Welcome

Working with GoLang Microservices

Hello

About me...

- 25+ years in the indusry
- 20+ years in teaching
- Certified Cloud architect
- Passionate about learning
- Also, passionate about Reese's Cups!

Prerequisites

This course assumes you:

- Have at least an intermediate level of experience with Go
- That experience could be gathered through any combination of previous courses, personal experience, or both

Why study this subject?

- Go is cool! ☺
- Microservices are cool! ☺
- Microservices in Go are REALLY cool! ☺ ☺

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

Objectives

At the end of this course you will be able to:

- Construct & deploy microservices utilizing Go
- Develop microservices that cross-compile to Linux or Windows
- Describe the fundamental philosophy around microservices architecture, their benefits, and their cost

Agenda

- Microservices – What they are and why do we use them?
- GoLang Deep Dive – Go routines, channels, and cross-compilation
- Serverless Architectures as an alternative
- Containerization and container orchestration for Go microservices

How we’re going to work together

- Slides / lecture
- Demos
- Team discussions
- Labs

Microservices

SOLID Principles & Good Design

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

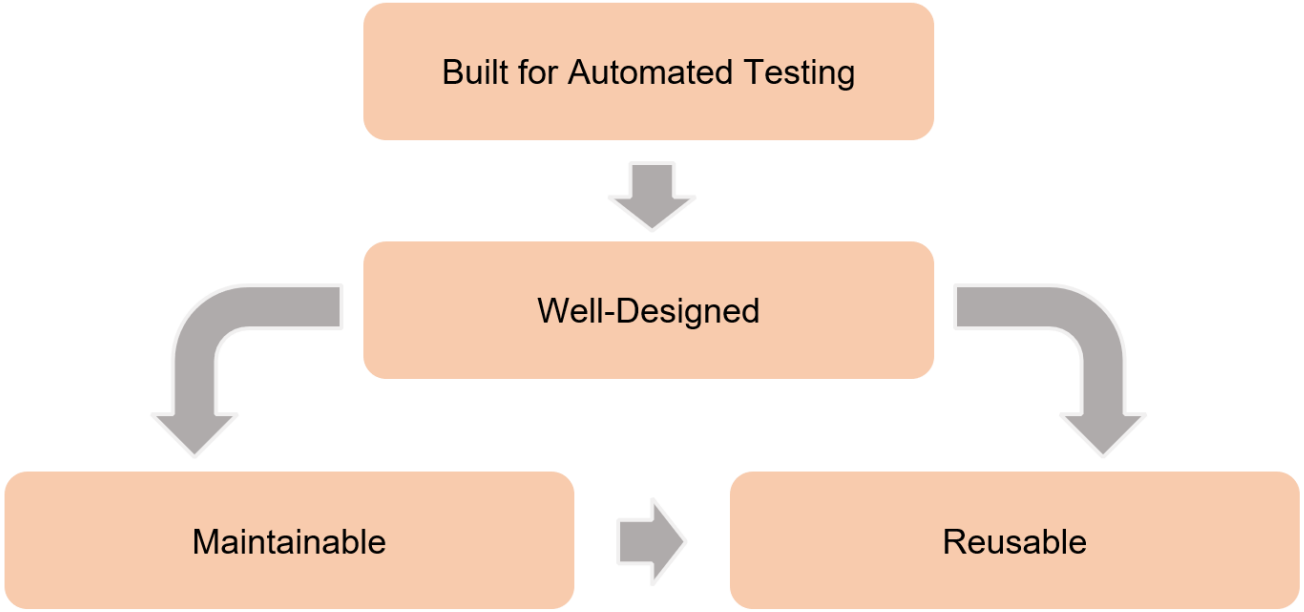
- Coupling between components or modules in a system causes problems, especially for maintaining the system over time
- System components that are tightly coupled, are more difficult to change (or enhance) – changes to one part of the system may break one or more other areas
- With coupling, you now 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

