Northeastern University - Seattle

CS6650 Building Scalable Distributed Systems

Professor Ian Gorton

Building Scalable Distributed Systems

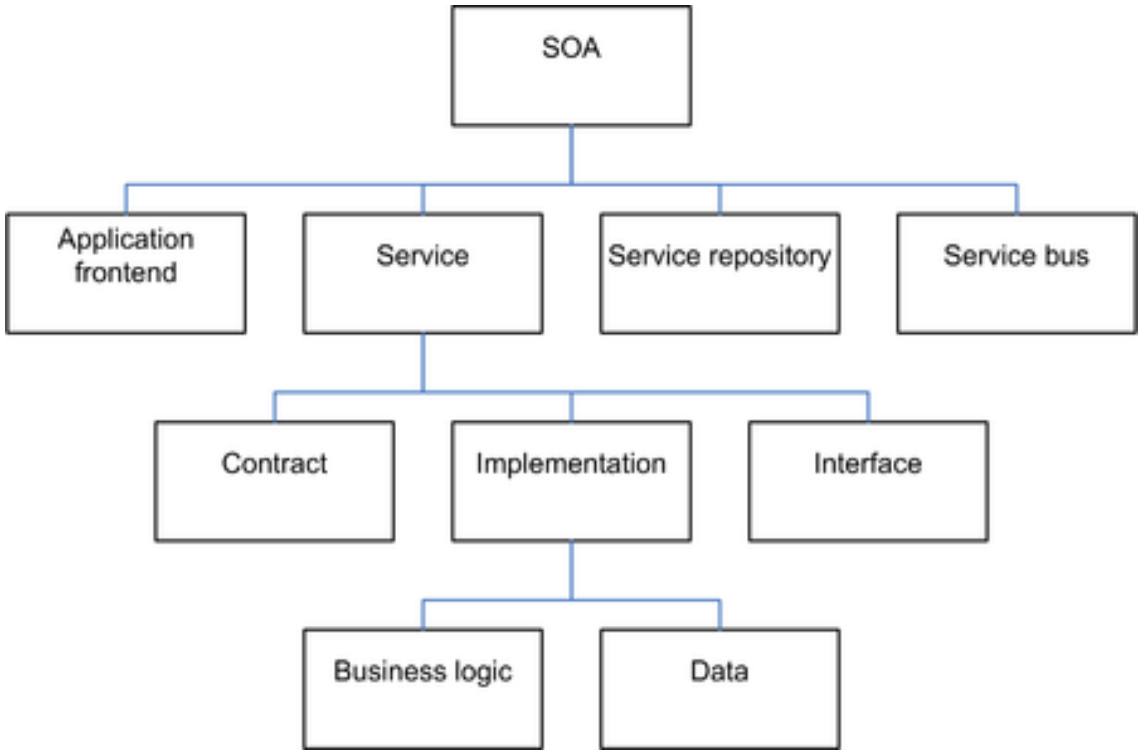
Week 9 – Microservices

Outline

- The movement to microservices
- Characteristics of a microservices based system
- Resilience
- Circuit Breaker Pattern
- Bulkhead Pattern

The Movement to Microservices

Once upon a time …

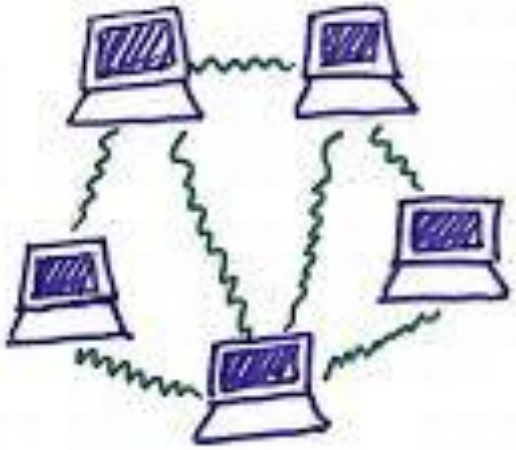


WTF is SOA?
(let's ask Wikipedia)

- **Service-oriented architecture** (**SOA**) is a style of software design where services are provided to the other components by application components, through a communication protocol over a network.
- A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently, such as retrieving a credit card statement online.
- A service has four properties according to ***one of many definitions*** of SOA:
  - It logically represents a business activity with a specified outcome.
  - It is self-contained.
  - It is a black box for its consumers.
  - It may consist of other underlying services


OK, WTF is
SOA?

Sounds suspiciously like …



A distributed system involves multiple entities talking to one another in some way, while also performing their own operations.

A personal perspective

- The term SOA emerged around the time of Web Services technologies
  - XML over HTTP stuff
- Opened the door for the first real generation of distributed systems in businesses
  - As it was relatively easy and multi language/platform
- Vendors needed a way to differentiate their offering and do marketing. Hence
  - SOA
  - ESB

Open Group Definition

- *Service-Oriented Architecture (SOA)* is an *architectural style* that supports *service-orientation*.
- *Service-orientation* is a way of thinking in terms of services and service-based development and the outcomes of services.
- A *service*:
  - Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports)
  - Is self-contained
  - *May be* composed of other services
  - Is a “black box” to consumers of the service

