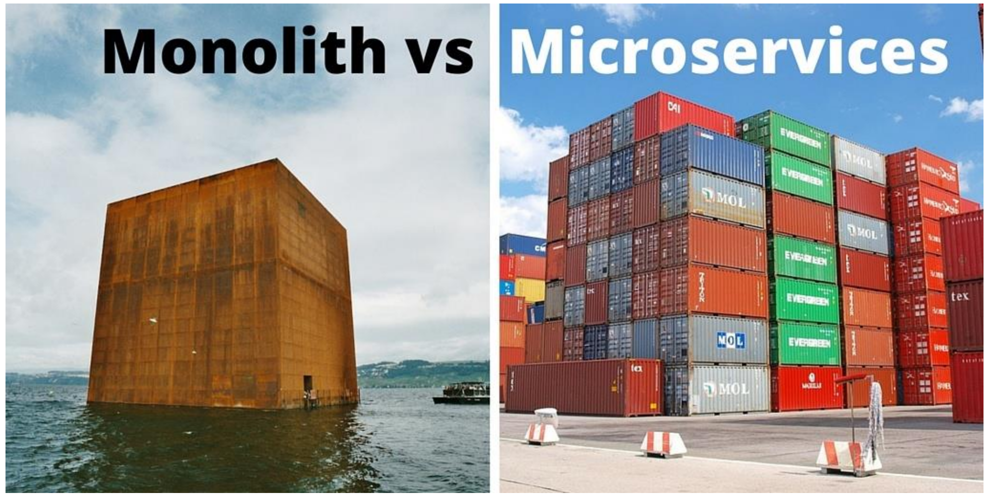
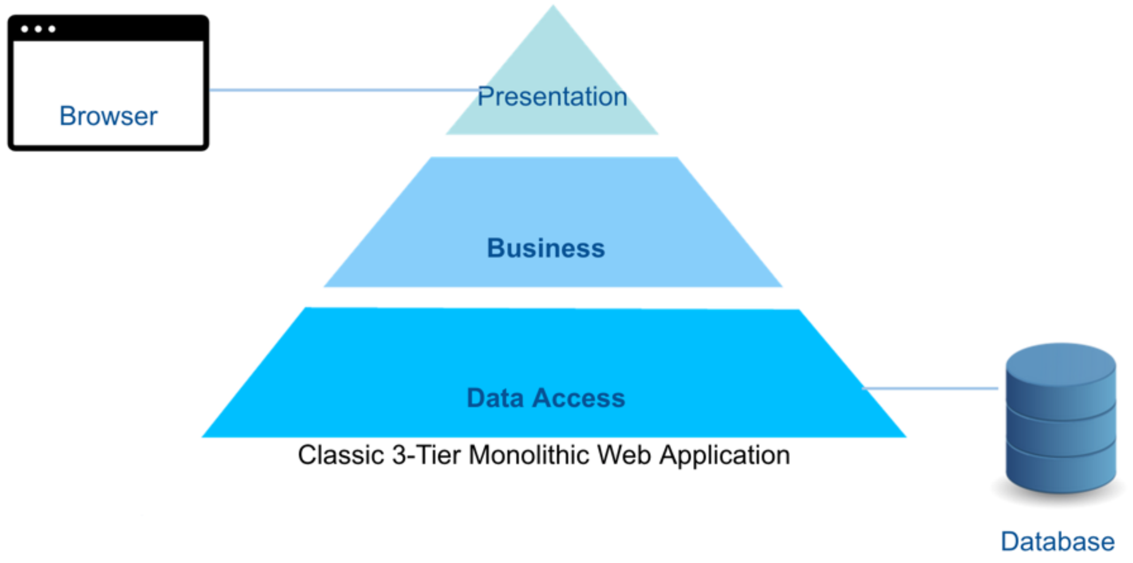


Microservice Architecture and Docker

Agenda

- Monolithic Applications
- Service Oriented Architecture (SOA)
- Microservices
- Docker
- Gartner Hype Cycle



Monolithic Applications

Traditional Applications:

Everything is integrated
Stereo Radio Cassette Recorder
AIWA
FULL AUTOMATIC STOP CUE/ REVIEW
TAPE COUNTER RESET
TAPE RUN
●ステレオ録音に威力を発揮する
ワンポイントステレオマイク
別売：CM-2000 ¥7,900

But, what happens if...

- We want a double cassette
- Or a CD player
- Or a digital radio
- ...

We have to change the whole thing

Again

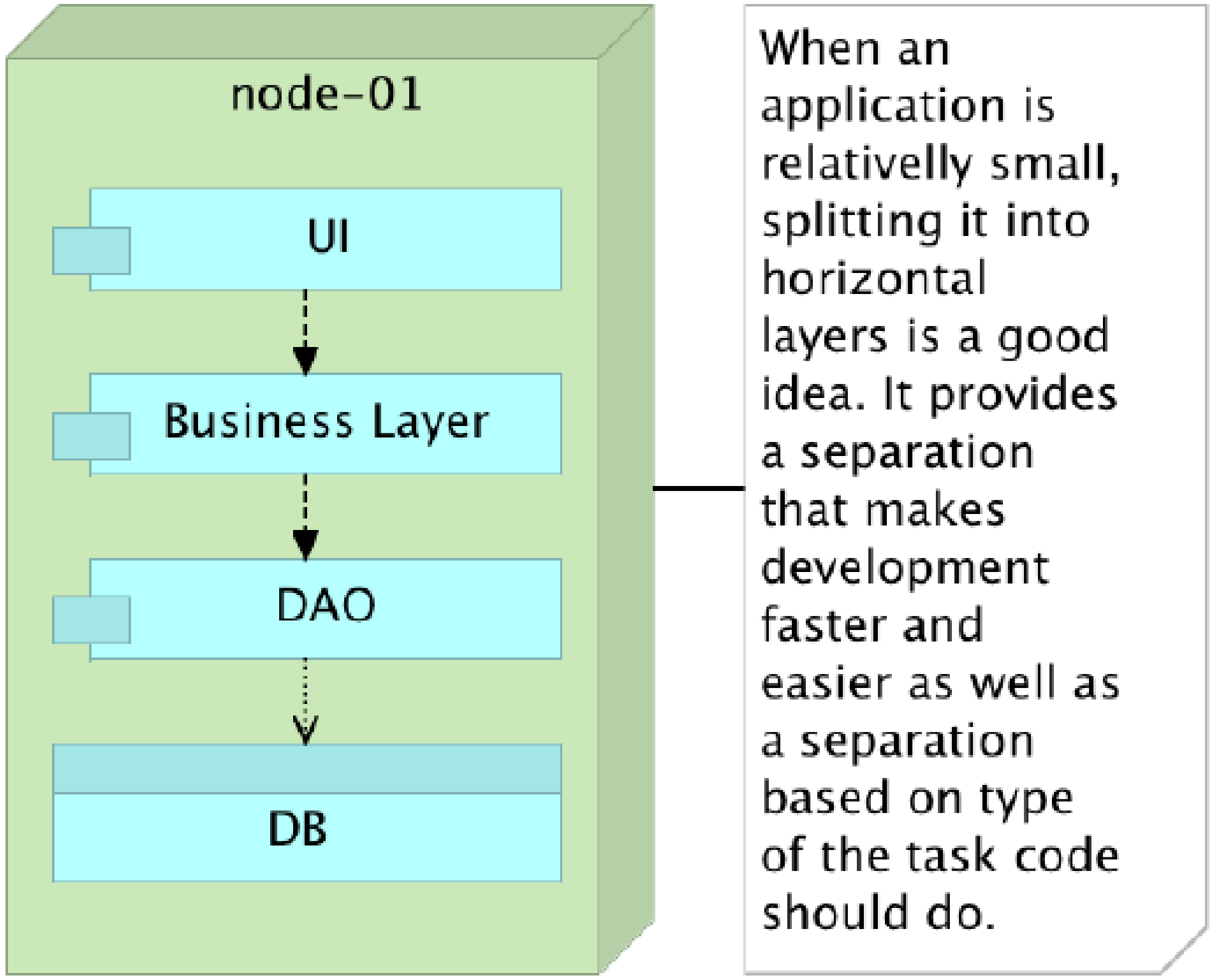
With Modular Applications

With Modular Applications

Each part is independent

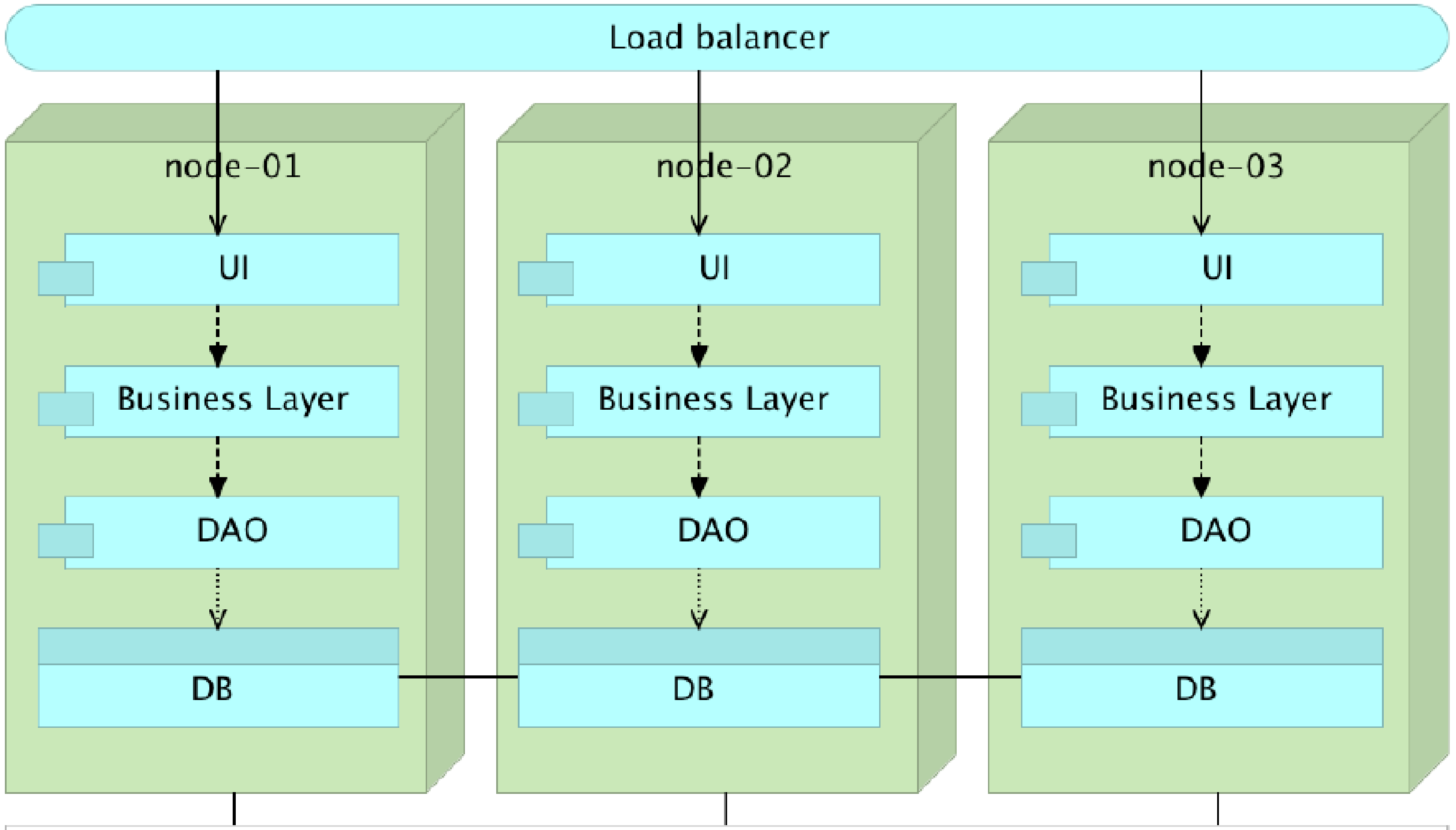
Do you Need a CD Player?

Monolithic Applications



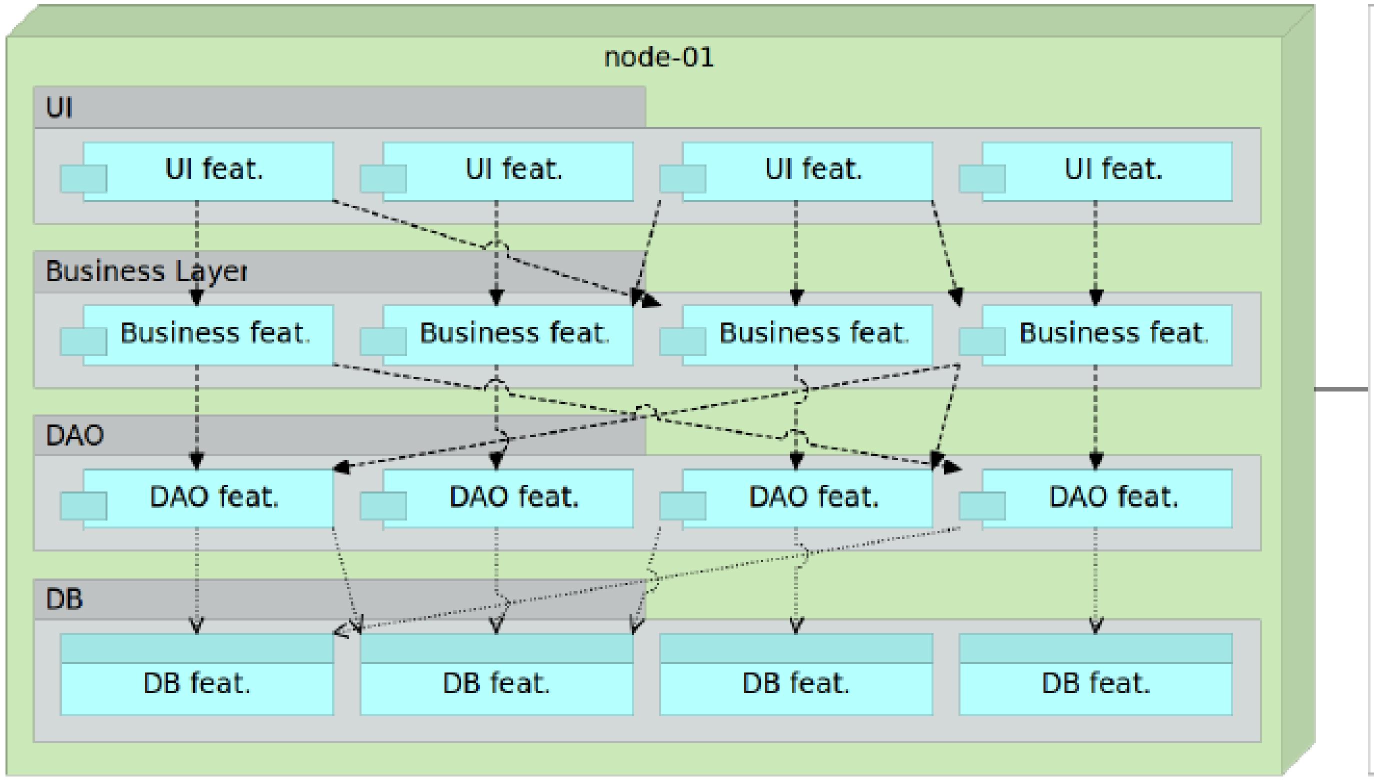
When an application is relativelly small, splitting it into horizontal layers is a good idea. It provides a separation that makes development faster and easier as well as a separation based on type of the task code should do.

Scaling Monolithic Application



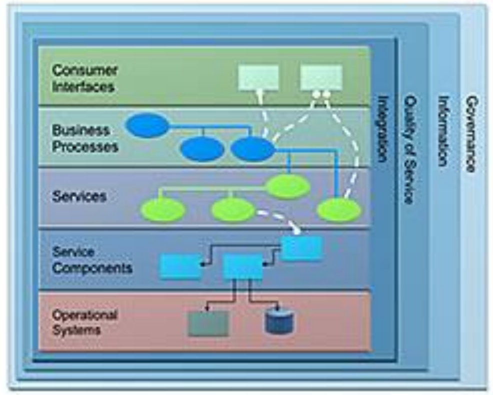
Scaling monolithic applications is very resource inefficient since everything needs to be duplicated on multiple nodes. There is no option to detect bottlenecks and scale or separate them from the rest of the application.

Monolithic Application with Increased Number of Features



When an application becomes bigger and the number of features increase, initial design based on horizontal layers becomes less efficient. Tight coupling between separate features, longer paths for potentially simple solutions, increased complexity, increased development and testing time, and so on.

SERVICE ORIENTED ARCHITECTURE



Service Oriented Architecture (SOA)

How to Buy Tomatos?

- Lookup for the tomato sellers

  Yellow Pages: contain companies that are selling tomatoes, their location, and contact information.

- Find the service offered according to my needs

  Where, when and how can I buy tomatoes?

- Buy the tomatoes

  Do the transaction

How to Access the Service?

