Welcome

API Design & Development

Hello

About me...

- 27+ years in the industry
- 23+ years in teaching
- Certified Cloud architect
- Passionate about learning
- Also, passionate about Reese's Cups!

We teach over 400 technology topics

You experience our impact on a daily basis!

My pledge to you

I will...

- Make this interactive
- Ask you questions
- Ensure everyone can speak
- Use an on-screen timer

Agenda

- Architecting APIs
- Building APIs
- Securing APIs
- Testing APIs
- Operating & Maintaining APIs

How we’re going to work together

- Slides / lecture
- Demos
- Team discussions
- Labs
- Hack-a-thon

Architecting APIs

Architecting for the Future

SOLID Principles

SOLID principles help us build testable code

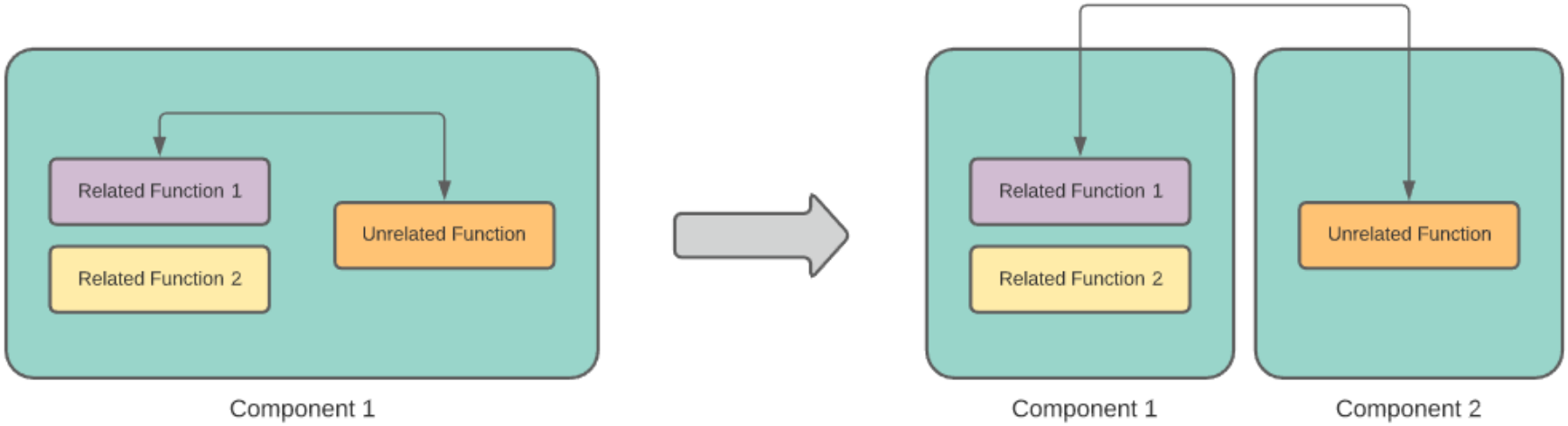
- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

A system module or component should have only one reason to change

Single Responsibility Principle (SRP)

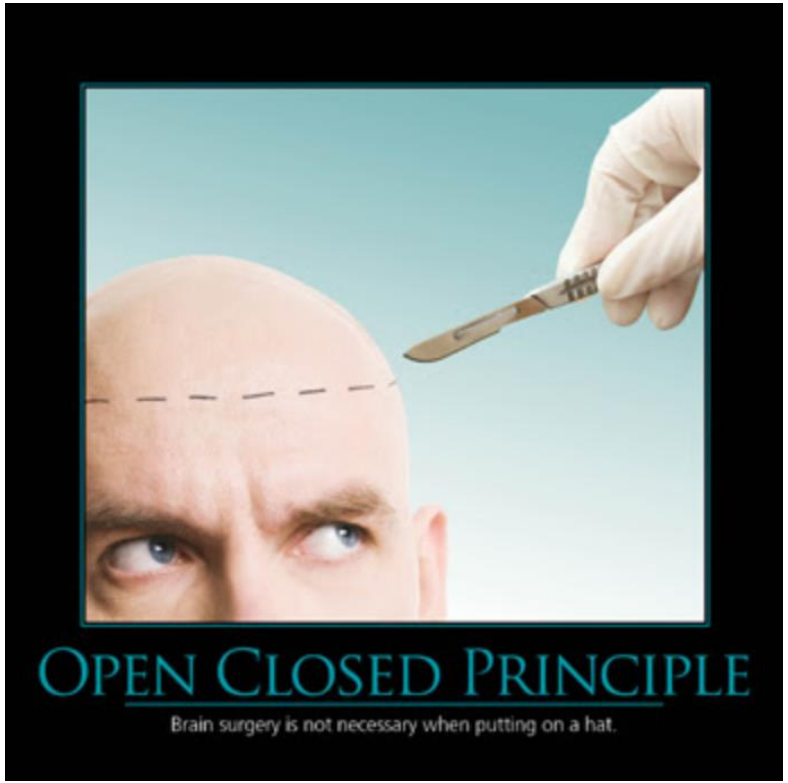
Single Responsibility Principle (SRP)



Open-Closed Principle (OCP)

Software entities should be open for extension but closed for modification

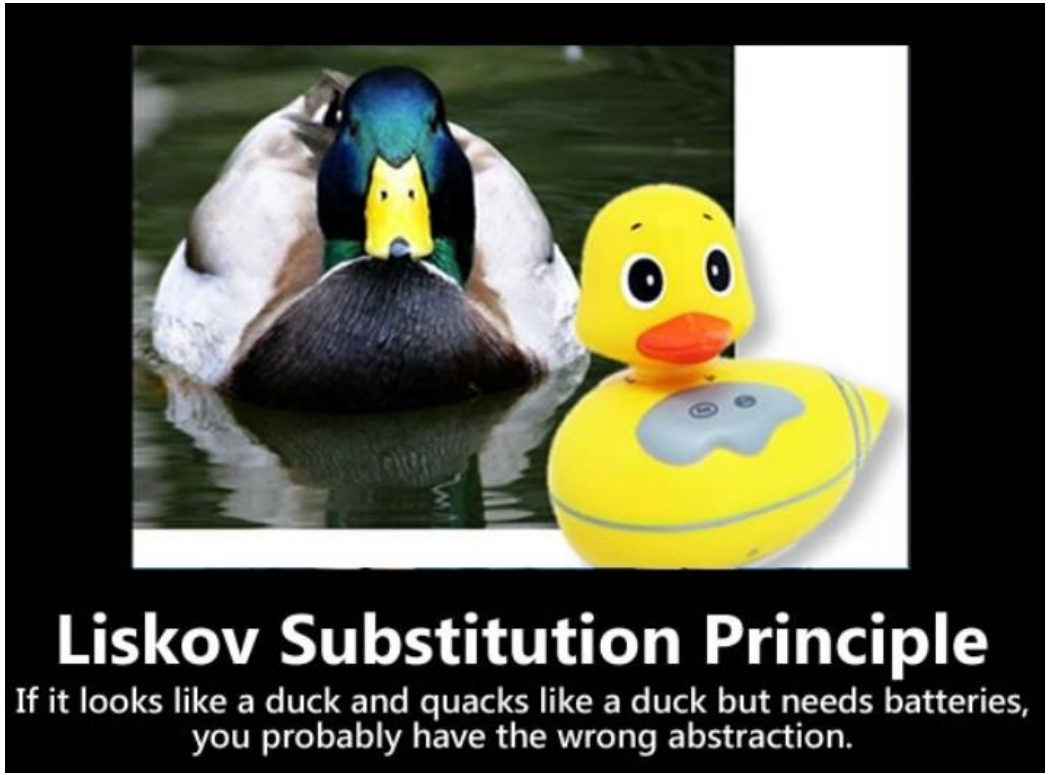
Open-Closed Principle (OCP)



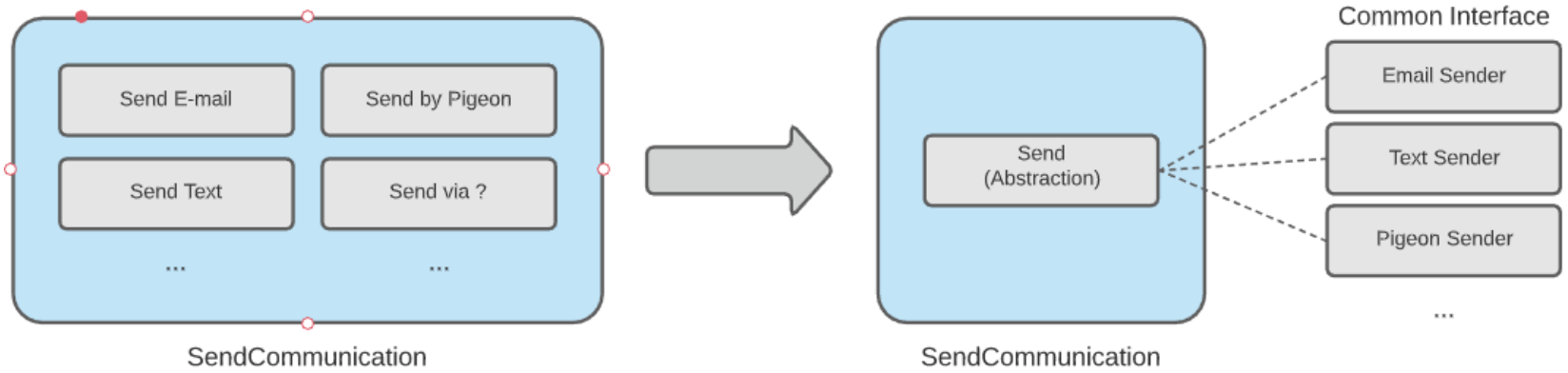
Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types

Liskov Substitution Principle (LSP)



