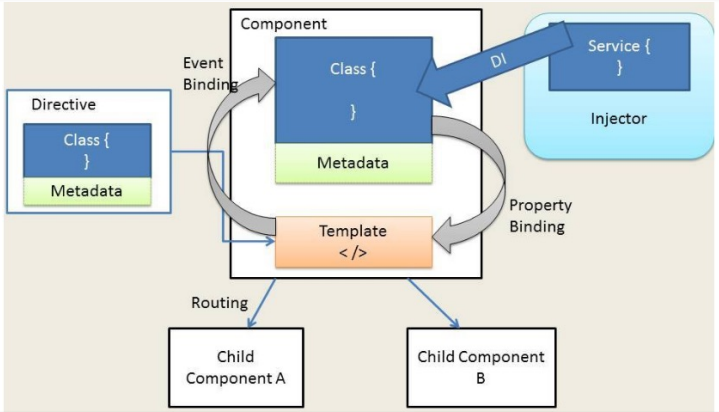
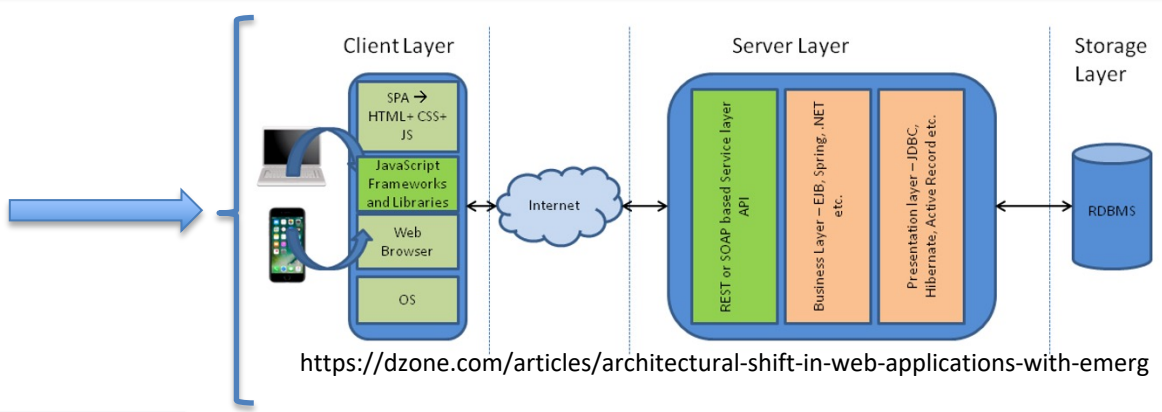
E6156 – Topics in SW Engineering: Cloud Computing

Lecture 2: Microservices, REST, PaaS, S3

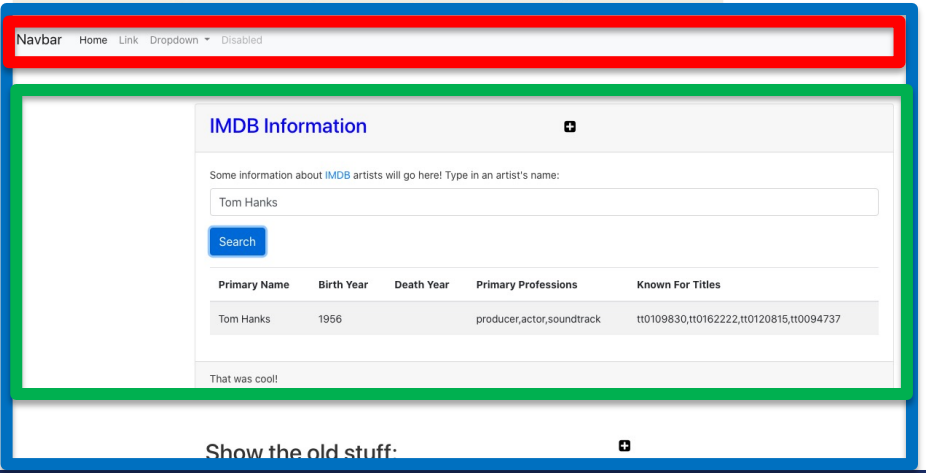
Static Web Hosting

Web Application Architecture





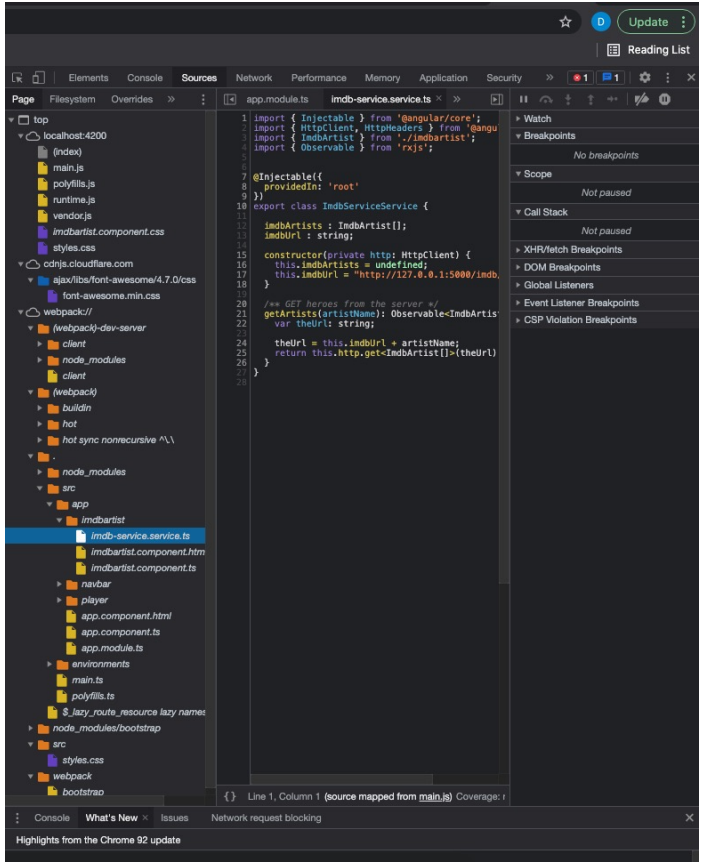
