Next Generation Cloud Leadership

Cloud Foundations – Part 2

Develop Intelligence

Allen R. Sanders
Senior Technology Instructor

Objectives

- Provide AVP & VP level management with the skills & knowledge they need to be more Cloud conversant both from a practical and strategic perspective
- Ultimate objective is for some participants to get certified as AWS Cloud Practitioners.

Plan-of-Attack

CLOUD FOUNDATIONS – PART 2

Achieve baseline understanding of Cloud and its role in accelerating digital transformation

- Monolith vs. microservices
- Microservices and containers overview
- Consumption model & efficiency metrics
- Cloud features, service models, and design
- Migration vs. modernization

Architecting for the Future

- When we architect and build an application at a “point in time”, we hope that the application will continue to be utilized to provide the value for which it was originally built
- Since the “only constant is change” (a quote attributed to Heraclitus of Ephesus), we have to expect that the environment in which our application “lives and works” will be dynamic
- Change can come in the form of business change (change to business process), the need to accommodate innovation and ongoing advancement in technology
- Often, the speed at which we can respond to these changes is the difference between success and failure

Architecting for the Future

In order to ensure that we can respond to change “at the speed of business”, we need to build our systems according to best practices and good design principles:

- ➢ Business-aligned design
- ➢ Separation of concerns
- ➢ Loose coupling
- ➢ Designing for testability

Business-Aligned Design

- Build systems that use models and constructs that mirror the business entities and processes that the system is intended to serve
- Drive the design of the system and the language used to describe the system based around the business process not the technology
- AKA Domain Driven Design
- Results in a system built out of the coordination and interaction of key elements of the business process – helps to ensure that the system correlates to business value
- Also helps business and technology stakeholders keep the business problem at the forefront

Separation of Concerns

- Break a large, complex problem up into smaller pieces
- Drive out overlap between those pieces (modules) to keep them focused on a specific part of the business problem and minimize the repeat of logic
- Logic that is repeated, and that might change, will have to be changed in multiple places (error prone)
- Promotes high cohesion and low coupling (which we will talk about in a minute)
- Solving the problem becomes an exercise in “wiring up” the modules for end-to-end functionality and leaves you with a set of potentially reusable libraries

Loose Coupling

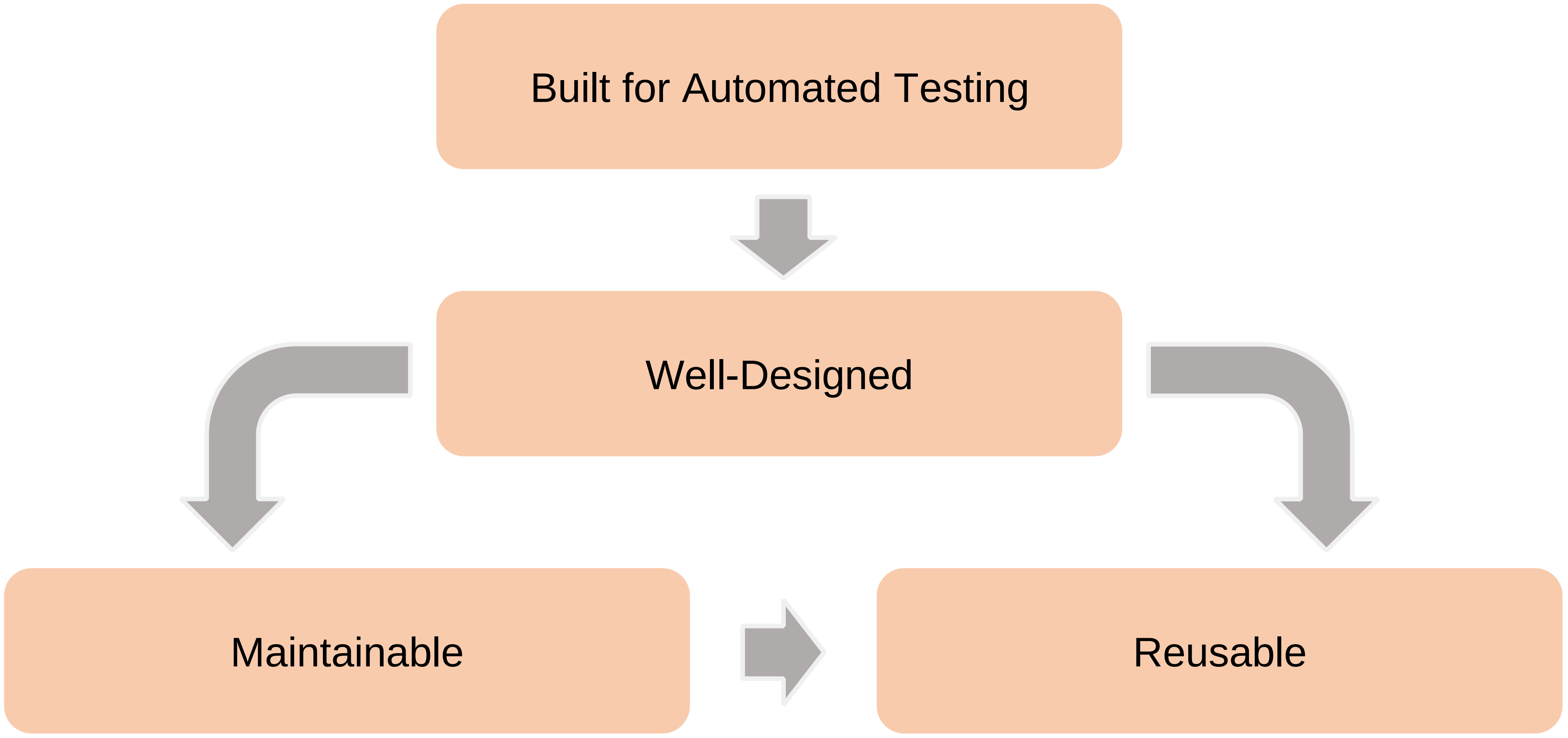
- Coupling between components or modules in a system causes problems, especially for maintaining the system over time
- System components that are tightly coupled, are more difficult to change (or enhance) – changes to one part of the system may break one or more other areas
- With coupling, you now must manage the connected components as a unit instead of having the option to manage the components in different ways (e.g. production scalability)
- Makes the job of unit testing the components more difficult because tests must now account for a broader set of logic and dependencies (e.g. tight coupling to a database makes it difficult to mock)

Designing for Testability

- Practicing the previous principles helps lead to a system that is testable
- Testability is important because it is a key enabler for verifying the quality of the system – at multiple stages along the Software Development Lifecycle (SDLC)
- When building a system, quality issues become more expensive to correct the later they are discovered in the development lifecycle – good architecture practices help you test early and often
- Ideally, testing at each stage will be automated as much as possible in support of quickly running the tests as and when needed

