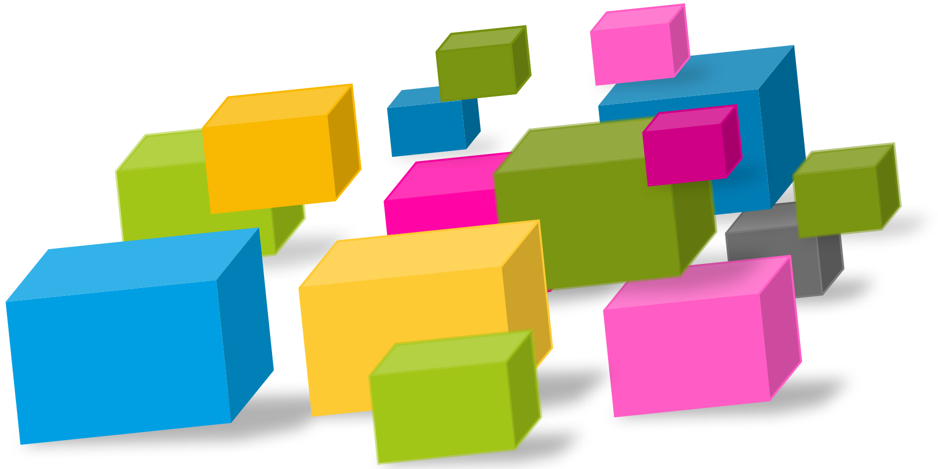
Microservice Architecture Blueprint Episode 3 “The Real Thing“