https://dzone.com/articles/architectural-shift-in-web-applications-with-emerg



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

Show the Code

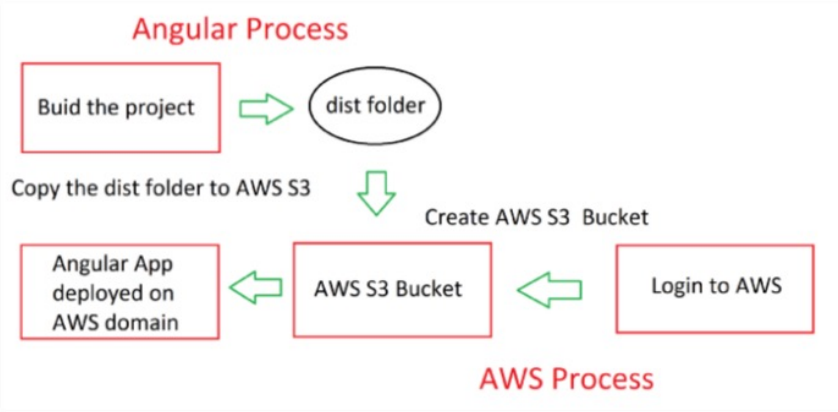
What's in the Browser



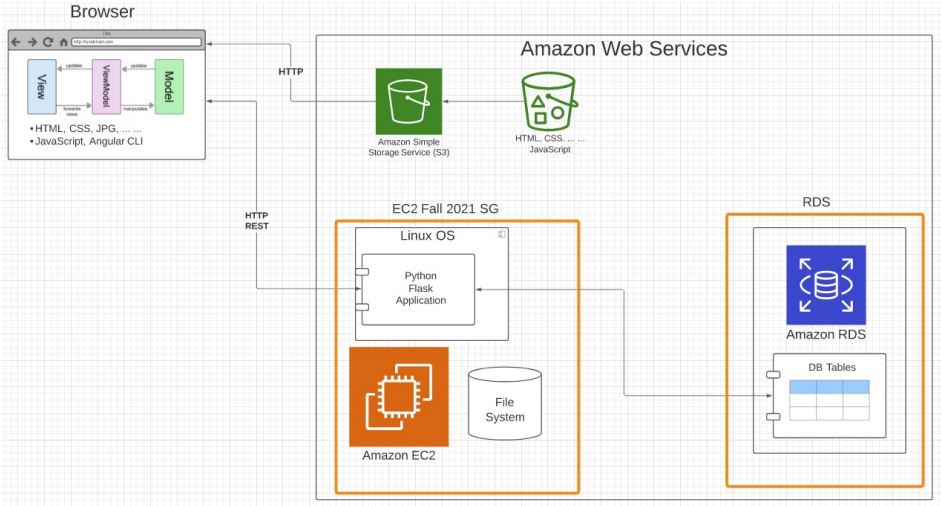
- The browser is an application that interprets the files:
  - HTML, CSS, etc. define the document and content.