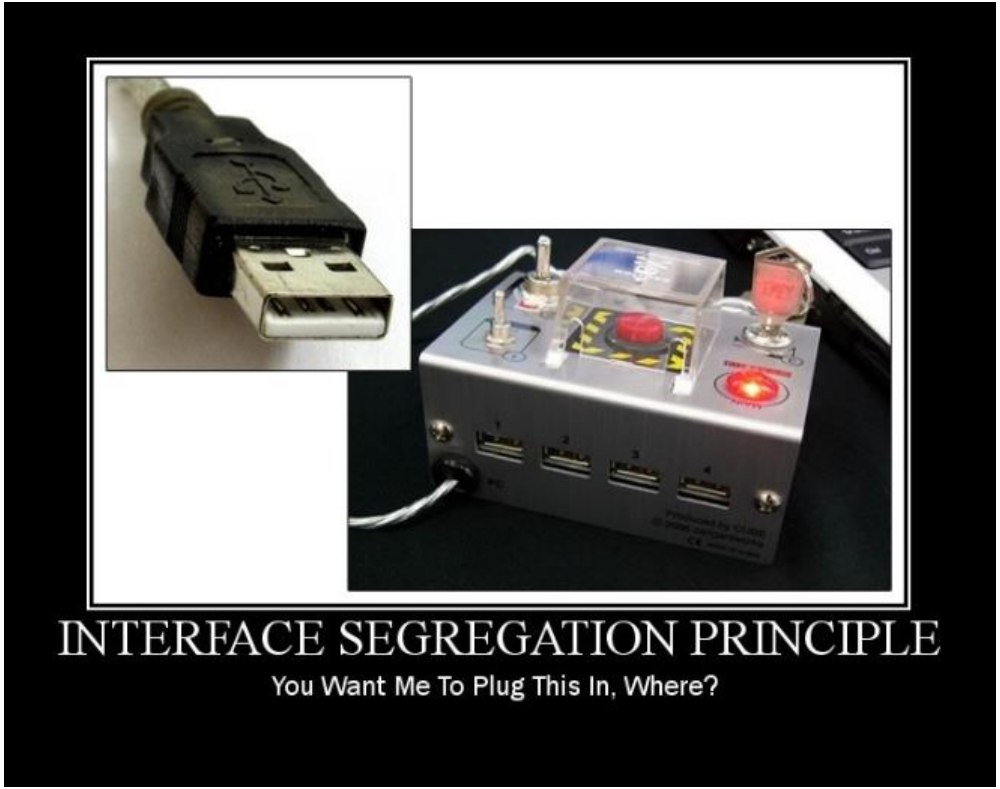
OCP & LSP


Send E-mail
Send by Pigeon
Send Text
Send via ?
...
...
SendCommunication
Send
(Abstraction)
Common Interface
Email Sender
Text Sender
Pigeon Sender
...
SendCommunication

Interface Segregation Principle (ISP)

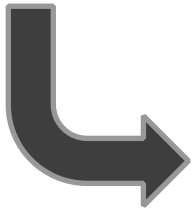
Clients should not be forced to depend on methods they do not use

Interface Segregation Principle (ISP)



Interface Segregation Principle (ISP)

```
7   namespace CoolCompany.Financials
8   {
9       public interface ITaxProcessor
10      {
11          double CalculateSalesTax(double onAmount);
12          double CalculatePropertyTax(double onAmount);
13          double CalculateIncomeTax(double onAmount);
14      }
15  }
```



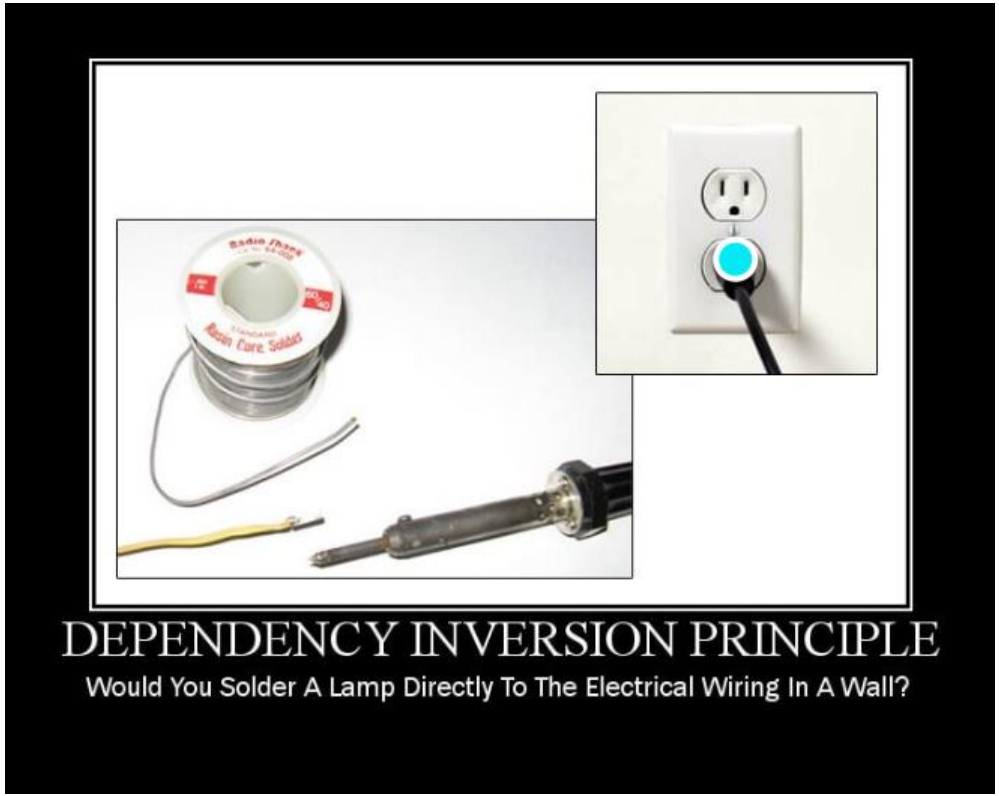
```
7   namespace CoolCompany.Financials
8   {
9       public interface ITaxProcessor
10      {
11          double Calculate(double onAmount);
12      }
13
14      public interface ISalesTaxProcessor : ITaxProcessor
15      {
16          double LookupStateTaxRate(string state);
17      }
18
19      public interface IPropertyTaxProcessor : ITaxProcessor
20      {
21          double LookupTownshipTaxRate(string township);
22      }
23
24      public interface IIncomeTaxProcessor : ITaxProcessor
25      {
26          double CalculateBackTaxes(string taxpayerId);
27      }
28  }
```

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules – both should depend on abstractions

Abstractions should not depend upon details – details should depend upon abstractions

Dependency Inversion Principle (DIP)



Architecting for the Cloud

Application Hosting

By Application Hosting, we mean the target infrastructure and runtime platform used for deployment and execution of an application or system; can include compute (CPU and server resources), storage, network, data and operating system

Application Hosting – An “Interesting” Example?

Here’s an example of someone thinking “outside-of-the-box” when it comes to application hosting!

https://mashable.com/article/pregnancy-test-doom/

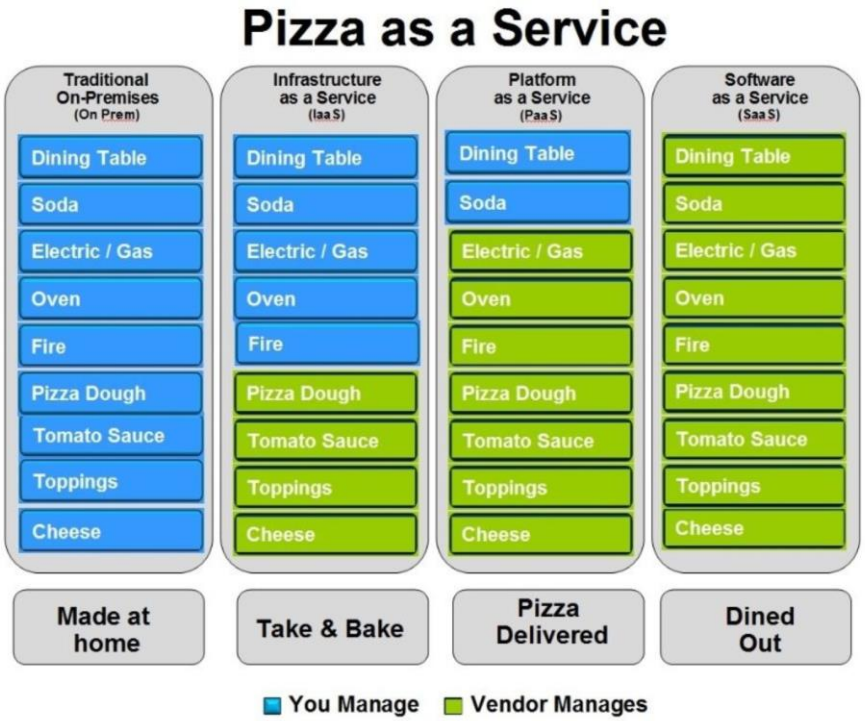
What Are the Hosting Options with Cloud?