Designing for Testability

Testable code is…



Monolith vs. Microservices

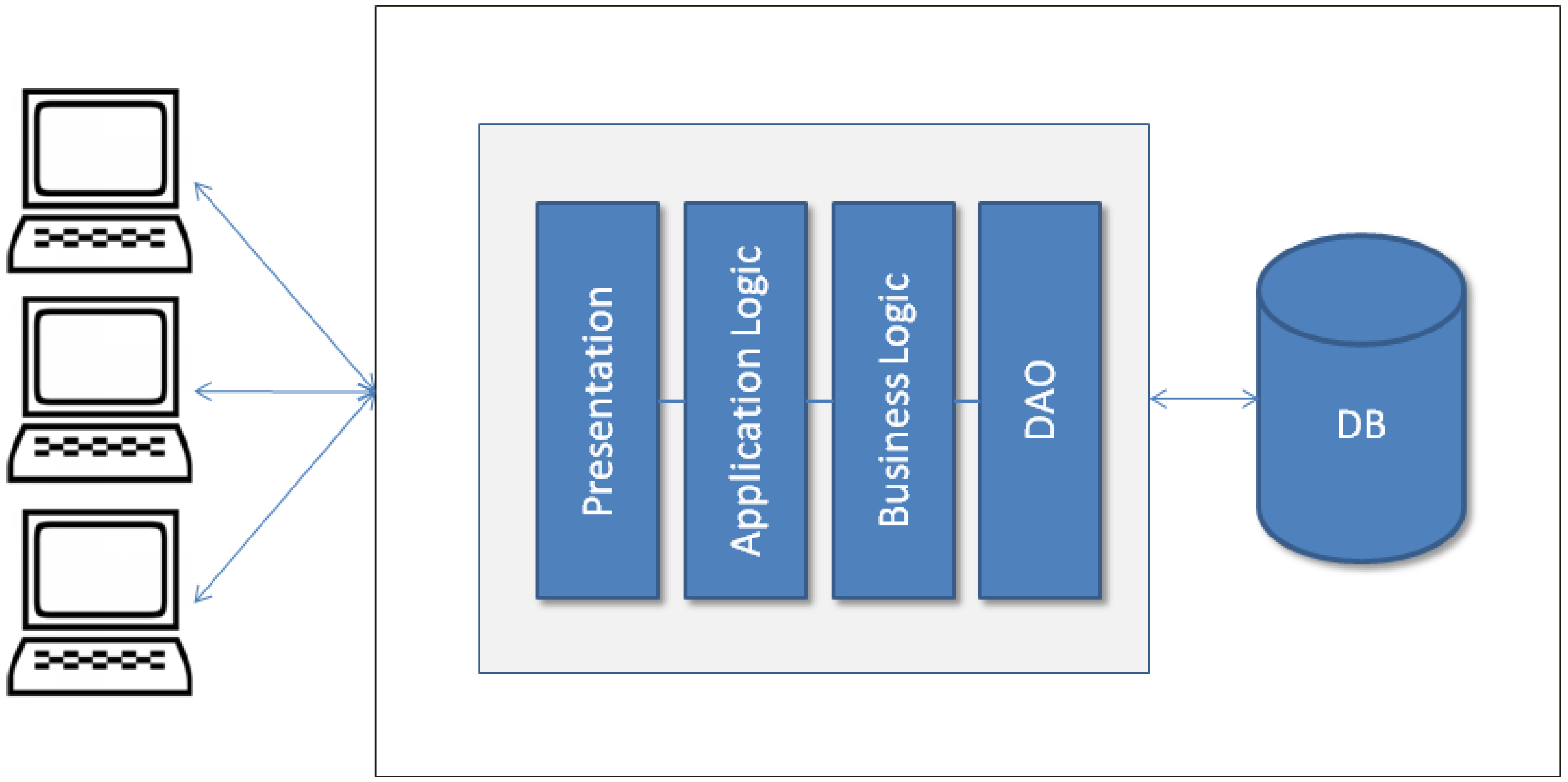
What are Microservices?

- An architectural style in which a distributed application is created as a collection of services that are:
  - Highly maintainable and testable
  - Loosely coupled
  - Independently deployable (by extension, independently-scalable)
  - Domain centric and organized around business capabilities
- The Microservices architecture enables the continuous deployment and delivery of large, complex distributed applications

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

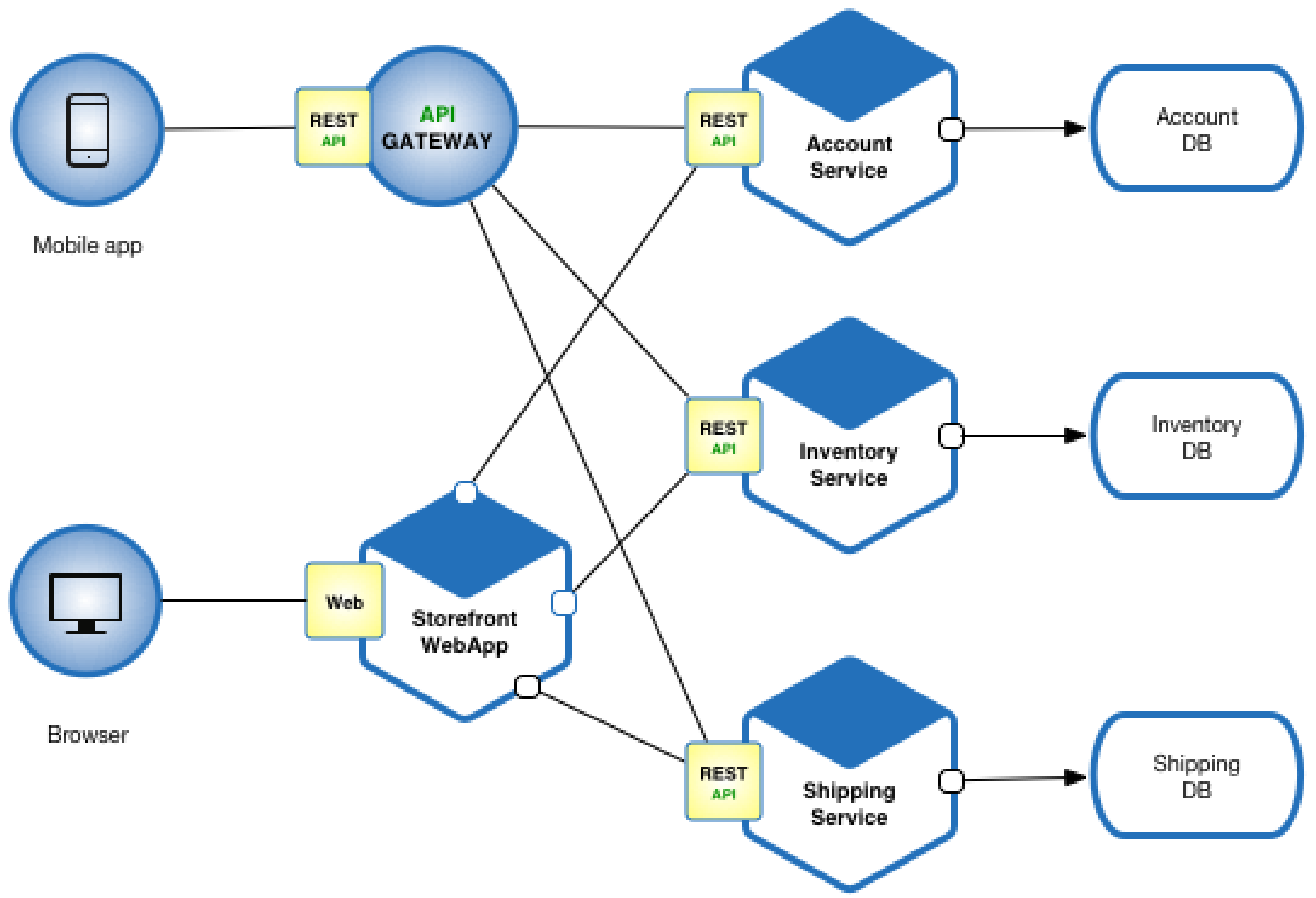
Monolithic Architecture Example



Moving to Microservices

- We have moved towards a refinement of “what is a service?”
- Even in SOA, we can create a service that does more than one thing – using it to solve an overall business problem, rather than one part of the actions needed to solve that problem
  - This was moving us back towards monolithic
- We arrive at a point where we want individual, purposeful services that do one thing and do it well – the microservice

