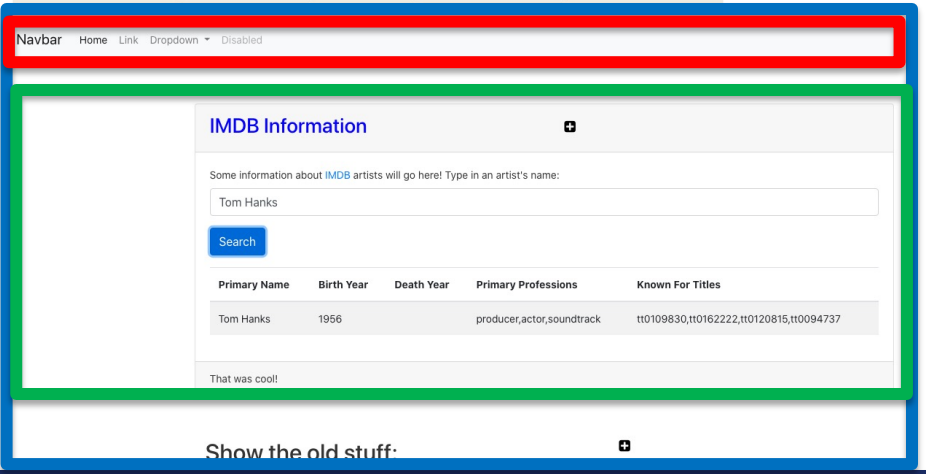
Static Web Hosting

Web Application Architecture





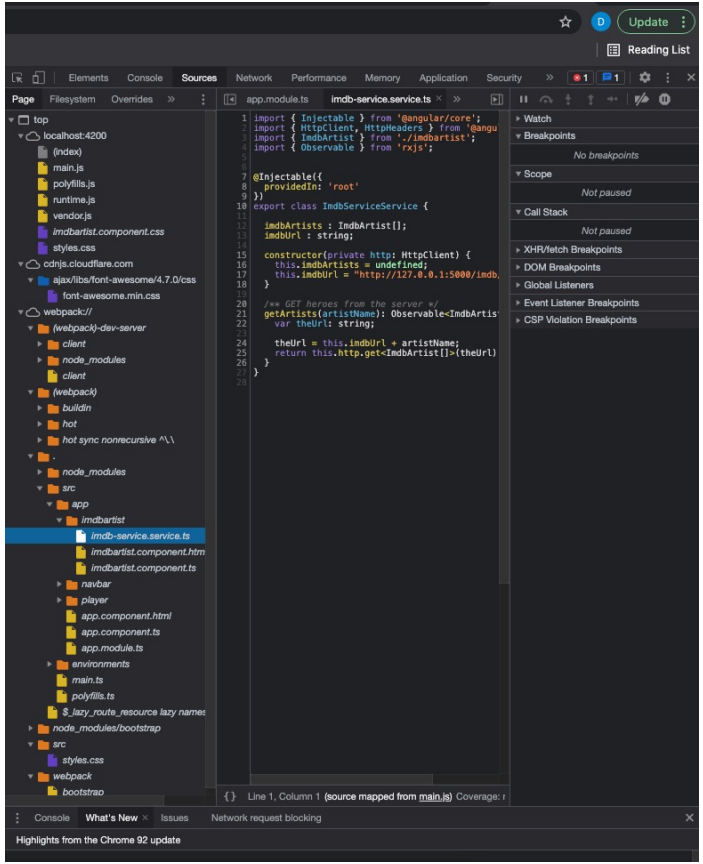
https://dzone.com/articles/architectural-shift-in-web-applications-with-emerg



- app.component
- navbar.component
- imdbartist.component
  - Template (HTML)
  - CSS
  - Component implementation
  - Service

Show the Code

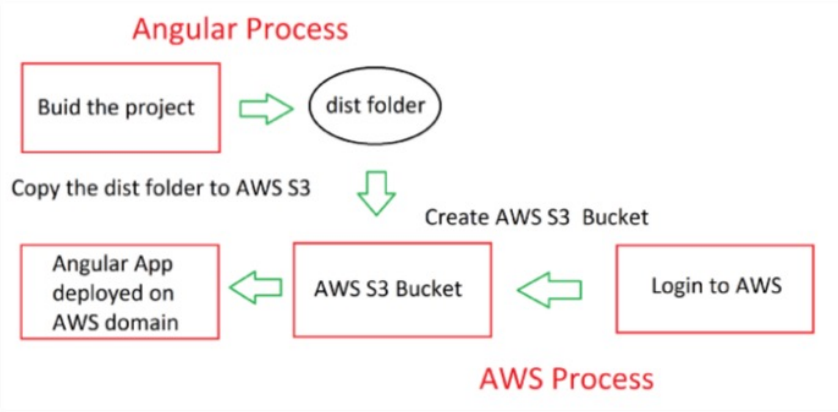
What's in the Browser



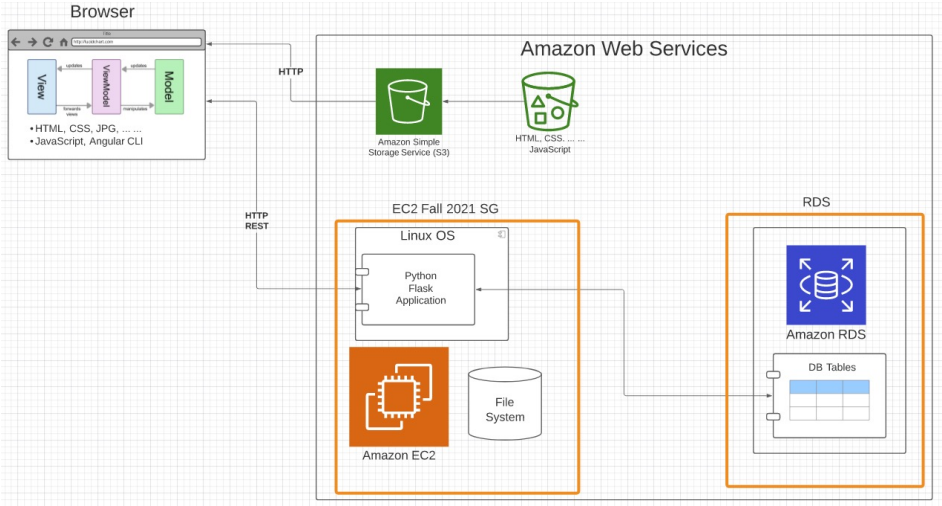
- The browser is an application that interprets the files:
  - HTML, CSS, etc. define the document and content.
  - JavaScript defines dynamic behavior.
- Browser loads index.html
  - index.html contains links to files.
  - The browser loads the linked files.
  - These also have links, which the browser loads.
  - etc.
- How did all of this get in the browser?
  - A *web server* delivers these files from "the file system."
  - During development, Angular uses a simple, embedded web server.
  - "build" compiles the files into a webpack (see dist folder)
- Deploying the application to the cloud →
  - We have seen deploying the application logic to the cloud.
  - What about the content?
- **Notes:**
  - Show loading the demo-ui.
  - Show loading cnn.com.