- 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…



SOLID Principles

SOLID principles help us build well-architected systems

- 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

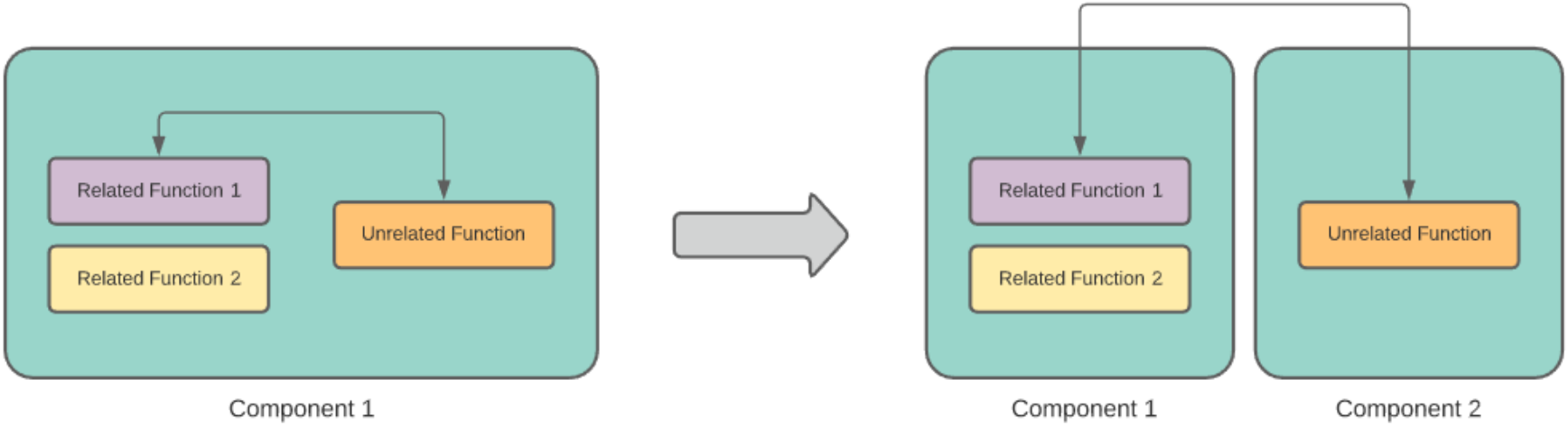
Single Responsibility Principle (SRP)

Single Responsibility Principle (SRP)

Why important?

- Related functions organized together breed understanding (logical groupings) – think *cohesion*
- Multiple, unrelated functionalities slammed together breed coupling
- Reduced complexity (cleaner, more organized code)
- Promotes smaller code modules
- Reduced regression – tension of change

Single Responsibility Principle (SRP)



Single Responsibility Principle (SRP)

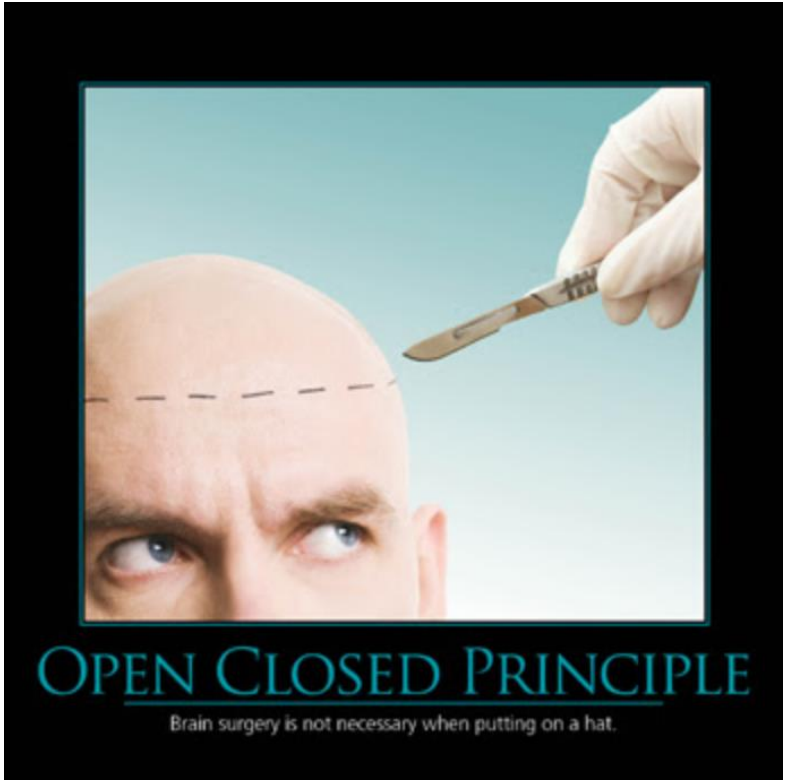
How to practice?

