W4153 – Cloud Computing

Lecture 4:

Microservices(3), Web Content, OAuth2

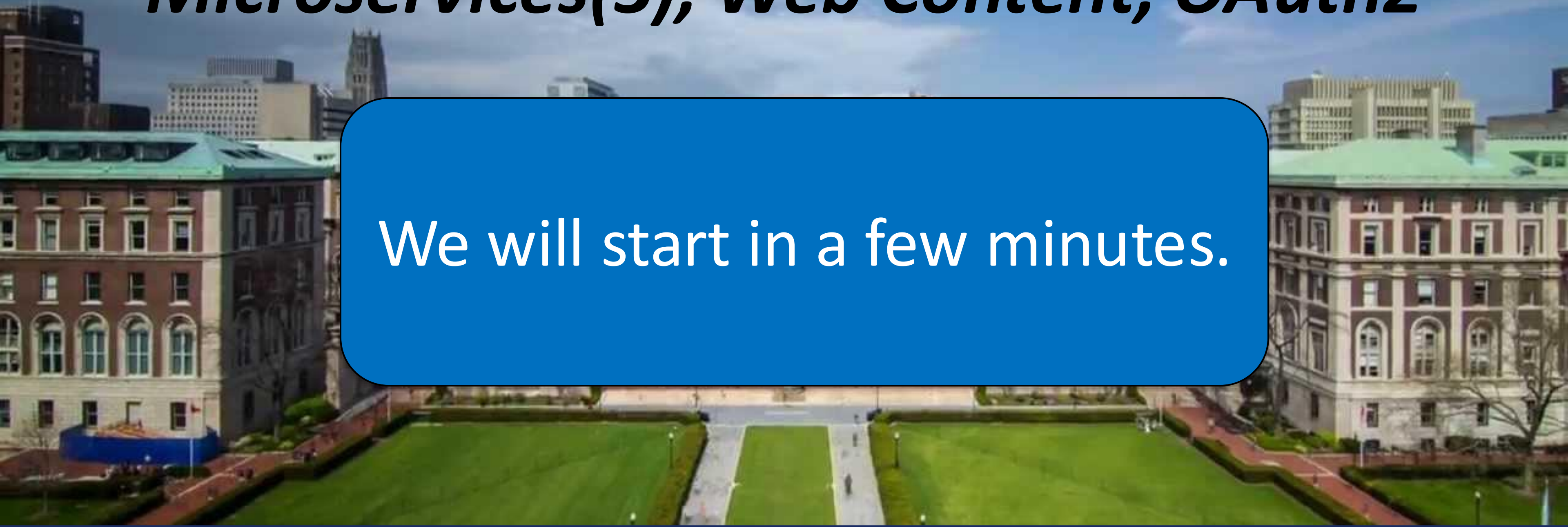
© Donald F. Ferguson, 2024

Columbia | Engineering
The Fu Foundation School of Engineering and Applied Science

W4153 – Cloud Computing

Lecture 4:

Microservices(3), Web Content, OAuth2



We will start in a few minutes.

© Donald F. Ferguson, 2024

Contents

© Donald F. Ferguson, 2024

Contents

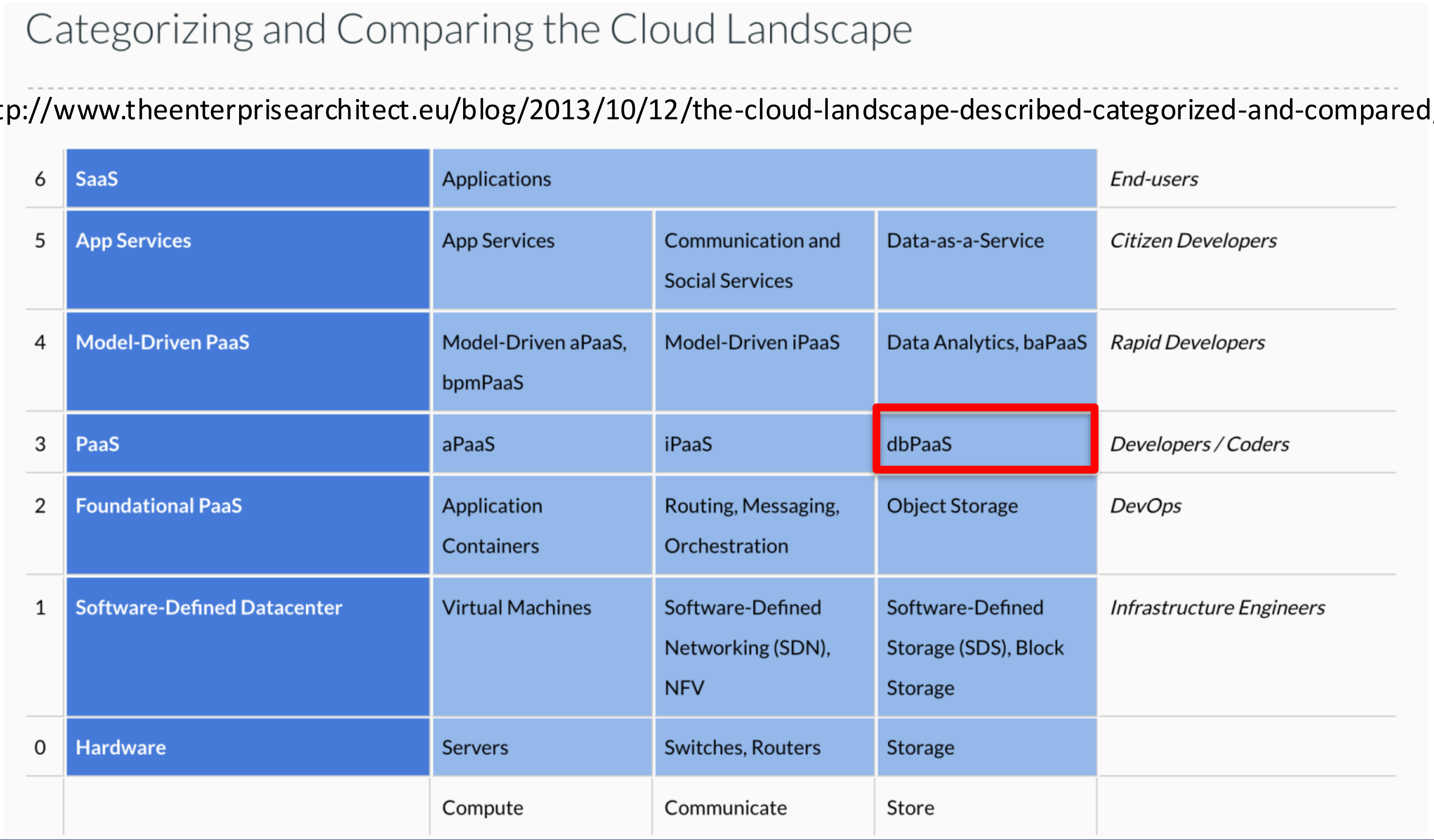
© Donald F. Ferguson, 2024

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

DBaaS

© Donald F. Ferguson, 2024

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	

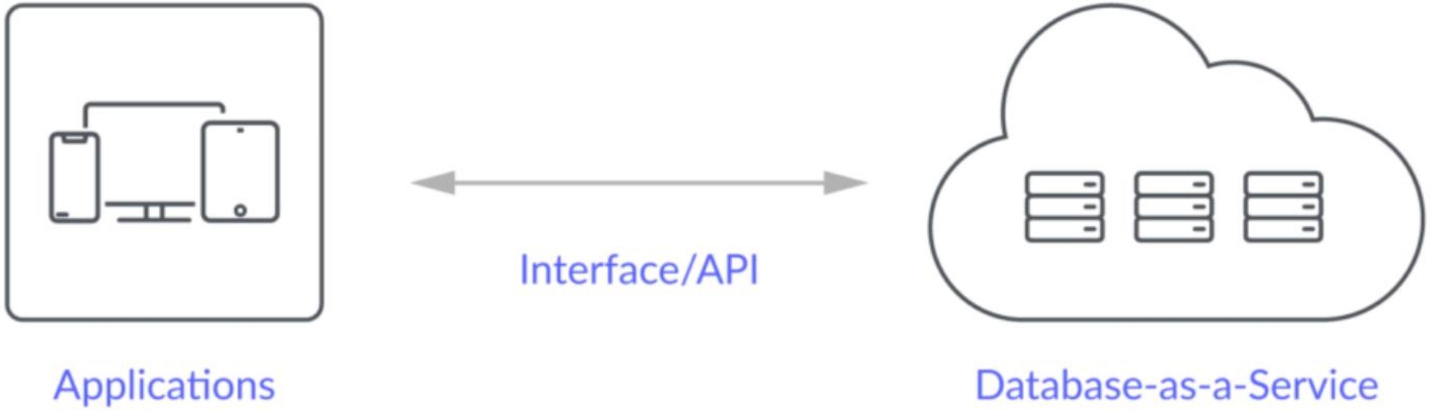
© Donald F. Ferguson, 2024

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

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.

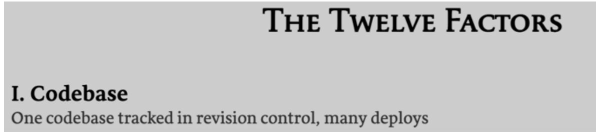
Database services take care of scalability and high availability of the database. Database services make the underlying software-stack transparent to the user." (Wikipedia)



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

Example and Walkthrough

- My simple approach (We do something for sophisticated in practice):
  - A simple full-stack web application has 3 “C4 containers:”
    - UI application that runs in the browser.
    - Microservice implementation in an application server.
    - Database “application” in a database management system.
  - There is a single implementation of each “container.” There are two deployments for each container:
    - Local (on my development laptop)
    - Deployed in the cloud.
  - I have 3 configurations that I can use for testing.
- Walkthrough of AWS approach.
  - DB setup and configuration, DataGrip
  - Environment variables
  - Microservice
- Google equivalent.



THE TWELVE FACTORS

I. Codebase
One codebase tracked in revision control, many deploys

Test Config	UI	Microservice	DB App
All local	Angular Embedded	Local FastAPI	Local MySQL
Mostly Local	Angular Embedded	Local FastAPI	DBaaS
Mostly Cloud	Angular Embedded	Cloud Deployed	DBaaS
Fully Deployed	Cloud Deployed	Cloud Deployed	DBaaS

