IBM Training
Red Hat OpenShift Application Development

—

Luca Floris, Grzegor Smolko
IBM Cloud Technical Consultant

Agenda

Day 4

DevOps: Continuous Delivery
- DevOps and DevSecOps
- WebSphere Liberty on OpenShift
- Lab 13: Set up a CI/CD pipeline on OpenShift using Jenkins to deploy a simple web application
- Transformation Advisor - Migrating WebSphere Applications to OpenShift

Microservices Architecture
- Microservices application architecture
- Developing microservices
- Twelve factor applications
- Refactoring monolith applications into microservices
- Lab 14 Build and deploy a polyglot microservices application on OpenShift

Microservices application architecture

Cloud native

An application architecture designed to use the strengths and accommodate the challenges of a standardized cloud environment, including:

- Elastic scaling
- Immutable deployment
- Disposable instances
- Less predictable infrastructure

What it means to be cloud native

- Clean contract with underlying OS to ensure maximum portability
- Scale elastically without significant changes to tooling, architecture, or development practices
- Resilient to inevitable failures in the infrastructure and application
- Instrumented to provide both technical and business insight
- Use cloud services such as storage, queuing, and caching
- Rapid and repeatable deployments to maximize agility
- Automated setup to minimize time and cost for new developers

Microservices: Using technology more efficiently

- Microservices is an application architectural style
  - The application is composed of microservice components
- Microservice, a component in this architecture
  - Each is a miniature application
  - Each is focused on one task, a business capability
    - The Single Responsibility principle
  - Each can be deployed and updated independently
  - They are loosely coupled
  - Each has a well-defined interface
    - REST APIs

Microservices: Making developers more efficient

- Each microservice is developed and deployed by a small team
  - The team owns the entire lifecycle of the service
- Microservices accelerate delivery
  - Minimize communication and coordination between people
  - Reducing the scope and risk of change
- Microservices facilitate agile development
  - They make their development teams loosely coupled

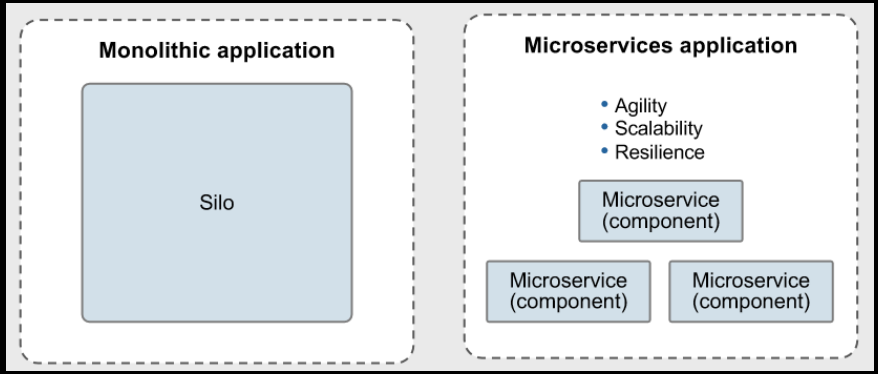
Microservices are an evolution

Evolution of architectural styles

- Monolithic
  One large application that does everything
- Microservices
  Several smaller applications that each does part of the whole

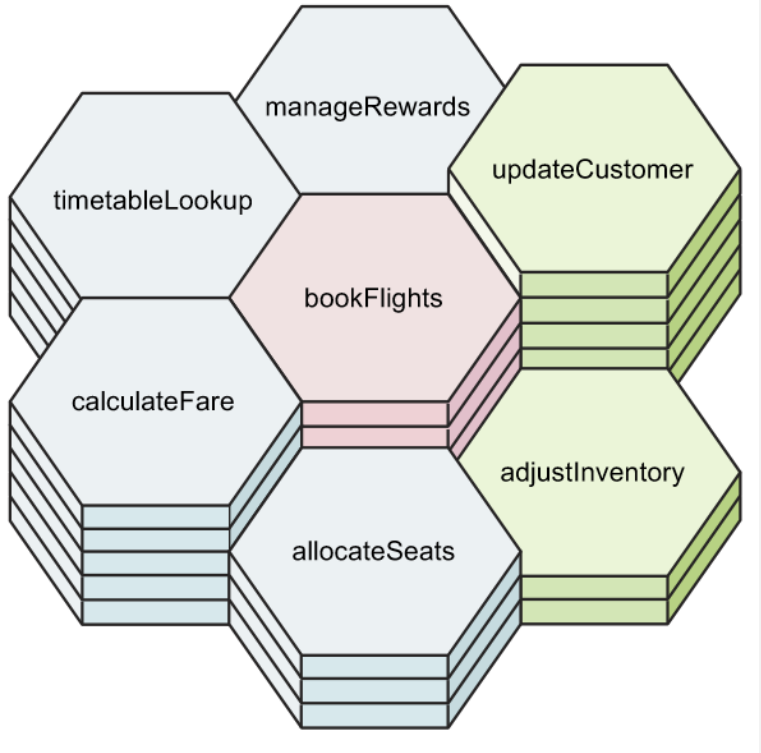
Evolution of service orientation

- SOA
  Focused on reuse, technical integration issues, technical APIs
- Microservices
  Focused on functional decomposition, business capabilities, business APIs



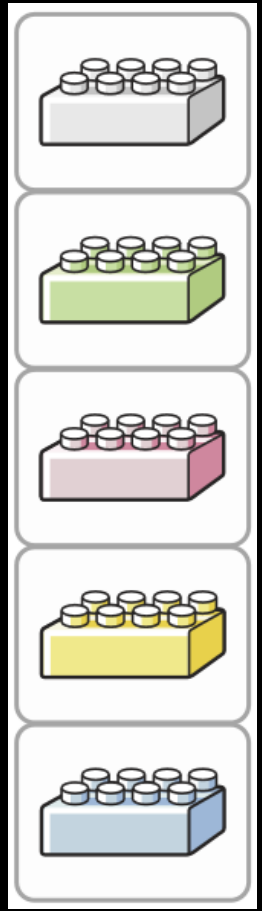
Sample application that uses microservices

- Airline reservation application
  - Seven services (in this example)
- Each service includes
  - Logging
  - Metrics
  - Health check
  - Service endpoint
  - Service registry
  - Service management
- What do the different colors of the tiles mean (red, blue, and green)?
- The tiles are in stacks, some higher than others. What does the height mean?



Key tenets of a microservices architecture

1. Large monoliths are broken down into many small services
   - Each service runs in its own process
   - One service per container
2. Services are optimized for a single function
   - There is only one business purpose per service
   - The Single Responsibility Principle
3. Communication via REST API and message brokers
   - Avoid tight coupling introduced by communication through a database
4. Per-service continuous integration and continuous deployment (CI/CD)
   - Services evolve at different rates
   - You let the system evolve but set architectural principles to guide that evolution
5. Per-service high availability (HA) and clustering decisions
   - One size or scaling policy is not appropriate for all
   - Not all services need to scale; others require autoscaling up to large numbers



Advantages of microservices

- Developed independently
  - Limited, explicit dependencies on other services
- Developed by a single team
  - The team is small
  - All team members can understand the entire code base
- Developed on its own timetable
  - New versions delivered independently of other services
- Polyglot: Each can be developed in a different language
  - Select the best language
- Manages its own data
  - Select the best technology and schema
- Scales and fails independently
  - Isolates problems

Comparing monolithic and microservices architectures

Category	Monolithic architecture	Microservices architecture
Architecture	Built as a single logical executable	Built as a suite of small services
Modularity	Based on language features	Based on business capabilities
Agility	Changes require building and deploying a new version of the entire application	Changes can be applied to each service independently
Scaling	Entire application scaled when only one part is the bottleneck	Each service scaled independently when needed
Implementation	Typically entirely developed in one programming language	Each service can be developed in a different programming language
Maintainability	Large code base is intimidating to new developers	Smaller code bases easier to manage
Deployment	Complex deployments with maintenance windows and scheduled downtimes	Simple deployment as each service can be deployed individually, with minimal downtime

Technology advances have made microservices possible

- Ease and feasibility of distributing components
  - Internet, intranet, or network maturity
  - RESTful API conventions or perceived simplicity, and lightweight messaging
- Ease and simplicity of hosting
  - Lightweight runtimes − Examples: Node.js and WebSphere Liberty
  - Simplified infrastructure
    − OS virtualization (hypervisors), containerization (Docker), infrastructure as a service (cloud infrastructure)
    − Workload virtualization (examples: Cloud Foundry, Kubernetes, OpenWhisk, Swarm)
  - Platform as a service
    − Autoscaling, SLA management, messaging, caching, build management
- Agile development methods
  - Examples: IBM Cloud Garage Method, XP, TDD, Scrum, Continuous Delivery
  - Standardized code management, such as GitHub