Example Microservices Architecture



Microservices – Benefits vs. Costs

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

Microservices – Benefits vs. Costs

Costs:

- 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)

Microservices & Good Architecture

- With microservices, key concepts:
  - Cohesion (goal to increase)
  - Coupling (goal to decrease)

- SOLID principles & DDD (Domain Driven Design) lay the foundation for:
  - Clean, logical organization of components
  - Maintainability
  - Clear boundaries, encapsulation and separation of concerns in the components used to build out complex systems
  - Techniques that minimize coupling
  - Being “surgical” with our change

SOLID Principles

SOLID principles help us build testable code

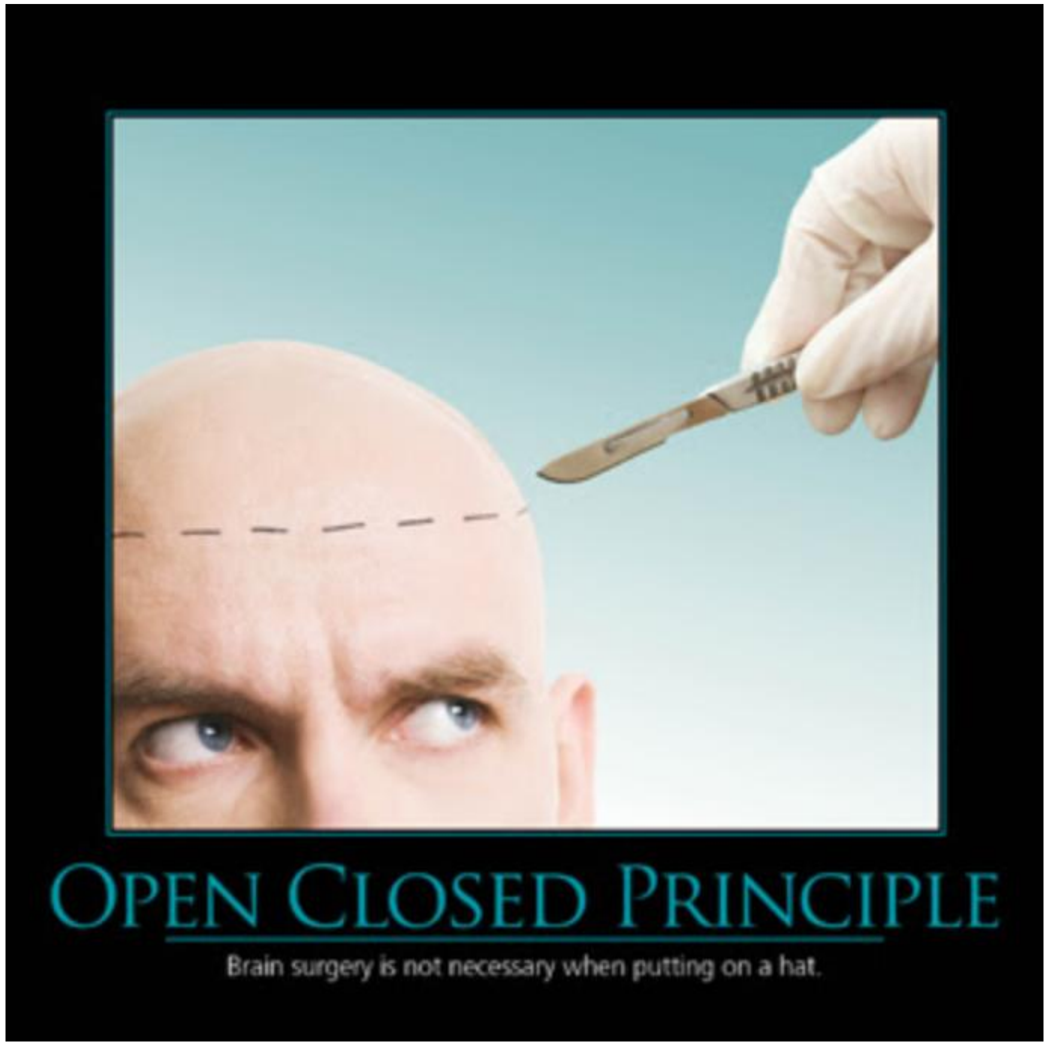
- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

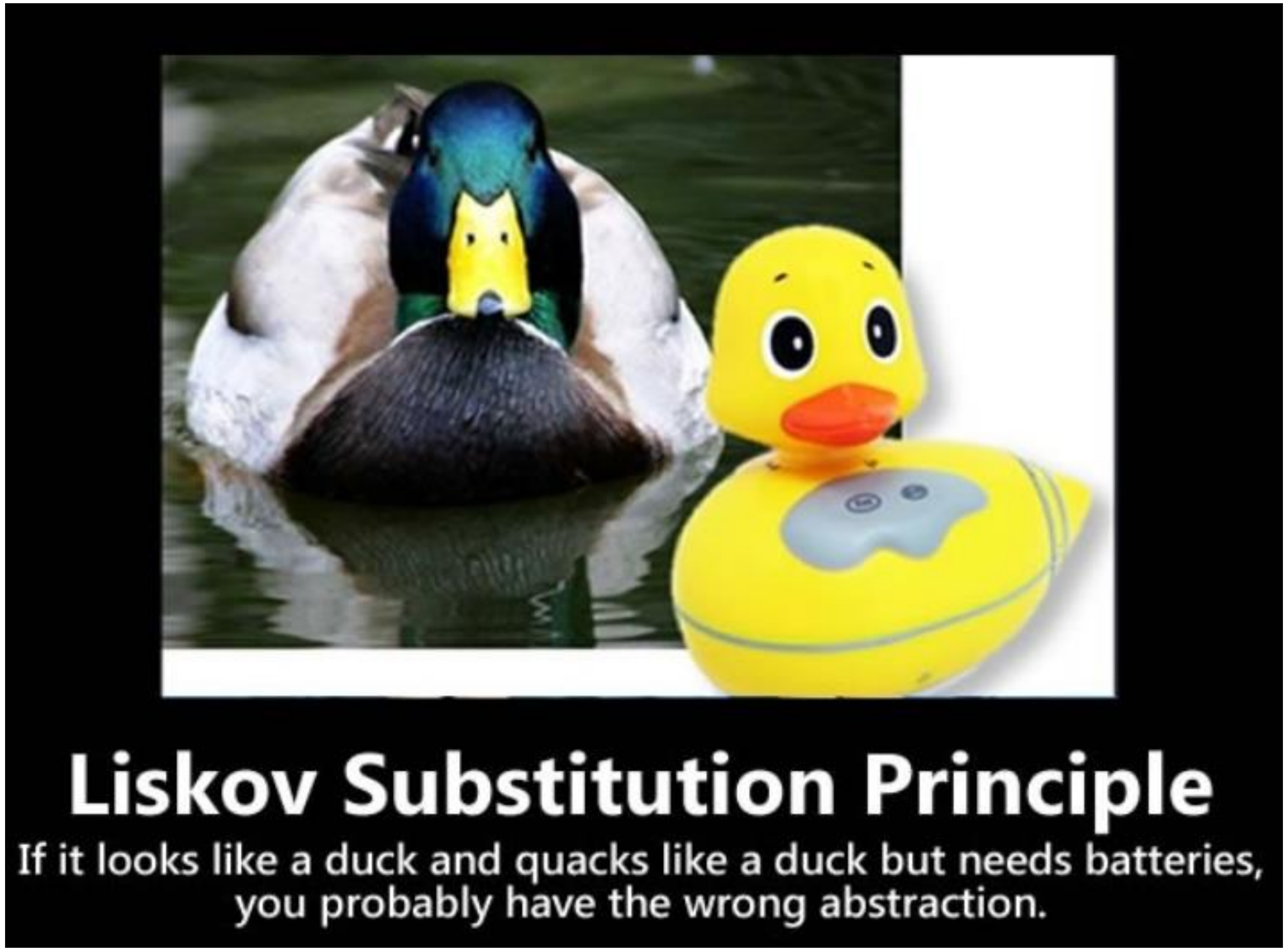
SINGLE RESPONSIBILITY PRINCIPLE

Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.