© Donald F. Ferguson, 2024

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

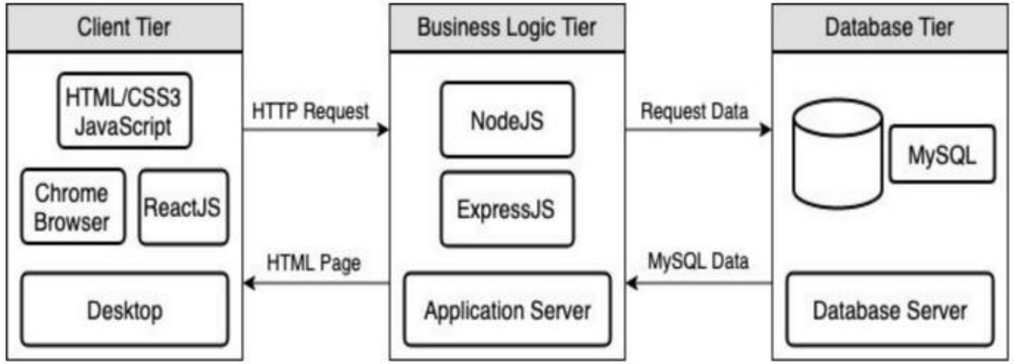
BLOBs
Web Content

© Donald F. Ferguson, 2024

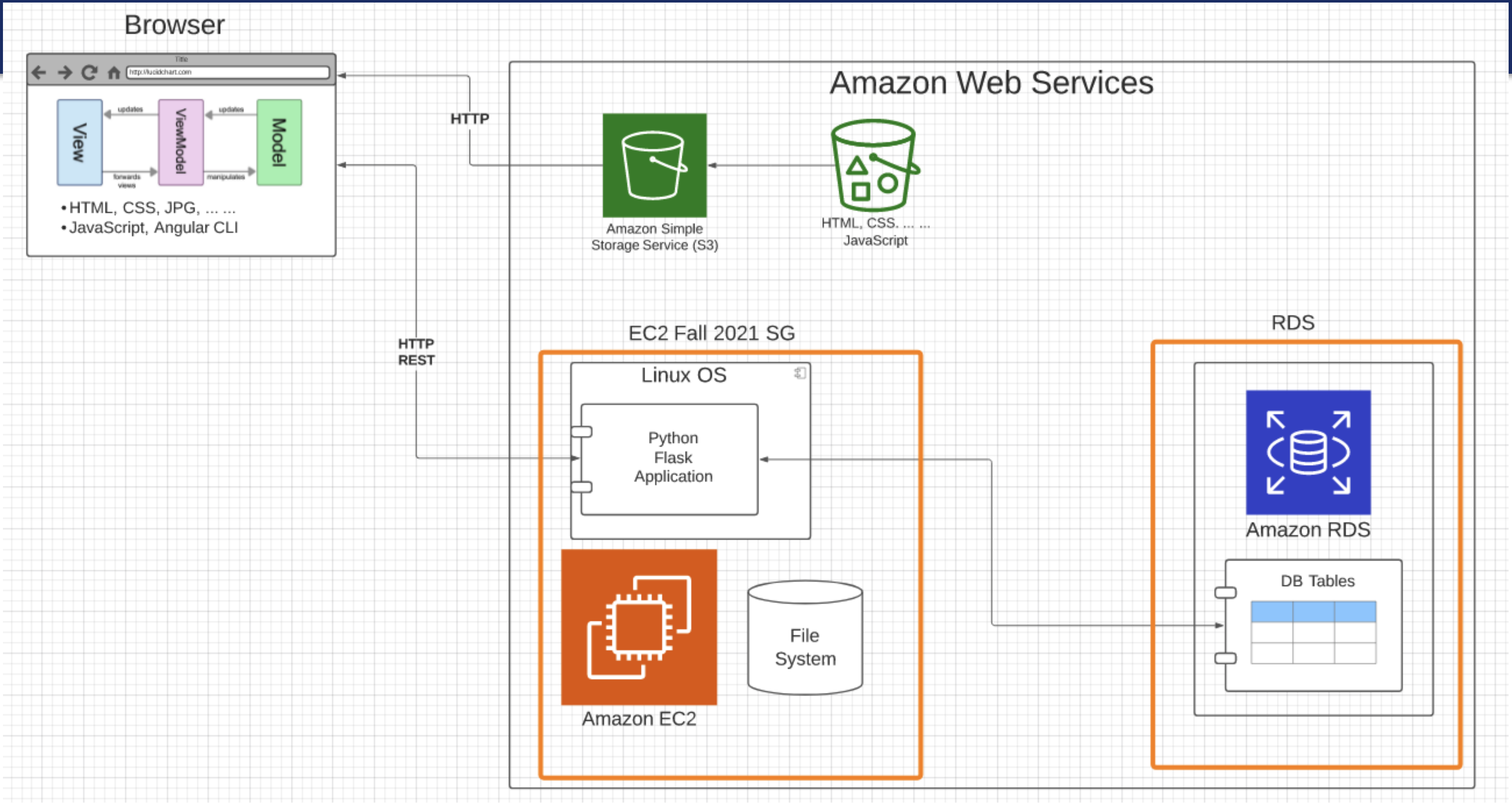
COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

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.
  - The page can be static and returned “as is.” There are also template languages for parameterized returns, e.g. Jinja2
- But, running application logic and simply delivering files are two different “concerns.”
  - Having one runtime implement both would violate the SOLID principle.
  - The design points are different: “You can be a floor wax or a mouth wash, but not both.”
- So, we need a separate “web server.”



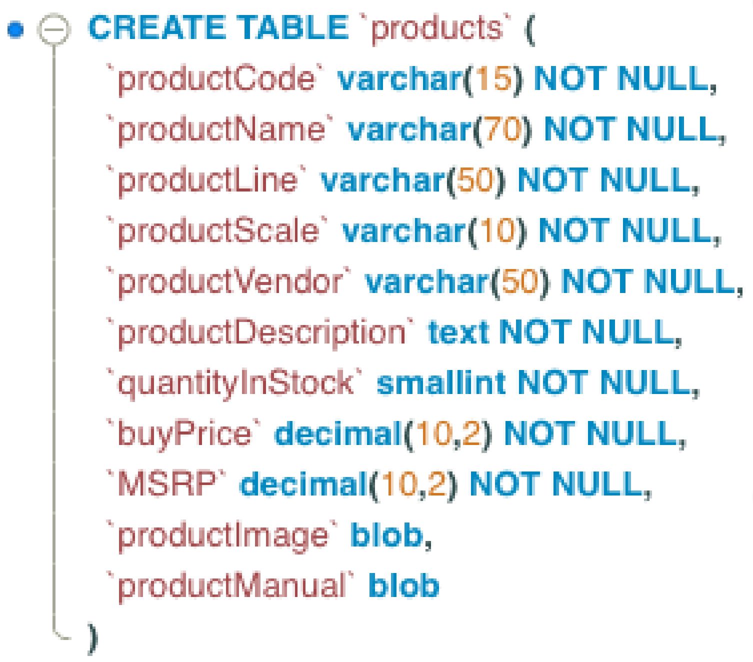
© Donald F. Ferguson, 2024

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

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
)
```

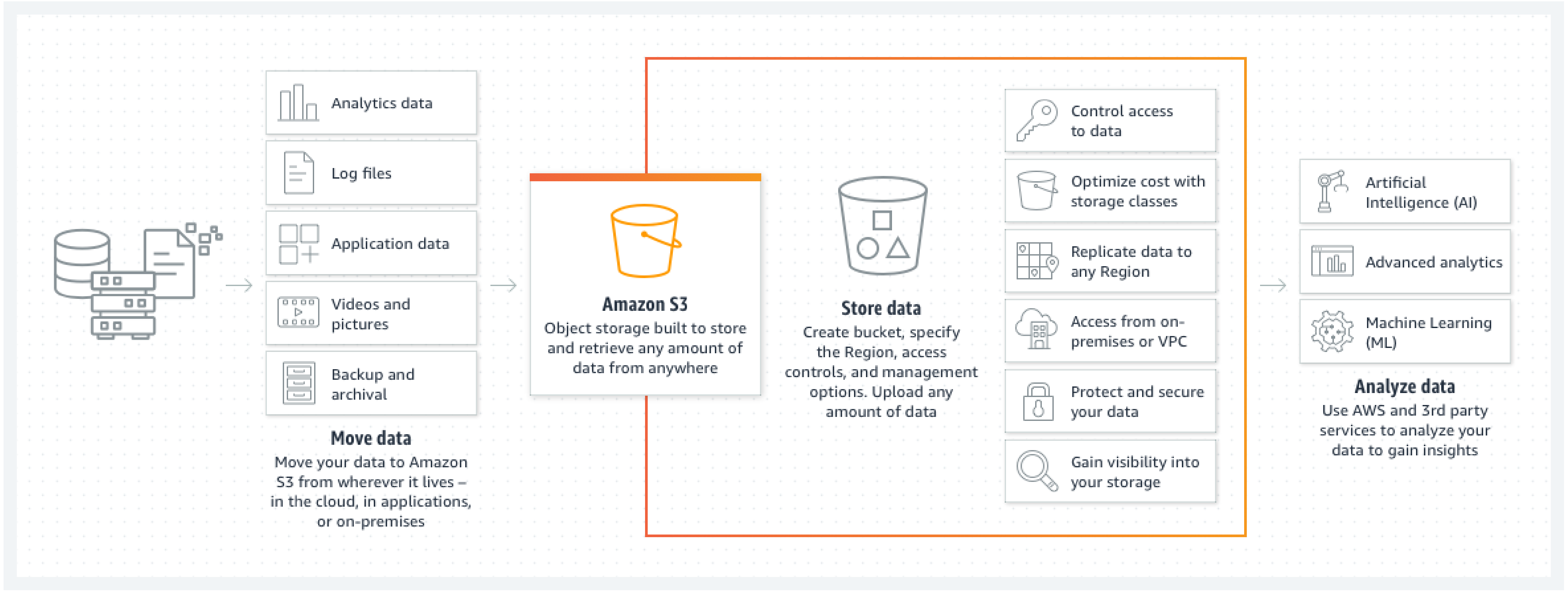
- Structured or semi-structured information about an entity.
  - Queryable
  - Editable on forms
  - etc.
- Unstructured: A file-like, opaque property associated with the entity. This is a BLOB.

URLs have primarily replaced BLOBs.

© Donald F. Ferguson, 2024

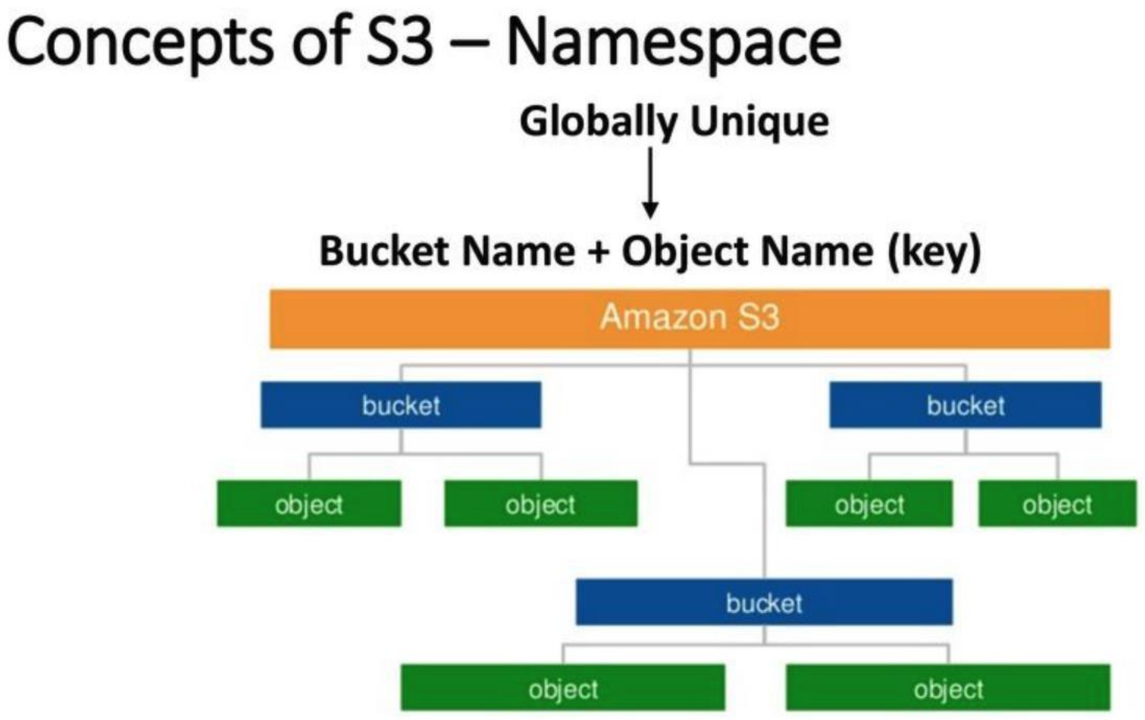
COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Amazon S3


Analytics data
Log files
Application data
Videos and pictures
Backup and archival
Move data
Move your data to Amazon S3 from wherever it lives – in the cloud, in applications, or on-premises
Amazon S3
Object storage built to store and retrieve any amount of data from anywhere
Store data
Create bucket, specify the Region, access controls, and management options. Upload any amount of data
Control access to data
Optimize cost with storage classes
Replicate data to any Region
Access from on-premises or VPC
Protect and secure your data
Gain visibility into your storage
Artificial Intelligence (AI)
Advanced analytics
Machine Learning (ML)
Analyze data
Use AWS and 3rd party services to analyze your data to gain insights

There are equivalent capabilities on most clouds.

AWS S3



Can support a logical hierarchy with a convention on keys.

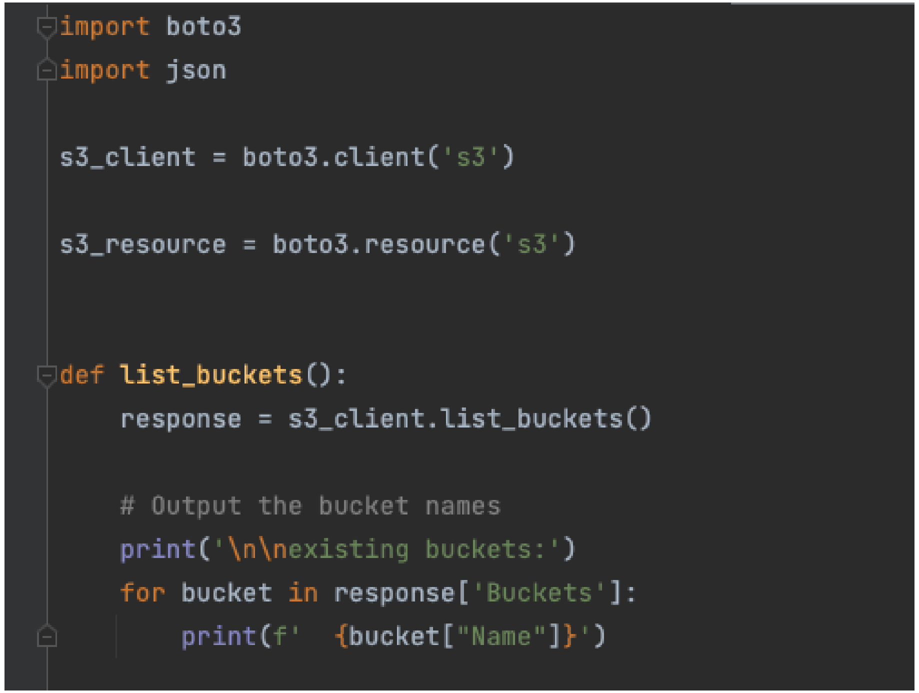
- x/y
- x/z
- x/y/z
- … …

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

© Donald F. Ferguson, 2024

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

Simple S3 Example



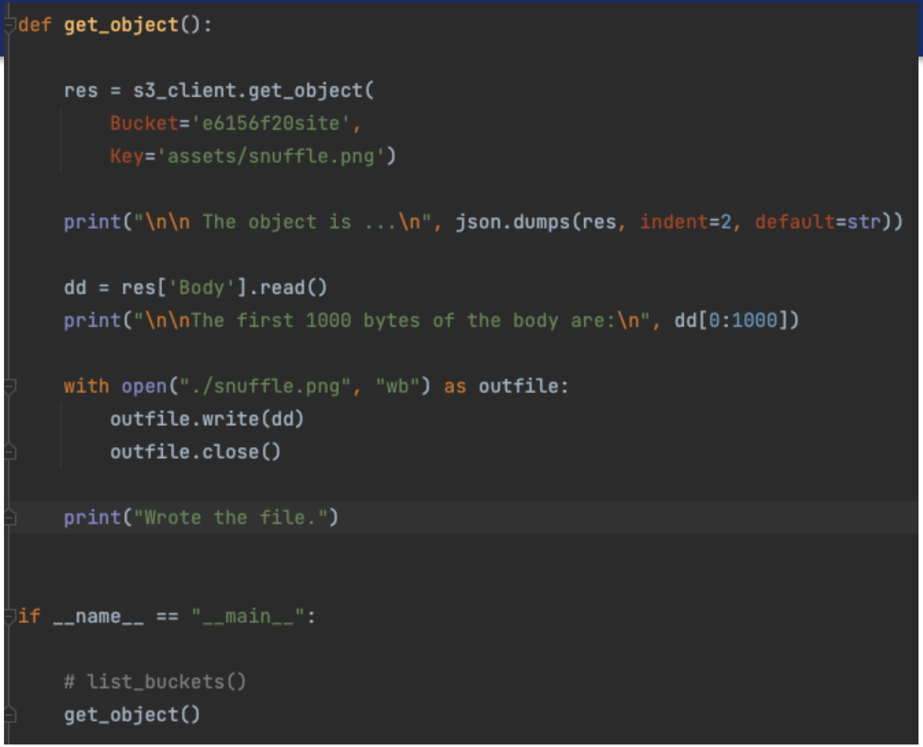
```python
import boto3
import json