- Lookup for the Service Provider

  Registry: contain providers that are selling services, their location, and contact information.

- Find the service offered according to my needs

  Where, when and how can I get the service?

- Access the service

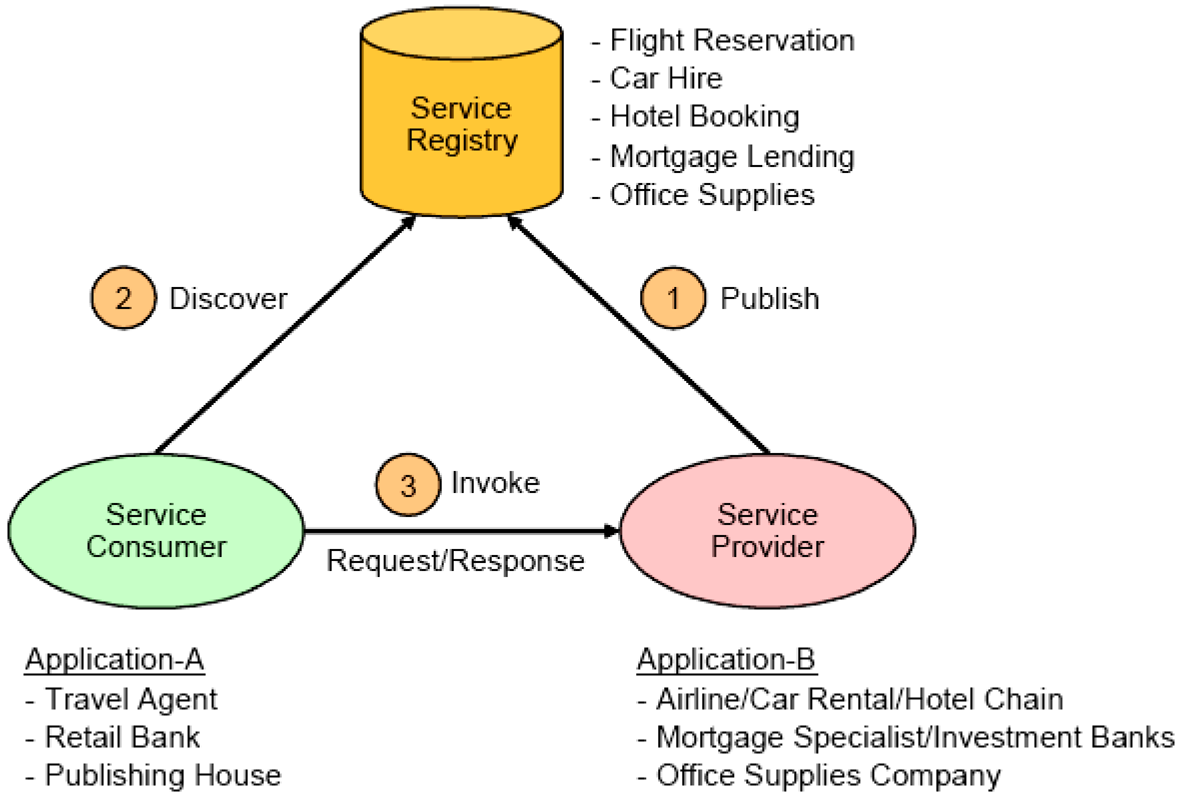
  do the transaction

Service-Oriented Architecture (SOA):

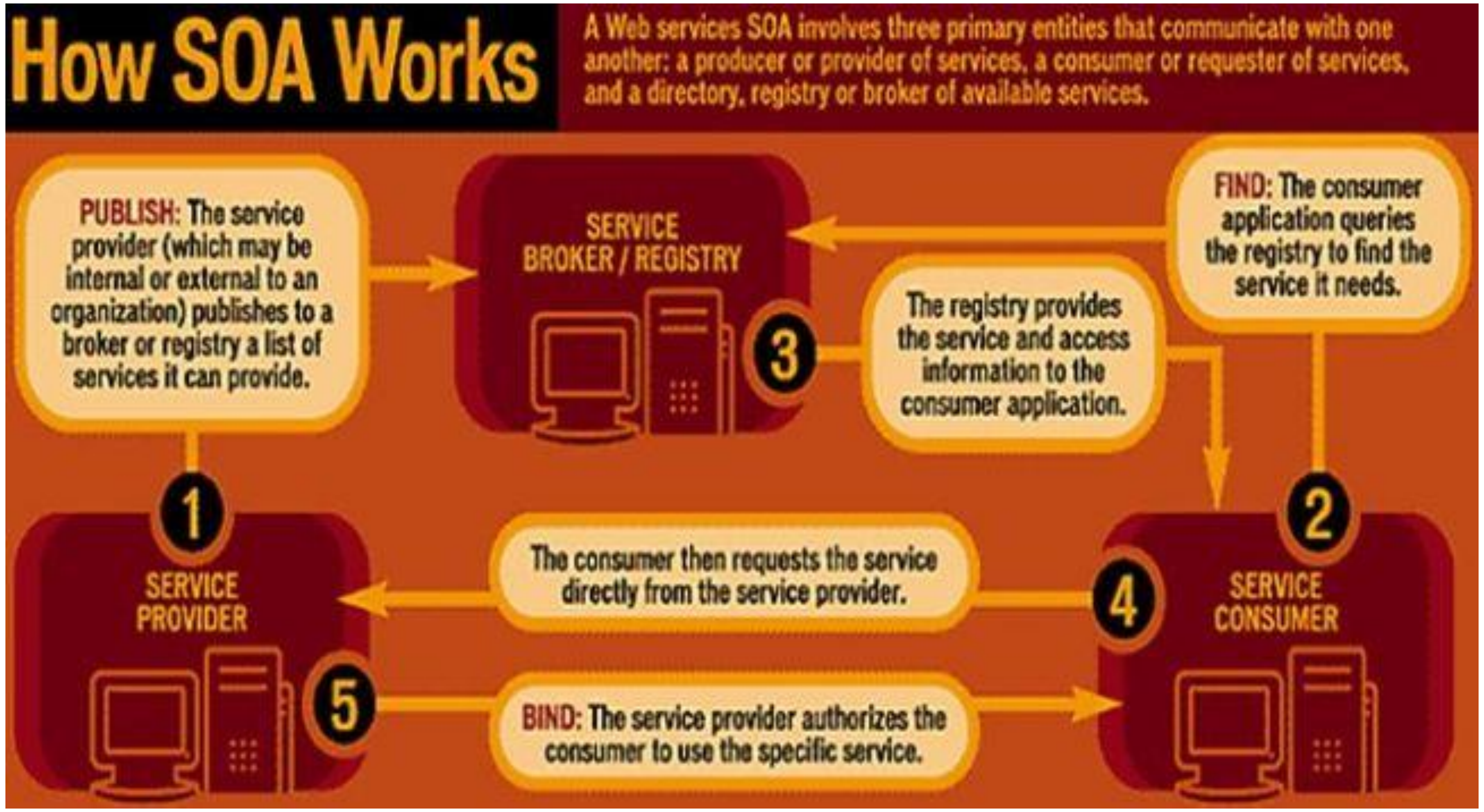
- *The policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer.*
- *Services can be*
    - *invoked,*
    - *published and*
    - *discovered,*

and are abstracted away from the implementation using a single, standards-based form of interface.

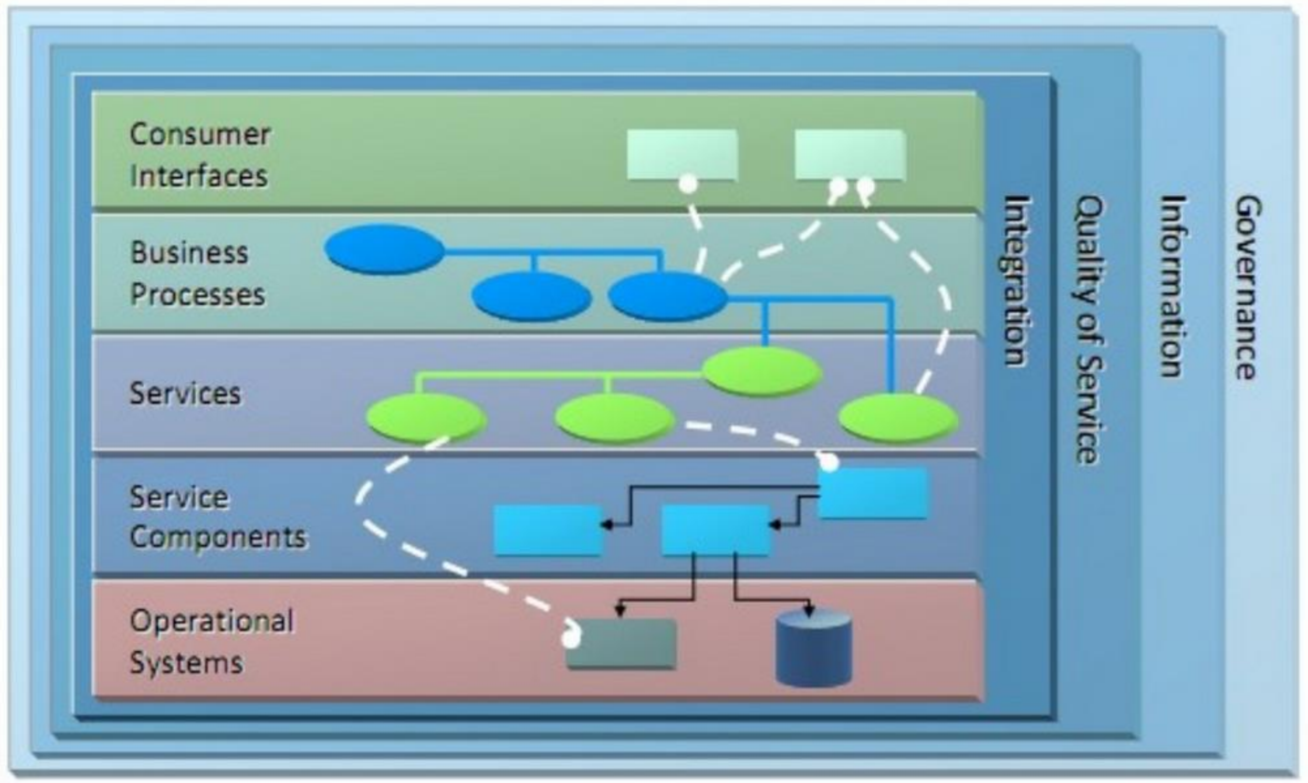
SOA Components and Operations



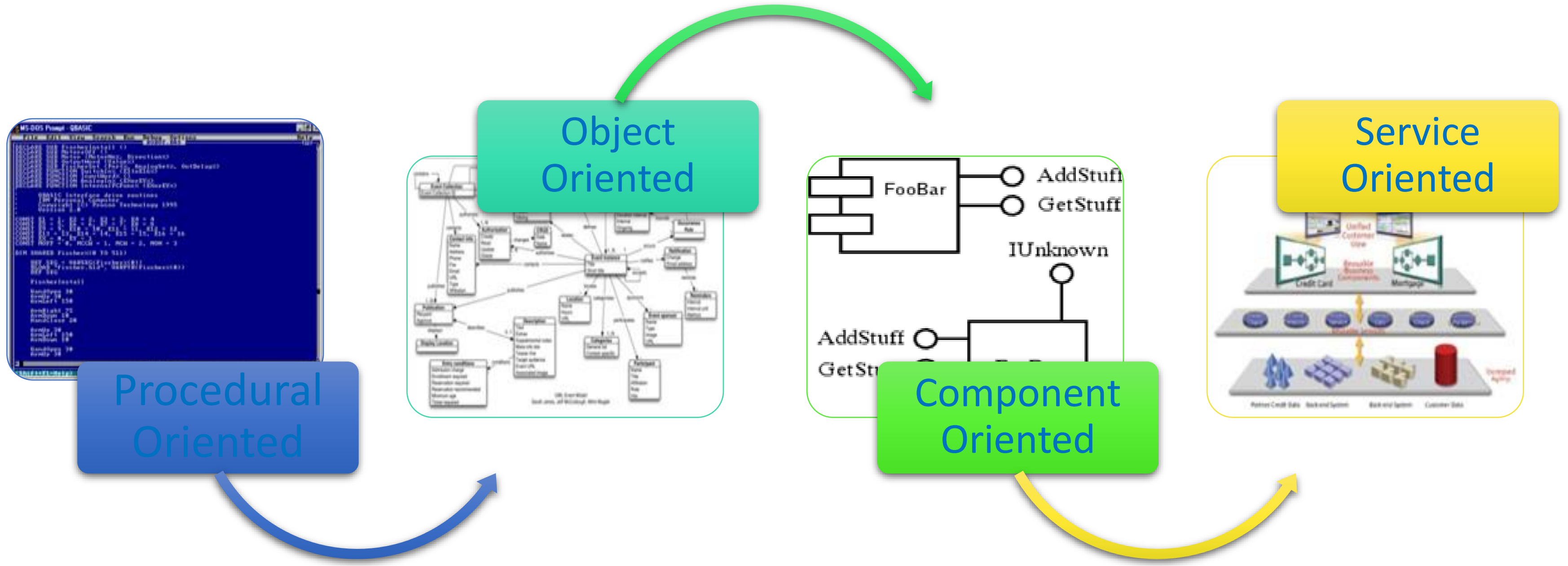
How SOA Works?


How SOA Works
A Web services SOA involves three primary entities that communicate with one another: a producer or provider of services, a consumer or requester of services, and a directory, registry or broker of available services.
PUBLISH: The service provider (which may be internal or external to an organization) publishes to a broker or registry a list of services it can provide.
SERVICE BROKER / REGISTRY
FIND: The consumer application queries the registry to find the service it needs.
3
The registry provides the service and access information to the consumer application.
1
SERVICE PROVIDER
2
SERVICE CONSUMER
The consumer then requests the service directly from the service provider.
4
5
BIND: The service provider authorizes the consumer to use the specific service.

Open Group SOA Reference Architecture (SOA RA)



How did it come to SOA?



SOA Challenges

SOA Alphabet Soup

What Would You Choose?

DISCO BEPL MoM

EAI rpc OASIS XML UDDI

XPath WS-I SODA

SOA ReST JBI XSD XSLT

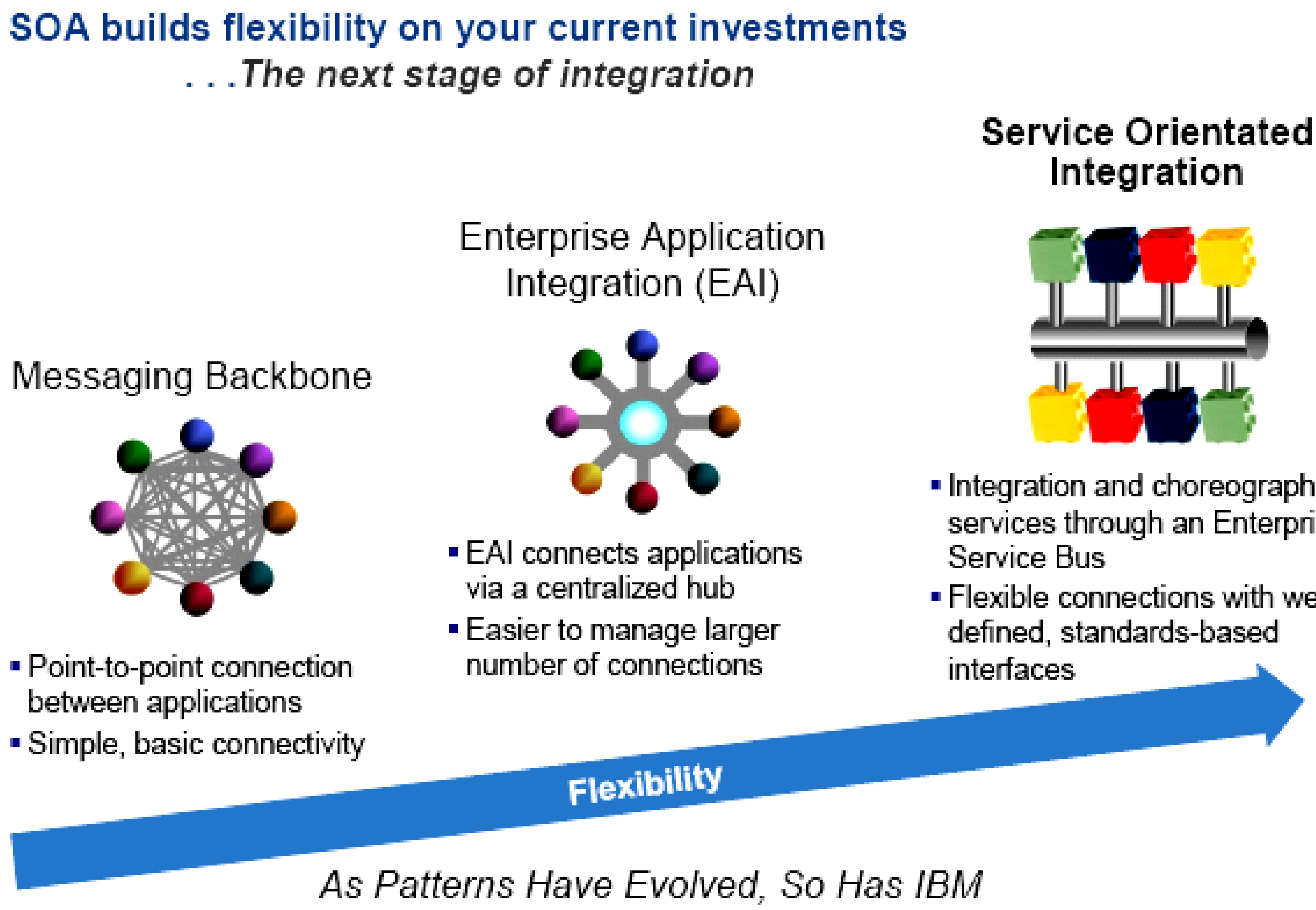
MESSAGING SOAP

SAML Web Service Axis DIME

WSDL SAX Digital Signature DTD

BEPL4WS DOM XSLT BPM Schema

Evolution of Application Integration Patterns



SOA builds flexibility on your current investments

. . .The next stage of integration

Messaging Backbone

- Point-to-point connection between applications
- Simple, basic connectivity

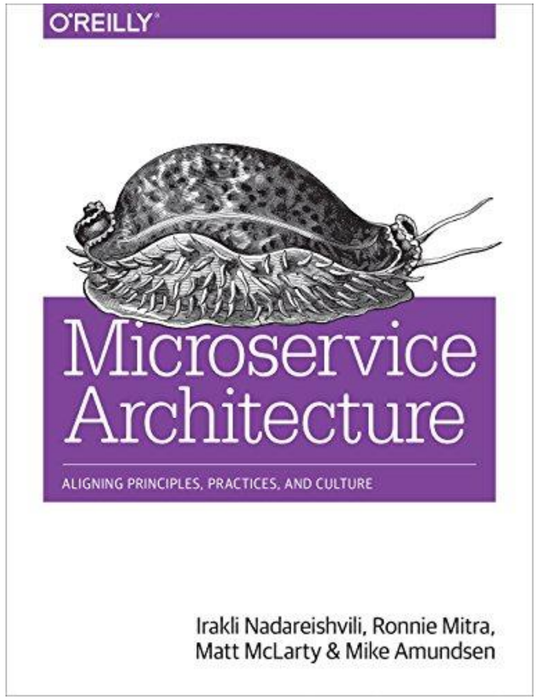
Enterprise Application Integration (EAI)