Deploy to S3

https://baljindersingh013.medium.com/angular-app-deployment-with-aws-s3-42d9008734ab



Block Diagram: Angular-AWS Deployment process



- Compile the browser application (ng build -- prod)
- You can test with ng serve -- prod
- Generates “bundles” that contain:
  - JavaScript application logic.
  - HTML, content, ... In the app logic.

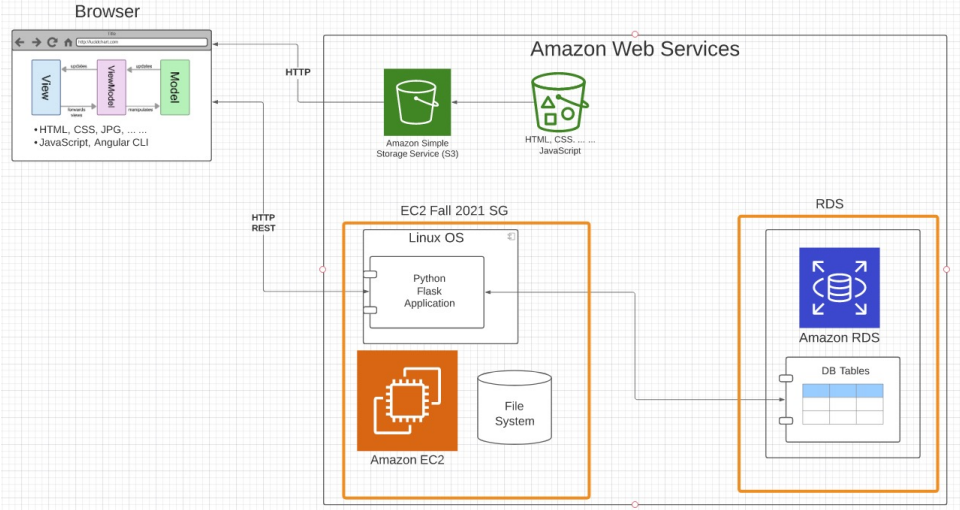
- Create an S3 bucket.
- Upload files from /dist folder.
- Enable static web hosting.
- Set access permissions.
- Set index.html.
- We will learn how to automate later.

Some Comments

- The code is not quite correct:
  The browser code needs to
  - Access the local application during dev/test.
  - The EC2 deployed app during production.
  - I hacked the code. There are better approaches.
- I could have deployed the code on EC2 and delivered from application (Show demo).
  - This would be suboptimal.
  - Web serving and web application serving have different design points for scaling, availability, … …
- I could have deployed MySQL on the EC2 instance, but **web app serving and DB serving have different design points**.
- Why two security groups.
  - Only the web application can connect to the DB.
  - This makes it harder to break into my database and steal information.

```
getImdbServiceUrl(): string {
  const theUrl = window.location.href;
  let result: string;

  // This is some seriously bad code.
  // If you do this on a job interview, you did not learn this in my class.
  if (theUrl.includes('amazonaws')) {
    /* This can change over time */
    result = 'ec2-54-242-71-165.compute-1.amazonaws.com:5000/imdb/artists/';
  }
  else {
    result = 'http://127.0.0.1:5000/imdb/artists/';
  }
  return result;
}
```



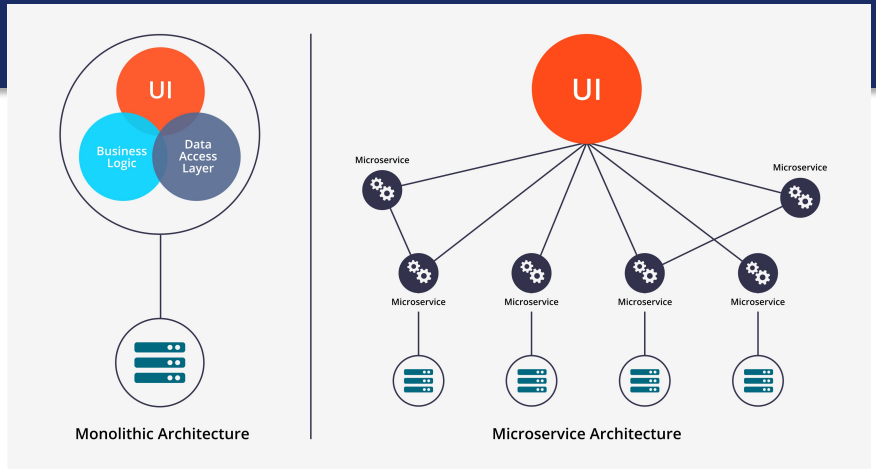
Introduction to Microservices

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.

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



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

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 our project purposes, a thing can be:
  - A user profile service
  - Or
  - A commerce catalog service
  - Or
  - …

  But not both.

12 Factor Applications

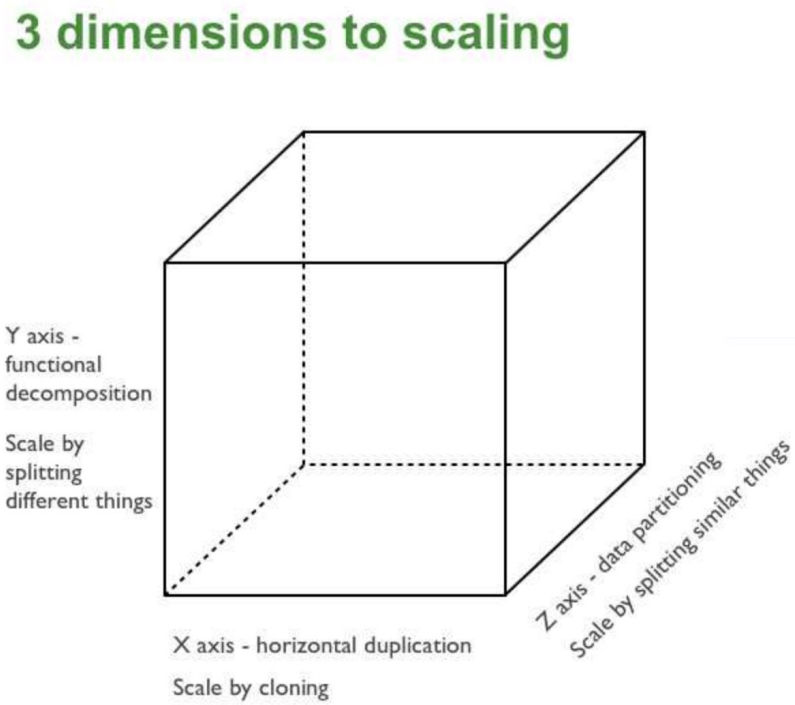
https://dzone.com/articles/12-factor-app-principles-and-cloud-native-microser