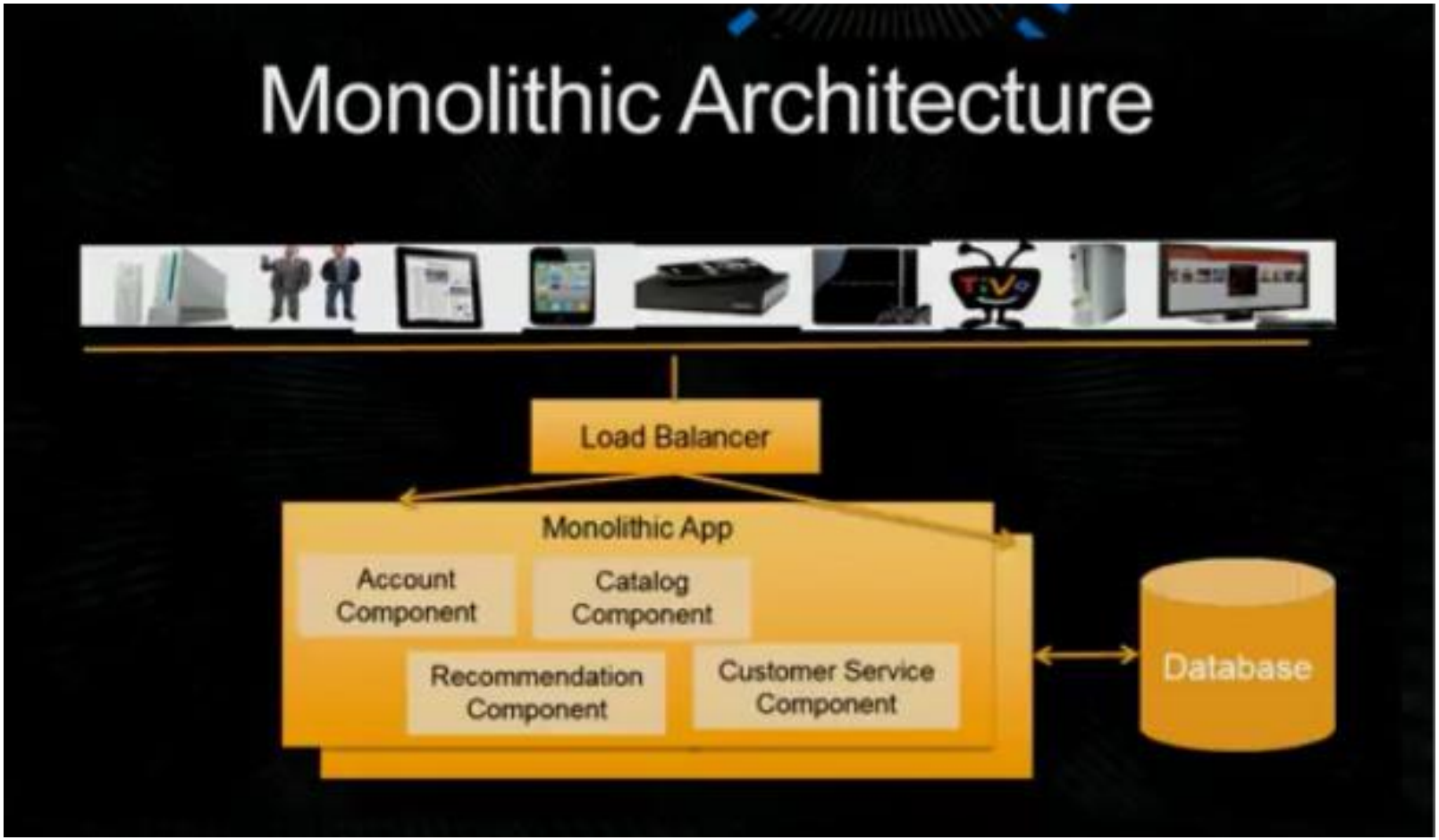
So now you know
#interviewfodder

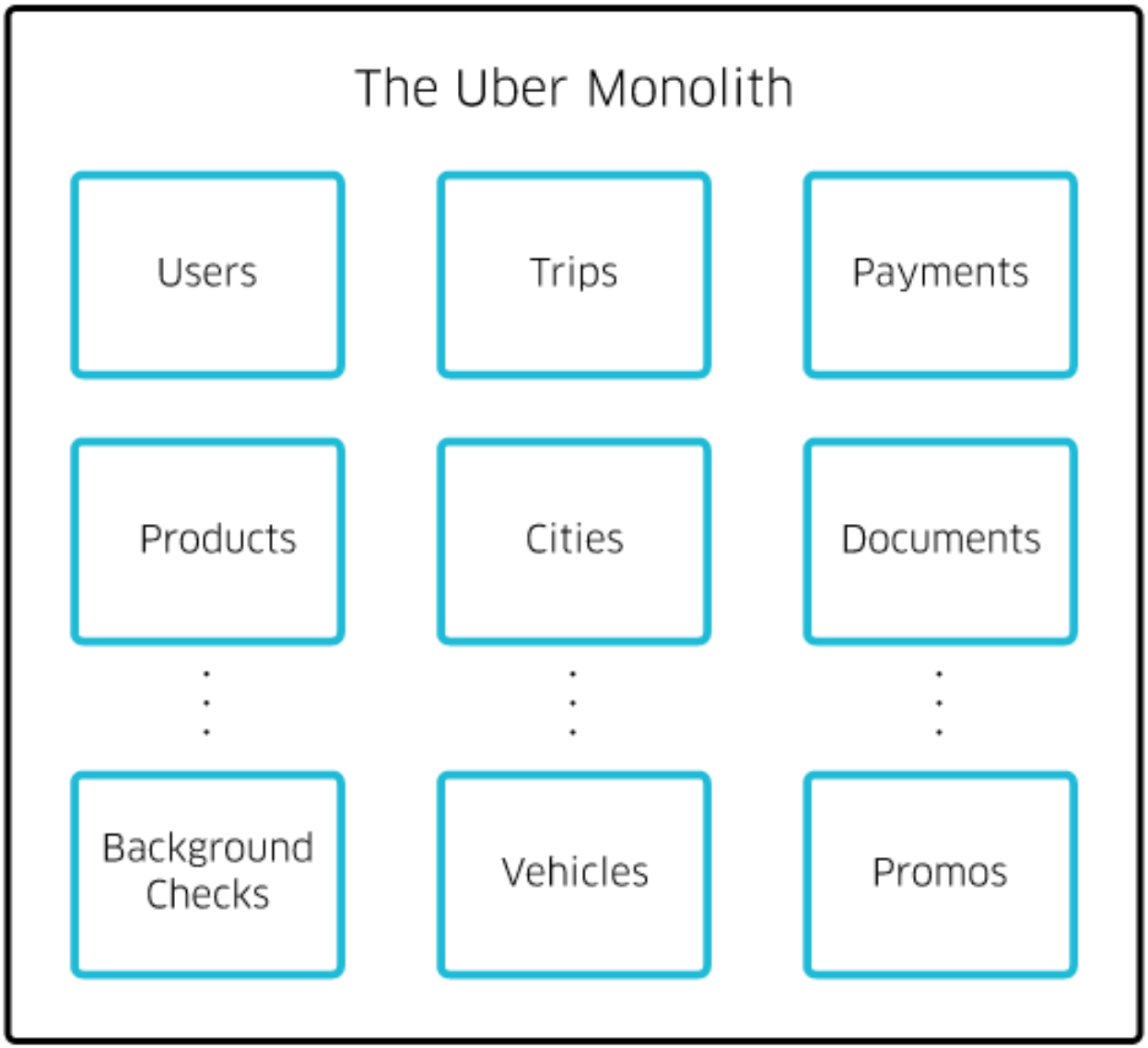
WTF has this got to do with Microservices?

Microservices are services only smaller

So what are Microservices?

Let’s go back to SOA (whatever that is!!)




The Uber Monolith
Users
Trips
Payments
Products
Cities
Documents
Background Checks
Vehicles
Promos

