W4153 – Cloud Computing

Lecture 3:

Rest(2), Microservices(2), CaaS, DB

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Contents

Contents

- Course updates
- A Diversion on Modeling
- REST Continued:
  - HATEOAS
  - Query parameters
  - Response codes
- Container-as-a-Service:
  - Concepts
  - Local execution of multiple-containers (cloud execution comes later)
  - First cloud deployment
- Web Content and Browser Applications
- Database-as-a-Service
  - Concepts
  - Example
- What's Next
- Microservices Continued
  - 12 Factor App: Dependencies, Config
  - SOLID: Dependency Inversion (Injection)
  - Composition
- Summary and Next Steps

Course Updates

Course Updates

- Teams
  - Well, based on the fun we had, you can see why I want to build a course management app.
  - Rattandeep and I formed and published teams for people who did not have teams. Please meet, choose a focal point and get started.
  - Rattandeep and I fixed the errors and issues with the submitted team forms. Please verify and validate.
- Credits
  - I have $100 credits for Google Cloud. I will republish my cloud app that allocates the credits.
  - AWS:
    - You should receive email invitations to AWS Learner Lab. Please accept and join.
    - We can use AWS Lerner Lab for use cases not covered by free tier.
    - Read/Skim the AWS Academy Learner Lab Student Guide under modules.
- Budget alerts:
  - MANAGE your AWS usage and stay in free tier. Set alerts. https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/tracking-free-tier-usage.html
  - Set up budget alerts on Google: https://cloud.google.com/billing/docs/how-to/budgets
  - **Shut down VMs, etc. when not using them.**

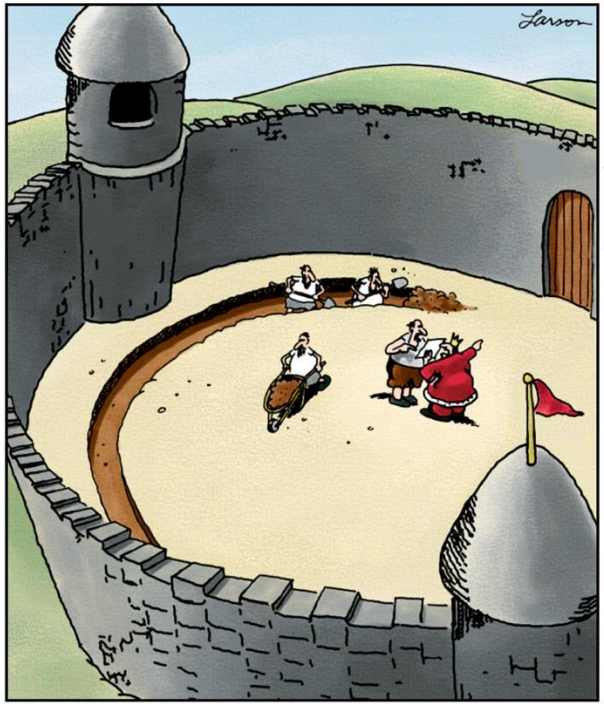
A Diversion on Modeling

Concepts

The Value of Just Enough Modeling

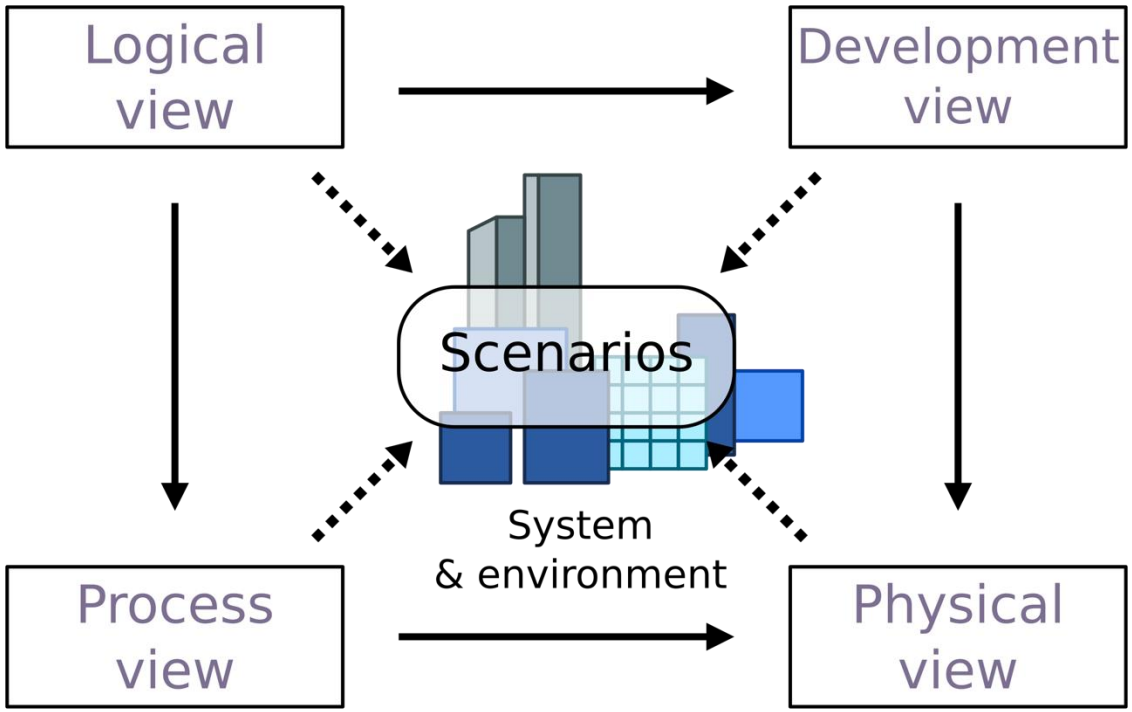
- Weinberg's Second Law: “If builders built buildings the way programmers wrote programs, **then the first woodpecker that came along would destroy civilization.”**
- Give your project, code, … a little upfront thought. Focus on simple steps, test and then next step.
- Most students and teams build their systems and write their code like a slam poet on a triple espresso.
- Just enough modeling and thought, with simple steps and frequent testing seems slow, but … …

“Slow is smooth. Smooth is fast.”



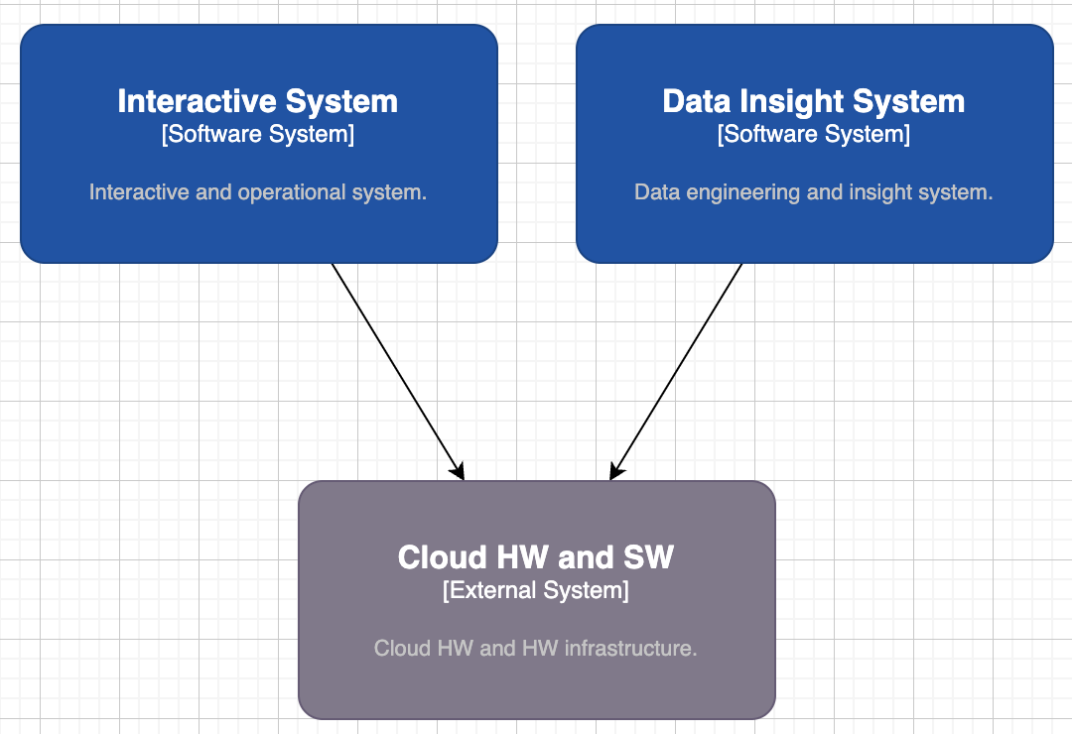
Suddenly, a heated exchange took place between the king and the moat contractor.

4+1 Architecture Model



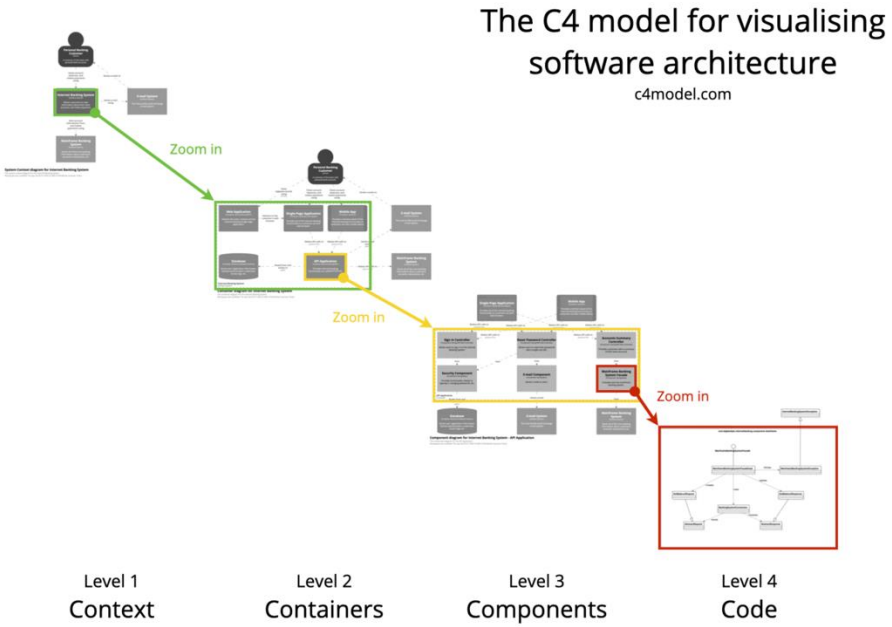
- I try to apply “just enough” modeling in 5 areas.
- The 4+1 Architecture Model is a simple, conceptual approach.
- I have a preferred ”notation” for each “view.”
- I am not pedantically rigorous about the modeling details.
- I use some modifications and notes to explain.

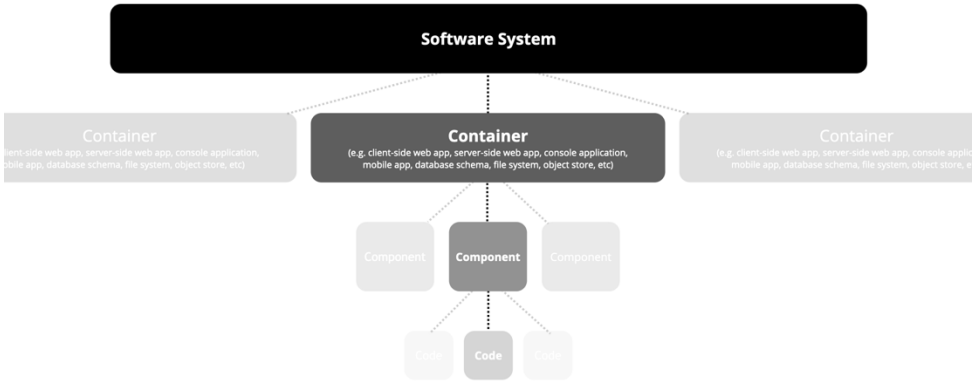
Class Project Solution – Logical View



- I use C4 and UML for the logical model.
- Many complex solutions have two major elements:
  - Operational/Interactive
  - Analytical/Data Analysis
- In this course, we place more emphasis on interactive cloud applications because there are topics courses on cloud and data.