Kim Horn

Version 1.11 1 August 2017

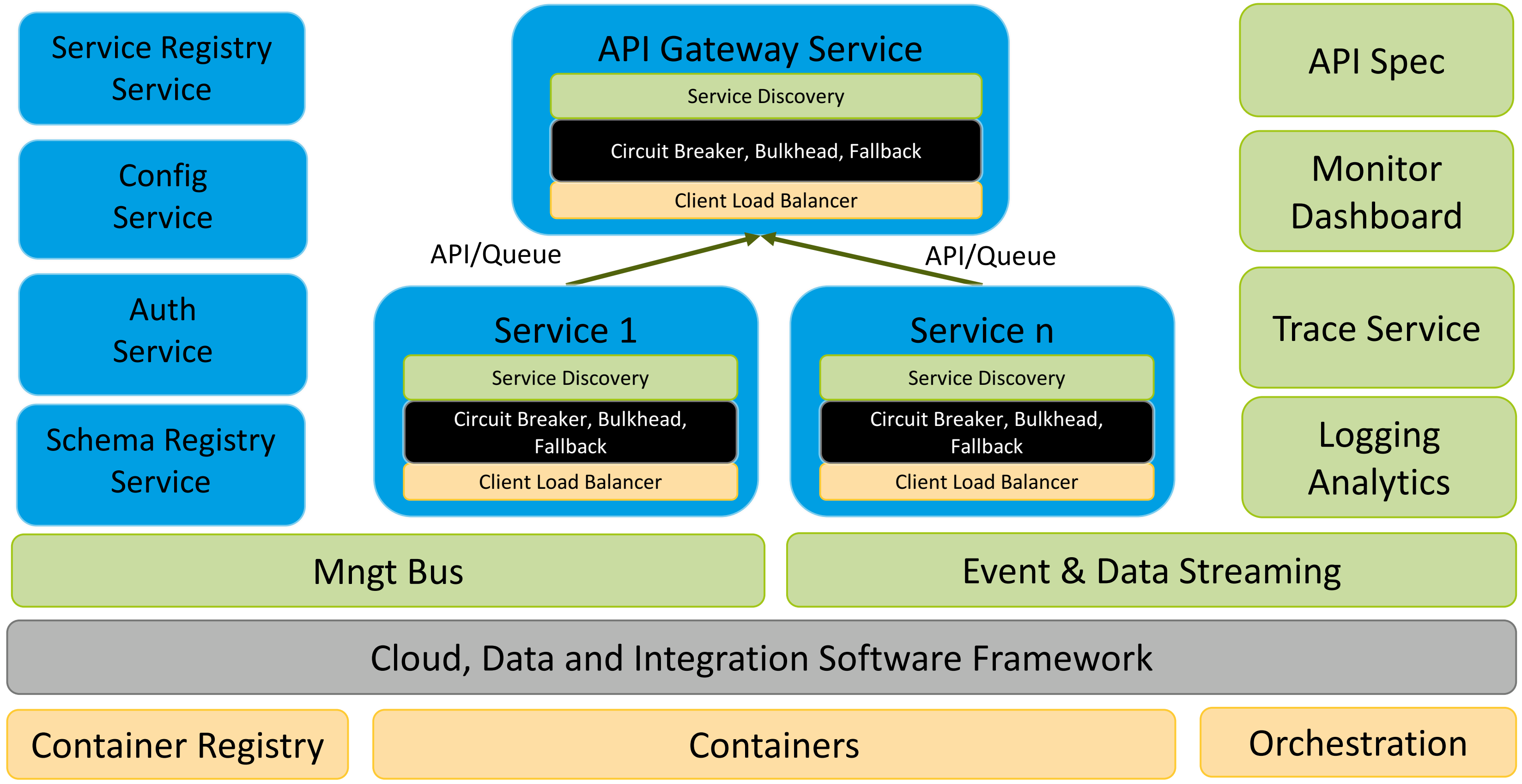


Agenda

- Recap Episode 1
- Introduce ‘Real’ Example Problem
- Decompose – Bounded Context
- Deployment in Containers
- REST
- HAL
- Demo
  - JSON – First Pass at API calls for Business Services

Working Blueprint

Distributed Cloud Blueprint – NFRs, use what you need



Common Tech For Patterns and Stack

Pattern / Capability	Tech
Gateway (reverse proxy)	Zuul 1&2 (Netflix) – wrapped by SpringCloud, Spring Cloud Gateway
Circuit Breaker, Bulkhead, Fallback	Hysterix (Netflix) – wrapped by SpringCloud
Client Side Load Balancer	Ribbon (Netflix) – wrapped by SpringCloud
Service Discovery Registry	Eureka (Netflix) – wrapped by SpringCloud / Consul
REST HTTP and Service Discovery Client	Feign (Netflix) – wrapped by SpringCloud
Configuration Service	Spring Cloud Configuration
Authorisation Service	Spring Cloud Security, SAML, OATH, OpenID
Event Driven Architecture, Mngt Bus	Spring Cloud Stream wraps Kafka (also supports Rabbit MQ), Spring Cloud Bus
Data Analysis and Flow Pipelines	Spring Cloud Data, Spring Cloud Data Flow, Sparks
Core Software Cloud Frameworks	Spring Boot, Spring Actuator, Spring Cloud, Spring Data, Spring Integration
IaaS, Container, Orchestration	Docker, Compose [Kubernites, Swarm]
Logging, Tracing & Analysis	SL4J, ELK[Elastisearch, Logstash, Kibana], Sprint Cloud Sleuth, Spring Cloud Zipkin.
In memory Cache	Spring Data Redis
Analytics, Monitoring Dashboard, Orchestration(bpm)	Atlas, Turbine, Visceral, Camuda, Conductor (lightweight)
API/ Service Specs, Schema Registry,Testing	Spring Hateoas (HAL), Spring Fox (Swagger), Avro, Protobuf, Spring Cloud Contract, WireMock

Microservice Architecture Principles

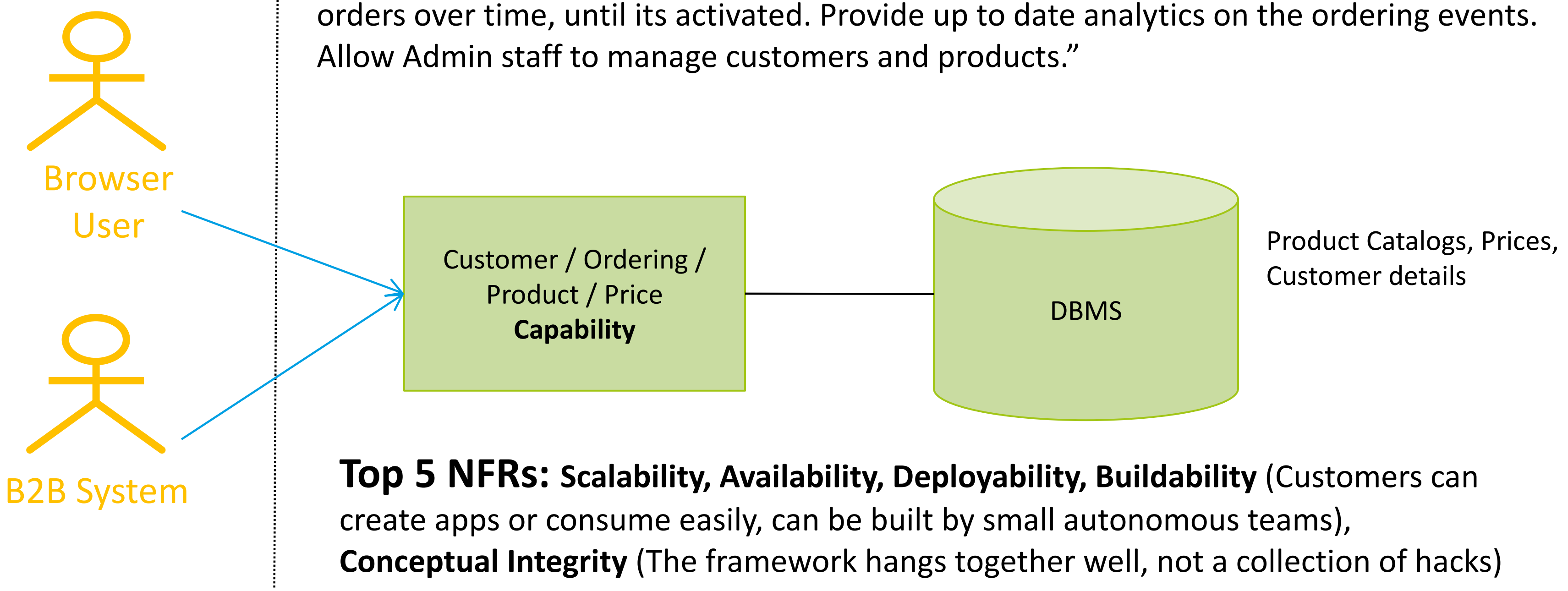
Microservices are:

