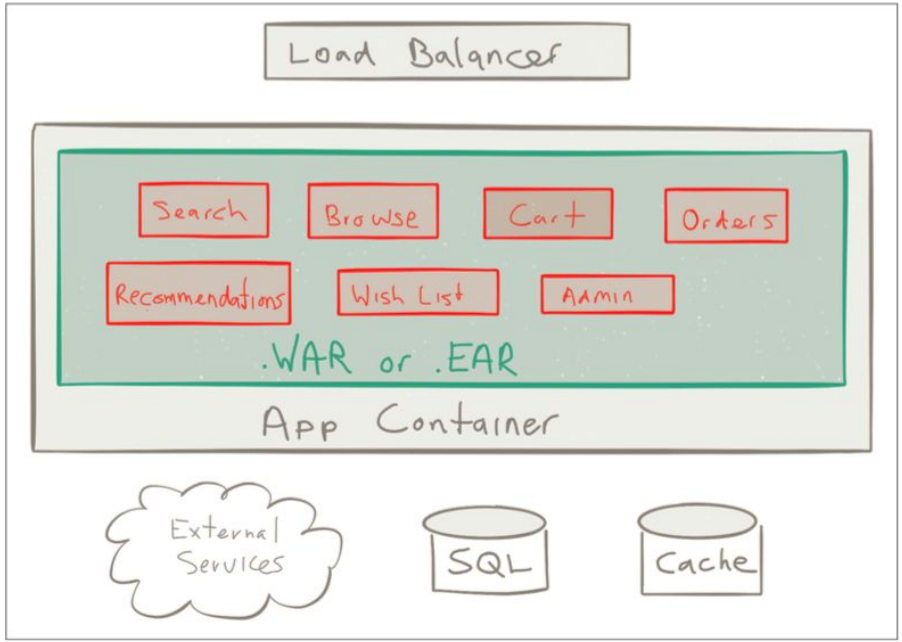
Microservice Design Patterns & Lagom

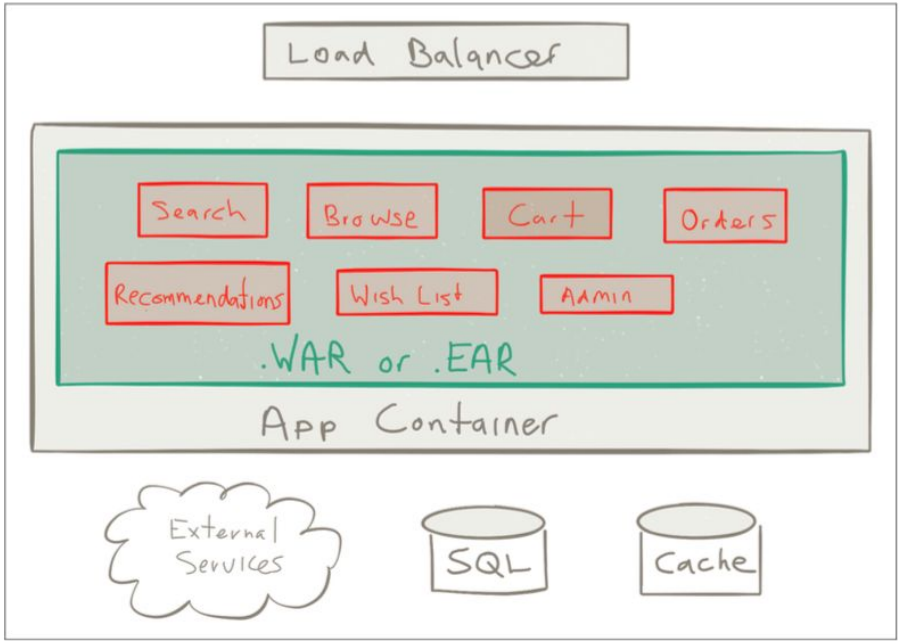
Sean Glover, Senior Consultant, Lightbend
@seg1o

Once upon a time there was the Monolith..