Microservices Scalability Cube

The Scale Cube

https://microservices.io/articles/scalecube.html

The book, The Art of Scalability, describes a really useful, three dimension scalability model: the scale cube.



- 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

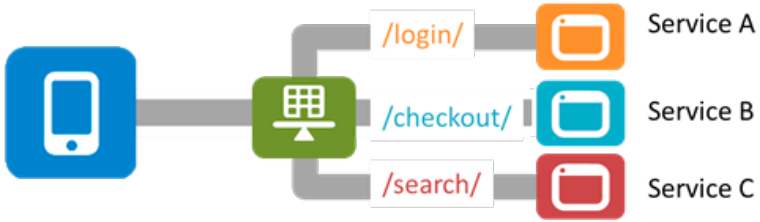
XYZ – Scaling

X-AXIS SCALING

Network name: Horizontal scaling, scale out

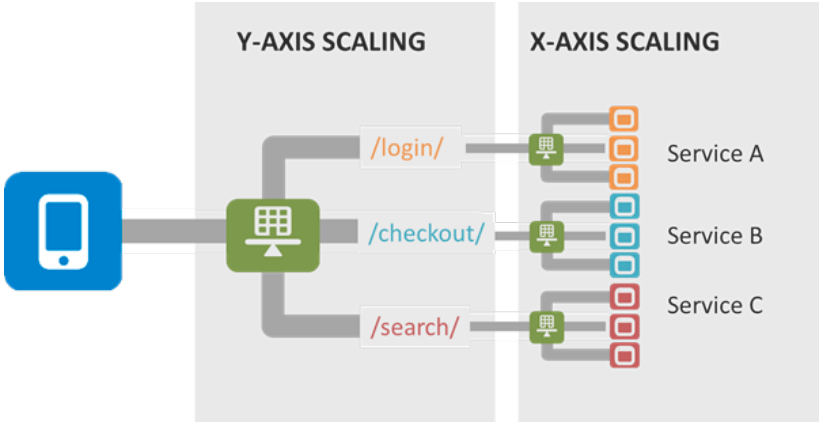
Y-AXIS SCALING

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

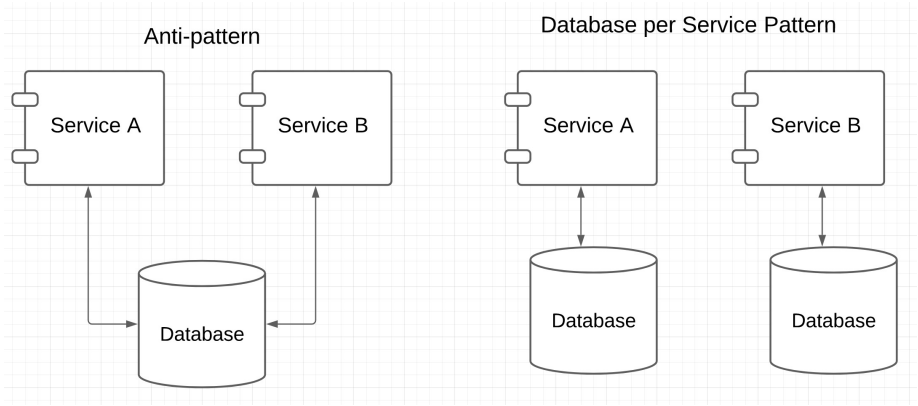


- There are three dimensions.
- A complete solution/environment typical is a mix and composition of patterns.
- Application design and data access determines opitions.

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.

Project Sprint 1

- You are going to build 3 microservices. The exact ones you build will depend on the business/application problem you choose. My examples will be:
  - Customer registration and information. (Mandatory for all teams)
  - Catalog (Things I am selling)
  - Orders (Things people have bought and their status)
- Databases:
  - Use RDS. You can use one RDS instance with 3 separate databases.
  - This approach is not good, but will simplify things for you.
- You will deploy one of the services on each of:
  - EC2 – Directly on OS
  - Elastic Beanstalk
  - In a Docker container on an EC2 instance.

We will discuss more in This lecture.

REST, Part I

Concepts

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.

Resources

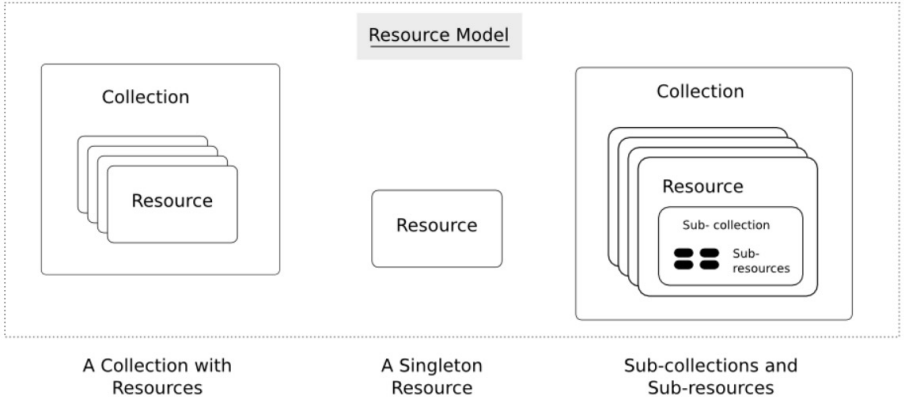
Resources are an abstraction. The application maps to create things and actions.

"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.