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()
```

See simple_examples/S3

© Donald F. Ferguson, 2024

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

Simple Example

- https://e6156f20site.s3.us-east-2.amazonaws.com/index.html
- Why not deliver static content out of the application?
  After all it is serving HTTP.
  - You get a lot of things figured out and implemented for you.
    - Scalability
    - Caching
    - Availability
    - Encryption
    - etc.
  - You want to focus on things that are your apps value, and not implementing commodity functions.
- In the Ansys cloud platform, we use BLOBs to store CAD files, images, complex result data files, … …

OpenID, OAuth2

© Donald F. Ferguson, 2024

OpenIDC and Oauth2

- **OAuth 2.0**
  - OAuth 2.0 is a protocol for **authorizing client applications** to access protected resources. OAuth 2.0 provides consented access and restricts actions of what the client application can perform on resources on behalf of the user, without ever sharing the user's credentials.
- **OpenID Connect**
  - OpenID Connect is an identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable manner.
  - OpenID exists to enable **Federated Authentication**, where users are able to authenticate with the same identity across multiple web applications. Both users and web applications trust identity providers, such as Google, Yahoo!, and Facebook, to store user profile information and authenticate users on behalf of the application. This eliminates the need for each web application to build its own custom authentication system, and it makes it much easier and faster for users to sign up and sign into sites around the Web.

https://medium.com/software-development-turkey/understanding-oauth-2-0-and-openid-connect-777eb1fc27f

© Donald F. Ferguson, 2024
COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

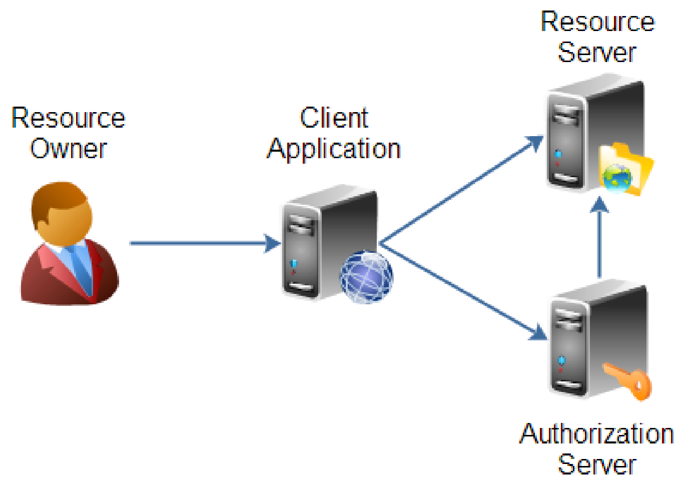
Overview (http://tutorials.jenkov.com/oauth2/index.html)

- Resource Owner
  - Controls *access* to the "data."
  - Facebook **user**, LinkedIn **user**, …
- Resource Server
  - The website that holds/manages info.
  - Facebook, LinkedIn, …
  - And provides access API.
- Client Application
  - "The product you implemented."
  - Wants to read/update
    - "On your behalf"
    - The data the data that the Resource Server maintain
- Authorization Server
  - Grants/rejects authorization
  - Based on Resource Owner decisions.
  - Usually (logically) the same as Resource Server.

OAuth 2.0 defines the following roles of users and applications:

- Resource Owner
- Resource Server