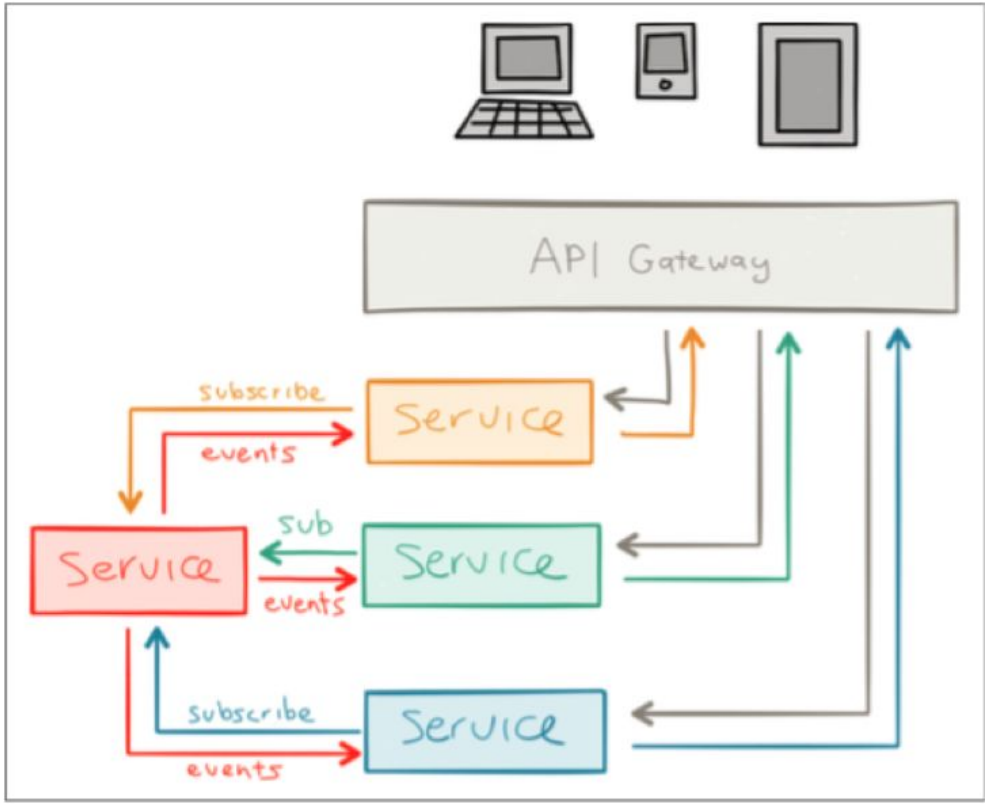


Monoliths

- Easy to get started, hard to maintain
- Tangled responsibilities, lead to infrequent, “big-bang” deployments
- Upgrades are painful
- Small changes become harder to do over time
- App lifetimes months to forever!