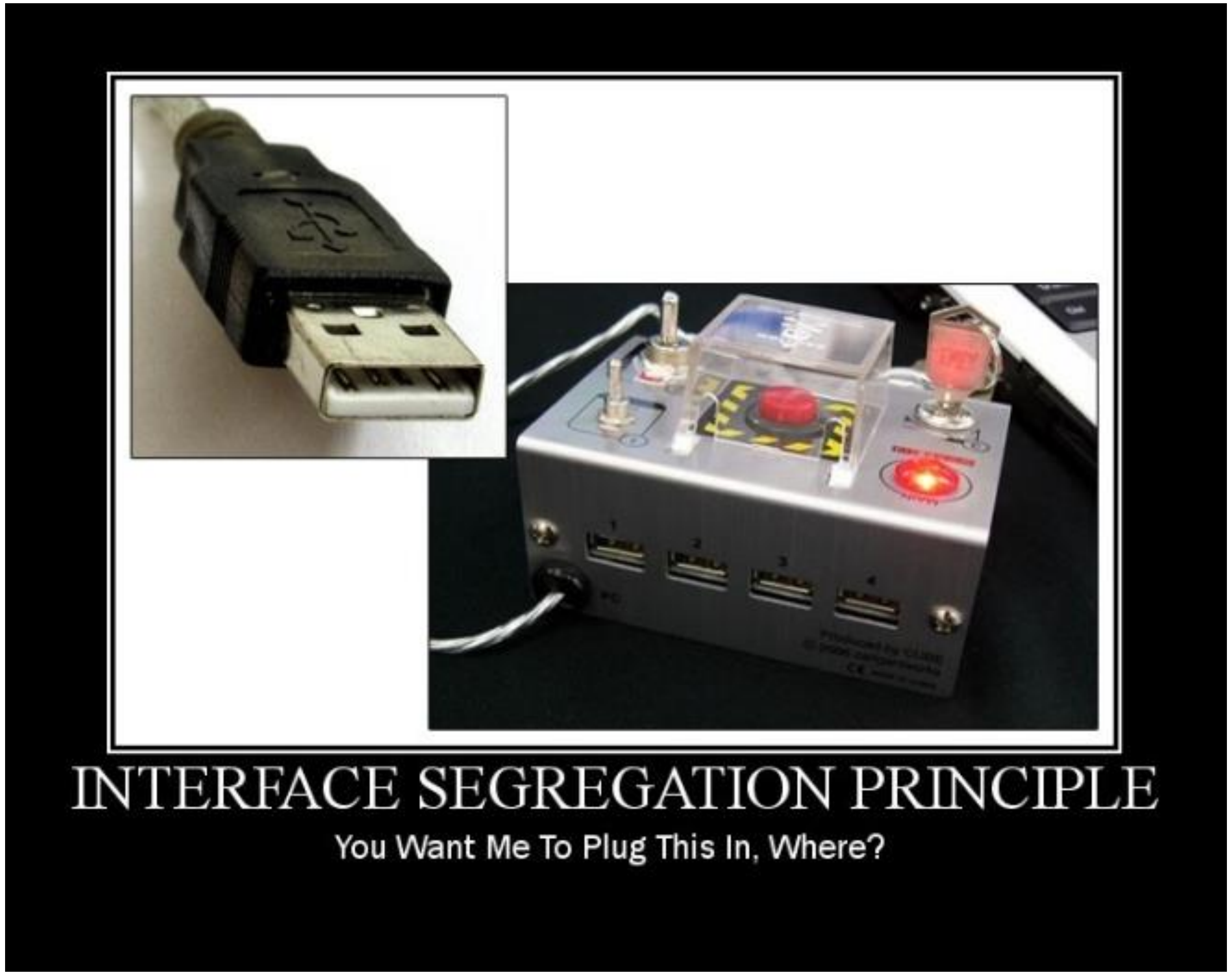
Open-Closed Principle (OCP)



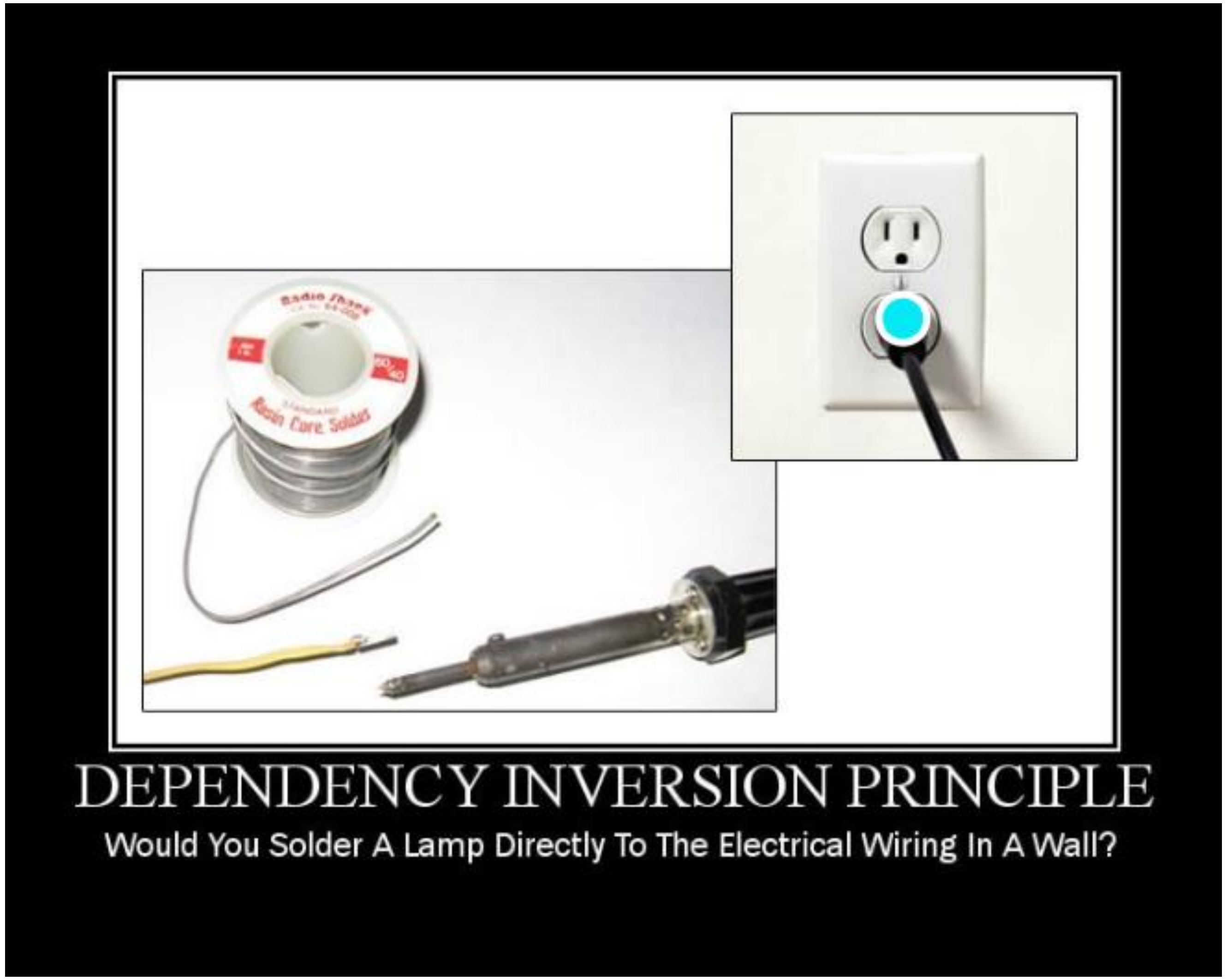
Liskov Substitution Principle (LSP)


Liskov Substitution Principle
If it looks like a duck and quacks like a duck but needs batteries,
you probably have the wrong abstraction.