- IaaS
- PaaS
- Serverless / FaaS
- SaaS
- Containers

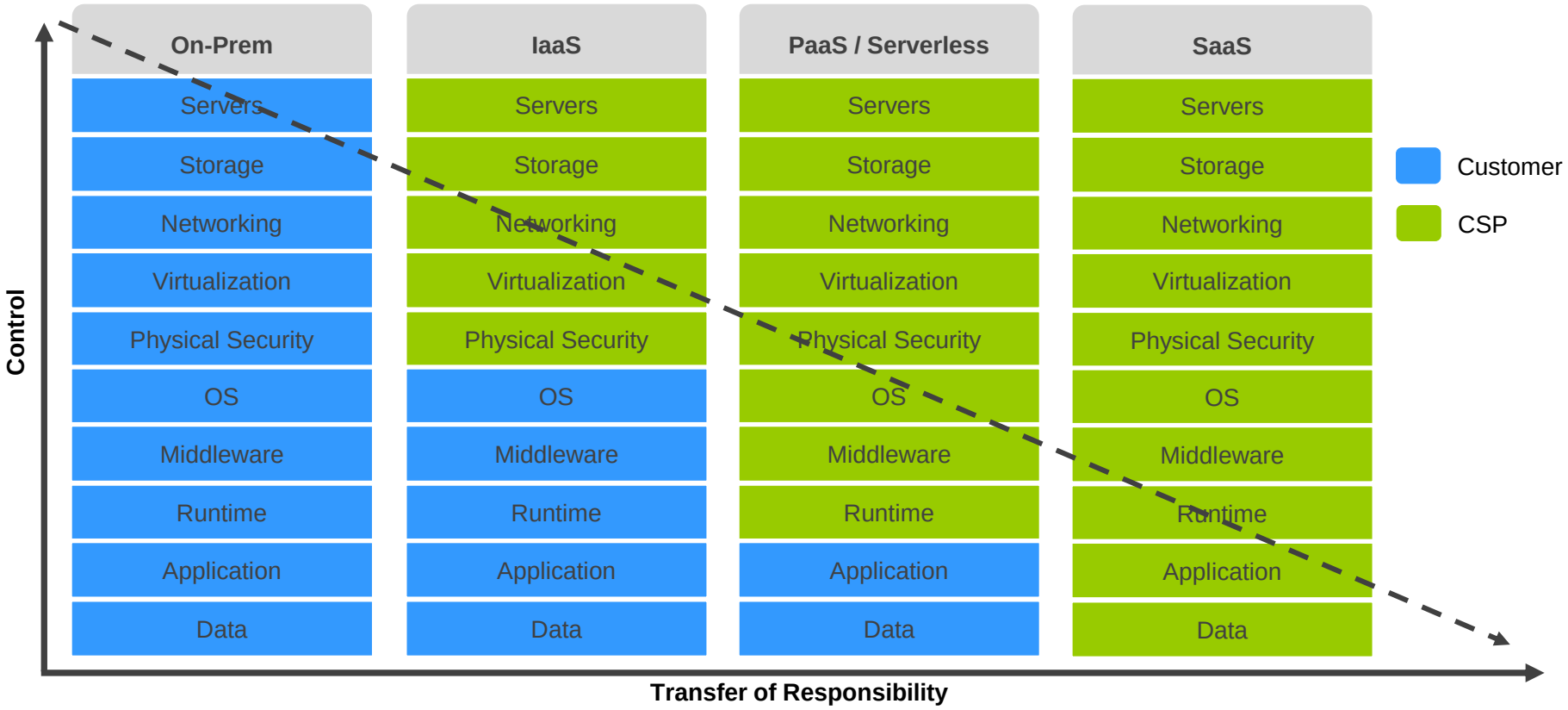
What do they all mean?

Pizza-as-a-Service

From a LinkedIn post by Albert Barron from IBM (https://www.linkedin.com/pulse/20140730172610-9679881-pizza-as-a-service/**)**



Side-by-Side Comparison



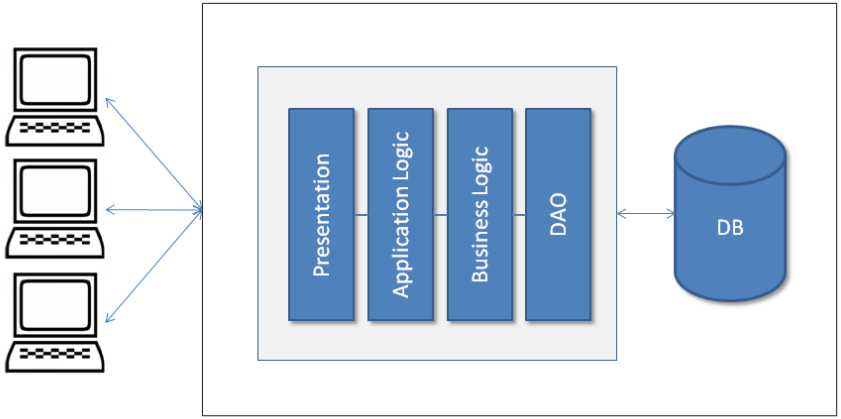
Monolith vs. Microservices

Discussion: Monolith vs. Microservices

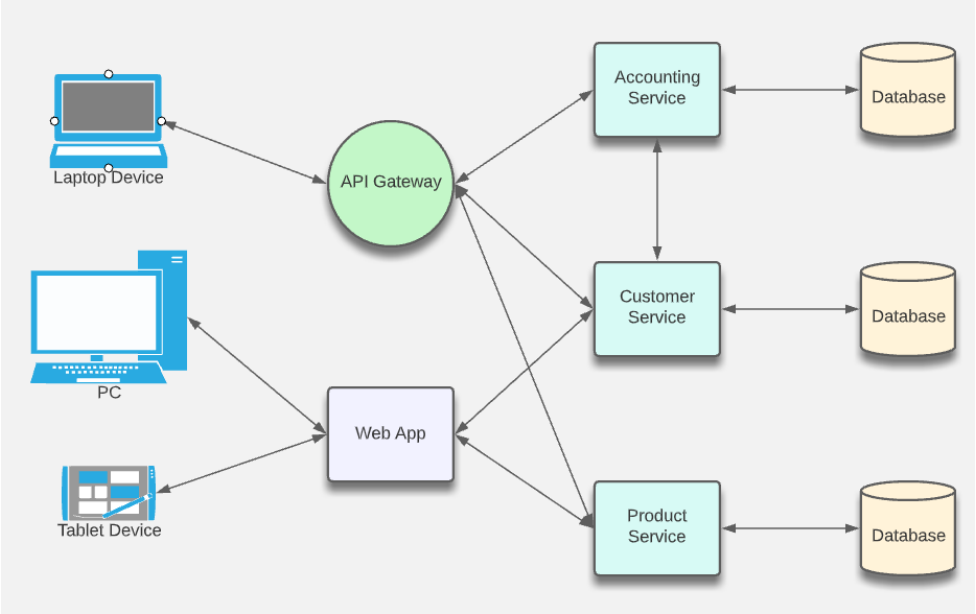
- What is a “monolith”?
- What is a “microservice”?
- What are some of the benefits of a “microservice”?
- What are some implications with a “microservice” architecture?

What is a Monolith?

- Typical enterprise application
- Large codebases
- Set technology stack
- Highly coupled elements
- Whole system affected by failures
- Scaling requires the duplication of the entire app
- Minor changes often require full rebuild



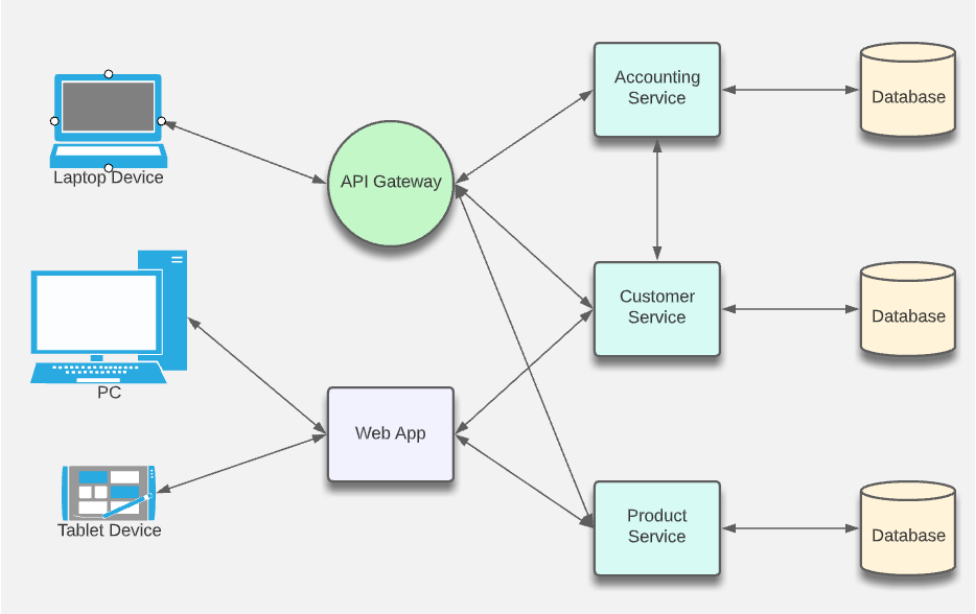
Microservices Architecture



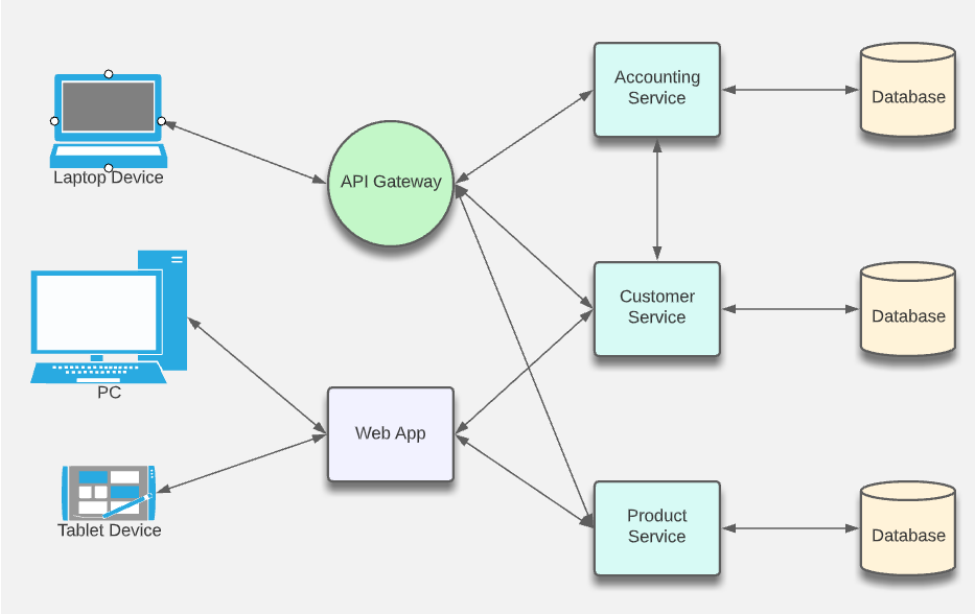
Microservices Architecture

Benefits

- Enables work in parallel
- Promotes organization according to business domain
- Advantages from isolation
- Flexible in terms of deployment and scale
- Flexible in terms of technology
- Allows multiple agile teams to maximize autonomy in effort and technology