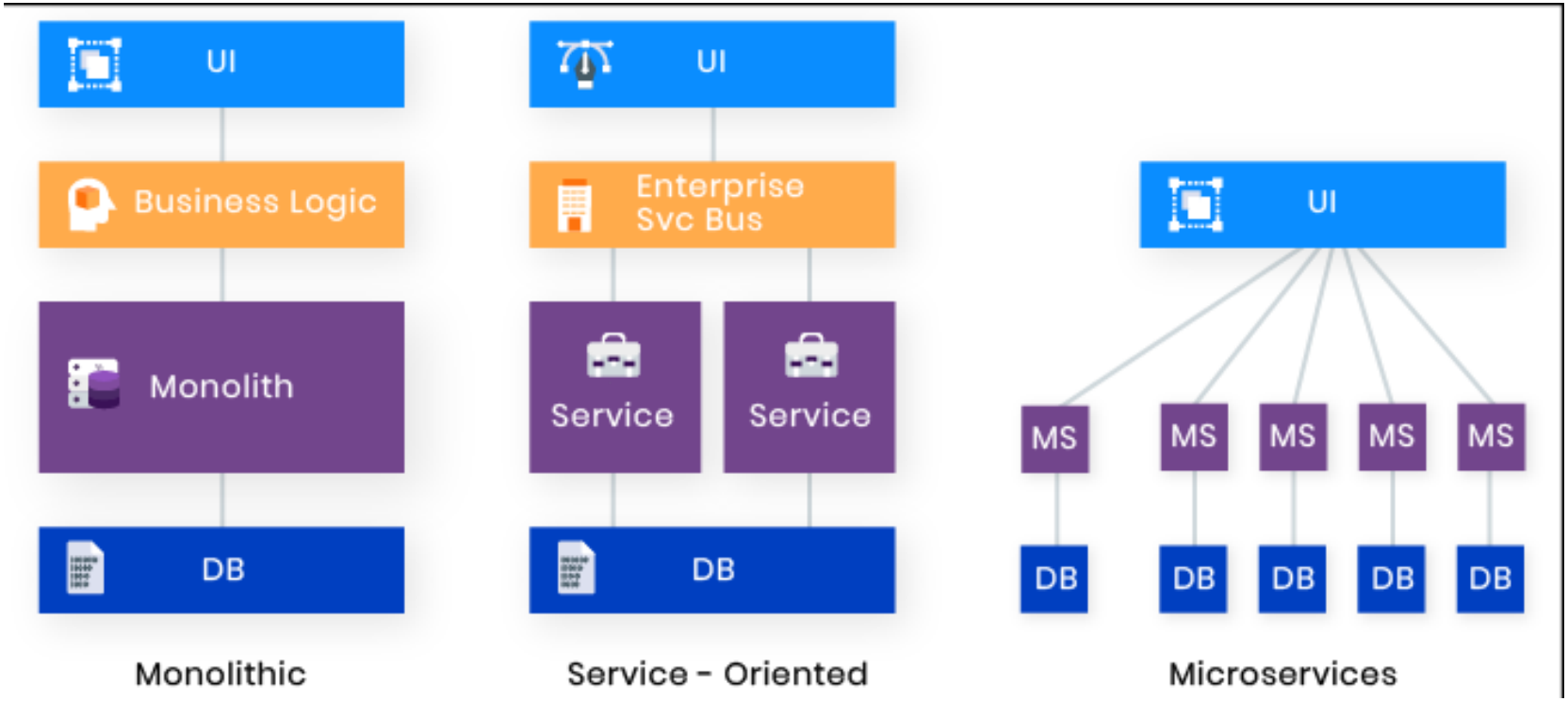
Monolithic SOA

- All services are bundled together in a server
  - One deployable unit
- If we want to scale out, we have to replicate the whole monolith
  - Kinda heavyweight, uses more resources
- Monoliths also:
  - Get more complex to evolve as they grow
  - Testing/deployment is complex and slow

More on scaling

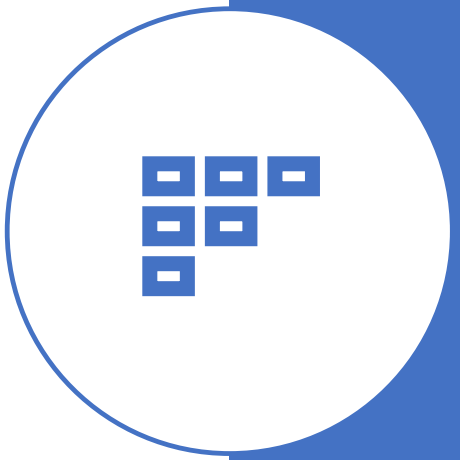
- In our monolith:
  - Some services may over time attract more demand
  - Need to scale out
- Scaling out involves deploying the whole monolith
- Even though many services hardly used
  - Uses more resources
  - Costs more
  - Probably less efficient