1. Modelled around a bounded context; the business domain.
2. Responsible for a single capability;
3. Individually deployable, based on a culture of automation and continuous delivery;
4. The owner of its data;
5. Consumer Driven;
6. Not open for inspection; Encapsulates and hides all its detail;
7. Easily observed;
8. Easily built, operated, managed and replaced by a small autonomous team;
9. A good citizen within their ecosystem;
10. Based on Decentralisation and Isolation of failure;

Example

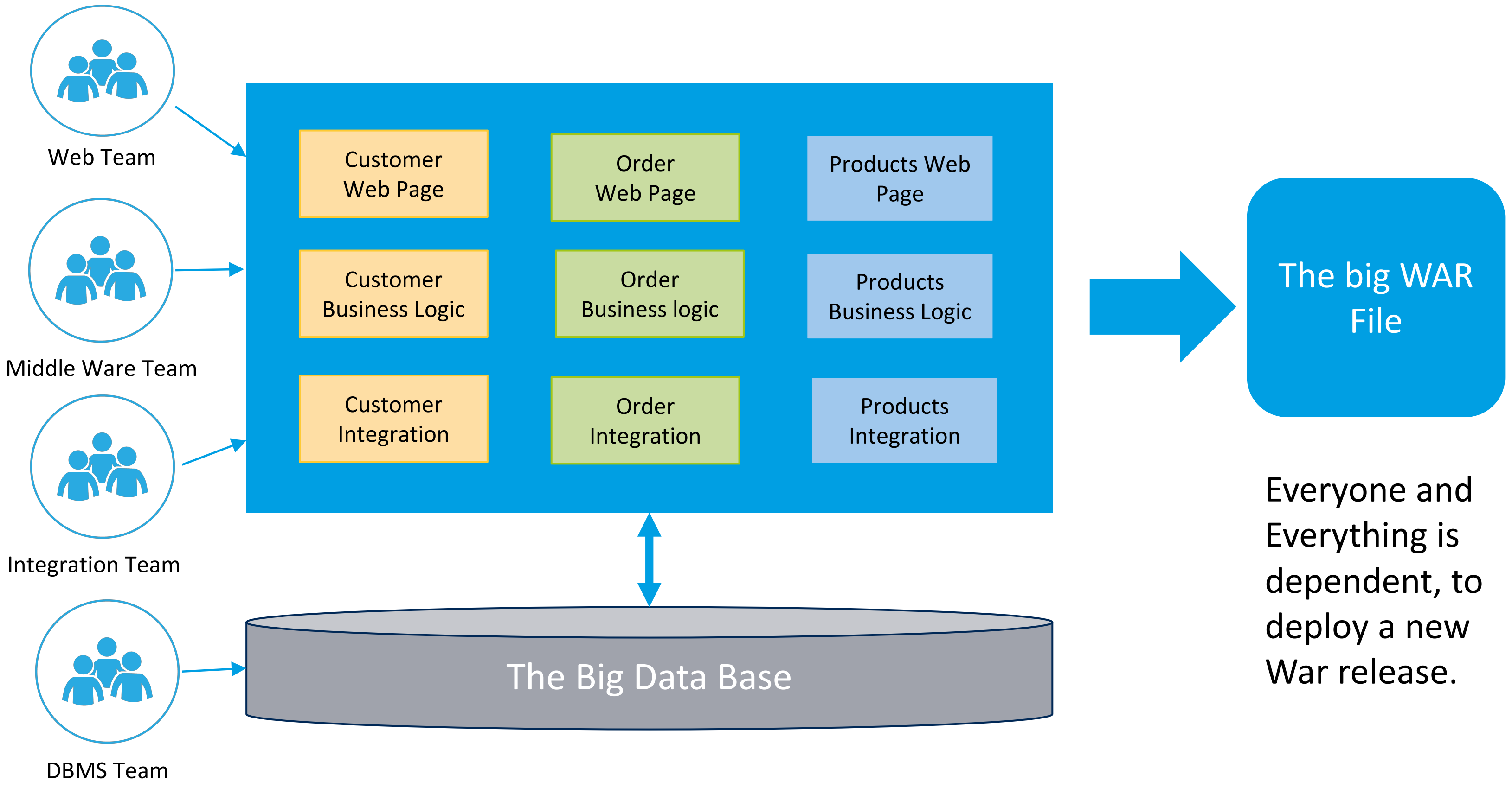
Product Ordering Example – ‘Concept’