  - JavaScript defines dynamic behavior.
- Browser loads index.html
  - index.html contains links to files.
  - The browser loads the linked files.
  - These also have links, which the browser 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.
- 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.

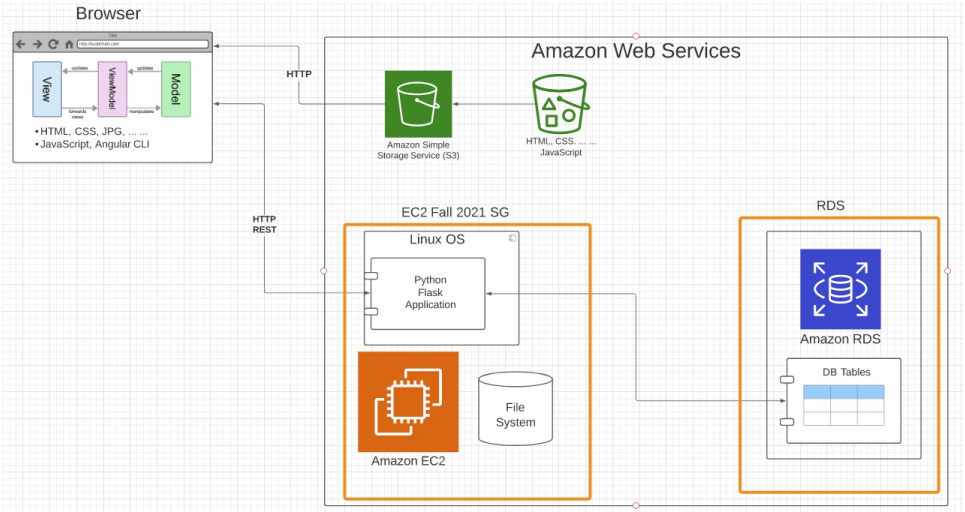
- Create an S3 bucket.
- Upload files from /dist folder.
- Enable static web hosting.
- Set access permissions.
- Set index.html.
- We will learn how to automate later.

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('127.0.0.1')) || (theUrl.includes('localhost')))
  {
    result = 'http://127.0.0.1:5000/imdb/artists/';
  } else {
    result = 'ec2-54-242-71-165.compute-1.amazonaws.com:5000/imdb/artists/';
  }
  return result;
}
```



Introduction to REST

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

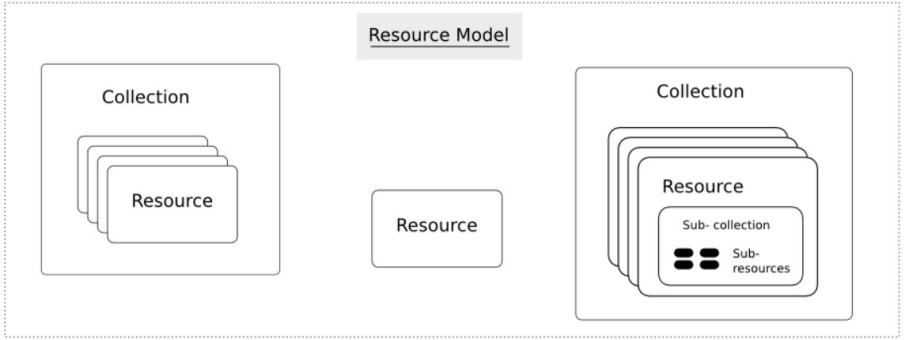
Resources are an abstraction. The application maps to create things and actions.

"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.