C4 Model (https://c4model.com/)



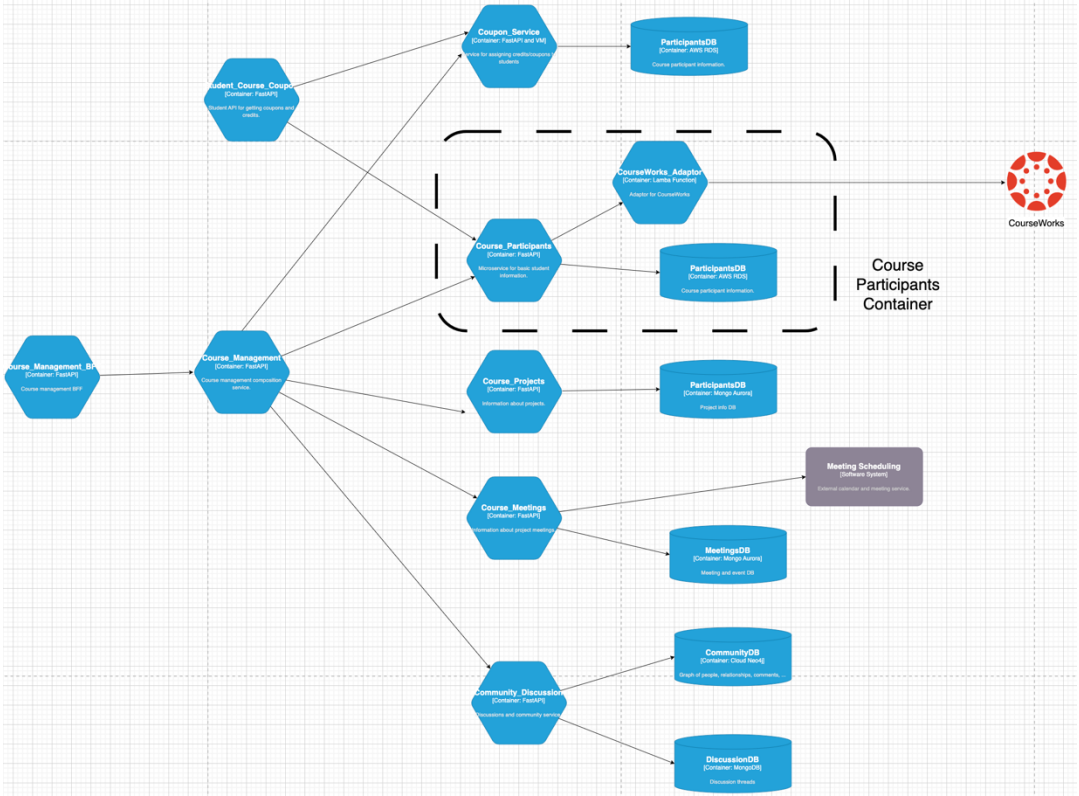


A **software system** is made up of one or more **containers** (applications and data stores), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (classes, interfaces, objects, functions, etc).

- Incremental, progressive design and refinement.
- Again, “just enough” to think it through and explain.

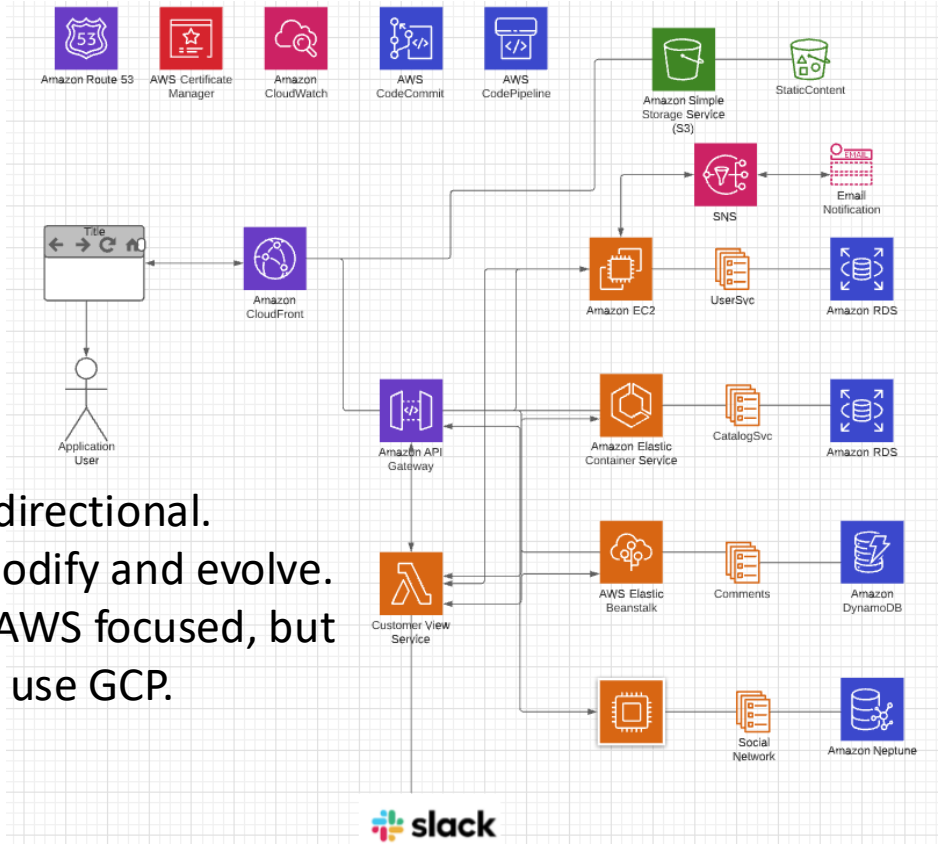
(Partial) Logical View – Interactive System

- 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.”*



Physical View

- Note: Cannot show all of the technology and connections with creating spaghetti.
- Databases/Storage:
  - RDS
  - DynamoDB
  - MongoDB (or Document DB)
  - Neptune (or Neo4j)
  - S3
- Networking/Communication:
  - Route 53 (DNS)
  - Certificate Management
  - CloudFront
  - VPC
  - API Gateway
- Compute:
  - EC2
  - Elastic Beanstalk
  - Elastic Container Service (or Docker)
  - Lambda Functions
- Integration/collaboration:
  - SNS, SQS
  - Email
  - Slack
- Continuous Integration/Continuous Deployment:
  - Code Commit, Code Pipeline
  - Or GitHub and actions



- This is directional.
- I will modify and evolve.
- This is AWS focused, but we will use GCP.

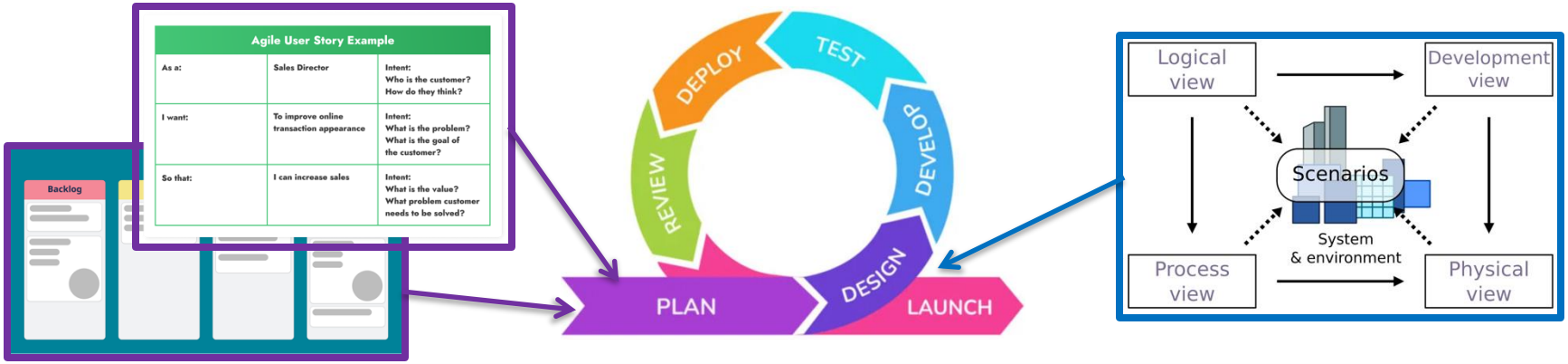
Summary
And
Application in this Course

Summary and Application in this Course

- Summary (and my preferences):
  - Views:
    - Logical view: C4 and UML.
    - Physical view: Standard visual elements for AWS, GCP, … …
    - Process view: (to be covered) UML Sequence Diagrams, BPMN.
    - Development view: I normally just do something simple.
    - REST Resource view: (something I do) UML Class Diagrams or ER Diagrams.
    - Scenarios: Agile development user stories.
  - Summary: Use common sense and focus on "just enough."
- Application to this course:
  - Your interim status reports will explain your architecture and high-level design. The explanation with be text/prose and architecture view diagrams.
  - Your final project will be a demo and presentation. The presentation should use the views.
  - The TAs and I will provide templates for reports and presentations.

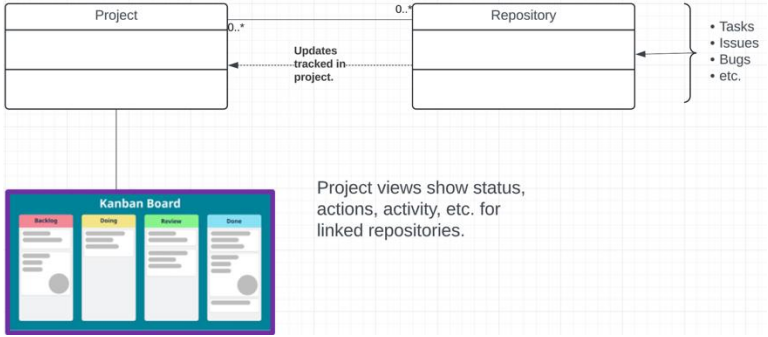
Architecture and Development Processes

- Architecture as Code is an emerging concept:
  - "Architecture as Code" (AasC) aims to devise and manage software architecture via a readable and version-controlled codebase, fostering a robust understanding, efficient development, and seamless maintenance of complex software architectures." (https://devops.finos.org/docs/working-groups/aasc/)
  - We see more tools enabling you to create software architecture and other diagrams as code. The main benefit of this concept is that most diagram-as-code tools can be scripted and integrated into a built pipeline to generate automatic documentation. (https://medium.com/@techworldwithmilan/software-architecture-as-code-tools-331a11222da0)
  - Some interesting perspective in The Architect Elevator (https://architectelevator.com/cloud/iac-architecture-as-code/)
- Agile Development: We will use Kanban boards and user stories for planning, and 4+1 Architecture View for Design.