“Allow our customers to create, manage and submit orders, and add items to their order for purchase. Get itemised and total price. Inform the customer of the status of their orders over time, until its activated. Provide up to date analytics on the ordering events. Allow Admin staff to manage customers and products.”


Browser User
B2B System
Customer / Ordering / Product / Price **Capability**
DBMS
Product Catalogs, Prices, Customer details

Top 5 NFRs: Scalability, Availability, Deployability, Buildability (Customers can create apps or consume easily, can be built by small autonomous teams), **Conceptual Integrity** (The framework hangs together well, not a collection of hacks)

A ‘Standard’ Layered Monolith Architecture



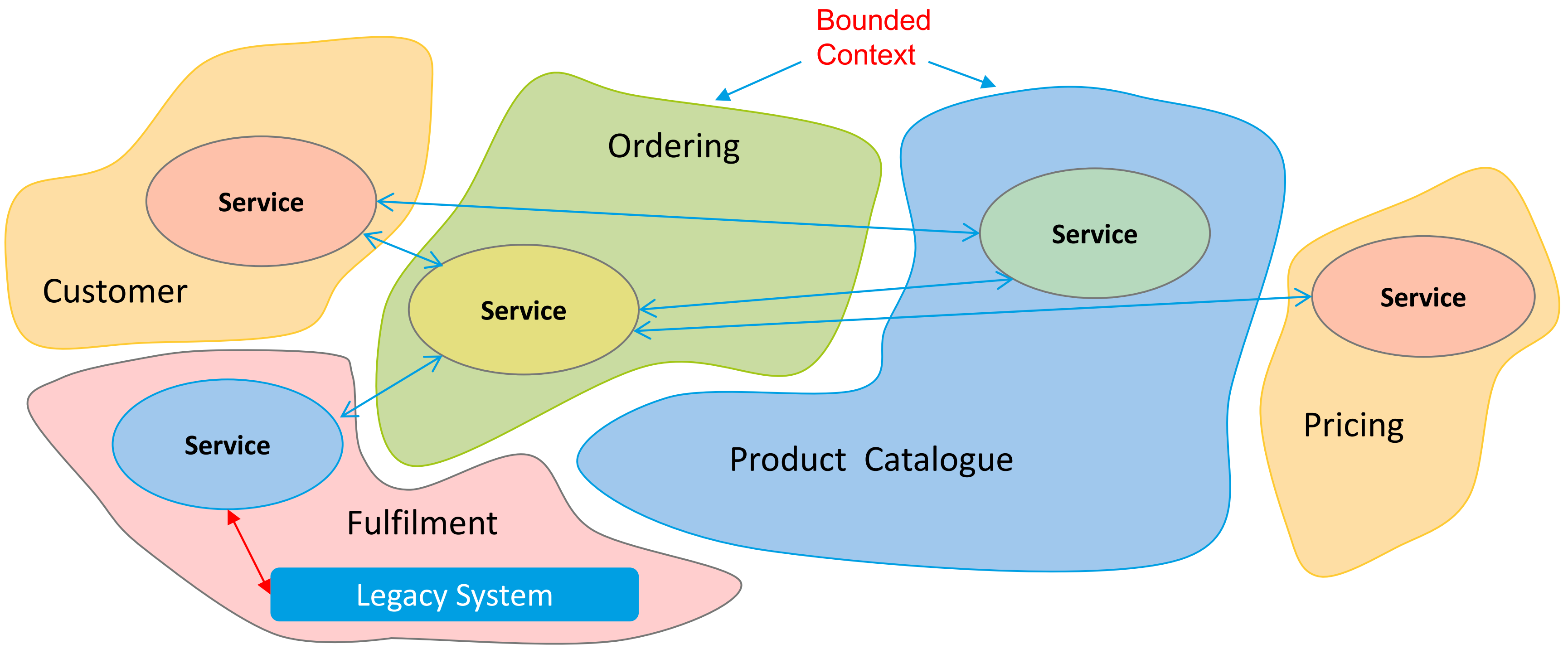
Everyone and Everything is dependent, to deploy a new War release.

