INTRODUCTION TO SOFTWARE ARCHITECTURES

MODULE 6

SOFTWARE COMPLEXITY AND ARCHITECTURE

INTRODUCTION TO SOFTWARE ARCHITECTURES

Software Complexity

- Decade after decade, software systems have seen orders-of-magnitude increases in their size and complexity
- Imagine how hard basketball would be if it scaled up the same way, with 5 people on the floor one decade, then 50, then 500.
- Because of this growth, today's software systems are arguably the largest and most complex things ever built.
- Software developers are always battling the ever-stronger foes of ***complexity*** and ***scale***.

Developer's Weapons

- Since developers cannot grow bigger brains, they have instead improved their weapons.
- An improved weapon gives developers two options:
  - to more easily conquer yesterday's problems, or
  - to combat tomorrow's.
- We are no smarter than developers of the previous generation, but our improved weapons allow us to build software of greater size and complexity.
- Software developers wield some ***tangible*** weapons
  - Integrated Development Environments (IDEs)
  - Programming languages
- But ***intangible*** weapons arguably make a bigger impact.

Partitioning, knowledge, and abstractions

> To be successful at combating the scale and complexity of software in the next decade, developers will need improved weapons
  - Partitioning
  - Knowledge
  - Abstraction

Partitioning

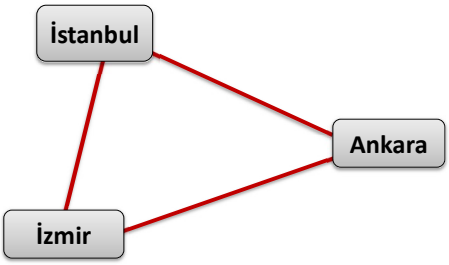
> Partitioning is effective as a strategy to combat complexity and scale when two conditions are true:
  - first, the divided parts must be sufficiently small that a person can now solve them;
  - second, it must be possible to reason about how the parts assemble into a whole.
> Parts that are encapsulated are easier to reason about
  - Need to track fewer details when composing the parts into a solution.
> You can forget about the details inside the other parts.
  - Allows the developer to more easily reason about how the parts will interact with each other.

Knowledge

> Software developers use knowledge of prior problems to help them solve current ones.
> This knowledge can be implicit know-how or explicitly written down.
> It can be specific, as in which components work well with others, or general, as in techniques for optimizing a database table layout.
> It comes in many forms, including books, lectures, pattern descriptions, source code, design documents, or sketches on a whiteboard.

Abstraction

> Abstraction can effectively combat complexity and scale because it shrinks problems, and smaller problems are easier to reason about.
> If you are driving from İstanbul to Ankara, you can simplify the navigation problem by considering only highways.
> By hiding details, you have shrunken the number of options to consider, making the problem easier to reason about.



ALL SOFTWARE SYSTEMS HAVE ARHITECTURES

INTRODUCTION TO SOFTWARE ARCHITECTURES

A tale of two systems

Plain Old Telephone System	Skype
• Feature:	• Feature:
▪ Call subscriber	▪ Call subscriber
• Architecture:	• Architecture:
▪ Centralized hardware switch	▪ Peer-to-peer software
• Good qualities	• Good qualities
▪ Works during power outages	▪ Scales without central hardware changes
▪ Reliable	▪ Easy to add new features (e.g., video calling)
▪ Emergency calls get location information	

Same feature

Architects pay more attention to **qualities** that arise from architecture choices.

LET'S
DESIGN A SYSTEM

Let's design a system!

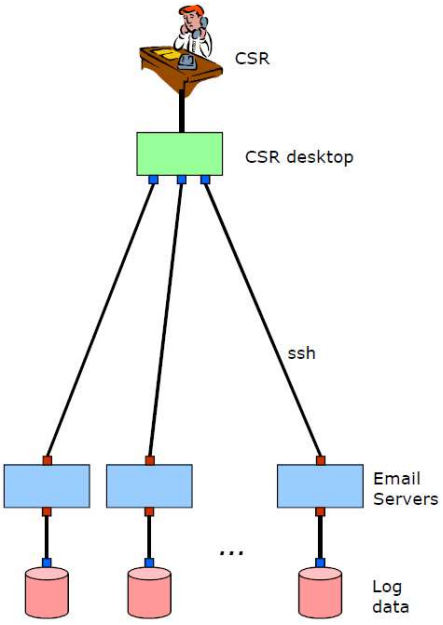
- Here's the situation
  - You are a hosting provider
  - You rent mail servers
  - Customers have problems
  - You use the mail log files to diagnose their problems
- The big question:
  - How would you build it?
- Let's assume you can build it
  - … but different architectures yield different qualities
- Why is this hard?
  - You have hundreds of servers
  - You generate GBs of logs daily
  - Collecting logs takes time
  - Searching logs takes time
- Hints and options
  - Central collection of logs?
  - Distributed searching of logs?
  - Can you pre-process logs to speed up queries?

Surprise! The system is real: Rackspace

- Exercise based on real experience
  - Rackspace is a hosting provider
  - Huge growth in customers, mail servers – and problems
  - Re-designs: 3 major versions (6 total versions)
- Let's review the 3 systems they built
  - All 3 had the same functionality (!)
  - … but different architectures
- Why this is so cool
  - Very expensive to build the same system 3 times
  - The only big change was the architecture
  - So, we can see the effect of architecture
  - … especially on quality attributes

Rackspace: Architecture #1

- Hosting provider of email service
- Email log files
- Task: debug user problem
- Architecture
  - CSR desktop computer
  - ssh connections to servers
  - Servers with local log files
- Procedure
  - Write query as grep expression
  - Script runs via ssh on every server
  - Results aggregated



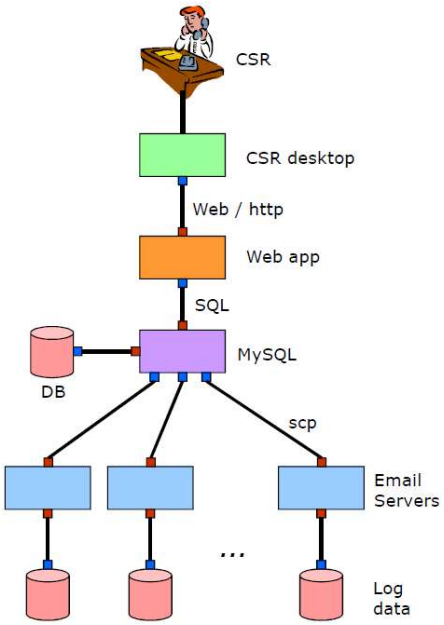
Rackspace: Architecture #2

- Hosting provider of email service
- Email log files

- Task: debug user problem

- Architecture
  - CSR desktop computer
  - Web application
  - MySQL database
  - scp log transfer
  - Servers with local log files

- Procedure
  - Every 10 minutes, send log files to MySQL server; delete original
  - Parse and load logs into MySQL
  - Combine new logs with old
  - Send query to MySQL server; answered from DB data



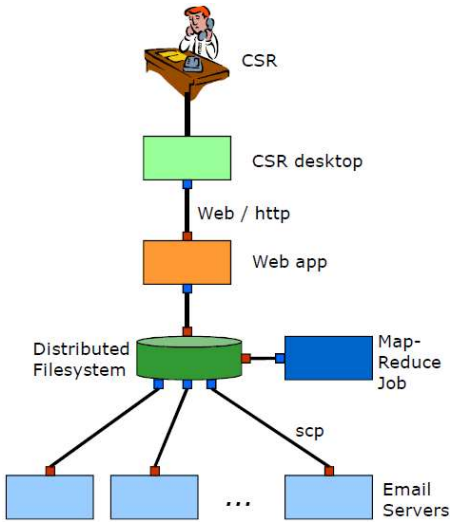
Rackspace: Architecture #3

- Hosting provider of email service
- Email log files

- Task: debug user problem

- Architecture
  - CSR desktop computer
  - Web application
  - Distributed filesystem
  - Map-Reduce job cluster
  - Servers with local log files

- Procedure
  - Log data continuously streamed from email servers to distributed filesystem (HDFS)
  - Every 10 minutes, Map-Reduce job runs to process log files, create index
  - Web app queries index



Rackspace: Quality attribute tradeoffs

> **Tradeoff**: Data freshness
  - V1: Queries run on current data
  - V2: Queries run on 10 minute old data
  - V3: Queries run on 10-20 minute old data

> **Tradeoff**: Scalability
  - V1: Noticeable email server slowdown (dozens of servers)
  - V2: MySQL speed/stability problems (hundreds of servers)
  - V3: No problems yet

> **Tradeoff**: Ad hoc query ease
  - V1: Regular expression
  - V2: SQL expression
  - V3: Map-Reduce program

LET'S DEFINE
SOFTWARE ARCHITECTURE

Architects architecting architectures

> Job title

Architect

> Development processes

Architecting

> Engineering artifacts

Software **Architect**ure

What is software architecture?

The **software architecture** of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. [Documenting Software Architectures (SEI) 2010]

Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. [IEEE 2000]

> In loose language:
  – It's the "big picture" or "macroscopic" organization of the system
> Problem with these definitions
  – Why are some detailed designs architectural, others not?
  – Architecture includes whatever architects say it does

What is software architecture?

> You can read the SEI collection of definitions, or contribute your own, at http://www.sei.cmu.edu/architecture/start/glossary/community.cfm

Software architecture

> **Software architecture** has emerged as an important subdiscipline of software engineering.
> Architecture is roughly the ***prudent partitioning*** of a whole into parts, with specific relations among the parts.
> This ***partitioning*** is what allows groups of people—often separated by organizational, geographical, and even time-zone boundaries—to work ***cooperatively*** and ***productively*** together to solve a ***much larger*** problem than any of them could solve individually.
> Each group writes software that interacts with the other groups' software through ***carefully crafted interfaces*** that reveal the minimal and most ***stable*** information necessary for interaction.

Software architecture

> From that interaction emerges the functionality and quality attributes—security, modifiability, performance, and so forth—that the system's stakeholders demand.
> The **larger** and **more complex** the system, the **more critical** is this partitioning—and hence, architecture.
> The **more demanding** those quality attributes are, the **more critical** the architecture is.
> Architecture is what makes the sets of parts work together as a coherent and successful whole.

Architecture documentation

> Architecture documentation
  – helps architects make the right decisions;
  – tells developers how to carry them out
  – records those decisions to give a system's future caretakers insight into the architect's solution.

All programs have an architecture

- Every program has an architecture
  But not every architecture suits the program
- System requirements
  - Functional needs
  - Quality needs (e.g., performance, security)
- Alignment
  - Different architectures support different requirements
    supporting high throughput vs. interactivity
  - Right: Suitable vs. unsuitable
    - Wrong: Good vs. bad
- Hard to change architecture later
  - Does not mean BDUF
  - But, need to think "enough"

What if you don't think architecturally?

- Developers optimize locally, miss the big picture
  - Lousy choice of frameworks, languages, …
- Project success depends on having virtuosos in the team
  - But how many James Goslings and Jeff Deans are there?
- Poor communication
  - Idiosyncratic notations, fuzzy semantics
- Shallow (or no) analysis of design options
  - Ad hoc; no use of best practices
  - From first principles, therefore high effort
  - Little attention to tradeoffs and rationale
- Architectural patterns ignored
  - … or incorrectly chosen
  - Squandering known-good designs

DOCUMENTING ARCHITECTURE

Architecture and Quality Attributes

- For nearly all systems, quality attributes such as performance, reliability, security, and modifiability are every bit as important as making sure that the software computes the correct answer.
- A software system's ability to produce correct results isn't helpful
  - if it takes ***too long*** doing it,
  - or the system ***doesn't stay up long enough*** to deliver it,
  - or the system ***reveals the results*** to your competition or your enemy.
- Architecture is where these concerns are addressed.

Quality Attributes

> If you require ***high performance***, you need to
  - Exploit potential parallelism by decomposing the work into cooperating or synchronizing processes.
  - Manage the inter-process and network communication volume and data access frequencies.
  - Be able to estimate expected latencies and throughputs.
  - Identify potential performance bottlenecks.

Quality Attributes

> If your system needs ***high accuracy***,
  - you must pay attention to
    - how the data elements are defined and used
    - how their values flow throughout the system.

Quality Attributes

- If ***security*** is important, you need to
  - Legislate usage relationships and communication restrictions among the parts.
  - Identify parts of the system where an unauthorized intrusion will do the most damage.
  - Possibly introduce special elements that have earned a high degree of trust.

Quality Attributes

- If you need to support ***modifiability*** and ***portability***,
  - you must carefully separate concerns among the parts of the system, so that when a change affects one element, that change does not ripple across the system.

Quality Attributes

- If you want to ***deploy*** the system ***incrementally***, by releasing successively larger subsets,
  - you have to keep the dependency relationships among the pieces untangled, to avoid the "nothing works until everything works" syndrome.

Quality Attributes & Architecture Documentation

- Architecture documentation has three obligations related to quality attributes.
  - it should indicate which quality attribute requirements drove the design.
  - it should capture the solutions chosen to satisfy the quality attribute requirements.
  - it should capture a convincing argument why the solutions provide the necessary quality attributes.
- The goal is to capture enough information so that the architecture can be analyzed to see if, in fact, the system(s) derived from it will possess the necessary quality attributes.

Difference Between Architecture and Design

- Architecture *is* design, but not all design is architecture
- Many design decisions are left unbound by the architecture and are happily left to the discretion and good judgment of downstream designers and even implementers.
- Architecture consists of architectural design decisions, and all others are non-architectural.
  - So what decisions are non-architectural?
  - What design decisions does the architect leave to the discretion of others?

Difference Between Architecture and Design

- Architectural decisions are ones that permit a system to meet its quality attribute and behavioral requirements.
- All other decisions are non-architectural.
- Any design decisions resulting in element properties that are *not* visible —that is, make no difference outside the element—are non-architectural:
  - the selection of a data structure
  - the selection of an algorithms to manage and access that data structure.

Difference Between Architecture and Design

> The classes, packages and their relations in a UML class diagram are architectural?

> Sequence diagrams are non-architectural?

> Defining the services of an SOA system is architectural?

> Designing the internal structure of each service provider component is non-architectural?

Can Architectural decisions be detailed?

"The element

- delivers its computational result through this published interface
- is thread-safe
- puts no more than three messages on the network per invocation
- returns its answer in less than 20 ms."

Can Architectural decisions be detailed?

> Some architectural decisions can be quite “detailed”
  – The adoption of specific protocols,
  – An XML schema
  – Communication or technology standards.
> Such decisions are usually made for the purpose
  – Interoperability
  – Scalability
  – Extensibility
  – …

Uses of Architecture Documentation

> Fundamentally, architecture documentation has three uses
  – Architecture serves as a means of education.
  – Architecture serves as a primary vehicle for communication among **stakeholders**
  – Architecture serves as the basis for system analysis and construction.

Architecture Documentation and Quality Attributes

1. Any major design approach (such as an architecture pattern or style) chosen by the architect will have quality attribute properties associated with it.
   - Client-server is good for scalability
   - Layering is good for portability
   - An information-hiding-based decomposition is good for modifiability,
   - services are good for interoperability
   - …

> Explaining the choice of approach is likely to include a discussion about the satisfaction of quality attribute requirements and trade-offs incurred: called ***rationale***.

AD and Quality Attributes

2. Individual architectural elements that provide a service often have quality attribute bounds assigned to them.
   - Consumers of the services need to know how fast, secure, or reliable those services are.

> These quality attribute bounds are defined in the interface documentation for the elements, sometimes in the form of a Quality of Service contract.

> Or they may simply be recorded as *properties* that the elements exhibit.