- Client Application
- Authorization Server

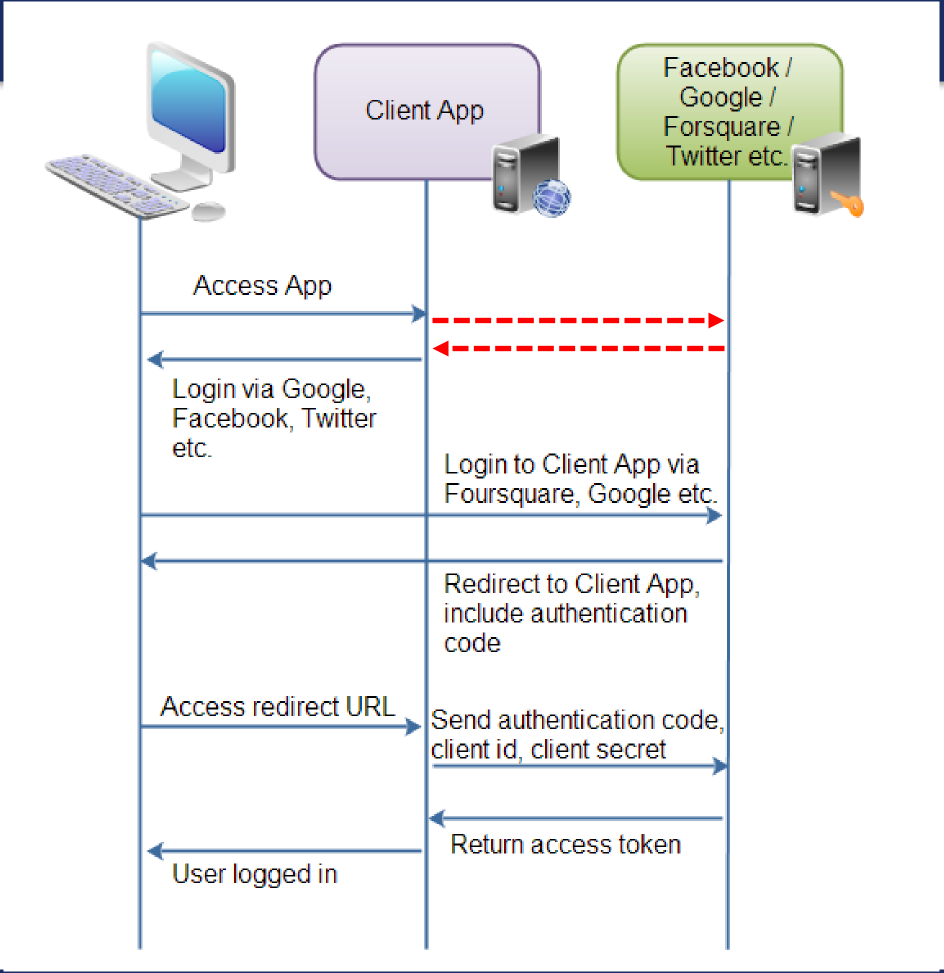
These roles are illustrated in this diagram:



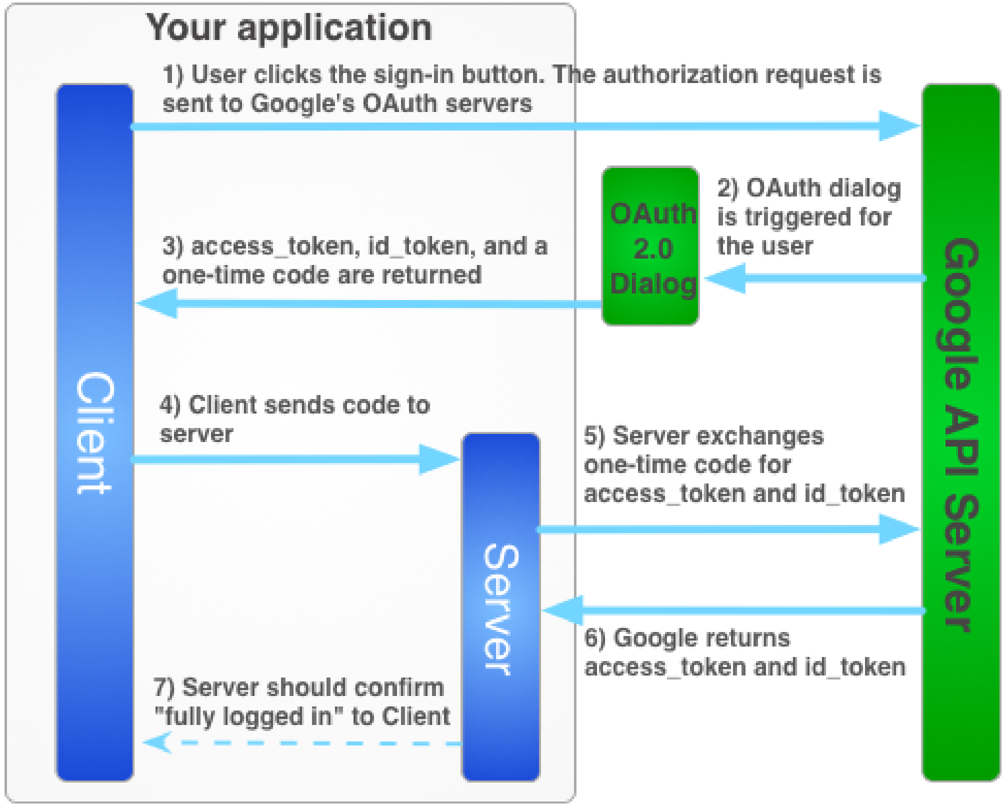
Columbia Engineering
The Fu Foundation School of Engineering and Applied Science

Roles/Flows

- User Clicks “Logon with XXX”
  - Redirect user to XXX
  - With Client App application ID.
  - And permissions app requests.
  - Code MAY have to call Resource Server to get a token to include in redirect URL.
- Browser redirected to XXX
  - Logon on to XXX prompt.
  - Followed by “do you grant permission to …”
  - Clicking button drives a redirect back to Client App. URL contains a temporary token.
- User/Browser
  - Redirected to Client App URL with token.
  - Client App calls XXX API
  - Obtains access token.
  - Returns to User.
- Client App can use access token on API calls.


Client App
Facebook / Google / Forsquare / Twitter etc.
Access App
Login via Google, Facebook, Twitter etc.
Login to Client App via Foursquare, Google etc.
Redirect to Client App, include authentication code
Access redirect URL
Send authentication code, client id, client secret
Return access token
User logged in

Google


Your application
1) User clicks the sign-in button. The authorization request is sent to Google's OAuth servers
OAuth 2.0 Dialog
2) OAuth dialog is triggered for the user
3) access_token, id_token, and a one-time code are returned
Client
4) Client sends code to server
Server
5) Server exchanges one-time code for access_token and id_token
6) Google returns access_token and id_token
7) Server should confirm "fully logged in" to Client
Google API Server

© Donald F. Ferguson, 2024

Walkthrough

- There are good tutorials: https://medium.com/software-development-turkey/understanding-oauth-2-0-and-openid-connect-777eb1fc27f
- Show Coupon App code
  - Show the in-progress coupon issue app.
- Google Cloud setup
  - Consent
  - URLs
- Overall setup of application:
  - URLs
  - Local example
  - Cloud example

© Donald F. Ferguson, 2024
COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

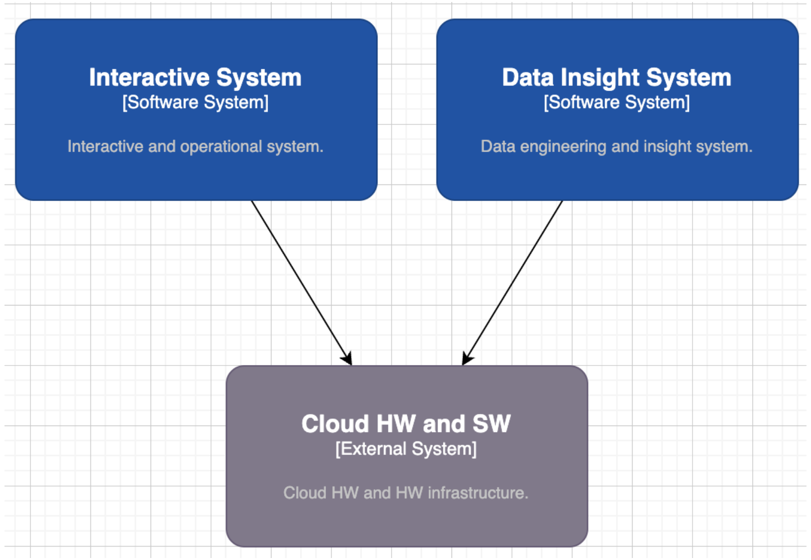
Microservices Continued

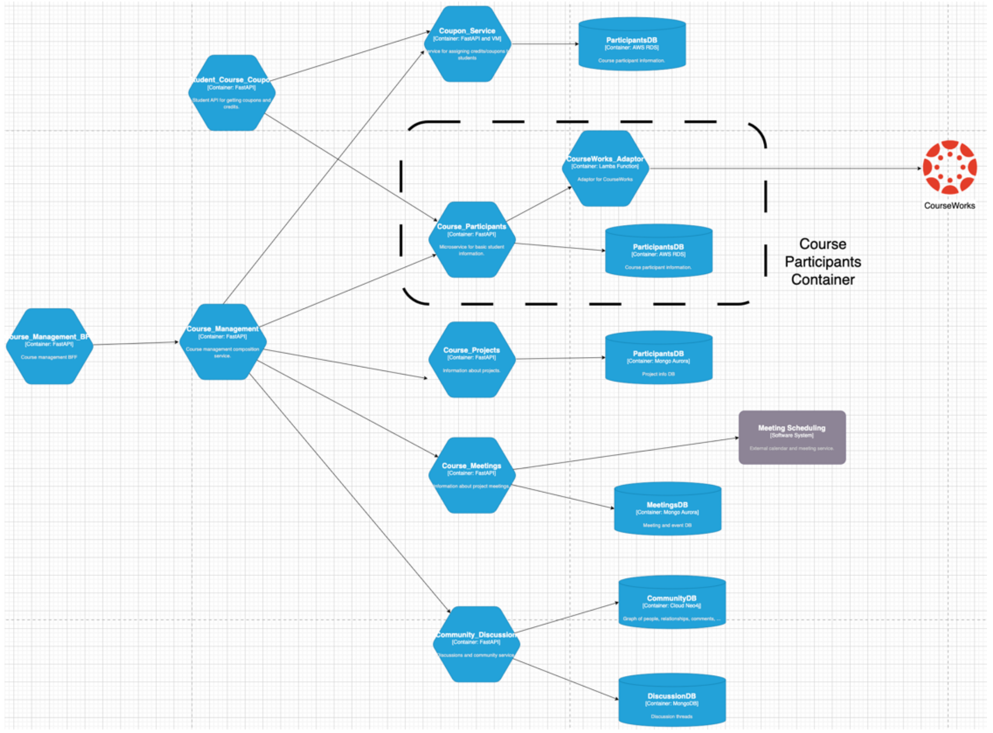
© Donald F. Ferguson, 2024

Composites

© Donald F. Ferguson, 2024

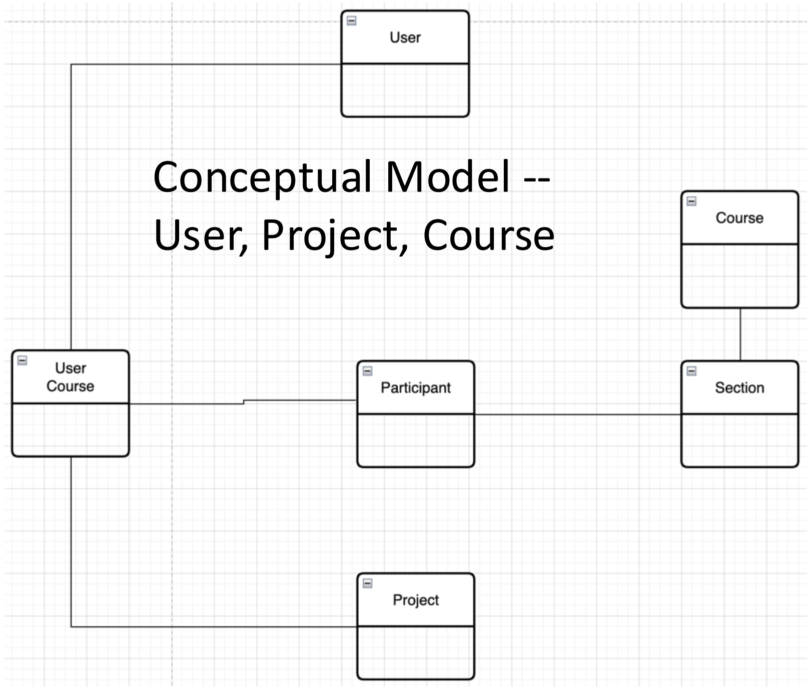
(Partial) Logical View – Interactive System





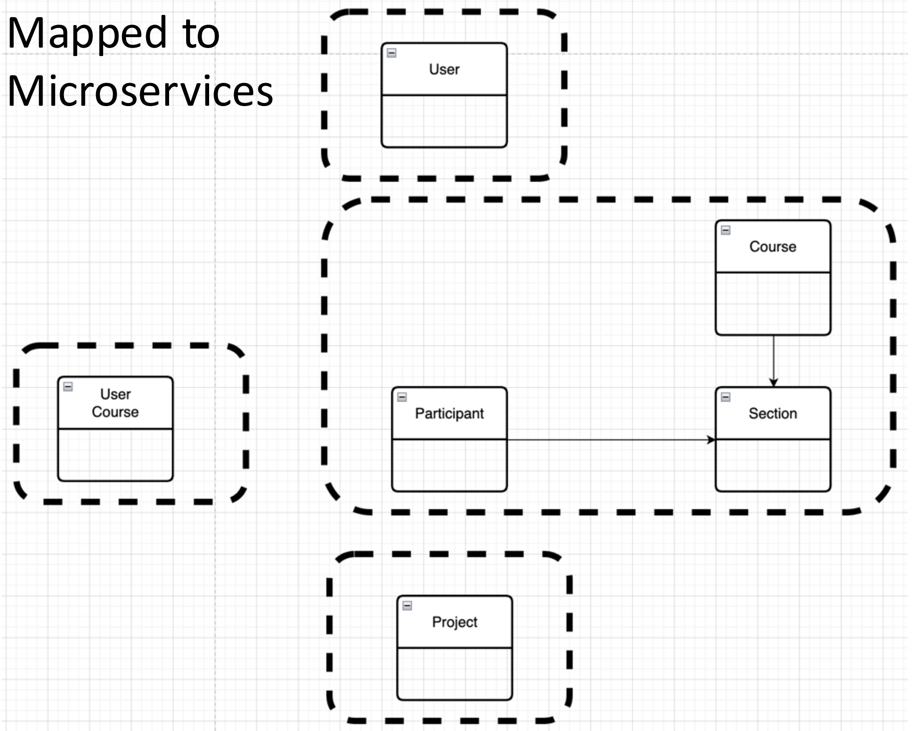
© Donald F. Ferguson, 2024
COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Resource Oriented and Microservices – My Project



Conceptual Model -- User, Project, Course

Mapped to Microservices