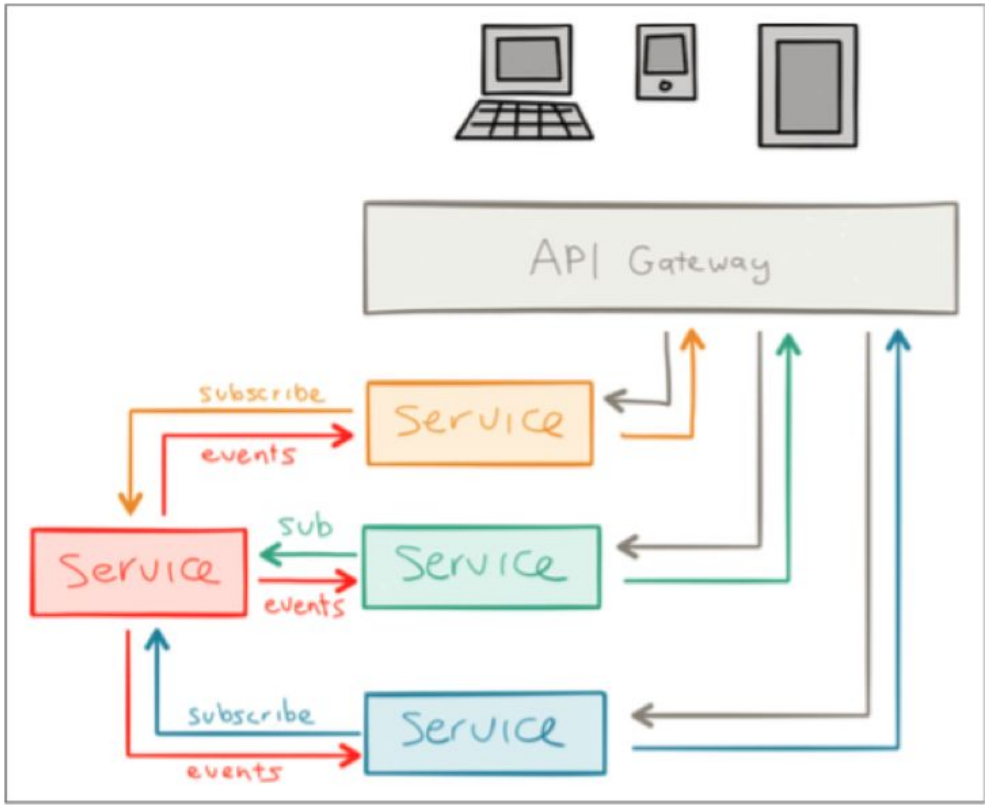


Microservices to the rescue!



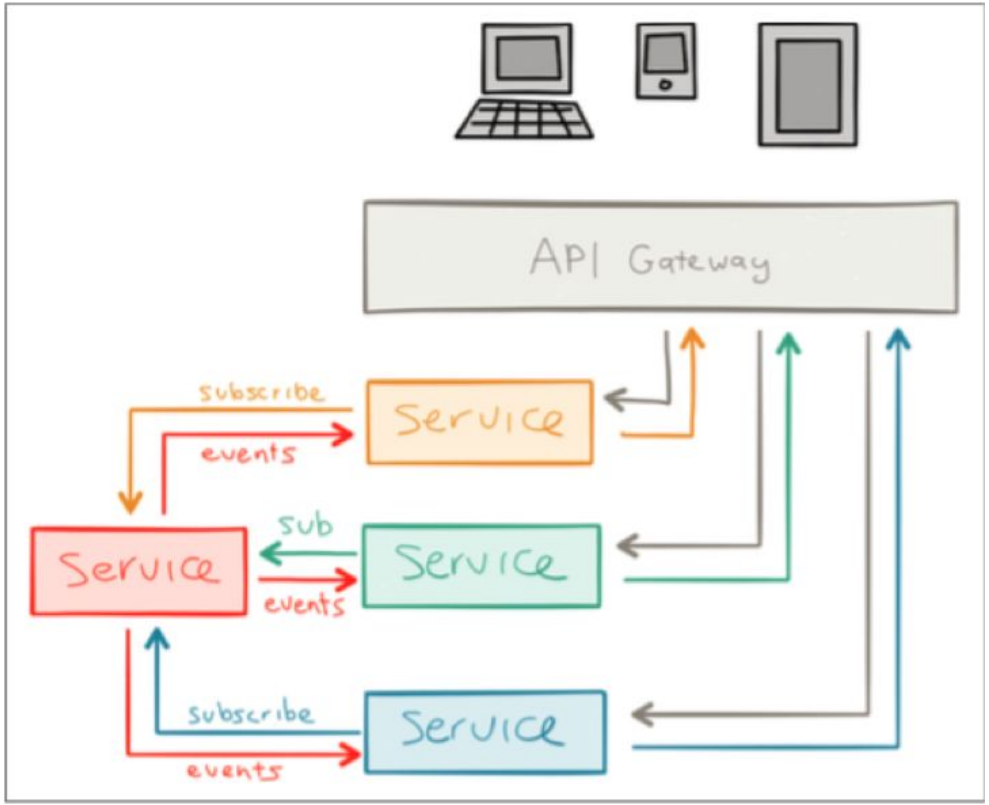
Microservices

- Services/concerns split into independent services
- Asynchronous communication (pub/sub)
- Manages their own data