- EAI connects applications via a centralized hub
- Easier to manage larger number of connections

Service Orientated Integration

- Integration and choreography of services through an Enterprise Service Bus
- Flexible connections with well defined, standards-based interfaces

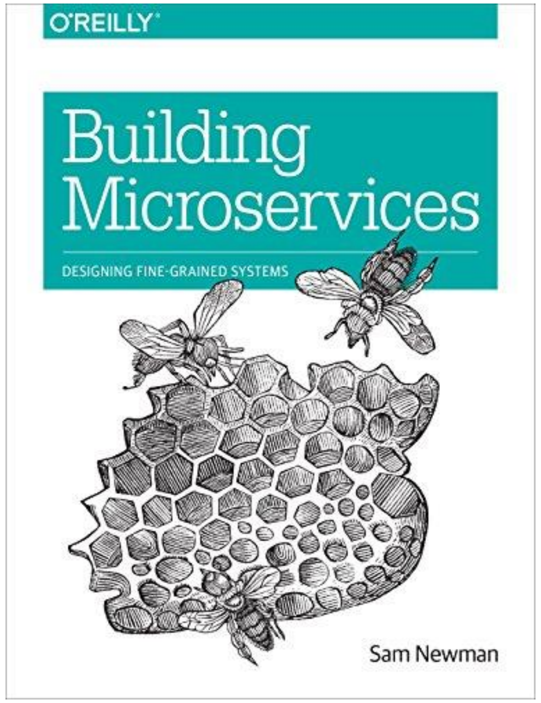
As Patterns Have Evolved, So Has IBM


O'REILLY

Microservice Architecture

ALIGNING PRINCIPLES, PRACTICES, AND CULTURE

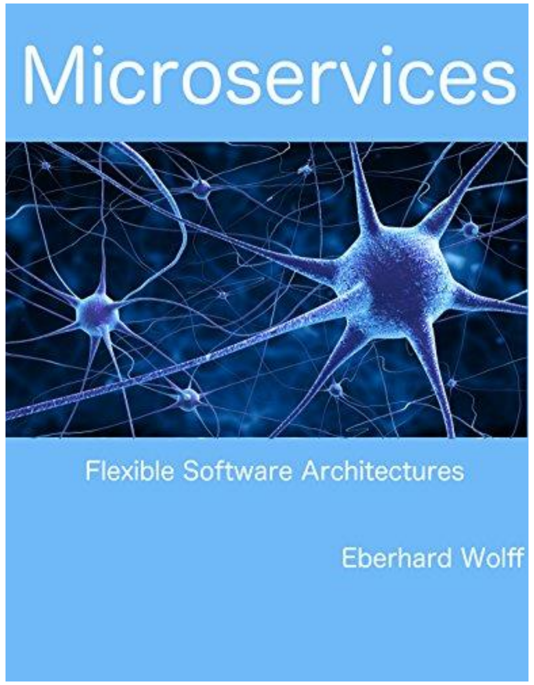
Irakli Nadareishvili, Ronnie Mitra, Matt McLarty & Mike Amundsen


O'REILLY

Building Microservices

DESIGNING FINE-GRAINED SYSTEMS

Sam Newman


Microservices

Flexible Software Architectures

Eberhard Wolff

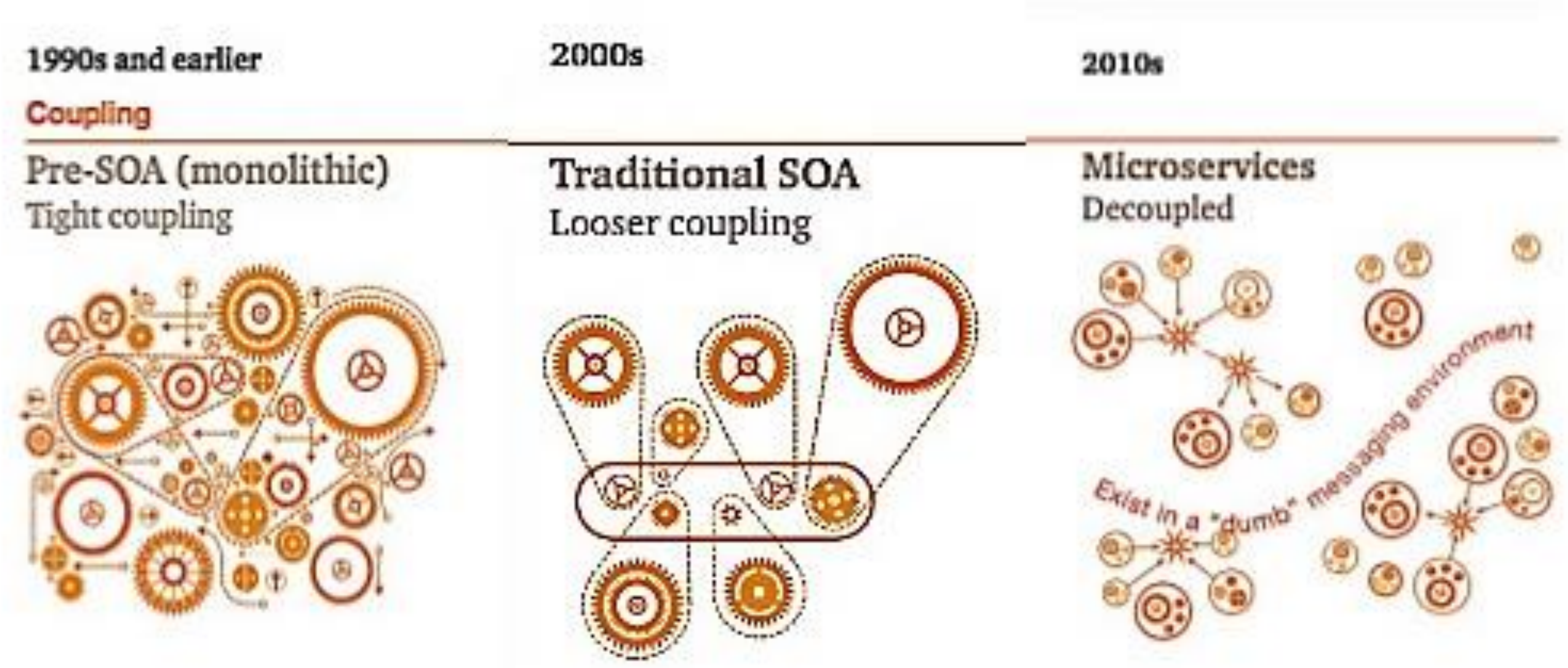
Microservices

MONOLITHS

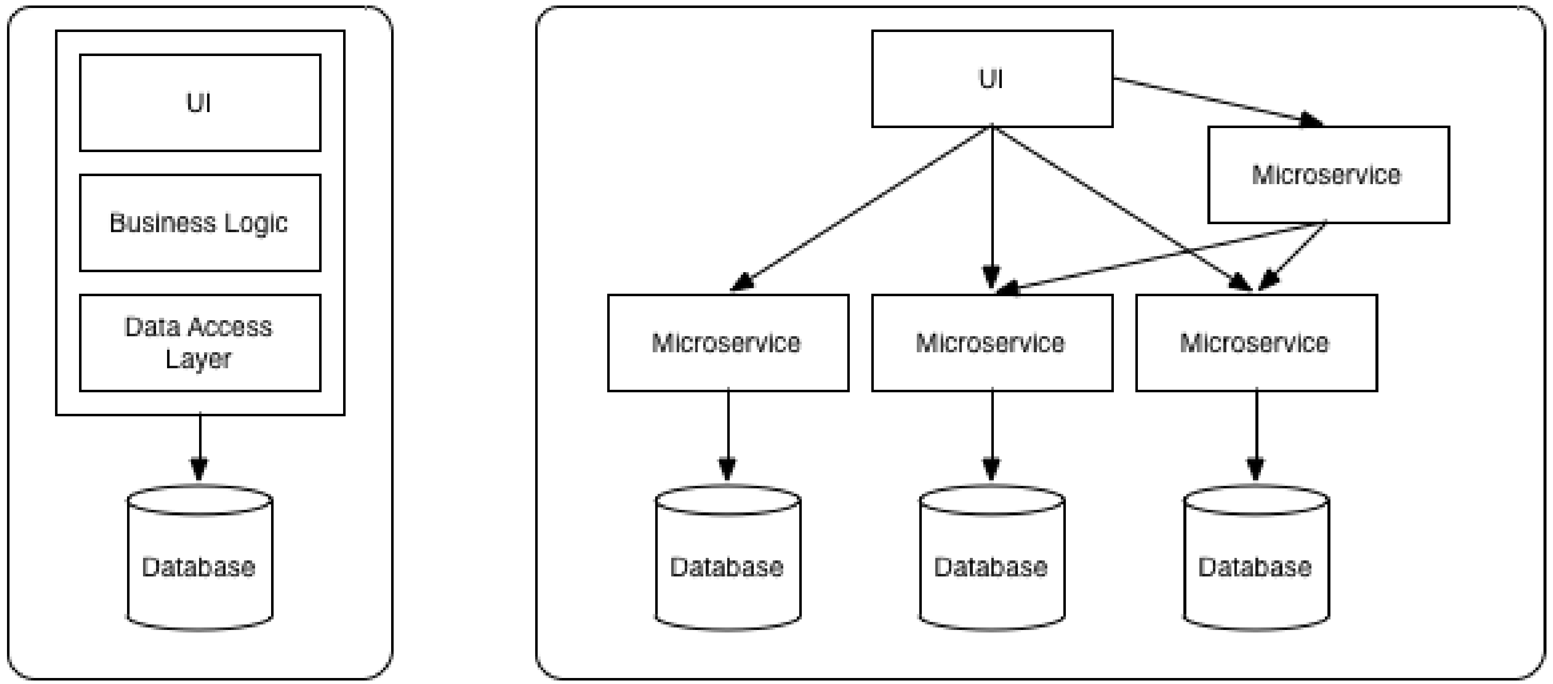
Hard to deliver, even harder to test and impossible to maintain

THE DEPENDENCIES WILL KILL YOU

Microservices



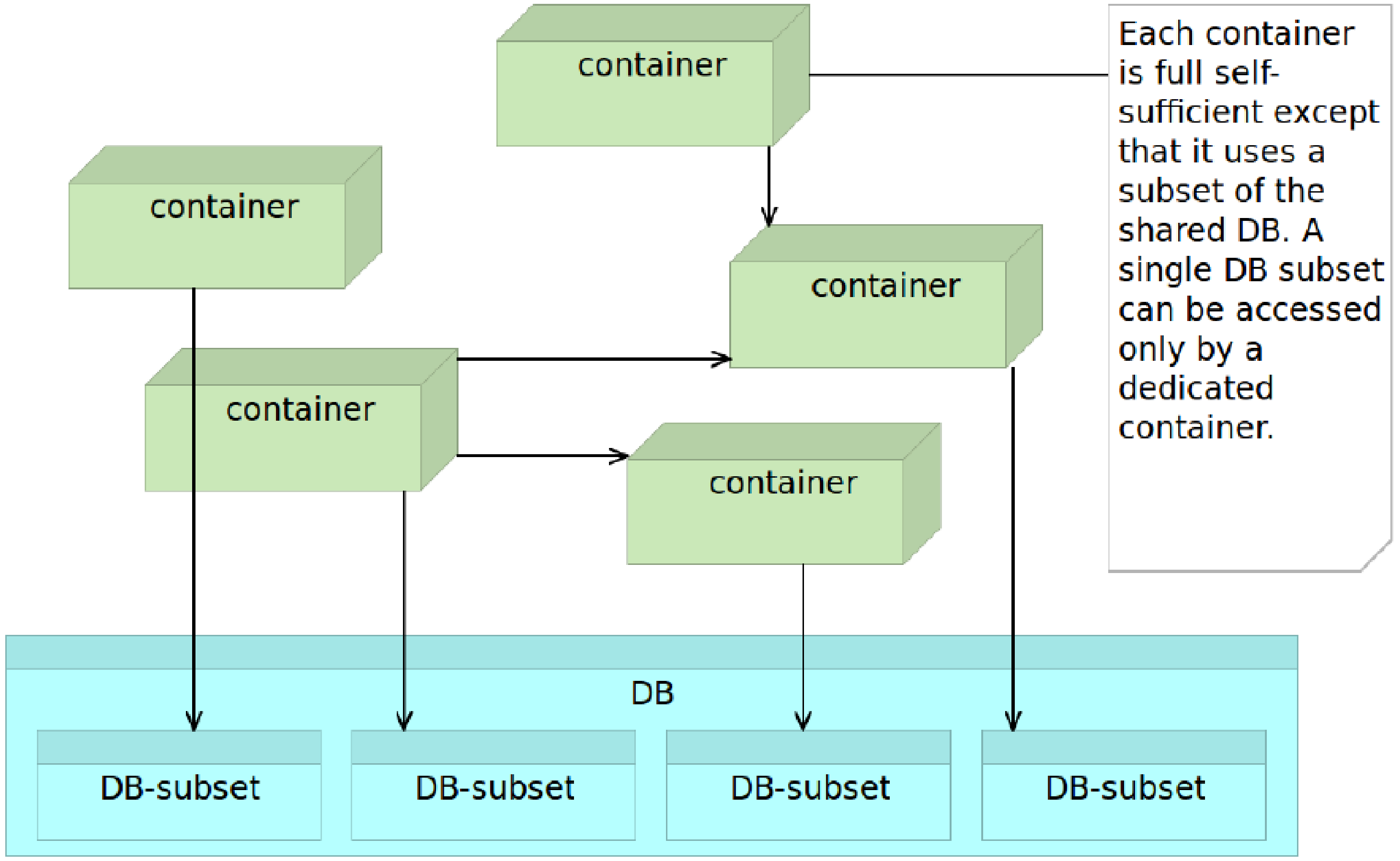
Microservices



Monolithic Architecture

Microservices Architecture

Microservices Accessing the Shared Database



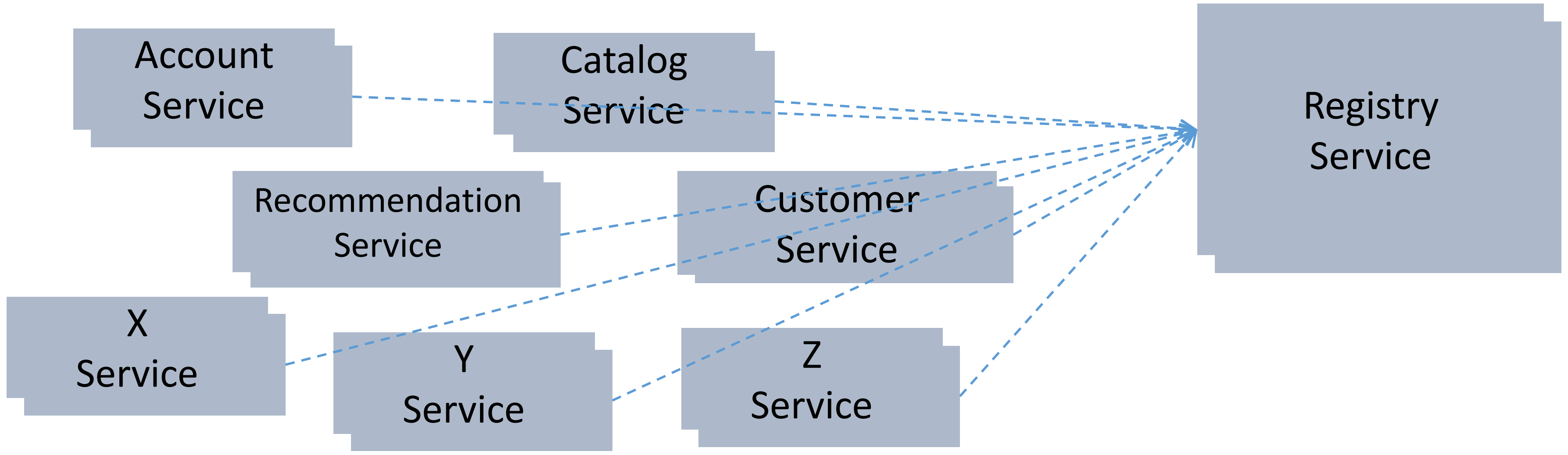
Each container is full self-sufficient except that it uses a subset of the shared DB. A single DB subset can be accessed only by a dedicated container.