Microservice challenges

- Greater operational complexity because there are more moving parts to monitor and manage
- Developers must have significant operational skills (DevOps)
- Service interfaces and versions
- Duplication of effort across service implementations
- Extra complexity of creating a distributed system
  - Network latency
  - Fault tolerance
  - Serialization
- Designing decoupled, non-transactional systems is difficult
- Locating service instances
- Maintaining availability and consistency with partitioned data
- End-to-end testing

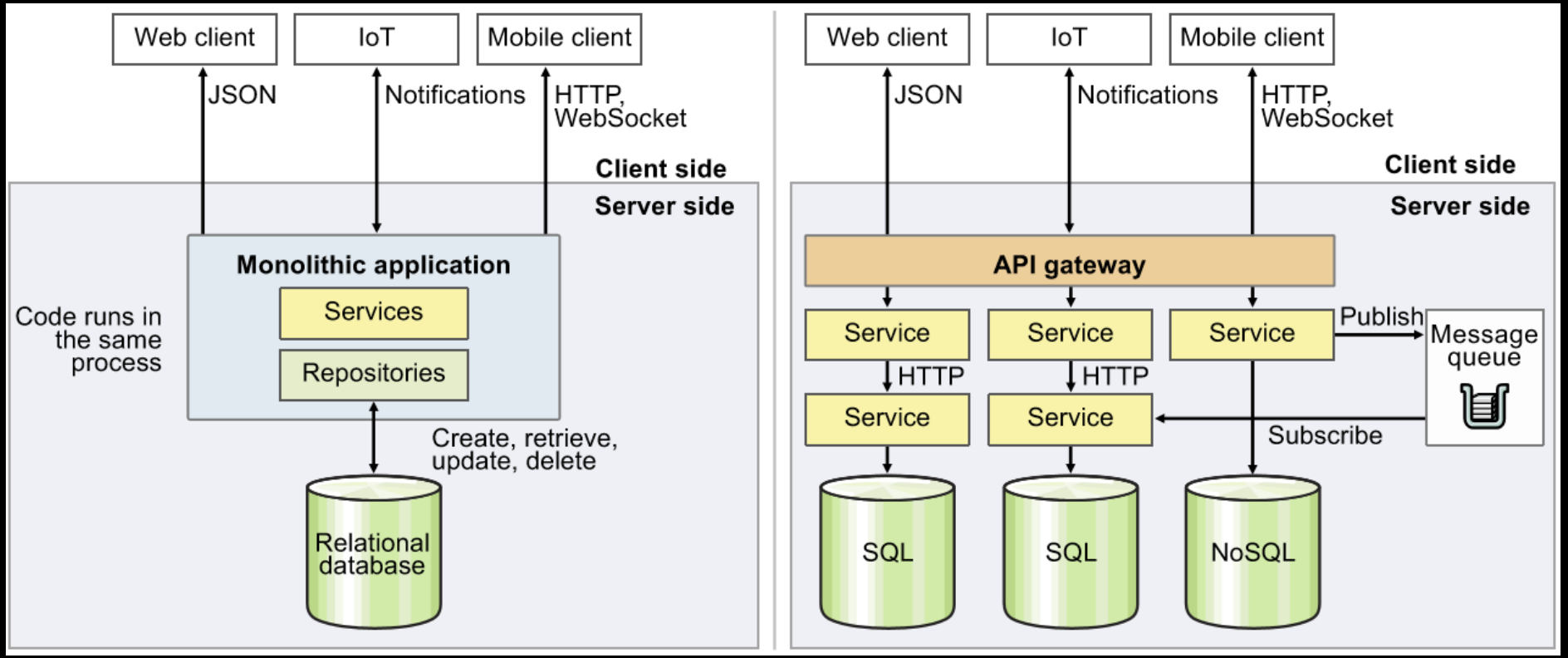
Microservices and SOA

- Both SOA and microservices deal with a system of services that communicate over a network But there are differences

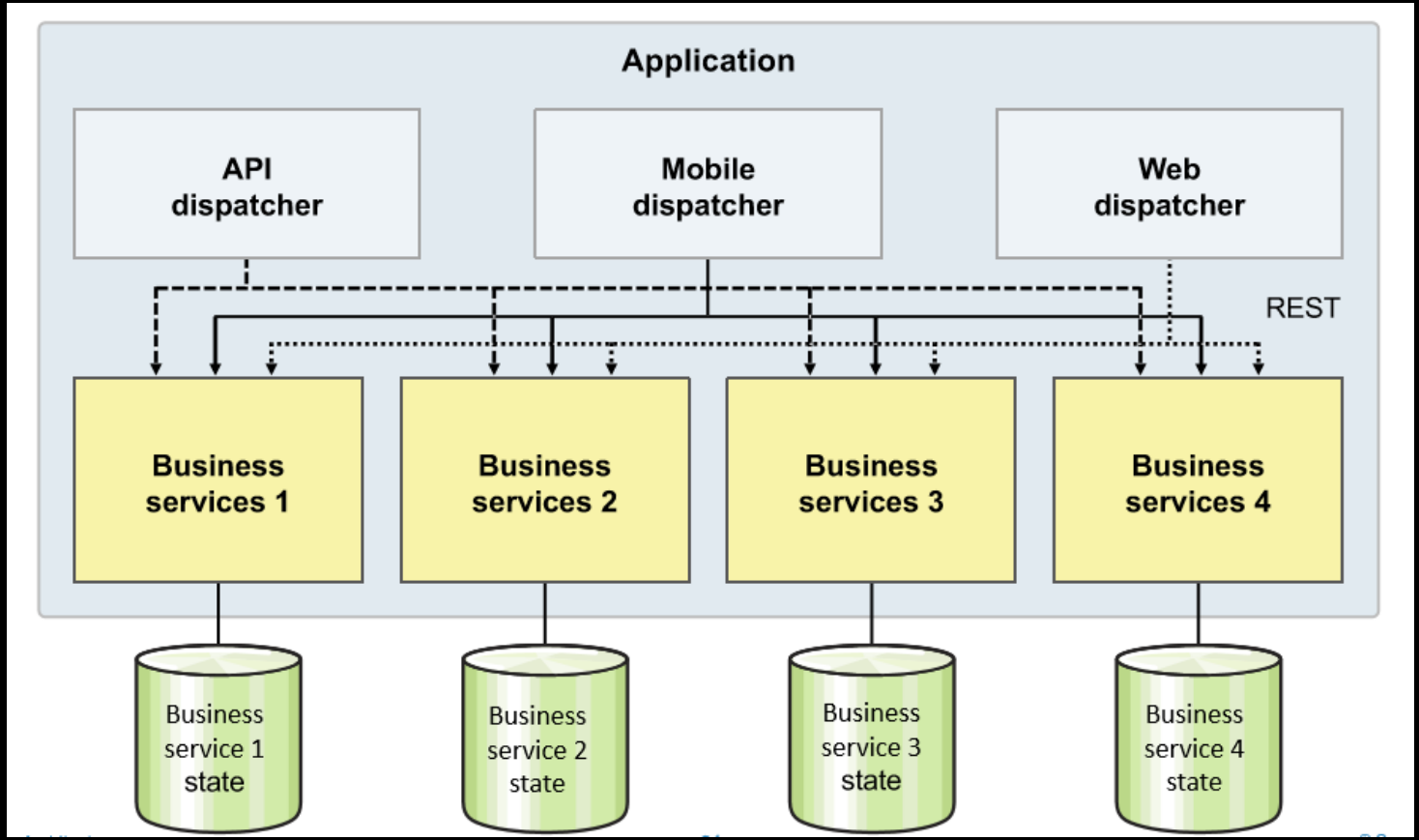
- The focus of SOA is on reuse
  - This tends to align with a centrally funded model
  - SOA services tend to be “servants of many masters”
  - This means that a change to a SOA service might impact multiple consumers

- The focus of microservices is on breaking down a potentially monolithic application into smaller, more manageable components
  - With the objective of more flexible, decoupled, faster development
  - Challenges here relate to needing good DevOps, management views, and controls