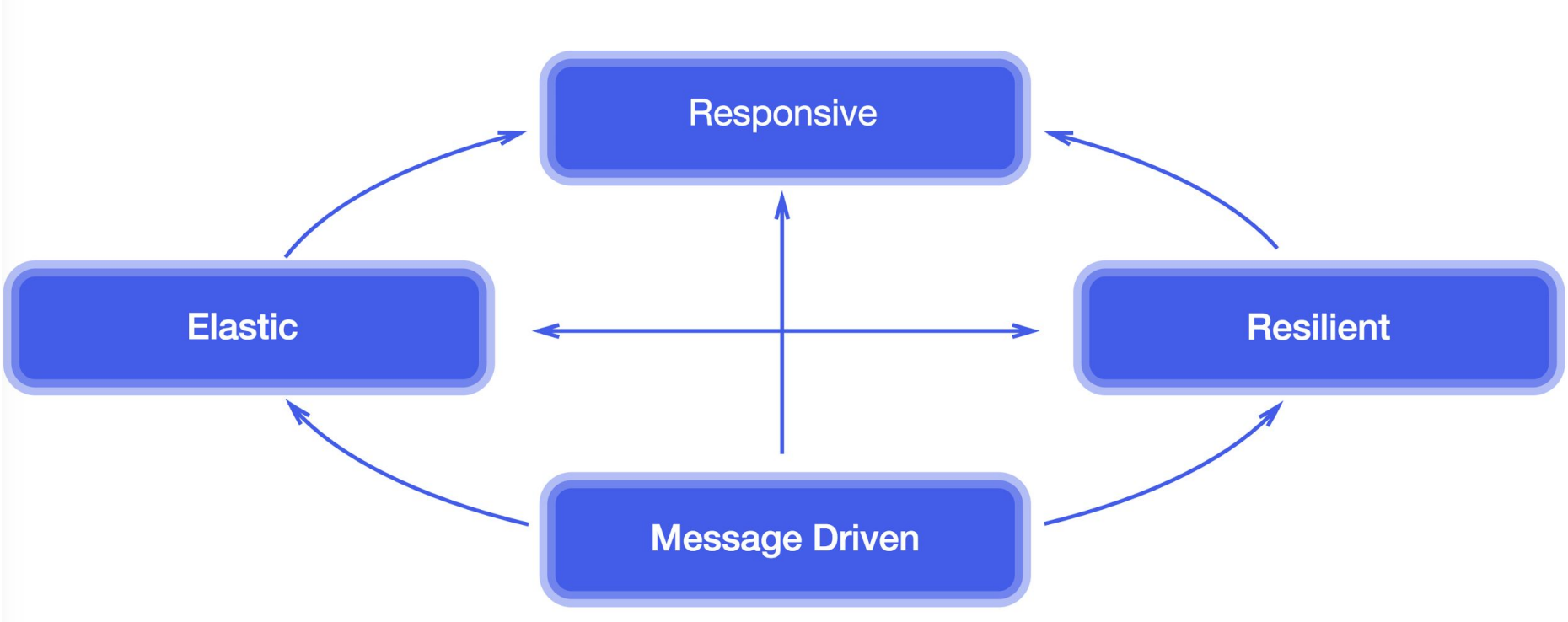


Microservice drawbacks (surprises?)

- Aren’t really drawbacks, just new stuff to people used to monoliths
- Longer latency between services
- Deployment operations: orchestration, automation, etc.