This creates some interesting challenges, e.g.

- Joins
- Foreign Keys

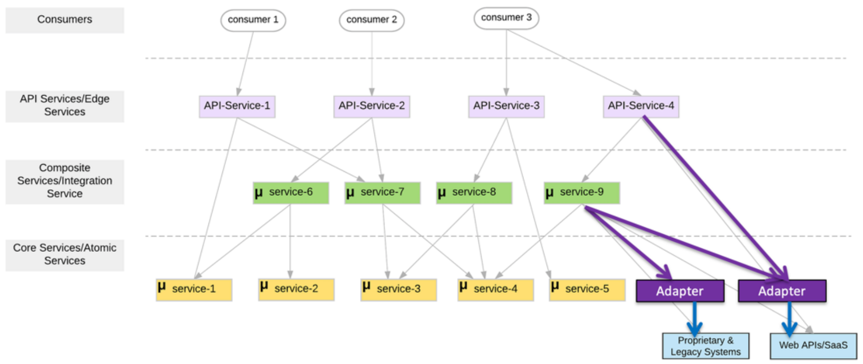
If the cluster of resources might be useful in other solutions → It may be a microservice.

© Donald F. Ferguson, 2024

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

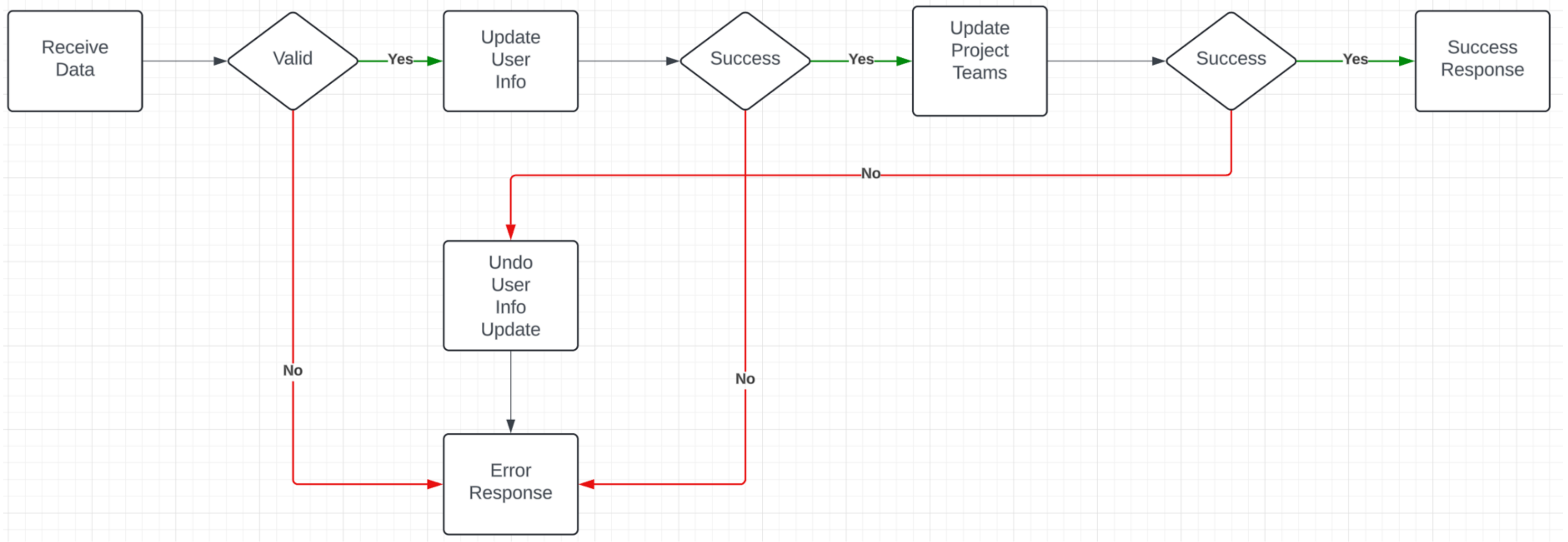
First Group of Microservices

- 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.



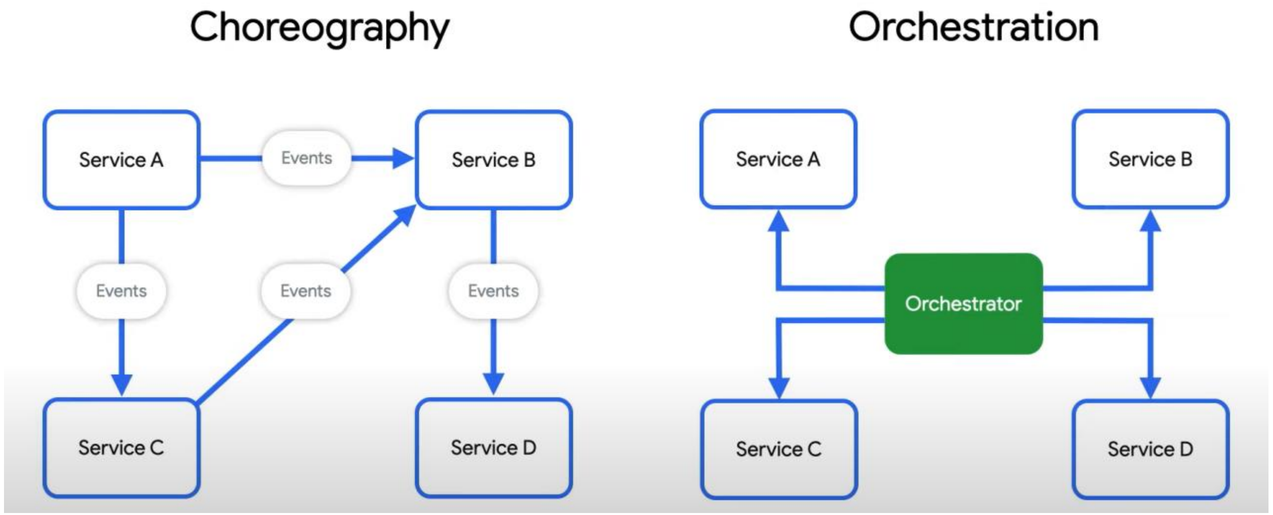
© Donald F. Ferguson, 2024
Columbia | Engineering
The Fu Foundation School of Engineering and Applied Science

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.

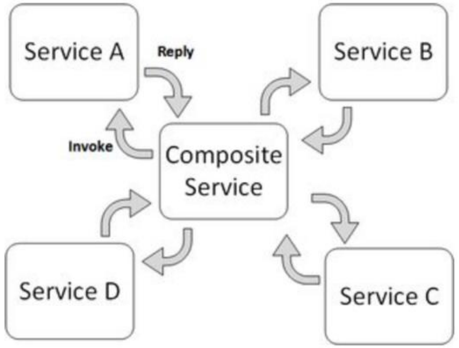
COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Concepts

Service orchestration

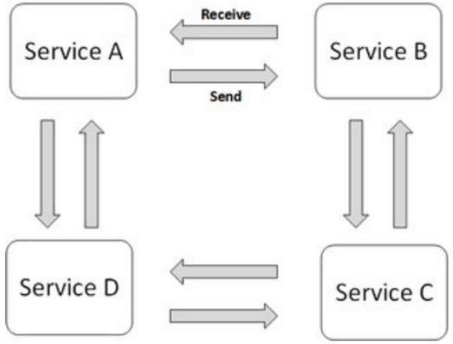
Service orchestration represents a single centralized executable business process (the orchestrator) that coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services.

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.

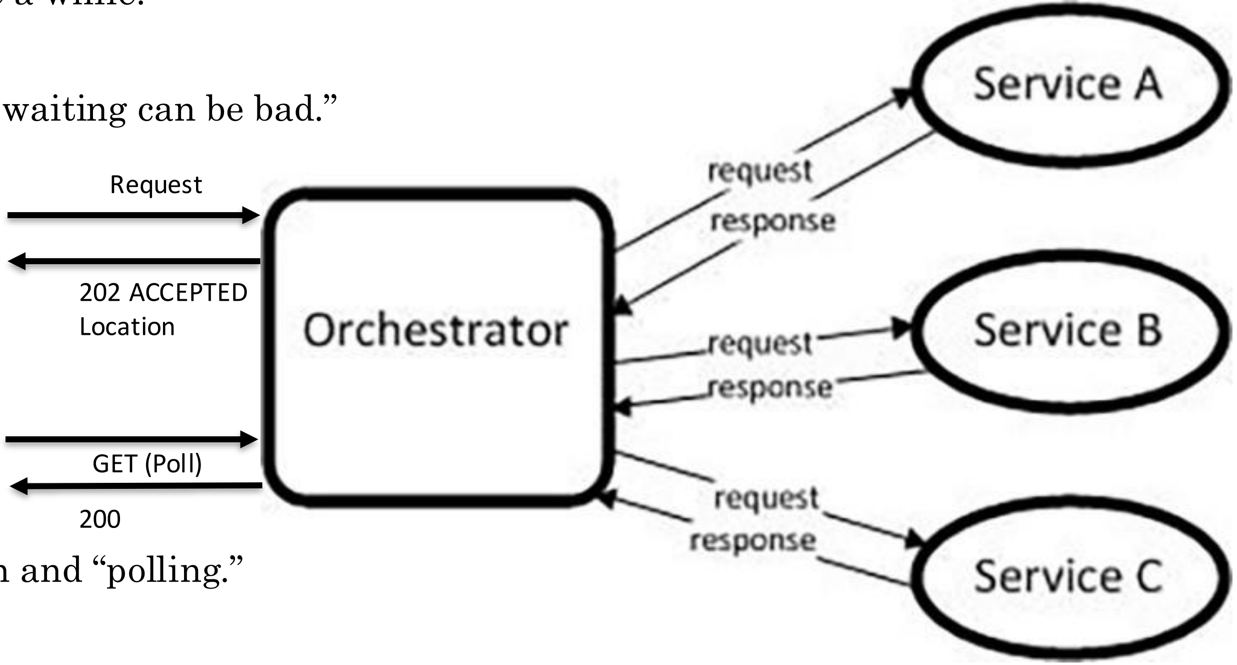
In Code: Composition and Asynchronous Method Invocation

- Composite operations can "take a while."

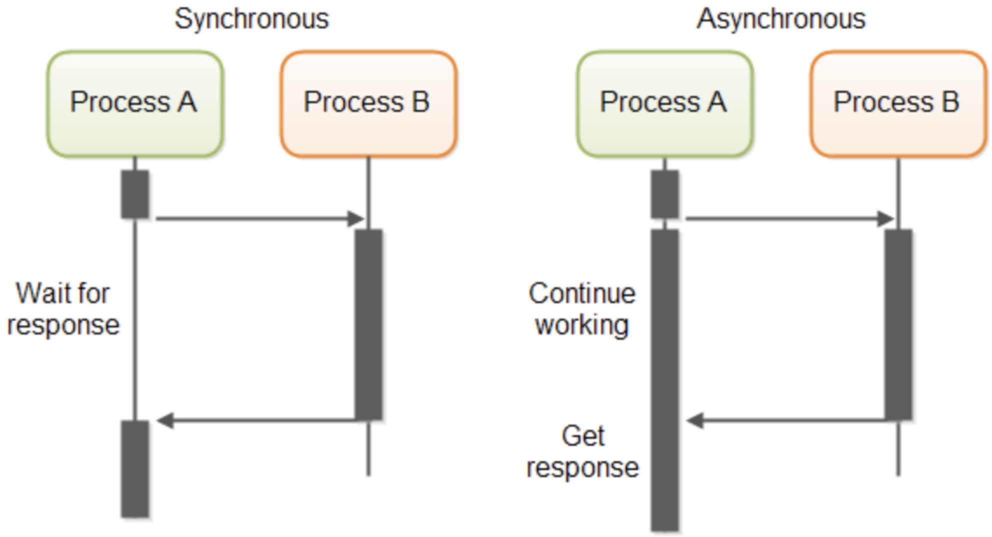
- "Holding open connections and waiting can be bad."
  - Consumes resources.
  - Connections can break.
  - … …

- 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

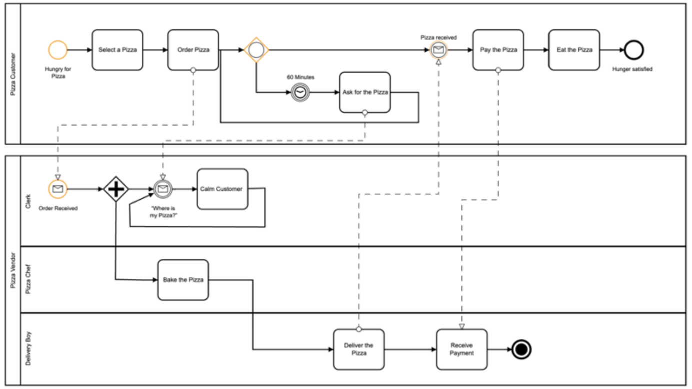


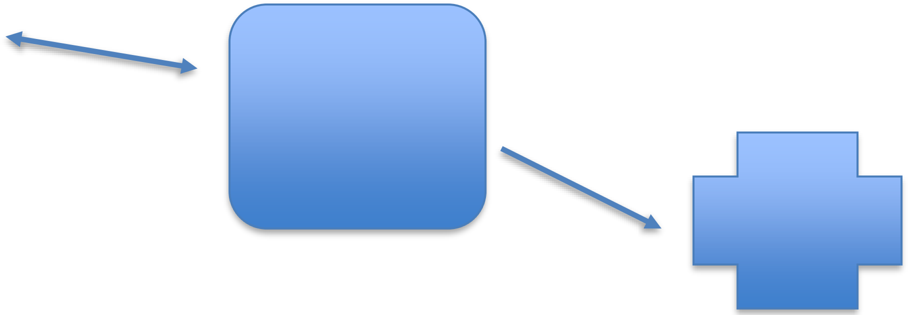
RESTFul HTTP calls can be implemented in both a synchronous and asynchronous fashion at an IO level.

- 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." expand in new window

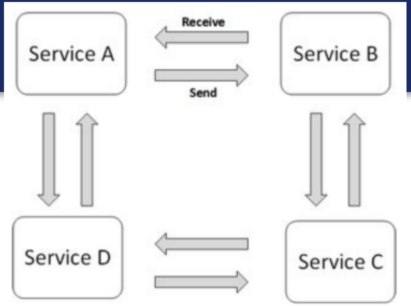




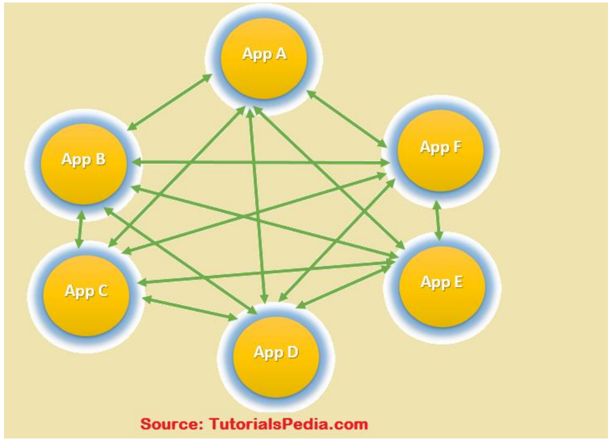
  - There are products for defining BPMN processes, generating code, executing, … …, e.g. Camunda: https://camunda.com/bpmn/
- There are other languages, tools, execution engines, … …

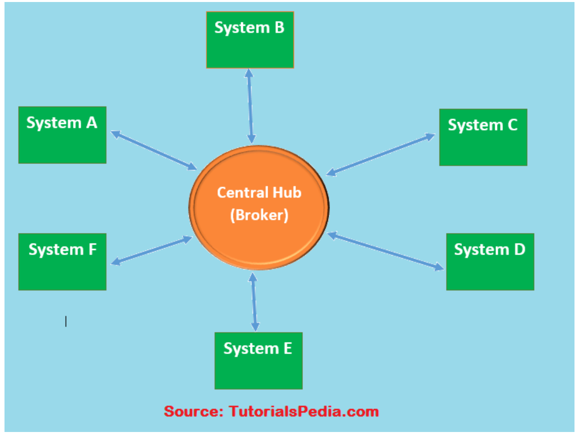
COLUMBIA | ENGINEERING The Fu Foundation School of Engineering and Applied Science

Choreography



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

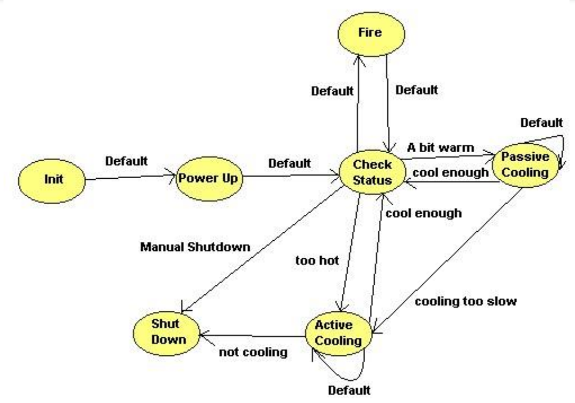


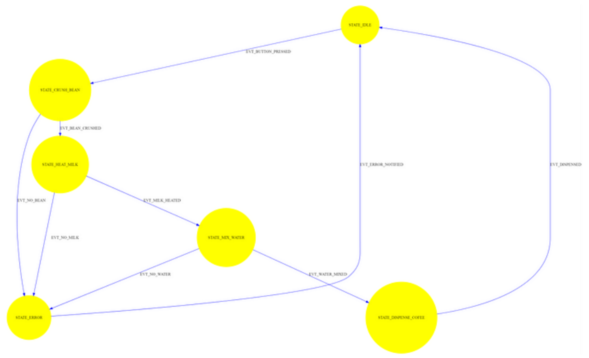


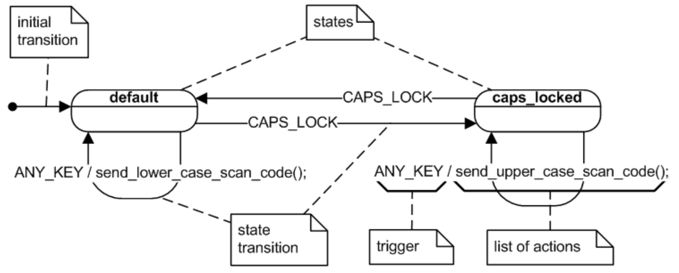
- But, how do you write the event driven microservices?
  - Well, you can just write code … …
  - Or use a state machines abstraction.

COLUMBIA ENGINEERING The Fu Foundation School of Engineering and Applied Science

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

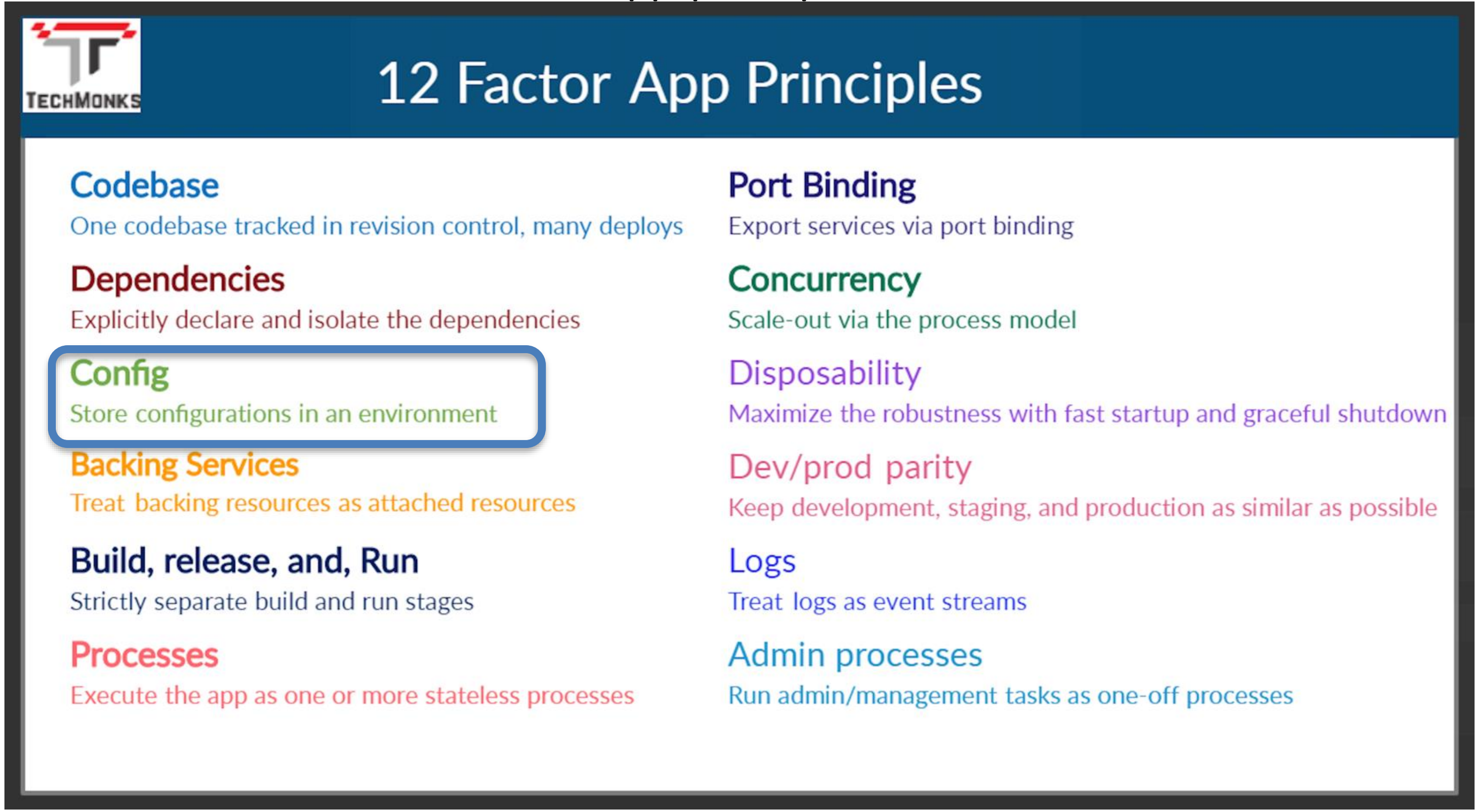
12 Factor Apps
SOLID

© Donald F. Ferguson, 2024

Columbia | Engineering