Microservices Architecture



Implications

- Requires a different way of thinking
- Complexity moves to the integration layer
- Organization needs to be able to support re-org according to business domain (instead of technology domain)
- With an increased reliance on the network, you may encounter latency and failures at the network layer
- Transactions must be handled differently (across service boundaries)

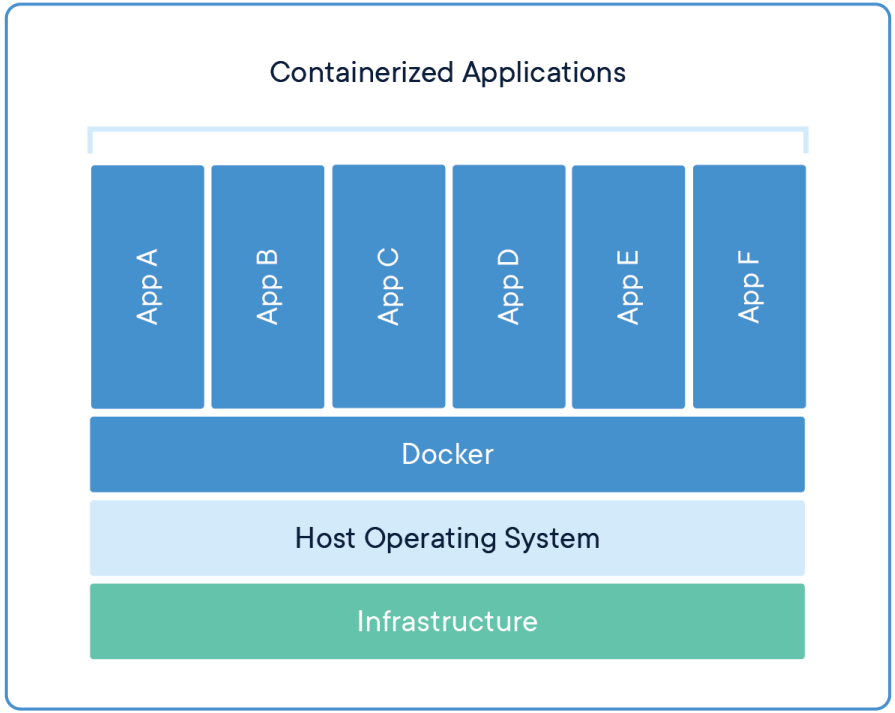
So, What Are Containers?

- Form of virtualization at the app packaging level (like virtual machines at the server level)
- Isolated from one another at the OS process layer (vs VM’s which are isolated at the hardware abstraction layer)
- Images represent the packaging up of an application and its dependencies as a complete, deployable unit of execution (code, runtime and configuration)

So, What Are Containers?

- A platform (e.g., Docker) running on a system can be used to dynamically create containers (executable instances of the app) from the defined image
- Typically, much, much smaller than a VM which makes them lightweight, quickly deployable and quick to “boot up”
- An orchestration engine (e.g., Kubernetes) might be used to coordinate multiple instances of the same container (or a “pod” of containers) to enable the servicing of more concurrent requests (scalability)

So, What Are Containers?



So, Why Containers?

- Abstracts away the code implementation so you can deploy in a platform-agnostic manner, writing in the language of your choice
- Aligns strongly with the principles and practices of DevOps
- Helps leverage the power of the cloud
- Speeds up important non-coding activities (infrastructure spin-up, testing, CI/CD tasks, DevSecOps, code quality checks, etc.
- Helps breed consistency vs. “snowflake”

So, How Do Containers & Microservices Fit Together?

- Microservices – with their smaller size, independently-deployable and independently-scalable profile, and encapsulated business domain boundary – are a great fit for containers
- Using Kubernetes, sophisticated systems of integrated microservices can be built, tested and deployed
- Leveraging the scheduling and scalability benefits of Kubernetes can help an organization target scaling across a complex workflow in very granular ways
- This helps with cost management as you can toggle individual parts of the system for optimized performance

So, How Do Containers & the Cloud Fit Together?

- One option includes standing up VM's (IaaS) and installing / managing a Kubernetes cluster on those machines or
- Another option includes leveraging a managed service (PaaS) provided by the CSP
- Options in AWS include Elastic Container Service (ECS), Elastic Container Registry (ECR), and EKS (Elastic Kubernetes Service)

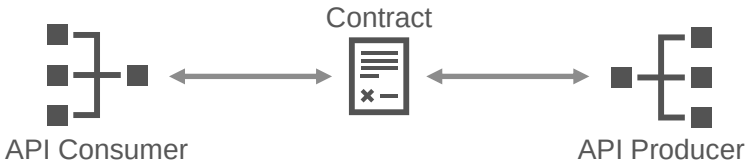
Where Microservices & APIs Overlap

What is an API?

- Stands for Application Programming Interface
- Describes the public interface exposed by one or more components
- Enables the reuse of functionality across multiple workflows
- Allows us to break a larger problem into smaller pieces
- Often “bound” by the specific technology used for connectivity and interaction

APIs can be implemented with or without a microservices architecture

Where Microservices & APIs Overlap



APIs:

- Model the exchange of information between a consumer & producer
- Support business integration between two internal systems (“in process”) or between an internal system & an external partner (“out of process”)
- Built around technology specifications for interaction & type definition
- Enable integration over the network/Internet

Demo: System of APIs

Reference Application: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/microservice-application-design

Repo: https://github.com/dotnet-architecture/eShopOnContainers

Domain-Driven Design (DDD)

What is DDD?

- Process of designing systems aligned to business domain
- Drives business language into the technology – engineer in terms of business process & impact
- Can enable IT to be an enabler and even an accelerator (vs. another “cost center”)
- Helps with management of types of coupling
- Direct correlation can be drawn between domains and microservices

Types of Coupling

- Implementation
- Temporal
- Deployment
- Domain
- Coupling can increase “blast radius” for a change
- Can also limit autonomy and constrain innovation

Defining Boundaries

- Enforcing strong boundaries and controlled interfaces can help mitigate coupling
- Domains become realms of cohesion
- Controlled integration – i.e., what do I allow others to “see”?
- DDD is about helping us determine where the boundaries should exist

Some Key Terms

Bounded Context

- Provides a place for a set of terms to have a specific business meaning
- Contexts allow for variance in technology from domain-to-domain
- An entity may exist with the same name in two different bounded contexts, but its context will dictate its meaning

Some Key Terms

Ubiquitous Language

- Shared language between tech and business
- Using language that models the business
- Promotes collaboration between domain experts and tech

Some Key Terms

Domain

- A “slice” of your software system
- Ideally, maps to a bounded context (and potentially one or more microservices)
- Can encapsulate subdomains
- Three major types: Generic, Supporting, and Core

Some Key Terms

Generic Domain – “Buy vs. Build”, SaaS, use 3rd party or framework

Supporting Domain – Develop, maybe outsource

Core – Unique and distinguishing; where lion’s share of efforts are focused

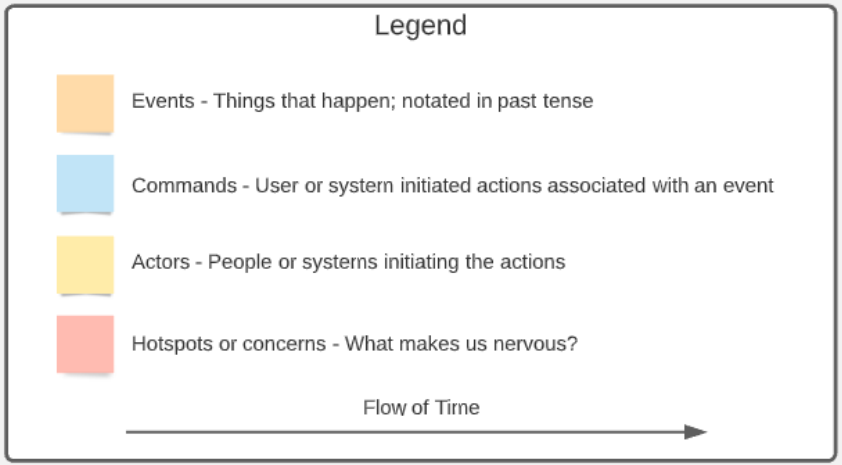
EventStorming

- One DDD strategy that can be used to “crowdsource” an understanding of the business process and domain
- Team works together to define the business process in terms of “events” that happen (past tense)
- Those events are sequenced across a timeline to help document a workflow
- Able to associate additional elements that help inform the business process – e.g., actors, inputs, outputs, and commands