Agile Planning – Simplistic Introduction

- There is a lot of documentation, examples, tutorials, … on Agile Development.
  - The Ansys teams use:
    - Atlassian: https://www.atlassian.com/agile/tutorials
    - GitHub: https://github.com/resources/articles/devops/what-is-agile-methodology
    - Azure DevOps: https://azure.microsoft.com/en-us/products/devops
  - We will use a simple approach in this course using GitHub.
    - HW0 – my example: https://github.com/users/donald-f-ferguson/projects/8
    - Course management application: https://github.com/users/donald-f-ferguson/projects/9
- The basic, overly simple approach:
  - Your project is a set of repositories linked to a "master" project.
  - Your status reports will include Kanban board and views from project.
  - *We will keep it simple just so you can say on your CVs that you did basic Agile Development.*



REST Continued

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.

Resource Oriented

- There are several good, simple tutorials:
  - https://www.restapitutorial.com/introduction
  - https://restfulapi.net/
  - The documentation agrees on a common core, but details and application are subjective.
- Identification of resources is URLs:
  - Resource collections: http://api.contoso.com/api/students
  - Resource: http://api.contoso.com/api/students/dff9
  - Links: http://api.contoso.com/api/students/dff9/courses/COMSW4153_002_2024_001/projects
- Manipulation of resources through representations: applications, Python/Java objects, relational databases, air conditioner management APIs, … … are resources.
- Self-descriptive message: HTTP verbs, MIME content types, … …
- Hypermedia as the engine of application state – HATEOS
  Just because it sounds so cool and is awesome to day.

A Note HTTP Status Codes

- A student asked about “422 – Validation Error.”
  - Look at a simple example: https://github.com/donald-f-ferguson/W4153-Hello-World-FastAPI
  - @app.get("/") only has 200 in the status responses.
  - @app.get("/hello/{name}") has 200 and 422.
  - 422 normally means “bad input.”
- There is not hard and fast, standard definition for status codes.
  - There are several “recommended” best practices and style guides.
  - The most import point is to pick a style/pattern and be consistent over all your APIs.
- There are no hard and fast, standards for a REST API, in general.
  - The most important point is to be consistent on your style and design choices.
  - A caller does not want to write different client code from one resource/service to another.
- I will make some recommendations in my examples. You will need reasonable status codes in your projects.
  - 401 and 403 have generally agreed interpretations. (We will cover).
  - POST should return 201 with a link header to created resource.
  - 304 and 412 apply to eTag processing, which we will cover.
- There are also some interesting status codes:
  - 418 – I’m a teapot.
  - 420 – Enhance Your Calm

HATEOAS

HATEOAS (Hypermedia as the Engine of Application State) is a constraint of the REST application architecture. HATEOAS keeps the REST 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 API response contents. It allows the client to dynamically navigate to the appropriate resources by traversing the hypermedia links.

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

https://restfulapi.net/hateoas/

https://grapeup.com/blog/how-to-build-hypermedia-api-with-spring-hateoas/

In a nutshell:

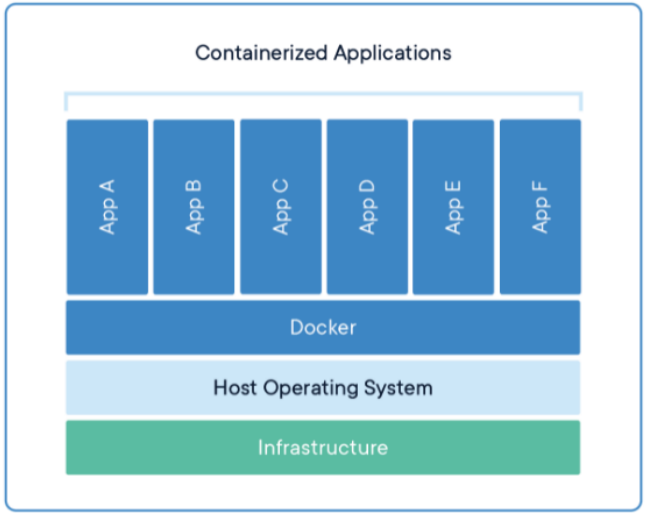
- REST application is basically a graph databases.
- And this explains the emergence of GraphQL.

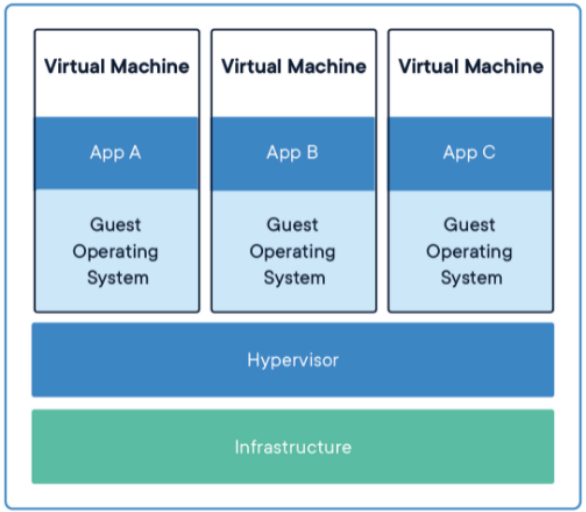
CaaS

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)

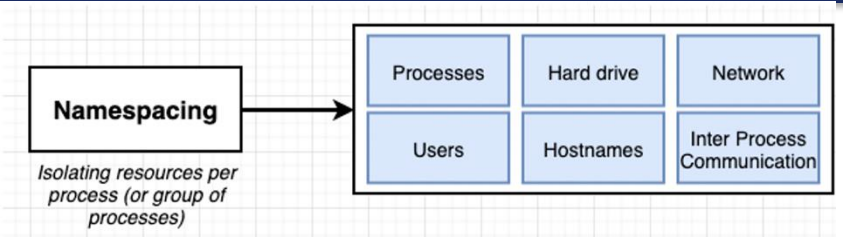
- “cgroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.”

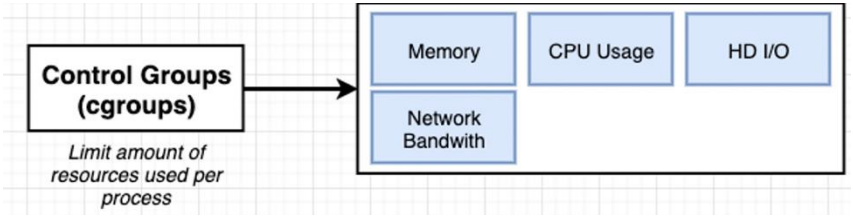
- “Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.
  ... ...
  those namespaces refer to distinct resources. Resources may exist in multiple spaces.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

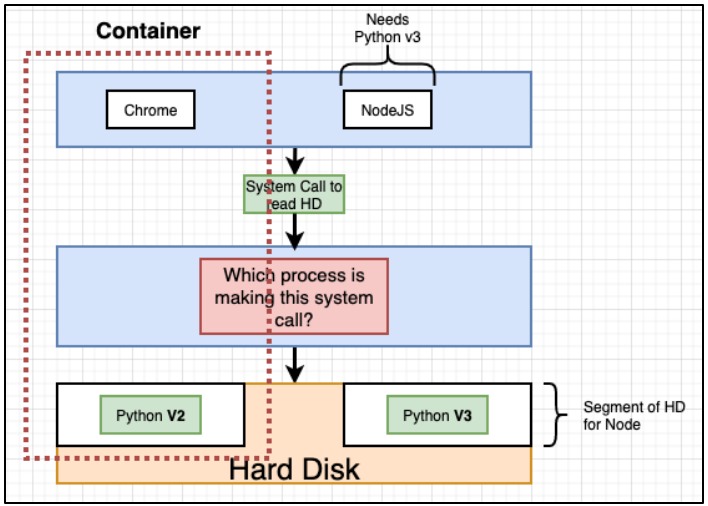
Containers vs VMs

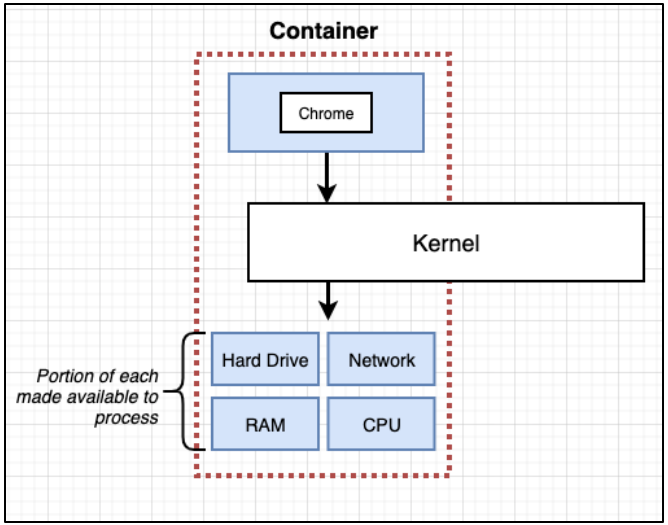
	Process	Container	VM
Definition	A representation of a running program.	Isolated group of processes managed by a shared kernel.	A full OS that shares host hardware via a hypervisor.
Use case	Abstraction to store state about a running process.	Creates isolated environments to run many apps.	Creates isolated 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





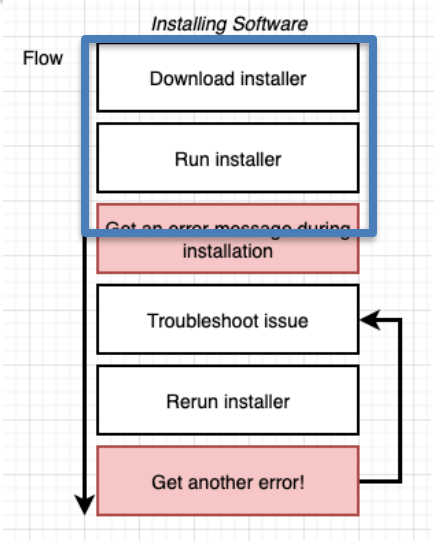




Isolates libraries, paths, packages, PIDs, ...

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.

Demo 1 – Simple Flask Application

Simple Tutorial

- We will follow two tutorials
  - https://docs.docker.com/language/python/
  - https://www.docker.com/blog/containerized-python-development-part-2/

  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. (I had to use GitHub)
  - Deploy on the cloud. AWS gives you two or three options:
    - Docker on EC2
    - Elastic Container Service

  We will start with EC2.

Simple Flask Example

- Walk through project:
  - /Users/donaldferguson/Dropbox/000-NewProjects/E6156F23/python-docker
  - https://github.com/donald-f-ferguson/python-docker.git
- Install Docker on AWS
  - https://medium.com/@mehmetodabashi/how-to-install-docker-on-ec2-and-create-a-container-75ca88e342d2

Deploying to EC2

- The simplest thing to do is to:
  - Tag and push your instance to Docker Hub.
  - Create an EC2, then install and start Docker
  - Pull the image, run it, remembering to expose the ports.
  - Test it locally with curl
  - Update security group to expose port.
- In my case:
  - I have an ARM Mac.
  - The Amazon instances is x86
  - So, I clone the project and rebuild locally.
- There is a way to specify the architecture in an environment variable when building. This allows build on ARM for x86 and vice-versa.