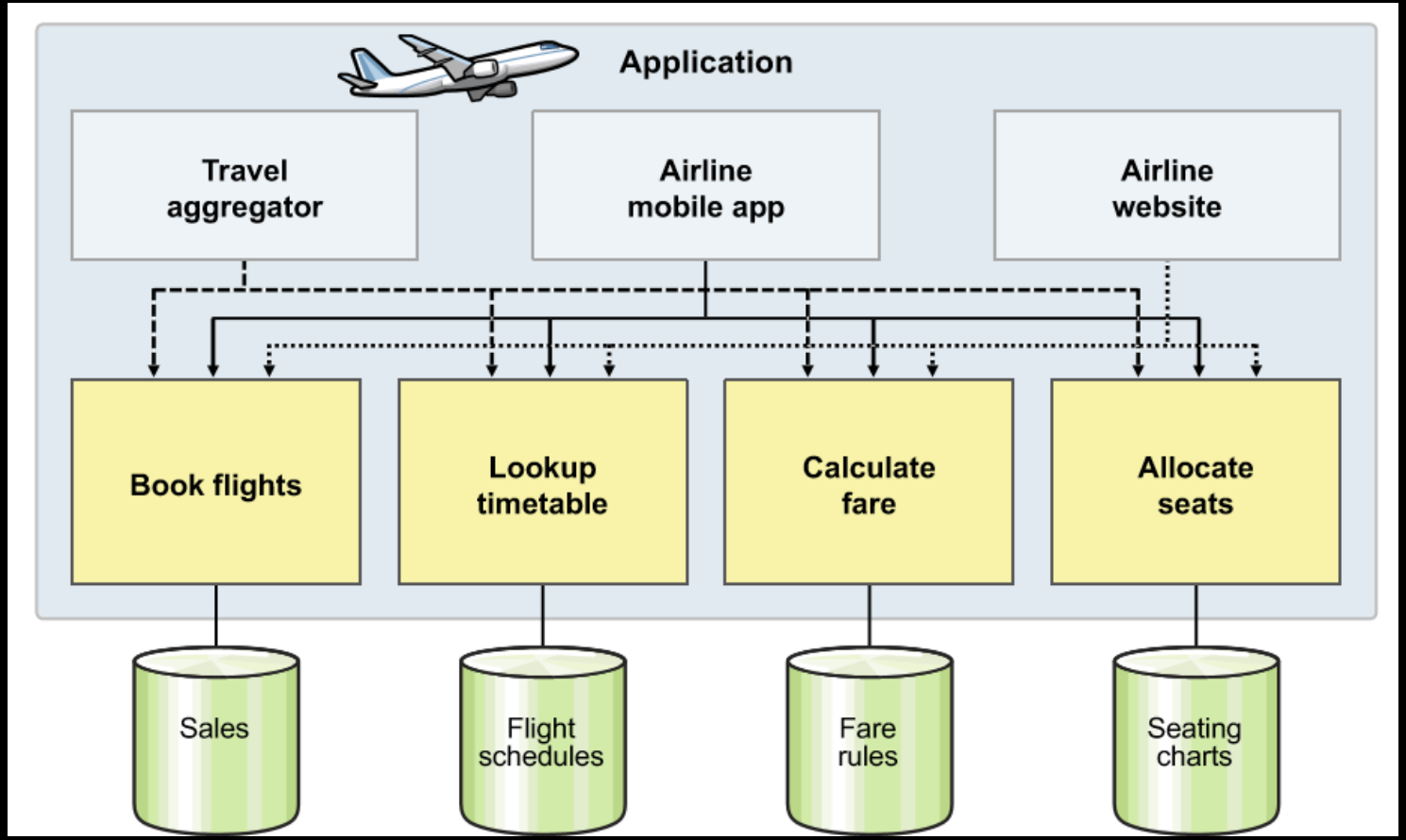
Monolithic architecture versus microservices architecture



Microservices architecture

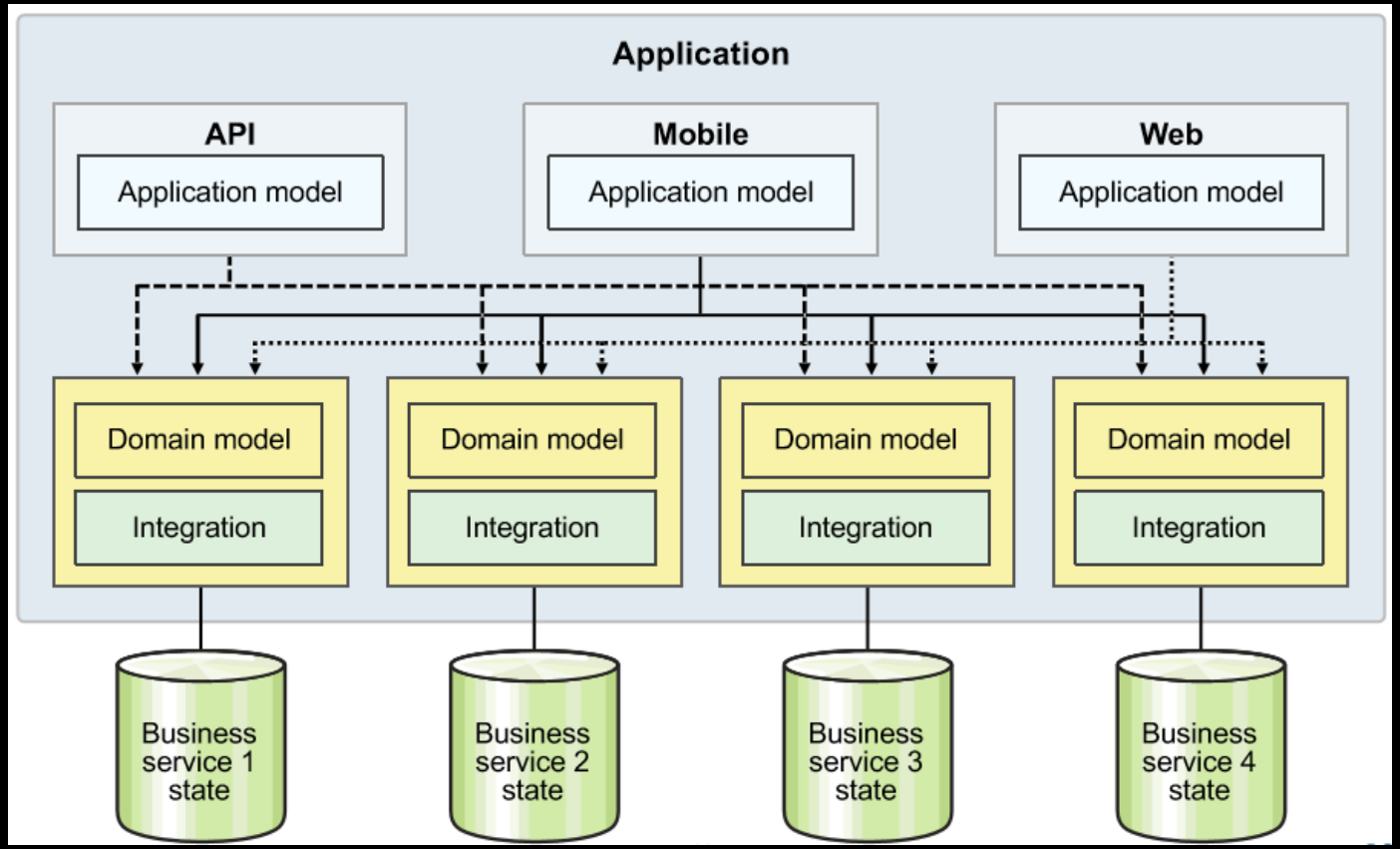


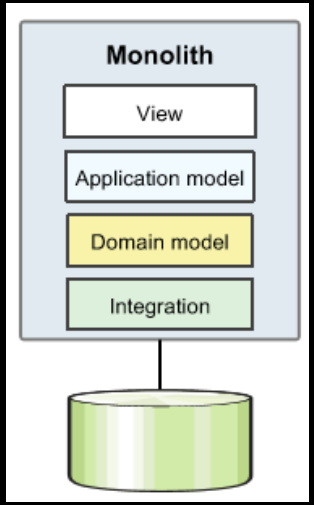
Sample airline architecture



Microservices layers

Each service is a smaller half-monolith





Language decisions

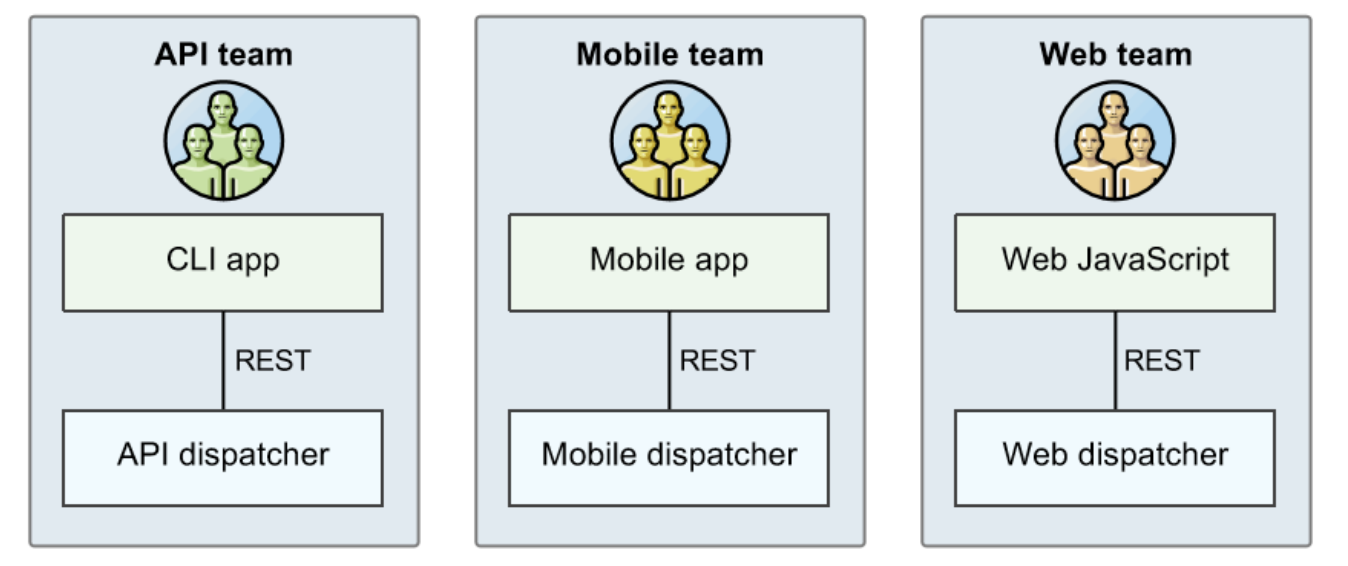
- Dispatchers are most often written in Node.js
  - Better fidelity with the clients
    - Especially clients that run JavaScript
    - iOS teams might want to use the Swift server-side runtime
  - I/O intensive, supports numerous concurrent clients