Building Reactive Systems



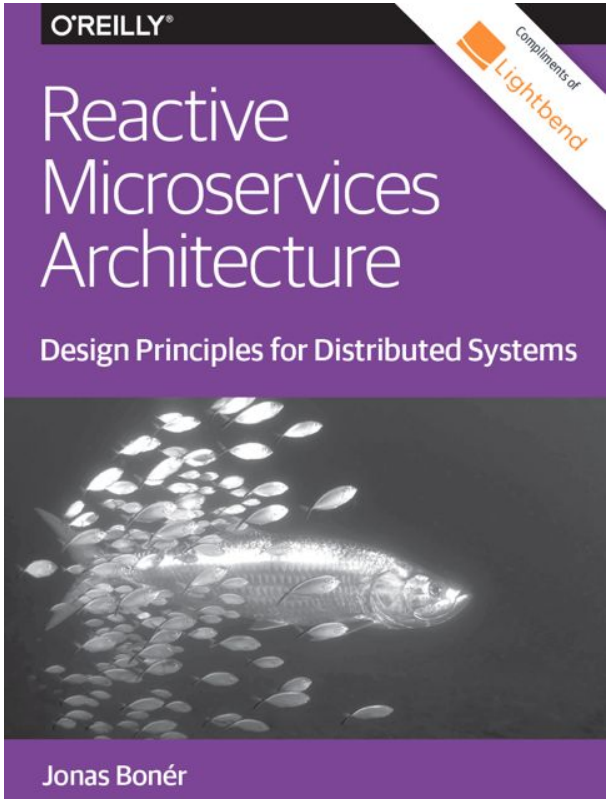
The Reactive Manifesto

Microservice Design Patterns

“Microservices-Based Architecture is a simple concept: it advocates creating a system from a collection of small, isolated services, each of which owns their data, and is independently isolated, scalable and resilient to failure.”

- Jonas Bonér, CTO Lightbend

[Book link]



Properties of a microservice

Based on the definition of microservice from the previous slide, let’s iterate over the properties of such a microservice.

- Isolation - Decoupling, failure bulk heading, scale independently
- Acting autonomously - Have enough information to make decisions independently
- Do one thing - A core or sub domain, as defined in DDD
- Own your own state - Share nothing architecture, including data, at the sacrifice of normalization and single source of truth.
- Embrace async messaging

Isolation

- The impact on initial design, something to be considered from day one.
- Failure isolation - bulk heading
- Scaling out