- When building new modules (or refactoring existing), think in terms of logical groupings
- Look for “axes of change” as points of separation
- Build new modules to take on new entity or service definitions – provides abstraction
- Structure clean integrations between separated modules
- Use existing tests (or build new ones) to verify a successful separation

Open-Closed Principle (OCP)

Software entities should be open for extension but closed for modification

Open-Closed Principle (OCP)



Open-Closed Principle (OCP)

Why important?

- Our systems need to be able to evolve
- We need to be able to minimize the impact of that evolution

Open-Closed Principle (OCP)

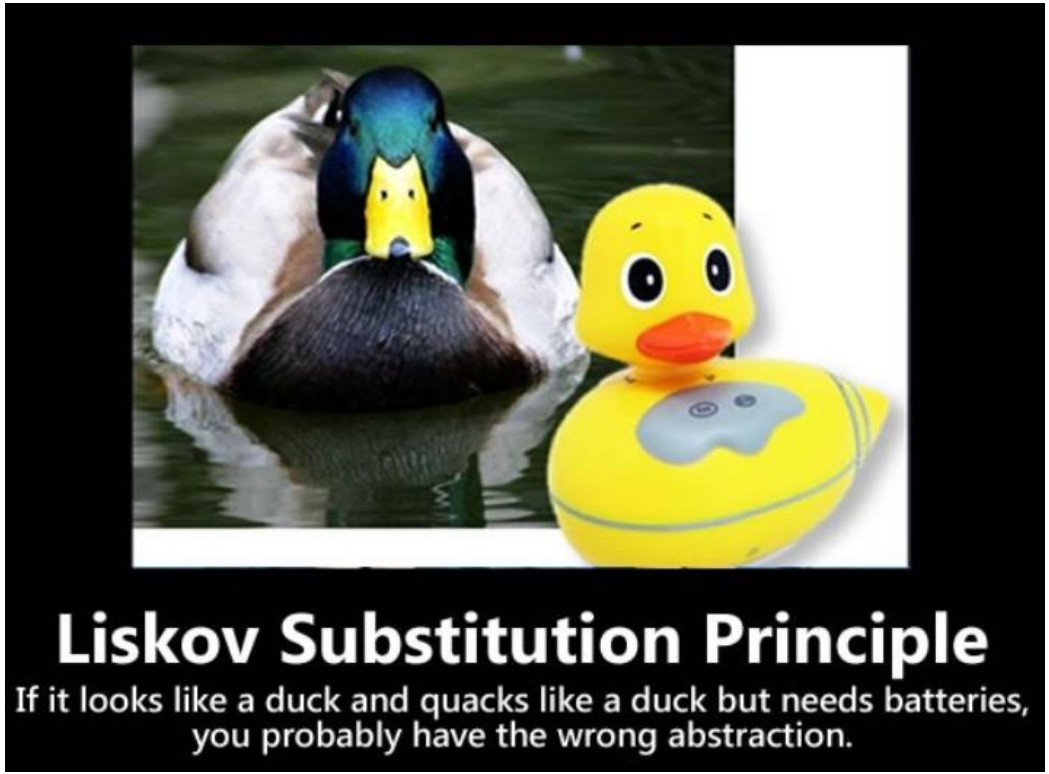
How to practice?