The Fu Foundation School of Engineering and Applied Science

12 Factor Applications

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



TECHMONKS

12 Factor App Principles

Codebase
One codebase tracked in revision control, many deploys

Dependencies
Explicitly declare and isolate the dependencies

Config
Store configurations in an environment

Backing Services
Treat backing resources as attached resources

Build, release, and, Run
Strictly separate build and run stages

Processes
Execute the app as one or more stateless processes

Port Binding
Export services via port binding

Concurrency
Scale-out via the process model

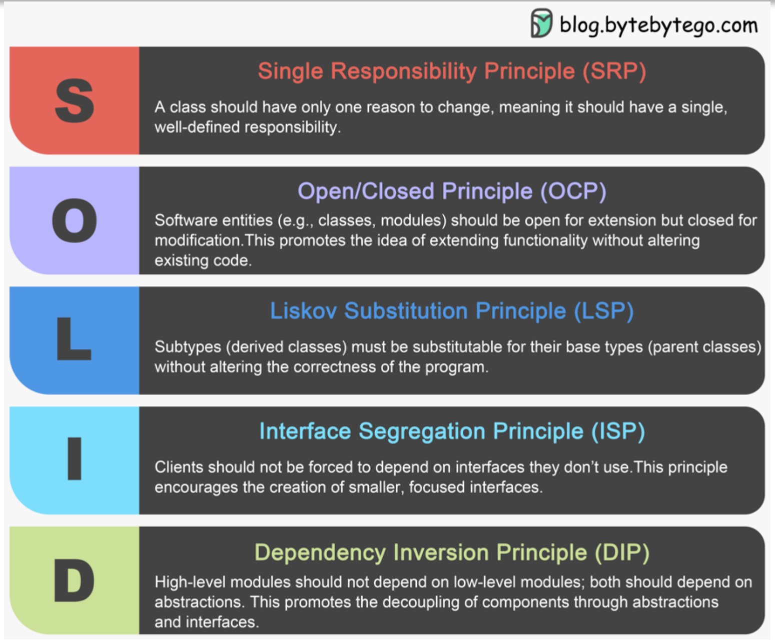
Disposability
Maximize the robustness with fast startup and graceful shutdown

Dev/prod parity
Keep development, staging, and production as similar as possible

Logs
Treat logs as event streams

Admin processes
Run admin/management tasks as one-off processes

SOLID Principles



blog.bytebytego.com

S — Single Responsibility Principle (SRP)

A class should have only one reason to change, meaning it should have a single, well-defined responsibility.

O — Open/Closed Principle (OCP)

Software entities (e.g., classes, modules) should be open for extension but closed for modification. This promotes the idea of extending functionality without altering existing code.

L — Liskov Substitution Principle (LSP)

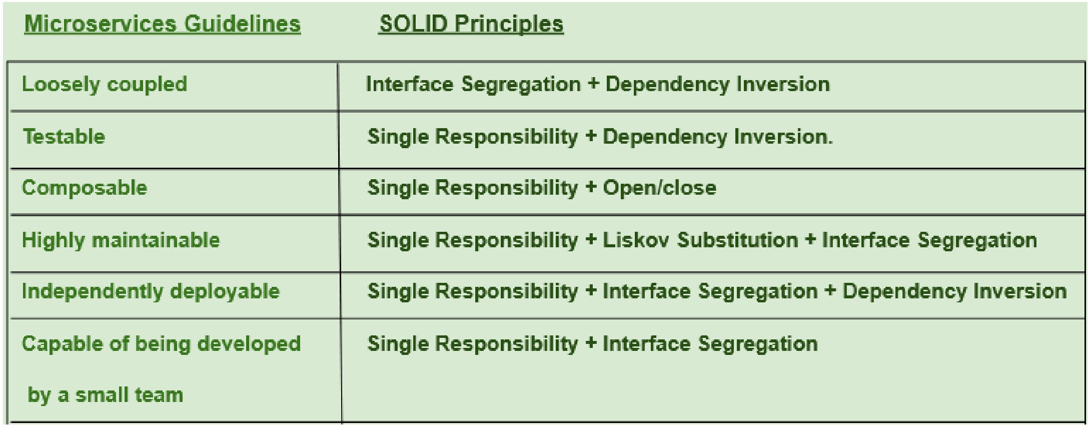
Subtypes (derived classes) must be substitutable for their base types (parent classes) without altering the correctness of the program.

I — Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they don't use. This principle encourages the creation of smaller, focused interfaces.

D — Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules; both should depend on abstractions. This promotes the decoupling of components through abstractions and interfaces.



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

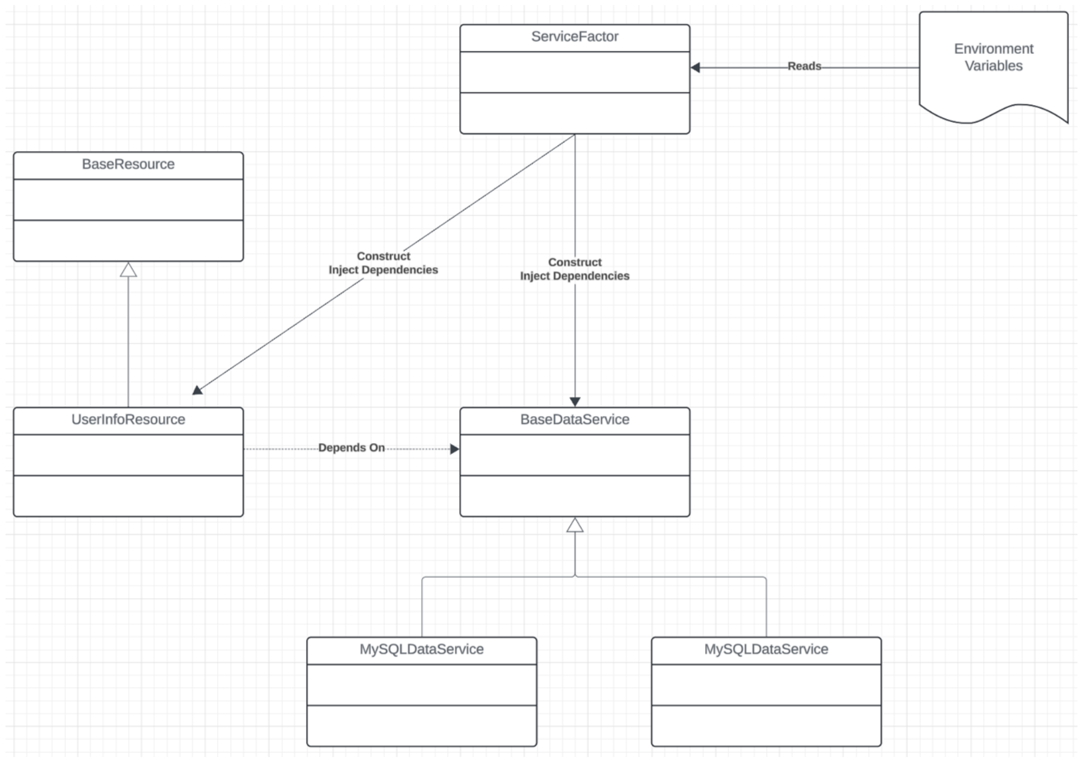
https://medium.com/@saurabh.engg.it/microservices-designing-effective-microservices-by-following-solid-design-principles-a995c3a033a0

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

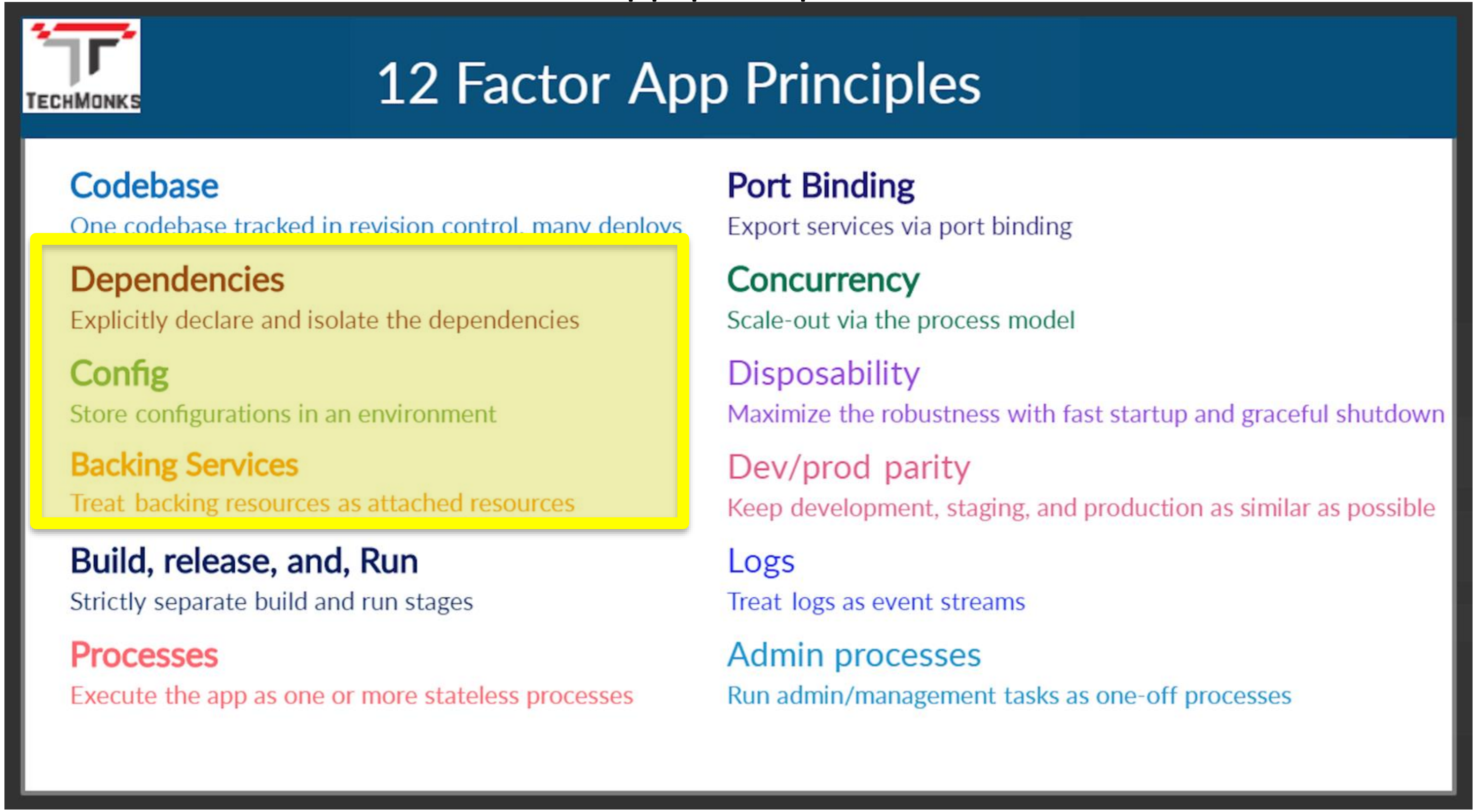
- Use interfaces, or the language equivalent/approximation.
- A constructor receive a "config" object that is a dictionary of
  - Interface name.
  - Reference to implementation.
- Parameters and configuration properties are also in the config, e.g.
  - URLs
  - "Passwords"
  - Database and table names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