- Business services are most often written in Java
  - Better for CPU-intensive tasks
  - Better connectivity to external systems

- The team selects the language
  - Choose the language that best fits the job

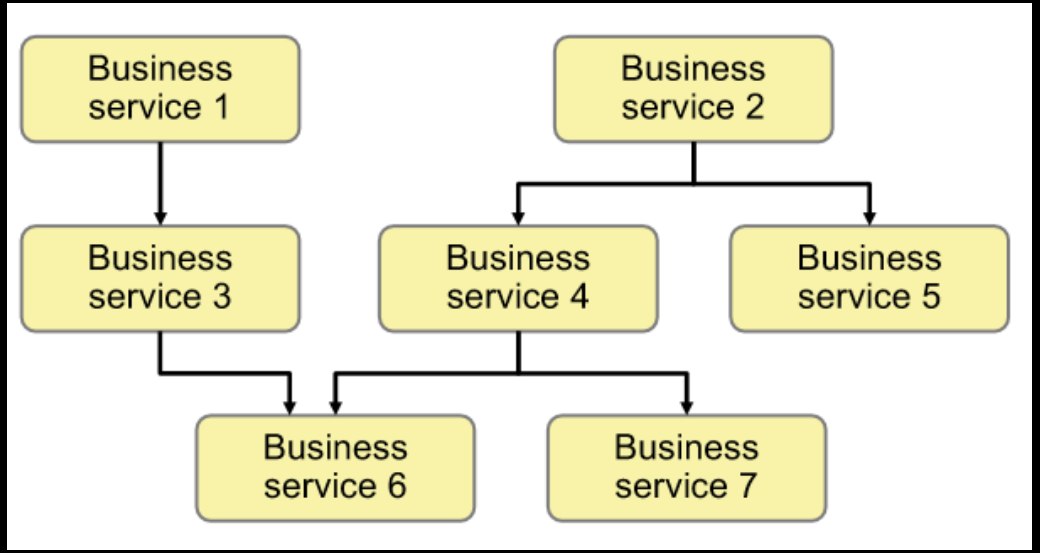
Back ends for front ends

- Each dispatcher is a back end for an external front end, typically a GUI
- The same team develops each back end and front end pair
  - Use the same or compatible languages in the pair



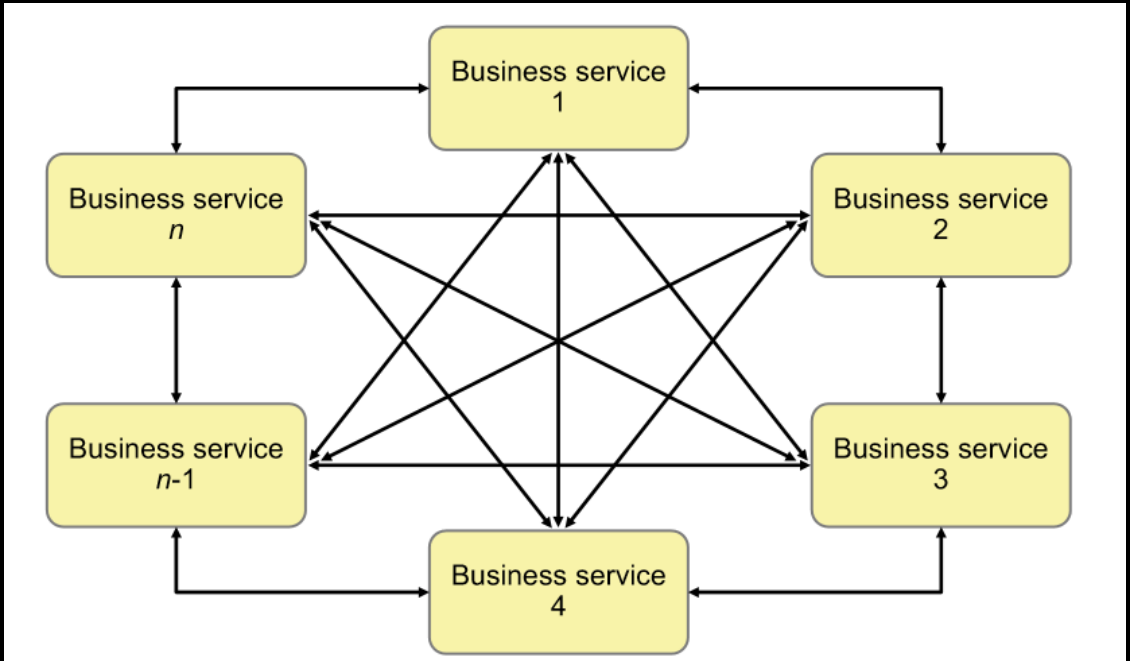
Business service microservices dependencies: Typical

- Business services can delegate to other business services
  - Avoid circular dependencies
- Be careful that each service still implements a complete task
  - They are not separate layers



Business service microservices dependencies: Death Star

- Practically every service delegates to or coordinates with every other service
- Difficult to deploy and maintain



Twelve factor applications

The twelve-factor app

- A set of best practices for creating applications
  - Implementing, deploying, monitoring, and managing
- Typical modern applications
  - Deployed in the cloud
  - Accessible as web applications that deliver software-as-a-service (SaaS)
- Can be applied to any application
  - Implemented in any programming language
  - Using any backing services such as database, messaging, and caching
- Addresses common problems
  - The dynamics of the growth of an app over time
  - The dynamics of collaboration between developers
  - Avoiding the cost of software erosion
  - Systemic problems in modern application development
- Provides a shared vocabulary for addressing these problems

The twelve factors

I. **Codebase**: One codebase that is tracked in revision control, with many deployments
II. **Dependencies**: Explicitly declare and isolate dependencies
III. **Configuration**: Store Configuration in the environment
IV. **Backing services**: Treat backing services as attached resources
V. **Build, release, run**: Strictly separate build and run stages
VI. **Processes**: Execute the app as one or more stateless processes
VII. **Port binding**: Export services with port binding
VIII. **Concurrency**: Scale out using the process model
IX. **Disposability**: Maximize robustness with fast startup and efficient shutdown
X. **Development and production parity**: Keep development, staging, and production as similar as possible
XI. **Logs**: Treat logs as event streams
XII. **Admin processes**: Run administrative and management tasks as one-off processes

Factor 1: Codebase

I. **Codebase**
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- One codebase tracked in source code management (SCM) with versioning
- Multiple deployments from the same codebase

Factor 2: Dependencies

I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Explicitly declare and isolate dependencies
- Typically language-specific
  - Node.js: Node Package Manager (NPM)
  - Liberty: Feature manager
  - Ruby: Bundler
  - Java EE: Application resources
- Never rely on system-wide dependencies

Factor 2: Dependencies

I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Explicitly declare and isolate dependencies
- Typically language-specific
  - Node.js: Node Package Manager (NPM)
  - Liberty: Feature manager
  - Ruby: Bundler
  - Java EE: Application resources
- Never rely on system-wide dependencies

Dependencies: Liberty feature manager

- Server configuration is part of deploying the application
- You manage it in source control

```
<server description="Portfolio server">
    <featureManager>
        <feature>microProfile-3.2</feature>
        <feature>jdbc-4.2</feature>
        <feature>jms-2.0</feature>
        <feature>jca-1.7</feature>
        <feature>jndi-1.0</feature>
        <feature>appSecurity-2.0</feature>
        <feature>monitor-1.0</feature>
        <feature>jpa-2.2</feature>
        <feature>localConnector-1.0</feature>
        <feature>mpReactiveMessaging-1.0</feature>
        <feature>ejbLite-3.2</feature>
    </featureManager>
```