- When building new modules (or refactoring existing), leverage abstractions
- Use the abstractions as levers of extension
- Use existing tests (or build new ones) to verify the abstractions

Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types

Liskov Substitution Principle (LSP)



Liskov Substitution Principle (LSP)

Why important?

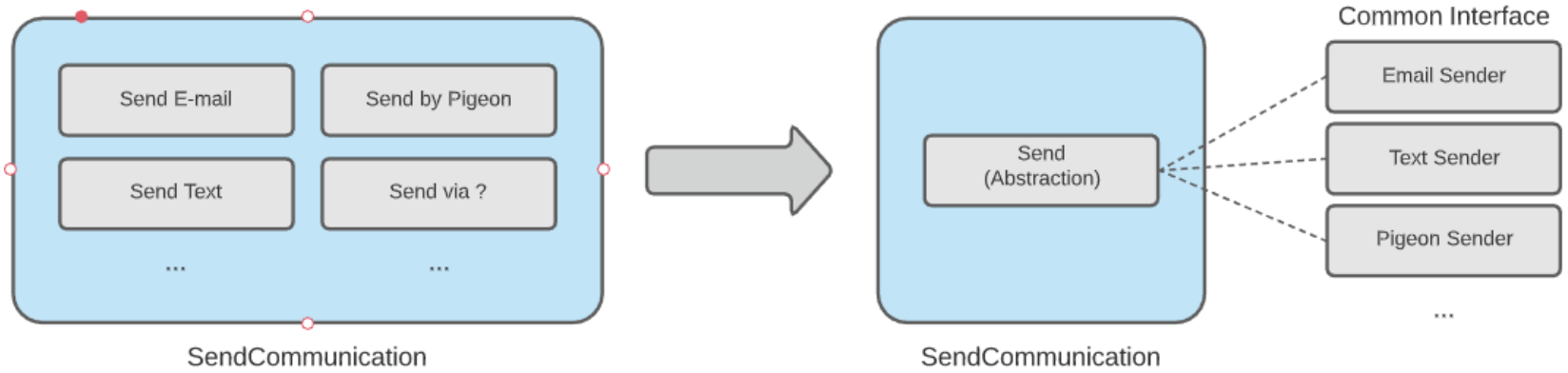
- Want to be able to use the abstractions created by OCP to extend existing functionality (vs. modify it)
- Especially useful when multiple variants of a type need to be processed as a single group
- Promotes looser coupling between our modules
- Without it, we may have to include if/else or switch blocks to route our logic
- Or keep adding parameters/properties for new but related types

Liskov Substitution Principle (LSP)

How to practice?

- Module members should reference the abstractions in consuming code
- Look for ways to encapsulate type-specific (or type-aware) logic in the type instead of in the code using the type
- Use existing tests (or build new ones) to verify our ability to effectively substitute