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

12 Factor Applications

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



TECHMONKS

12 Factor App Principles

Codebase
One codebase tracked in revision control, many deploys

Dependencies
Explicitly declare and isolate the dependencies

Config
Store configurations in an environment

Backing Services
Treat backing resources as attached resources

Build, release, and, Run
Strictly separate build and run stages

Processes
Execute the app as one or more stateless processes

Port Binding
Export services via port binding

Concurrency
Scale-out via the process model

Disposability
Maximize the robustness with fast startup and graceful shutdown

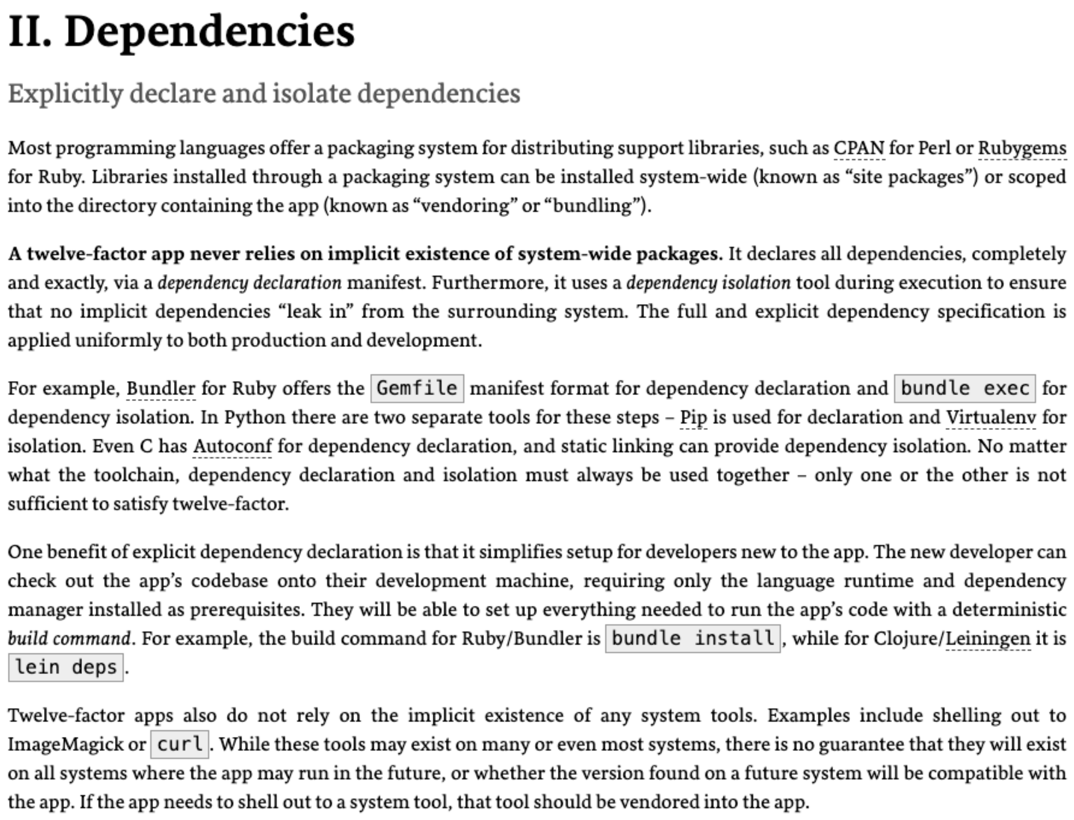
Dev/prod parity
Keep development, staging, and production as similar as possible

Logs
Treat logs as event streams

Admin processes
Run admin/management tasks as one-off processes

© Donald F. Ferguson, 2024

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.

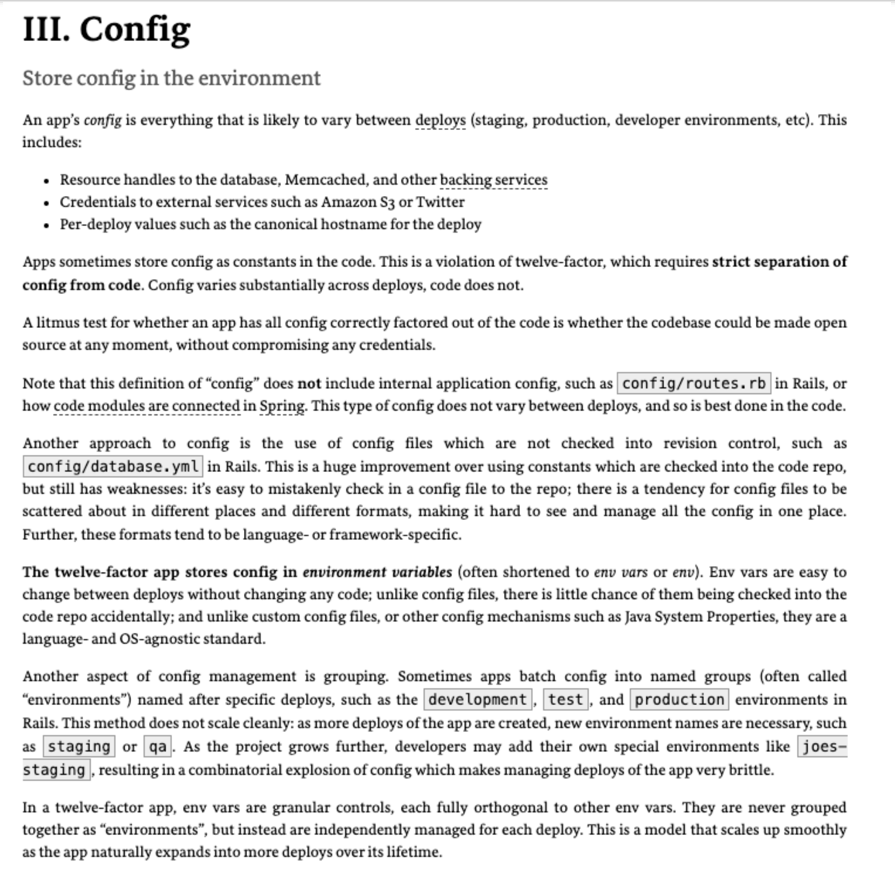
One benefit of explicit dependency declaration is that it simplifies setup for developers new to the app. The new developer can check out the app's codebase onto their development machine, requiring only the language runtime and dependency manager installed as prerequisites. They will be able to set 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.

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

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

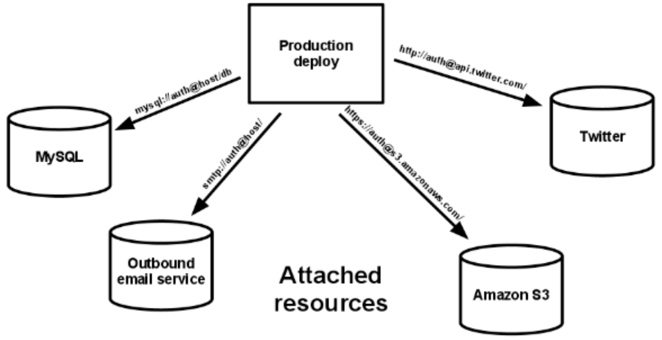
Treat backing services as attached resources

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).

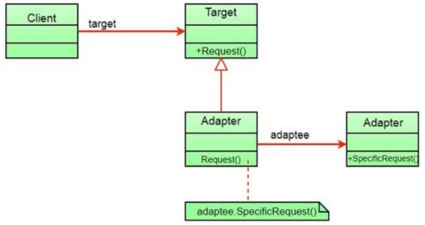
The code for a twelve-factor app makes no distinction between local and third party services. To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the config. A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed 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)

© Donald F. Ferguson, 2024
Columbia | Engineering
The Fu Foundation School of Engineering and Applied Science

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.

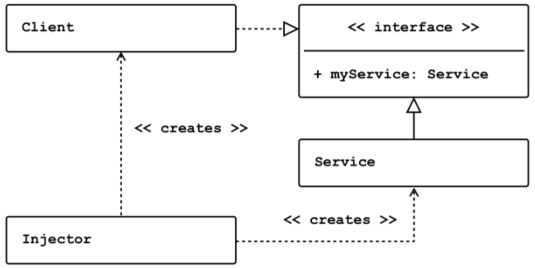
- Or actually, I asked ChatGPT to produce simple examples.

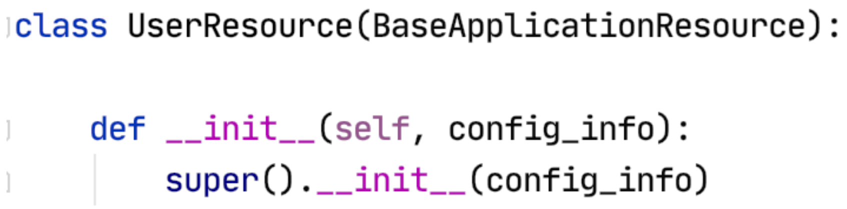
- And, these are conceptual, and I am not super happy with them.

© Donald F. Ferguson, 2024
Columbia | Engineering
The Fu Foundation School of Engineering and Applied Science

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.





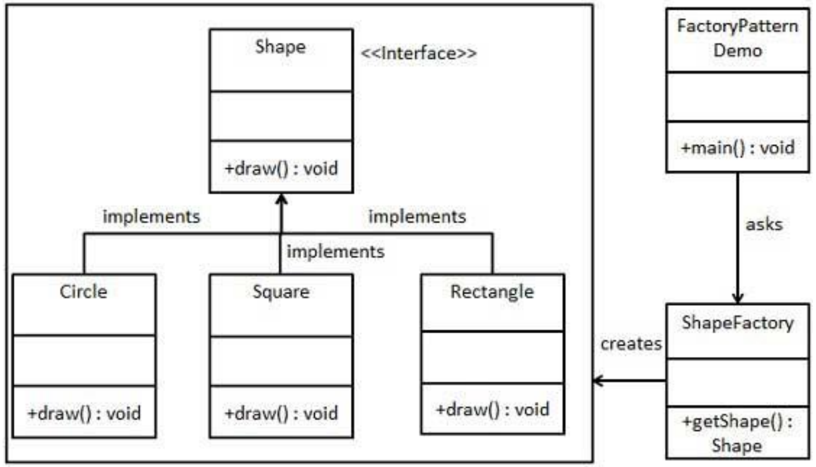
```
class UserResource(BaseApplicationResource):