Factor 3: Configuration

I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Store configuration in the environment
- Separate configuration from source
- Enables the same code to be deployed to different environments

Externalizing Configuration

- Liberty
  - Variables in the configuration that can be taken from properties or environment entries

```
<dataSource id="PortfolioDB" jndiName="jdbc/Portfolio/PortfolioDB">
    <jdbcDriver>
        <library description="DB2 JDBC driver jar" name="DB2">
            <file id="db2jcc4" name="db2jcc4.jar"/>
        </library>
    </jdbcDriver>
    <properties.db2.jcc databaseName="${JDBC_DB}" portNumber="${JDBC_PORT}" serverName="${JDBC_HOST}"
        user="${JDBC_ID}" password="${JDBC_PASSWORD}" />
</dataSource>
```

- Kubernetes/OpenShift
  - Uses ConfigMap and Secret resources

```
kubectl create secret generic apikey --from-literal=API_KEY=123—456
```

```
kubectl create configmap language --from-literal=LANGUAGE=English
```

Factor 4: Backing services

I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Treat backing services as attached resources:
  - Databases
  - Messaging systems
  - LDAP servers
  - Others
- Local and remote resources should be treated identically
  Possible locations for run time and resources:
  - In the same process
  - On the same host
  - On different hosts in the same data center
  - In different data centers

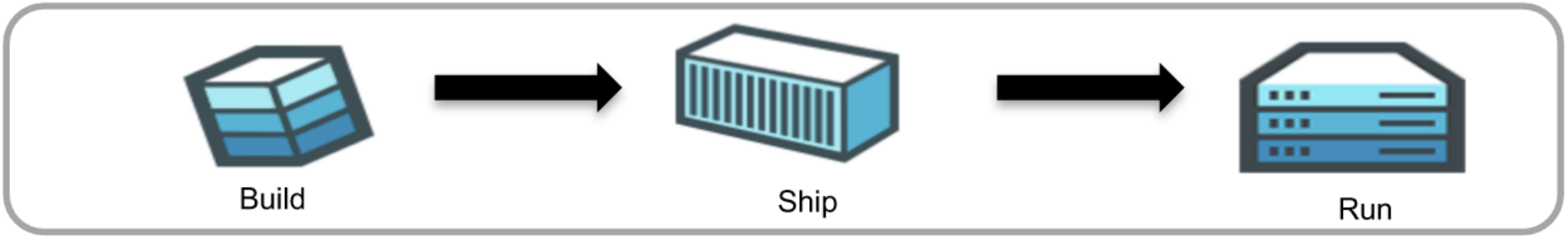
Factor 5: Build, release, run

I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Strictly separate build and run stages

Docker: Build, ship, run

- Build: Create the runtime that goes inside the container
  - Application
  - Application server
  - Operating system
- Ship: Crate a snapshot of the runtime
  - A container image
  - Stored in a registry
- Run: Create a running application
  - A container running in a container engine



Factor 6: Processes

I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Run the application as one or more stateless processes
- Do not rely on session affinity, also called sticky sessions

Factor 7: Port binding

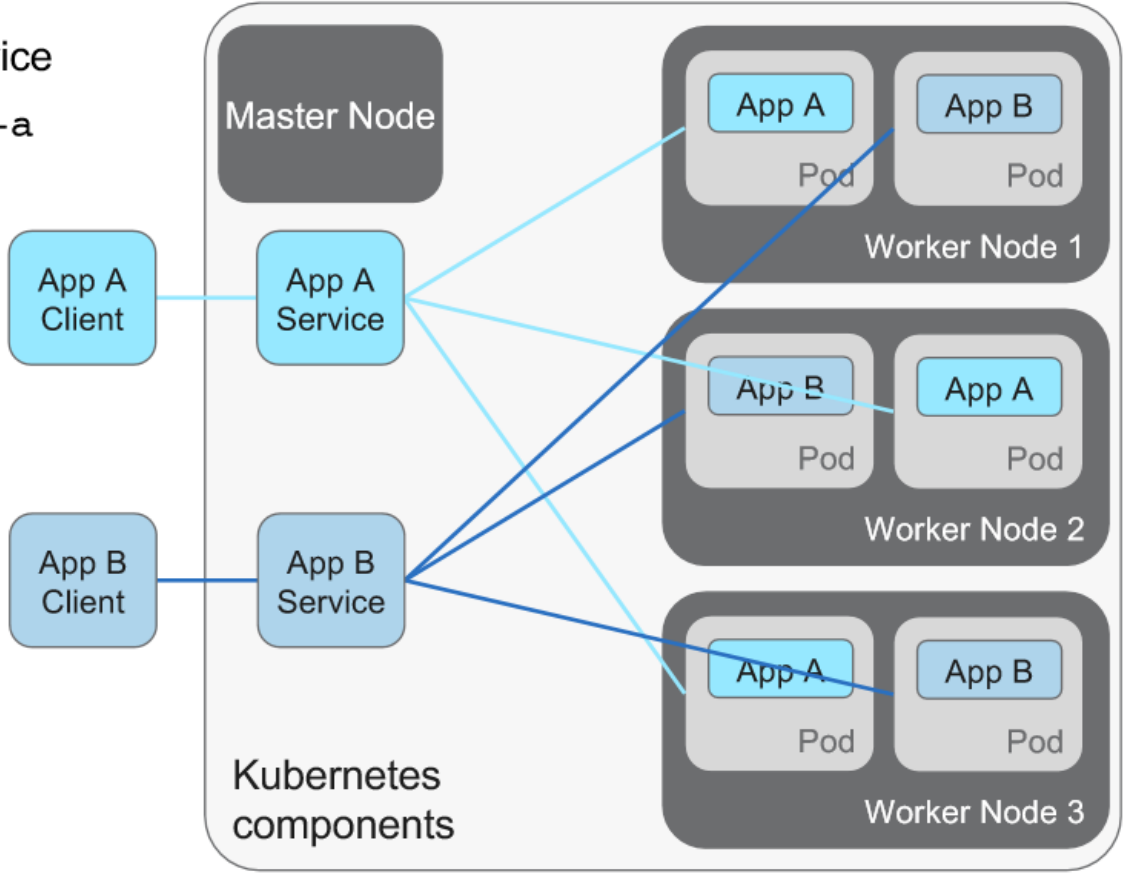
I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Export services with port binding
- Web app binds to an HTTP port and listens for requests coming in on that port

Port binding: Kubernetes service

- Expose a set of pod replicas as a service

```
kubectl expose deployment/app-a
        --type=LoadBalancer
        --port=8080
        --name=app-a-service
        --target-port=8080
```



Factor 8: Concurrency

I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Scale out using the process model
  To add capacity, run more instances
- There are limits to how far an individual process can scale
- Stateless applications make scaling simple

Factor 9: Disposability

I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Maximize robustness with fast startup and efficient shutdown
- Application instances are disposable
- Application should handle shutdown signal or hardware failure with crash-only design

Factor 10: Development and production parity

I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Keep development, staging, and production as similar as possible
- Use the same backing services in each environment

Factor 11: Logs

I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

- Treat logs as event streams
- Each process writes to stdout
  - Application should not write to specialized log files
  - Environment decides how to gather, aggregate, and persist stdout output

Factor 12: Administrative processes

I. Codebase
II. Dependencies
III. Configuration
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes

Run administrative and management tasks as single processes, such as these examples:

- Tasks for performing database migrations
- Debugging

Developing Microservices

Developing a microservices application

- Identify a set of independent business tasks
  - Build initial microservices around those tasks
  - Teams that can work independently
  - Components that can be deployed, scale, and fail independently