Moving to Microservices

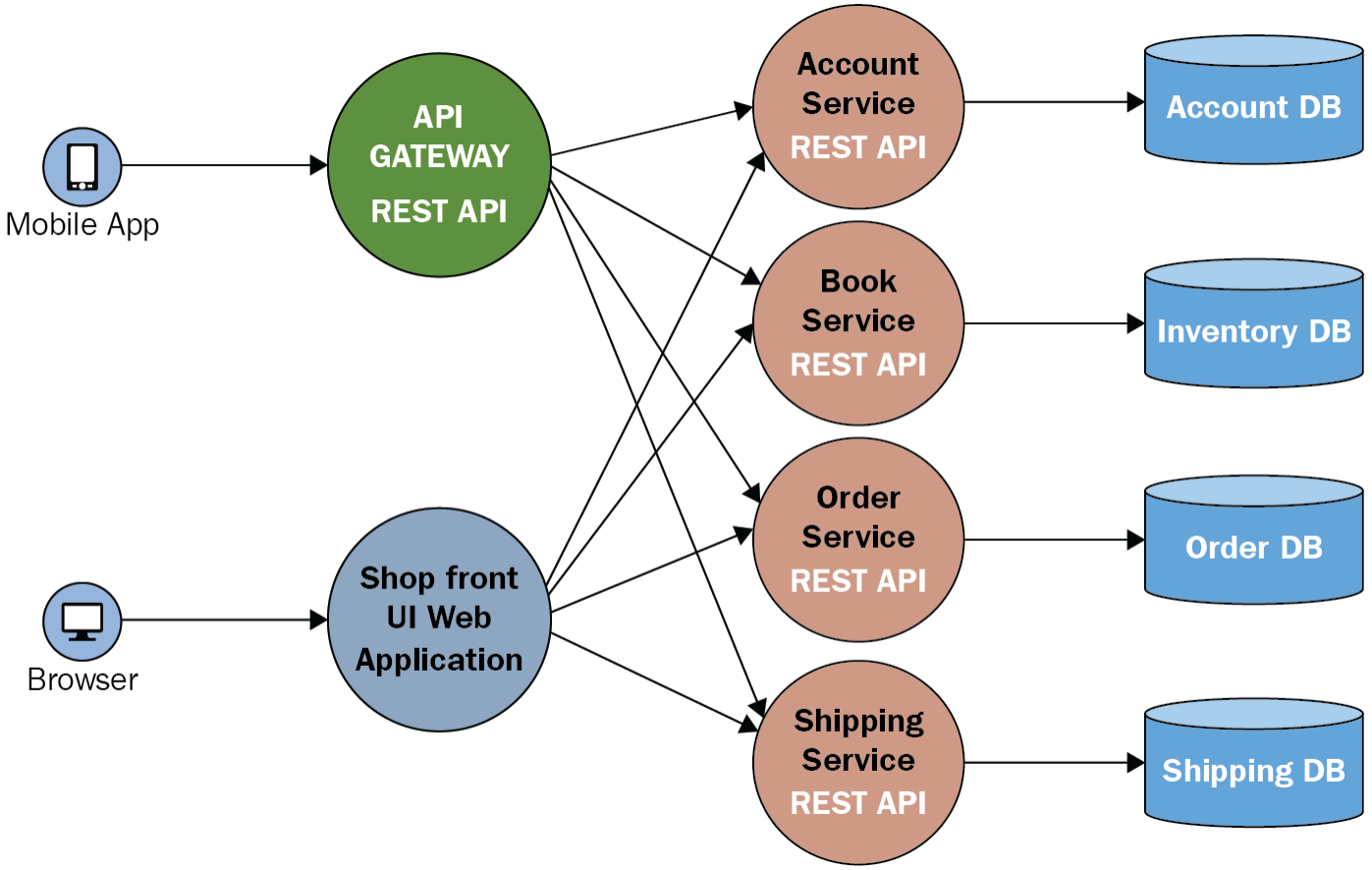


Advantages

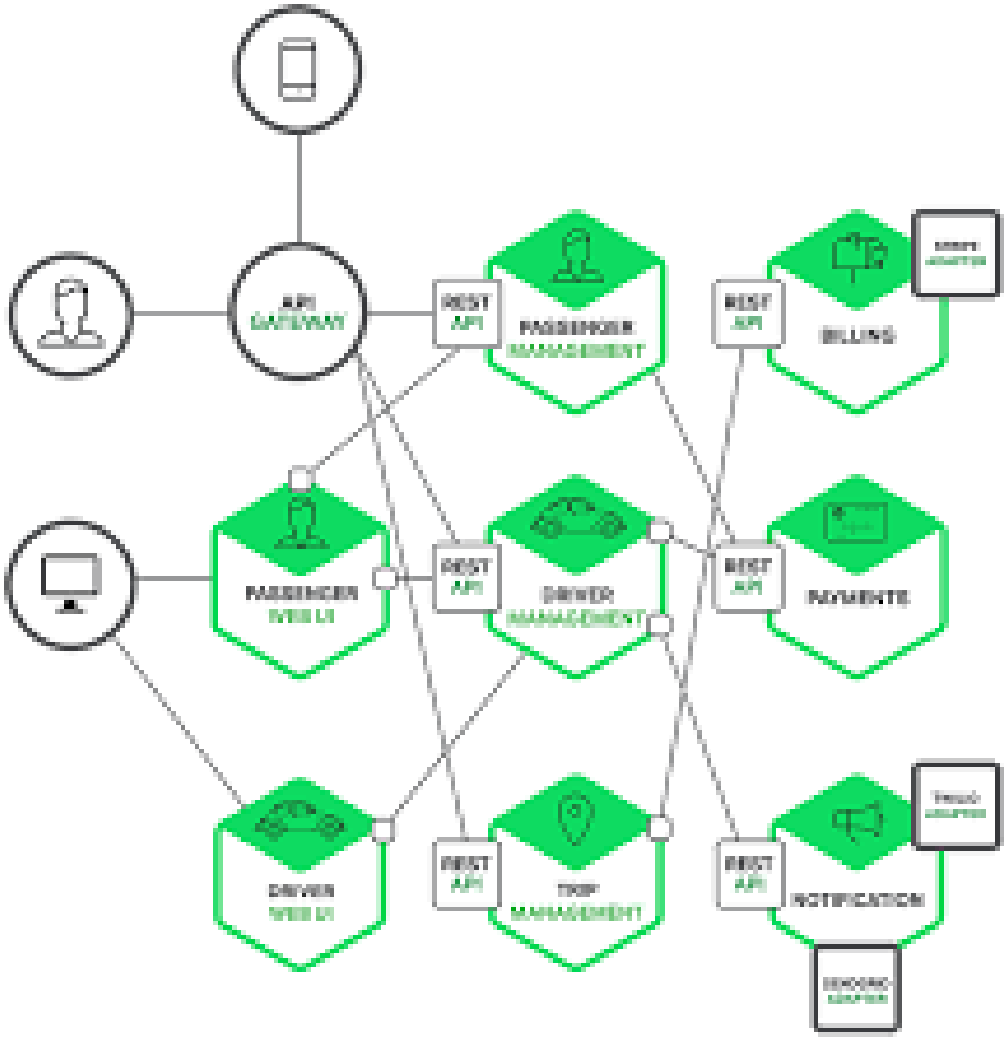
- A better name? – fine grained services
- Single responsibility principle
  - Do one (business/application oriented) task well
  - Have clear boundary (API)
  - Developed supported by a small team
    - 2 pizza rule?
  - Easier to change independently
  - Independently deployable
  - Independently scalable

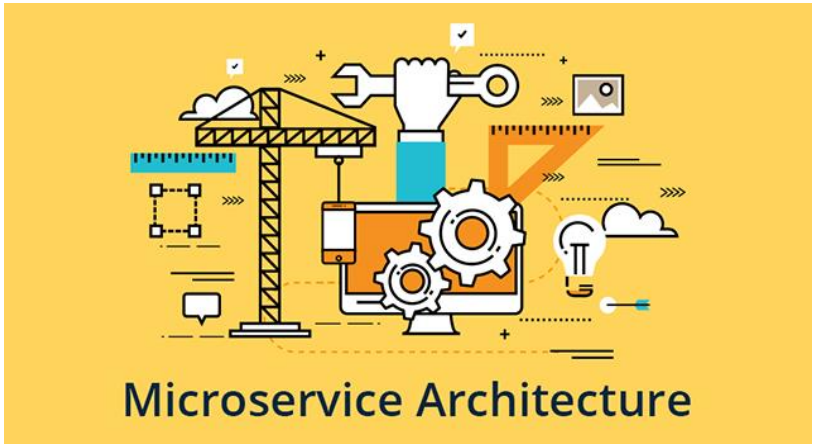


Example


Mobile App
Browser
API GATEWAY REST API
Shop front UI Web Application
Account Service REST API
Book Service REST API
Order Service REST API
Shipping Service REST API
Account DB
Inventory DB
Order DB
Shipping DB

Another Example



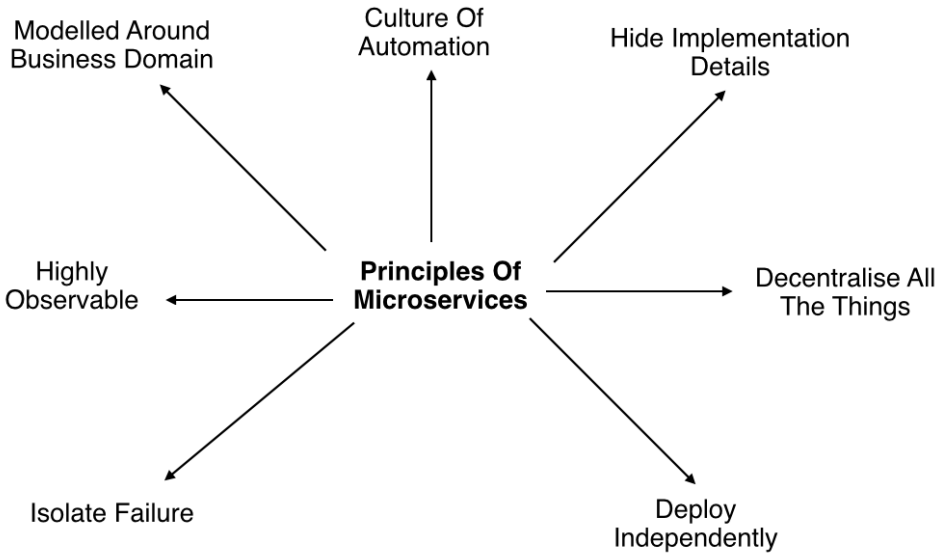


So Let's Summarize

- Applications comprise multiple communicating microservices
- How do we decompose them?
  - Rules?
  - Methods?
- Services are
  - Loosely coupled
  - Cohesive
- Services typically:
  - Single responsibility
  - Small team (`10)
  - Independently deployable
- Communications:
  - Reliable
  - Low latency