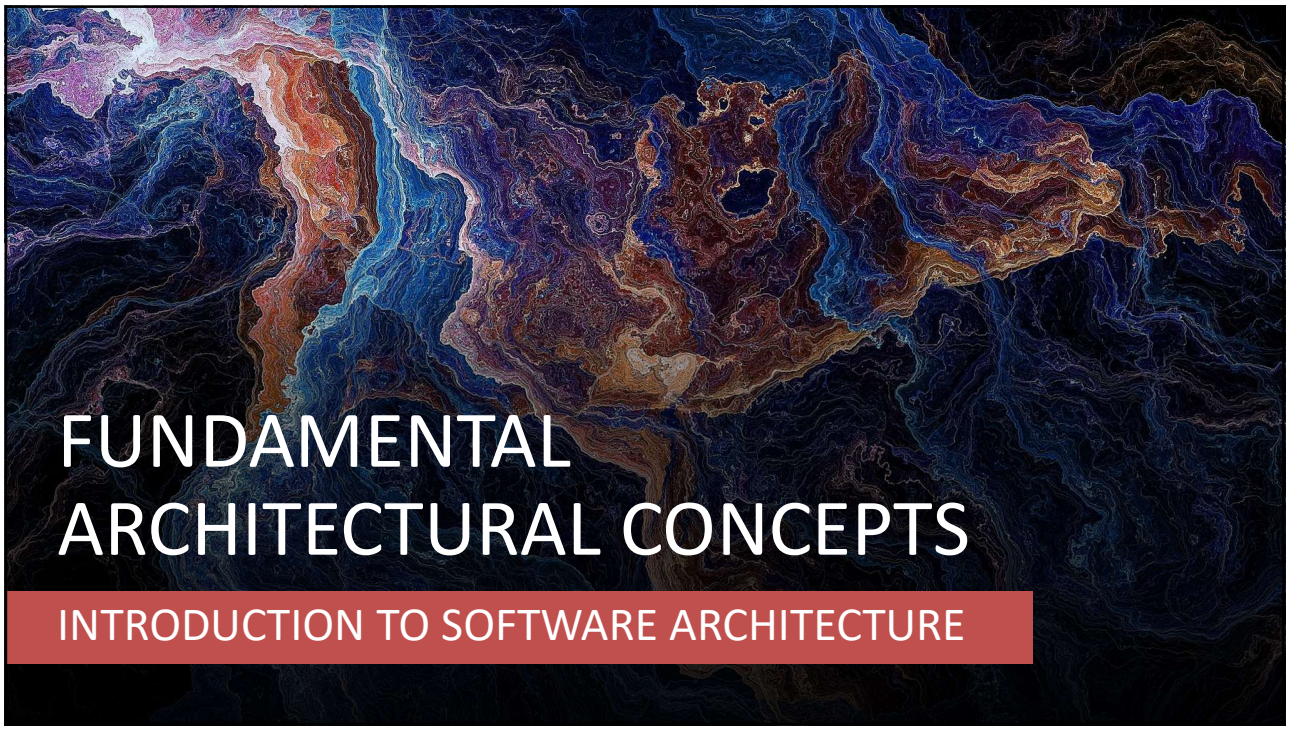
AD and Quality Attributes

3. Quality attributes often impart a “language” of things that you would look for.
   - Security involves things like security levels, authenticated users, audit trails, firewalls, and the like.
   - Performance brings to mind buffer capacities, deadlines, periods, event rates and distributions, clocks and timers,….
   - Availability conjures up mean time between failure, failover mechanisms, primary and secondary functionality, critical and noncritical processes, and redundant elements.

AD and Quality Attributes

4. Architecture documentation often contains a mapping to requirements that shows how requirements (including quality attribute requirements) are satisfied.

> If your requirements document establishes a requirement for availability, for instance, then you should be able to look up that requirement by name or reference in your architecture document to see the place(s) where that requirement is satisfied.



FUNDAMENTAL ARCHITECTURAL CONCEPTS

INTRODUCTION TO SOFTWARE ARCHITECTURE

Objectives

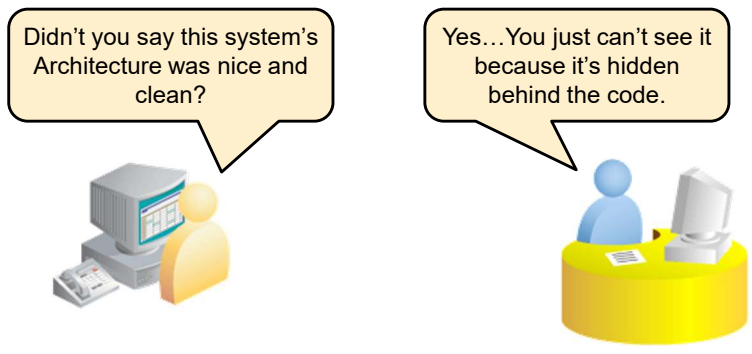
After completing this lesson, you should be able to:

- Distinguish between architecture and design
- Describe the use of architectural patterns
- Describe the architecture workflow
- Describe the diagrams of the key architecture views
- Select the architecture type
- Describe common enterprise architecture frameworks

Discussion Questions

- What, if anything, makes architecture different from design?
- Does object-oriented analysis and design have anything to do with architecture?
- What goes on in the environment in which architecture is created and turned into a product? Why does the architect care?
- What documents does the architect produce? What purpose do these documents serve?
- Is it important for the architect to be experienced in evaluating Spring technologies? If so, why?
- What are patterns? Why are they relevant?

Which Is More Important, Architecture or Design?



Distinguishing Between Architecture and Design

	Architect	Designer
Abstraction level	High/broad Focus on few details	Low/specific Focus on many details
Deliverables	System and subsystem plans, architectural prototype	Component designs, code specifications
Area of focus	Nonfunctional requirements, risk management	Functional requirements

Common Principles Between Architecture & Design

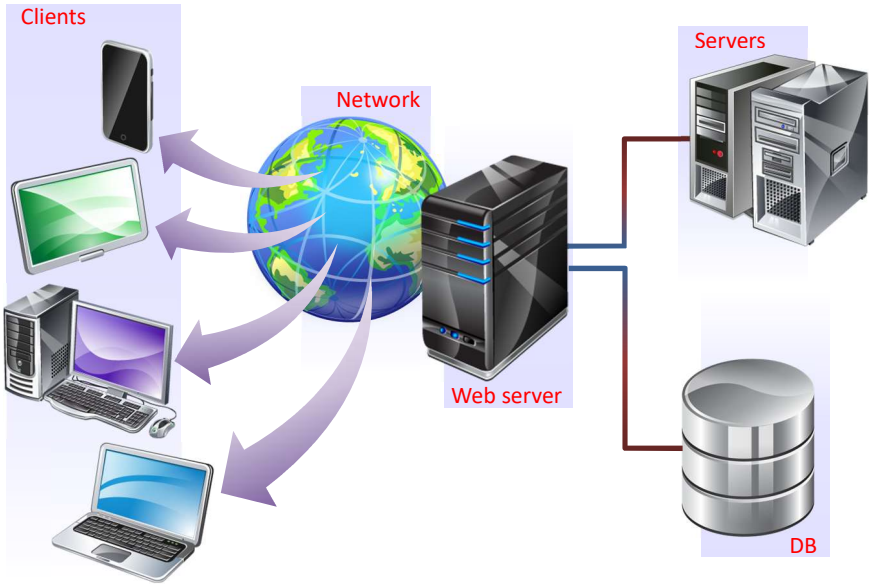
> Common characteristics of architecture and design:
  - Abstraction
  - Encapsulation
  - Cohesion
  - Coupling
> Like design and implementation, the architecture also requires refactoring to incorporate changes.

Architectural Principles

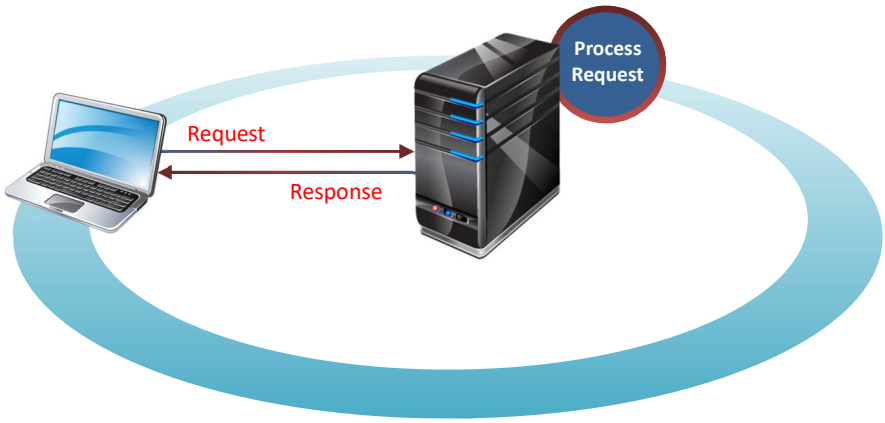
> Architecture principles are axioms or assumptions that suggest good practices when constructing a system architecture:
  - Separation of Concerns
  - Dependency Inversion Principle
  - Separate volatile from stable components
  - Use component and container frameworks
  - Keep component interfaces simple and clear
  - Keep remote component interfaces coarse-grained