Interface Segregation Principle (ISP)


INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

Dependency Inversion Principle (DIP)


DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

The Twelve-Factor App

https://12factor.net/

- Methodology for building SaaS applications for the Cloud
- Uses a *declarative* (vs. *imperative*) format for defining setup automation
- Maintains clean contract with the underlying operating system
- Drives toward consistent interface but pluggable implementation based on target OS / platform (see SOLID principles)

The Twelve-Factor App

- ➢ Supports modern cloud platforms
- ➢ Targets minimal divergence between development and production to enable continuous deployment (agility)
- ➢ Targets “elastic scalability” – the ability to dynamically automate scaling of a system to quickly adjust to demand while optimizing cost

Microservices and Containers

What is a Container?

- Loosely isolated environment used to package and deploy an application in a platform-agnostic manner

Evolution of Containers

- Started with client/server architecture – dedicated server for each app
- Moved to Virtual Machines – better, but had problems of OS overhead, etc.
- Containers mean we don't have to worry about specific hardware, OS, language, etc. – we just care about the code

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)

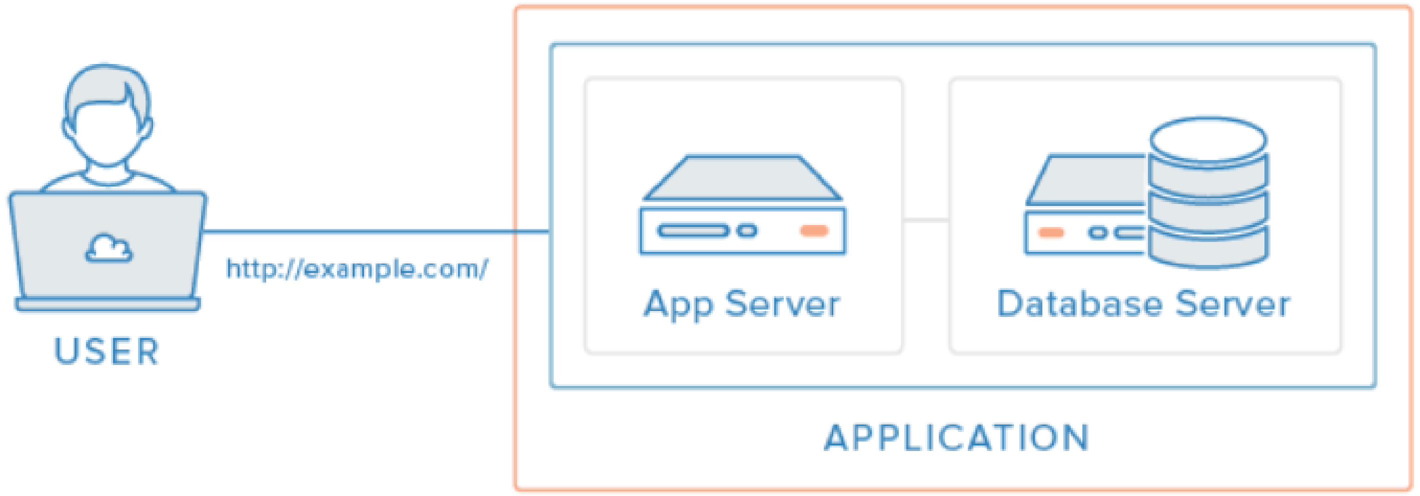
Containers

- 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

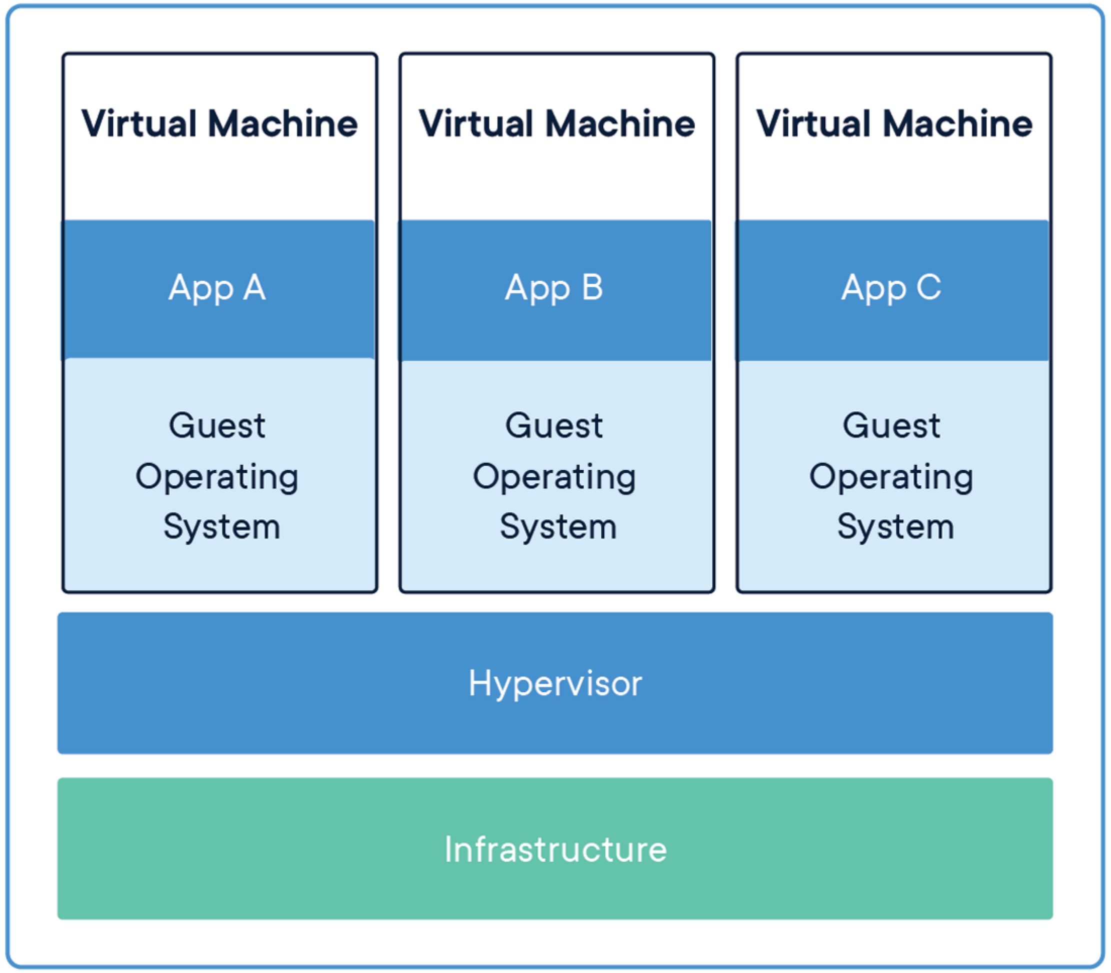
- Typically, much, much smaller than a VM which makes them lightweight, quickly deployable and quick to “boot up”

- 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)