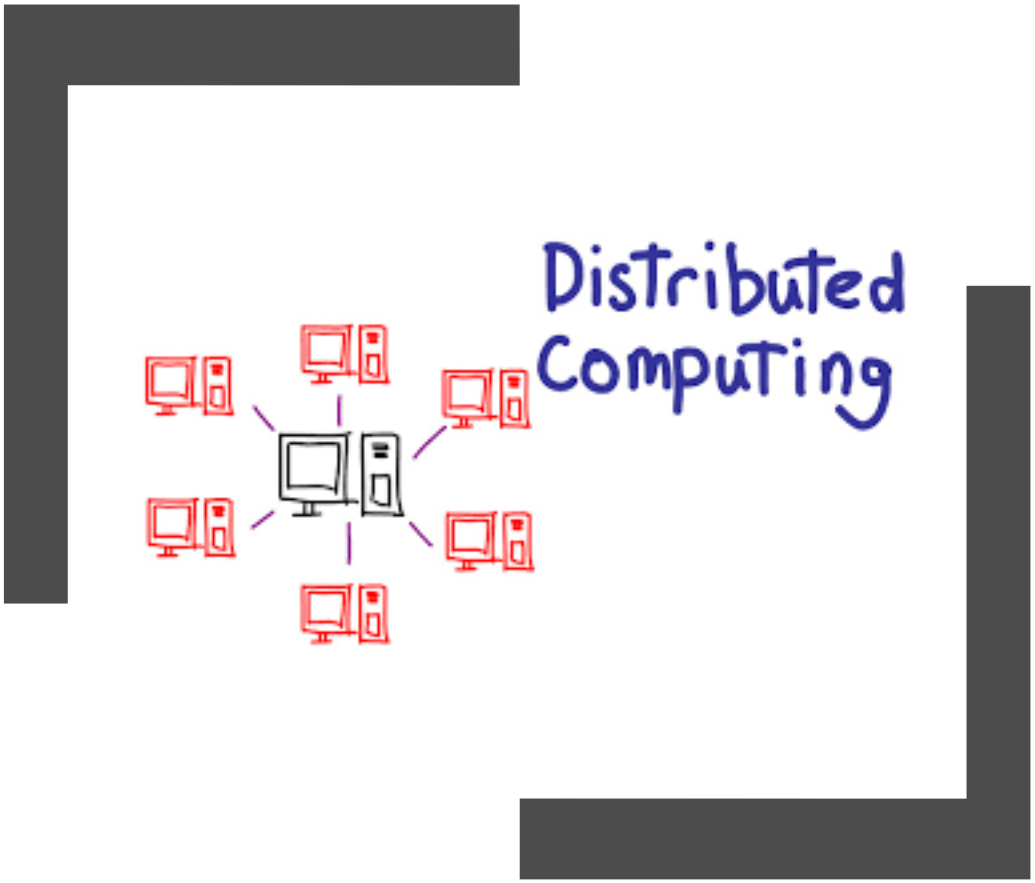
Worthy Reading

Microservices



Sam Newman Talk

Scaling Microservices

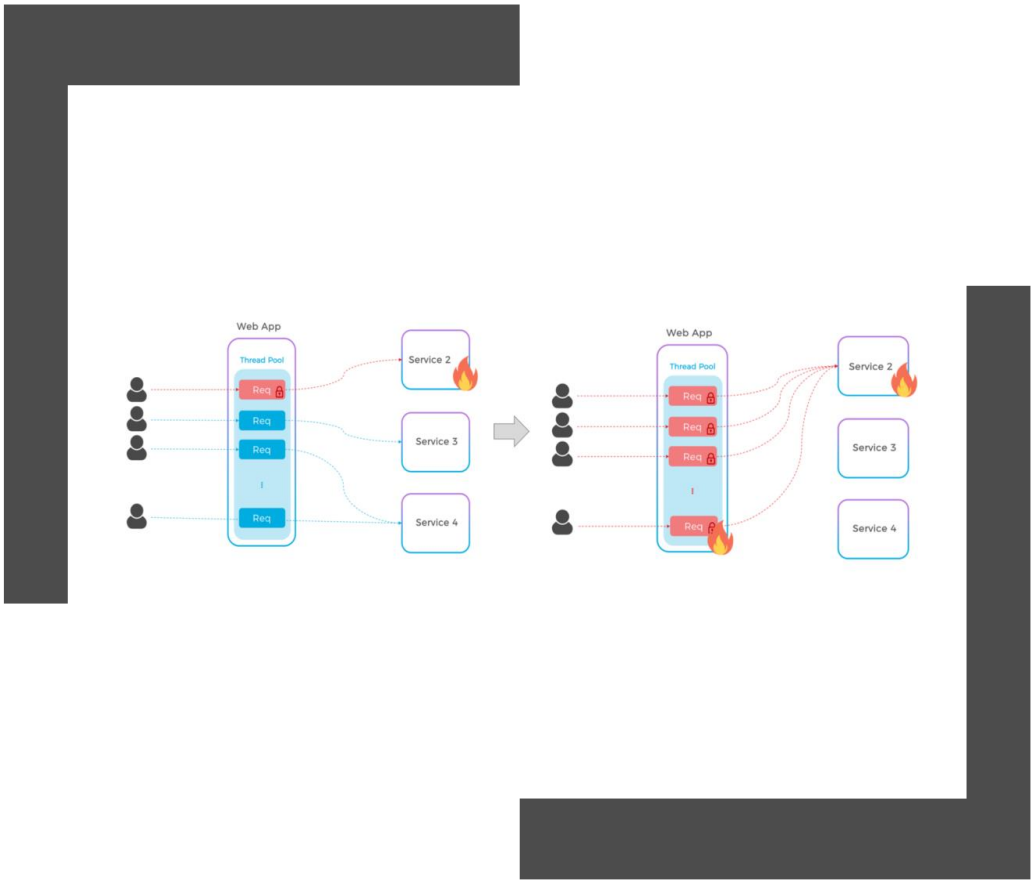


- At its core a microservices-based system is:
  - A distributed system
  - (A)synchronous communications
  - Distributed data repositories
- Sound familiar?
- Scaling is hard …

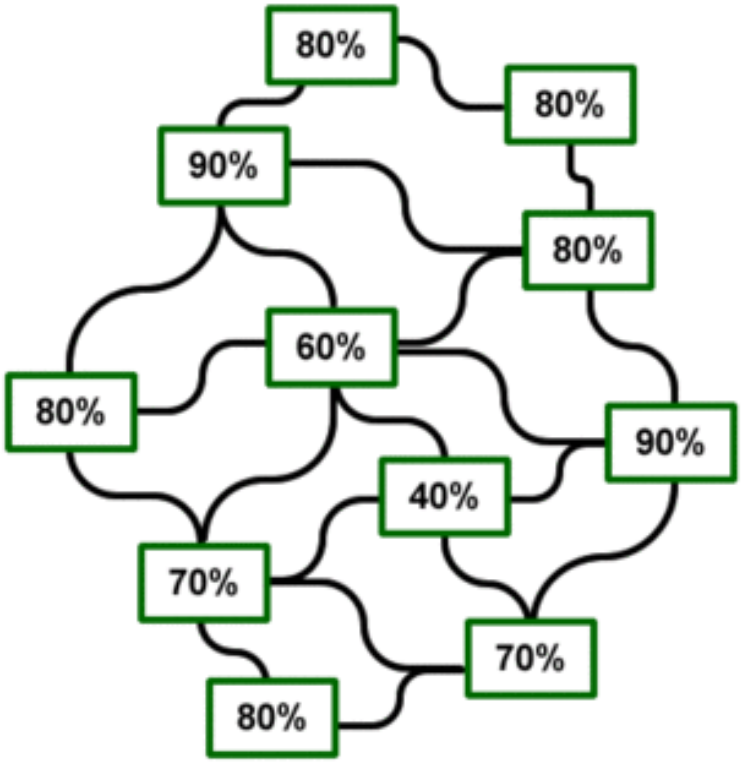
Tolerating Failures

- What should a system do if just one microservice is down?
- Reject request?
- Degraded functionality?
- Users logs into music service
- Personalization microservice that hold user preferences is unavailable?
- What do we do?

What if a service is slow?



- A downstream service suddenly slows down
- New requests keep arriving
- We send these requests to the upstream service
- It keeps waiting for the slow services



Network running normally

Cascading Failures

- A slow service causes requests to build up ‘upstream’
- If we keep sending requests to the slow service …
  - Will it get faster?
- Requests on upstream service consume threads
  - Block as downstream service slow
  - Runs out of threads …
- It soon becomes unresponsive …
- Cascading failure ….

Cascading Failures

- Under heavy load, cascading failures can happen very quickly
- We need to be tolerant to slow services
  - Why are slow services more evil than failed ones?
- Detect slow services
- Limit their scope for causing cascading failures

Timeouts – Fail Fast!!