WEB ARCHITECTURES

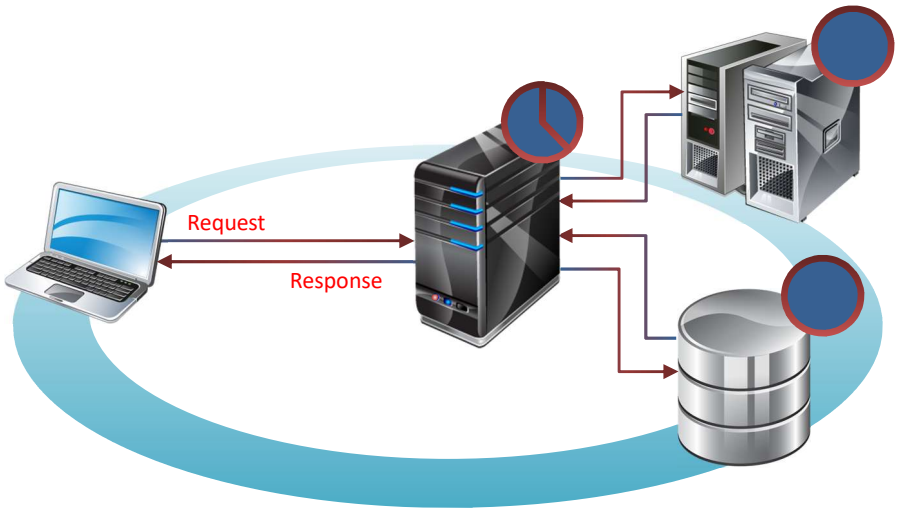
Web Architecture



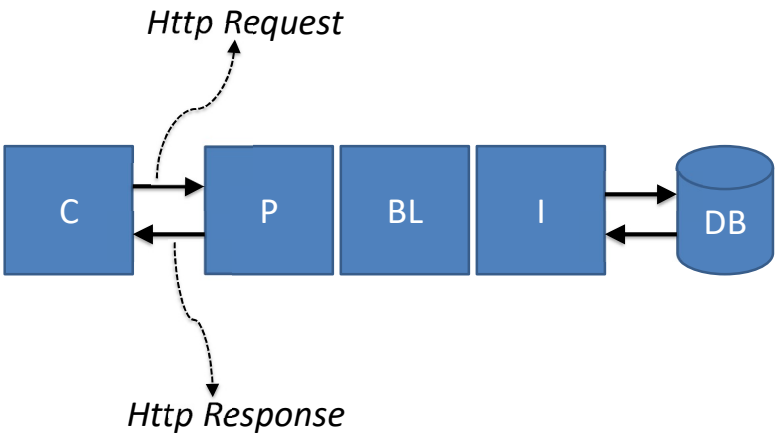
How Web Servers Work



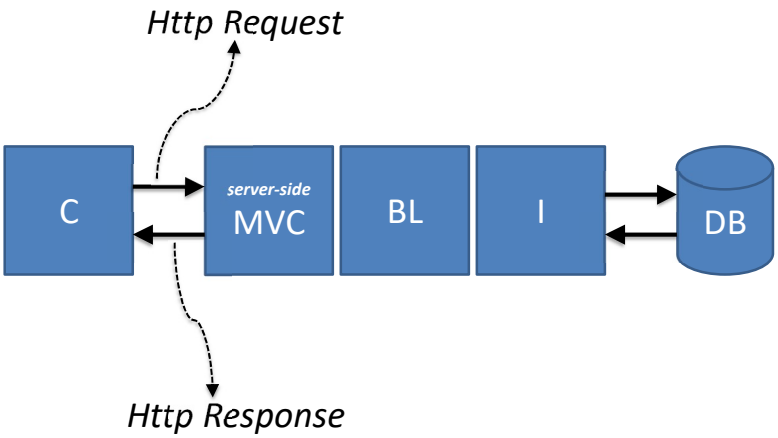
How Web Servers Work



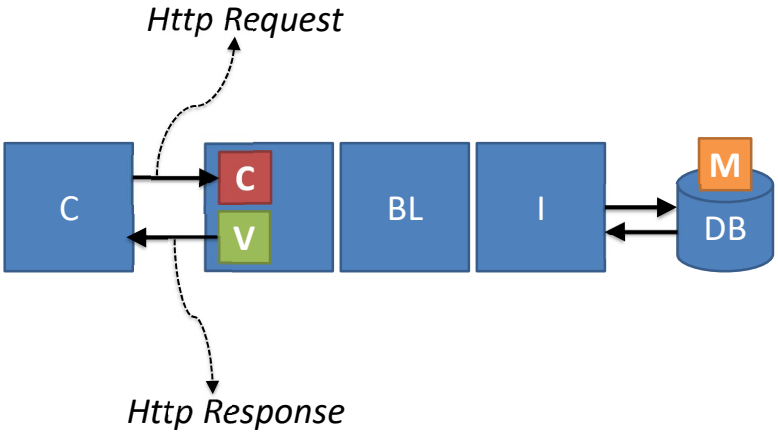
Request-Response



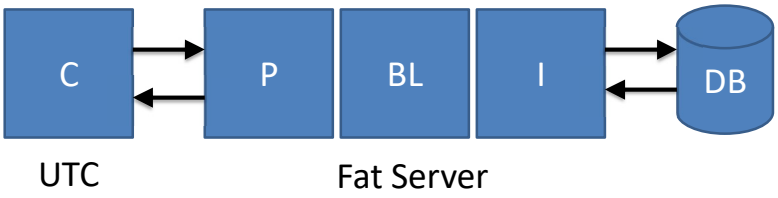
Request-Response



Request-Response



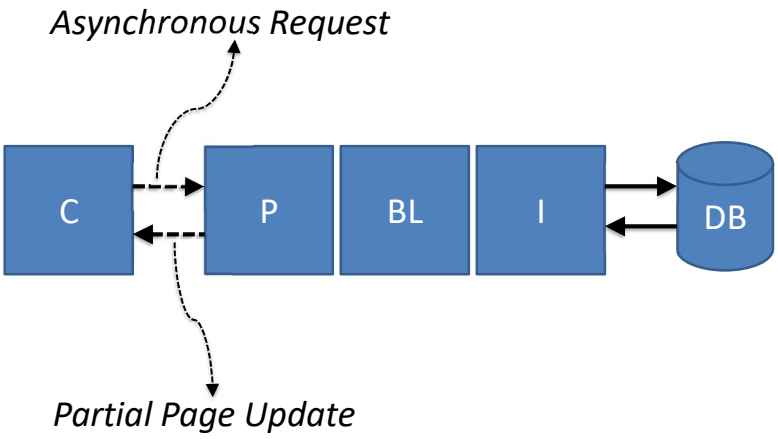
Request-Response

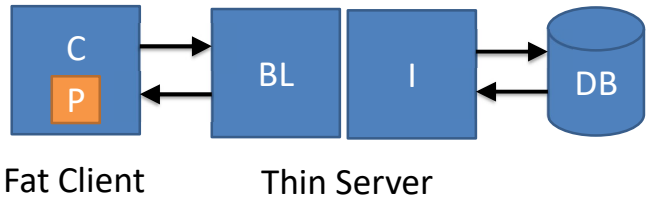


UTC

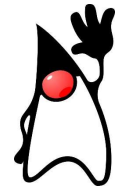
Fat Server

Ajax

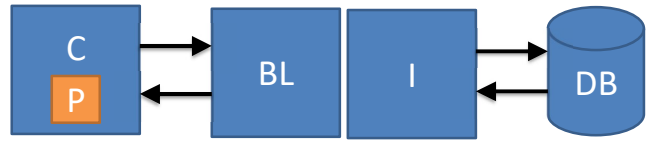




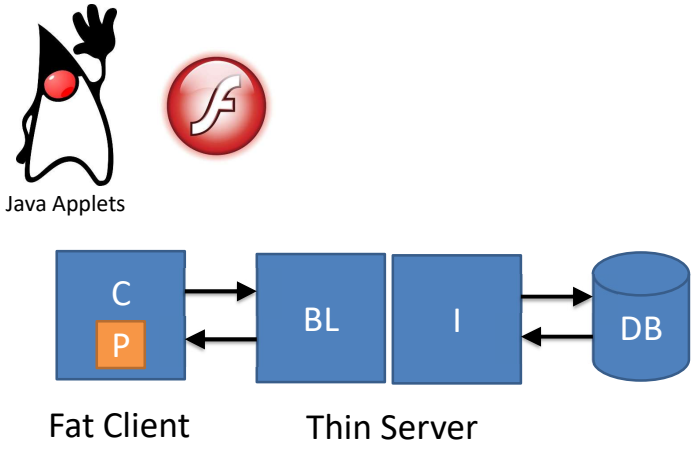
Fat Client Thin Server

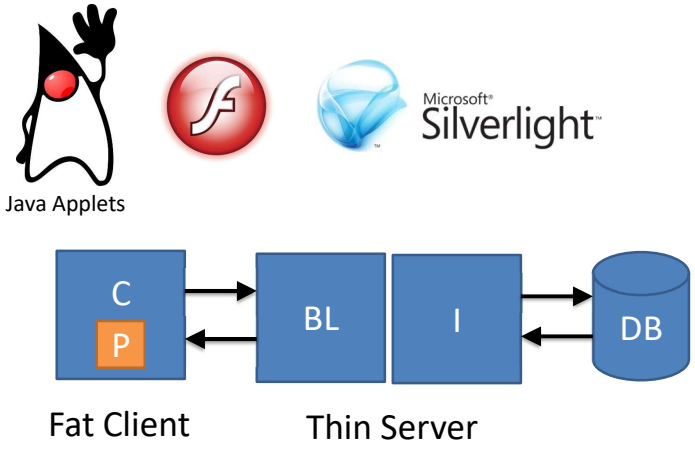


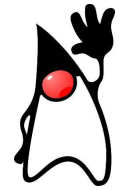
Java Applets



Fat Client Thin Server


Java Applets
C
P
BL
I
DB
Fat Client
Thin Server


Java Applets
Microsoft
Silverlight
C
P
BL
I
DB
Fat Client
Thin Server



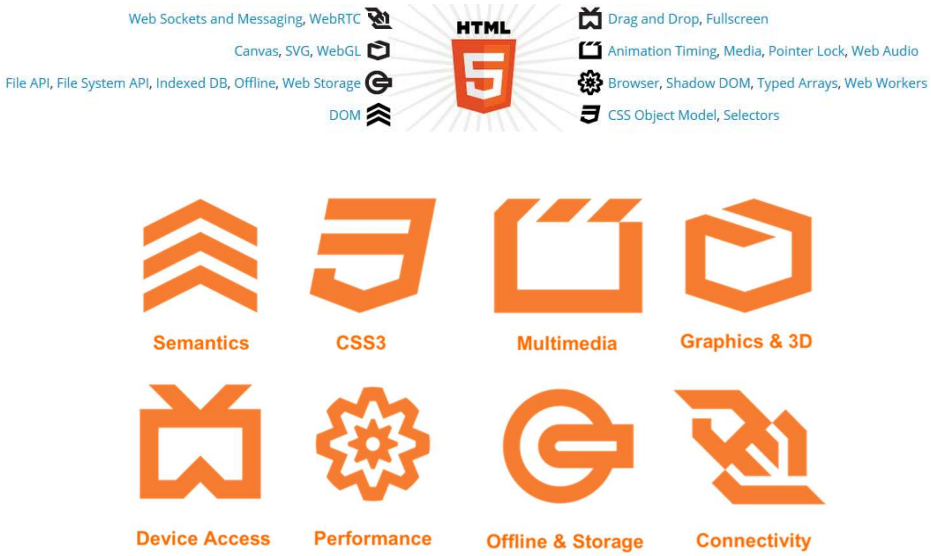
Java Applets

Fat Client Thin Server

Java Applets

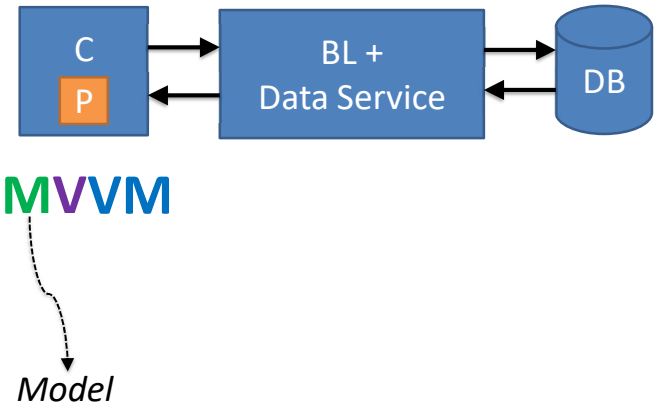
Fat Client Thin Server

HTML5 APIs

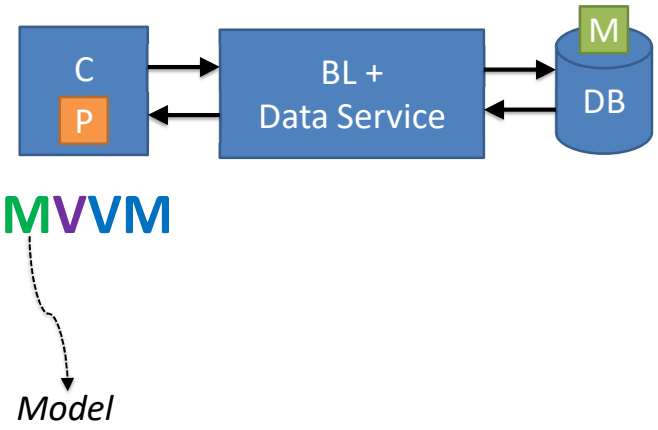


Data Service

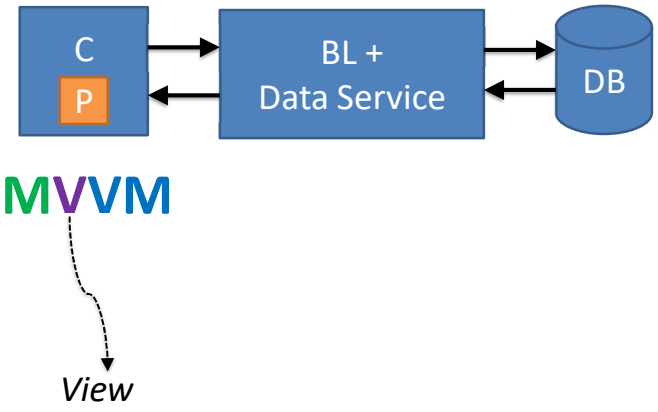
UI Logic



UI Logic



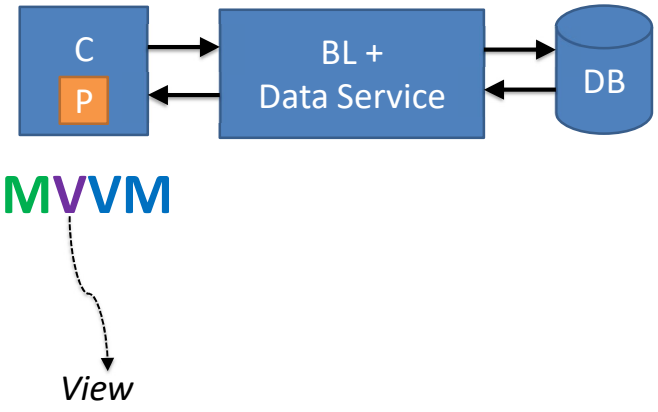
UI Logic



MVVM

View

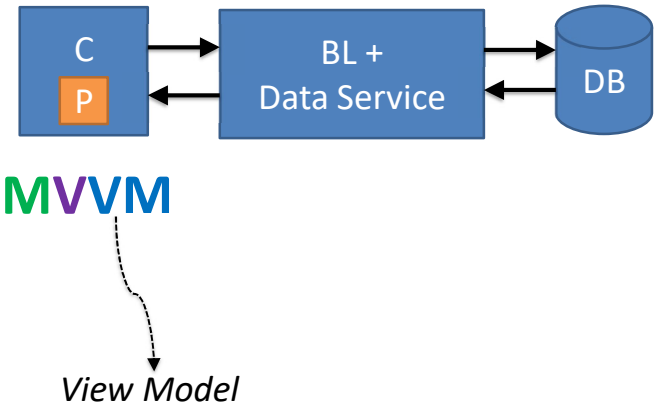
UI Logic



MVVM

View

UI Logic

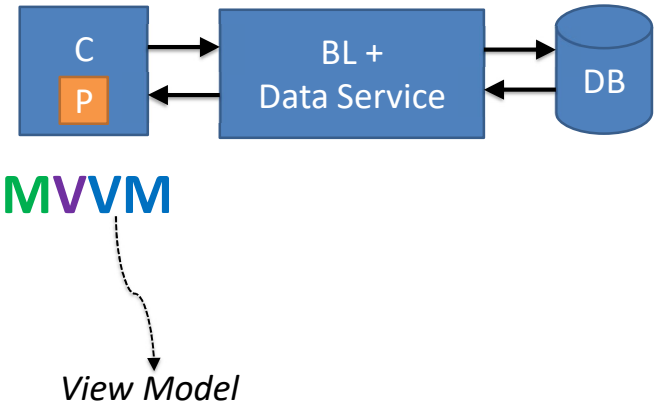


MVVM

View Model

UI Logic

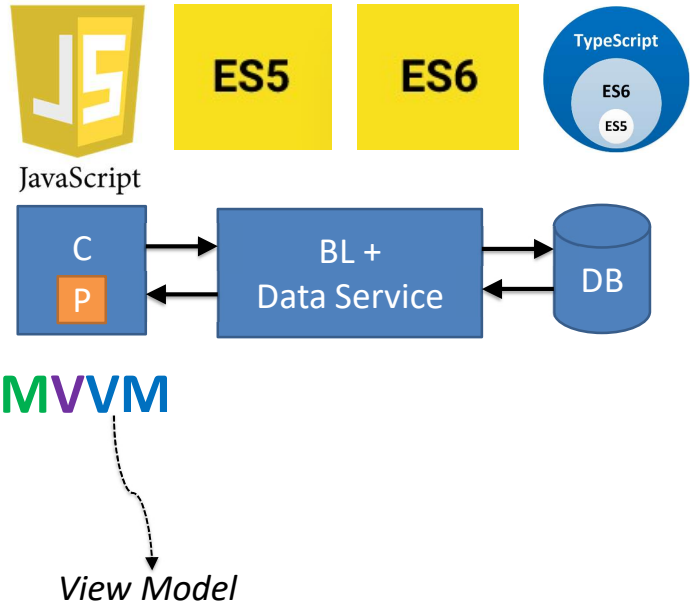
JavaScript



MVVM

View Model

UI Logic

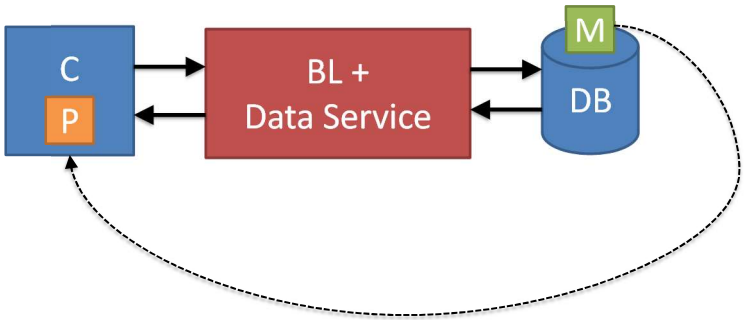


MVVM

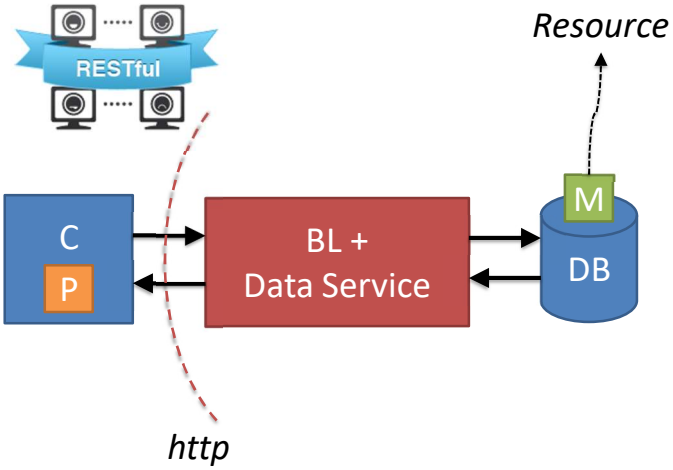
View Model

Data Service

Data Service



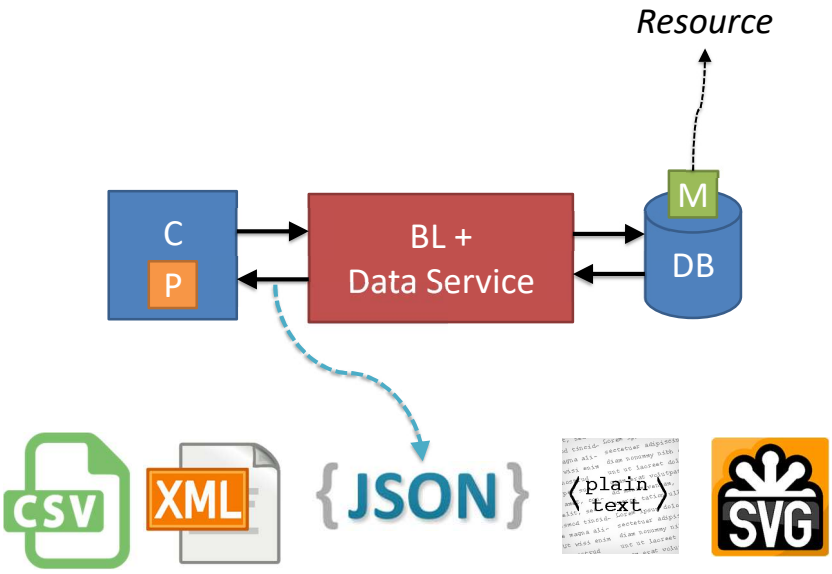
Data Service



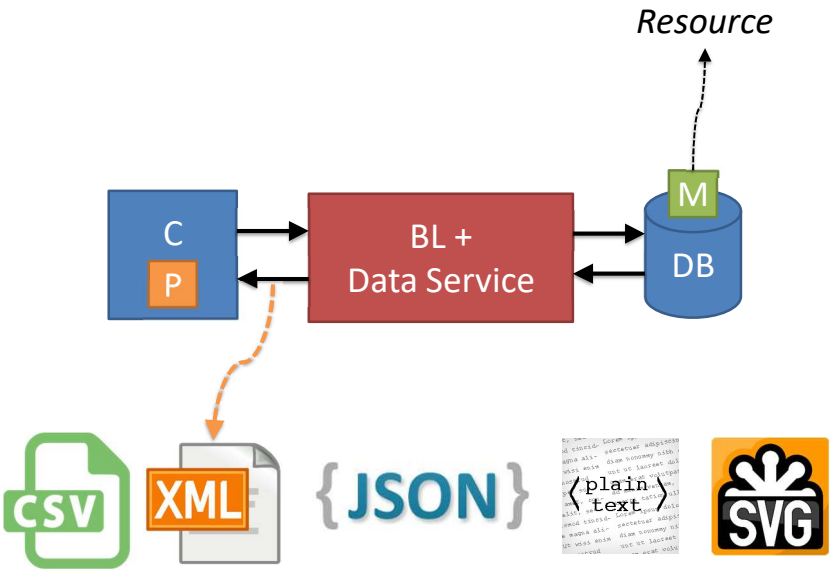
Data Service

HTTP	SQL
GET	SELECT
POST/PUT	INSERT