Microservices Characteristics

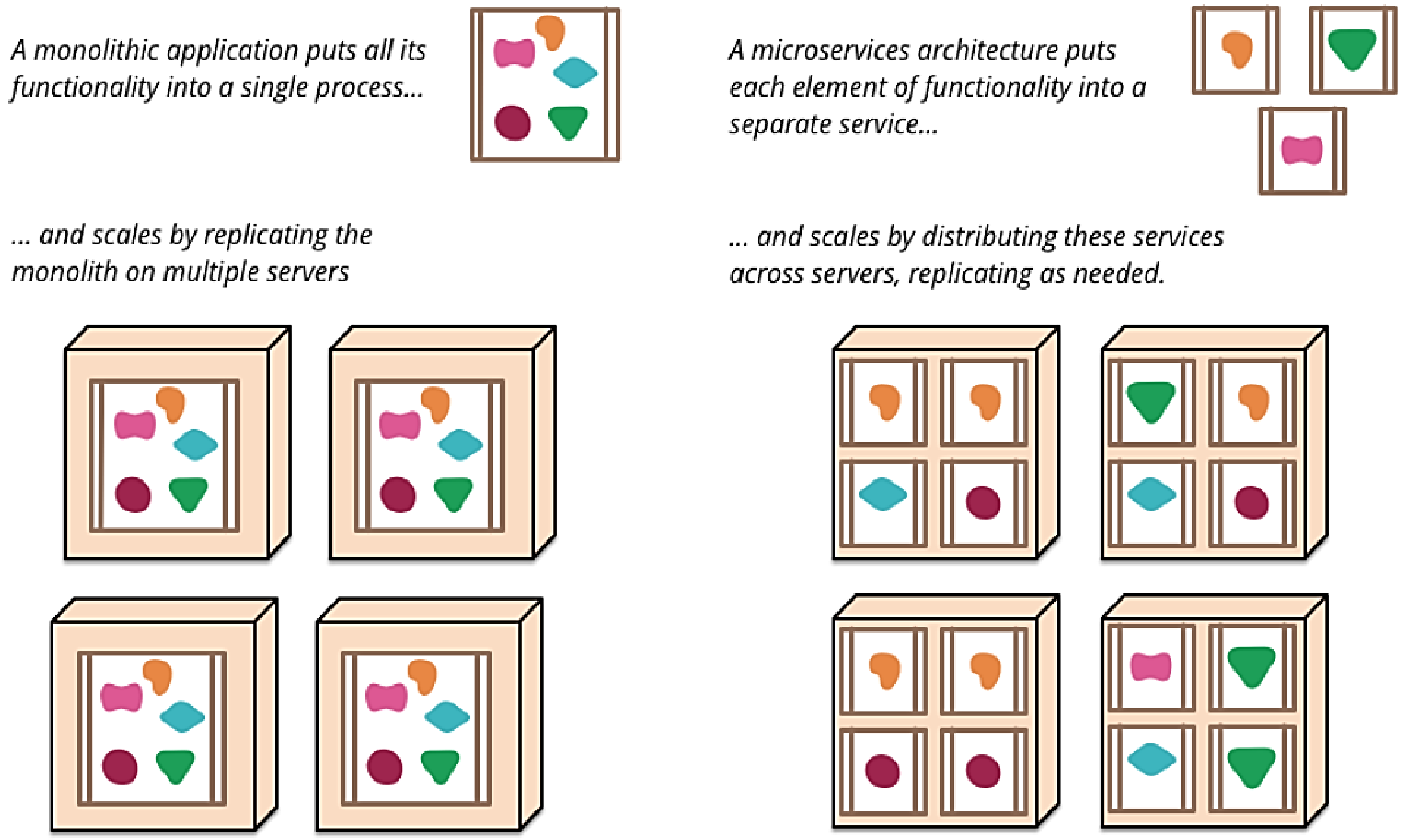
- Many smaller (fine grained), clearly scoped services
  - Single Responsibility Principle
  - Independently Managed
- Clear ownership for each service
  - Typically need/adopt the "DevOps" model

Service Discovery

- 100s of MicroServices
  - Need a Service Metadata Registry (Discovery Service)



Microservices. Scalability



A monolithic application puts all its functionality into a single process...

... and scales by replicating the monolith on multiple servers

A microservices architecture puts each element of functionality into a separate service...

... and scales by distributing these services across servers, replicating as needed.

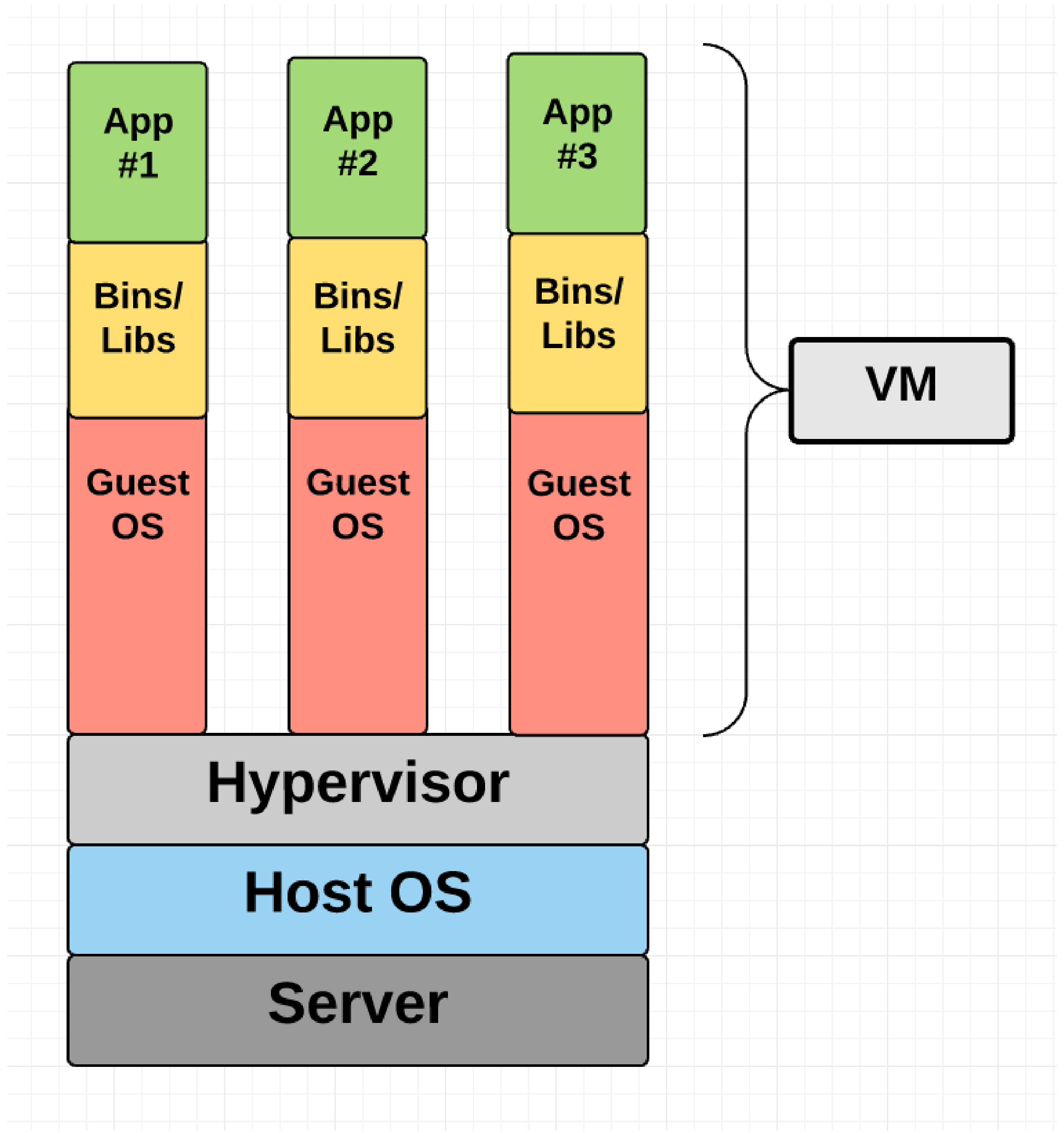
Docker

Docker: Containerization for Software

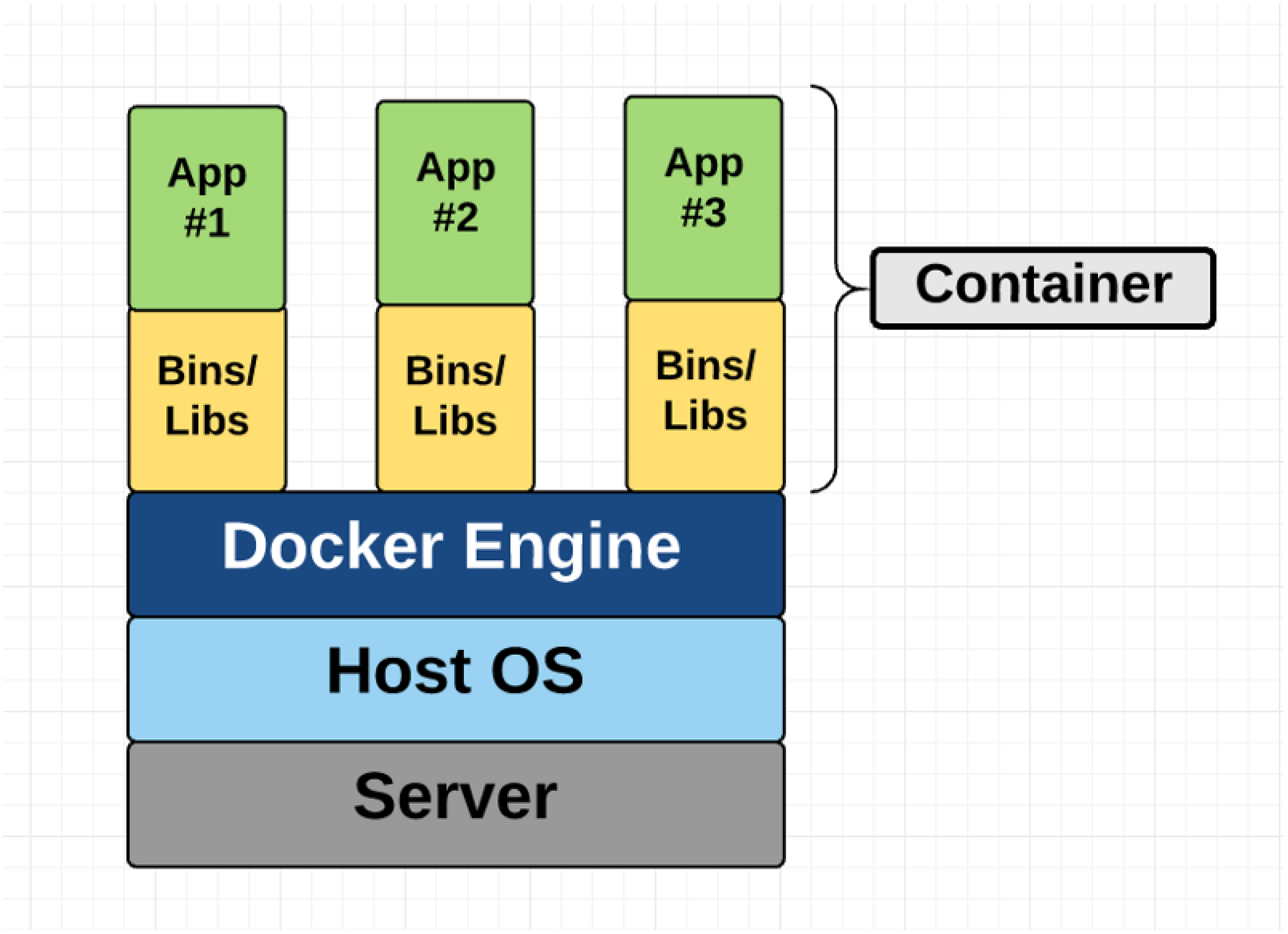
Docker

“Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications”

Virtual Machine



Container


App #1
App #2
App #3
Bins/ Libs
Bins/ Libs
Bins/ Libs
Container
Docker Engine
Host OS
Server

So why Docker?

- Containers are far from new;
  - Google has been using their own container technology for years.
  - Others Linux container technologies include
    - Solaris Zones,
    - BSD jails, and
    - LXC, which have been around for many years.
- Docker is an open-source project based on Linux containers. It uses Linux Kernel features.

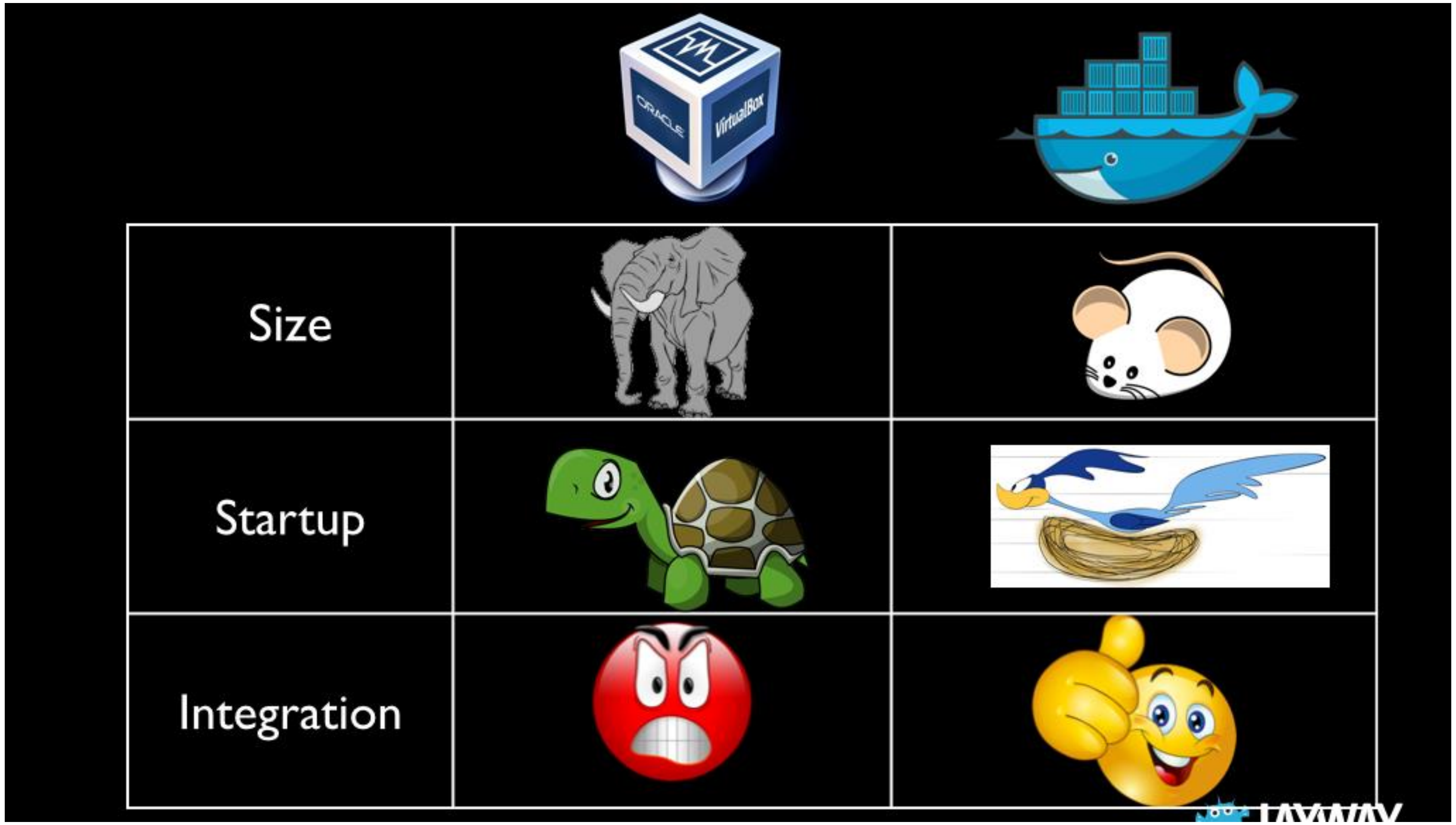
Docker Benefits

1. Local development environments can be set up that are exact replicas of a live environment/server.
2. It simplifies collaboration by allowing anyone to work on the same project with the same settings, irrespective of the local host environment.
3. Multiple development environments can be run from the same host each one having different configurations, operating systems, and software.
4. Projects can be tested on different servers.
5. It gives you instant application portability. Build, ship, and run any application as a portable container that can run almost anywhere.