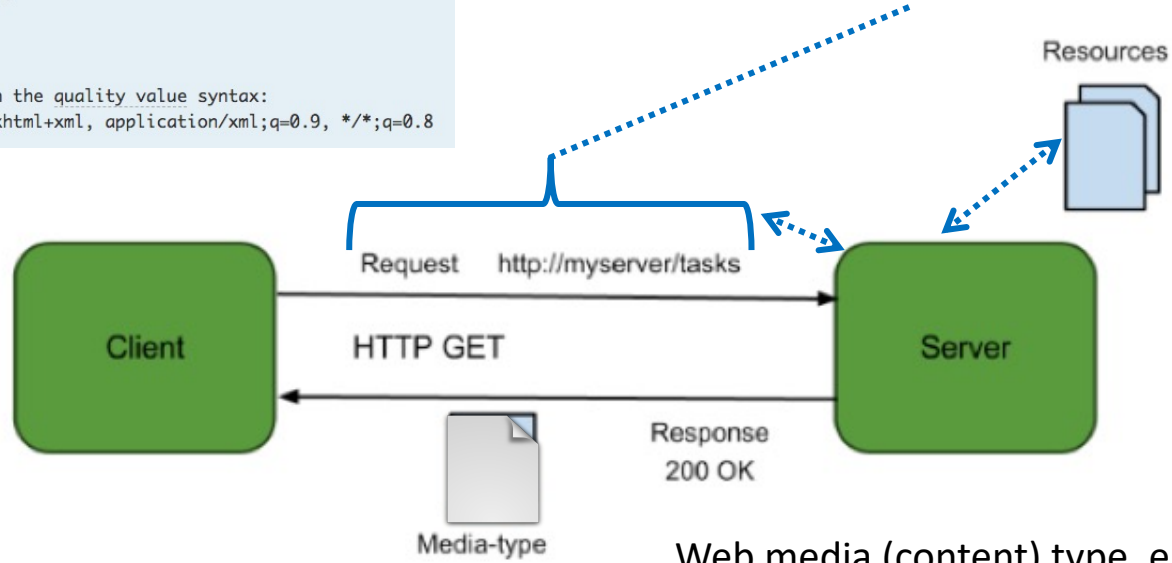
Resources, URLs, Content Types

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.

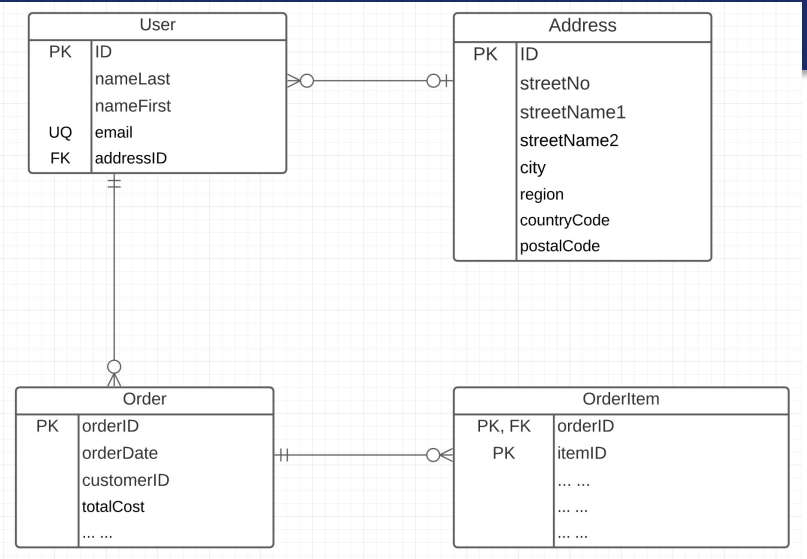
- text/html
- application/json

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 om REST.

Introduction to PaaS
Elastic Beanstalk

Platform-as-a-Service

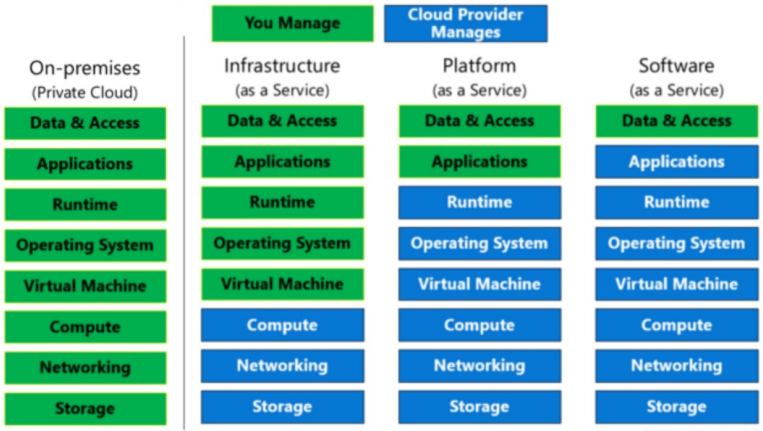
- ” Platform as a service (PaaS) is an enabler for software development where a third-party service provider delivers a platform to customers so they can develop, run, and manage software applications without the need to build and maintain the underlying infrastructure themselves.” (https://www.infoworld.com/article/3223434/what-is-paas-a-simpler-way-to-build-software-applications.html)

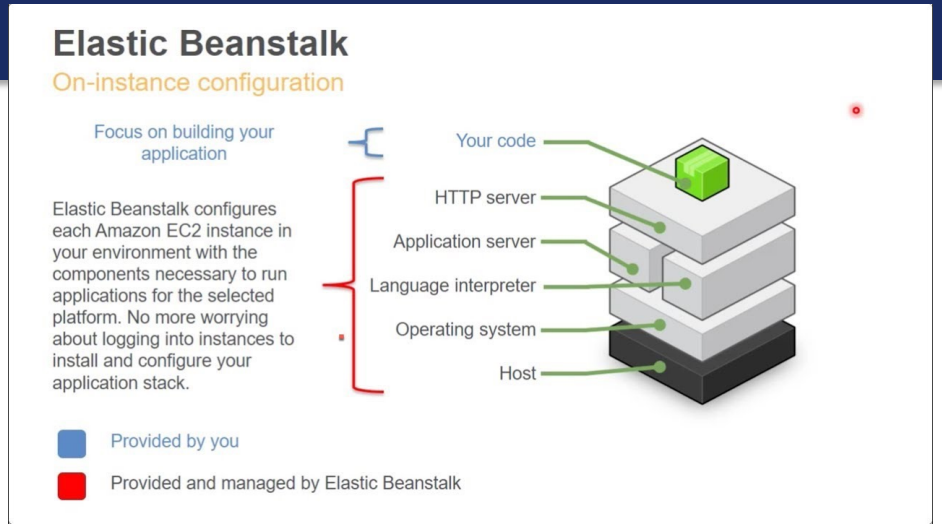
- “Platform as a service (PaaS) is a complete development and deployment environment in the cloud, with resources that enable you to deliver everything from simple cloud-based apps to sophisticated, cloud-enabled enterprise applications.