PUT/POST	UPDATE
DELETE	DELETE

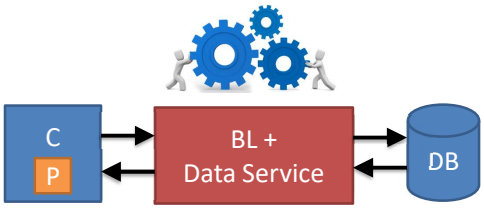
RESTful Data Service



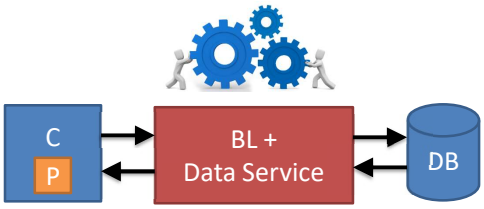
RESTful Data Service



RESTful Services

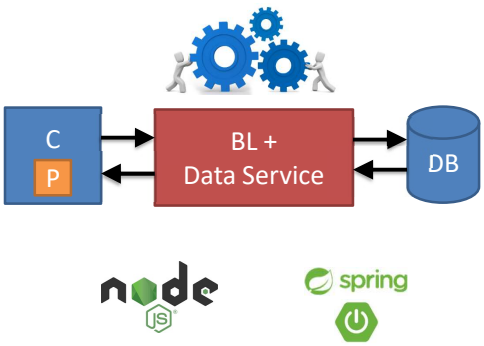


RESTful Services

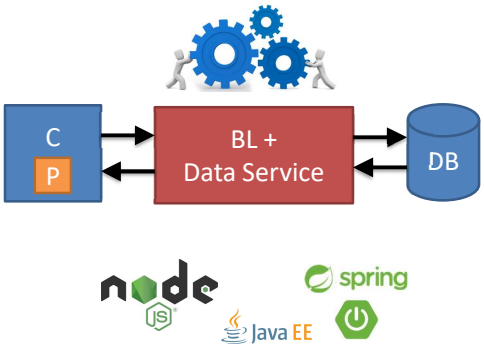


node

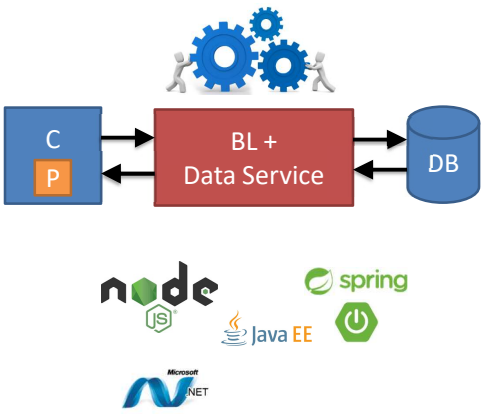
RESTful Services



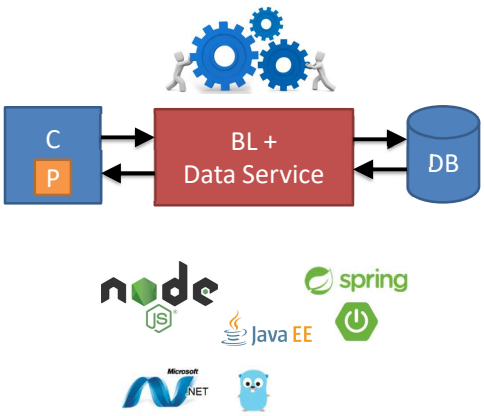
RESTful Services



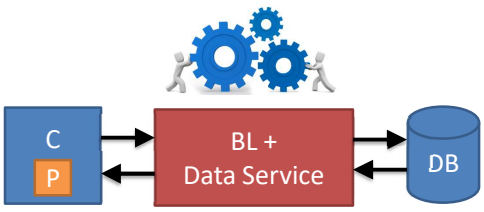
RESTful Services



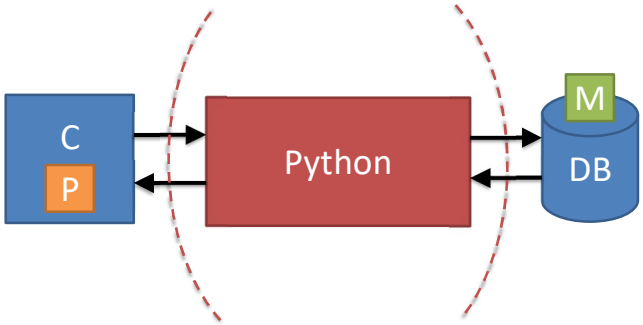
RESTful Services



RESTful Services

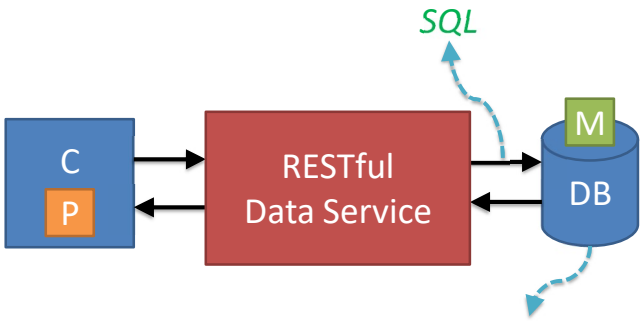


RESTful Data Service: Python



Flask/*Django/Tornado* SQLAlchemy/Ponyorm/Storm/**PyMongo**

Data Service

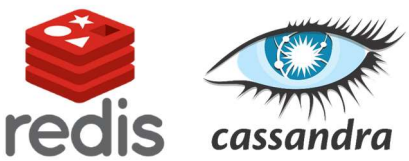


Relational

JOIN

NOT CLUSTER FRIENDLY

Data Service



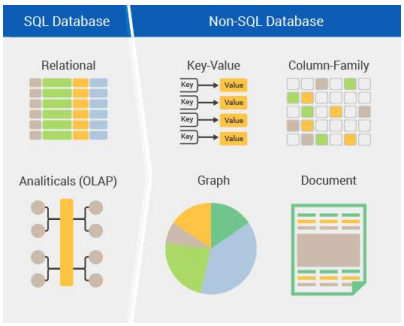
NO-SQL DBs

SCHEMA-LESS

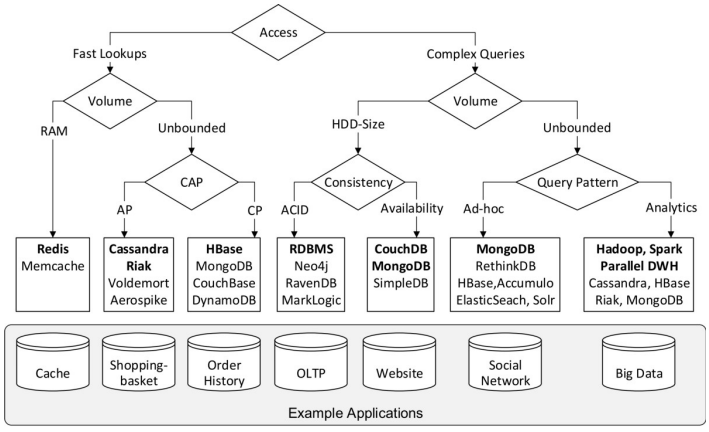
Scalability vs. Consistency

Cluster Friendly

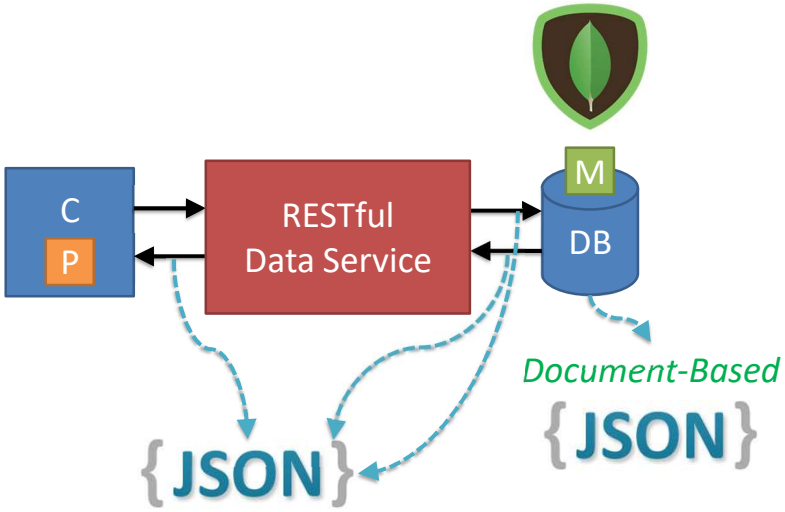
NoSQL Databases



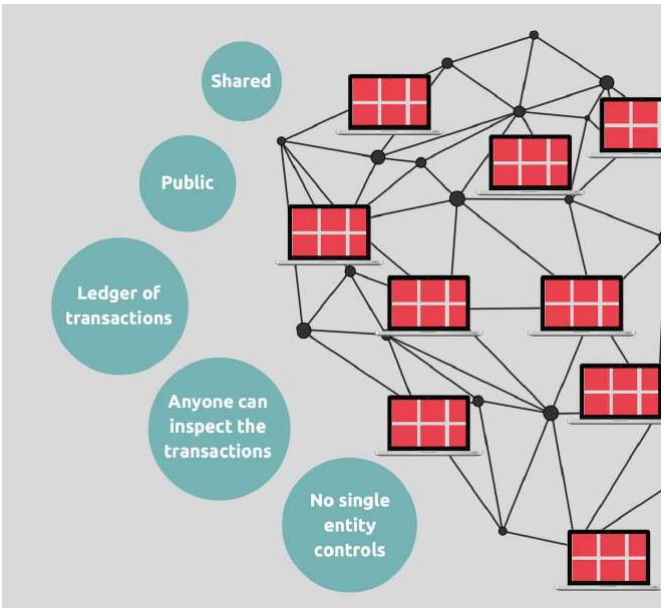
Decision Tree

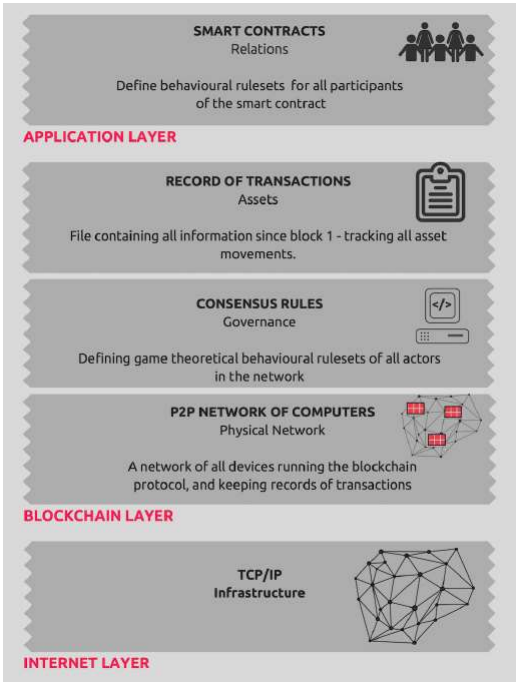


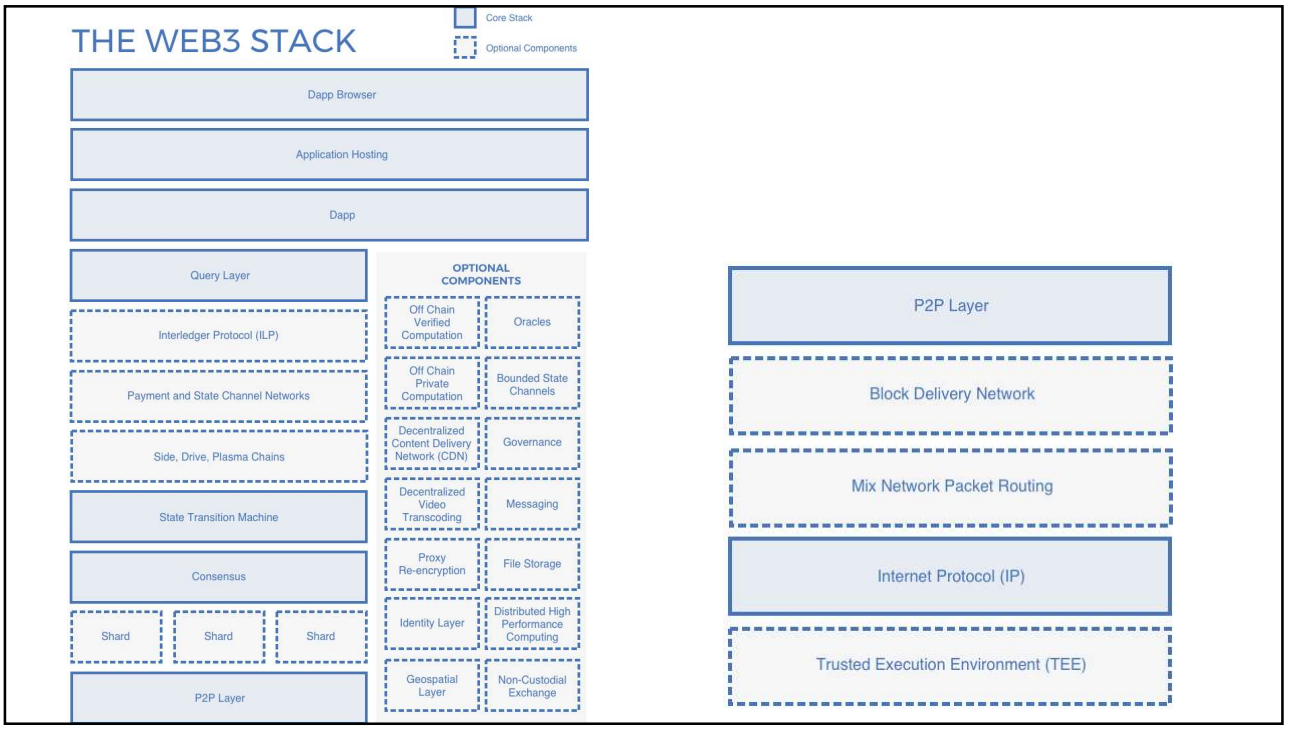
Data Service



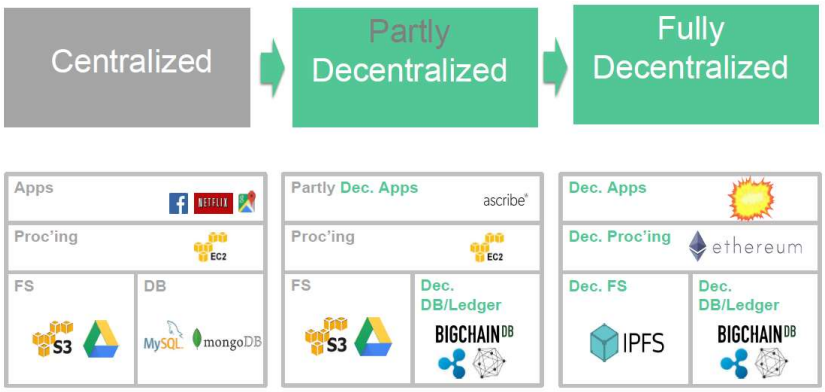
Distributed Ledger




SMART CONTRACTS
Relations
Define behavioural rulesets for all participants of the smart contract
APPLICATION LAYER
RECORD OF TRANSACTIONS
Assets
File containing all information since block 1 - tracking all asset movements.
CONSENSUS RULES
Governance
Defining game theoretical behavioural rulesets of all actors in the network
P2P NETWORK OF COMPUTERS
Physical Network
A network of all devices running the blockchain protocol, and keeping records of transactions
BLOCKCHAIN LAYER
TCP/IP
Infrastructure
INTERNET LAYER