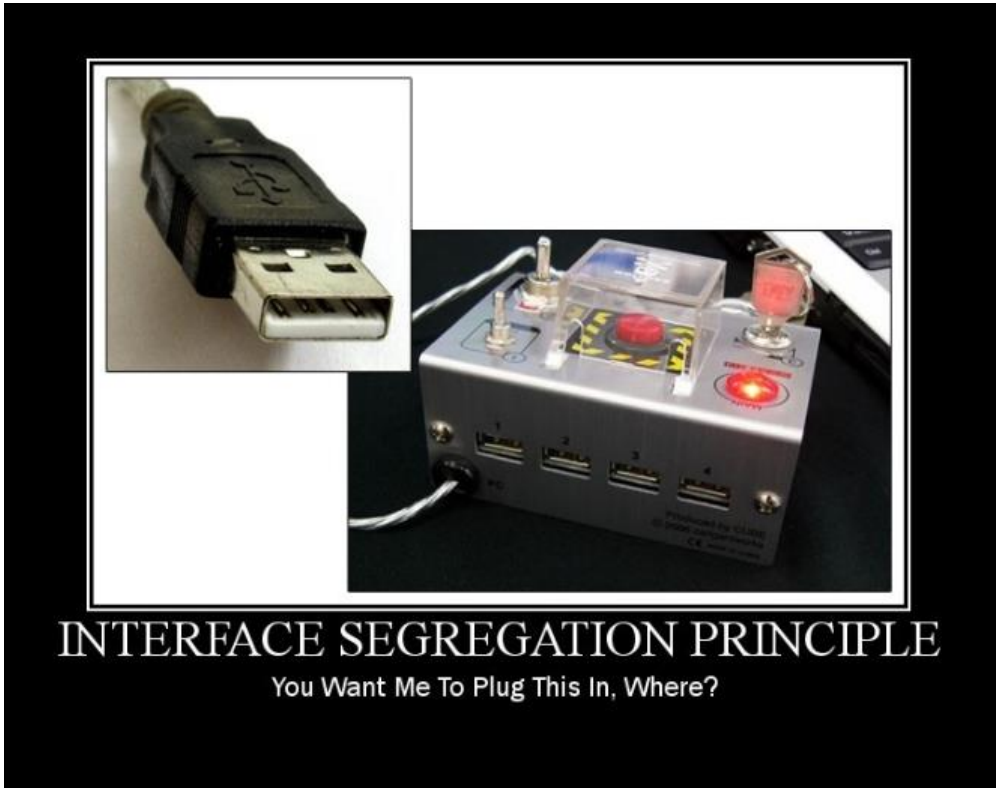
OCP & LSP


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)

Why important?

- By following SRP, OCP and LSP, we can build a set of cohesive abstractions enabling reuse in multiple clients
- However, sometimes the abstractions we need are not cohesive (even though they may seem like it at first)
- We need a mechanism for logical separation that still supports combining functions together in a loosely-coupled way
- Otherwise, we'll see "bloating" in our abstractions that can cause unintended/unrelated impact during normal change

Interface Segregation Principle (ISP)

How to practice?

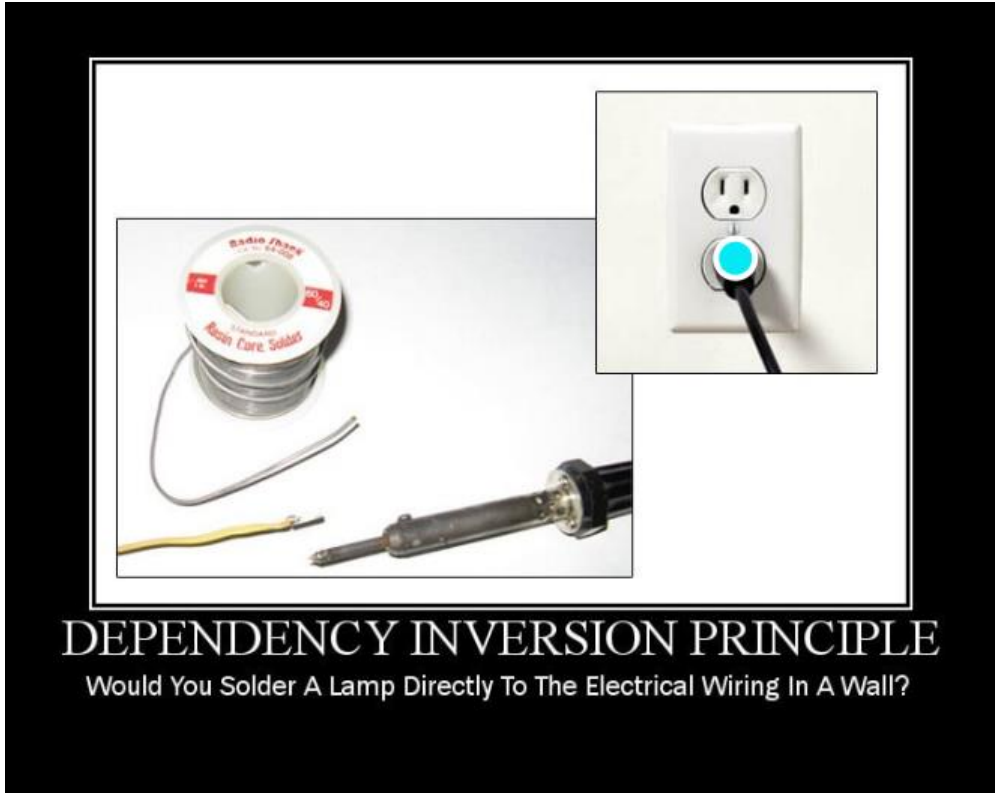
- In your abstractions, don’t force functions together that don’t belong together (or that you might want to use separately)
- Leverage delegation in the implementation of those abstractions to support variance
- Use OCP to bring together additional sets of features in a cohesive way (that still adheres to SOLID)
- Use existing tests (or build new ones) to verify aggregate features