  ... ...

  PaaS allows you to avoid the expense and complexity of buying and managing software licenses, the underlying application infrastructure and middleware, container orchestrators such as Kubernetes, or the development tools and other resources. You manage the applications and services you develop, and the cloud service provider typically manages everything else. (https://azure.microsoft.com/en-us/overview/what-is-paas/)

Platform-as-a-Service





- Simplistically, PaaS adds two additional layers on top of IaaS:
  - Runtime means the language environment, libraries, etc. your application needs. For example, in our case this will mean a predefined and configured Flask environment.
  - "Middleware is a type of computer software that provides services to software applications beyond those available from the operating system." (https://en.wikipedia.org/wiki/Middleware)
- Basically, you do not have to build an application execution environment by installing libraries, services, ... You application "goes into a premade environment."
- Less flexible and customizable than IaaS but can be more productive to use for application scenarios.

Clarifying PaaS

- This is a little hard to understand until you have done it a few times.
  - IaaS basically stops at the operating system layer. You are responsible for everything else your application code needs:
    - Database
    - Logging Service
    - … …
  - PaaS provides a prebuilt environment with “your code goes here.”
- The easiest way to see the difference is
  - You are using EC2 (or will be).
  - We will use a PaaS (Elastic Beanstalk)
    And you will get a feel for the differences.
- *So, let’s do this.*

Start with Elastic Beanstalk

- There are several reasonably good tutorials on using EB and Flask
  - https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-flask.html
  - https://medium.com/analytics-vidhya/deploying-a-flask-app-to-aws-elastic-beanstalk-f320033fda3c
- AWS and technology evolve. So, sometimes you have to tinker with them.
- The first time you deploy, it can be error prone. But after you succeed, modifying or deploying new applications is "rinse and repeat."
- I normally just:
  - Create an environment with the sample application.
  - Download the sample and un-compress.
  - Modify the application to add functions.
  - Upload a new version.
- There are a couple of issues:
  - Make start with a new environment and add just the packages you need.
  - If you add packages, you must use pip freeze > requirements.txt.
  - Zip acts weird, especially on Mac.
    - Do not zip the folder. Go inside the folder and zip just the contents you need.
    - On Mac: zip -r -X Archive.zip *

- Note: Walk through EB console and development process.
- Show sample from my GitHub repo.

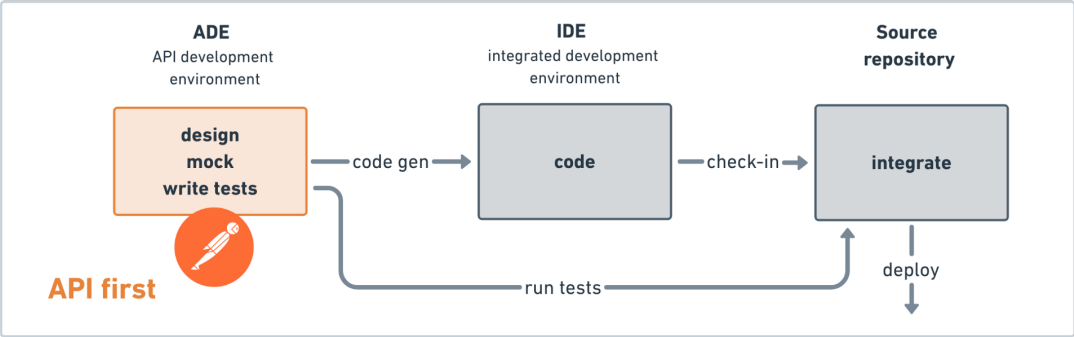
Elastic Beanstalk

- Set up environment.
- Add some code.
- Reupload.
- Explain how this works in the long term.

API First Design

API First Design

- There used to be two approaches to application development:
  - Data First: Define your datamodel and then implement functions/APIs.
  - UX First: Define and mockup the UI. Test with users. Implement functions.
- Cloud Native Microservices often follow API First.
  - Define APIs.
  - Implement mockups, simple tests, etc.
  - Get feedback from potential users and partners.
  - Deploy and document,



(https://medium.com/better-practices/api-first-software-development-for-modern-organizations-fdbfba9a66d3)

(https://medium.com/adobetech/three-principles-of-api-first-design-fa6666d9f694)

There are three principles of API First Design:

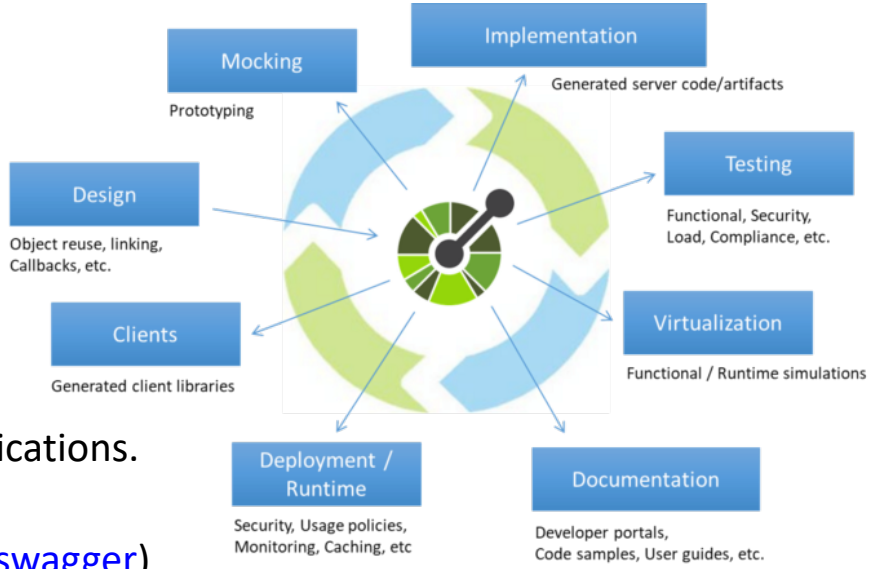
1. Your API is the first user interface of your application

2. Your API comes first, then the implementation

3. Your API is described (and maybe even self-descriptive)

Open API

- "The OpenAPI Initiative (OAI) was created by a consortium of forward-looking industry experts who recognize the immense value of standardizing on how APIs are described. As an open governance structure under the Linux Foundation, the OAI is focused on creating, evolving and promoting a vendor neutral description format." (https://www.openapis.org/about)
- There is an interactive "map" that explains the API concepts (http://openapi-map.apihandyman.io/)
- I (sometimes) use:
  - SwaggerHub
  - Swagger Code Generation
- To produce implementation templates from specifications.
- Demo the project. (https://github.com/donald-f-ferguson/f21-demo-swagger)
- Show SwaggerHub.