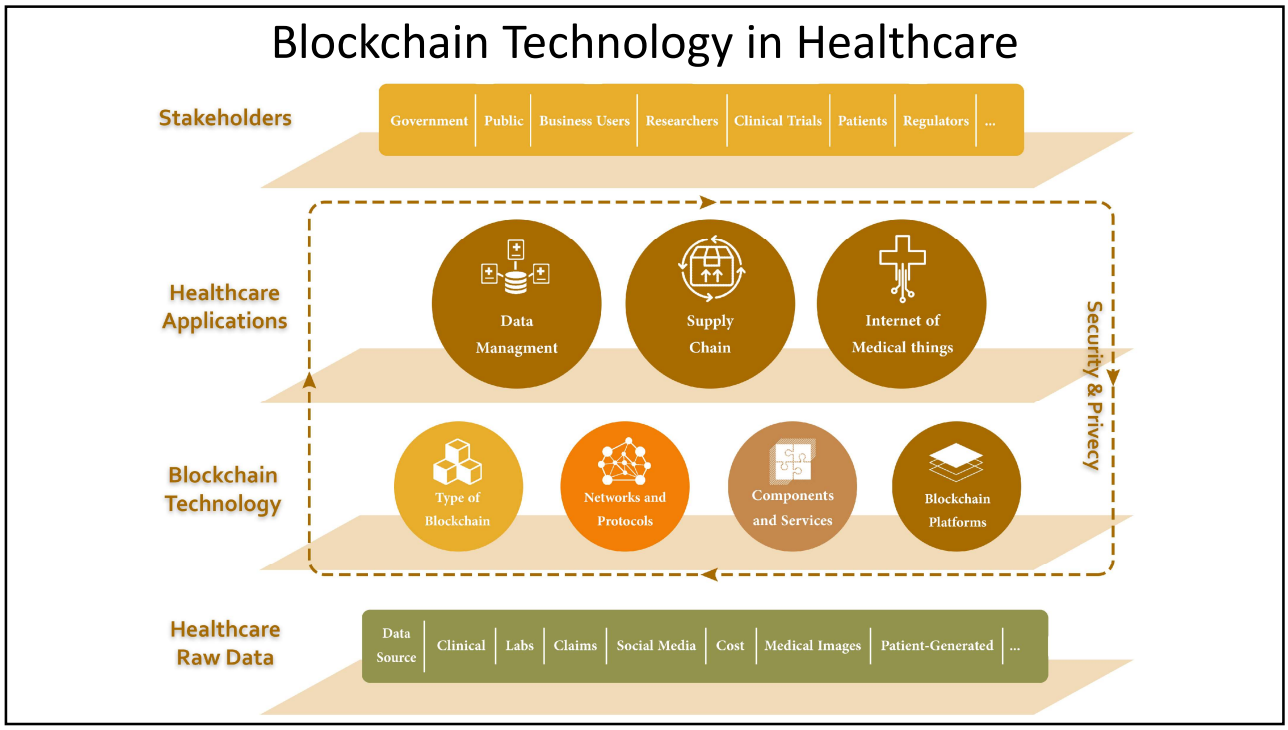

THE WEB3 STACK
Core Stack
Optional Components
Dapp Browser
Application Hosting
Dapp
Query Layer
Interledger Protocol (ILP)
Payment and State Channel Networks
Side, Drive, Plasma Chains
State Transition Machine
Consensus
Shard
Shard
Shard
P2P Layer
OPTIONAL COMPONENTS
Off Chain Verified Computation
Oracles
Off Chain Private Computation
Bounded State Channels
Decentralized Content Delivery Network (CDN)
Governance
Decentralized Video Transcoding
Messaging
Proxy Re-encryption
File Storage
Identity Layer
Distributed High Performance Computing
Geospatial Layer
Non-Custodial Exchange
P2P Layer
Block Delivery Network
Mix Network Packet Routing
Internet Protocol (IP)
Trusted Execution Environment (TEE)

Decentralization of the Cloud



Emergence of applications based on Blockchain

	Web 2.0 apps	Web 3.0 apps (powered by blockchain)
Browser		brave
Storage		STORJ IPFS
Video and audio calls		EXPERTY
Operating system	iOS	essentia.one EOS
Social network		steemit
Messaging		status
Remote job		Ethlance


Blockchain Technology in Healthcare
Stakeholders
Government
Public
Business Users
Researchers
Clinical Trials
Patients
Regulators
...
Healthcare Applications
Data Managment
Supply Chain
Internet of Medical things
Security & Privecy
Blockchain Technology
Type of Blockchain
Networks and Protocols
Components and Services
Blockchain Platforms
Healthcare Raw Data
Data Source
Clinical
Labs
Claims
Social Media
Cost
Medical Images
Patient-Generated
...


AN EVOLUTIONARY VISION
FOR THE ARCHITECT

Zoning

- Service Boundries
  - What happens *between* the zones
  - How services talk to each other
  - Monitor the overall health of system
- Within each service
  - The team who owns that zone may pick a different technology stack or data store
  - Harder to hire people or move them between teams if you have 10 different technology stacks to support

A Principled Approach

- Making decisions in system design is all about trade-offs
- Microservice architectures give us lots of trade-offs to make!
- When picking a datastore, do we pick a platform that we have less experience with, but that gives us better scaling?
- Is it OK for us to have two different technology stacks in our system? What about three?
- Some decisions can be made completely on the spot with information available to us, and these are the easiest to make.
- What about those decisions that might have to be made on incomplete information?

Principles

- Principles are rules you have made in order to align what you are doing to some larger goal, and will sometimes change.
- Example #1:
  - Goal: Decrease the time to market for new features
  - Principle: Delivery teams have full control over the lifecycle of their software to ship whenever they are ready, independently of any other team.
- Example #2
  - Goal: Aggressively grow offering in other countries
  - Principle: the entire system must be portable to allow for it to be deployed locally in order to respect sovereignty of data

Principles

- Heroku's 12 Factors are a set of design principles structured around the goal of helping you create applications that work well on the Heroku platform.
  - https://www.12factor.net

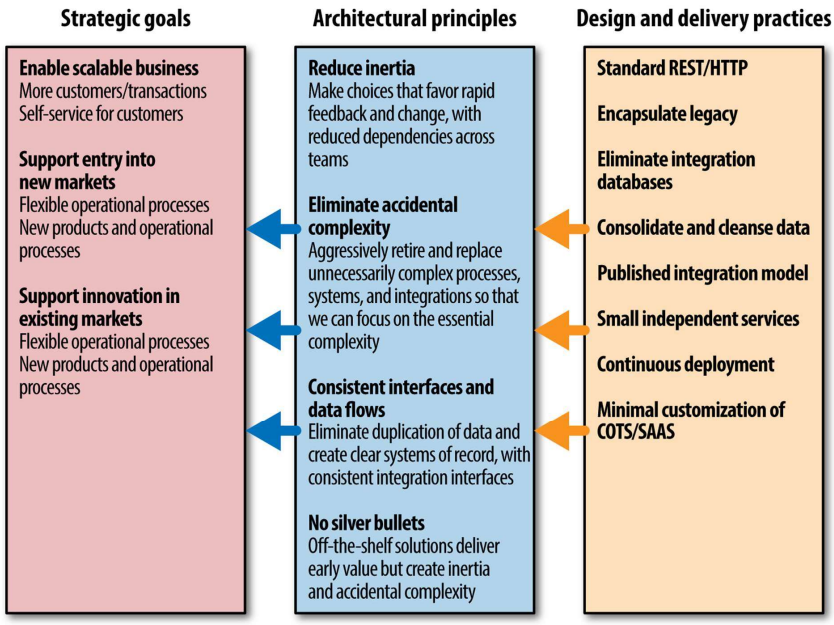
Practices

- Our practices are how we ensure our principles are being carried out.
- They are a set of detailed, practical guidance for performing tasks.
- They will often be technology-specific, and should be low level enough that any developer can understand them.
- Practices could include
  - Coding guidelines
  - All log data needs to be captured centrally
  - HTTP/REST is the standard integration style
- Due to their technical nature, practices will often change more often than principles.

Practices

- As with principles, sometimes practices reflect constraints in your organization.
- For example
  - if you support only CentOS, this will need to be reflected in your practices.
- Practices should underpin our principles.
- A principle stating that delivery teams control the full lifecycle of their systems may mean you have a practice stating that all services are deployed into isolated AWS accounts, providing self-service management of the resources and isolation from other teams.

A Real-World Example



Outline

- **Defining MicroServices**
- MicroServices Explanation
  - Understanding the Monolith
  - Understanding MicroServices
- Practical Considerations

What Are MicroServices

- Best described as
  - Architectural style
  - An alternative to more traditional “monolithic” applications
  - Decomposition of single system into a suite of small services, each running as independent processes and intercommunicating via open protocols
  - With all the benefits and risks this implies

Definitions from the Experts

> Developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API

Martin Fowler

Definitions from the Experts

> Fine-grained SOA

Adrian Cockcroft

MicroServices - Working Definition

> Composing a single application using a suite of small services
  - Rather than a single, monolithic application
  - each running as independent processes
    - Not merely modules or components within a single executable
  - intercommunicating via open protocols
    - e.g. REST on HTTP, Messaging
  - separately written, deployed, scaled and maintained
    - potentially in different languages

MicroServices - Working Definition

> Composing a single application using a suite of small services
  - encapsulating business capability
    - rather than language constructs (classes, packages) as primary way to encapsulate
  - independently replaceable and upgradable

MicroServices are NOT

> The same as SOA
  – SOA is about integrating various enterprise applications
  – MicroServices are mainly about decomposing single applications
> A silver bullet
  – The MicroServices approach involves drawbacks and risks
  – New!
    • You may be using MicroServices now and not know it!

Current Trends

> Twitter moved from Ruby/Rails monolith to MicroServices
> Facebook moved from PHP monolith to MicroServices
> Netflix moved from Java monolith to MicroServices

eBay

- **eBay** started in 1995.
- They are on the 5th generation of their architecture.
- Started as a monolithic Perl application that the founder wrote over a Labor Day weekend in 1995.
- Then it moved to a monolithic C++ application which ended up with 3.4 million lines of code in a single DLL.
- The previous experience spurred the move to a far more distributed partitioned system in Java.
- The eBay of today has quite a bit of Java, but a polyglot set of microservices.

Twitter

- **Twitter**'s evolution looks very similar.
- They are on the 3rd generation of their architecture.
- Started as a monolithic Ruby on Rails application.
- Moved to a combination of Javascript and Rails on the frontend with a lot of Scala on the backend.
- Ultimately they've moved to what we'd call today a set of polyglot microservices.

Amazon

> **Amazon** followed a similar path.
> Started with a monolithic C++ application.
> Then services written in Java and Scala.
> Ending up with a set of polyglot microservices.

Outline

> Defining MicroServices
> **MicroServices Explanation**
  – **Understanding the Monolith**
  – Understanding MicroServices
> Practical Considerations

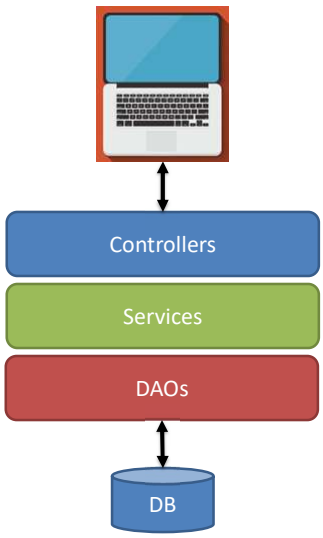
Monolithic Application Example

> Consider a monolithic shopping cart application:
  - Web or mobile interfaces
  - Functions for
    - Searching for products
    - Product catalog
    - Inventory management
    - Shopping cart
    - Checkout
    - Fulfillment

> How would this look like with MicroServices?