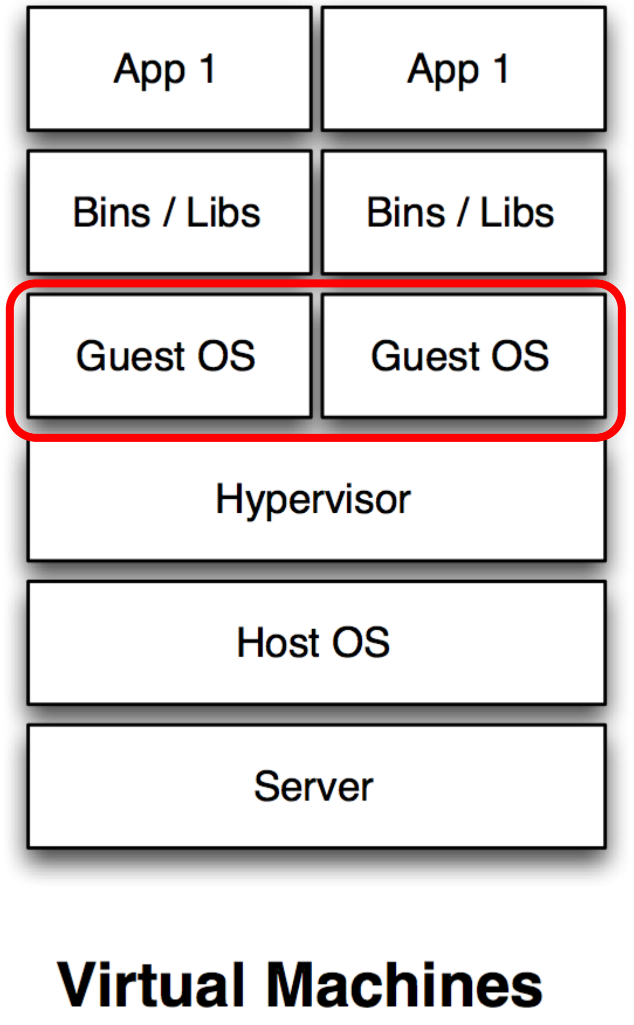
Why Docker?

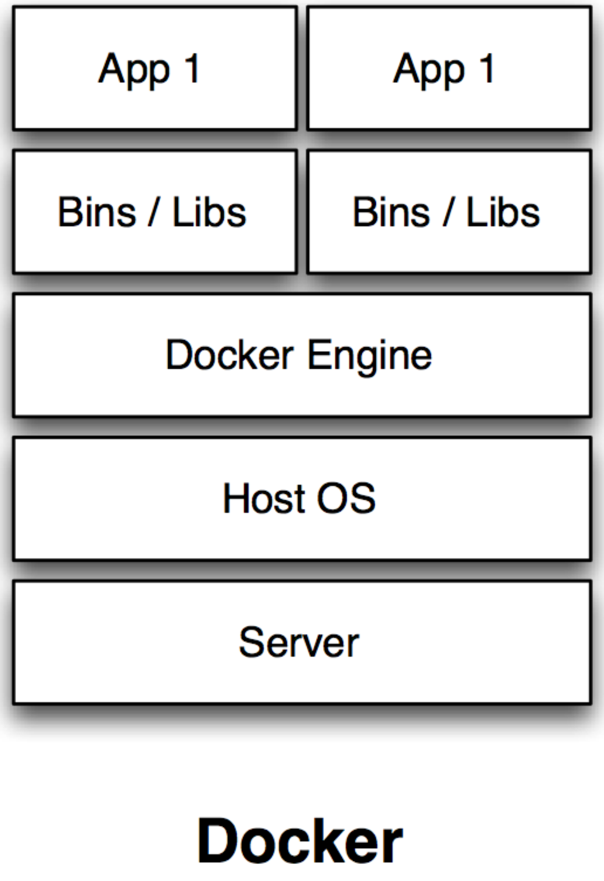
- **Ease of use.** It allows anyone to package an application on their laptop, which in turn can run unmodified anywhere
  - The mantra is: “build once, run anywhere.”
- **Speed.** Docker containers are very lightweight and fast. Since containers are just sandboxed environments running on the kernel, they take up fewer resources. You can create and run a Docker container in seconds, compared to VMs which might take longer because they have to boot up a full virtual operating system every time.
- **Docker Hub.** Docker users also benefit from the increasingly rich ecosystem of Docker Hub, which you can think of as an “app store for Docker images.” Docker Hub has tens of thousands of public images created by the community that are readily available for use.
- **Modularity and Scalability.** Docker makes it easy to break out your application’s functionality into individual containers. With Docker, it’s become easier to link containers together to create your application, making it easy to scale or update components independently in the future.

VM vs. Docker

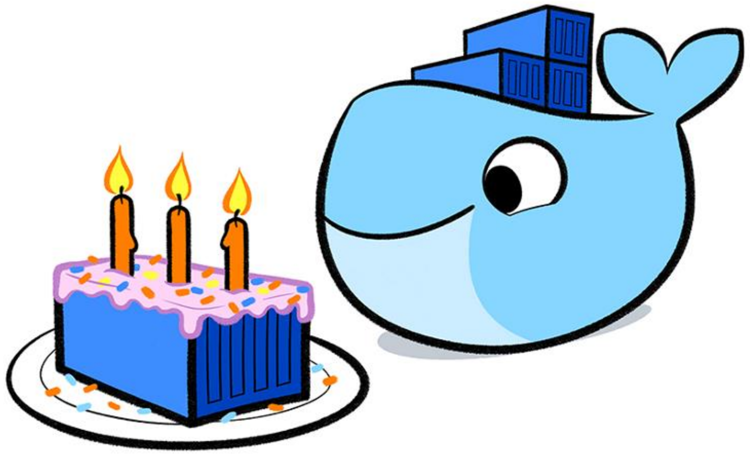


VM vs. Docker (Containers)



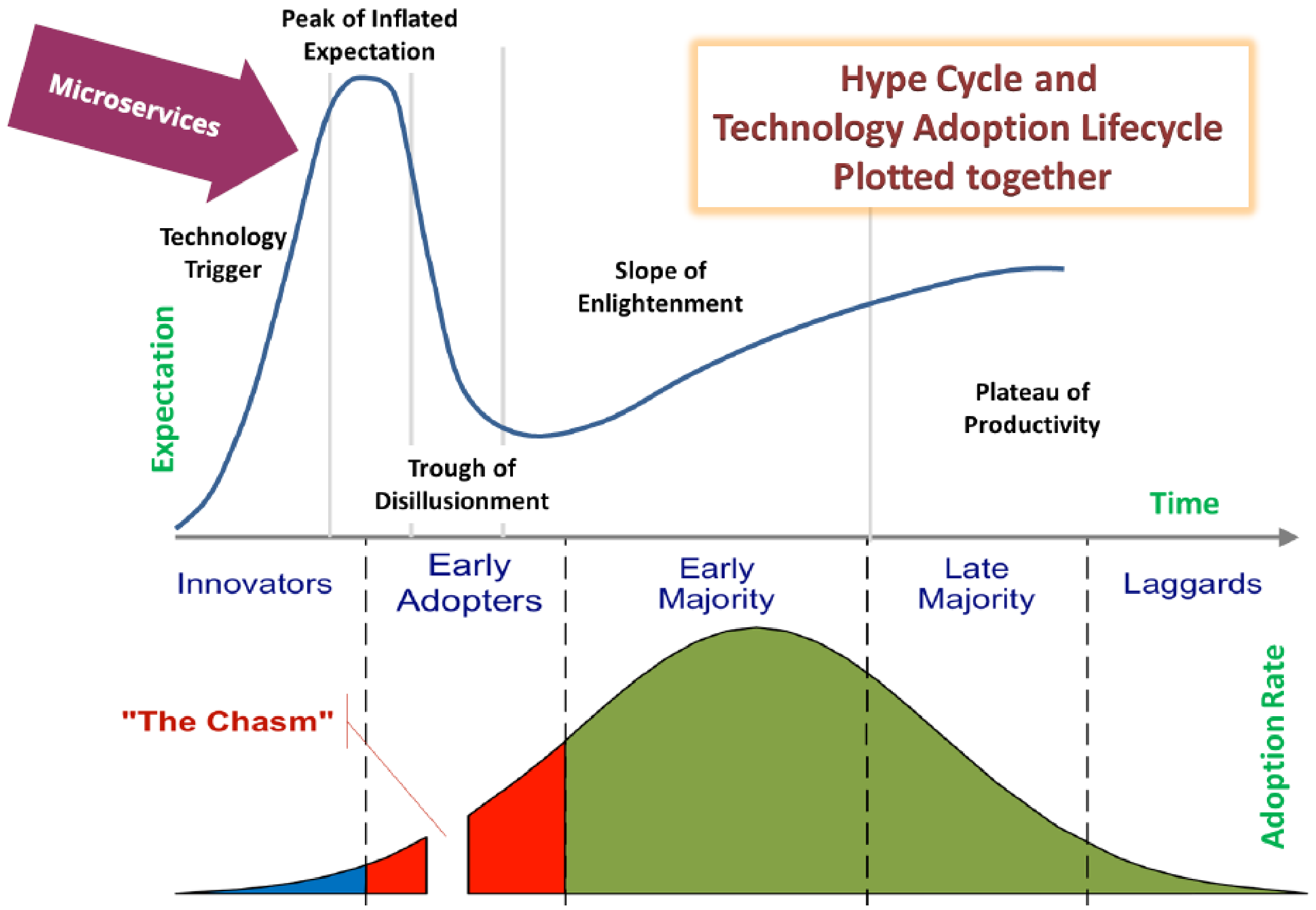


Docker Engine
Docker engine is the layer on which Docker runs.
It’s a lightweight runtime and tooling that manages containers, builds, and more.



Gartner Hype Cycle

Gartner Hype Cycle



general overview for developing and deploying containerized ASP.NET Core applications with Docker.

Introducing Microservices

- Twitter, PayPal, and Netflix had serious problems.
- Problems like scaling, quality, and downtime became common and increasing issues. Each had a large, single-code base application known as a “monolith.”
- Each hit different frustration points where a fundamental architecture change had to occur. Development and deployment cycles were long and tedious, causing delays in feature delivery.
- Each deployment meant downtime or expensive infrastructure to switch from one set of servers to another.
- As the code base grew, so did the coupling between modules. With coupled modules, code changes are more problematic, harder to test, and lower overall application quality.

Twitter

- For Twitter, scaling servers was a huge factor that caused downtime and upset users.
- All too often, users would see an error page stating Twitter is overcapacity.
- Many users would see the "Fail Whale" while the system administrators would reboot servers and deal with the demand.
- As the number of users increased, so did the need for architecture changes. From the data stores, code, and server topology, the monolithic architecture hit its limit.

Paypal

- For PayPal, their user base increased the need for guaranteed transactions.
- They scaled up servers and network infrastructure.
- But, with the growing number of services, the performance hit a tipping point, and latency was the result.
- They continuously increased the number of virtual machines to process the growing number of users and transactions.
- This added tremendous pressure on the network, thereby causing latency issues.

Netflix

- Netflix encountered problems with scaling, availability, and speed of development.
- Their business required 24 X 7 access to their video streams.
- They were in a position where they could not build data centers fast enough to accommodate the demand.
- Their user base was increasing, and so were the networking speeds at homes and on devices.
- The monolithic application was so complex and fragile that a single semicolon took down the website for several hours.

Monolith

- **there is nothing wrong with a monolith**
- when they need more server resources, it is usually cheap enough to add more servers.
- With good coding practices, a monolith can sustain itself very well.
- **Applications can grow into a burden over time**. With changes in developers, skillsets, business priorities, etc., those applications can easily turn into a "spaghetti code" mess.
- **By pulling functionality away from monolithic applications, development teams can narrow their focus on functionality and respective deployment schedule.**
- **This allows a faster pace of development and deployment of business functionality.**

Next: Benefits and more

- you will learn about the benefits of using a microservices architecture and the challenges of architecture changes.
- Differences between a monolithic architecture and a microservices architecture.
- microservices patterns, messaging, and testing.
- deploying microservices and examine the architectured infrastructure with cross-cutting concerns.

Team Autonomy

- Companies constantly need to deliver more features in production in the fastest way possible.
- By separating areas of concern in the architecture, development teams can have autonomy from other teams.
- This autonomy allows teams to develop and deploy at a pace different than others.
- Time to market is essential for most companies.
- It also **allows** for (*but does not require*) different teams to leverage different programming languages
- Monoliths typically require the whole code base to be in the same language.

Team autonomy (2)

- Because microservices are distinctly different applications, they open the door to using different languages, allowing flexibility in fitting the tool to the task at hand.
- With data analytics, for example, Python is the most common programming language used and works well in microservice architectures.
- Mobile and front-end web developers can leverage languages best suited for those requirements, while C# is used with back-end business transaction logic.

Team Autonomy (3)

- Clients need to know how to call these services with details such as HTTP verb and payload model, as well as the return data model.
- There is an API specification available to help guide the structure of your API.
- OpenAPI Initiative (https://www.openapis.org/) for more information. (Swagger)

Service Autonomy

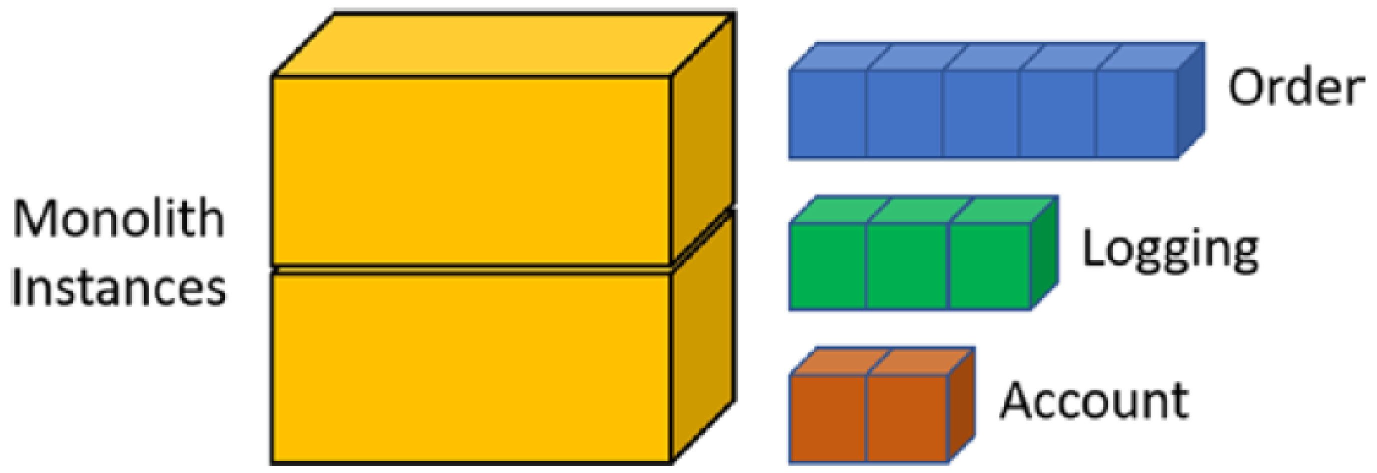
- Separation of concerns
- Single Responsibility Principle
- No microservice should have more than one reason to change.
- For example, an Order Management microservice should not also consist of business logic for Account Management.
- By having a microservice dedicated to specific business processes, the services can evolve independently.
- With loose coupling between microservices, you receive the same benefits as when applied at the code level.
- Upgrading microservices is easier and has less impact on other services.
- Features and business processes to evolve at different paces.