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)



Dependency Inversion Principle (DIP)

Why important?

- As with all the SOLID principles, we want to build reusable but loosely-coupled code modules
- We don’t want to limit reuse to lower-level utility classes only
- If higher-level modules are tightly-coupled to and dependent on low-level modules, change impact can cascade up
- The change impact can be transitive (flowing through multiple layers in-between)
- There are patterns and utility libraries built to enable

Dependency Inversion Principle (DIP)

How to practice?

- As with the other principles, use (and depend on) abstractions
- Use mechanisms like dependency injection and IoC (Inversion of Control) to build looser coupling between logic providers and consumers
- Build layering into your architectures and limit references to the same or immediately adjacent layer only
- Keep ownership of abstractions with the clients that use them or, even better, in a separate namespace/library
- Use existing tests (or build new ones) to verify functionality in each layer and use mocking techniques to isolate testing

Lab Discussion

Lab

Microservices vs. Monolith

Microservices

- One of the current trends in software development (though it’s not really new)
- Built around concept of SOA (Service Oriented Architecture)
- Requires an approach that favors decomposition
- The architecture, design and testing concepts we’ve been discussing complement

Microservices – Key Characteristics

- “Small” units of Service Oriented Architecture, or SOA
- Independently-deployable (KEY)
- By extension, independently-scalable
- Modeled around business domain (instead of tech domain)

Microservices – Key Characteristics

- Interact over networks (including the Internet)
- Technology can be flexible
- Encapsulate both business capability AND data
- Data sources are not directly shared but exposed through well-defined interfaces
- About cohesion of business functionality vs. technology

vs. Monolithic

- Monolithic applications can exhibit some of the same characteristics as microservices:
  - Distributed
  - Interact via network
  - Might use unshared data sources
- Key difference, though, is requirement to deploy all parts of the system together (not independently-deployable)