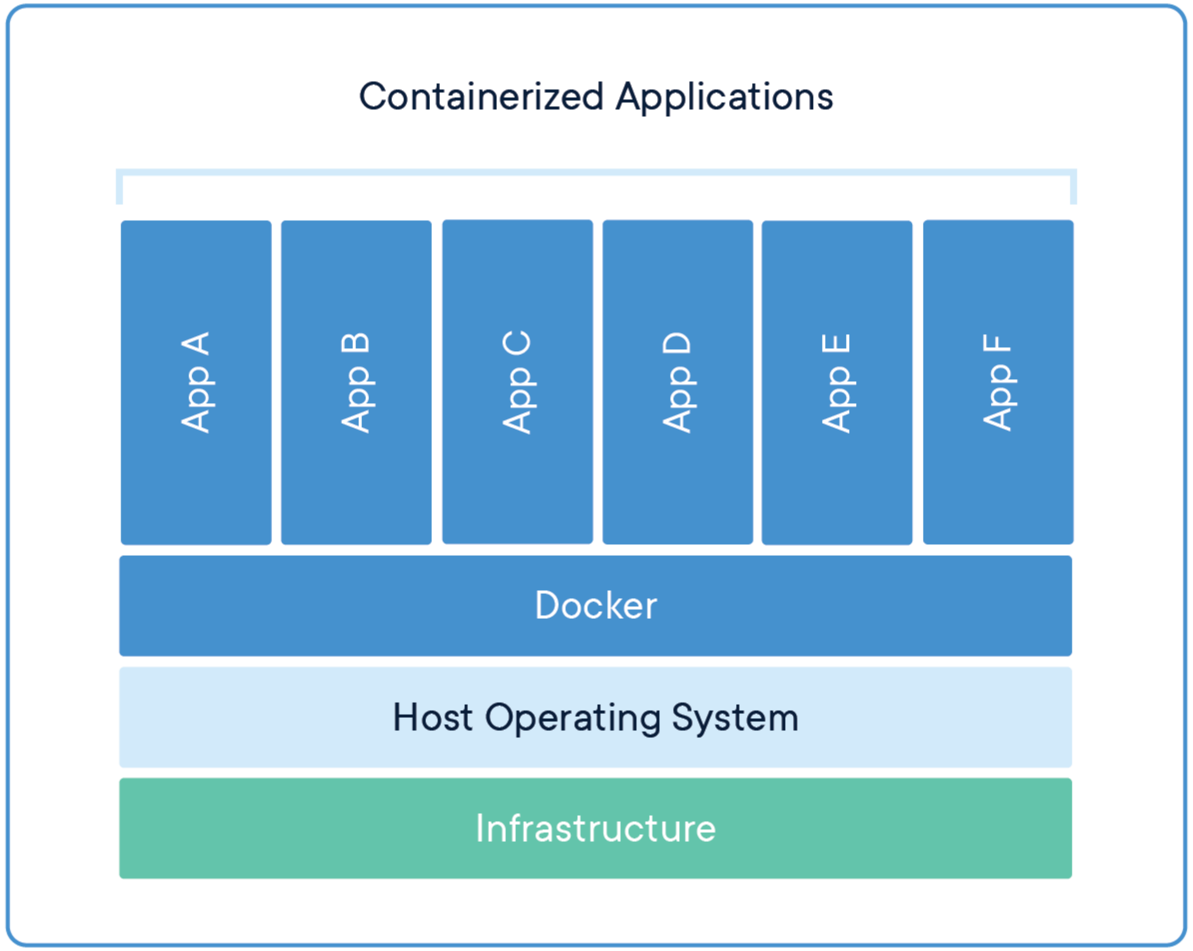
Client/Server



Virtual Machines



Containers


Containerized Applications
App A
App B
App C
App D
App E
App F
Docker
Host Operating System
Infrastructure

Microservices & Containers

- 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

Microservices & the Cloud

- Microservices have broad support across multiple Cloud providers
- 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
- Microservices are a great option for the Cloud because the elastic scalability provided by the Cloud infrastructure can directly support the independent scalability needed with a microservices architecture

Microservices Summary

- Applications that can
  - Efficiently scale
  - Are flexible
  - Are high performance
- These apps are powered by multiple services, working in concert
- Each service is small and has a single focus
- We use a lightweight communication mechanism to coordinate the overall system
- This ends up giving us a technology agnostic API

Demo: Microservices Application

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

Migration vs. Modernization

Application Portfolio Assessment

- Helps with planning for the migration of an enterprise’s system and software estate to the Cloud
- During this process, applications created or utilized by the enterprise will be reviewed for migration disposition
- Assessment may be completed for a small subset of the apps (as a starting point or proof of concept) or may include the entire estate

Application Portfolio Assessment

- Through this analysis, a disposition for Cloud readiness will be assigned to each app (based on several factors which we’ll review shortly)

- There may also be a determination of which apps are best suited to move first, on the way to defining a plan for migration of all apps in waves

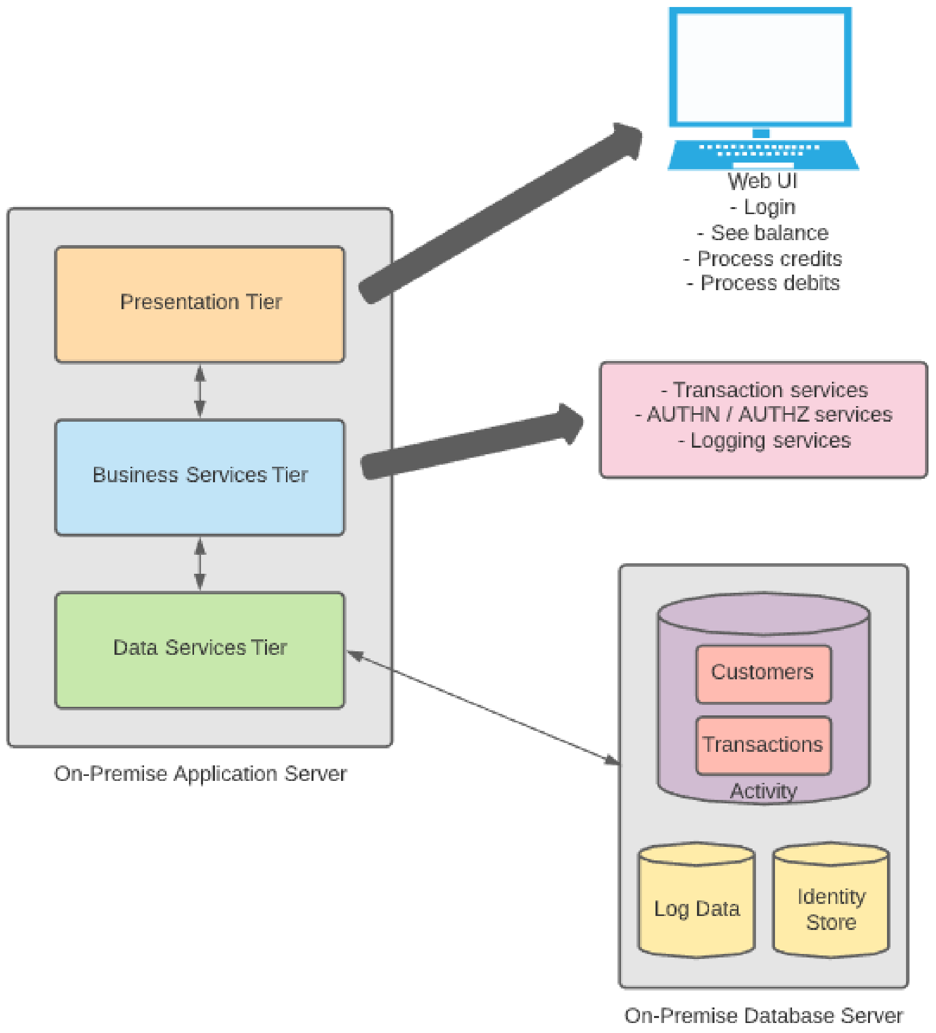
- The assessment will likely include an interview with the application owner (to gain SME knowledge about the application)