Service Autonomy (2)

- Individual resiliency and availability needs.
- For example, the microservice handling credit card payment has a higher availability requirement than handling account management.
- Clients can use retry and error handling policies with different parameters based on the services they are using
- Deployment of microservices is also a benefit of service autonomy.
- As the services evolve, they release separately using “Continuous Integration/Continuous Deployment” (CI/CD) tools like Azure DevOps, Jenkins, and CircleCI.

Scalability

- allows for the number of instances of services to differentiate between other services and a monolithic application
- By utilizing a microservice architecture, the applications can leverage servers of diverse sizes.
- One microservice may need more CPU than RAM, while others require more in-memory processing capabilities. Other microservices may only need enough CPU and RAM to handle heavy I/O needs

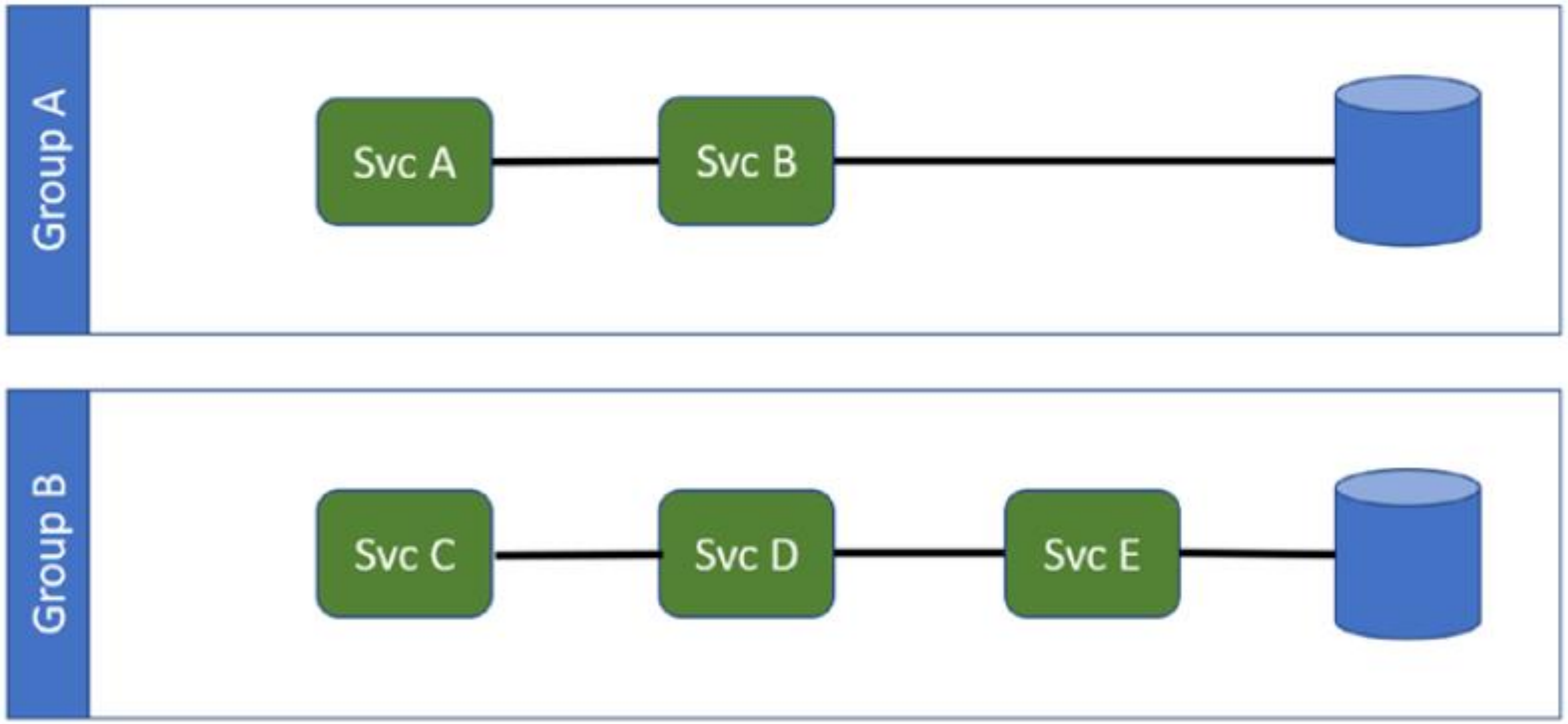


Fault Isolation

- Fault isolation is about handling failures without them taking down an entire system
- When a monolith instance goes down, all services in that instance also go down.
- no isolation of services exists when failures occur
- Several things can cause failure:
  - Coding or data issues
  - Extreme CPU and RAM utilization
  - Network
  - Server hardware
  - Downstream systems

Fault Isolation (2)

- Example of oversimplification of services dependent on other services and a dependency on a data store



Fault Isolation (3)

- Always consider microservices ephemeral
- This may be from prolonged CPU or RAM usage exceeding a threshold.
- Orchestrators like Kubernetes will “evict” a pod that contains an instance of the microservice in those conditions.
  - This is a self-preservation mechanism, so a runaway condition does not take down the server/node.
- An unreasonable goal is to have a microservice with an uptime of 100% or 99.999% of the time.
- If a monolithic application or another microservice is calling a microservice, then retry policies must be in place to handle the absence or disappearance of the microservice.
- This is no different than having a monolithic application connecting with a SQL Server.
- Retry policies in a **circuit breaker pattern** help tremendously in handling issues when calling microservices. (http://www.thepollyproject.org)

Data Autonomy

- Data integrity is crucial to the business
- Microservices incorporate loose coupling, so changes deploy independently.
- Most often, these changes also contain schema changes to the data.
- New features may require new columns or a change to an existing column, as well as for tables.
- The real issue occurs when the schema change from one team impacts others.
- This, in turn, requires the changes to be backward compatible.
- Additionally, the other team affected may not be ready to deploy at the same time.
- This isolation is another factor that encourages quicker time to production for the business.
  - Starting a new feature with a new microservice with new data is great. Of course, that is easy to implement.

Challenges

- Migrating to a microservice architecture is not pain-free and is more complex than monoliths
- Even with a small microservice, it may take several iterations to get to exactly what you need.
- And you may need to complete many rounds of refactoring on the monolith before you can support relocating functionality to a microservice.
- If coming from a monolith, you will need to make code changes to communicate with the new microservice instead of just a simple method call.
- Communicating with microservices requires calls over a network and, most often, using a messaging broker. You will learn more about messaging later

Challenges (2)

- The size of the monolithic applications and team sizes are also factors.
- Small applications, or large applications with small teams, may not see the benefits
- Many companies are not ready to take on the challenges and simply host monolithic applications on additional servers and govern what business logic they process
- Servers are relatively cheap, so spreading the processing load is usually the easiest "quick" solution.
- Developers may push back on the idea of microservice development.
- There is a large learning curve for the intricate details that need to be understood.
  - And many developers will remain responsible for various parts of the monolithic applications, so it may feel like working on two projects simultaneously.
- There will be the ongoing question of quality vs. just getting something to production.
  - Cutting corners will only add code fragility and technical debt and may prolong a successful completion

Microservice Beginning

- “tightly coupled”
  - With a primary system needing to work with other systems, there arose an issue of the primary system being required to know all the communication details of each connected system.
- Service-Oriented Architecture (SOA) aimed to eliminate the hassle and confusion.
- Enterprise Service Bus (ESB), introduced in 2002, was used to communicate messages to the various systems. An ESB provides a way for a “Publish/Subscribe” model in which each system could work with or ignore the message as they were broadcasted. Security, routing, and guaranteed message delivery are also aspects of an ESB.
  - When needing to scale a service, the whole infrastructure had to scale as well.
- Microservices
  - each service can scale independently.
  - By shifting from ESB to protocols like HTTP, the endpoints become more intelligent about what and how to communicate.
  - The messaging platform is no longer required to know the message payload, only the endpoint to give it to. “

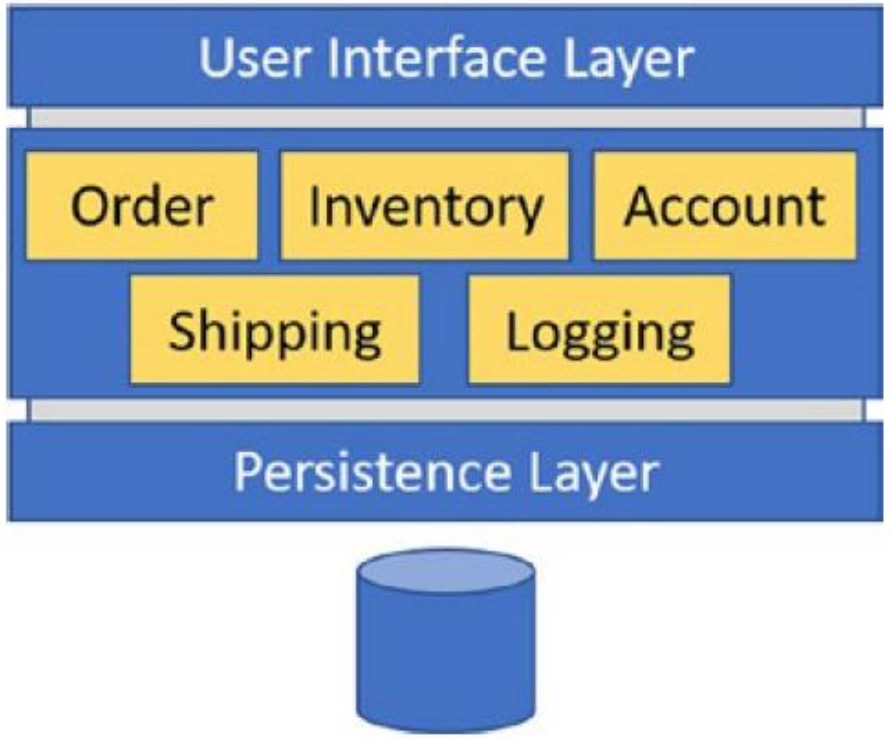
Microservice Beginning (2)

- Cost is a huge factor, but so are the programming languages and platforms.
- more than a handful of languages like C#, Python, and Node are great for microservices.
- Platforms like Kubernetes, Service Fabric, and others are vastly capable of maintaining microservices running in Docker containers.

Beware

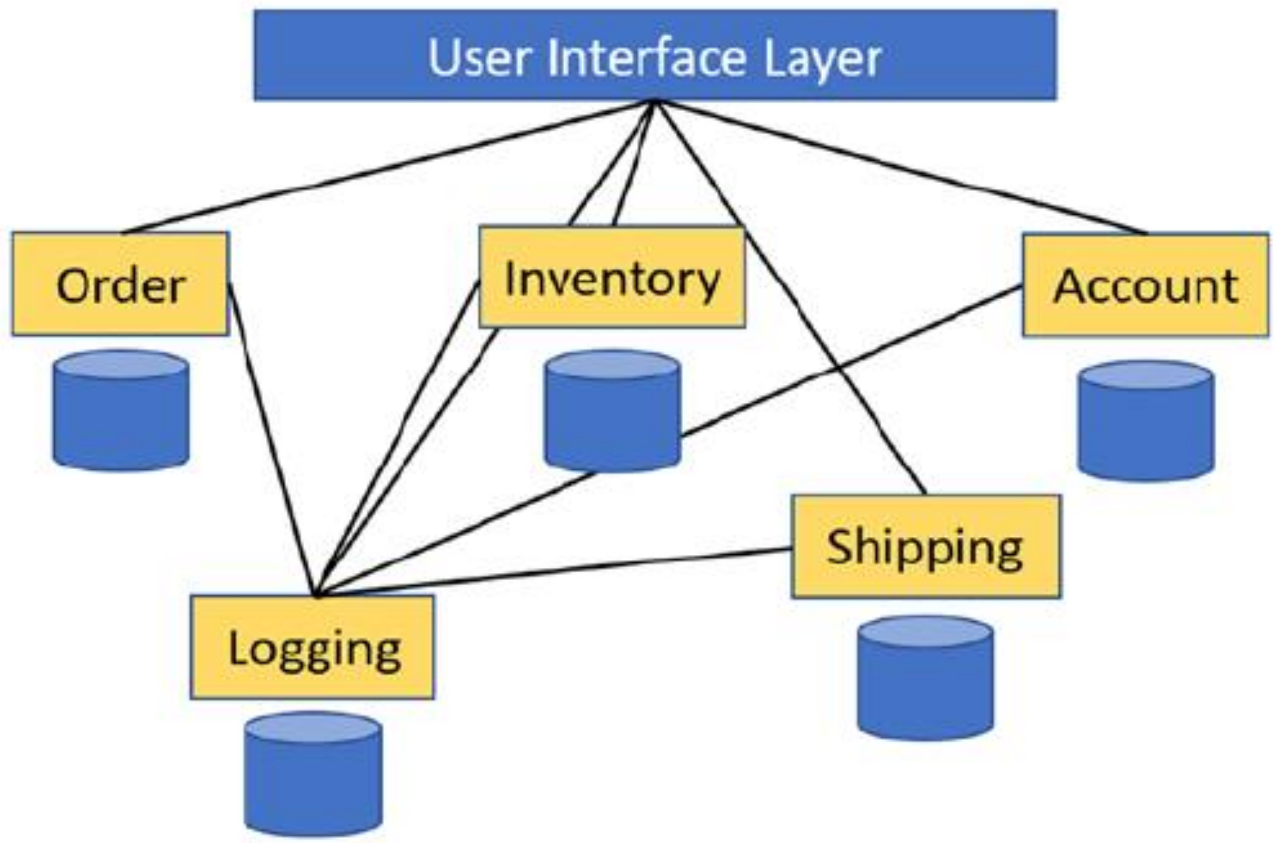
- With the ever-increasing demand for software programmers, there also exists the demand for quality.
- It is way too easy for programmers to solve simple problems and believe they are “done.”
- In reality, quality software is highly demanding of our time, talents, and patience.
- Just because microservices are cheaper and, in some cases, easier to create, they are by no means easy.

Architecture Comparison



- most microservices stem from a monolithic application, we will compare the two architectures.
- A monolith, in the simplest term, is a single executable containing business logic
  - This includes all the supportive DLLs. When a monolith deploys, functionality stops and is replaced. Each service (or component) in a monolith runs “in process.” This means that each instance of the monolith has the entire code base ready for instantiation

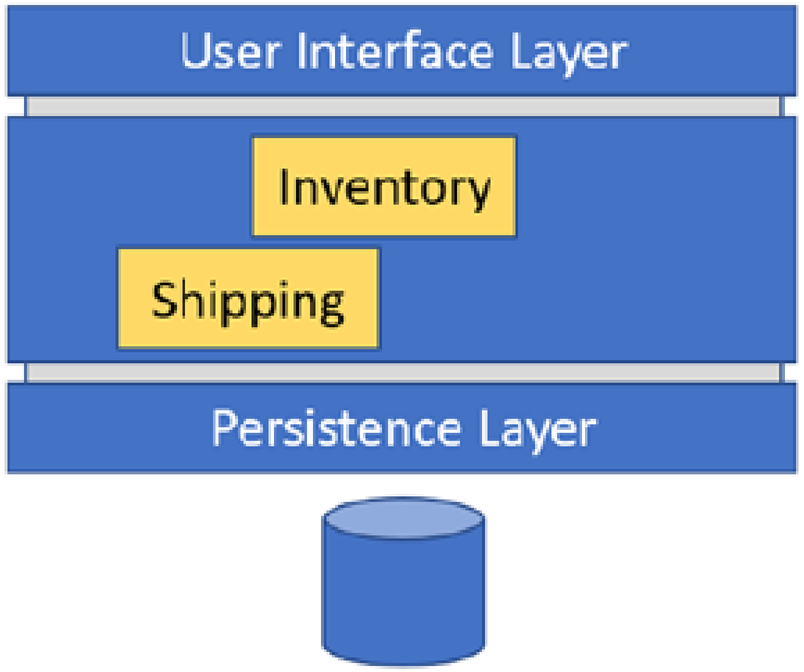
Microservice architecture

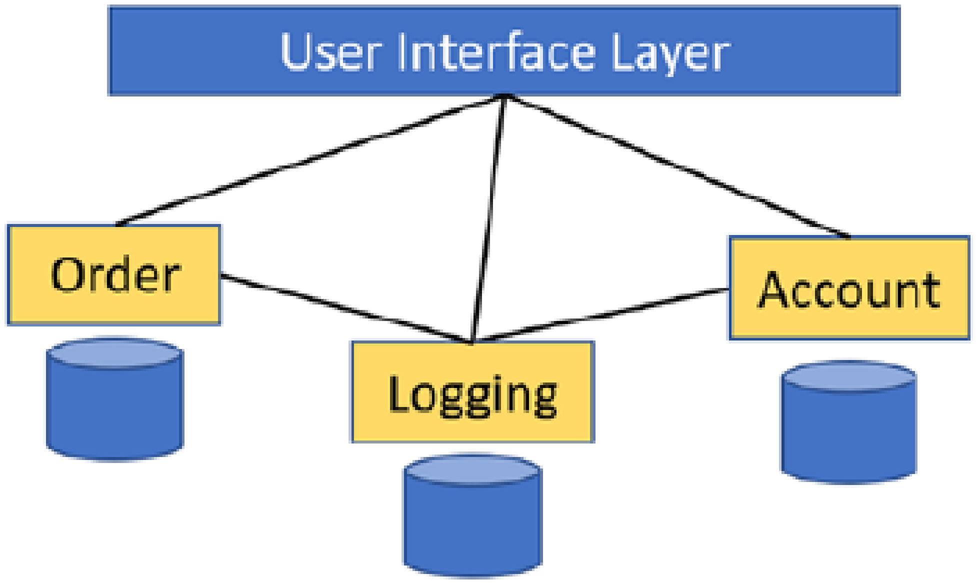


- business logic is separated out into out-of-process executables.
- fault isolation is gained with this separation.
- If, for example, shipping was unavailable for a while, orders would still be able to be taken.