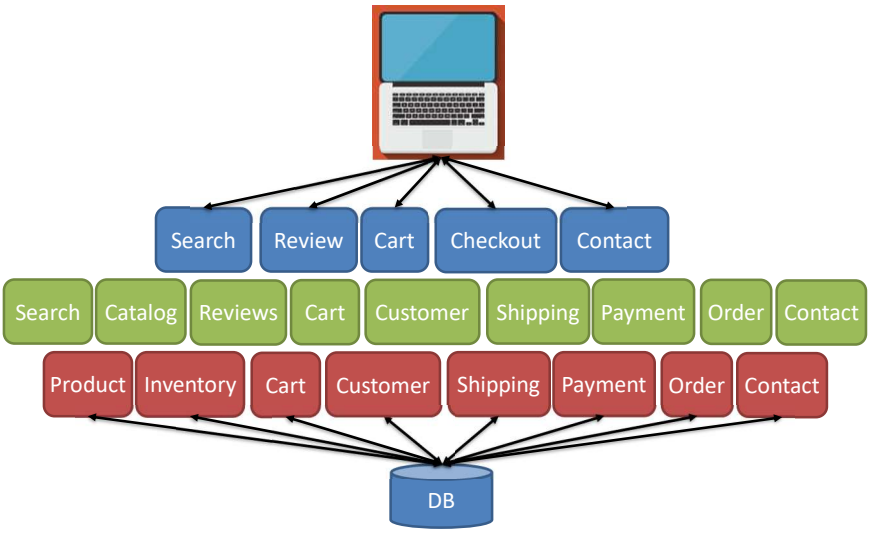
Monolithic Application Example

> Monolithic shopping cart application



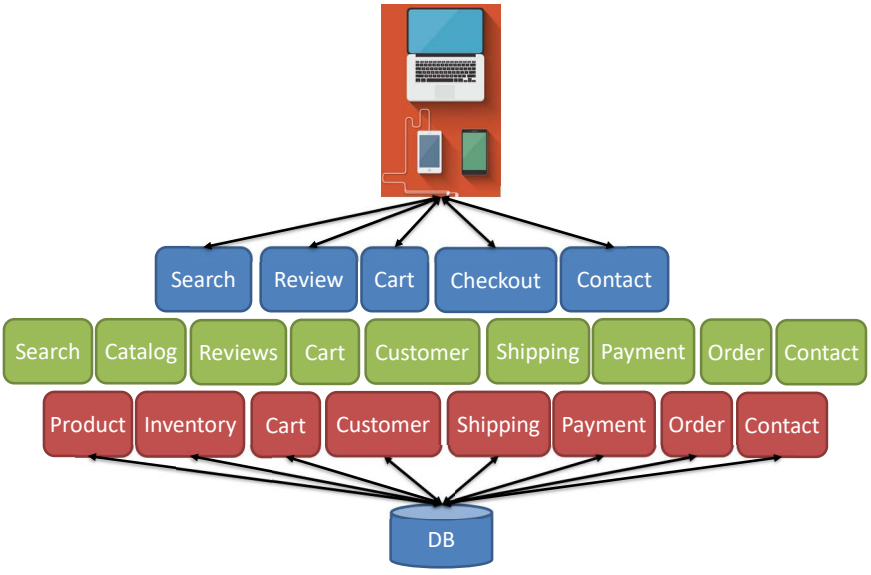
Monolithic Application Example

> Monolithic shopping cart application



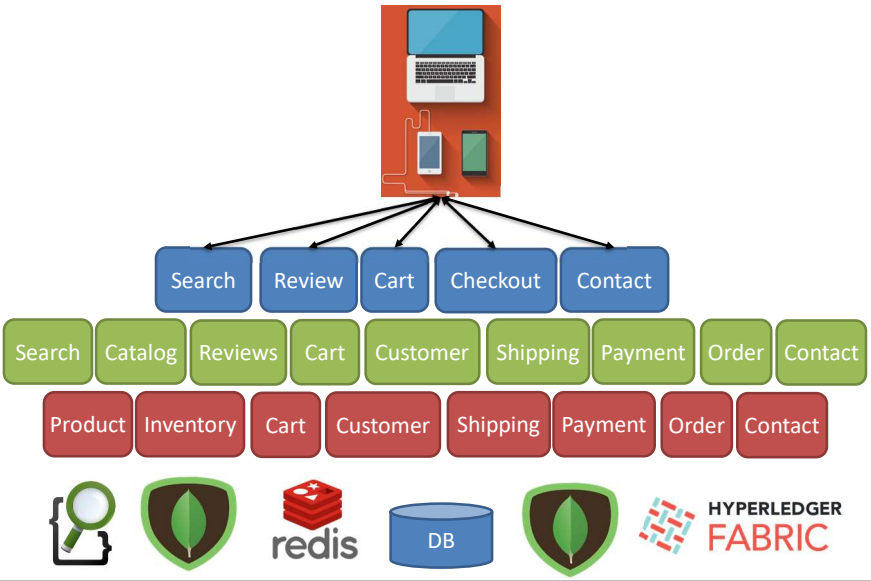
Monolithic Application Example

> New types of client applications



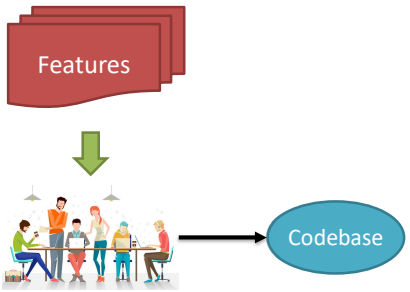
Monolithic Application Example

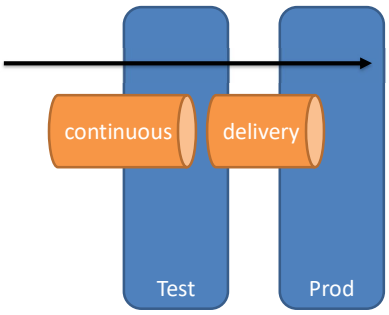
> New types of persistence services



Monolithic Challenges

> Single Codebase, Deployment, Versioning, Team Size

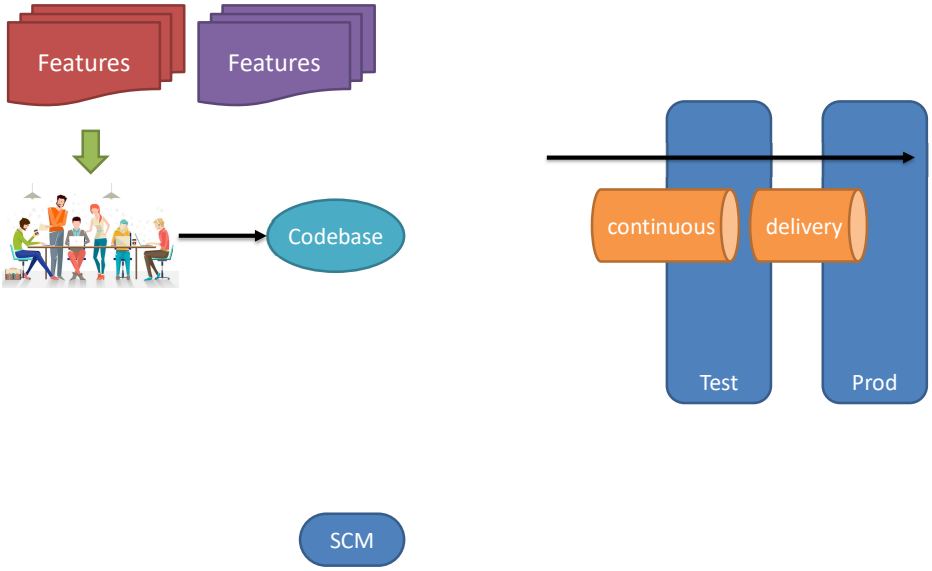




SCM

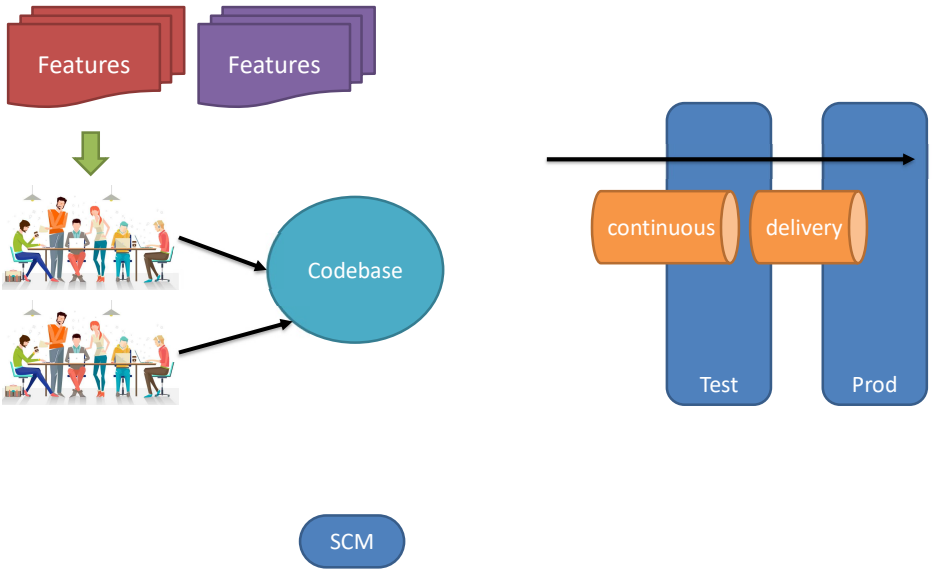
Monolithic Challenges

> Single Codebase, Deployment, Versioning, Team Size



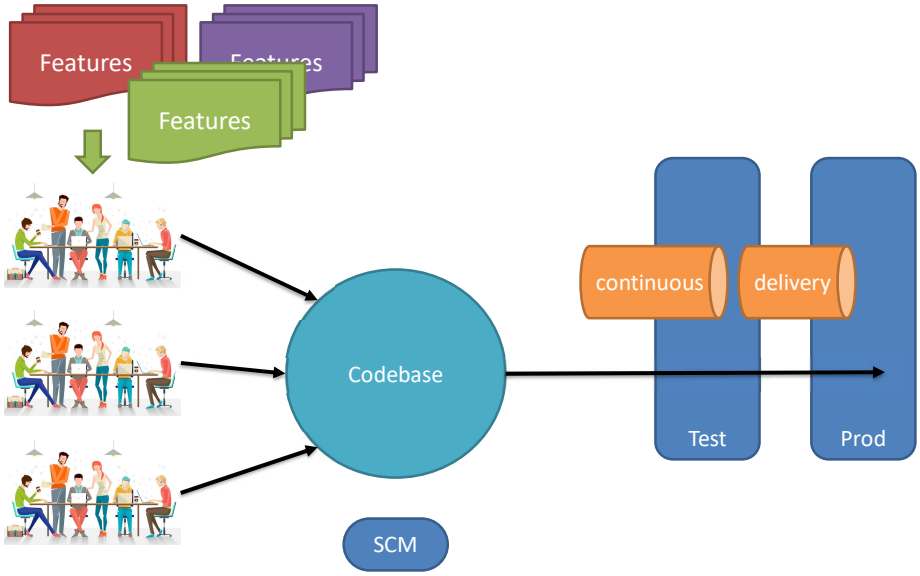
Monolithic Challenges

> Single Codebase, Deployment, Versioning, Team Size



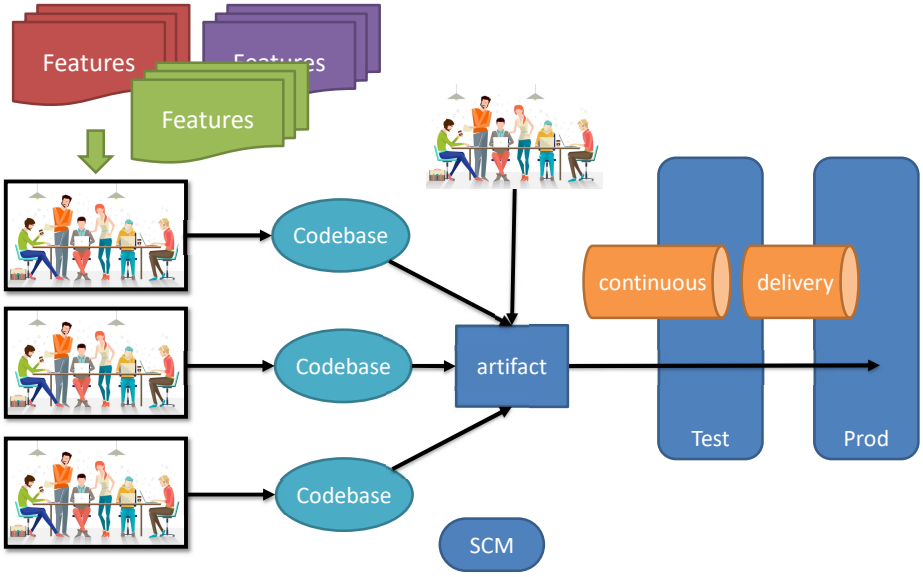
Monolithic Challenges

> Single Codebase, Deployment, Versioning, Team Size



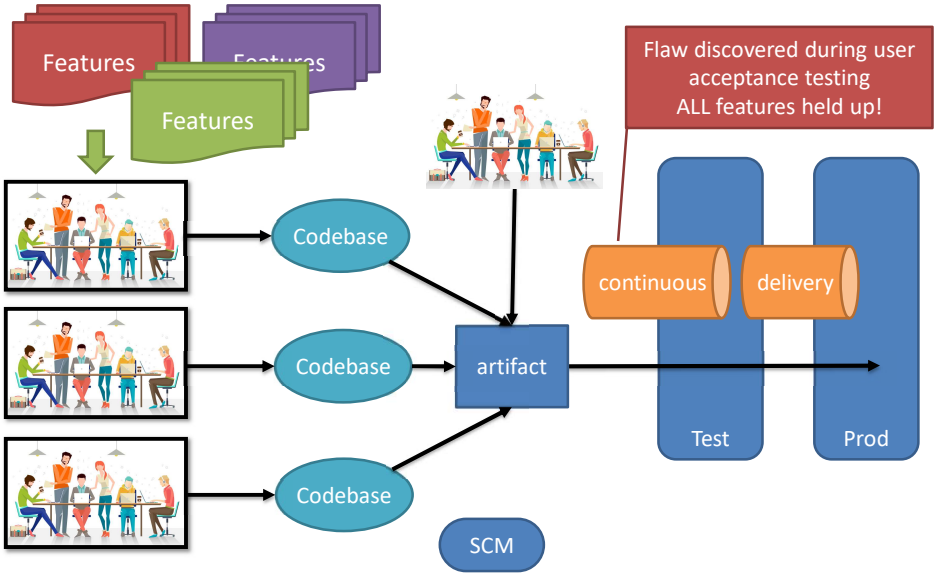
Monolithic Challenges

> Single Codebase, Deployment, Versioning, Team Size



Monolithic Challenges

> Single Codebase, Deployment, Versioning, Team Size



Understanding Monolithic Implementation

> Single application executable
  - Easy to comprehend, but not to digest
  - Must be written in a single language

> Modularity based on Program Language
  - Using the constructs available in that language
    - Packages, classes, functions, namespaces, frameworks
  - Various storage or service technologies used
    - RDBMS, Messaging, eMail, etc

Monolithic Advantages

- Easy to comprehend (but not digest)
- Easy to test as a single unit (up to a size limit)
- Easy to deploy as a single unit
- Easy to manage (up to a size limit)
- Easy to manage changes (up to a point)
- Easy to scale (when care is taken)
- Complexity managed by language constructs

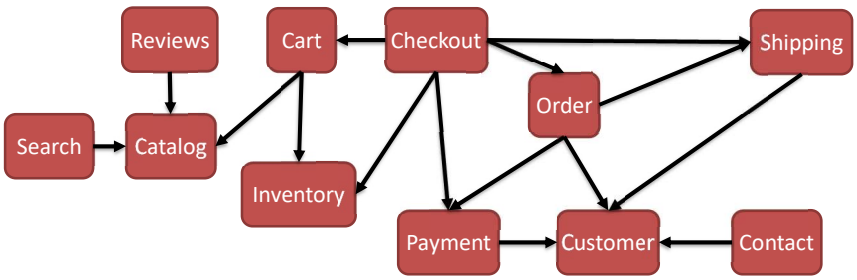
Monolithic Drawbacks

- Language or Framework Lock
  - Enterprise App's written with single technology stack
  - Cannot experiment/take advantage of emerging technologies
- Digestion
  - Single developer cannot digest a large codebase
  - Single team cannot manage a single large application
    - Amazon's "2 Pizza" rule
- Deployment as single unit
  - Cannot independently deploy single change to single component
  - Changes are "held-hostage" by other changes

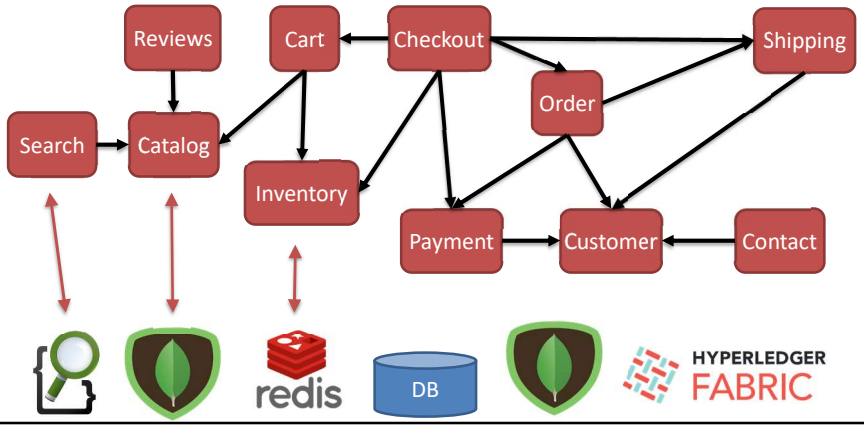
Outline

- Defining MicroServices
- **MicroServices Explanation**
  - Understanding the Monolith
  - **Understanding MicroServices**
- Practical Considerations