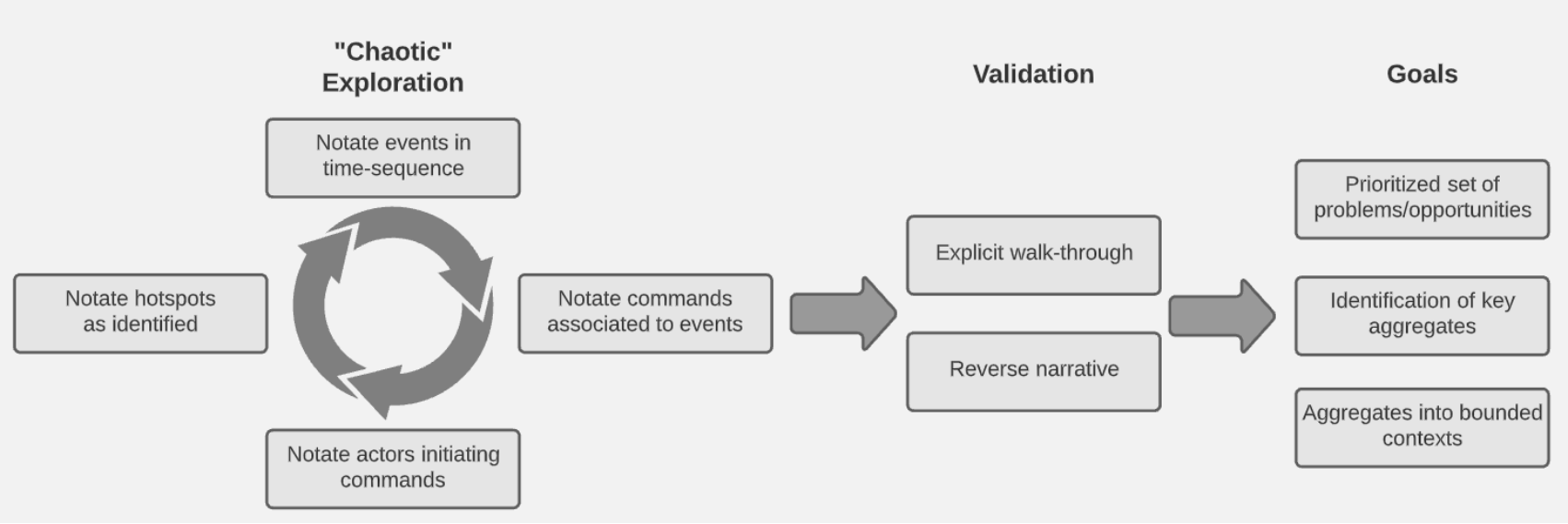
EventStorming

- Best served by an exercise with a group of people in the same physical location
- However, also able to complete via remote team (using Miro or LucidChart) – see COVID-19
- Through the exercise, team gains a shared understanding of the workflow from the business perspective
- Can also use to help identify key “aggregates” in the workflow as well as start to segment the domain by bounded context

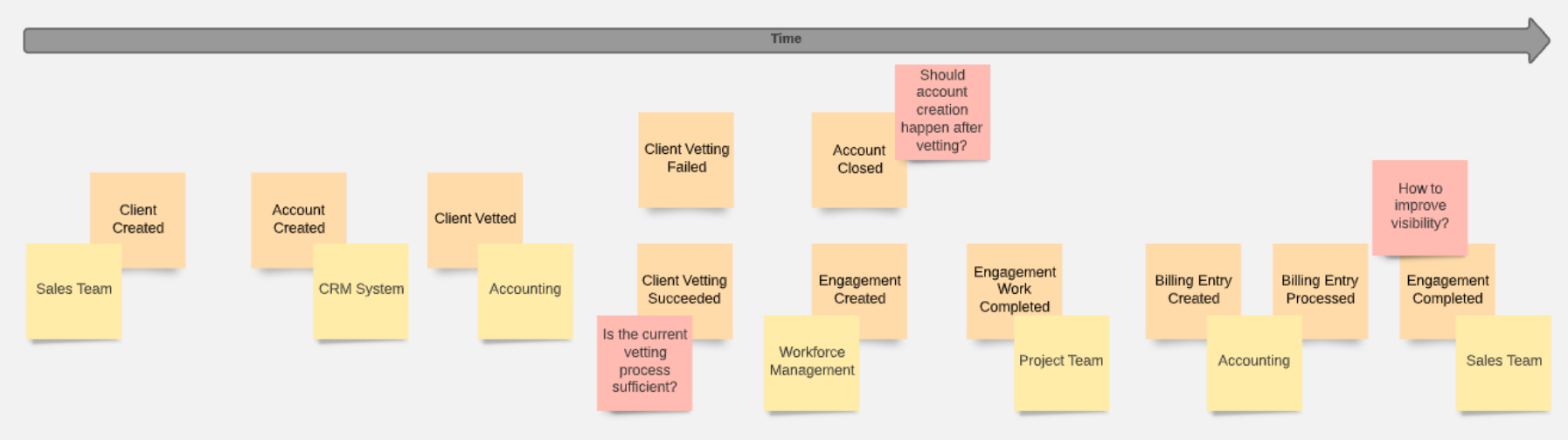
EventStorming



EventStorming



EventStorming - Example



LAB:

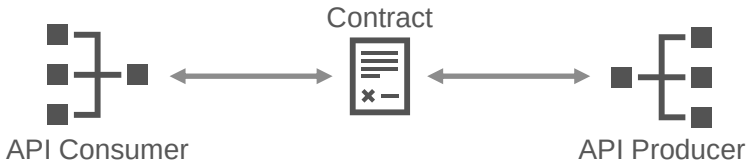
Where Are the Seams?

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/labs/lab01

Building APIs

Defining the Contract

Defining the Contract



- Describes an agreement between producer and consumer
- Defines operations available, inputs required, and expected results (structure, names, types, constraints)
- Can be used to test on either side in isolation (as long as adhered to)
- OpenAPI specification is a common standard used to define

Defining the Contract

Basic Structure: https://swagger.io/docs/specification/basic-structure/

LAB:

Baseline Orders API

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/labs/lab02

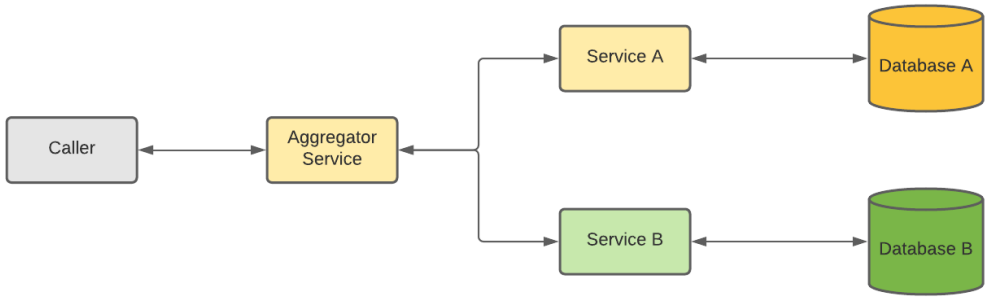
Microservice / API Design Patterns

Microservice / API Design Patterns

- Aggregator
- API Gateway
- Chain of Responsibility
- Asynchronous Messaging
- Circuit Breaker
- Anti-Corruption Layer
- Strangler Application
- Others as well (https://microservices.io/patterns/index.html)

Microservice / API Design Patterns - Aggregator

- Akin to a web page invoking multiple microservices and displaying results of all on single page
- In the pattern, one microservice manages calls to others and aggregates results for return to caller
- Can include update as well as retrieval ops



Microservice / API Design Patterns - Aggregator

Benefits:

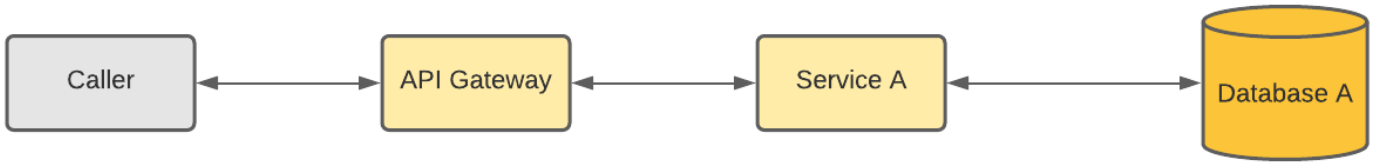
- Helps you practice DRY (Don’t Repeat Yourself)
- Supports logic at time of aggregation for additional processing

Potential “gotchas”:

- Performance if downstream microservices not called asynchronously

Microservice / API Design Patterns – API Gateway

- Specific type of infrastructure that helps manage the boundary
- Provides an intermediary for routing calls to a downstream microservice
- Provides a protection and translation layer (if calling protocol different from microservice)
- Can be combined with other patterns as well
- Considered a best practice from a security perspective as well



Microservice / API Design Patterns – API Gateway

Benefits:

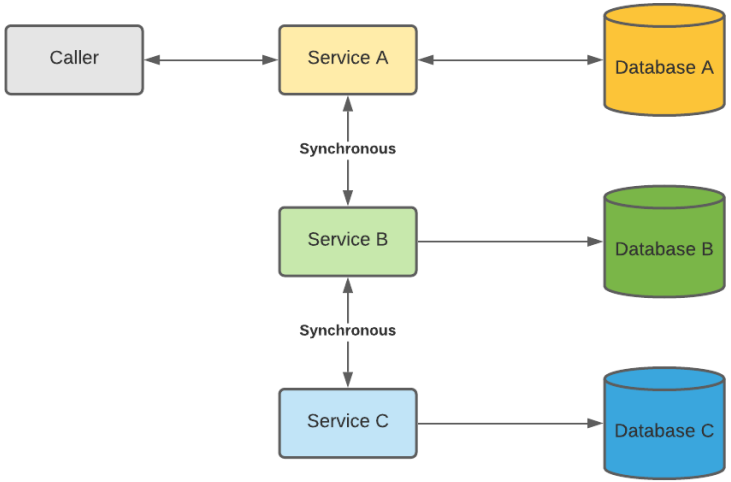
- Usually, built in throttling (to protect against DDoS)
- Can support translation between caller and downstream microservice for protocol, message format, etc.
- Provides a centralized point of entry for AuthN/AuthZ
- Can be combined with load balancing to enable scalability and resiliency
- Centralized logging

Potential “gotchas”:

- Can increase cost
- Requires additional configuration & management but enables additional capabilities as well
- Fortunately, frameworks can help simplify and hide complexity

Microservice / API Design Patterns – Chain of Responsibility

- Represents a chained set of microservice calls to complete a workflow action
- Output of 1 microservice is input to the next
- Uses synchronous calls for routing through chain