Decomposition of the Domain

- Domain Model
    - Subdomains for Microservices;
- Context Map
    - How the small service teams interact;
    - Types of relationships – the politics;
- Business Capability modeling;
- Non-Functional Services for cross cutting concerns.

Domains and Services

- Each Business Domain's Capability is offered as a 'Service'.
- A single verb can define each domain service.
- Preserve the domains and use microservices as the way forward to **help** provide autonomy.



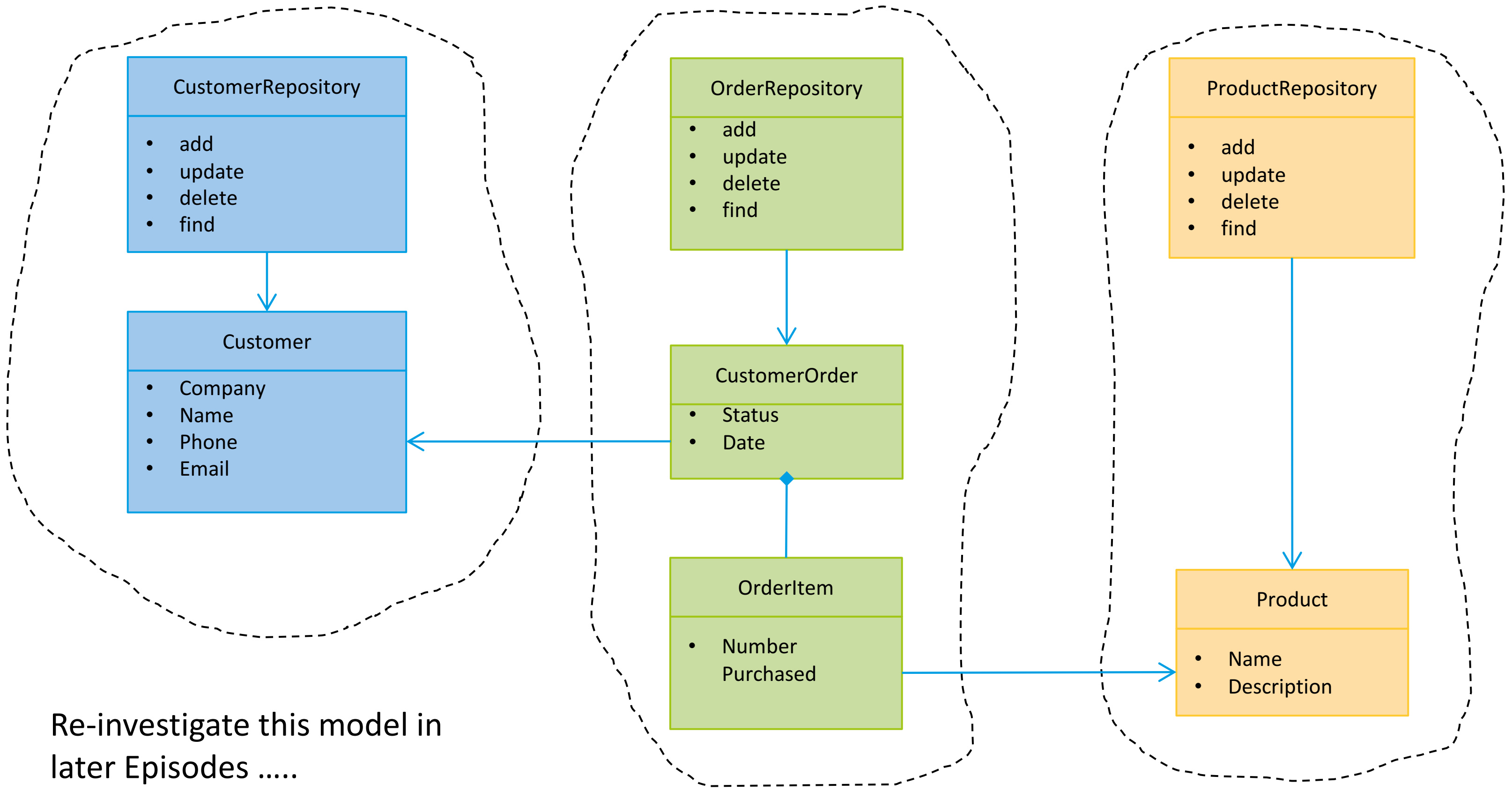
Things in a Domain Model

The main types of 'things', Building Blocks, in domain models includes:

- **Services** – operations that do not sit within an Entity, Services have a specific context;
- **Entities** – an object defined by its identity and continuity;
- **Aggregates** – a collection or cluster of objects that are bound together;
- **Domain Events** – happening of interest in the domain, can be used to decouple systems for high performance;
- **Value Objects** – an object with no identity, still has attributes as per entity;
- **Repositories** – save and retrieve entities and aggregates;
- **Factories** – a mechanism to create objects that decouples the implementation.