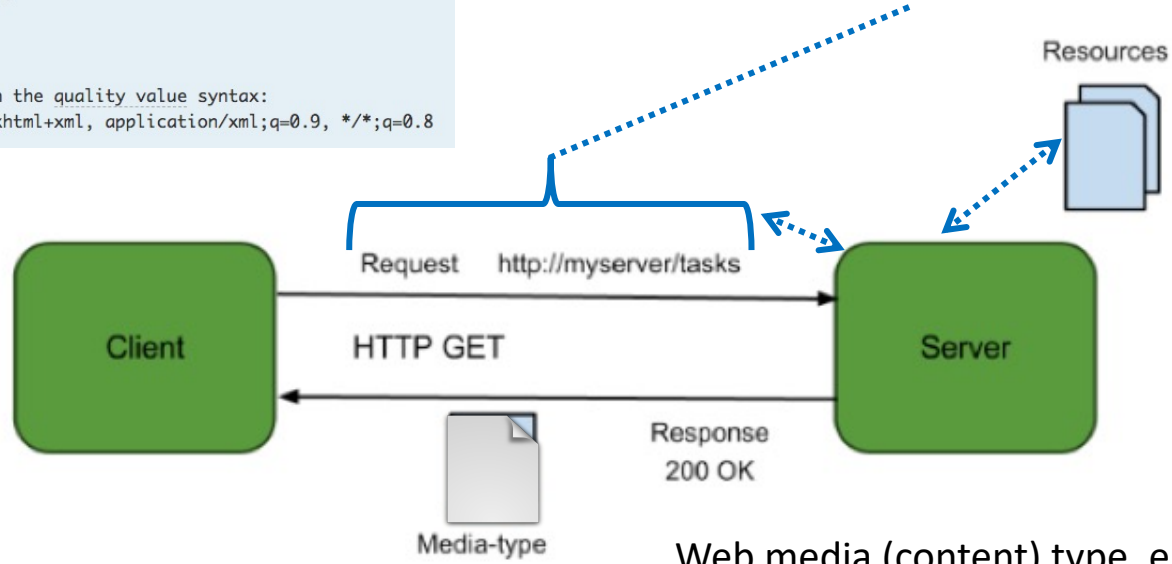
Resources, URLs, Content Types

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.
  Some of the principles are subtle and have major implications.
- We will go into the principles in later lectures, ... but first
  - A little about microservices.
  - Introduction to Platform-as-a-Service.
  - Build out our first resource oriented, REST microservice (s)

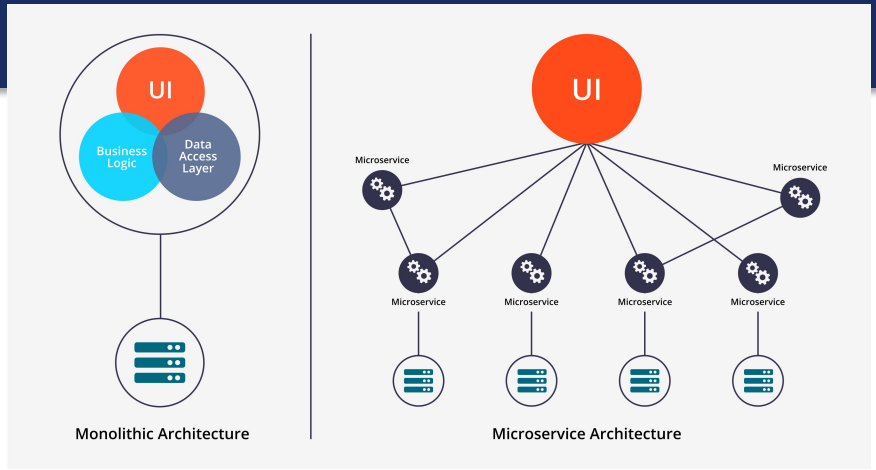
Introduction to Microservices

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

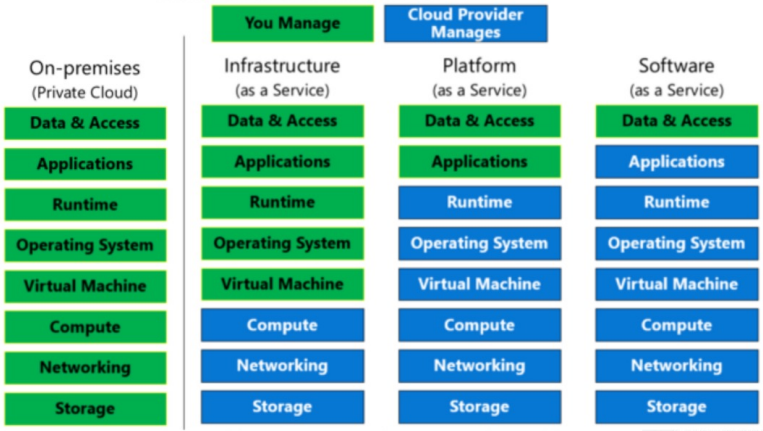
Microservices

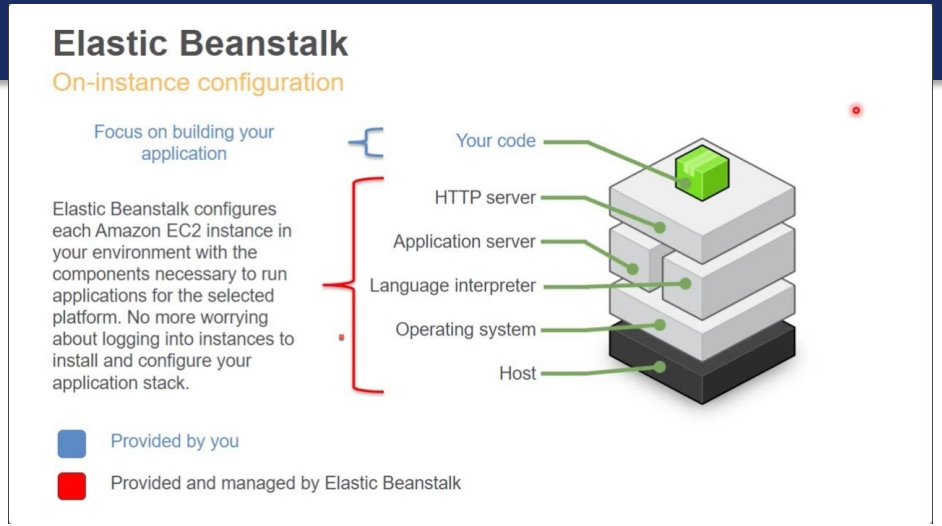
	Monolithic	Microservice