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

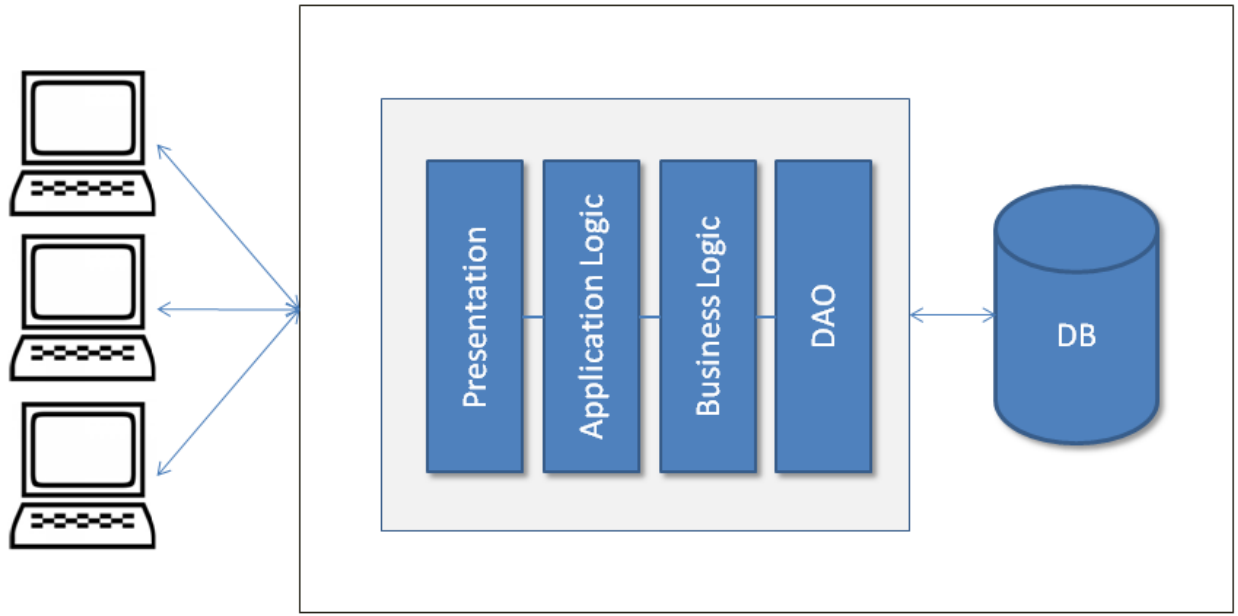
- With microservices, key concepts:
  - Cohesion (goal to increase)
  - Coupling (goal to decrease)

- Our earlier discussions on SOLID are applicable to targeting a microservices architecture:
  - 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

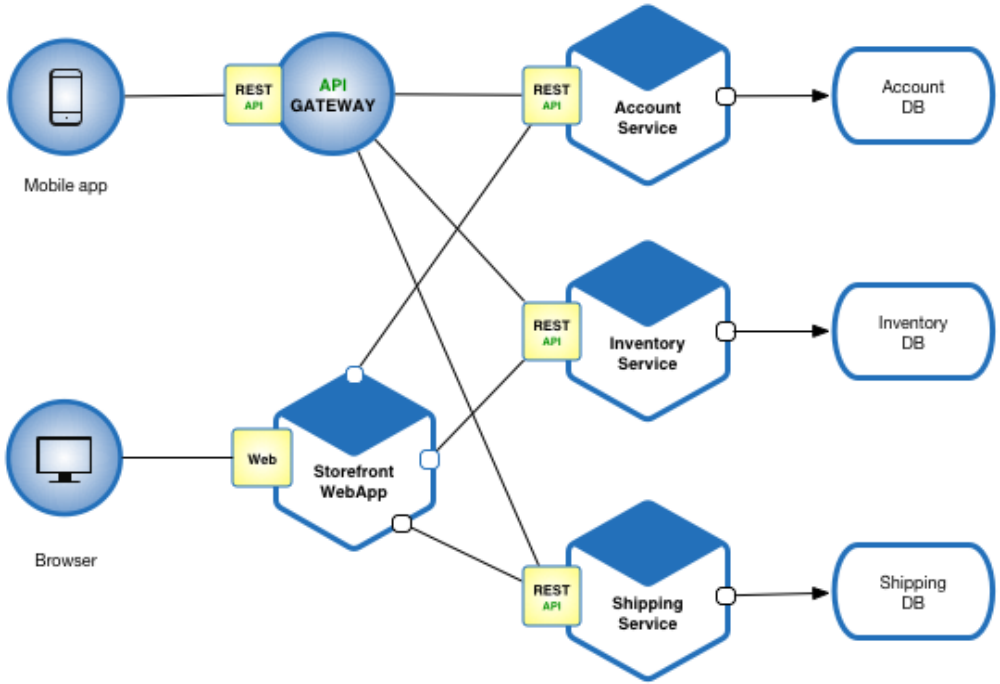
Microservices & Containers

- 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

Monolithic Architecture Example



Example Microservices Architecture



Lab

Concurrency in Go

See https://github.com/KernelGamut32/golang-live

Lab

Thank you!

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

DevelopIntelligence
A PLURALSIGHT COMPANY