    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.

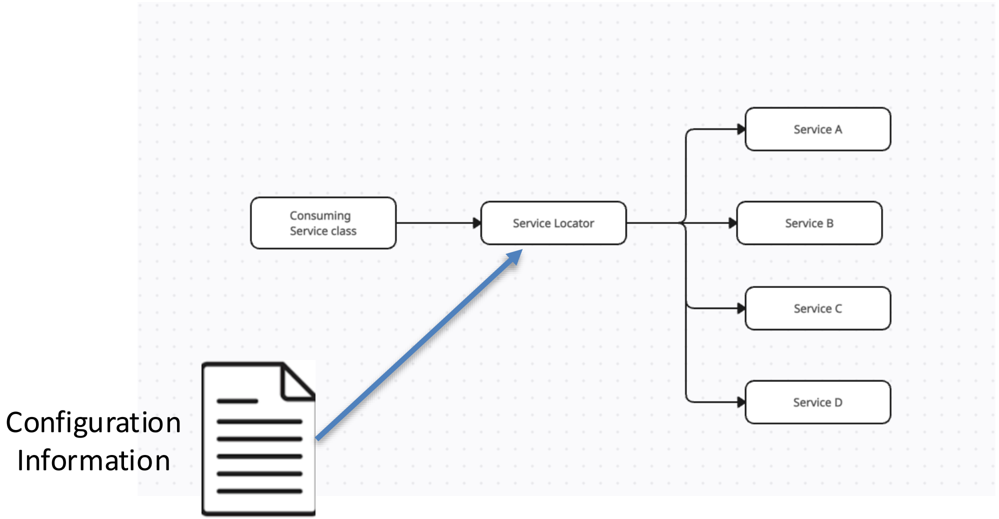


https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

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

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)



© Donald F. Ferguson, 2024

Where Are We Going With All Of This?

© Donald F. Ferguson, 2024

Diagrams

- Diagram the logical model of the first application
- Talk about deployment options.

© Donald F. Ferguson, 2024

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