Realistic hybrid architecture

- When venturing down the path of creating microservices, start small.



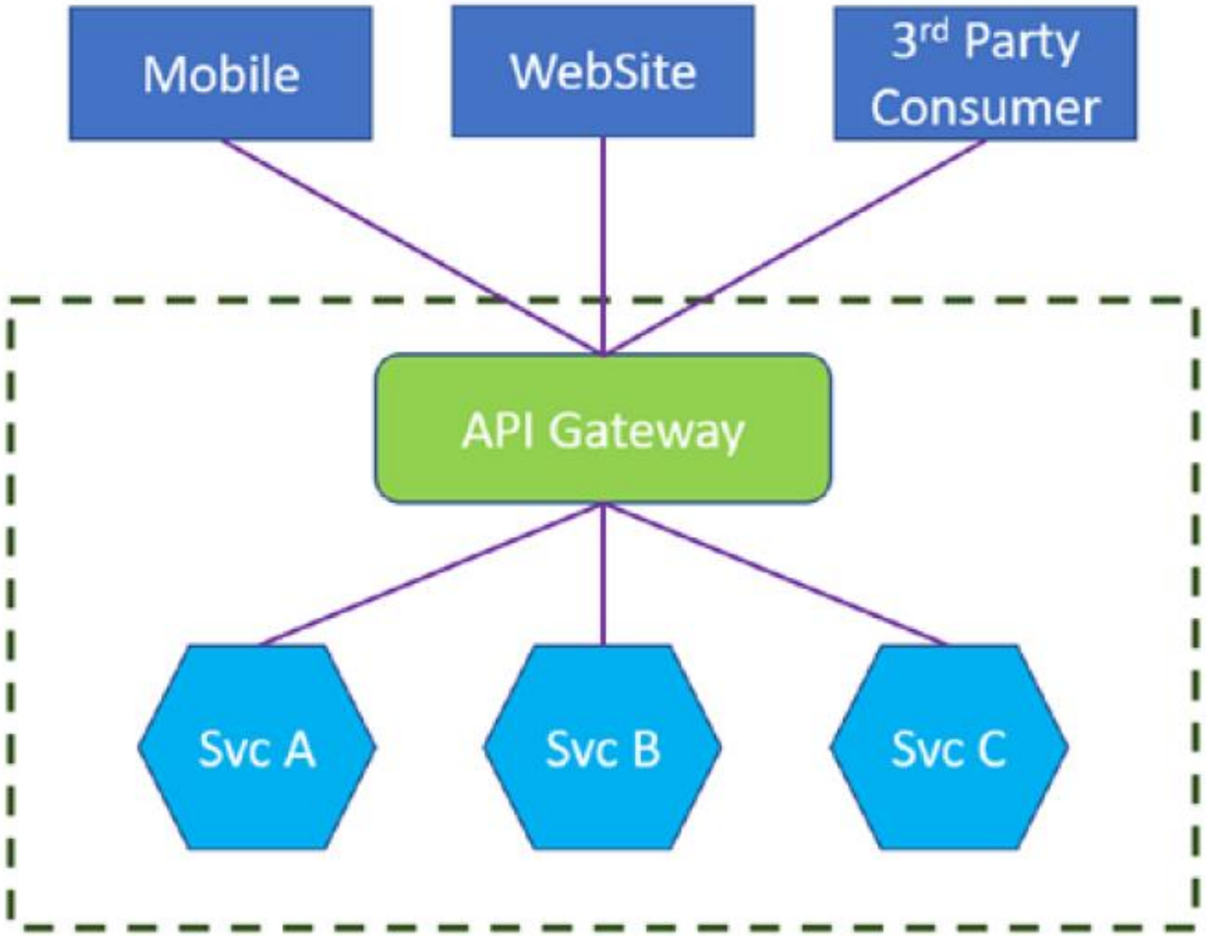


Microservice Patterns

- Every microservice architecture has challenges such as accessibility, obtaining configuration information, messaging, and service discovery.
- There are common solutions to these challenges called patterns.
- Various patterns exist to help solve these challenges and make the architecture **solid**.

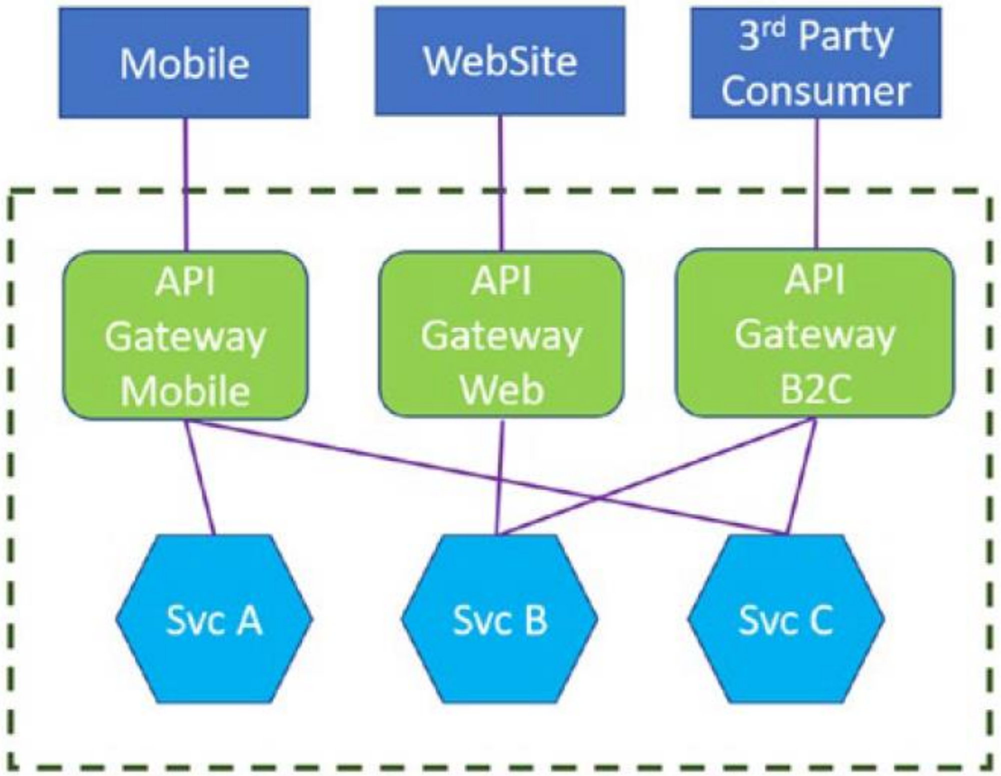
API Gateway/BFF

- The API Gateway pattern provides a single endpoint for client applications to the microservices assigned to it.



number of client applications increase

- demands from those client applications may grow.
- separation should be done to split client applications apart by using multiple API Gateways.
- Design pattern Backends for Frontends (BFF), helps with this segregation.
- There are multiple endpoints, but they are designated based on the types of clients being served.



Multiple API Gateways are created, and microservices split between them. This would help with another precaution where the API Gateway can be a **bottleneck** and may add to any latency issues.

External Configuration Store

- Nearly all microservices will need configuration information, just like monoliths.
- Updating information across all running instances would be overwhelming.
- using the External Configuration Store pattern provides a common area to store configuration information.
- This means there is one source of the configuration values.
  - The configuration information could be stored in a data store such as SQL Server or Azure Cosmos DB.
  - Different Configuration Stores allowing the same code to work in Dev vs. Staging or Production.
- challenge here is knowing when to get the settings.
- The code can either get all the settings at startup or as needed. If only retrieving at startup, then the behavior of the microservice will not change until restarted.

Messaging

- With monolithic applications, methods simply call other methods without the need to worry about where that method resides
- three main aspects of communication with microservices
- “why” there is communication with microservices
- “what” is the data format
  - Not all contents in the messages are the same, and they will vary based on purpose
- “how” messages are sent to endpoints (transport)

Business Process Communication

- There are multiple ways of communicating
- RPC
  - The synchronous, “direct” way is for when a request needs an immediate response
  - a microservice has a financial algorithm and responds with data based on the values passed into the request
  - The client (monolith client or another microservice) supplies the parameter values, sends the request, and waits for the response.
  - The business process does not continue until it has received an answer or an error.
  - This type of call is a Remote Procedure Call (RPC) as it is direct and synchronous, from the client to the service.
  - Using RPC should be limited in use.

Fire-and-Forget

- asynchronous call
- The client does not care if the microservice can complete the request.
- An example of this style is logging
- the need to continue processing outweighs the need to verify the microservice completed successfully.

Callback

- asynchronous call
- the microservice calls back to the client, notifying when it is done processing
- The business process continues after sending the request.
- The request contains information for how the microservice is to send the response. This requires the client to open ports to receive the calls and passing the address and port number in the request
- When passing a correlation ID in the request message and the microservice persisting that information to the response, the client can use the response for further processing.

Callback (2)

Barista

- You place your order for a freddo.
- The barista takes down the type of drink and your name. Then two parallel processes occur.
- One is the process of creating the drink.
- The other is processing payment.
- Only after the drink is ready and payment succeeds are you called. “Sean, your latte is ready.”
- Notice there are two pieces of information used as a correlation ID.
- The customer’s name and the drink name were taken at the time of the order and used to tell the customer the order has completed.

Callback (3)

- two asynchronous processes occurred:
- one to create the drink and the other to take payment.
- After those two processes were completed, another asynchronous process was initiated.
- I was notified the drink is ready and where to pick it up.
- This notification is called a “domain event.”

Pub/Sub

- asynchronous call style, Publish/Subscribe (Pub/Sub).
- listening on a message bus for messages about work to process
- The sender publishes a message for all the listeners to react on their own based on the message
    - newspaper company. It publishes a single issue daily to multiple subscribers. Each subscriber has the opportunity to read and react independently to the same content as all other subscribers

Message Format

- The format of the data in the messages allows your communication to be cross language and technology independent.
- Interoperability
- Text
- Binary
- small to medium size messages, JSON and XML are the most used formats
- message size increases, the extra information can increase latency.

Message Format (2)

- Utilizing a format such as Google's Protocol Buffers (https://developers.google.com/protocol-buffers/)
- Avro by Apache (https://avro.apache.org/), the messages are sent as a binary stream
- These are efficient with medium to large messages because there is a CPU cost to convert content to binary. Smaller messages may see some latency.
- Most of the time, using JSON is fine given the size of the payloads.

Transport

- multiple protocols available such as HTTP, TCP, gRPC, and Advanced Message Queuing Protocol (AMQP).
- Generally, HTTP is used, but using TCP web sockets is an alternative
- Representational State Transfer (REST) is an architectural style that is quite common today when creating Web APIs and microservices.
  - For example, to retrieve data, the call uses the HTTP verb GET. To insert, modify, or delete data, the HTTP verbs POST, PUT, UPDATE, and DELETE are used. The specific verb is declared in the code of the service endpoints.

Transport (2)

- microservices calling other microservices.
- There is an inherent risk of latency
- An RPC technology called “gRPC Remote Procedure Call” (gRPC) is better suited for the interprocess communication. gRPC is a format created by Google using, by default, protocol buffers.
- Where JSON is a string of serialized information, gRPC is a binary stream and is smaller in size and, therefore, helps cut down latency.

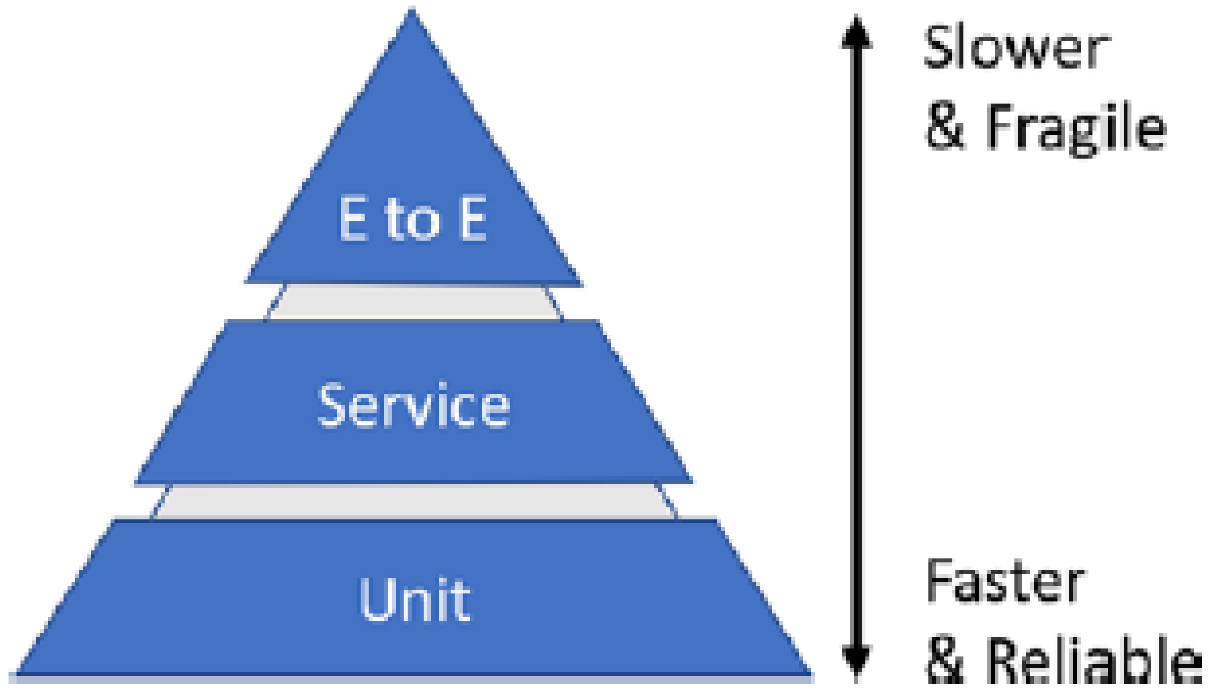
Transport (3)

- For asynchronous calls, messages are sent using a message broker such as RabbitMQ, Redis, Azure Service Bus, Kafka, and others.
- AMQP is the primary protocol used with these message brokers. AMQP defines publishers and consumers.
- Message brokers ensure the delivery of the messages from the producers to the consumers.
- With a message broker, applications send messages to the broker for it to forward to the receiving applications.

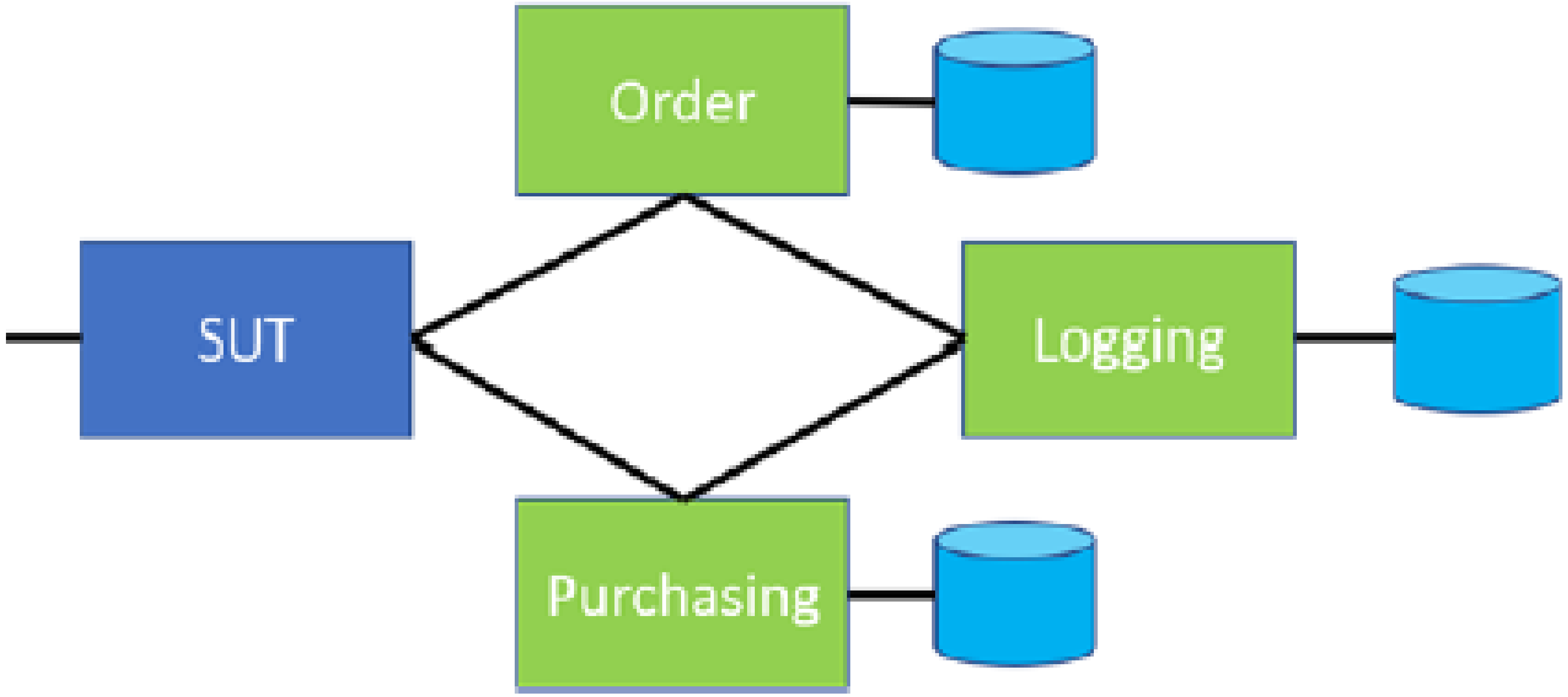
Testing

- A huge problem is when code performs incorrectly.
- The test pyramid is a visual representation of testing levels.
- The unit tests should be small and cover basic units of business logic.
- Service tests are for individual microservices.
- end-to-end tests are the slowest and most unreliable as they generally depend on manual effort and the least amount of automation.