Microservice / API Design Patterns – Chain of Responsibility

Benefits:

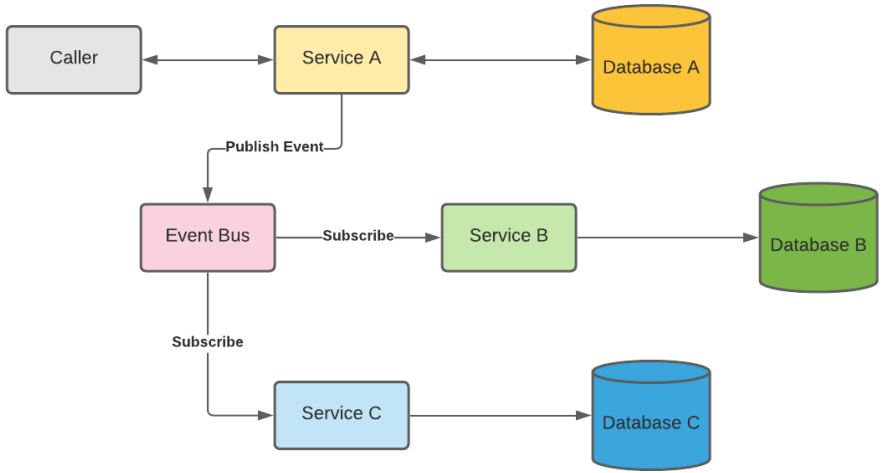
- Supports composition of microservices in a workflow that must happen in sequence
- Synchronous communication typically less complex

Potential “gotchas”:

- Increased response time – response time becomes the sum of each microservice’s response time in the chain

Microservice / API Design Patterns – Asynchronous Messaging

- Supports asynchronous interaction between independent microservices
- Messages published to a topic by a producer
- Topic subscribed to by one or more consumers



Microservice / API Design Patterns – Asynchronous Messaging

Benefits:

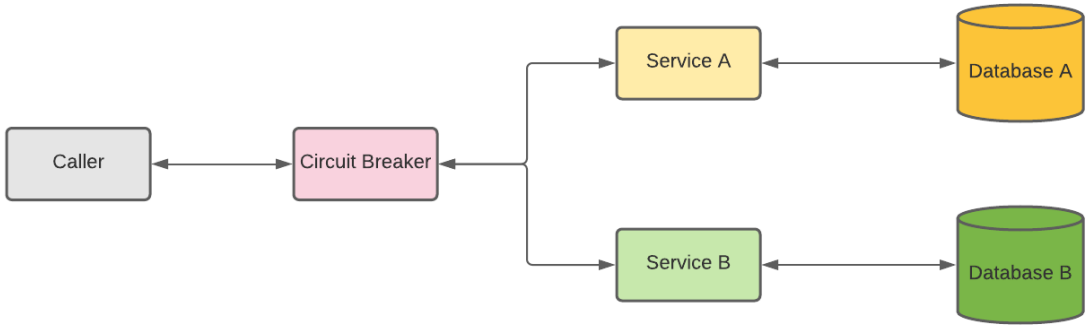
- Promotes looser coupling in cases where synchronous calls are not required
- Allows multiple services the option of being notified (vs. point-to-point)
- If given consumer is down, producer can continue to send messages and they won't be lost

Potential "gotchas":

- Additional complexity
- Can be harder to trace an action end-to-end (but there are ways to handle)
- Not a good fit if specific timing and sequencing required

Microservice / API Design Patterns – Circuit Breaker

- Prevents unnecessary calls to microservices if down
- Circuit breaker monitors failures of downstream microservices
- If number of failures crosses a threshold, circuit breaker prevents any new calls temporarily
- After defined time period, sends a smaller set of requests and ramps back up if successful



Microservice / API Design Patterns – Circuit Breaker

Benefits:

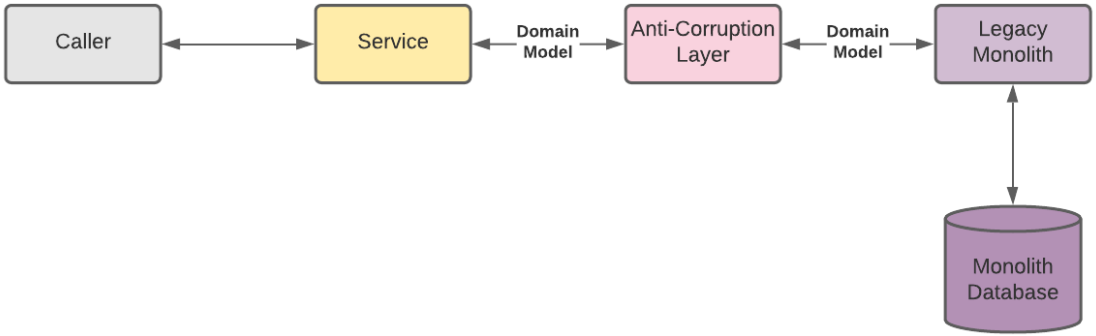
- Helps to optimize network traffic by preventing calls that are going to fail
- Can help prevent “noise” and network overload

Potential “gotchas”:

- Process supported needs to be able to absorb limited downtime
- If not managed correctly, can result in poor user experience

Microservice / API Design Patterns – Anti-Corruption Layer

- Prevents corruption of new microservice domain model(s) with legacy monolith domain model(s)
- Provides a proxy/translation layer to help keep models “clean”



Microservice / API Design Patterns – Anti-Corruption Layer

Benefits:

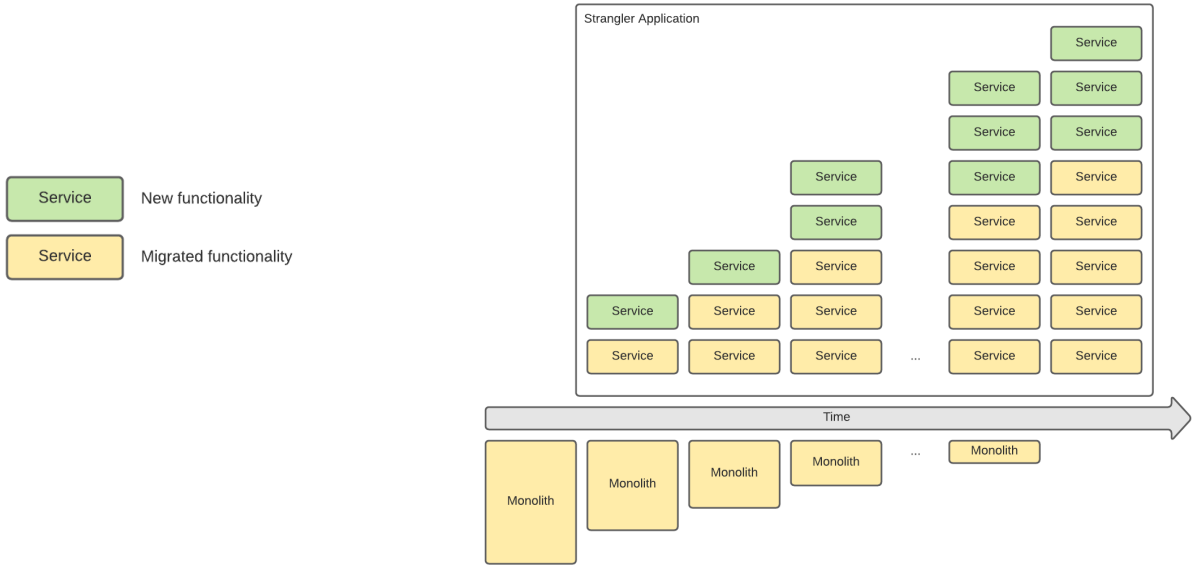
- Prevents crossing of boundaries and “leaking” of sub-optimal logic
- Helps to keep this insulation transparent between communicating parties

Potential “gotchas”:

- Additional complexity
- Additional testing required to validate the additional complexity

Microservice / API Design Patterns – Strangler Application

- Supports breaking up a monolith into microservices over time
- Can accommodate existing as well as new functionality



Microservice / API Design Patterns – Strangler Application

Benefits:

- Allows a gradual vs. “big bang” breakup
- Especially well-suited for large monoliths that are taking active traffic

Potential “gotchas”:

- Requires business to be able to absorb gradual
- Adds complexity – coordinating new functionality, migrated functionality, and proper integration between them
- While transition in progress, will have to maintain two paths

Managing Integration

- As part of the defined contract, your API can accept multiples types of input
- Could include JSON payloads as well as query string parameters provided on the URL
- As the creator of an API, you will want to validate inputs to ensure the API is being used in the manner intended
- Often, with APIs, we use the strategy of “smart endpoints, dumb pipes”

DEMO:

Query Parameters for Filtering

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/demos/demo01

LAB:

Orders API with Parameters & Validation

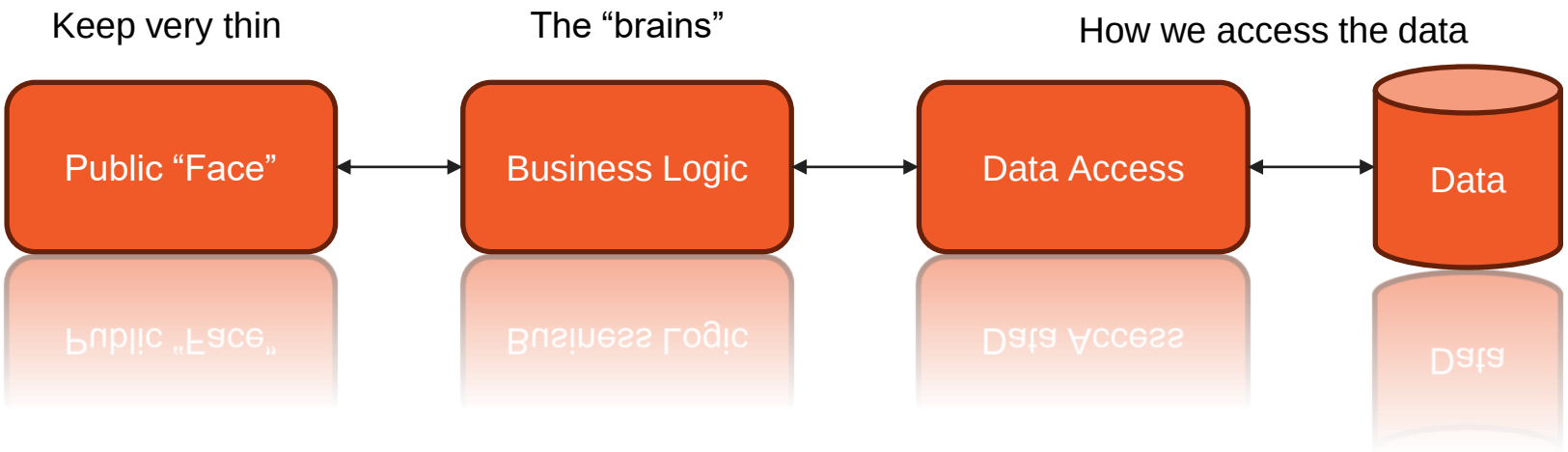
https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/labs/lab03

Partitioned Design

What is Partitioned Design?