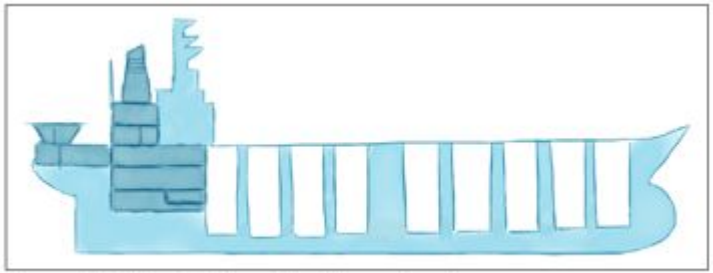


Figure 2-1. Using bulkheads in ship construction

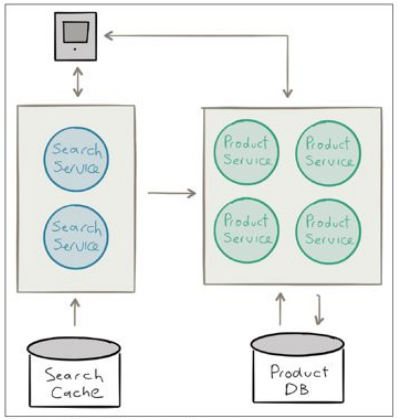


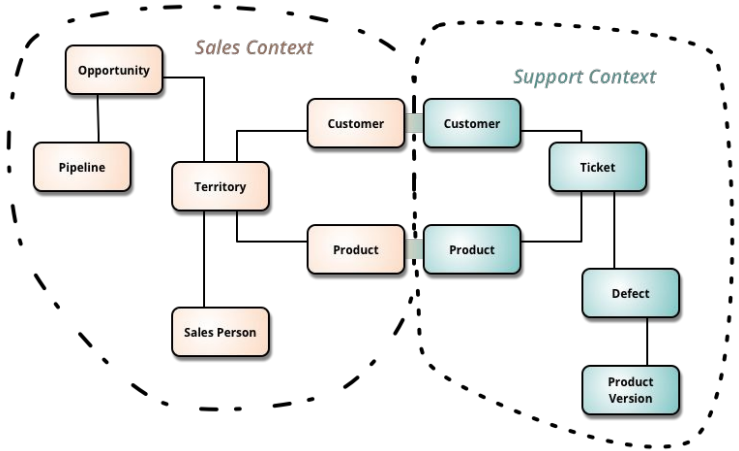
Figure 2-2. Bounded contexts of Microservices

Acting Autonomously

- Don’t create a distributed monolith - Whale analogy
- Isolation is a pre-requisite
- Resiliency when other services are down - If dependencies are unavailable, what do we do?
- Independently make decisions

Do one thing

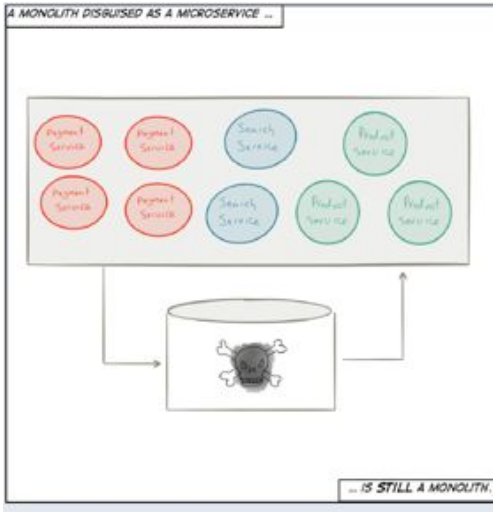
- Do one thing and do it really well
- The same idea as the Unix philosophy of building standalone tools for system programming. Single Responsibility Principle from SOLID.
- How big is a “microservice”?
- Modeling your services with DDD. Bounded context and service could be 1:1
- Scale your dev teams the same way!

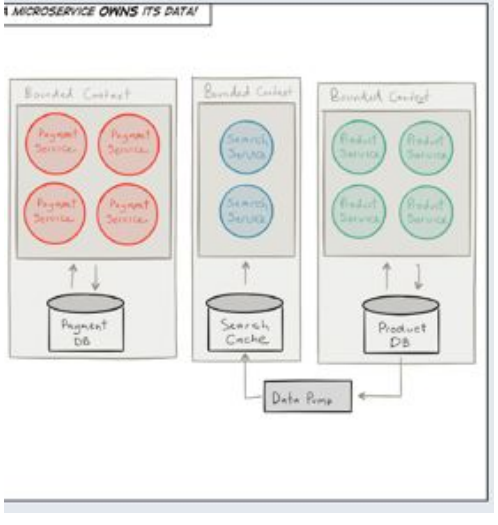


Source: [Bounded-context by Martin Fowler]

Own your state exclusively

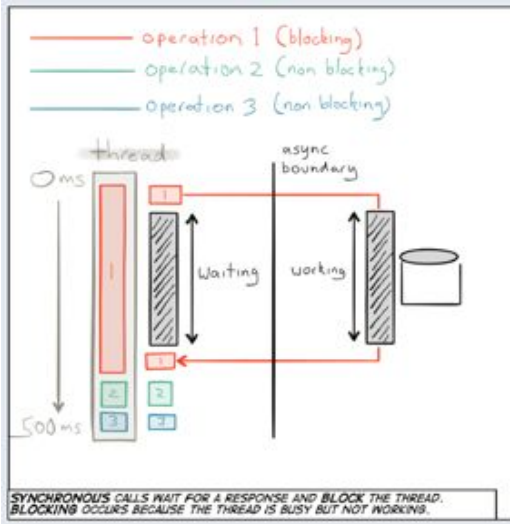
- Each service has state it has to deal with
- Don’t share state in databases. This delegates the problem of keeping state to a 3rd party which services can corrupt.
- Each service should manage its own state and underlying persistence
- State that needs to be shared should be pushed or pulled to other services
- Duplicate data is OK