DBaaS

Cloud Concepts – One Perspective

Categorizing and Comparing the Cloud Landscape

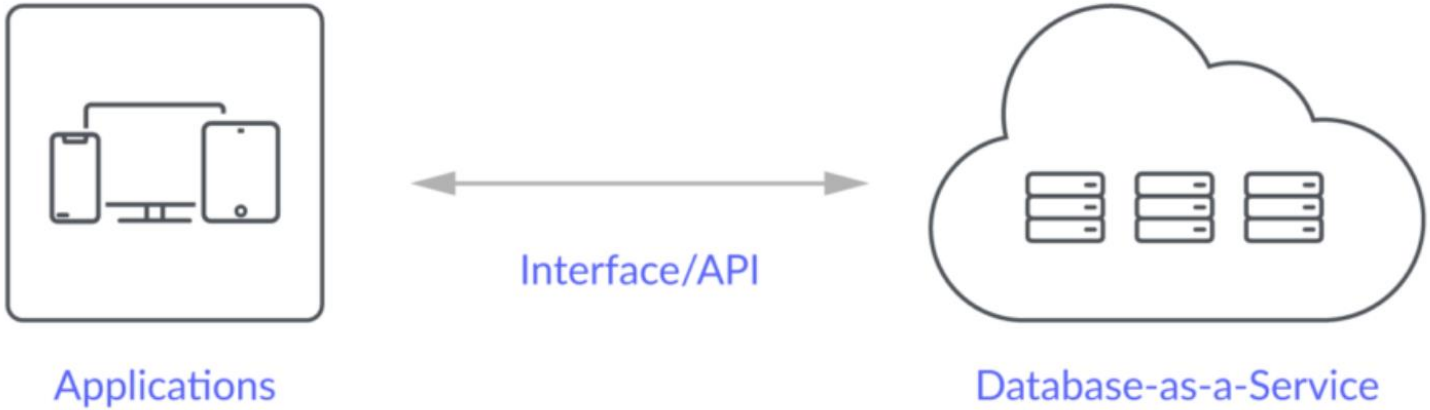
http://www.theenterprisearchitect.eu/blog/2013/10/12/the-cloud-landscape-described-categorized-and-compared/

6	SaaS	Applications			*End-users*
5	App Services	App Services	Communication and Social Services	Data-as-a-Service	*Citizen Developers*
4	Model-Driven PaaS	Model-Driven aPaaS, bpmPaaS	Model-Driven iPaaS	Data Analytics, baPaaS	*Rapid Developers*
3	PaaS	aPaaS	iPaaS	dbPaaS	*Developers / Coders*
2	Foundational PaaS	Application Containers	Routing, Messaging, Orchestration	Object Storage	*DevOps*
1	Software-Defined Datacenter	Virtual Machines	Software-Defined Networking (SDN), NFV	Software-Defined Storage (SDS), Block Storage	*Infrastructure Engineers*
0	Hardware	Servers	Switches, Routers	Storage	
		Compute	Communicate	Store	

Database-as-a-Service

"A cloud database is a database that typically runs on a cloud computing platform and access to the database is provided as-a-service. There are two common deployment models: users can run databases on the cloud independently, using a virtual machine image, or they can purchase access to a database service, maintained by a cloud database provider. Of the databases available on the cloud, some are SQL-based and some use a NoSQL data model.

Database services take care of scalability and high availability of the database. Database services make the underlying software-stack transparent to the user." (Wikipedia)



Let's take a look at 3: Relational Data Service, ~~Dynamo DB, MongoDB (Compass)~~

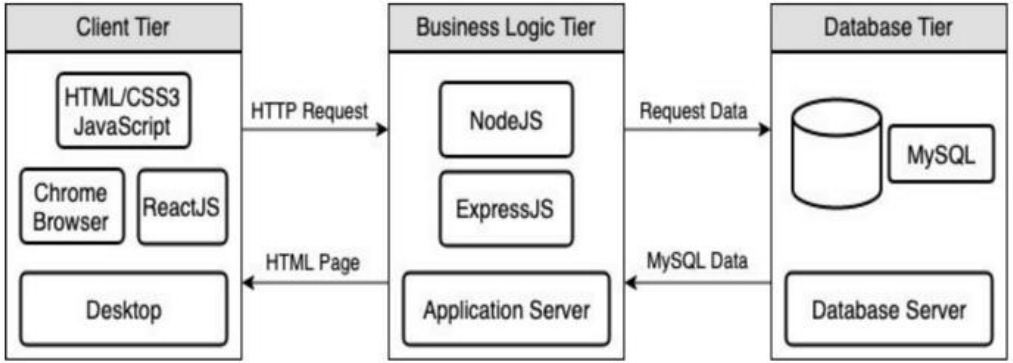
Walkthrough

- RDS
- Setup and configuration
- DataGrip
- Use from a program
- Google equivalent.

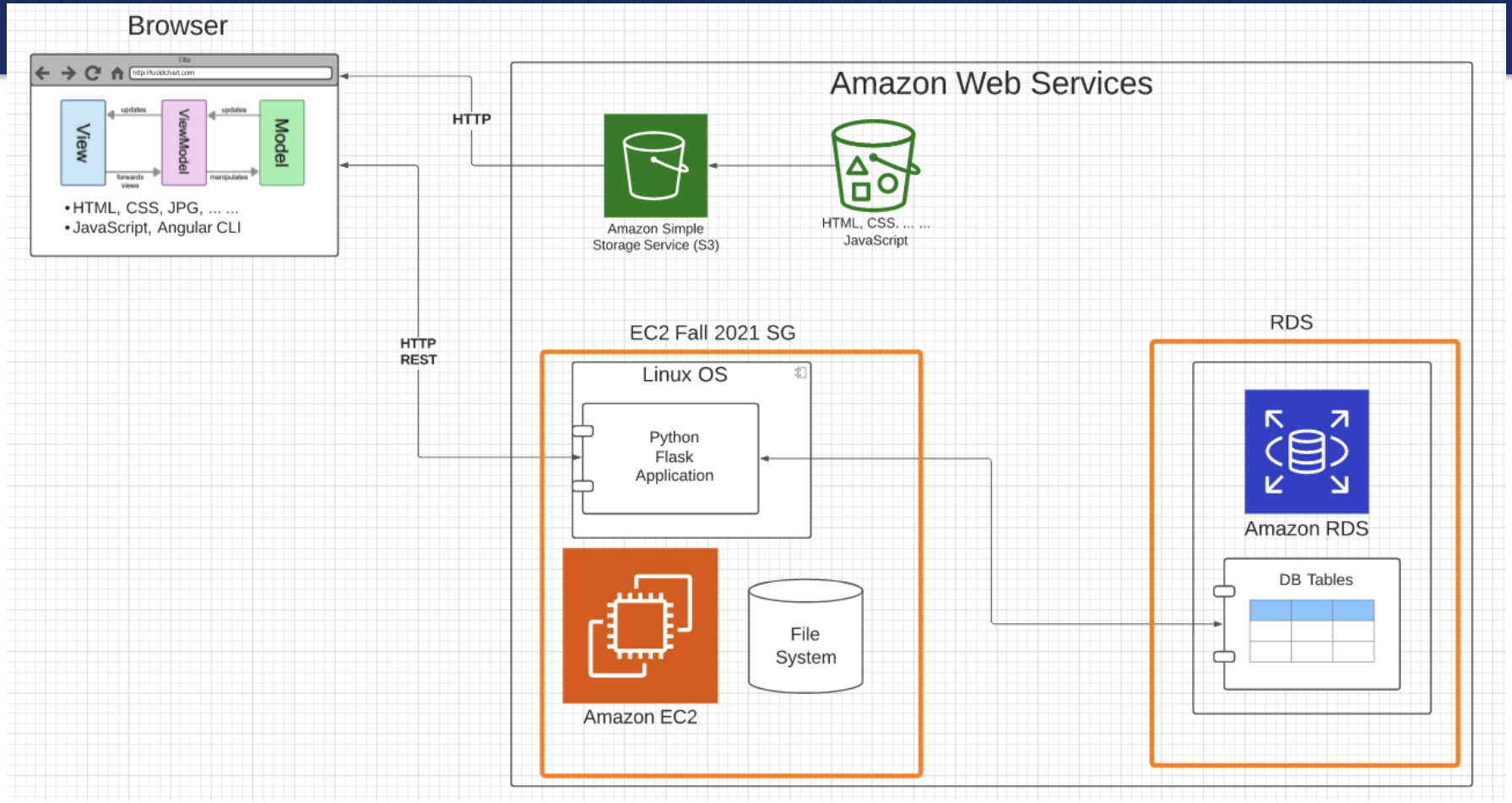
Web Content

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."



BLOB: Binary Large Object

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

Simple Example

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

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

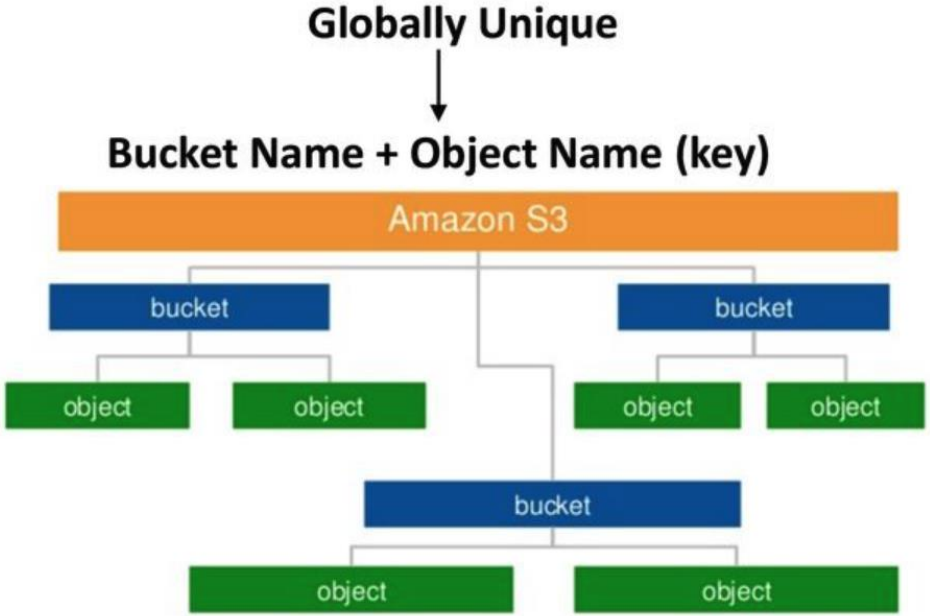
URLs have primarily replaced BLOBs.

Amazon S3

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

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.

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

Simple Example

- https://e6156f20site.s3.us-east-2.amazonaws.com/index.html

What's Next

What's Next

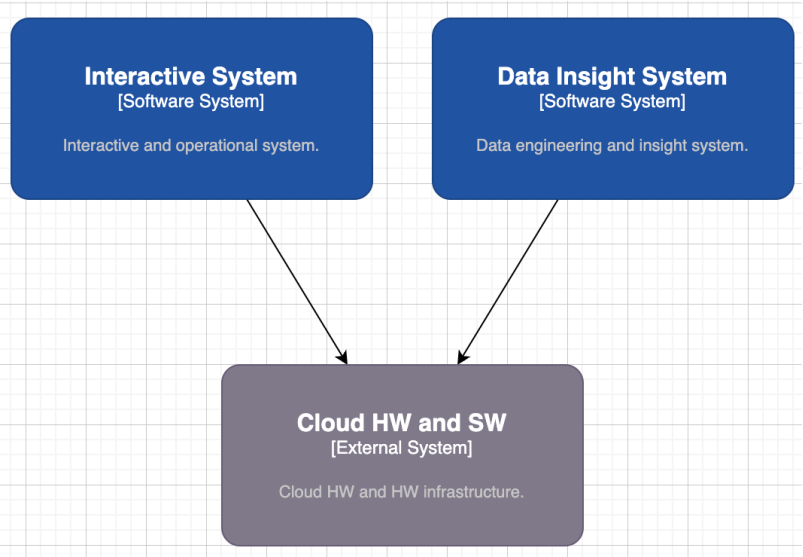
- Teams:
  - The TAs and I will be finalizing assignments for teams we create.
  - Teams should meet and begin defining your project.
- Your team should
  - Define a short writeup/presentation on your scenario. We will arrange discussions.
  - Identify
    - At least one database to get started with DBaaS.
    - Three microservices: two atomic services and one composite service.
    - One web UI project, which can be very simple.
  - Create GitHub repos and a unifying project. Start a Kanban planning board.
  - Setup Docker.
- I am going to hand out credits for Google Cloud and explain AWS Academy.
- Deploy at least one of your microservices in a container on AWS/Google.