Domain Models are not Data Models but OO Models; behaviour oriented.

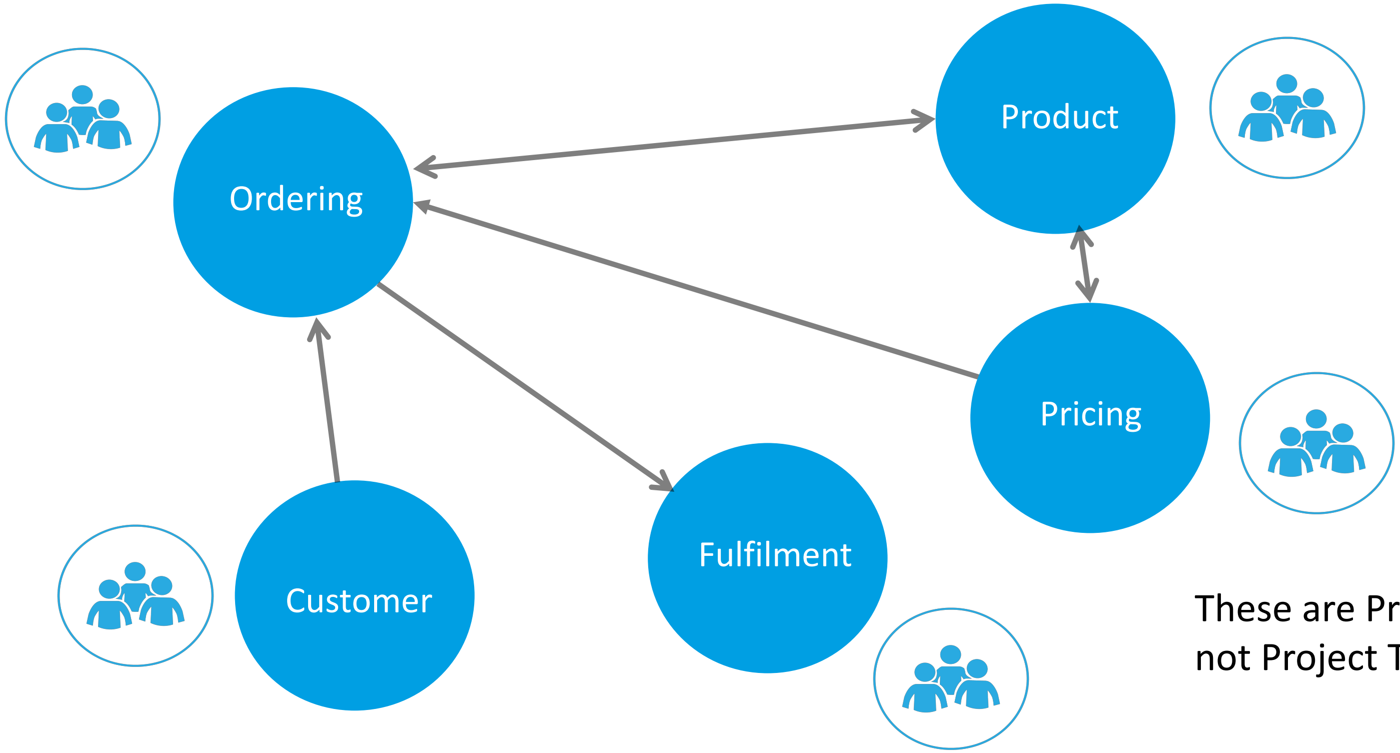
3 Services – 3 Entity Aggregates



Re-investigate this model in later Episodes .....