- If we wait ‘forever’ on a slow downstream service:
  - We consume resources (thread/connection)
  - Can cause service to become unresponsive
- Hence:
- Timeouts on all out of process calls
  - No blocking calls
  - Choose sensible defaults
- If timeout occurs:
  - Log it
  - Tune timeout settings based on system behaviors
  - Design upstream ‘default’ functions to deal with failed request

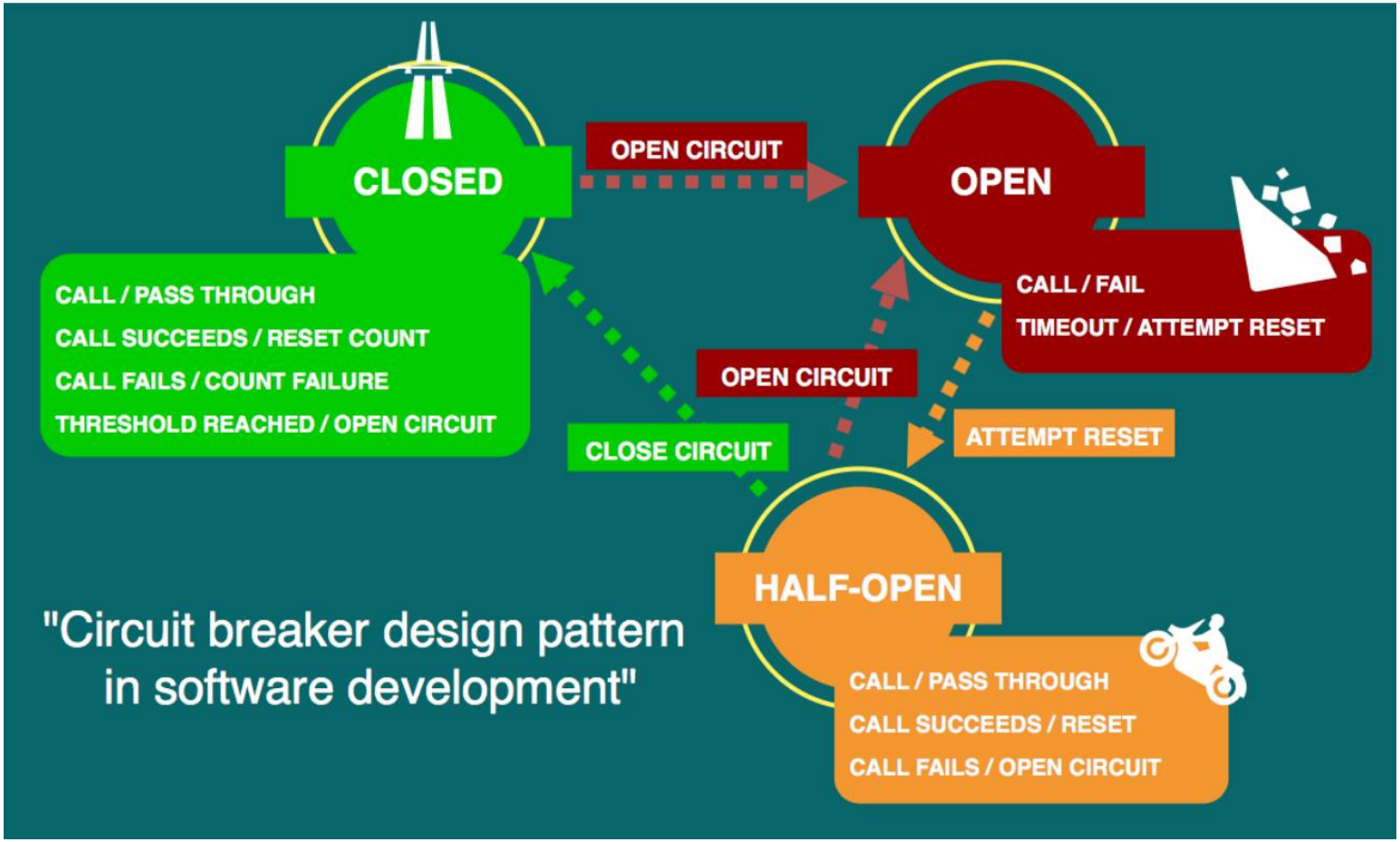
Circuit Breakers

- Analogous to electrical system
- If a power surge, the breaker blows to protect appliances
- In software if there's
  - a traffic surge
  - A slow/dead downstream service
- The breaker blows and stops sending requests

Circuit Breakers

- For HTTP requests, indicators are:
  - 5XX errors
  - Timeouts
- Circuit breaker monitors % of failed calls
- When threshold reached, it blows
- New requests are:
  - Queued if appropriate
  - Fail fast
- Circuit breaker waits a period and starts allowing requests through
  - How long?
  - If still fail?
  - If succeed?

Circuit Breaker



https://medium.com/@iamgique/circuit-breaker-design-pattern-in-software-development-1c79a4fa6838

Circuit breaker

- Multiple implementations available
  - Failsafe
  - Javaslang
  - Hystrix
  - Vert.x
  - Apache Circuit Breaker in org.apache.commons.lang3.concurrent
- Good comparison here

Apache EventCounterCircuitBreaker

```
EventCountCircuitBreaker breaker = new EventCountCircuitBreaker(1000, 1,
TimeUnit.MINUTE, 800);
 ...
 public void handleRequest(Request request) {
     if (breaker.incrementAndCheckState()) {
         // actually handle this request
     } else {
         // do something else, e.g. send an error code
     }
 }
```

Unreliable Service

```
EventCountCircuitBreaker breaker = new EventCountCircuitBreaker(5, 2,
TimeUnit.MINUTE, 5, 10, TimeUnit.MINUTE);
...
public void handleRequest(Request request) {
    if (breaker.checkState()) {
        try {
            service.doSomething();
        } catch (ServiceException ex) {
            breaker.incrementAndCheckState();
        }
    } else {
        // return an error code, use an alternative service, etc.
    }
}
```

Vert.x Example

- Configure the circuit breaker
  - **Max-failures = 2:** After *2 failures*, the circuit trips to the *open-state*.
  - **Timeout = 2000:** If the operation has not finished after *2 seconds*, it counts as a *failure*.
  - **Fallback-on-failure = true:** We're calling the fallback on failure.
  - **Reset-timeout = 2000:** We're waiting for *2 seconds* in the *open-state* before an *retry*.

Configure Circuit Breaker

```
Vertx vertx = Vertx.vertx();
CircuitBreaker breaker = CircuitBreaker
        .create("unstableAppBreaker", vertx,
                new CircuitBreakerOptions(). setMaxFailures(2)
                                .setTimeout(2000)
                                .setFallbackOnFailure(true)
                                .setResetTimeout(2000))
        .openHandler(h -> System.err.println("circuit-breaker opened"))
        .closeHandler(h -> System.out.println("circuit-breaker closed"))
        .halfOpenHandler(h -> System.err.println("circuit-breaker half-opened"));
```

Use the Circuit Breaker

```
// our test server – returns a String ID
UnstableApplication app = new UnstableApplication();

for (int i = 0; i < 10; i++) {
        Thread.sleep(1000);
        breaker.<String> execute(future -> {
            try {
                final String id = app.generateId();
                future.complete(id);
            } catch (SampleException e) {
                future.fail(e);
            }
            if (future.failed()) {
                System.err.printf("failed with exception: '%s' at '%s', circuit-breaker state is: '%s'\n",
                        future.cause(), ZonedDateTime.now(), breaker.state());
            }
        }).setHandler(id -> {
            if (id.succeeded())
                System.out.printf("VertxExample: id '%s' received at '%s'\n", id, ZonedDateTime.now());
        });
    }
```