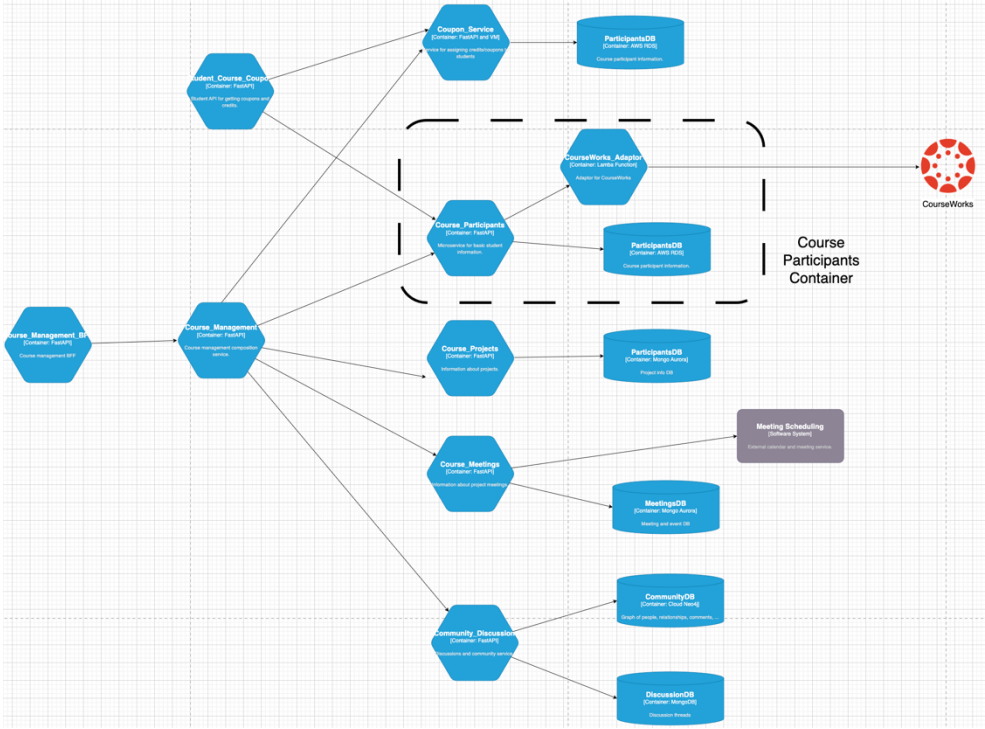
We will provide a more concrete set of steps. This is a lot to absorb.

Microservices Continued

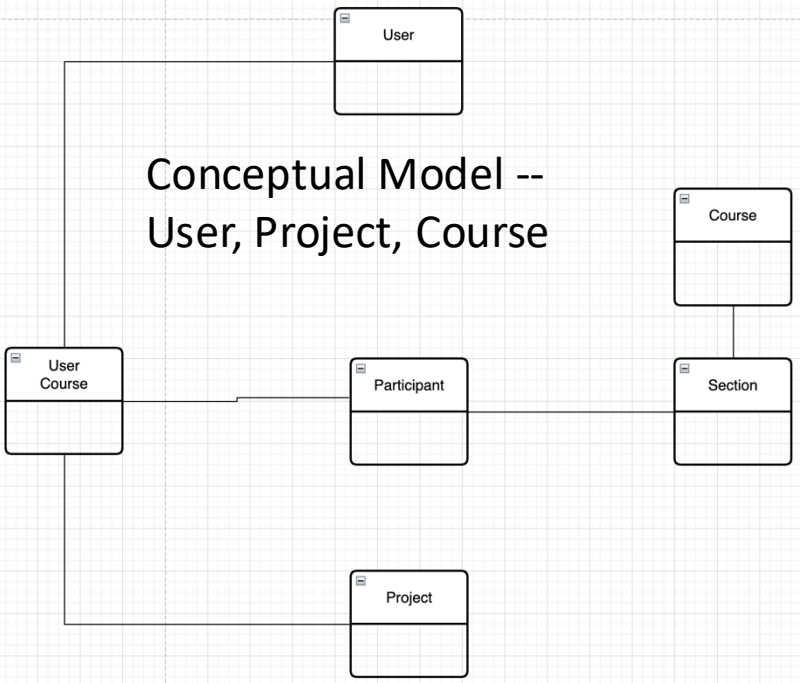
Composites

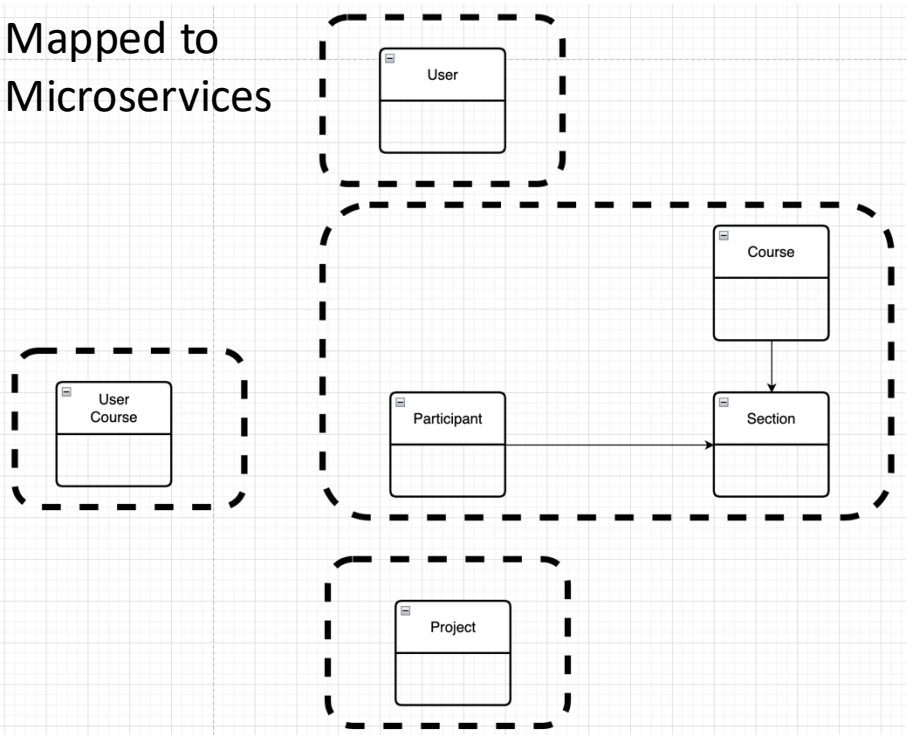
(Partial) Logical View – Interactive System





Resource Oriented and Microservices – My Project





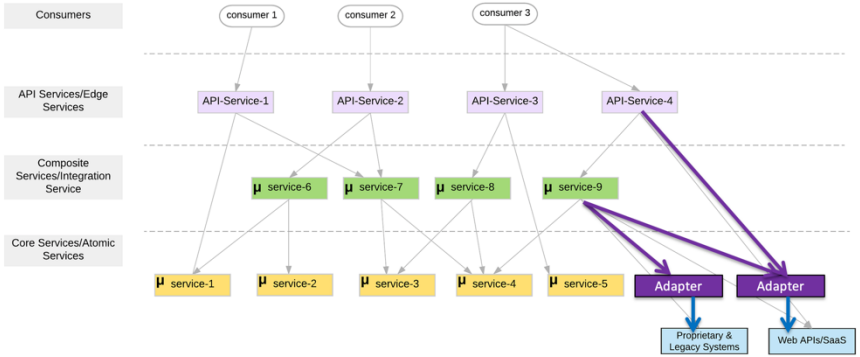
This creates some interesting challenges, e.g.

- Joins
- Foreign Keys

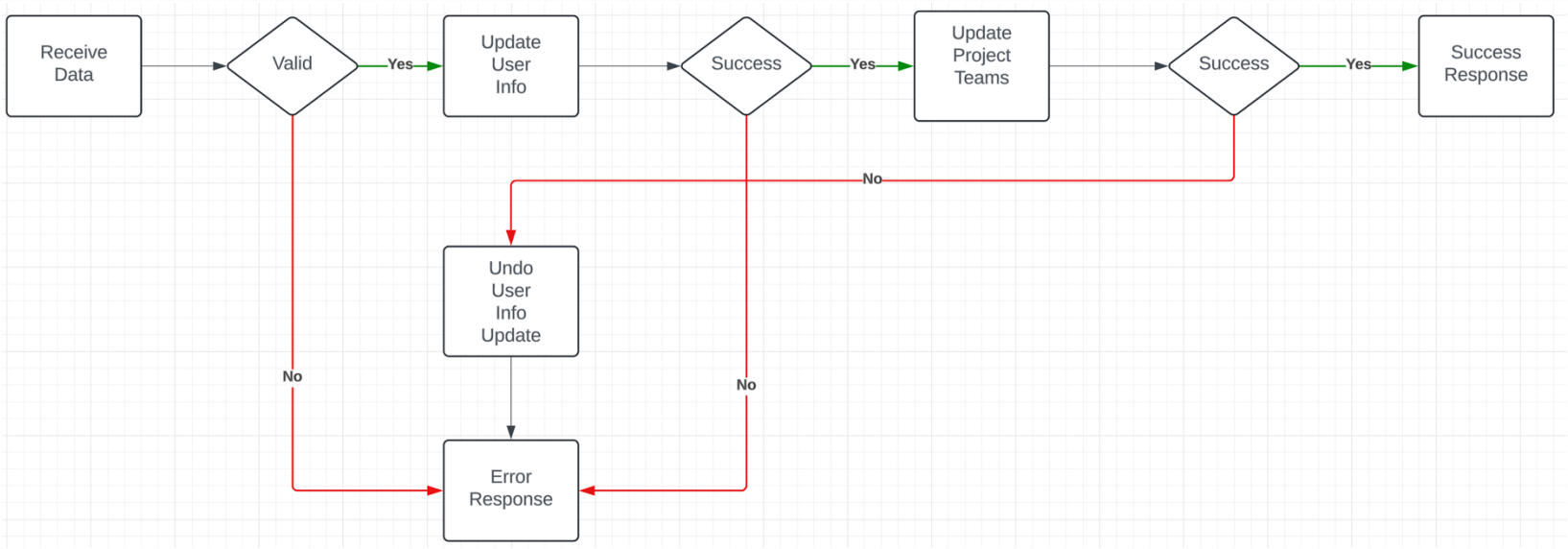
If the cluster of resources might be useful in other solutions → It may be a microservice.

First Group of Microservices

- My first sprint with have 5 microservices:
  - 5 core/atomic microservices:
    - user_info
    - courseworks_adaptor
    - course_info
    - student_coupons
    - project_teams
  - 1 composite/integration microservice: core_course_management
- The composite service:
  - Implements “consistency” in the data in the core services, e.g. “foreign keys.”
  - Eliminates the need for complex logic in the UI or client, which would be impacted by changes.
  - Exposes only information from the basic services that the client requires.
- The composite service is conceptually similar to a database view.