Demo Open API (Swagger Hub)

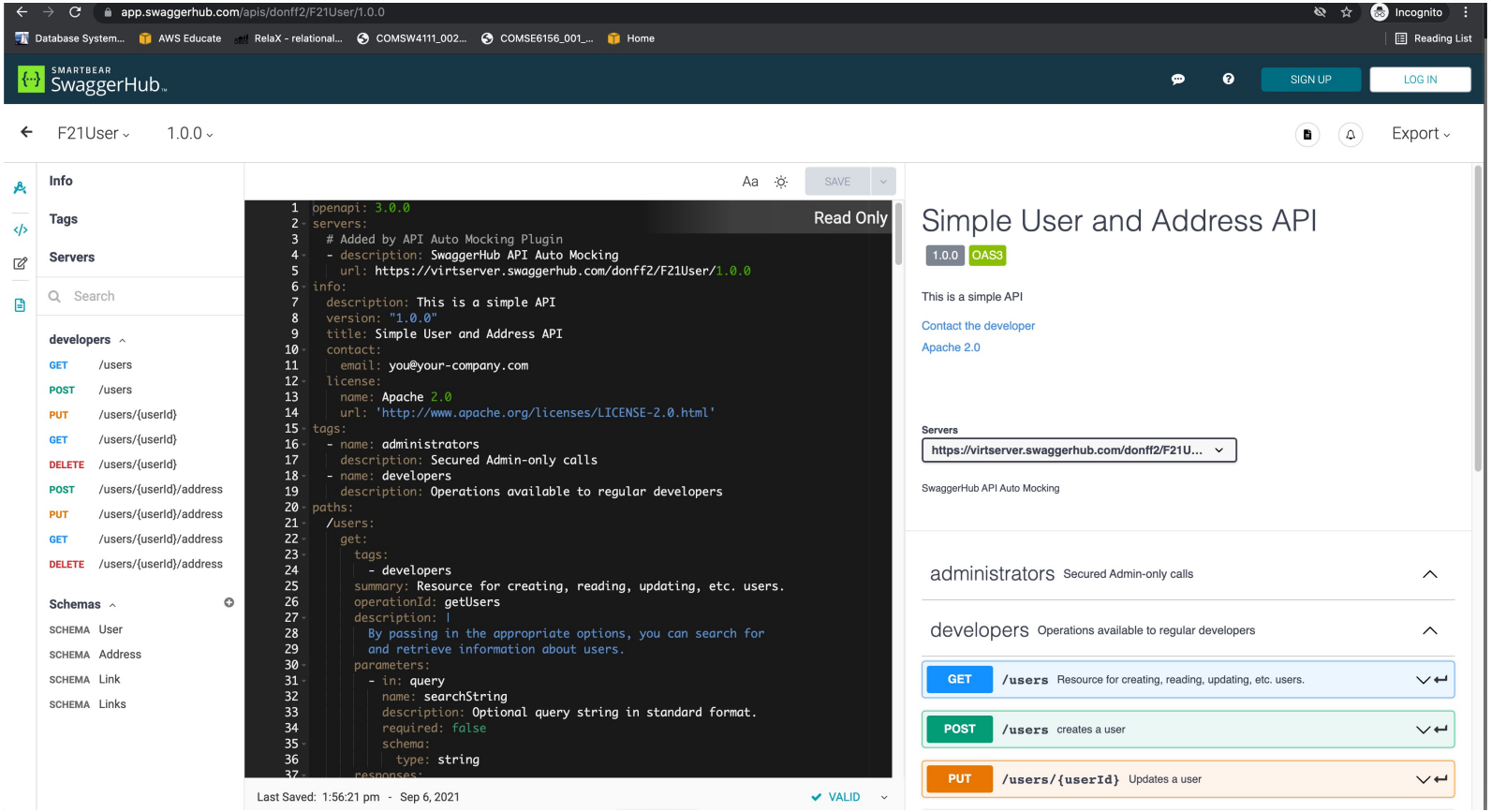
- There are two approaches to defining APIs:
  - Build Open API definition → Code generation tools.
  - Start with source code and add "annotations," e.g. https://flask-rest-api.readthedocs.io/en/stable/openapi.html