Running the example

circuit-breaker state is: CLOSED
UnstableApplication throws SampleException at '2017-02-05T20:52:46.466+01:00[Europe/Berlin]' failed with exception: 'com.hascode.tutorial.SampleException' at **'2017-02-05T20:52:46.468+01:00[Europe/Berlin]', circuit-breaker state is: 'CLOSED'**
UnstableApplication throws SampleException at '2017-02-05T20:52:47.469+01:00[Europe/Berlin]'
circuit-breaker opened
failed with exception: 'com.hascode.tutorial.SampleException' at **'2017-02-05T20:52:47.471+01:00[Europe/Berlin]', circuit-breaker state is: 'OPEN'**
circuit-breaker half-opened
UnstableApplication throws SampleException at '2017-02-05T20:52:49.473+01:00[Europe/Berlin]'
circuit-breaker opened
failed with exception: 'com.hascode.tutorial.SampleException' at **'2017-02-05T20:52:49.474+01:00[Europe/Berlin]', circuit-breaker state is: 'OPEN'**
circuit-breaker half-opened
UnstableApplication: id '30f446c5-0703-43b5-8af3-20fe28db52ec' generated at '2017-02-05T20:52:52.476+01:00[Europe/Berlin]'
circuit-breaker closed
VertxExample: id 'io.vertx.core.impl.FutureImpl@1450ed79' received at '2017-02-05T20:52:52.476+01:00[Europe/Berlin]'

Netflix Hystrix

- Library for latency and fault tolerance
- Control over latency and failure from dependencies accessed (typically over the network) via third-party client libraries.
- Stop cascading failures in a complex distributed system.
- Fail fast and rapidly recover.
- Fallback and gracefully degrade when possible.
- Enable near real-time monitoring, alerting, and operational control.

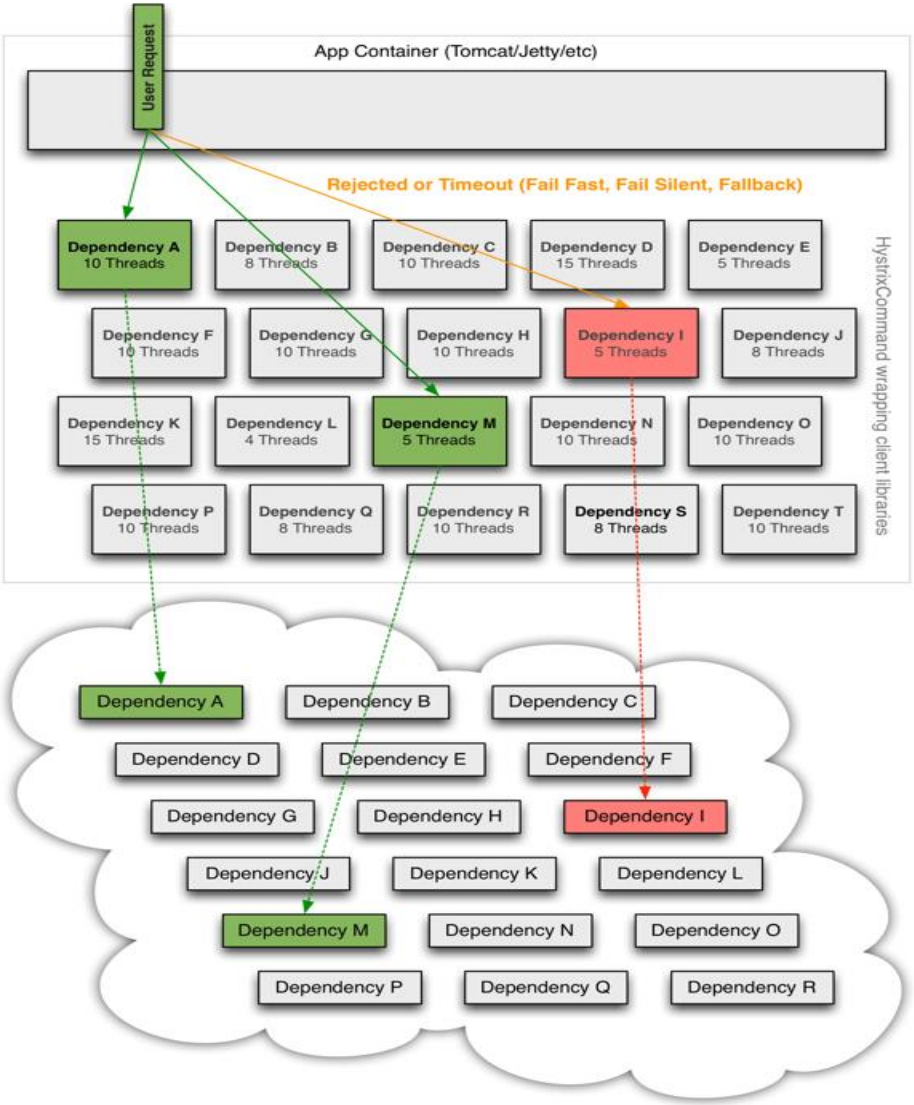
Hystrix

- Wraps all calls to external dependencies in a HystrixCommand/HystrixObservableCommand object
  - typically executes within a separate thread
  - an example of the command pattern
- Times-out calls longer than defined thresholds
- Measures successes, failures (exceptions thrown by client), timeouts,
- Trips a circuit-breaker for a particular service for a period of time, if the error percentage for the service passes a threshold.
- Performing fallback logic when a request fails, is rejected, times-out, or short-circuits.
- Monitoring metrics and configuration changes in near real-time.

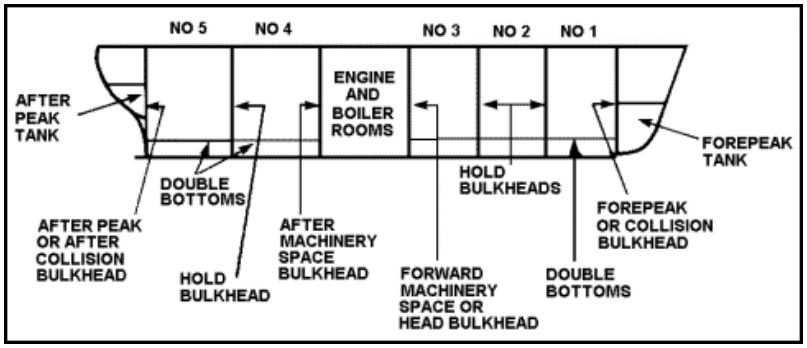
Hystrix CircuitBreaker (Spring)

```
public class BookService {

  private final RestTemplate restTemplate;

  public BookService(RestTemplate rest) {
    this.restTemplate = rest;
  }

  @HystrixCommand(fallbackMethod = "reliable")
  public String readingList() {
    URI uri = URI.create("http://localhost:8090/recommended");

    return this.restTemplate.getForObject(uri, String.class);
  }

  public String reliable() {
    // do default action
  }
}
```

Hystrix Example



Bulkhead Pattern



- Isolates requests into pools
- If one fails/runs out of resources, others unaffected
- Analogous to ship bulkheads