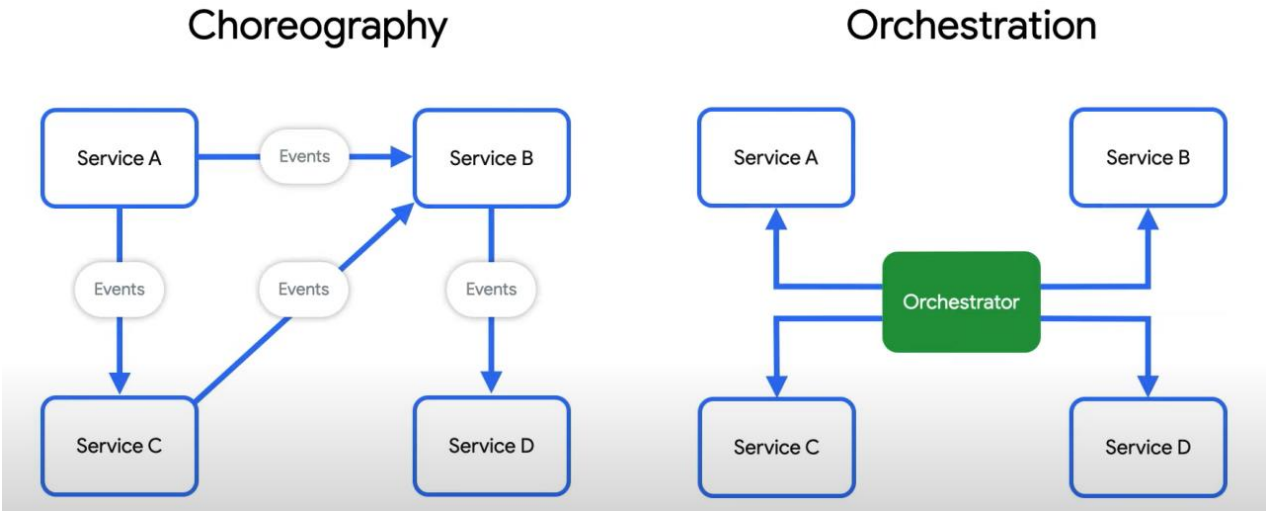


Composite Microservice Logic



- The GET on the composite invokes GET on several atomic services. This can occur synchronously or asynchronously in parallel.
- PUT, POST and DELETE are more complex. Your logic may have to undo some delegated updates that occur if others fail.
- To get started, we will just use simple logic but explore more complex approaches.

Behavioral Composition



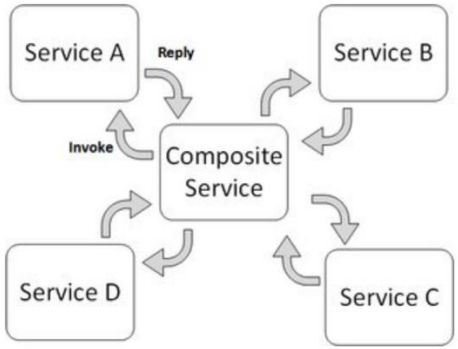
- There are two base patterns: choreography and orchestration.
- There also several different implementation technologies and choices.
- And many related concepts, e.g. Sagas, Event Driven Architectures, Pub/Sub, … …
- We will cover and explore incrementally, but for now we will keep it simple and do traditional “if … then …” code.

Concepts

Service orchestration

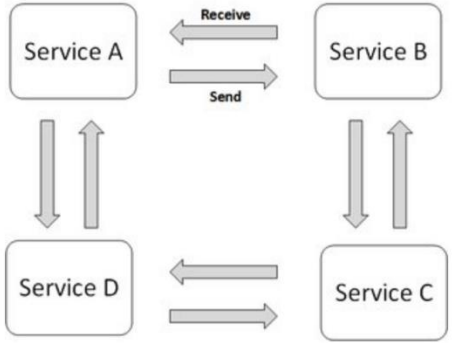
Service orchestration represents a single centralized executable business process (the orchestrator) that coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services.

The relationship between all the participating services are described by a single endpoint (i.e the composite service). The orchestration includes the management of transactions between individual services. Orchestration employs a centralized approach for service composition.



Service Choreography

Service choreography is a global description of the participating services, which is defined by exchange of messages, rules of interaction and agreements between two or more endpoints. Choreography employs a decentralized approach for service composition.

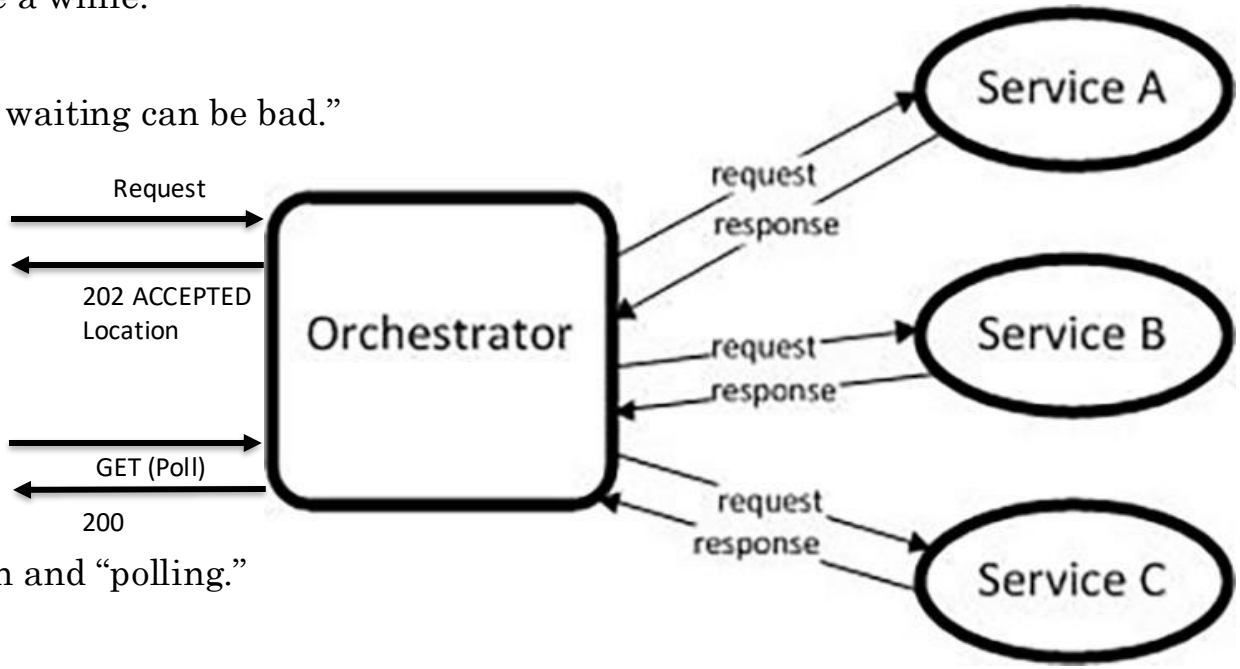


The choreography describes the interactions between multiple services, where as orchestration represents control from one party's perspective. This means that a **choreography** *differs* from an **orchestration** with respect to where the logic that controls the interactions between the services involved should reside.

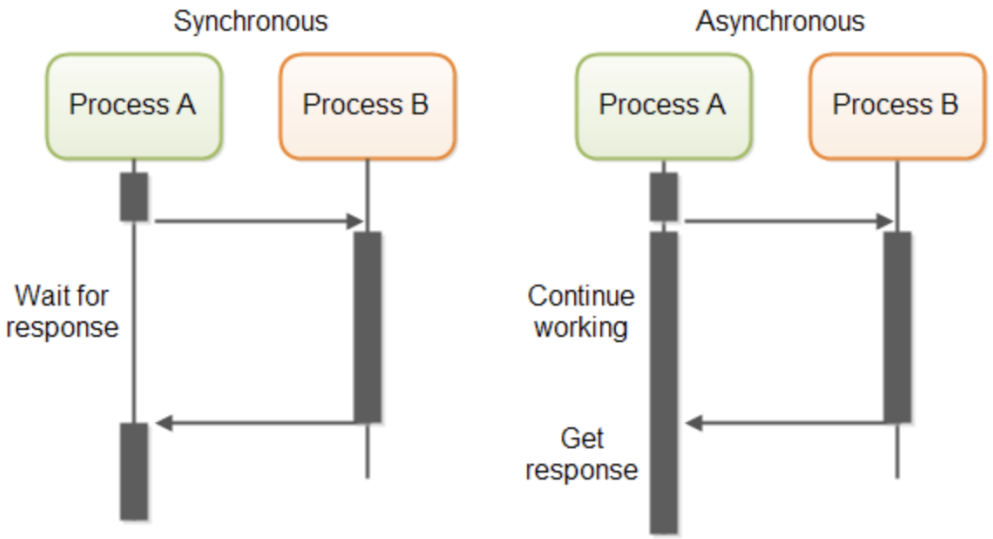
These are not formally, rigorously defined terms.

In Code: Composition and Asynchronous Method Invocation

- Composite operations can “take a while.”
- “Holding open connections and waiting can be bad.”
  - Consumes resources.
  - Connections can break.
  - … …
- There are two options:
  - Polling
  - Callbacks
- Our next pattern is composition and “polling.”
  Create a new user requires:
  - Updating the user database.
  - Verifying a submitted address.
  - Creating an account in the catalog service.



In Code: Service Orchestration/Composition

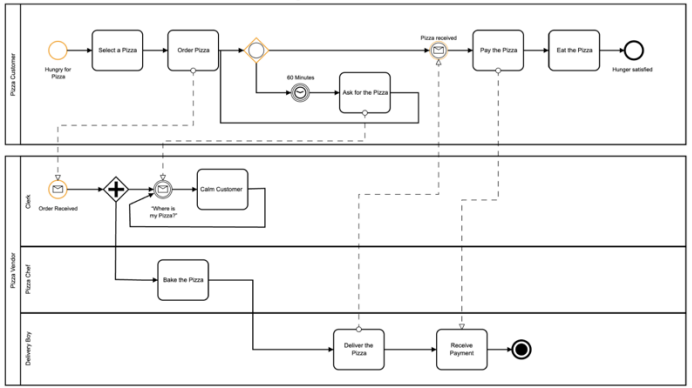


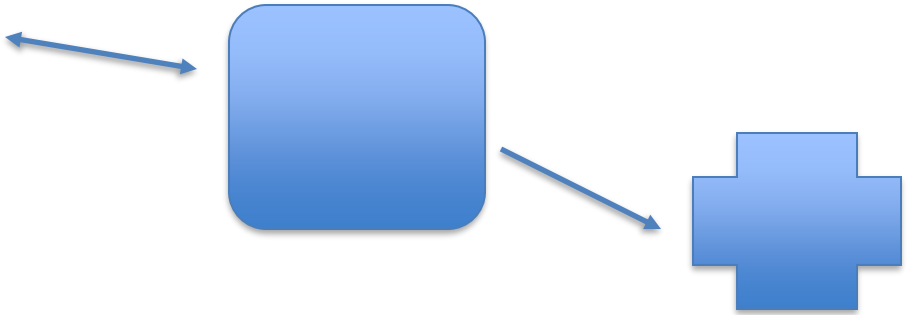
RESTFul HTTP calls can be implemented in both a synchronous and asynchronous fashion at an IO level.

- One call to a composite service
  - May drive calls to multiple other services.
  - Which in turn, may drive multiple calls to other services.
- Synchronous (Wait) and calling one at a time, in order is
  - Inefficient
  - Fragile
  - … …
- Asynchronous has benefits but can result in complex code.

Service Orchestration

- Implementing a complex orchestration in code can become complex.
- High layer abstractions and tools have emerged, e.g. BPMN.
  - "Business Process Model and Notation (BPMN) is a graphical representation for specifying business processes in a business process model."