Architecture	Built as a single logical executable (typically the server-side part of a three tier client-server-database architecture)	Built as a suite of small services, each running separately and communicating with lightweight mechanisms
Modularity	Based on language features	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.

Introduction to PaaS
Elastic Beanstalk

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.

Start with Elastic Beanstalk

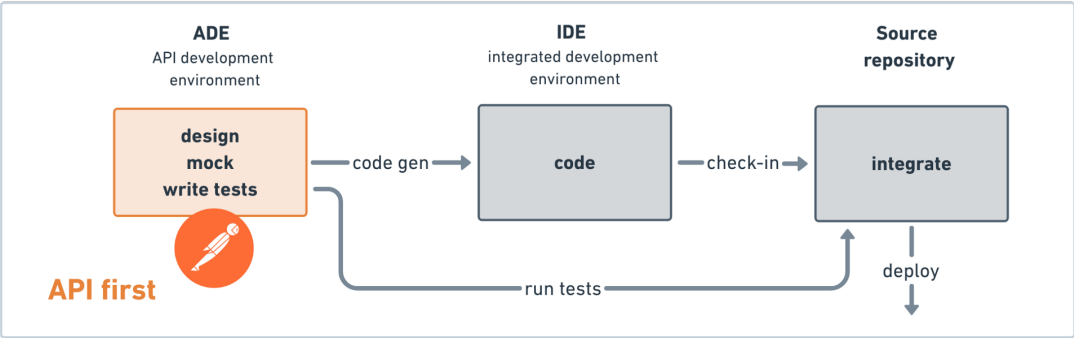
- There are several reasonably good tutorials on using EB and Flask
  - https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-flask.html
  - https://medium.com/analytics-vidhya/deploying-a-flask-app-to-aws-elastic-beanstalk-f320033fda3c
- AWS and technology evolve. So, sometimes 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.