- Simple example https://app.swaggerhub.com/apis/donff2/F21User/1.0.0

- Demo f21-demo-swagger

- http://localhost:8080/donff2/F21User/1.0.0/ui/

Example

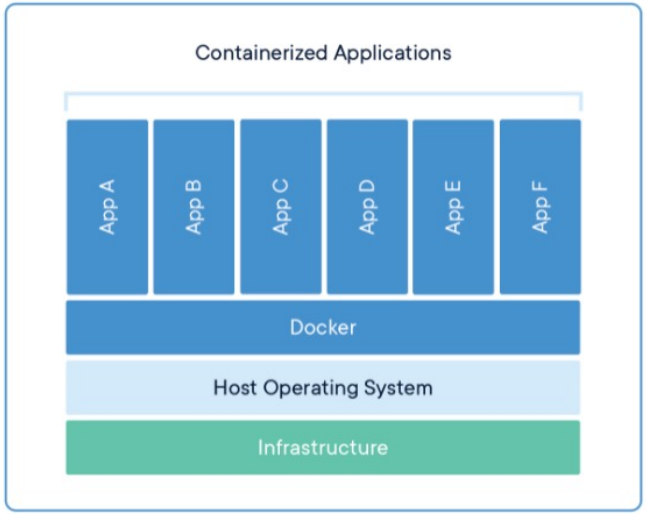


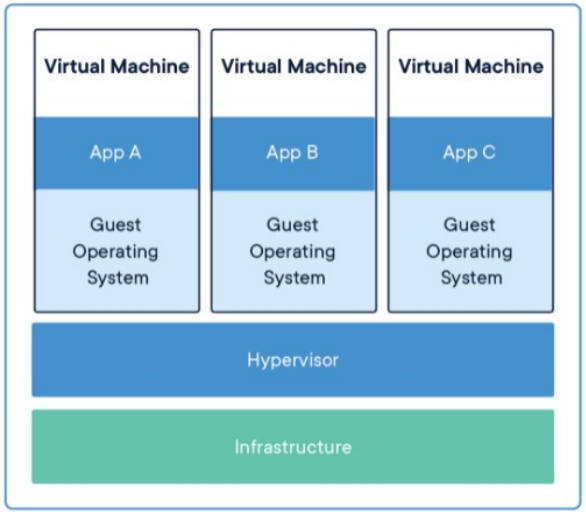
Containers

Fundamental, Recurring Problem

- I have one big computer and I want to run multiple applications.
- This creates several challenges, e.g.
  - Resource sharing and allocation (CPU, memory, disk, ...)
  - Isolation: Application A should not be able to corrupt/mess with application B's files, memory, ...
  - Configuration: Applications require libraries, versions of libraries, prerequisites, compiler/interpreter levels, ...
- Basically, I want a way to someone package or fence all of this.
  We see some of the issues with Java Classpath, Python environments, ...
- There are a few ways to do this:
  - OS process and paths.
  - VMs
  - Containers

Containers and VMs





CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Concept (from Wikipedia)

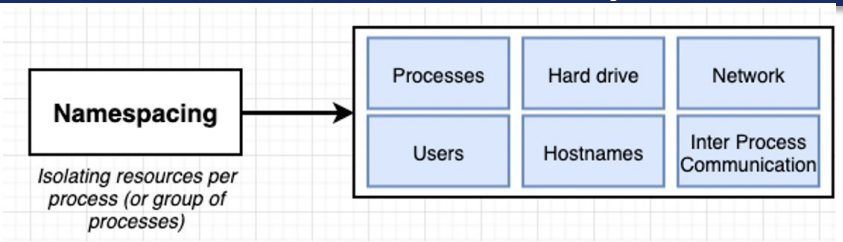
- “cgroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.”

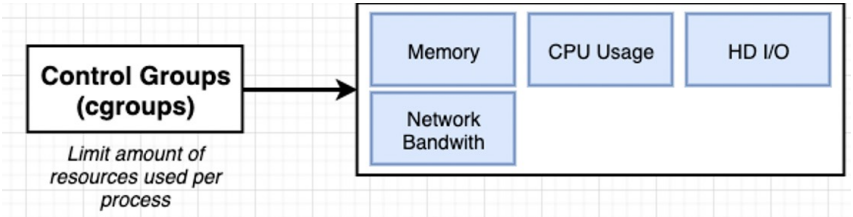
- “Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.
  … …
  those namespaces refer to distinct resources. Resources may exist in multiple spaces. Examples of such resources are process IDs, hostnames, user IDs, file names, and some names associated with network access, ...”

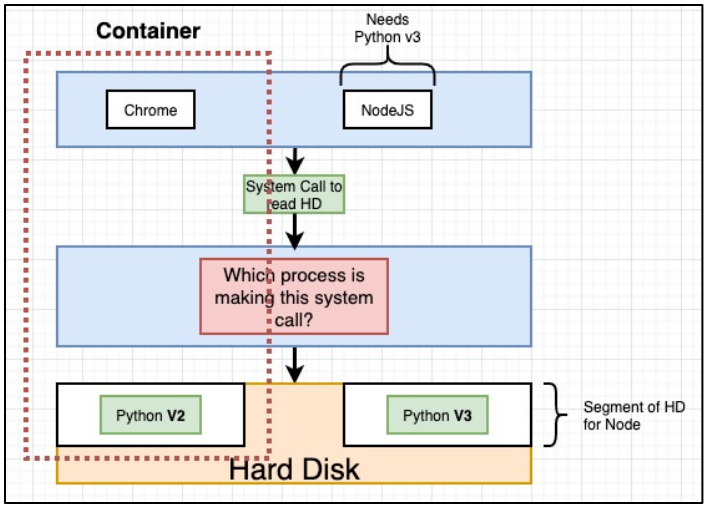
Containers vs VMs

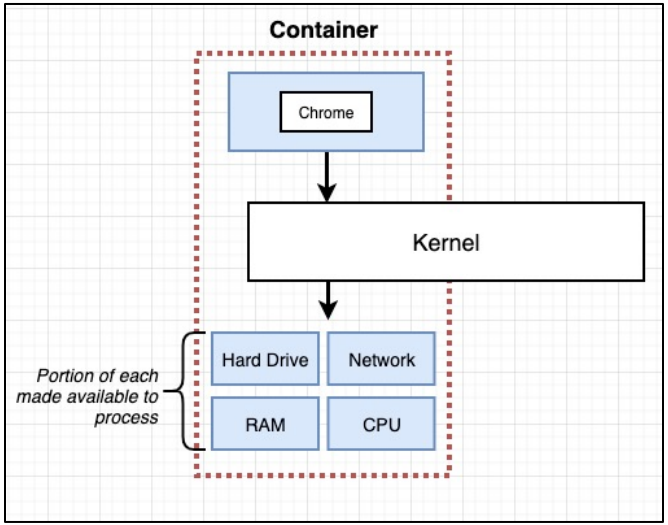
	Process	Container	VM
Definition	A representation of a running program.	Isolated group of processes managed by a shared kernel.	A full OS that shares host hardware via a hypervisor.
Use case	Abstraction to store state about a running process.	Creates isolated environments to run many apps.	Creates isolated environments to run many apps.
Type of OS	Same OS and distro as host,	Same kernel, but different distribution.	Multiple independent operating systems.
OS isolation	Memory space and user privilieges.	Namespaces and cgroups.	Full OS isolation.
Size	Whatever user's application uses.	Images measured in MB + user's application.	Images measured in GB + user's application.
Lifecycle	Created by forking, can be long or short lived, more often short.	Runs directly on kernel with no boot process, often is short lived.	Has a boot process and is typically long lived.

Some Container/Docker Implementation Concepts





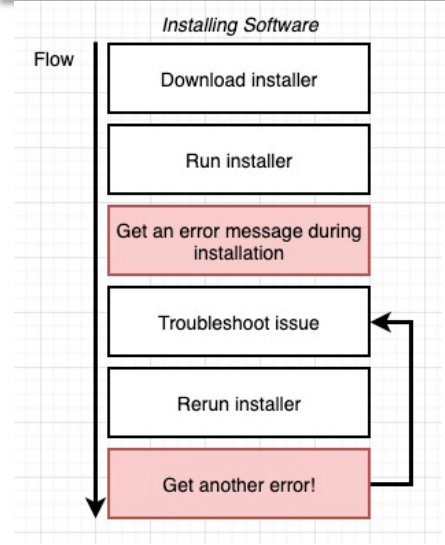




Isolates libraries, paths, packages, PIDs, ...