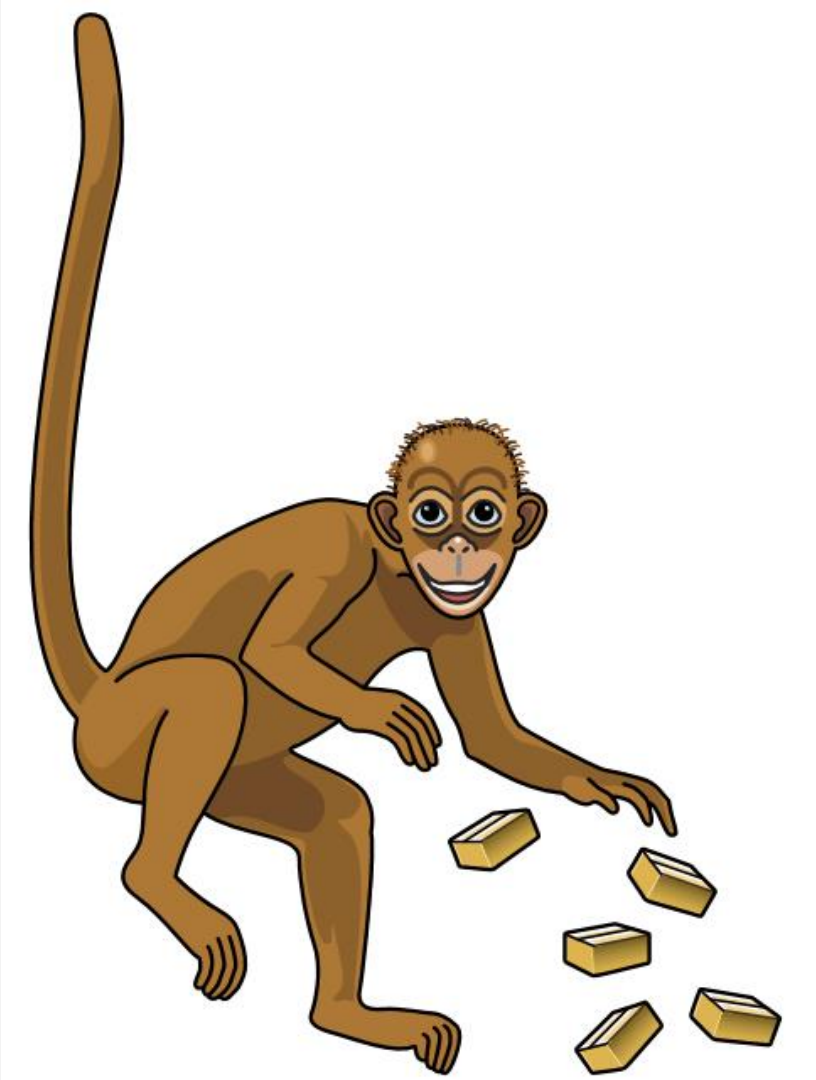
- Design for failure
  - Use microservices to make the application more robust

- Design for scale
  - Service discovery
    - For example, Eureka
  - Configuration repositories
    - For example Zookeeper
  - Common logging

- A service mesh is very helpful for steps 2 and 3

Design for failure

- Any service can fail
  - In a monolith, when one part stops working, it all stops working
  - Application must keep working and stay responsive
- Employ patterns for resiliency
  - Service Registry: Dynamic listing of service instances
  - Circuit Breaker: Block calls to a service that’s not working
  - Bulkhead: Separate connection pools for separate resources
  - Command: Make requests easy to retry, throttle, and monitor
- Test for failure
  - Purposely, randomly cause problems
  - Example: Simian Army framework



MicroProfile:
Java Microservices Programming Model

MicroProfile programming model

- Exposes REST APIs
  - Implemented using JAX-RS
- Uses Contexts and Dependency Injection (CDI)

```
@Path("props")
public class SystemProperties {

    @Produces(MediaType.APPLICATION_JSON)
    @GET
    public Map getProps() {
        return System.getProperties();
    }
}
```

```
@Path("props")
@RequestScoped
public class ServiceC {
    @Inject
    private MyBean bean;
```

MicroProfile Config feature

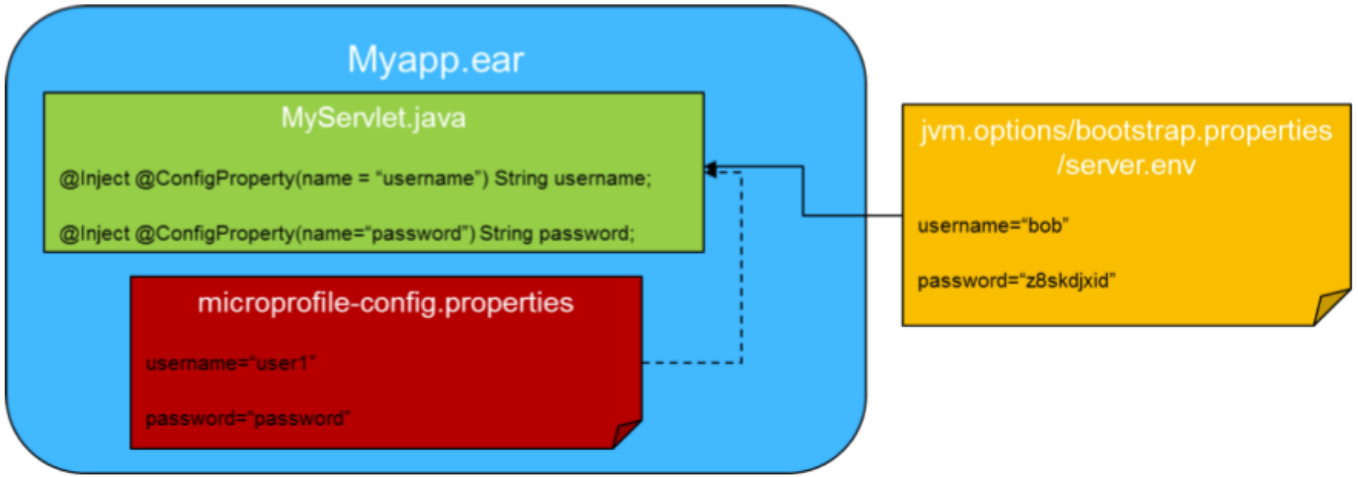
Config	Fault Tolerance	Health Check	Health Metrics	JWT Propagation
externalize configuration to improve portability	build robust behavior to cope with unexpected failures	common format to determine service availability	common REST endpoints for monitoring service health	interoperable authentication and role-based access control

mpConfig-1.0

- Write **once**
- Containerize **once**
- Liberty **injects** config from outside the container
- **Portable** across environments

The property file, *microprofile-config.properties*, packaged in the application can be overridden by

1. System variables (400 as the default priority)
2. Environment variables (300 as the default priority) or
3. A custom property files with a higher priority than *microprofile-config.properties* (100 as the default priority)



MicroProfile: Fault Tolerance feature

- Declarative policies to make microservices more resilient when runtime failures occur
  - Implemented as Java annotations
  - Support for common application resiliency patterns
    - Timeout, Retry, Circuit Breaker, Bulkhead, Fallback
  - Policies configured by properties
- Defined on the method that invokes the remote microservice
  - Handled by Contexts and Dependency Injection (CDI) interceptors

Fault Tolerance – Annotations

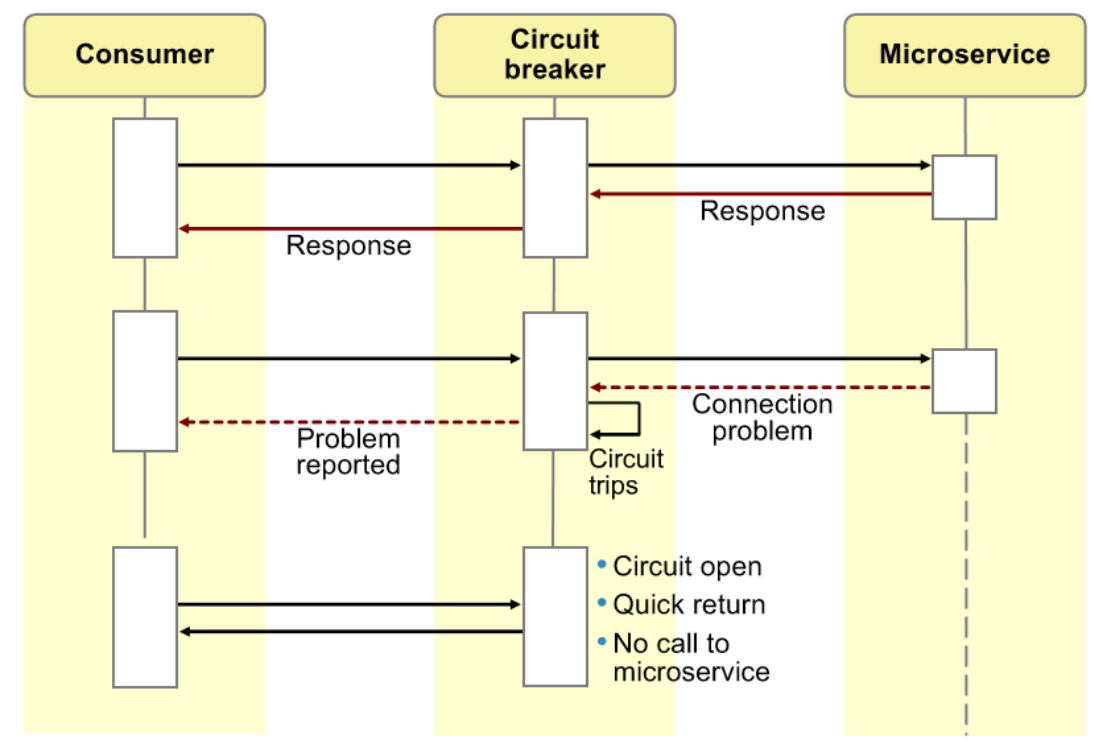
- Asynchronous: Invoke the method asynchronously
  - @Aynchronous
- Retry: Invoke a method again after failure
  - @Retry(maxRetries=3, delay=400, maxDuration=2000)
- Timeout :Specify a timeout for the method invocation
  - @Timeout(300)
- Bulkhead: Limit the concurrent requests
  - @Bulkhead(20) – semaphore model, at most 20 concurrent requests
  - @Asynchronous @Bulkhead(value=20, waitingTaskQueue=30) - thread pool style, at most 20 concurrent threads
- CircuitBreaker: Open a circuit so that the following invocation will fail immediately
  - @CircuitBreaker(delay=200, failureRatio=0.2)
- Fallback: A secondary service when the primary service invocation fails
  - @Fallback(MyFallbackHandler.class) - invoke MyFallbackHandler.handle() on exception
  - @Fallback(fallbackMethod="fbMethod") - invoke fbMethod() on the same instance as the method with this annotation