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:

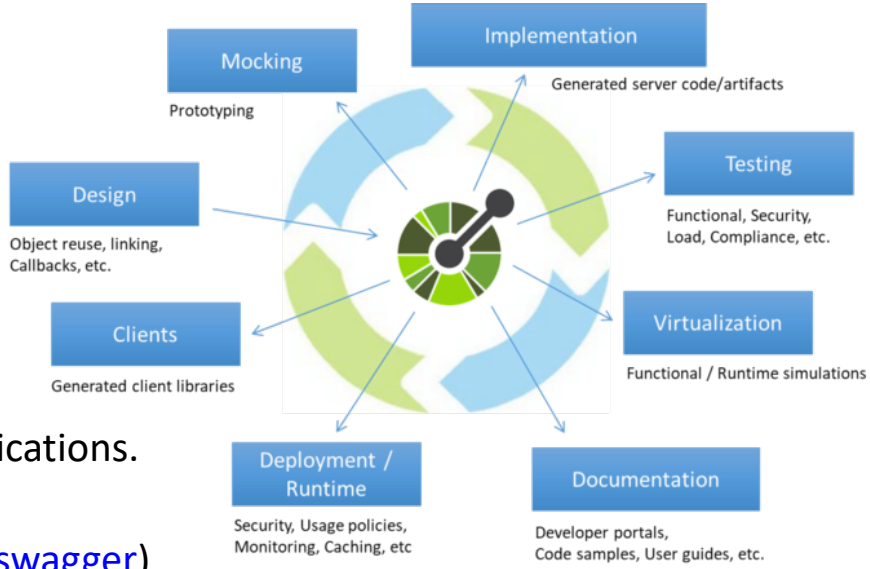
1. Your API is the first user interface of your application

2. Your API comes first, then the implementation

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

Open API

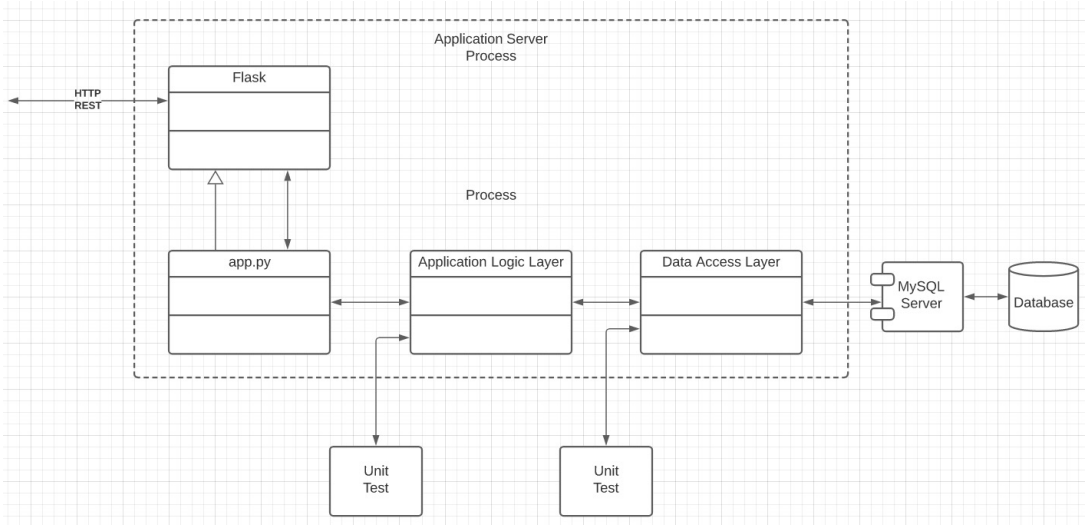
- "The OpenAPI Initiative (OAI) was created by a consortium of forward-looking industry experts who recognize the immense value of standardizing on how APIs are described. As an open governance structure under the Linux Foundation, the OAI is focused on creating, evolving and promoting a vendor neutral 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.



Introduction to REST

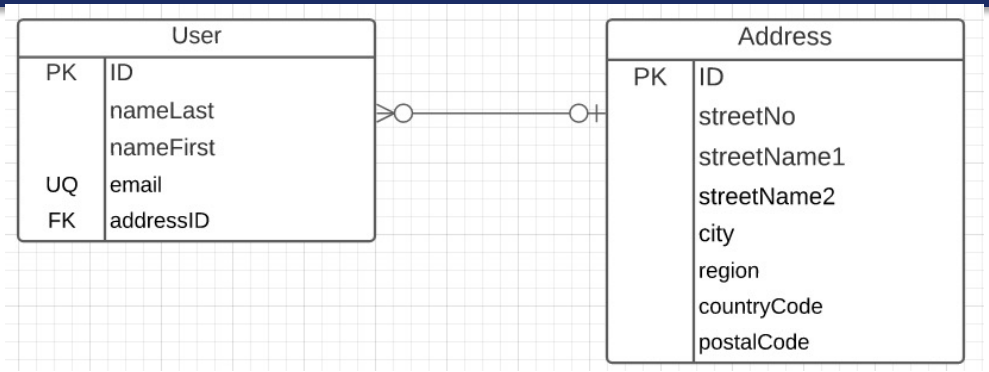
Specification

- Build two microservices:
  - users, addresses
  - Deploy users on AWS Elastic Beanstalk; deploy addresses on EC2.
  - 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/