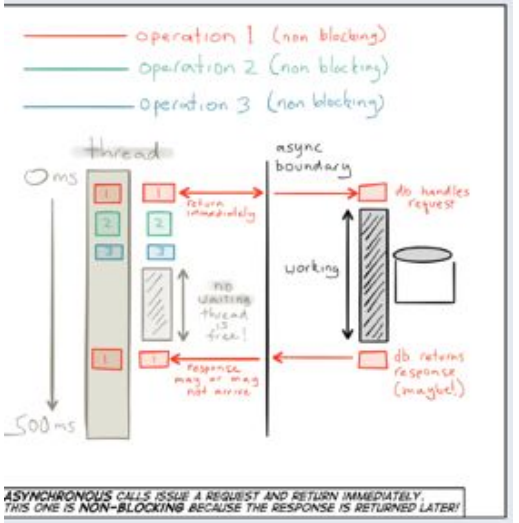




Embrace async message passing

- Synchronous request/response message passing can cause bottlenecks. Blocking is bad!
- Increases throughput
- Fire and forget. Responsibility of sender to detect and deal with failure.





Lagom - [lah-gome]

Adequate, sufficient, just right

Why Lagom?

- Opinionated
- Developer experience matters!
  - No brittle script to run your services
  - Inter-service communication just works
  - Services are automatically reloaded on code change
- Takes you through to production deployment

Under the hood

- sbt build tool (developer environment)
- Play 2.5
- Akka 2.4 (clustering, streams, persistence)
- Cassandra (default data store)
- Jackson (JSON serialization)
- Guice (DI)

Anatomy of a Lagom project

Each service definition is split into two sbt projects: api & impl

```
your-lagom-system          → Project root
 └ helloworld-api          → helloworld api project
 └ helloworld-impl         → helloworld implementation
project
 └ project                 → sbt configuration files
   └ plugins.sbt           → sbt plugins
 └ build.sbt               → Your project build file
```

Demo time

Service API

Service definition

```
// this source is placed in your api project
trait HelloService extends Service {
  override def descriptor(): Descriptor = {
    named("helloservice").withCalls(
      namedCall("/hello", sayHello _)
    )
  }

  def sayHello(): ServiceCall[String, String]
}
```

Strict Messages

```
override def descriptor(): Descriptor = {
  named("helloservice").withCalls(
    namedCall("/hello", sayHello _)
  )
}

def sayHello(): ServiceCall[String, String]
```

Strict messages are *fully* buffered into memory

Streamed Messages

```
override def descriptor(): Descriptor = {
  named("clock").withCalls(
    pathCall("/tick/:interval", tick _)
  )
}
def tick(interval: Int): ServiceCall[String, Source[String, _]]
```

WebSocket

Remember the **Service** definition?

```
// this source is placed in your api project
trait HelloService extends Service {
  override def descriptor(): Descriptor = {
    named("helloservice").withCalls(
      namedCall(sayHello _)
    )
  }

  def sayHello(): ServiceCall[String, String]
}
```

Here is the **Service** implementation

```
// this source is placed in your implementation project
class HelloServiceImpl extends HelloService {
  override def sayHello(): ServiceCall[String, String] = {
    name => Future.successful(s"Hello, $name!")
  }
}
```

Inter-service communication

```
class MyServiceImpl @Inject()(helloService: HelloService)
  (implicit ec: ExecutionContext) extends MyService {

  override def sayHelloLagom(): ServiceCall[NotUsed, String] = unused => {
    val response = helloService.sayHello().invoke("Lagom")
    response.map(answer => s"Hello service said: $answer")
  }
}
```