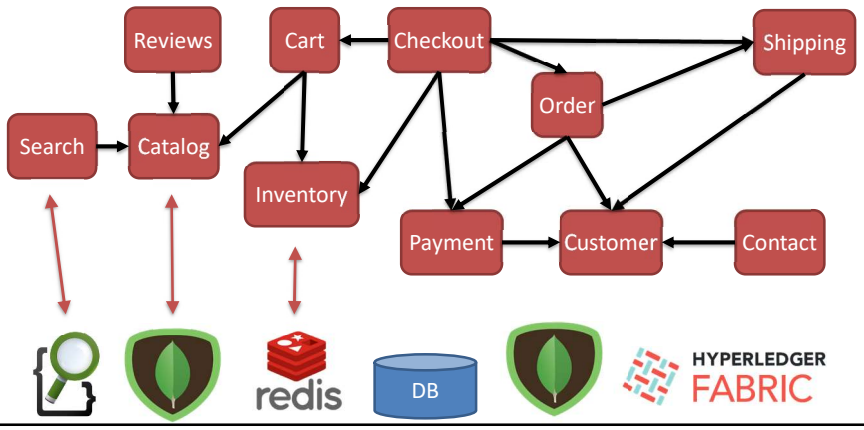
Enter MicroServices Architecture



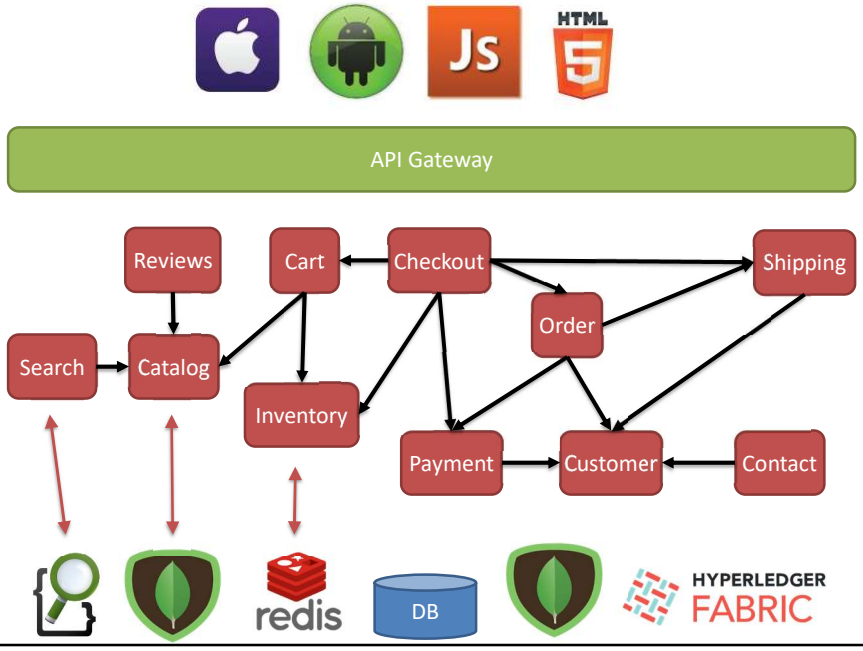
Enter MicroServices Architecture



Enter MicroServices Architecture



Enter MicroServices Architecture



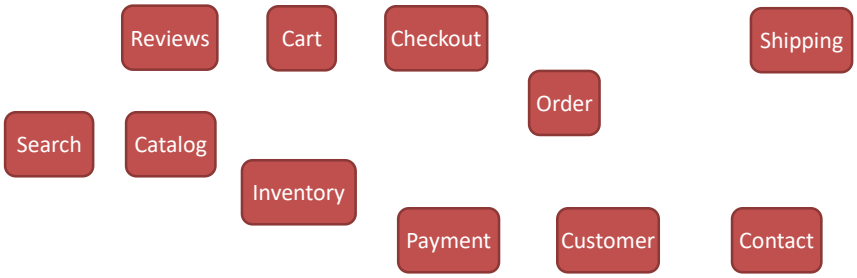
Componentization via Services

> NOT language constructs

> Services are small, *independently deployable* **applications**

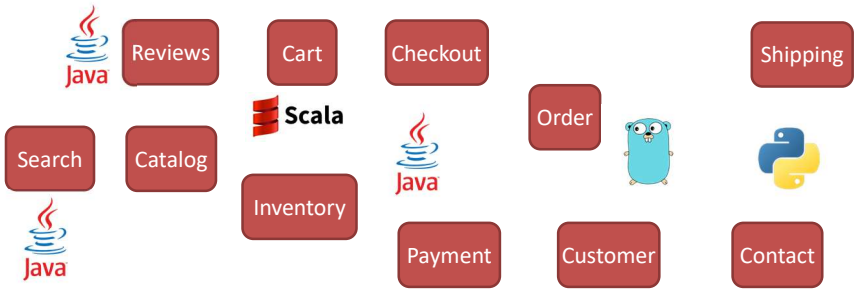
> Forces the design of clear interfaces

> Changes scoped to their affected service



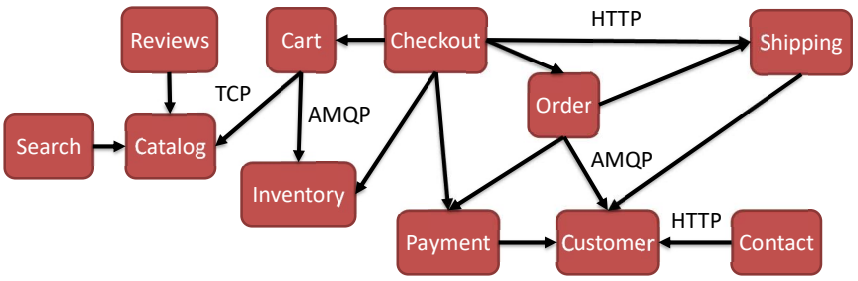
MicroServices: Composed using suite of small services

> Services are small, independently deployable applications
  – Not a single codebase
  – Not (necessarily) a single language or framework
  – Modularization not based on language or framework constructs



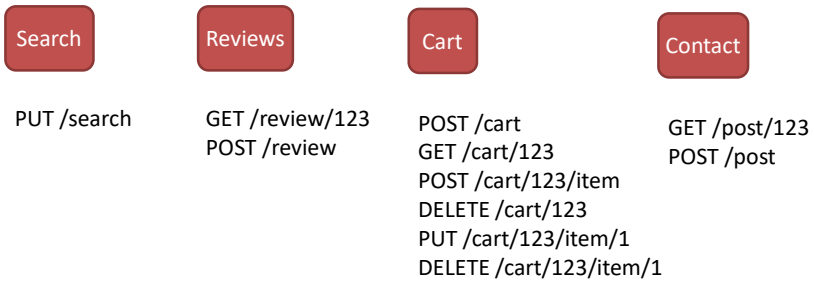
MicroServices: Communication based on lightweight protocols

> HTTP, TCP, UDP, WebSocket, Messaging, etc.
  – Payloads: JSON, BSON, XML, …
> Forces the design of clear interfaces
> Netflix's Cloud Native Architecture: Communicate via APIs
  – NOT Common Database



MicroServices: Services encapsulate business capabilities

> Not based on technology stack
> Vertical slices by business function (e.g. cart, catalog, checkout)
  – Though technology chunk also practical
> Suitable for cross-functional teams



MicroServices: Services easily managed

> Easy to comprehend, alter, test, version, deploy, manage, overhaul, replace
  – By small, cross-functional teams, or even individuals

Decentralized Governance

- Use the right tool (language, framework) for the job
- Services evolve at different speeds, deployed and managed according to different needs
- Make services be “Tolerant Readers”
- Consumer-Driven Contracts
- Antithesis of ESB
  - Services are not Orchestrated, but Choreographed

Polyglot Persistence

- Freedom to use the best technology for the job
  - Don’t assume single RDBMS is always best
  - Very controversial! Many DBAs will not like this!
    - Non pan-enterprise data model!
    - No transactions!

MicroService Advantages

- Easy to digest each service (difficult to comprehend whole)
- VERY easy to test, deploy, manage, version, and scale single services
- Change cycle decoupled
- Easier to scale staff
- No Language or Framework lock

Challenges with MicroServices

- Complexity has moved out of the application, but into the operations layer
  - Fallacies of Distributed Computing
- Services may be unavailable
  - Never needed to worry about this in a monolith!
  - Design for failure, circuit breakers
    - Everything fails all the time
  - Much more monitoring needed
- Remote calls more expensive than in-process calls

Challenges with MicroServices

- Transactions: Must rely on eventual consistency over ACID
- Features span multiple services
- Change management becomes a different challenge
  - Need to consider the interaction of services
  - Dependency management/versioning
- Refactoring Module Boundaries

Fallacies of Distributed Computing

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology does not change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

Outline

- Defining MicroServices
- MicroServices Explanation
  - Understanding the Monolith
  - Understanding MicroServices
- **Practical Considerations**

How do you break a Monolith into MicroServices

- Primary consideration: business functionality
  - Noun-based
  - Verb-based
  - Single Responsibility Principle
  - Bounded Context

How Micro is Micro

- Size is not the compelling factor
- Small enough for an individual developer to digest
- Small enough to be built and managed by small team
  - Amazon's two pizza rule
- Documentation small enough to read and understand
- Predictable
  - Easy to experiment with

INTRODUCING SERVICE-ORIENTED ARCHITECTURE CONCEPTS

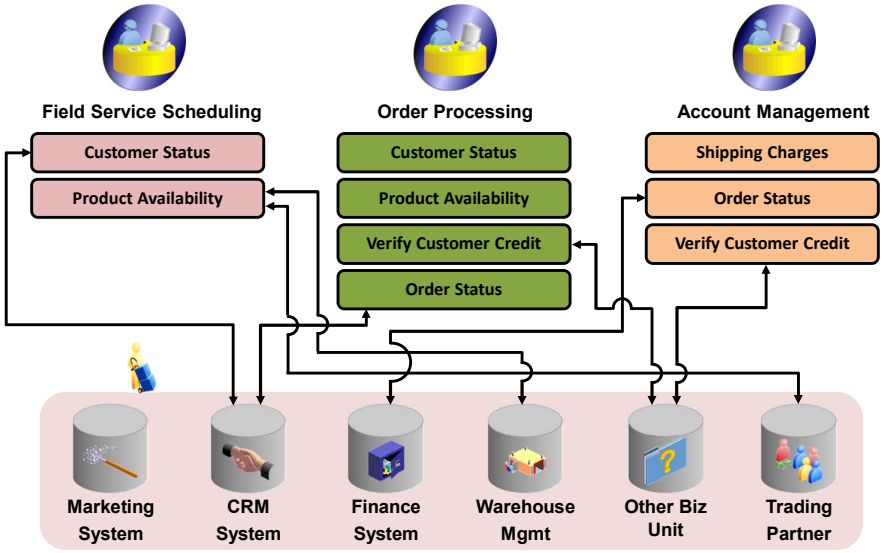
Roadmap

- What is SOA?
- SOA building blocks
- Standards used in SOA

The Problem with Enterprise Integration

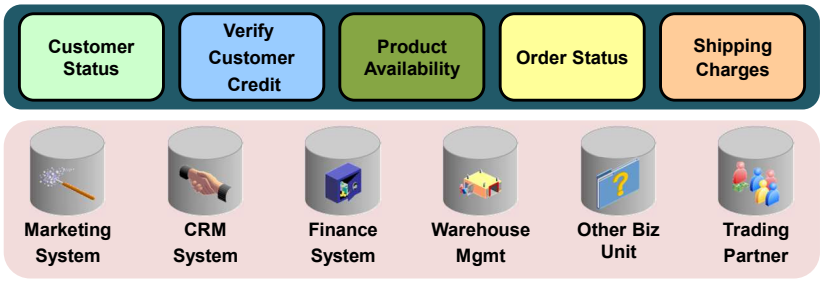
> Misalignment of business and IT
> Duplication of data and functionality
> Process/organizational silos

Example of Point-to-Point Integration

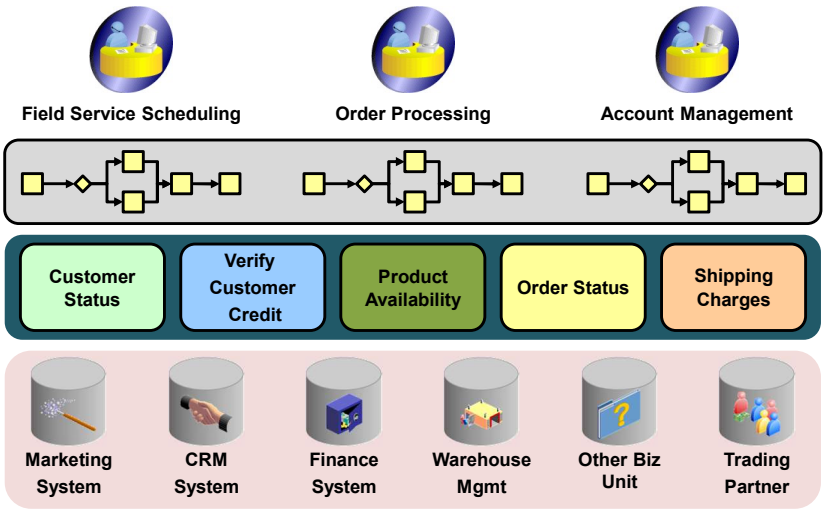


Definition of SOA

SOA is an IT strategy that organizes the discrete functions contained in enterprise applications into interoperable, standards-based services to be combined and reused quickly, within and between enterprises, to meet business requirements.



SOA Integration Solution



Characteristics of SOA

- Business-driven
- Standards-based
- Composition-centric

Benefits of Adopting SOA

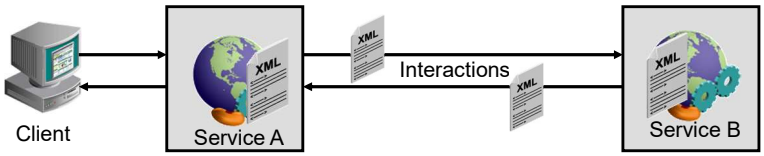
> Improved business agility
  - Align IT with the business.
  - Remove barriers between business units and business partners.
> Lower cost of maintaining IT systems
  - Speed up delivery of applications to meet business demands.
  - Protect IT investments by reusing the existing infrastructure.

Roadmap

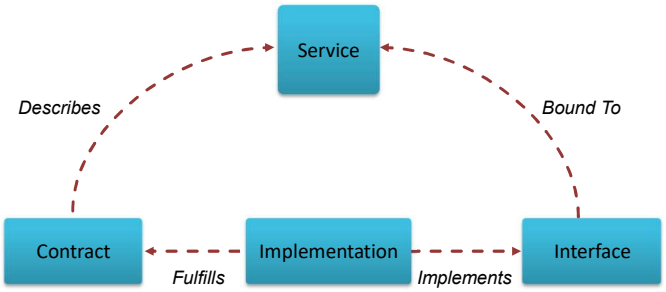
- What is SOA?
- SOA building blocks
- Standards used in SOA

What Is a Service?

> Services are:
  - IT's representation of business functionality
  - Described by a well-defined *interface*
  - Abstracted from the implementation
  - Accessed using standard protocols (the glue) to enable interoperability from decoupled functions
> Services are SOA building blocks.



Elements of a Service

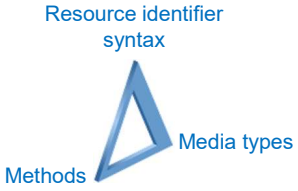


SOAP-Based Web Services

- Provide a simple and interoperable messaging framework
- Rely on common standards that include:
  - SOAP: A standard format for messaging over a network
  - Web Service Description Language (WSDL): The language that provides a description for web services
  - Universal Description, Discovery, and Integration (UDDI): A web-based distributed directory to publish and locate information about web services
- Include additional specifications (WS-*) to define functionality for web services discovery, security, reliability, transactions, and management

Representational State Transfer (REST) Web Services

- REST is a model based on resources.
- RESTful services rely on the HTTP protocol.
- REST is based on three fundamental elements:
  - Resource identifier syntax – Represent the actual resources that a service exposes: URI
  - Methods – Protocol mechanisms used to transfer the data: GET, PUT, POST, DELETE
  - Media types – Type of data is being transferred: XML, JSON



SOA, Web Services, and REST

- SOA is an architectural approach to composition and integration.
- SOAP-based web services is the most common way to implement SOA. They:
  - Address the low-level interactions between services
  - Help up to a certain level of complexity in the infrastructure
- REST architecture provides an alternative medium by which SOA can be implemented.
  - REST tries to establish application architecture that emulates the World Wide Web.