Application Portfolio Assessment

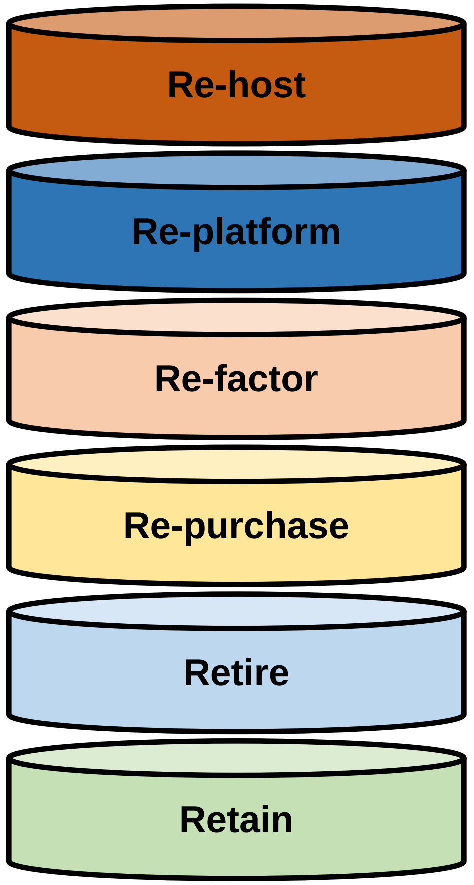
- Will likely include
  - Review of the current state architecture
  - Discussion with stakeholders about roadmap and plans for target state
  - Discussion on expected timing

- The goal is to ensure that migration efforts are focused on the activities that will bring greatest business benefit against optimized cost

Banking App – Current State



The 6 R’s – Application Migration Strategies



See https://docs.aws.amazon.com/whitepapers/latest/aws-migration-whitepaper/the-6-rs-6-application-migration-strategies.html for additional information

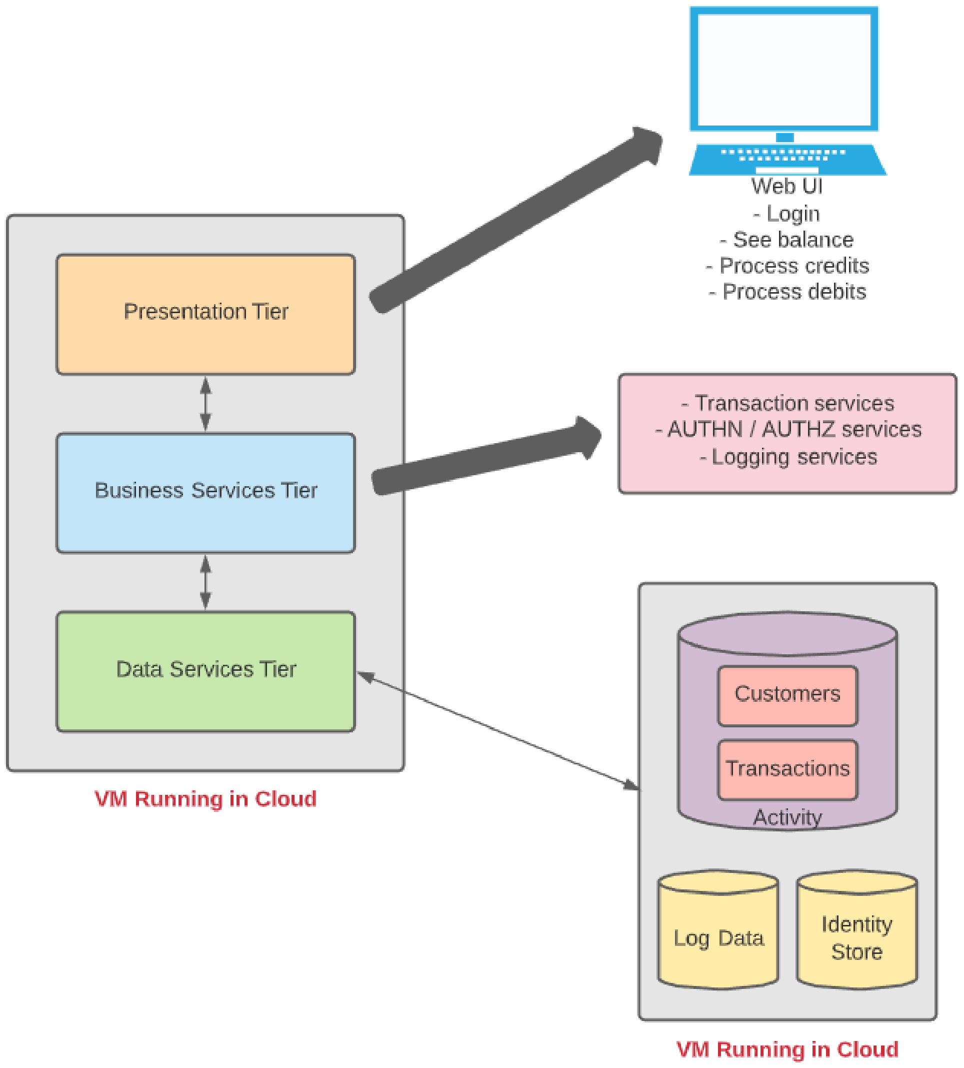
Re-host

- AKA “lift & shift”
- For all intents and purposes, involves recreating the on-premise infrastructure in the Cloud
- Sometimes used to expedite retirement of a data center

Re-host

- Can be a mechanism to quickly migrate workloads and see immediate cost savings, even without Cloud optimizations
- There are third-party tools available to help automate the migration
- Once the application is in the Cloud, it can be easier to apply Cloud optimizations vs. trying to migrate and optimize at the same time