Circuit breaker

- A service consumer is only as reliable as its service provider

- Avoid invoking an unreliable service instance
  - Service might throw an error
  - Service might time out
  - Network might fail

- Circuit breaker
  - Consumer can invoke reliably
  - Invokes unreliable service
  - Fails fast when the service isn't working



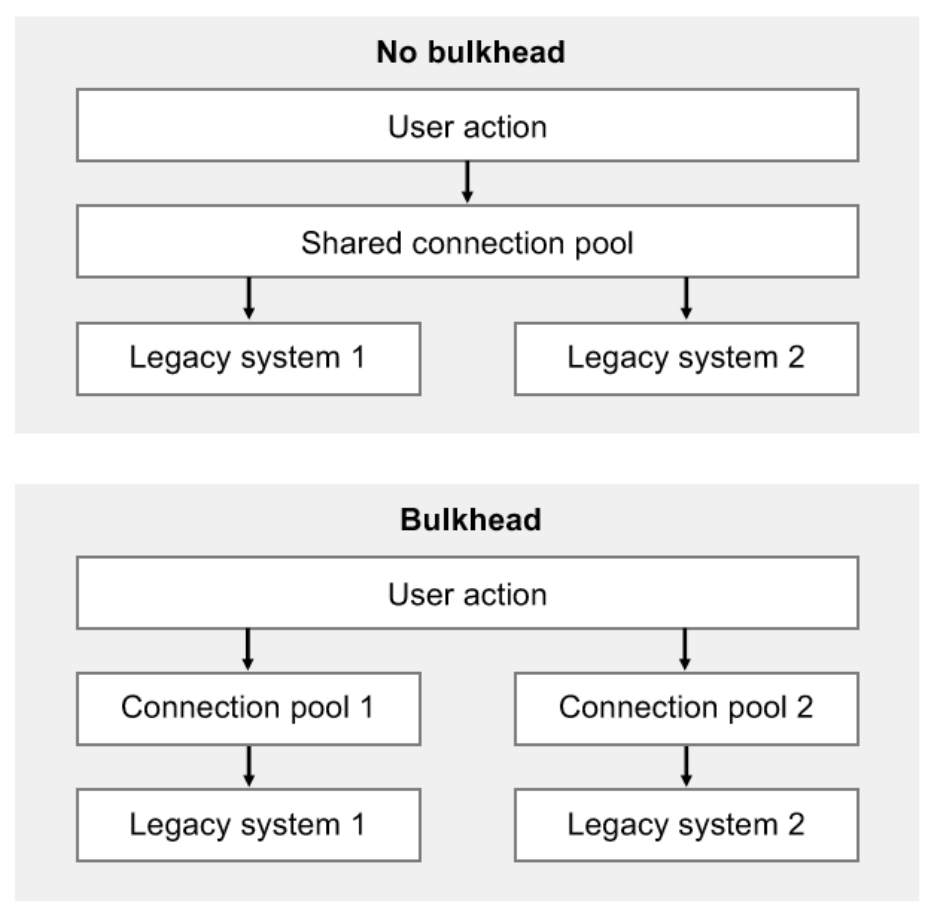
Bulkhead

No bulkhead

- Single connection pool shared to connect to multiple external systems
- If one system blocks connections
  - All connections block on the one system
  - No way to connect to the working systems

Bulkhead

- Separate connection pool for each external system
- If one system blocks connections
  - All its connections block
  - Meanwhile, connections to the working systems still work



MicroProfile: Health Check feature

- Exposes **/health** default endpoint for the server/container if feature enabled
- Offers health checks for both readiness and liveness.
  - Readiness tells whether a microservice is ready to process requests. For example, a readiness check might check dependencies, such as database connections.
  - Liveness check tells whether a microservice is running. If fails, the application can be terminated.

MicroProfile: Health Check Readiness

- The @Readiness annotation indicates that this particular bean is a readiness health check procedure. By pairing @ApplicationScoped, the bean is discovered automatically when the /health endpoint receives a request.
- Access the /health/ready endpoint to view the data from the readiness health checks

```java
@Readiness
@ApplicationScoped
public class SystemReadinessCheck implements HealthCheck {

  @Override
  public HealthCheckResponse call() {
    ...
    if (notReady) {
      return HealthCheckResponse.down(readinessCheck);
    }
    return HealthCheckResponse.up(readinessCheck);
  }
}
```

MicroProfile: Health Check Liveness

- The @Liveness annotation indicates that this particular bean is a liveness health check procedure. By pairing @ApplicationScoped, the bean is discovered automatically when the /health endpoint receives a request.
- Access the /health/live endpoint to view the data from the liveness health checks

```java
@Liveness
@ApplicationScoped
public class SystemLivenessCheck implements HealthCheck {

  @Override
  public HealthCheckResponse call() {
    MemoryMXBean memBean = ManagementFactory.getMemoryMXBean();
    long memUsed = memBean.getHeapMemoryUsage().getUsed();
    long memMax = memBean.getHeapMemoryUsage().getMax();

    return HealthCheckResponse.named(SystemResource.class.getSimpleName() + " Liveness Check")
                              .withData("memory used", memUsed)
                              .withData("memory max", memMax)
                              .state(memUsed < memMax * 0.9).build();
  }
}
```

MicroProfile: Health Metrics feature

- Exposes /metrics endpoint for the server/container if feature enabled
- Exposes system, vendor, and app-specific metrics
- Response in JSON
  - For collection from collectd or other JSON-friendly tools
  - Prometheus text formats
- App metrics provided
  - Dropwizard-based API
  - New CDI-enabled annotations

- Out-of-the-box metrics
  - JVM memory
  - Garbage Collection
  - JVM uptime
  - Threads
  - Thread Pools (stretch goal)
  - ClassLoading
  - CPU usage and availability

MicroProfile: Health Metrics annotations

- @Counted – counts innovations of annotated object
- @Gauge – samples the value of annotated object
- @ConcurrentGauge – counts parallel invocations
- @Metered – tracks frequency of invocations
- @Metric – contains the metadata information
- @Timed – aggregates timing durations and provides duration statistics, plus throughput statistics
- @SimplyTimed - only tracks elapsed time duration and count

MicroProfile: Metrics example

```
@Timed(name = "inventoryProcessingTime",
       tags = {"method=list"},
       absolute = true,
       description = "Time needed to process the inventory")
@Counted(name = "inventoryAccessCount",
         absolute = true,
         description = "Number of times the list of systems method is requested")
public InventoryList list() {
  return new InventoryList(systems);
}

@Gauge(unit = MetricUnits.NONE,
       name = "inventorySizeGauge",
       absolute = true,
       description = "Number of systems in the inventory")
public int getTotal() {
  return systems.size();
}
```

```
# TYPE application_inventoryProcessingTime_seconds summary
# HELP application_inventoryProcessingTime_seconds Time needed to process the inventory
application_inventoryProcessingTime_seconds_count{method="list"} 2
application_inventoryProcessingTime_seconds{method="list",quantile="0.5"} 2.2185000000000002E-5
...
```