Other SOA Building Blocks

- Service Registry and Enterprise Asset Repository
- Enterprise Service Bus (ESB)
- Event processing
- Business Activity Monitoring (BAM)
- Security
- Design and development tooling

Service Registry and Enterprise Asset Repository

> Problem being addressed: Manage services' life cycle and portfolio

> Service registry and enterprise asset repository:

– Store, manage, and publish service metadata

– Govern the life cycle of these services

Enterprise Service Bus

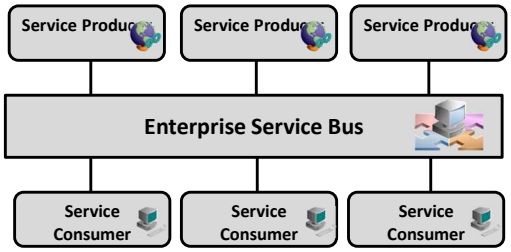
> Problem being addressed: Integrate different middleware products and provide interoperability with web services

> Enterprise Service Bus (ESB):

– Facilitates interactions between disparate partners without coupling them in a tight manner

– Offers the following capabilities:

- Message routing
- Transformation
- Mediation
- Virtualization



Event-Driven Architecture (EDA)

- Problem being addressed: Detect anomalies, risks, or opportunities in a business from collected business information and aggregated data.
- EDA:
  - A SOA extension
  - Based on publish-subscribe concepts
  - Related to processing and routing of asynchronous events
- Complex event processing (CEP)
  - A specific implementation of EDA
  - Focuses on identifying patterns in a large number of events and their contents

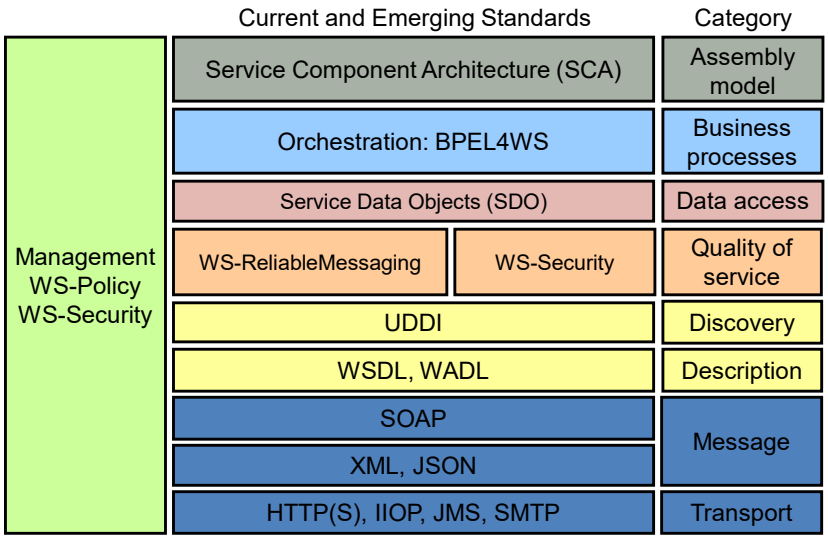
Business Activity Monitoring (BAM)

- Problems being addressed:
  - Detecting bottlenecks and other critical situations as soon as possible
  - Gaining operational insights into current business activities
- BAM:
  - Provides a real-time operational view of the business processes and activities within an organization
  - Allows defining alerts and notifications to be automatically notified when a critical situation occurs

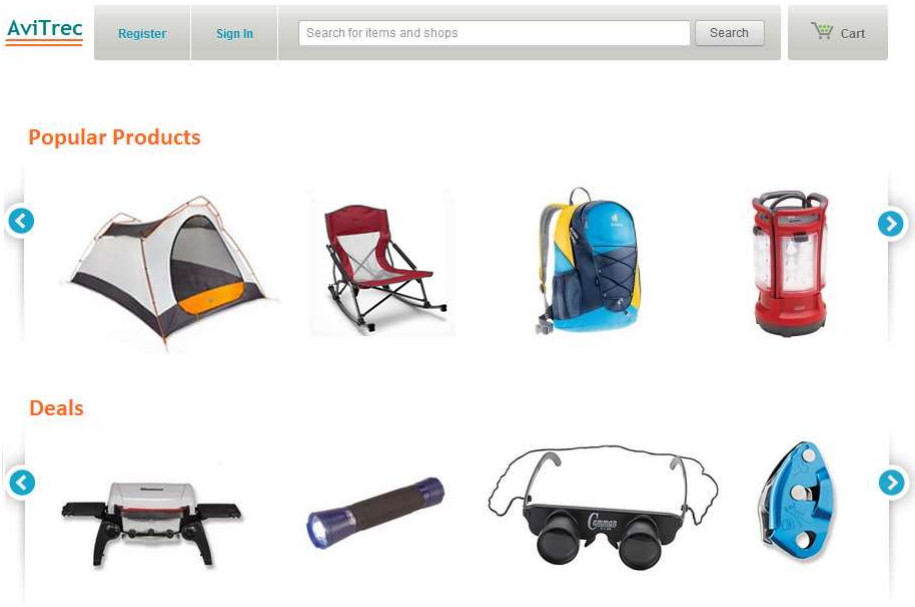
Roadmap

- What is SOA?
- SOA building blocks
- Standards used in SOA

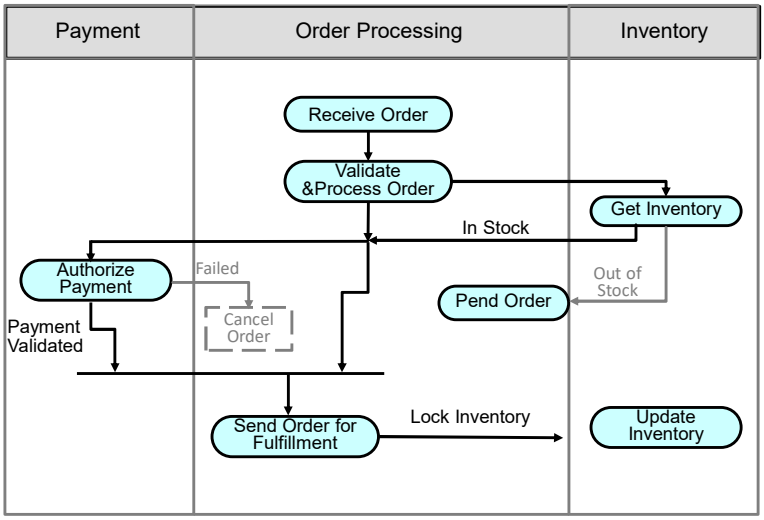
Standards Used in SOA



Case Study

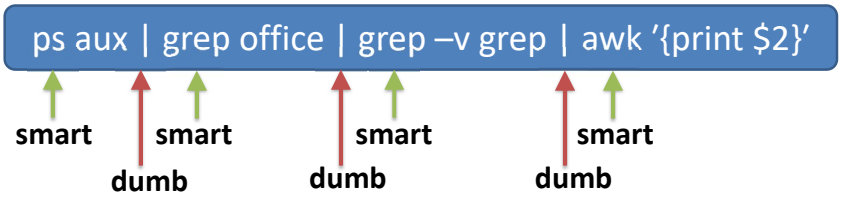


Order Management Workflow

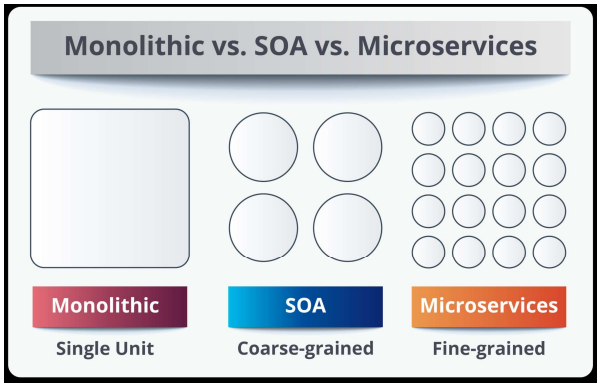


Differences with SOA

> SOA addresses integration between systems
  – MicroServices address individual applications
> SOA relies on orchestration
  – MicroServices rely on choreography
> SOA relies on smart integration technology, dumb services
  – MicroServices rely on smart services, dumb integration technology
  – Consider: Linux commands, pipes and filters

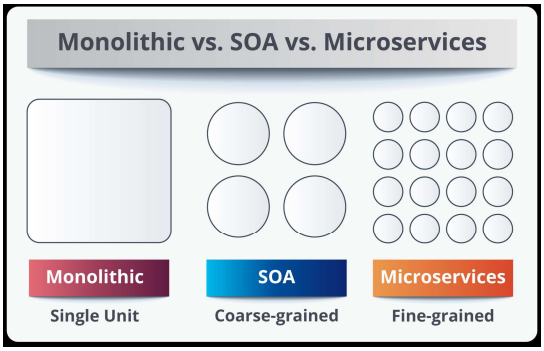


MicroServices vs SOA



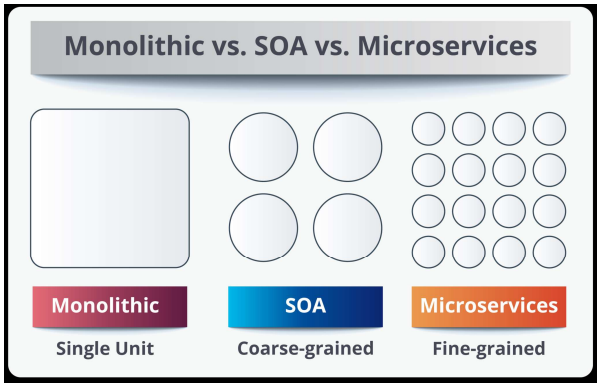
> A **monolith** is similar to a big container wherein all the software components of an application are assembled together and tightly packaged.

MicroServices vs SOA



- **Service-oriented architecture** is essentially a collection of services.
- These services communicate with each other.
- The communication can involve either simple data passing or two or more services coordinating some activity.

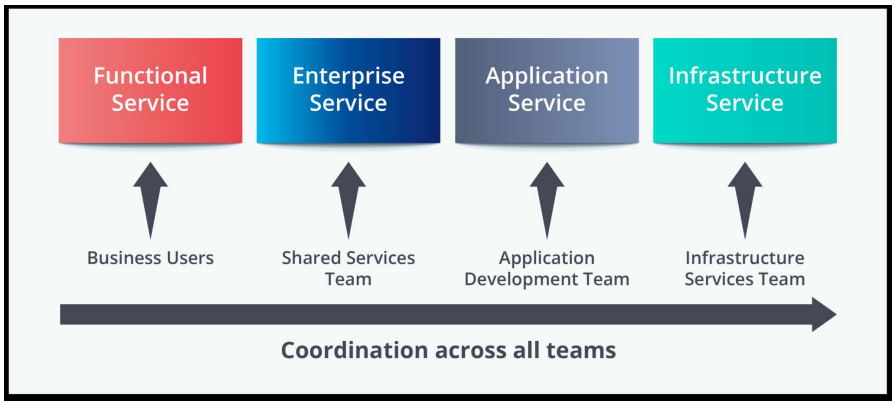
MicroServices vs SOA



- **Microservices** is an architectural style that structures an application as a collection of small autonomous services modeled around a business domain.

Service Oriented Architecture

> SOA defines four basic service types



Service Oriented Architecture

Business Services:

> Coarse-grained services that define core business operations.

> Represented through XML, Business Process Execution Language (BPEL), and others.

Service Oriented Architecture

Enterprise Services:

> Implement the functionality defined by business services.

> Mainly rely on application services and infrastructure services to fulfill business requests.

Service Oriented Architecture

Application Services:

> Fine-grained services that are confined to a specific application context.

> A dedicated user interface can directly invoke the services.

Service Oriented Architecture

Infrastructure Services:

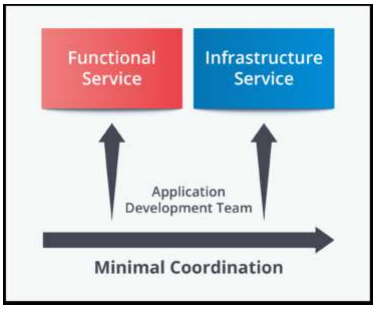
> Implement non-functional tasks such as authentication, auditing, security, and logging.

> Can be invoked from either application services or enterprise services.

MicroService Architecture

> Microservices have limited service taxonomy.

> They consist of two service types



MicroService Architecture

Functional Services:

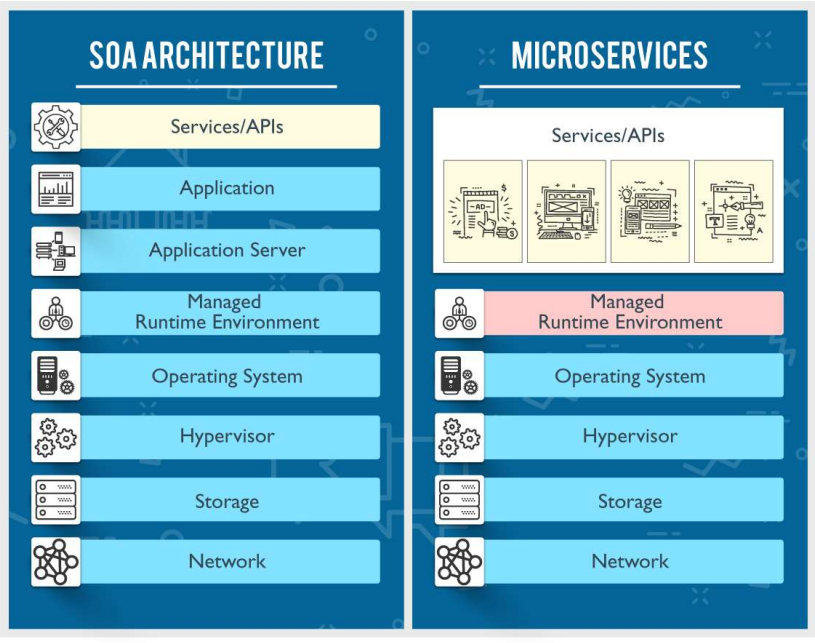
> Support specific business operations.

> Accessing of services is done externally and these services are not shared with other services.

> As in SOA, infrastructure services implement tasks such as auditing, security, and logging.

> In this, the services are not unveiled to the outside world.

Major Differences Between SOA and MSA



Major Differences Between SOA and MSA

SOA	MSA
Follows "**share-as-much-as-possible**" architecture approach	Follows "**share-as-little-as-possible**" architecture approach
Importance is on **business functionality** reuse	Importance is on the concept of "**bounded context**"
They have **common governance** and **standards**	They focus on **people, collaboration** and freedom of other options
Uses **Enterprise Service bus (ESB)** for communication	Simple messaging system
They support **multiple message protocols**	They use **lightweight protocols** such as **HTTP/REST** etc.
Multi-threaded with more overheads to handle I/O	**Single-threaded** usually with the use of Event Loop features for non-locking I/O handling
Maximizes application service reusability	Focuses on **decoupling**
Traditional Relational Databases are more often used	**Modern Relational Databases** are more often used
A systematic change requires modifying the monolith	A systematic change is to create a new service
DevOps / Continuous Delivery is becoming popular, but not yet mainstream	Strong focus on DevOps / Continuous Delivery

Major Differences Between MSA and SOA in Detail

Service Granularity

> Service components within a microservices architecture are generally single-purpose services that do one thing really, really well.

> With SOA, service components can range in size anywhere from small application services to very large enterprise services.

> It is common to have a service component within SOA represented by a large product or even a subsystem.

Major Differences Between MSA and SOA in Detail

Component Sharing

> Component sharing is one of the core tenets of SOA.

> Component sharing is what enterprise services are all about.

> SOA enhances component sharing, whereas MSA tries to minimize on sharing through “bounded context.”

> A bounded context refers to the coupling of a component and its data as a single unit with minimal dependencies.

> As SOA relies on multiple services to fulfill a business request, systems built on SOA are likely to be slower than MSA.

Major Differences Between MSA and SOA in Detail

Middleware vs API layer

> The microservices architecture pattern typically has what is known as an API layer, whereas SOA has a messaging middleware component.

> The messaging middleware in SOA offers a host of additional capabilities not found in MSA, including mediation and routing, message enhancement, message, and protocol transformation.

> MSA has an API layer between services and service consumers.

Major Differences Between MSA and SOA in Detail

Remote services

> SOA architectures rely on messaging (AMQP, MSMQ) and SOAP as primary remote access protocols.

> Most MSAs rely on two protocols – REST and simple messaging (JMS, MSMQ), and the protocol found in MSA is usually homogeneous.

Major Differences Between MSA and SOA in Detail

Heterogeneous interoperability

> SOA promotes the propagation of multiple heterogeneous protocols through its messaging middleware component.

> MSA attempts to simplify the architecture pattern by reducing the number of choices for integration.

> If you would like to integrate several systems using different protocols in a heterogeneous environment, you need to consider SOA.

> If all your services could be exposed and accessed through the same remote access protocol, then MSA is a better option.

Summary

- MicroServices are an architectural style
  - Decomposition of single system into independent running, intercommunicating services
  - Alternative to Monolithic applications
- MicroServices have advantages and disadvantages
  - As do monoliths