- Creating separate components each adhering to SRP
- Each component is “tasked” with a specific aspect of the API’s implementation
- Properly maintains separation of concerns and better enables test isolation
- Also, provides a layer of abstraction which allows new strategies to be swapped in as required while minimizing impact on the rest of the system

What is Partitioned Design?



- Code to abstractions, not to concretions
- Leverage techniques like dependency injection to keep the components loosely-coupled

LAB:

Orders API with Backing Datastore

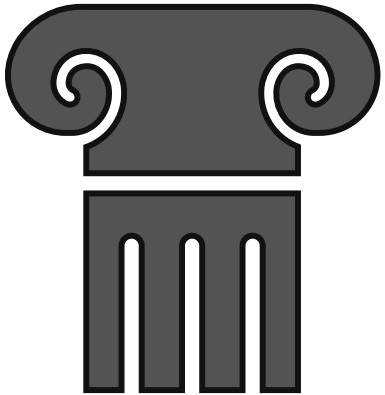
https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/labs/lab04

API Types

Types of APIs

Types of APIs

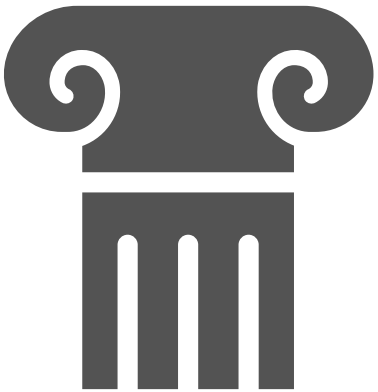
- Stands for Representational State Transfer
- Provides access to data across a network (including the Internet)
- Combines the concept of “resources” with a set of HTTP verbs, often for managing CRUD for a given entity or set of entities
- URLs provide logical routing through a hierarchy of related resources, allowing drilldown at multiple levels
- Typically use JSON format for requests to and responses from the API



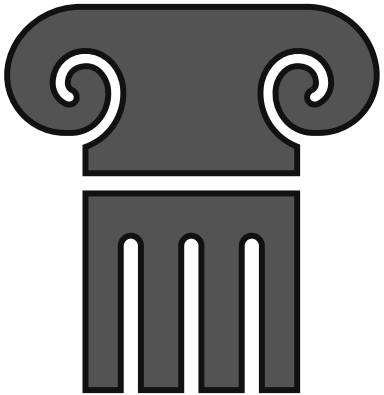
REST

GraphQL

Types of APIs



REST



GraphQL

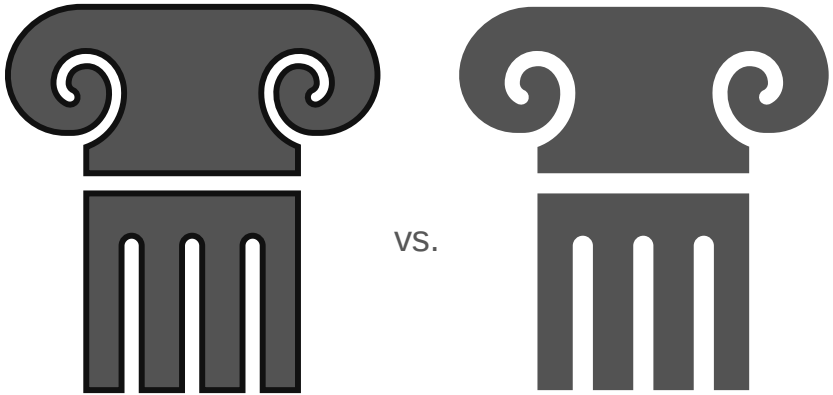
- Intended to help address use case where multiple service calls required to pull different data sets together for a given view
- Depending on network performance, multiple calls across multiple services to pull the subset of data needed can introduce latency
- Seeks to think about data as a holistic graph of related information vs. a set of service endpoints
- Using this unified data graph, client platforms can be targeted with their retrieval, pulling only the data needed in a single call
- Also, supports use case of streaming multiple API calls into a single interface
- Intended more for client-to-server rather than server-to-server communications

Defining the Contract

Schemas & Types in GraphQL: https://graphql.org/learn/schema/

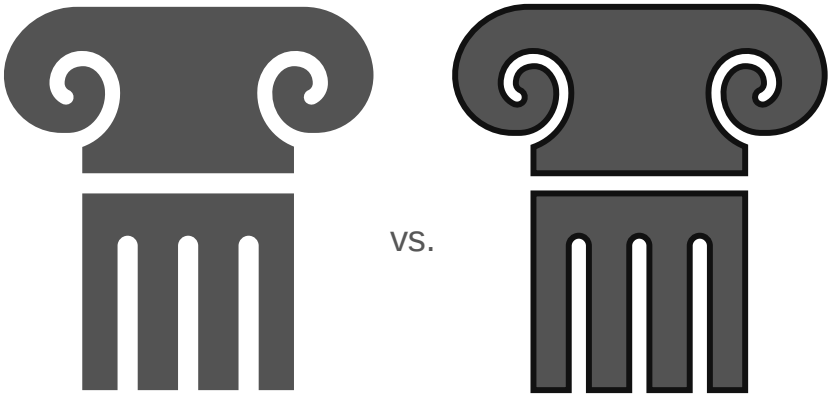
Types of APIs – When to Use Each Type

- Consistent interface across all clients of the service
- Reduces complexity by minimizing number of endpoints required for interaction
- Because interface is consistent, cache strategy is simplified since what you cache will be usable by all clients of the service
- Intended to provide access to fuller set of data results (full schema in hierarchical form)
- Good option for server-to-server communication
- Also, good option if you only have one type of front-end client



REST GraphQL

Types of APIs – When to Use Each Type



REST GraphQL

- Often used for client-to-server integration, especially where multiple client types are involved
- Enables multiple interfaces for multiple types of clients (e.g., mobile vs. browser) and each interface can be catered to (and optimized for) the individual client types
- Allows each client to only pull the data needed
- Supports a strong type system
- Also, enables return of that data in a format that best fits the client – instead of rigid schema defined for the underlying data sources, able to transform for specific client purposes

LAB:

Defining Schema For and Consuming GraphQL APIs

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/labs/lab05

LAB:

Building a GraphQL API with TypeScript

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/labs/lab06

LAB:

GraphQL API (Server & Client)

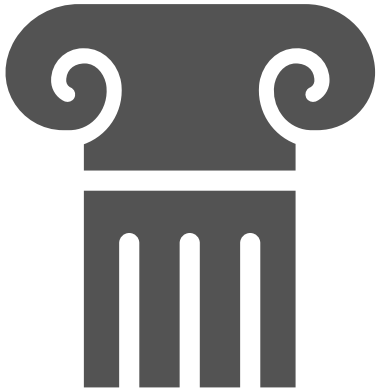
https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/labs/lab07

LAB:

Implementing a GraphQL API Server

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/labs/lab08

Types of APIs



gRPC

https://aws.amazon.com/compare/the-difference-between-grpc-and-rest/

LAB:

Building a gRPC API

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/labs/lab09

Securing APIs

AuthN / AuthZ

AuthN / AuthZ

How do I know you are who you say you are?