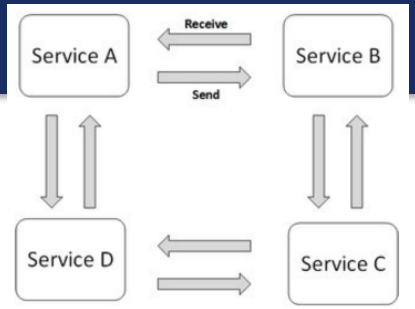


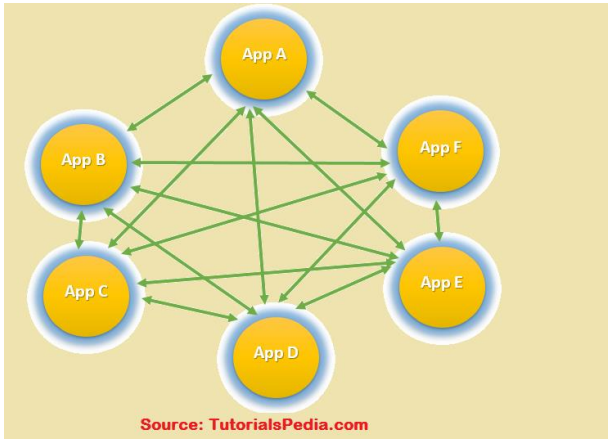


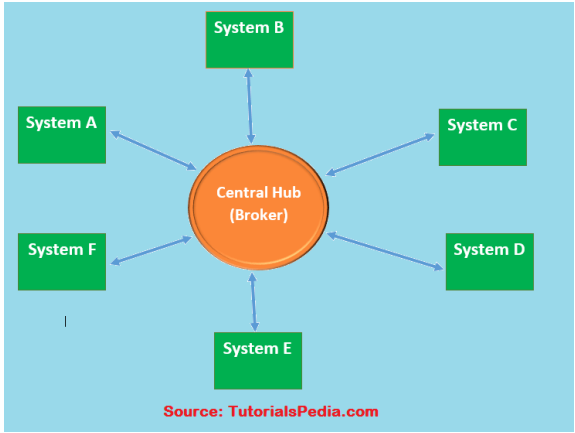
  - There are products for defining BPMN processes, generating code, executing, … …, e.g. Camunda: https://camunda.com/bpmn/
- There are other languages, tools, execution engines, … …

Choreography

- We saw the basic diagram, but …
  this is an anti-pattern and does not scale.

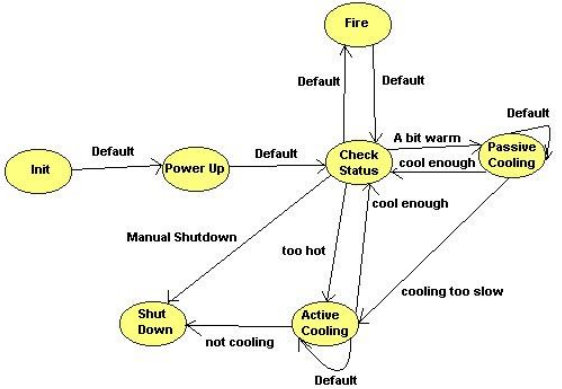


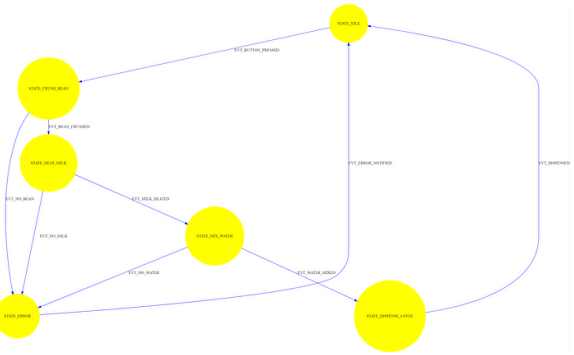


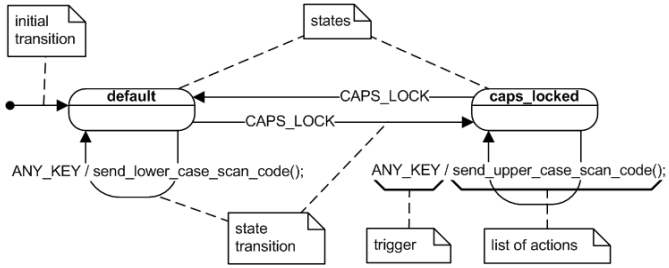


- But, how do you write the event driven microservices?
  - Well, you can just write code … …
  - Or use a state machines abstraction.

State Machine







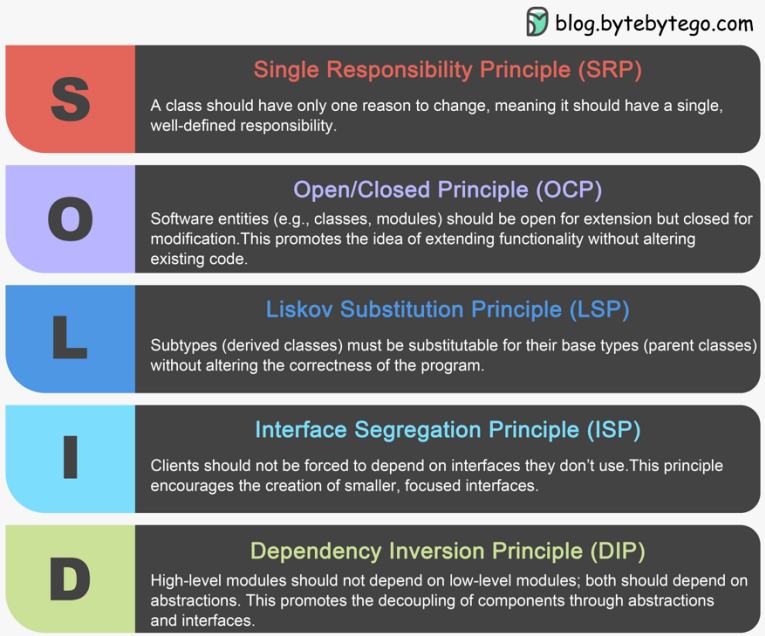
- A state machine is an abstraction.
- There are many models for defining state machines.
- There are also development tools, runtimes, … …
- We will look at a specific example: AWS Step Functions

12 Factor Apps
SOLID

12 Factor Applications

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

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

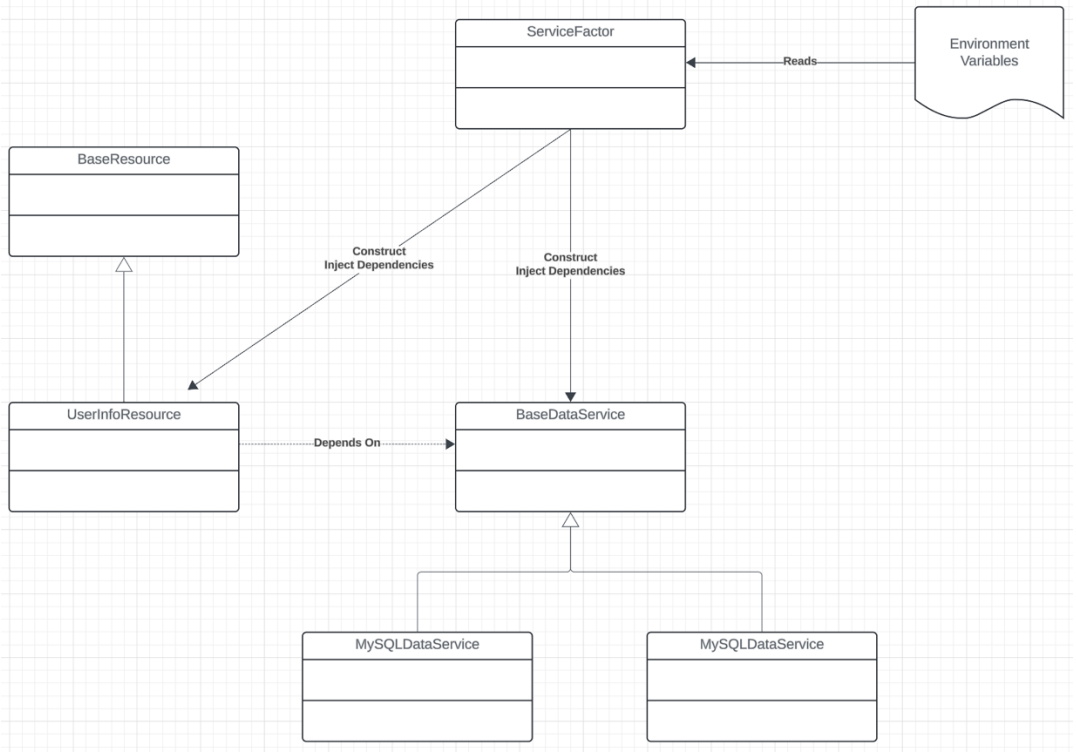
https://medium.com/@saurabh.engg.it/microservices-designing-effective-microservices-by-following-solid-design-principles-a995c3a033a0

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.

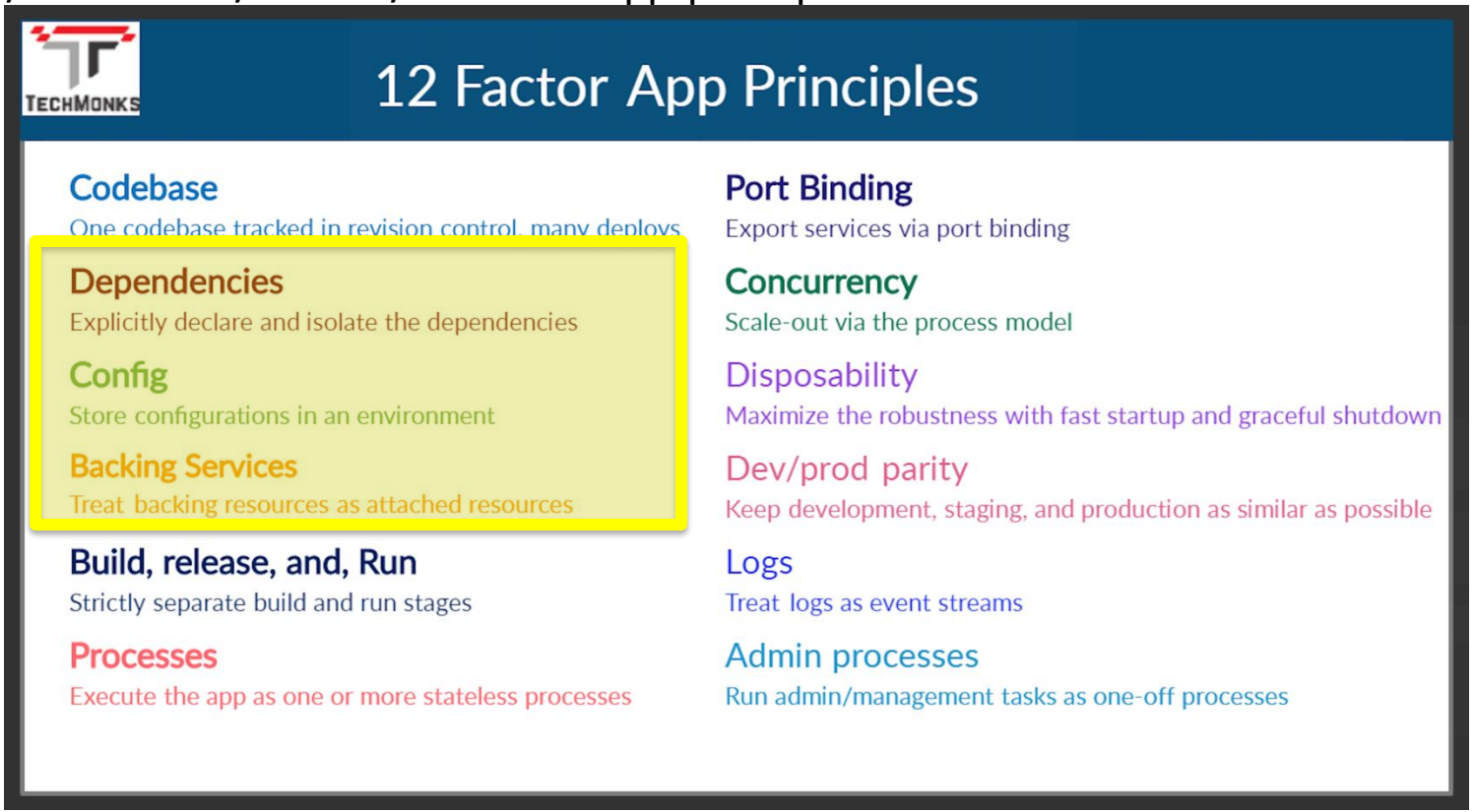
SW Architecture

- Use interfaces, or the language equivalent/approximation.
- A constructor receive a “config” object that is a dictionary of
  - Interface name.
  - Reference to implementation.
- Parameters and configuration properties are also in the config, e.g.
  - URLs
  - “Passwords”
  - Database and table names
  - … …
- A “service factory” creates the specific instances and configs,
  and wires it all together.