Context

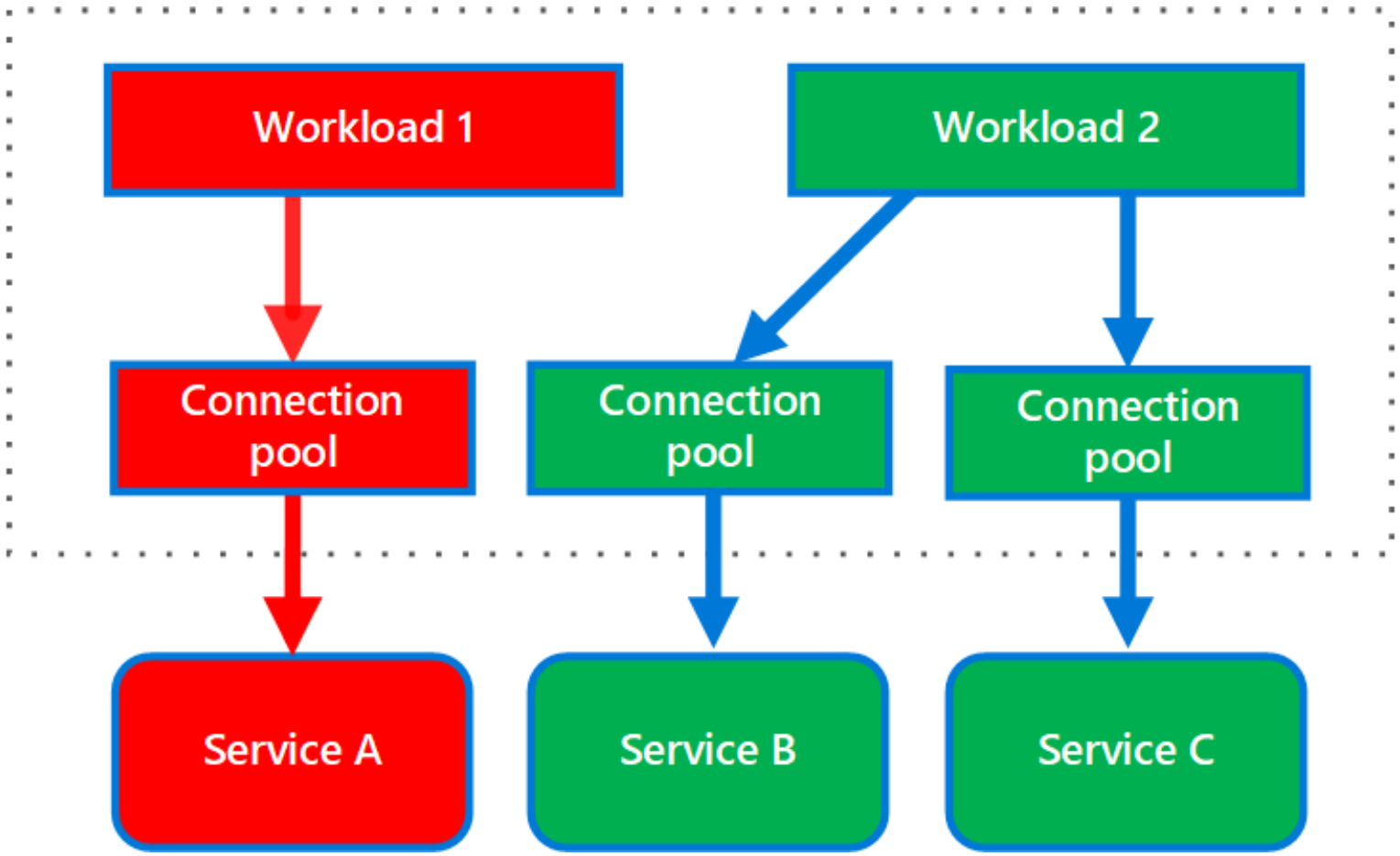
Server (eg Tomcat) has multiple services

- If one is latent, it will consume all threads
- All services affected by single delayed service

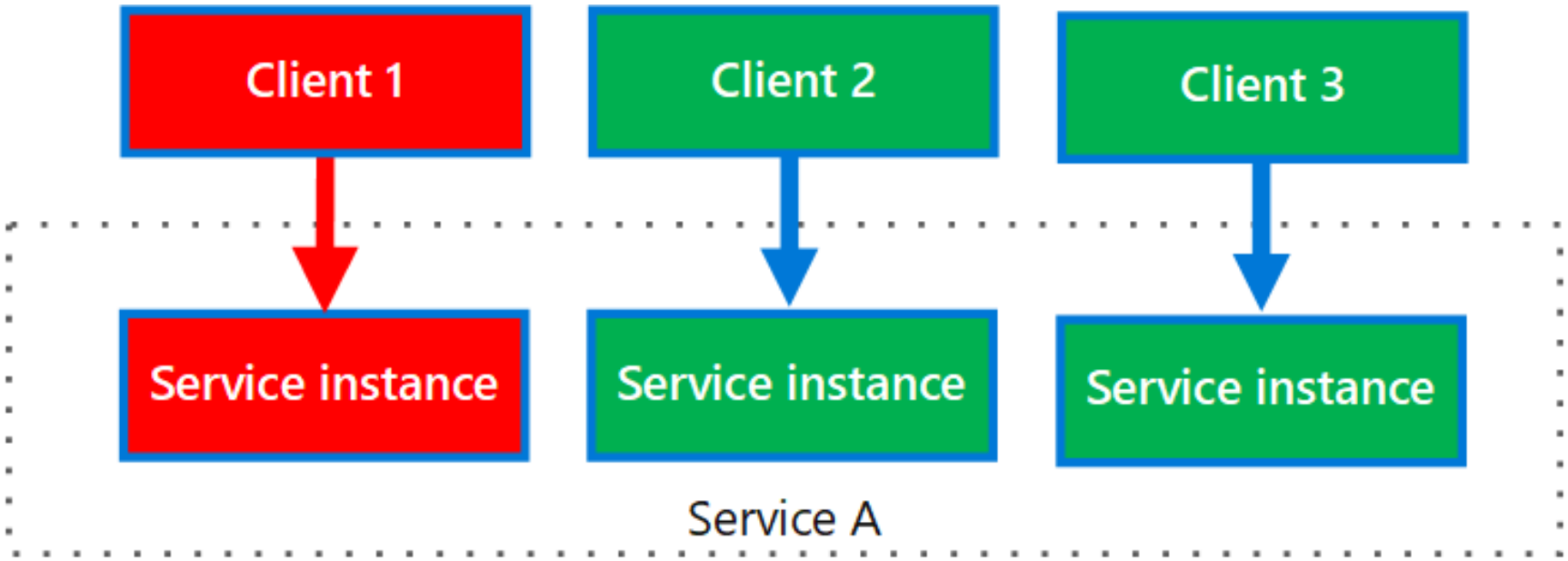
Client may issue requests to many services simultaneously

- Latent dependencies block client threads
- Potential for client thread pool exhaustion for all service requests

Bulkhead Example



Bulkhead Example



Resilience4j Example

```
class ServiceCallerClient {

	private Bulkhead bulkhead;
	private ExternalConcurrentService externalConcurrentService = new ExternalConcurrentService();

	public ServiceCallerClient() {
		 // Create bulk head of 5 max concurrent calls with 2 seconds wait time for entering bulkhead
		BulkheadConfig config = BulkheadConfig.custom().maxConcurrentCalls(5).maxWaitDuration(Duration.ofMillis(5000))
					.build();
		BulkheadRegistry registry = BulkheadRegistry.of(config);
		bulkhead = registry.bulkhead("externalConcurrentService");
	}
	public void callService() { // Wrap service call in bulkhead & call service.
		Runnable runnable = () -> externalConcurrentService.callService();
		bulkhead.executeRunnable(runnable);
	}
}

class ExternalConcurrentService {
	public void callService() {
		try {
			// Mock processing time of 2 seconds.
			Thread.sleep(2000);
			System.out.println(LocalTime.now() + " Call processing finished = " + Thread.currentThread().getName());
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```


The Pragmatic Programmers
Release It!
Second Edition
Design and Deploy
Production-Ready Software
Michael T. Nygard
Edited by Katharine Dvorak

Summary

Microservices are a dominant architecture for scalable systems

Loosely Coupled, cohesive, distributed

Resilience and Cascading Failures

Circuit Breaker Patterns

Bulkhead (Isolation) Pattern