Partitions and caps resource consumption.

Containers and Some Docker Overview



This is what Docker is trying to fix

Why use Docker? → Docker makes it really easy to install and run software without worrying about setup or dependencies

- We are familiar with the problem
  - We have a system with an OS and installed SW.
  - We need to install something new, and that requires installing prerequisites, changing versions, etc.
  - After many tries, we get the application installed only to find out we broke some other application.
- Docker and containers allow you to:
  - Develop an application or SW system.
  - Package up the entire environment (paths, versions, libs, ...) into an image that works and is self-contained.
  - Someone installing/starting your application simply gets the image from a repository and starts the image to create a running container.

Simple Tutorial

- We will follow this tutorial https://docs.docker.com/language/python/ to get experience.
- But, the in the future, basic idea is the same as we have previously followed.
- The steps:
  - Find an example of something cool.
  - Download it, set it up and run it.
  - Modify it and add my application code and logic.
  - Push it somewhere, in this case Docker Hub
  - Deploy on the cloud. AWS gives you two or three options:
    - Docker on EC2
    - Elastic Container Service
- I also used this: https://www.docker.com/blog/containerized-python-development-part-2/

Overview

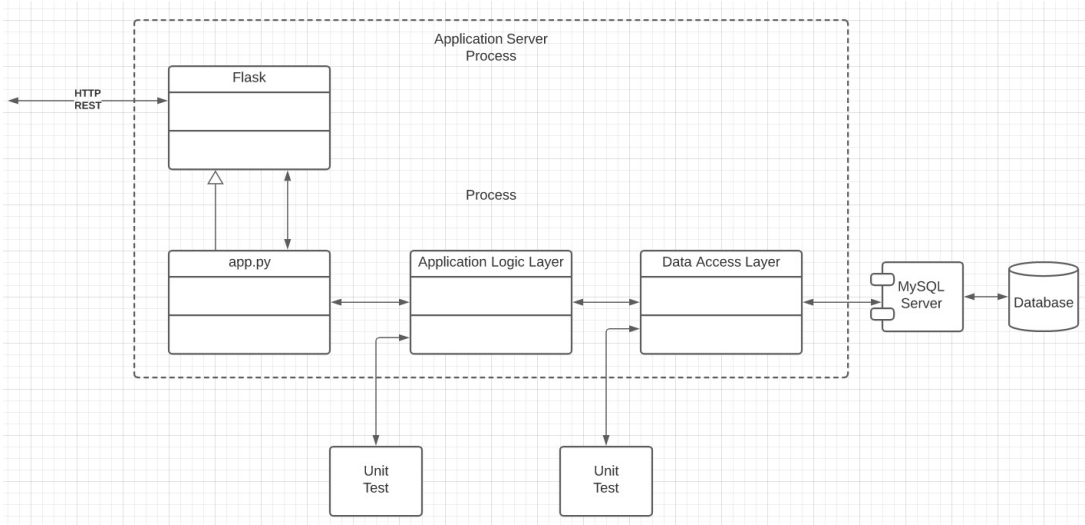
- Cloned the repository.
- Files
- Run container

Project – Sprint 1

(Note: Show Trello, Agile, etc).

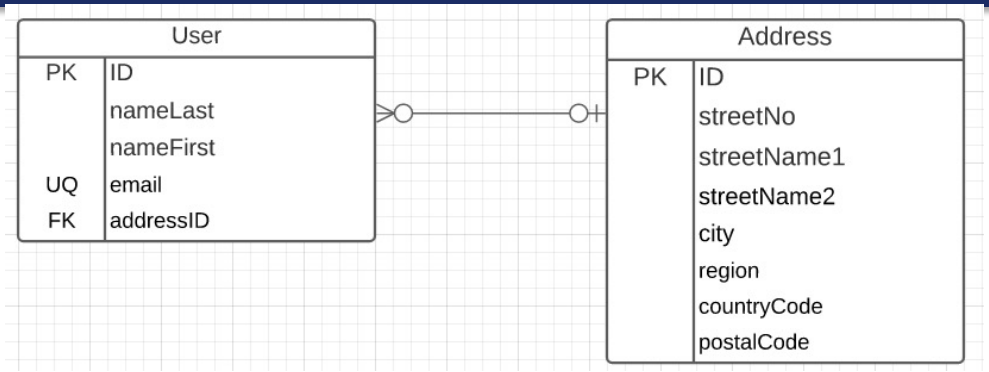
Specification (TBD)

- Build microservices:
  - Users, … …
  - Deploy users on Use two separate RDS instances.
- Implement a layered architecture:
  - Data objects for access DBs.
  - Application logic in app services.
  - Routing in app.py
- Implement paths:
  - /users
  - /users/{id}
  - /users/{id}address
  - /addresses
  - /addresses/{addressId}
  - /addresses/{addressId}/users
  - … …
- Do a very simple UI and deploy on S3.



Implement Linked Data

- A user has 0 or 1 addresses.
- Several people make have the same address.
- IDs must be unique and service generated.
- Country code must come from a valid list of country codes.
- City cannot contain numbers.
- Email must have valid format:
  - Contain “@”
  - End in one of:
    - .edu
    - .com
    - .org
    - .mil
- Only the following may be NULL
  - nameFirst
  - streetName2



HATEOAS Driven REST APIs

HATEOAS (Hypermedia as the Engine of Application State) is a constraint of the REST application architecture that keeps the RESTful style architecture unique from most other network application architectures. The term “***hypermedia***” refers to any content that contains links to other forms of media such as images, movies, and text.

REST architectural style lets us use the hypermedia links in the response contents. It allows the client can dynamically navigate to the appropriate resources by traversing the hypermedia links.

Navigating hypermedia links is conceptually the same as a web user browsing through web pages by clicking the relevant hyperlinks to achieve a final goal.

For example, below given JSON response may be from an API like `HTTP GET http://api.domain.com/management/departments/10`

```
{
  "departmentId": 10,
  "departmentName": "Administration",
  "locationId": 1700,
  "managerId": 200,
  "links": [
    {
      "href": "10/employees",
      "rel": "employees",
      "type" : "GET"
    }
  ]
}
```

https://restfulapi.net/hateoas/