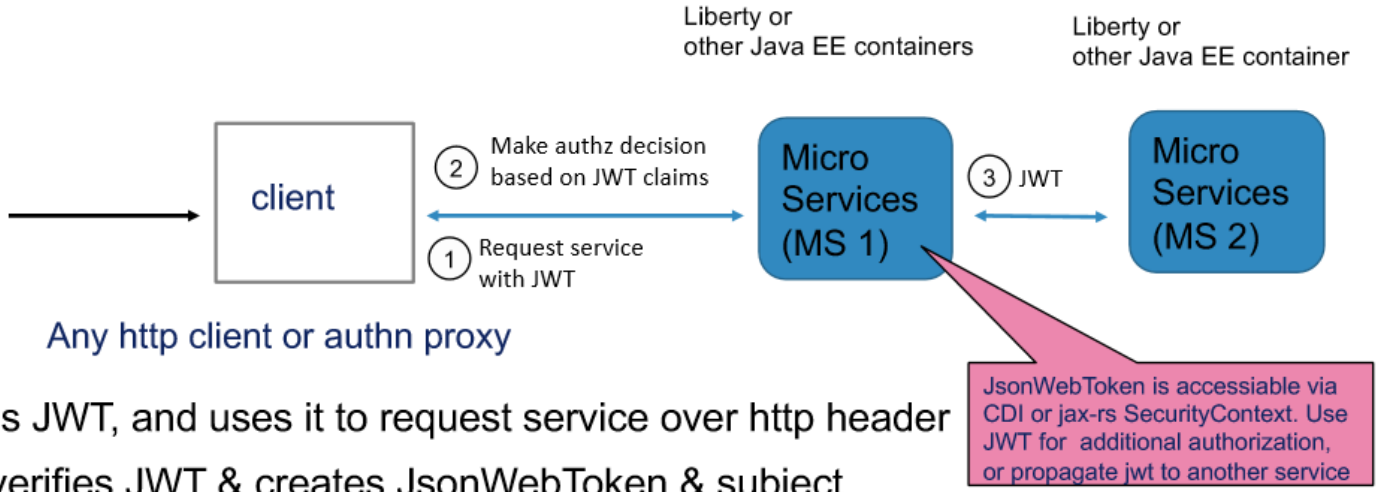
```
# TYPE application_inventoryAccessCount_total counter
# HELP application_inventoryAccessCount_total Number of times the list of systems method is requested
application_inventoryAccessCount_total 1
```

```
# TYPE application_inventorySizeGauge gauge
# HELP application_inventorySizeGauge Number of systems in the inventory
application_inventorySizeGauge 1
```

MicroProfile: JWT Propagation features

Config	Fault Tolerance	Health Check	Health Metrics	JWT Propagation
externalize configuration to improve portability	build robust behavior to cope with unexpected failures	common format to determine service availability	common REST endpoints for monitoring service health	interoperable authentication and role-based access control

mpJwt-1.0



1. Client has JWT, and uses it to request service over http header
2. Service verifies JWT & creates JsonWebToken & subject
3. Service authorizes request with JsonWebToken

Communication between services

- Communication between services should be language-neutral
  - Services can be written in different languages

- Most often REST synchronous protocol
  - JSON, HTTP
  - Network/firewall friendly

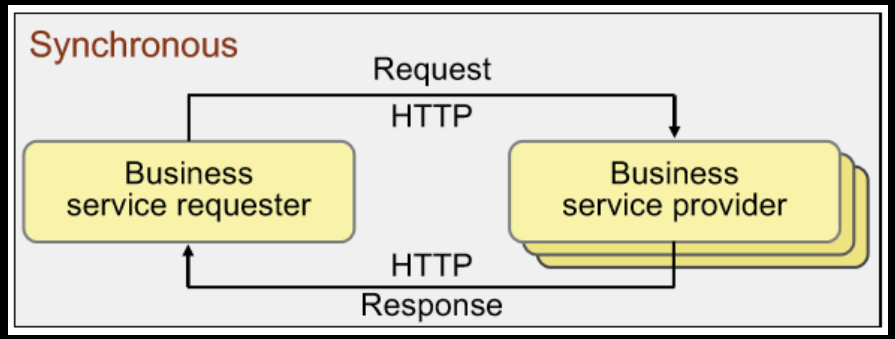
- For asynchronous protocol – Event driven apps
  - Messaging system, typically MQ or Kafka
  - JSON message payloads
  - P2P or Pub/Sub

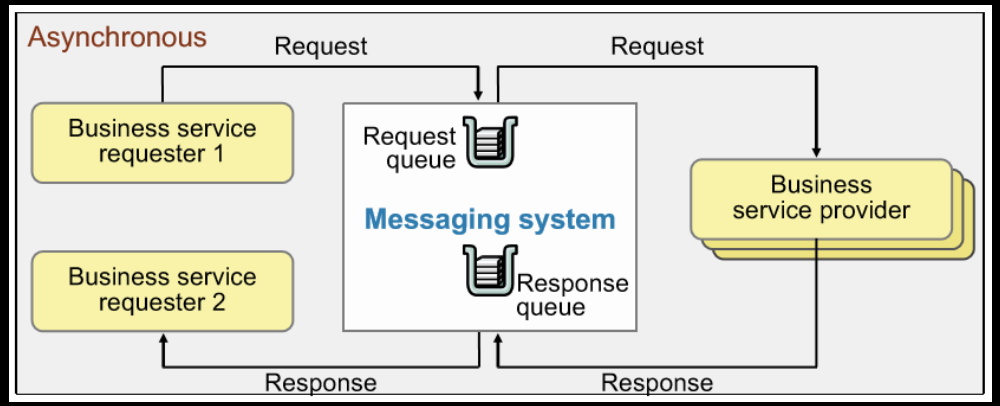
Asynchronous communication

Asynchronous communication can make microservices more robust

- Better decoupling
- Requester doesn’t have to block while provider runs
- Different requester instance can handle response
- Messaging system holds action and result

Be aware of additional latency and unpredicted response time





Configuration for microservices

Make configuration part of your DevOps process

- Treat infrastructure as code, software-defined data center
- All application deployment must be automated
- Start small with simple scripts and build from there
- All configuration properties externalized as environment variables

Refactoring to Microservices

Evolving to microservices

- Agile development: What's the simplest thing that could possibly work?
  - A monolith is simpler

- Start with a monolith
  - Make it modular, and plan that modules can become microservices
  - Each module should be a vertical slice, a mini-app with its own data

- Start with a minimally viable product (MVP)
  - MVP is a module – improvements to existing features go in the same module
  - Implement new functional tasks in new modules
  - When a module implements more than one task, refactor into separate modules

- Separate modules into microservices as needed
  - Multiple teams want to work independently
  - Monolithic code base is becoming unwieldy to maintain
  - Modules need to deploy, scale, or fail independently

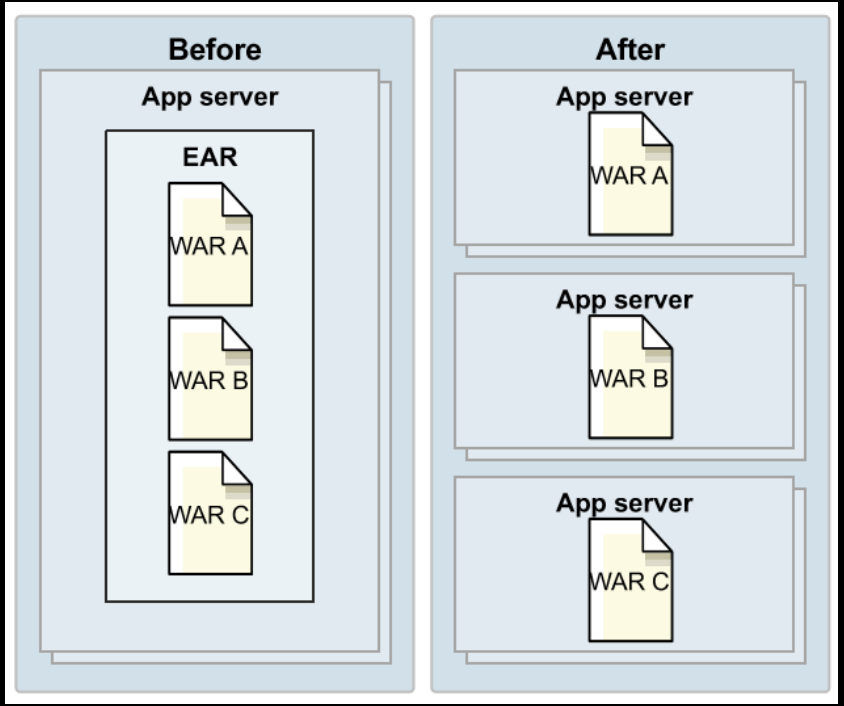
Refactoring an existing application

- Places to refactor
  - Each REST service is potentially a microservice
  - Each SOAP web service or EJB is potentially a microservice
    - Especially stateless session beans
    - Redesign function-oriented interfaces to asset-oriented interfaces
    - Make the interface RESTful, such as using command objects
  - Use domain-driven design to discover your assets, which might be microservices

- Refactor the data for the microservices code
  - Each microservice manages its own data
  - A set of tables that is only used by one module
  - A set of tables that is only used at a particular time, like a daily status summary
  - Refactor the tables so that each table is used by only one module
    - Optimize for query time, not storage efficiency
- Consider runtime modernization
  - Check if monolith can be run as whole on modern runtime e.g. WebSphere Liberty

Refactoring an existing application

- Split up EAR files
  - Divide along functional lines
  - Package independent WAR files
  - Might require minor code changes
- Deploy WAR files separately
  - Its own Liberty for Java instant runtime
  - Its own Liberty Docker container
- Build, deploy, and manage separately
  - Use independent DevOps pipelines for each WAR file
  - Scale each separately and manage independently



Questions/Discussions?