12 Factor Applications

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



Dependencies

II. Dependencies

Explicitly declare and isolate dependencies

Most programming languages offer a packaging system for distributing support libraries, such as CPAN for Perl or Rubygems for Ruby. Libraries installed through a packaging system can be installed system-wide (known as "site packages") or scoped into the directory containing the app (known as "vendoring" or "bundling").

A twelve-factor app never relies on implicit existence of system-wide packages. It declares all dependencies, completely and exactly, via a *dependency declaration* manifest. Furthermore, it uses a *dependency isolation* tool during execution to ensure that no implicit dependencies "leak in" from the surrounding system. The full and explicit dependency specification is applied uniformly to both production and development.

For example, Bundler for Ruby offers the `Gemfile` manifest format for dependency declaration and `bundle exec` for dependency isolation. In Python there are two separate tools for these steps – Pip is used for declaration and Virtualenv for isolation. Even C has Autoconf for dependency declaration, and static linking can provide dependency isolation. No matter what the toolchain, dependency declaration and isolation must always be used together – only one or the other is not sufficient to satisfy twelve-factor.

One benefit of explicit dependency declaration is that it simplifies setup for developers new to the app. The new developer can check out the app's codebase onto their development machine, requiring only the language runtime and dependency manager installed as prerequisites. They will be able to set up everything needed to run the app's code with a deterministic *build command*. For example, the build command for Ruby/Bundler is `bundle install`, while for Clojure/Leiningen it is `lein deps`.

Twelve-factor apps also do not rely on the implicit existence of any system tools. Examples include shelling out to ImageMagick or `curl`. While these tools may exist on many or even most systems, there is no guarantee that they will exist on all systems where the app may run in the future, or whether the version found on a future system will be compatible with the app. If the app needs to shell out to a system tool, that tool should be vendored into the app.

- Most application frameworks have a dependency declaration concept.
- I have been showing examples with:
  - Python import
  - requirements.txt
  - etc.
- My examples also show encapsulating libraries and APIs with Python classes.
- II. Dependencies handles "the code," but we still need to handle the instance. There is a difference between, e.g.
  - pymysql
  - A specific connection.

III. Configurations

III. Config

Store config in the environment

An app's *config* is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

- Resource handles to the database, Memcached, and other backing services
- Credentials to external services such as Amazon S3 or Twitter
- Per-deploy values such as the canonical hostname for the deploy

Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires **strict separation of config from code**. Config varies substantially across deploys, code does not.

A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.

Note that this definition of "config" does **not** include internal application config, such as `config/routes.rb` in Rails, or how code modules are connected in Spring. This type of config does not vary between deploys, and so is best done in the code.

Another approach to config is the use of config files which are not checked into revision control, such as `config/database.yml` in Rails. This is a huge improvement over using constants which are checked into the code repo, but still has weaknesses: it's easy to mistakenly check in a config file to the repo; there is a tendency for config files to be scattered about in different places and different formats, making it hard to see and manage all the config in one place. Further, these formats tend to be language- or framework-specific.

The twelve-factor app stores config in *environment variables* (often shortened to *env vars* or *env*). Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard.

Another aspect of config management is grouping. Sometimes apps batch config into named groups (often called "environments") named after specific deploys, such as the `development`, `test`, and `production` environments in Rails. This method does not scale cleanly: as more deploys of the app are created, new environment names are necessary, such as `staging` or `qa`. As the project grows further, developers may add their own special environments like `joes-staging`, resulting in a combinatorial explosion of config which makes managing deploys of the app very brittle.

In a twelve-factor app, env vars are granular controls, each fully orthogonal to other env vars. They are never grouped together as "environments", but instead are independently managed for each deploy. This is a model that scales up smoothly as the app naturally expands into more deploys over its lifetime.

- Each of the deployment environments we have seen has a method for setting environment variables.
  - https://docs.aws.amazon.com/cloud9/latest/user-guide/env-vars.html
  - https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/environments-cfg-softwaresettings.html
  - https://cloud.google.com/functions/docs/configuring/env-var
- CI/CD, GitHub actions, etc. can integrate environment variables with the development process.
- Vaults and secrets managers are a better approach for passwords.

IV. Backing Services

IV. Backing services

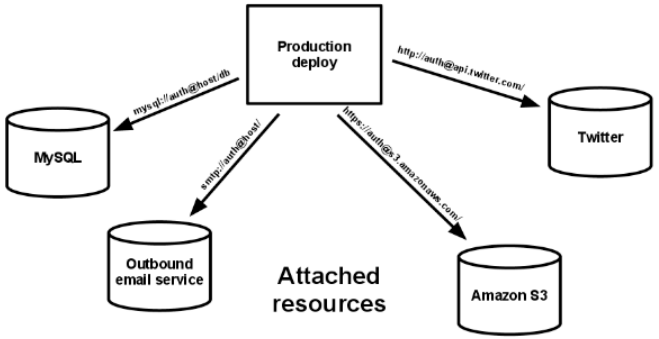
Treat backing services as attached resources

A *backing service* is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as MySQL or CouchDB), messaging/queueing systems (such as RabbitMQ or Beanstalkd), SMTP services for outbound email (such as Postfix), and caching systems (such as Memcached).

Backing services like the database are traditionally managed by the same systems administrators who deploy the app's runtime. In addition to these locally-managed services, the app may also have services provided and managed by third parties. Examples include SMTP services (such as Postmark), metrics-gathering services (such as New Relic or Loggly), binary asset services (such as Amazon S3), and even API-accessible consumer services (such as Twitter, Google Maps, or Last.fm).

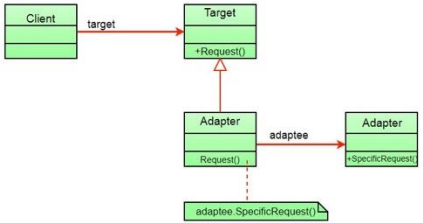
The code for a twelve-factor app makes no distinction between local and third party services. To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the config. A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third party (such as Amazon RDS) without any changes to the app's code. Likewise, a local SMTP server could be swapped with a third-party SMTP service (such as Postmark) without code changes. In both cases, only the resource handle in the config needs to change.

Each distinct backing service is a *resource*. For example, a MySQL database is a resource; two MySQL databases (used for sharding at the application layer) qualify as two distinct resources. The twelve-factor app treats these databases as *attached resources*, which indicates their loose coupling to the deploy they are attached to.



Resources can be attached to and detached from deploys at will. For example, if the app's database is misbehaving due to a hardware issue, the app's administrator might spin up a new database server restored from a recent backup. The current production database could be detached, and the new database attached – all without any code changes.

- My code has shown encapsulating backing services with a shallow adaptor.
- This is an example of the *adaptor pattern.*



- You seen examples of things like AbstractBaseDataService, … …
- The approach is also an application of several aspects of SOLID. (https://en.wikipedia.org/wiki/SOLID)

Structural Composition

- We can use four patterns/principles for service composition ”in code.”
  - Dependency Injection
  - Service Factory
  - Service Locator
  - Metadata in the environment for configuration.

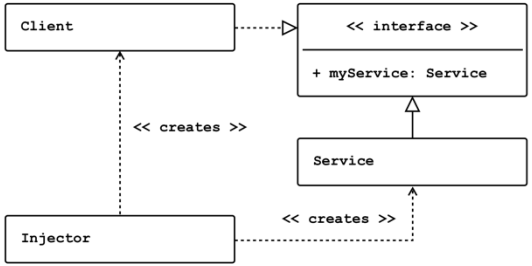
- I did some simple examples in https://github.com/donald-f-ferguson/E6156-Public-Examples/tree/master/old_examples/simple_pattern_examples.

- Or actually, I asked ChatGPT to produce simple examples.

- And, these are conceptual, and I am not super happy with them.

Dependency Injection

- Concepts (https://en.wikipedia.org/wiki/Dependency_injection):
  - In software engineering, dependency injection is a technique in which an object receives other objects that it depends on, called dependencies.
  - The intent behind dependency injection is to achieve separation of concerns of construction and use of objects. This can increase readability and code reuse.



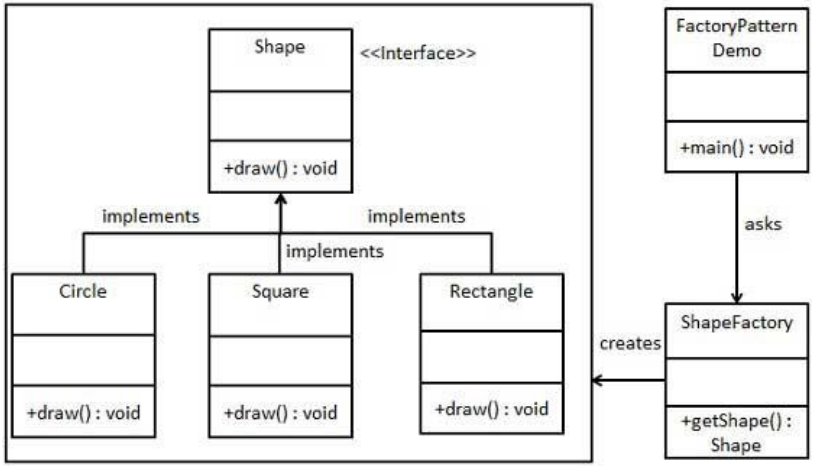

```
class UserResource(BaseApplicationResource):

    def __init__(self, config_info):
        super().__init__(config_info)
```

- There is no single approach. There are several frameworks, tools, etc.
- Like many other things, it is easy to get carried away and become dogmatic.
- I follow a single approach:
  - A Context class converts environment and other configuration and provides to application.
  - The top-level application injects a config_info object into services.

Service Factory

- "In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor." (https://en.wikipedia.org/wiki/Factory_method_pattern)
- You will sometimes see me use this for
  - Abstraction between a REST resource impl. and the data service.
  - Allows changing the database service, model, etc. without modifying the code.
  - Concrete implementation choices are configured via properties, metadata, …
- You can use in many situations.



https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Service Locator

- “The service locator pattern is a design pattern used in software development to encapsulate the processes involved in obtaining a service with a strong abstraction layer. This pattern uses a central registry known as the "service locator", which on request returns the information necessary to perform a certain task. Proponents of the pattern say the approach simplifies component-based applications where all dependencies are cleanly listed at the beginning of the whole application design, …” (https://en.wikipedia.org/wiki/Service_locator_pattern)