Re-host The Banking App



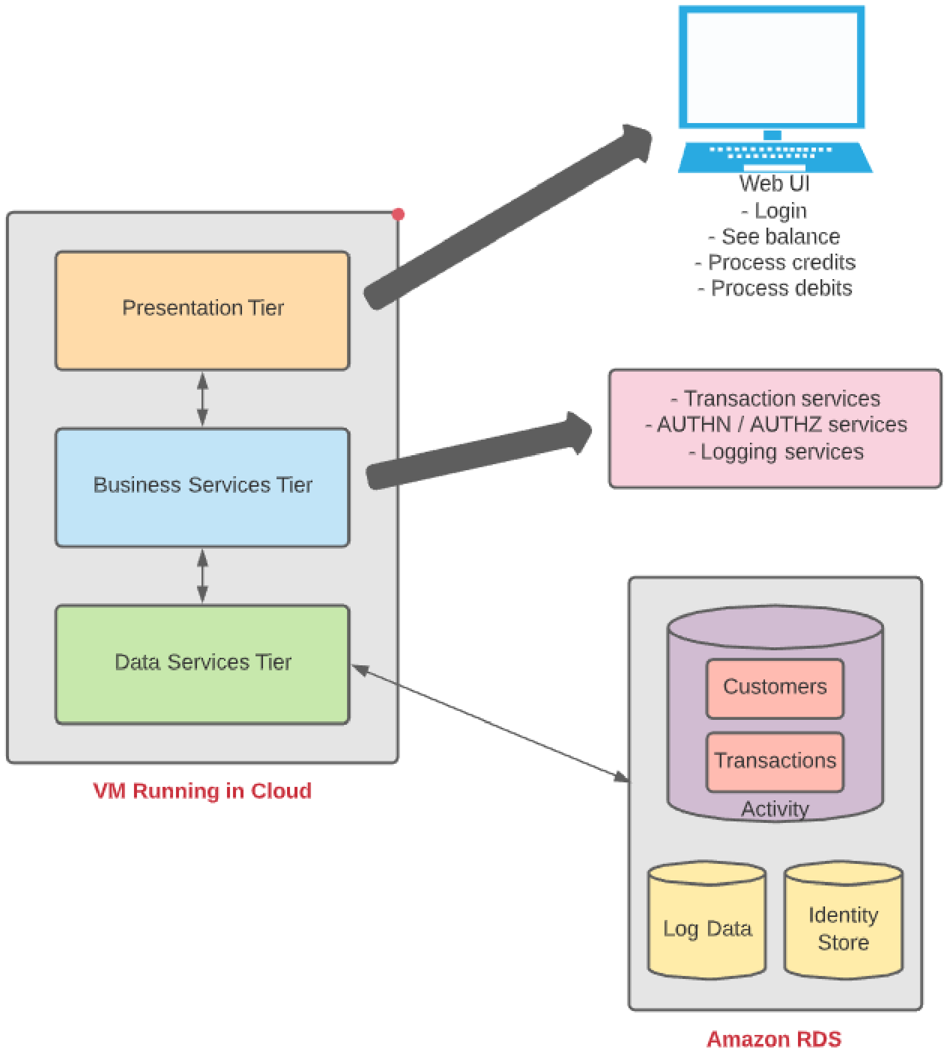
Re-platform

- AKA “lift, tinker & shift” or “lift & fiddle”
- Core architecture of the application will not change
- Involves recreating most of the on-premise infrastructure in the Cloud with a few Cloud optimizations

Re-platform

- Those optimizations will involve a move to one or more Cloud native services for a specific, tangible business benefit
- A common example is moving to a managed database (e.g. Relational Database Service, or RDS, in AWS)
- Enables migration speed while providing cost savings or benefit in a targeted portion of the application’s architecture

Re-platform The Banking App



Re-factor

- Involves rearchitecting the application to maximize utilization of Cloud native optimizations
- Likely the most expensive and most complex of the available options
- The application profile needs to fit, and business value must be identified commensurate with the cost required to execute

Re-factor

- Good option if the application can benefit from features, scalability or performance offered by the Cloud
- Examples include rearchitecting a monolith to microservices running in the Cloud or moving an application to serverless technologies for scale

Re-factor The Banking App

?

To Be Determined

Re-purchase

- Good fit for applications that are candidates for moving from on-premise licensing (for installed products) to a SaaS model
- Often applications that have been installed to manage a specific type of business capability
- Examples can include Customer Relationship Management (CRM), Enterprise Resource Planning (ERP), HR or e-mail (among others)

Retire

- In some cases, an enterprise may have a “healthy” percentage of legacy applications that are no longer being used or maintained (especially for larger organizations)

- When completing the Application Portfolio Assessment and associated interviews with application owners, look for opportunities to recommend retire of the unused legacy apps

- This type of application can represent a “quick win”

Retire

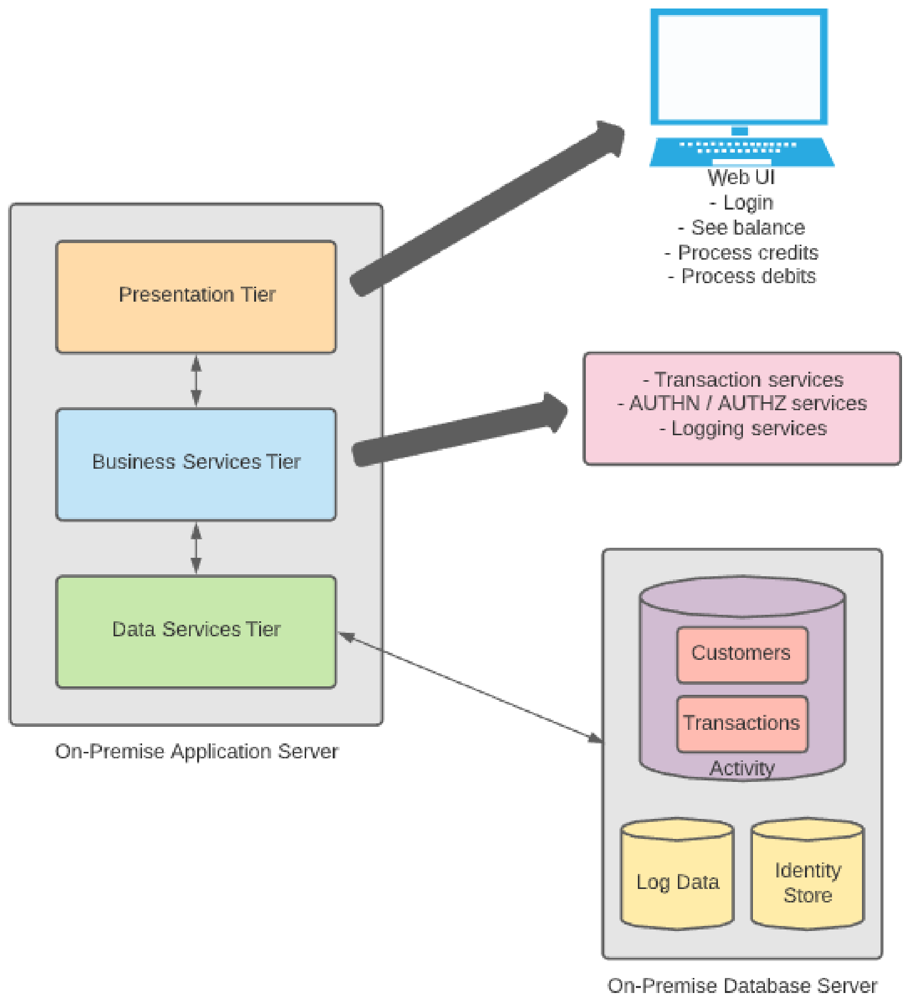
- The associated savings can potentially be advantageously factored into the business case for the Cloud modernization effort

- Finally, retiring unused apps reduces attack surface area from a security perspective

Retain

- Applications that must remain in place but that cannot be migrated to the Cloud without major refactor
- This type of application profile may prevent the ability to completely move out of the data center (at least in the interim)
- From a cost perspective, it can be difficult for an enterprise to take on both the operating expense of Cloud while continuing to carry the capital expense of a data center and on-premise infrastructure
- The hybrid model can be a good solution to support where needed

Banking App Re-factor



If you wanted to sell an investment project to a client for rearchitecting this application for migration to the Cloud, what are some of the potential benefits you might put forth as justification?

Cloud Features & Service Models

- As discussed, some distinguishing features of Cloud include:
  - Elastic scalability
  - Optimized costs (if done right)
  - Architectural flexibility (API’s, web UI’s, mobile, etc.)
  - Operational flexibility
  - Data as a “competitive advantage”

How Do We Measure Success?

- Every project is different…

- Establish KPI’s (Key Performance Indicators) specific to the use case

- Could include things like:
  - Reduction in cost
  - Reduction in downtime
  - Ability to efficiently keep up with increased volumes (e.g., seasonal)
  - Ability to exit (or significantly downsize investment in) a data center
  - % of application portfolio successfully dispositioned for clouse

- Likely a combo of quantitative and qualitative measures

THANK YOU

Develop Intelligence