Authentication

Authorization

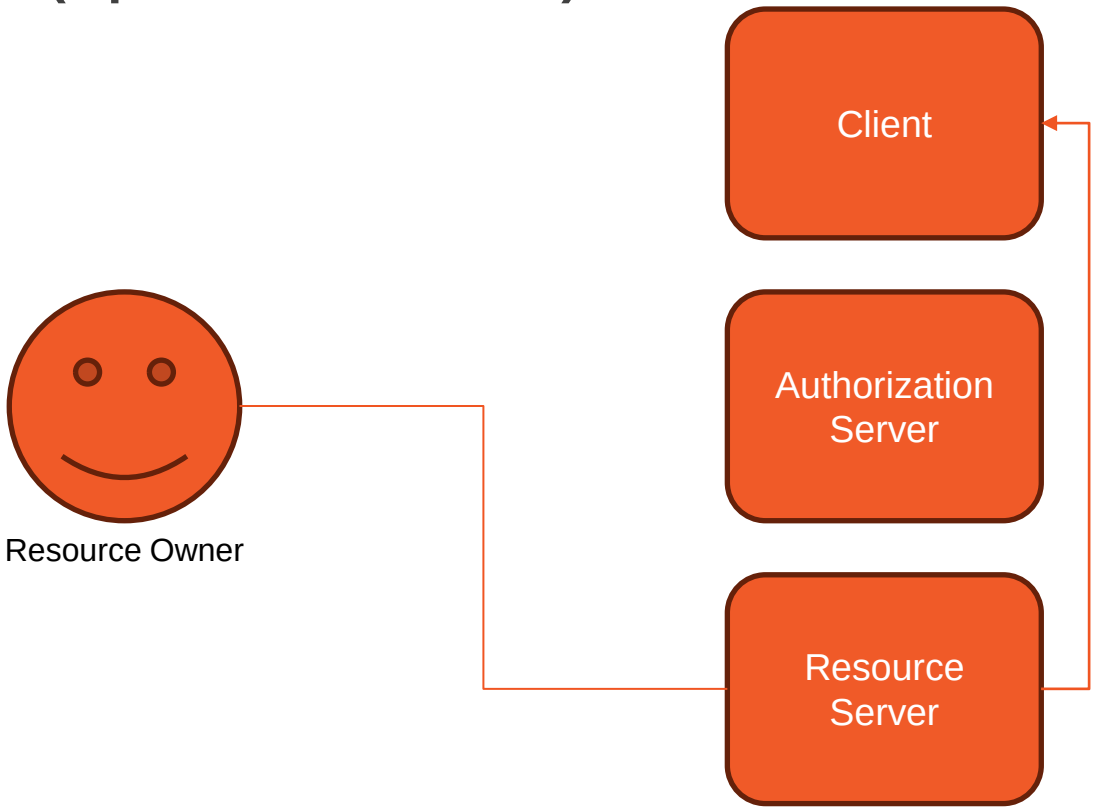
Once I know who you are, how do I manage what you're allowed to do?

OAuth

OAuth (Open Authorization)

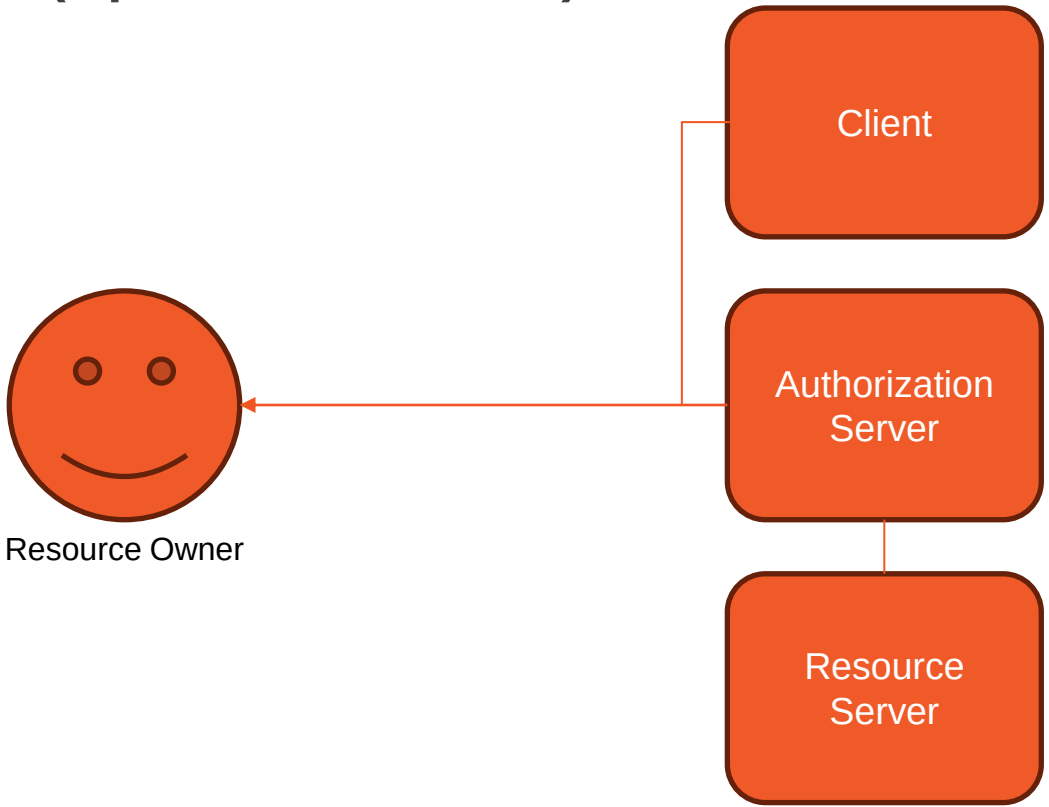
- Open standard that facilitates access grants between users and applications
- Leverages a token to access info on behalf of a user or system without requiring exposure of sensitive credentials

OAuth (Open Authorization)



Resource owner chooses to share secured resources (managed by the resource server) with another application or system (the client)

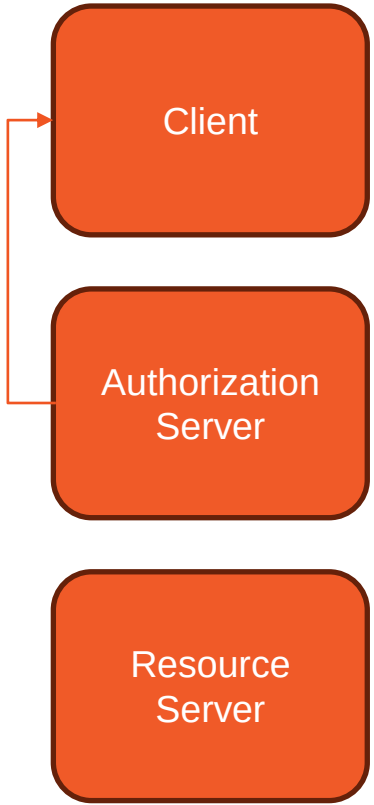
OAuth (Open Authorization)



Client reaches out to authorization server (which provides direct interaction with resource owner) to gain authorization from the owner to access the secured resources on the resource server

OAuth (Open Authorization)

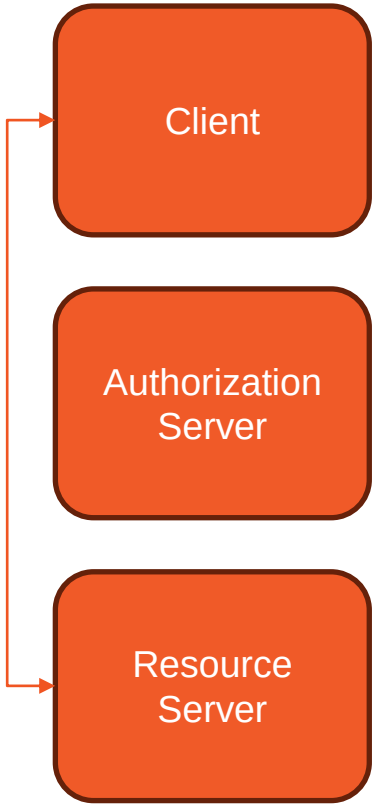
Resource Owner



Once identity is confirmed and access is granted by owner, authorization server provides the client with a time-based token that can be used for delegated operations against the resource server

OAuth (Open Authorization)

Resource Owner



Using the token, the client can access the secured resources

OAuth Flows

- Authorization Code Flow
- Proof of Key for Code Exchange (PKCE) Flow
- Client Credentials Flow
- Refresh Token Flow

Authorization Code Flow

- Client server exchanges a secret with the authorization server that produces a signing URL
- Provides a one-time use code that can be exchanged for an access token
- Requires the use of a client secret which makes it inappropriate for applications where code might be publicly exposed
- OAuth 2.1 recommends combining with the PKCE flow for added security with these types of applications

Proof of Key for Code Exchange (PKCE) Flow

- Provides added security through an extension on top of code flow
- Adds an additional layer of exchange to help protect sensitive client secrets
- Client generates an encoded secret called the code verifier
- Client sends both the code verifier along with the standard code challenge when attempting to access secured resources
- Server produces an authorization code that can be exchanged by the client for an access token – requires that the client send both the authorization code and the standard code challenge

Client Credentials Flow

- Good fit for server-to-server (vs. user-to-server) communication
- Through interaction between internal resources, able to securely exchange a secret for an access token

Refresh Token Flow

- Access tokens are only valid for a limited time (to guard against impact of token exposure/leakage)
- This flow allows retrieval of a new access token once original has expired
- Clients typically receive both an access token and a refresh token
- Refresh tokens are valid for one-time use and get refreshed on refresh to access token

OIDC

OIDC (Open ID Connect)

- Open standard for identity verification built on top of OAuth
- Enables authentication for a user to a website or service through a 3rd-party identity provider
- For example, using your Google account or one of your social media accounts to log you into a site or service
- Keeps identity provider more decoupled and allows user to reuse accounts across services

OIDC (Open ID Connect)

- Identity Providers that support OIDC an endpoint that can be used to get OIDC configuration at the provider (.well-known/openid-configuration)
- AKA the Discovery Endpoint
- For example, https://accounts.google.com/.well-known/openid-configuration

Tokens

OAuth Flows

Carry identity of the user or principal – used exclusively for verifying identity not access

Used to validate access to a service or API

OAuth Flows

Audience is Authorization Server

ID Tokens

Access Tokens

Audience is API Server

JWTs (JSON Web Tokens)

- Take a look at http://jwt.io to explore

LAB:

Orders API Including AuthN / AuthZ

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/labs/lab10

Testing APIs

DEMO:

Testing APIs Using Pact

https://docs.pact.io/implementation_guides/workshops

Creating APIs Using Lambdas

DEMO:

Creating APIs Using Lambdas

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/labs/lab11/categorize-example

Hack-a-thon

Step 01:

Build User Service/API

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/hackathon/supporting-docs/step-01.pdf

Step 02:

Build Toy Service/API

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/hackathon/supporting-docs/step-02.pdf

Step 03:

Build Inventory Service/API

https://github.com/KernelGamut32/intermediate-api-dev-public/tree/main/hackathon/supporting-docs/step-03.pdf

Thank you!

If you have additional questions, please reach out to me at:
asanders@gamuttechnologysvcs.com

DevelopIntelligence
A PLURALSIGHT COMPANY