Testing Pyramid



End to end testing



System Under Test (SUT)

Automation

- Continuous Integration/Continuous Deployment (CI/CD) pipeline should be considered a must-have
- It is highly recommended to use a build step in your CI/CD pipeline that is performing the unit tests.
- Integration tests are generally much longer in execution time, so many companies do not add them to the CI/CD pipeline.

Deploying Microservices

- Microservices are independent applications.
- Although their code and business functionality may have come from a monolith, they now have a life of their own.
- Part of that independence is deployment.

Parameters:

- versioning tactics,
- wrapping in containers,
- hosting in orchestrators, and
- deployment pipelines.

Versioning

- As newer releases of microservices are deployed, versioning takes consideration.
- Leveraging a versioning semantic from SemVer (https://www.semver.org), there are three number segments that are used:
  - Major – version when you make incompatible API changes
  - Minor – version when you add functionality in a backward-compatible manner
  - Patch – version when you make backward-compatible bug fixes

Containers

- Containers allow for executables, their dependencies, and configuration files to be packaged together.
- Although there are a few container brands available, Docker is the most well known.
- Deploying microservices can be done straight to servers or more likely virtual machines.
- benefits from running containers
- easier to manage
- Orchestrators like Docker Swarm, Service Fabric, Kubernetes, and others manage containers with all the features just mentioned but also include features like network and security.

Cross-Cutting Concerns

- There are aspects that are not specific to microservices but do apply to the infrastructure.
- These are just as important as microservices. If a microservice became unavailable at 2 am, how would you know? When would you know? Who should know?
- These cross-cutting concerns (CCC) help you understand the system's health as a whole, and they are a part of it.
- Understanding the health of a system helps with capacity planning and troubleshooting.

Monitoring

- Evaluate good monitoring solutions.
- Prometheus (https://prometheus.io) is a great option, but it is not the only good one available.
- Prometheus is great, but it has a steep learning curve.
- Whatever you find, make sure you build in time to learn how to use it well.
- After data is captured, you will need something to display that information. Tools like Grafana (https://grafana.com) are for displaying captured metrics via Prometheus and other sources
- Very good solution also: Application Insights (Azure)

Health Check

- Implementing a form of health check provides you access to information in various ways.
- One of which is knowing the microservice is not dead and responds to requests.
- This is a “liveness probe.” The other is in what information goes back to the client.
- Consider adding an entry point that uses HTTP GET, for example, HTTP://{service endpoint}/health.

Health Check (2)

- Can connect to data store
- Can connect to next microservice hop, if there is one
- Consider returning necessary data for troubleshooting and incident reporting
- Consider returning a version that can be used to verify latest version has been deployed
- There are .NET libraries for Prometheus that capture data and are customizable.

Logging

- Logging information from microservices is as vital as monitoring, if not more so.
- Monitoring metrics are great, but when **diagnosing fault** events, exceptions, and even messages of properly working calls, **logging** information is an absolute must.
- A centralized logging system provides a great way to accept information that may come from multiple systems.
- This is known as "Log Aggregation." Services like Splunk are 3rd party tools for log management.
- Microservices log information to stdout or stderr. This information is captured and sent to Splunk, where it can be correlated and viewed
- ***Loggly (https://www.loggly.com) is a logging system with many connectors for a variety of systems.***

Alerting

- With data monitored and captured, development teams need to know when their microservices have issues.
- The following is a list of example metrics you should create alerts for:
  - High network bandwidth usage
  - CPU usage over a threshold for a certain amount of time
  - RAM utilization
  - Number of inbound and outbound simultaneous calls
  - Errors (exceptions, HTTP errors, etc.)
  - Number of messages in a service broker’s Dead Letter Queue (DLQ)
- Tools like Grafana, Azure Log Analytics, and RabbitMQ have their alerting features
- An additional option is using webhooks to send information to a Slack channel. This way, there is a message that multiple people can see.

Summary

- We just went over a lot of information with a large, broad stroke.
- The topics mentioned are just the tip of the knowledge iceberg compared to the depth of knowledge encompassing microservice architecture.

ASP.NET Core Overview

- history and features of .NET, the runtime, and ASP.NET Core, the web server.

A Brief History of .NET

- .NET Framework in 2000
- .NET Core has become .NET: a unified platform for building great modern apps.
- runs equally well on x64 and ARM processors.
- You can install it on a machine just as easily as embed it in a Docker container.
- One can bundle .NET into a single executable or leverage the .NET runtime installed on the machine.

.NET (2)

- the next version of .NET Core would not be called .NET Core 4.0 but rather would be called .NET 5.
- This one unified base class library and runtime can now power cloud properties, desktop apps, phone apps, IoT devices, machine learning applications, and console games: “One .NET.”
- .NET 6.0 carries on this tradition
- “Current release” versions are supported for 18 months. .NET Core 2.2 is designated as a “Current release.” It was released in late 2018, so it’s support has already ended.
- “Long-term support” versions are supported for 3 years from their release date. .NET Core 3.1 is designated as an LTS release. It came out in 2019, so it will be supported until 2022.
- .NET 6 is designated as an LTS release. It shipped in November 2021, so it will be supported until 2024.
- https://dotnet.microsoft.com/platform/support/policy/dotnet-core.

Stacks

- On the web server stack, we have ASP.NET Core MVC, Web API, and SignalR.
- Added to this, we now have gRPC and Blazor, a web-component framework that runs in the browser on WebAssembly.
- On the desktop stack, we have WinForms, including a designer new in .NET Core 5 and WPF.
- Added to this, we have XAML-based apps built-in Xamarin for use on mobile devices, and MAUI is a new cross-platform UI toolkit previewing during the .NET 6 timeframe.
- Unfortunately, WCF and WebForms have not been ported
- There’s some great work that allows an old WebForms app to run on Blazor.
- If you’re in need of WCF, CoreWCF was donated to the .NET Foundation and may meet the need.

Installing .NET 6.0 and ASP.NET Core

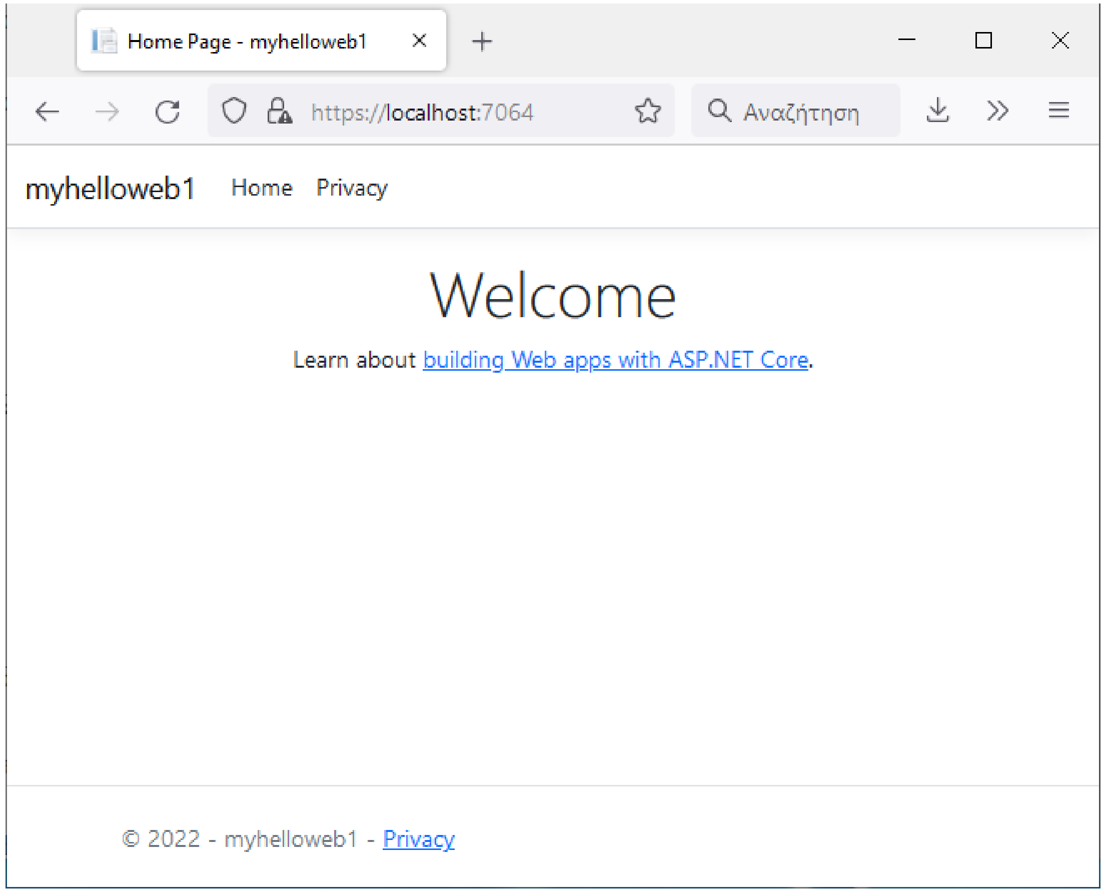
- https://dotnet.microsoft.com/download
- let’s build an ASP.NET Core project. There’s a lot more going on in this template.

```
mkdir helloweb
cd helloweb
dotnet new mvc --name myhelloweb
```

- Like the console app, this scaffolds out a new ASP.NET Core website with controllers,
- views, and JavaScript and CSS files. You can run the website in the same way:
- dotnet run

MVC .NET ports

- As the website starts up, note the web address the site runs on. The HTTP port
- will be a random unused port between 5000 and 5300, and HTTPS will be randomly
- selected between 7000 and 7300. For example, the console output could list https://
- localhost:7000/. Copy the URL from your console, open a browser, and paste the
- URL. You'll see your sample website running.



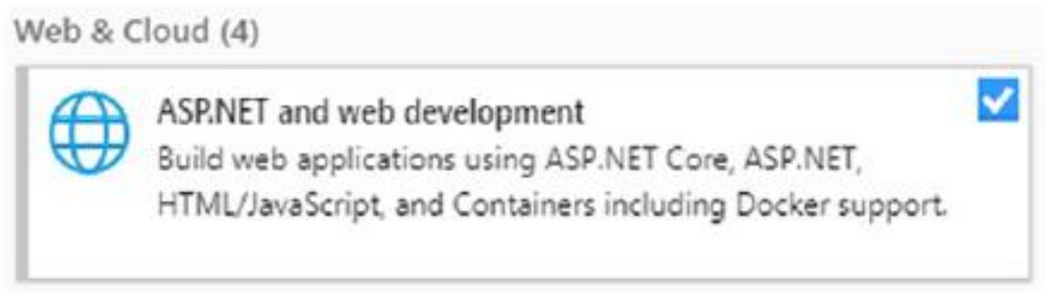
Welcome

Learn about building Web apps with ASP.NET Core.

© 2022 - myhelloweb1 - Privacy

Visual Studio

- https://visualstudio.microsoft.com/vs
- Community Edition


Web & Cloud (4)

ASP.NET and web development ✓
Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker support.

Figure 2-2. *ASP.NET install component*

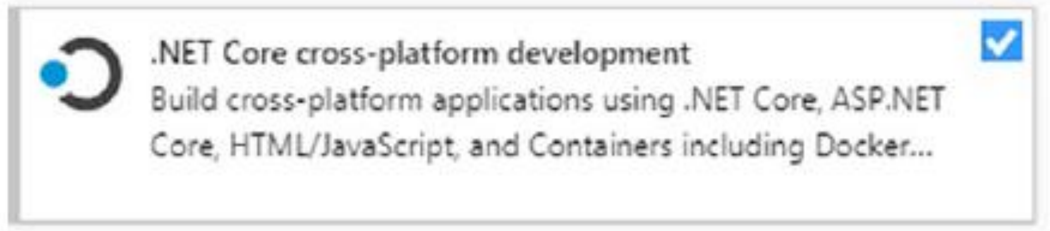

.NET Core cross-platform development ✓
Build cross-platform applications using .NET Core, ASP.NET Core, HTML/JavaScript, and Containers including Docker...

Figure 2-3. *.NET Core cross-platform install component*

MVC at a Glance

- MVC has three main components: the Model, View, and Controller.
- Combined with ASP.NET there is a fourth component, Routing
- Convention over Configuration,
- Model Binding,
- Anti-Forgery Token

MVC (2)

- The MVC pattern is a wheel of steps beginning with the request and ending with the response.
- First, the URL is parsed into sections and flows into the router.
- The router looks at the URL pieces and selects the correct controller and action (class and method) to satisfy the request.
- The controller’s action method is responsible for calling into the rest of the application to process the data, harvest the results, and pass a model (class) to the view.
- The view is a templating engine, mapping HTML fragments together with the data to produce an HTTP response.
- The action method could also choose to forgo the view for API requests, choosing instead to hand data to a serialization engine that might produce JSON, XML, or gRPC.
- Once the response is constructed, the HTTP headers are attached, and the data is returned to the calling browser or application.
- This MVC pattern is indeed more complex than a simple URL to File path system like we had with WebForms or Classic ASP.

Controller flexibility

- The controller’s action can make choices.
- Does the data requested not exist? Then let’s send them to the not found page instead of the data page.
- Would they like JSON data instead of an HTML page? Let’s forgo the view and choose a response method that serializes data instead.

Convention over Configuration

- Convention over Configuration refers to how ASP.NET MVC uses naming conventions to tie URLs to controller classes and methods, controller methods to views and URL segments and post bodies to C# classes and URL segments and post bodies to C# classes.
- It is not required to have every view programmatically added to any list or association.

Routing

- As a browser or service makes an HTTP connection, the URL pattern matching determines which controller and action receive the request.
- Using a default template, the Home controller is created and handles default requests.
- In the URL, the controller is defined right after the domain name, or port number if one is specified.
- For example, with the URL http://localhost/Home/Index, the controller class name is Home and the action method name is Index.
- The actions are the methods in the controller class that receive the requests, execute any business logic, and return information or just status

Routing (2)

- With ASP.NET Core, most typically the routes are configured with attributes on each
- class. Let's look at an example controller inside an ASP.NET Core MVC project:

```
[Route("[controller]")]
public class CustomerController : Controller
{
 [Route("get/{id}")]
 public ActionResult GetCustomer(int id)
 {
  return View();
 }
}
```

- Here the Route attribute at the top specifies that if the URL starts with https://some-site/
- Customer, it should run methods within this class. Next, the Route attribute on the method maps to the next part of the URL. If the URL is https://some-site/Customer/get/7, then the GetCustomer() method should be called.

Endpoints

- Beginning in ASP.NET Core 2.2, you can use endpoints instead of the route table.
- Endpoint mapping allows for routing rules to be defined without impacting other web frameworks, meaning that it is possible to leverage SignalR along with MVC, Razor, and others without them colliding in the middleware pipeline.

```
app.UseEndpoints(endpoints => {
endpoints.MapControllerRoute(name: "default",
pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

- Adding a Routing Attribute to the action helps control the routing logic as well.

Routing Example

- In the following example, the Index method is called with any of these URLs: /, /home/, or /home/index. The use of routing attributes also allows for method names not to match. Using the URL /home/about, in this example, will call the ShowAboutPage method.

```
[Route("")]
[Route("Home")]
[Route("Home/Index")]
public IActionResult Index()
{
return View();
}
[Route("Home/About")]
public IActionResult ShowAboutPage()
    {
    return View("About");
    }
```

- Based on Elsevier Book .NET 6 Pro Microservices 2022
- Based on Modul 7.11 course public slides by Paul Leis