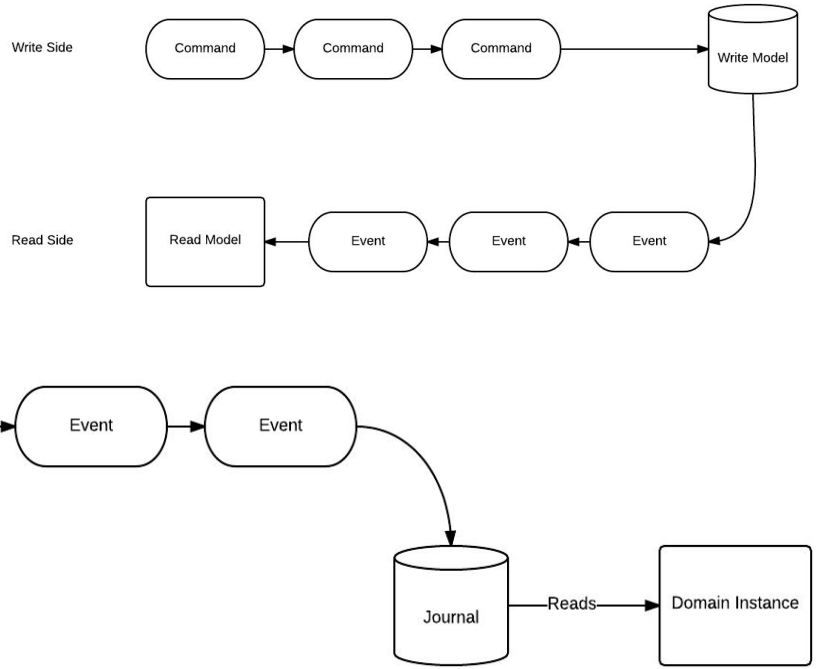
Persistence API

Principles

- Each service *owns* its data
  - Only the service has direct access to the DB
- We advocate the use of Event Sourcing (ES) and CQRS
  - ES: Capture *all* state's changes as events
  - CQRS: separate models for write and read

Benefits of Event Sourcing/CQRS

- Allows you to time travel
- Audit log
- Future business opportunities
- No need for ORM
- No database migration script, ever!
- Performance & Scalability
- Testability & Debuggability



Event Sourcing: Write Side

- Create your own *Command* and *Event* classes
- Subclass **PersistentEntity**
  - Define *Command* and *Event* handlers
  - Can be accessed from anywhere in the cluster
  - (corresponds to an *Aggregate Root* in DDD)

Event Sourcing: Read Side

- Tightly integrated with Cassandra
- Create the query tables:
  - Subclass **CassandraReadSideProcessor**
  - Consumes events produced by the **PersistentEntity** and updates tables in Cassandra optimized for queries
- Retrieving data: Cassandra Query Language
  - e.g., `SELECT id, title FROM postsummary`

Lagom Supports

- Java API!
- Maven support
- Message broker integration
- Scala API
- Support for other cluster orchestration tools
  - Want Kubernetes support? Contribute! https://github.com/huntc/kubernetes-lib
- Coming soon:
  - Support for writing integration tests
  - Swagger integration

Lagom Resources

- Try Lagom yourself
  - https://lightbend.com/lagom
- Using Scala with Lagom
  - https://github.com/dotta/activator-lagom-scala-chirper/releases/tag/v02_scaladays_berlin_2016
- Lagom on Github
  - https://github.com/lagom/lagom
- Read Jonas Bonér's *free* ebook *Reactive Services Architecture*
  - https://lightbend.com/reactive-microservices-architecture
- Great presentation by Greg Young on why you should use ES
  - https://www.youtube.com/watch?v=JHGkaShoyNs

That’s it!

Questions?

@seg1o