Service Relationships – Context Map

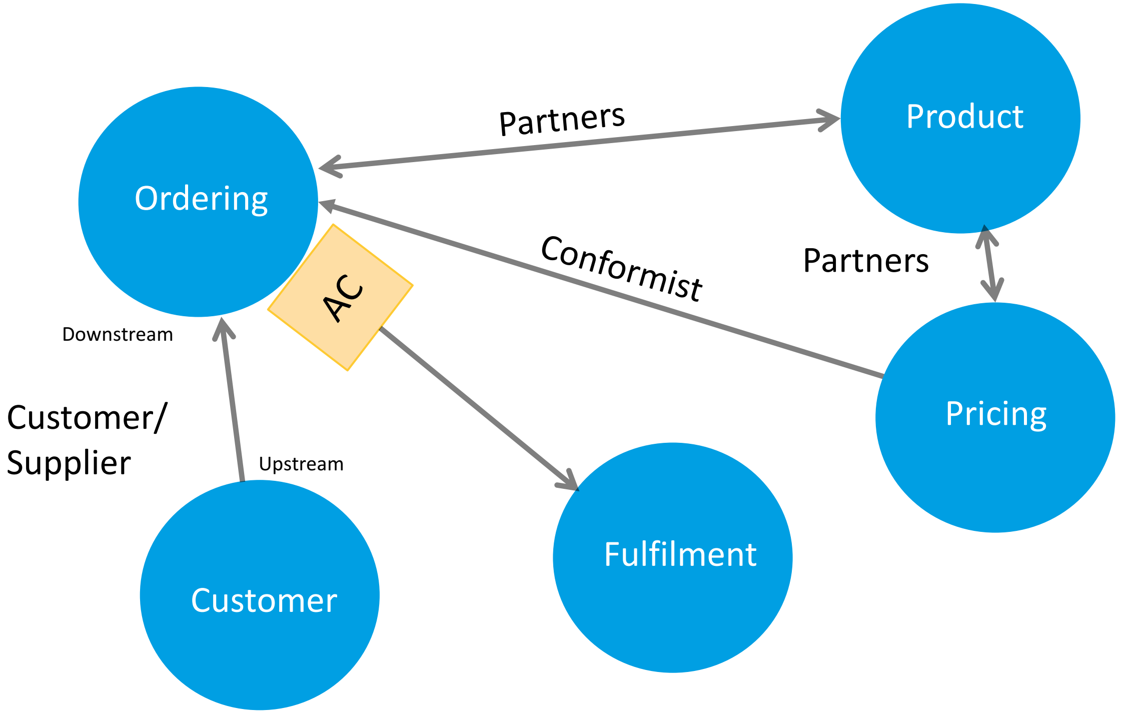
Goal is to bind teams to their Bounded Context; they work in a business capability domain not a technology domain.



These are Product Teams not Project Teams

Service Relationships – Context Map Relationships

Arrow Points to Power - Not simply 'Autonomous', this needs to be modelled



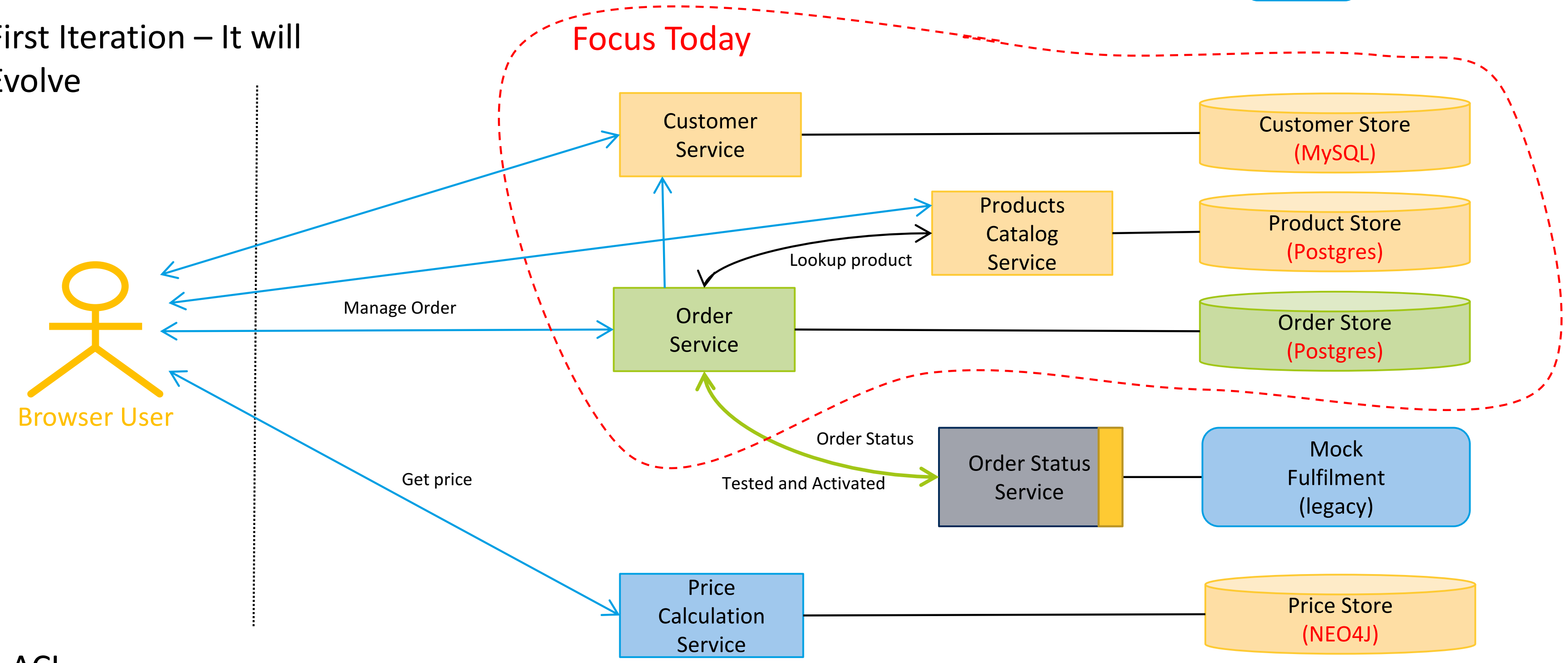
Relationship Types:

- Partnership
- Customer/Supplier
- Conformist
- ACL
- Shared Kernel
- Open Host Service
- Partners
- Published Language
- Big Ball of Mud
- Separate Ways

Decompose: 5 Functional Services

DP

First Iteration – It will Evolve



▮ ACL

Pattern: Service Granularity, Polyglot, DataBase per Service, ACL

Re-Use

Autonomy Vs Re-use

Microservices are primarliy about how you build a single applications. Reuse is a separate concern and is not a prime motive of Microservices. Indeed a single application should in may ways provide novel capabilities, bounded by context, that do not necessitate reuse. However, Internal re-use will become a concern, as it is natural to re-use service that exist and work.

- Every new consumer using your service creates a new dependency, that has to be managed into the future;
- Reuse has a cost, and in some case this can start small but get very large;
- Reuse raises concerns about isolation, of the domain and the components;
- When a team becomes responsible for a service, they need to decide how new dependencies impact them; how autonomous they want to be, and can afford ?
- It comes back to the 'kind' of relationship, and the degree of team autonomy, how much do you want to protect (bound) your domain model from bleeding by others;
- The boundary and way a service is exposed is important; Microservices are internal to the bounded context, not a way for exposing APIs;

Given microservices should be easliy replaced, a team wanting to use your service should be able to build their own, if the provider just cannot support the consumer.
A microservice that is similar to another, functionally, but delivers a very different quality of service is a different component.

DRY – Don’t Repeat Yourself

Rule of thumb: don’t violate DRY within a microservice, but be relaxed about violating DRY across all services. The evils of too much coupling between services are far worse than the problems caused by code duplication.

DRY means that we want to avoid duplicating our system *behavior and knowledge*. More than just duplicating code.

- DRY is what leads us to create code that can be reused. We pull duplicated code into abstractions that we can then call from multiple places, maybe as a shared library that we can use everywhere!
- This approach, however, can be deceptively dangerous in a microservice architecture.

Microservices are all about Separation of Concerns and using ‘modules’ to do this, but not driven by re-use.

One of the things we want to avoid at all costs is overly coupling a microservice and consumers such that any small change to the microservice itself can cause unnecessary changes to the consumer.

- If your use of shared code ever leaks outside your service boundary, you have introduced a potential form of coupling.
- For example, a library of common domain objects that represented the core entities in use in our system. This library was used by all the services. But when a change was made to one of them, all services had to be updated, when each could schedule and afford to do this.

What about Client Libraries – Is this bad ?

Some service teams suggest creating client libraries to help the use of services, makes it easier for their consumers. However, sharing Client Libraries creates coupling. Code re-use is the wrong approach.

The Framework proposed by Netflix does involve sharing client libraries.

The Netflix client libraries handle service discovery, failure modes, logging, and other aspects that aren't actually about the nature of the service itself.

- Without these shared clients, it would be hard to ensure that each piece of client/server communications behaved well at the massive scale at which Netflix operates.
- Their use at Netflix has certainly made it easy to get up and running and increased productivity while also ensuring the system behaves well.
- However, according to at least one person at Netflix, over time this has led to a degree of coupling between client and server ;
- The acknowledgement of this coupling is what important, how autonomous ca we afford to be;
- Important is that the quality driver is not about code-reuse but resiliency and scalability.

Its about the Prime Driver of an Architectural Style.

The **Microservice Architecture Style** is about decomposition of an application for agility and isolation. Share as little as possible.

The **Service Oriented Architecture Style (SOA)** is about decomposition of the enterprise for re-use. Share as much as possible.

- The concept of a *share-as-much-as-possible* architecture solves issues associated with the duplication of business functionality but lead to tightly coupled components and increases the overall risk associated with change.
- **Using both styles is fine but means we have to be aware of the tradeoffs. How Context Relationships effect Autonomy.**
- **Re-use sounds ideal but can be costly and severely reduce agility.**

DEPLOYMENT

4 by 3 Matrix of Deployment Options

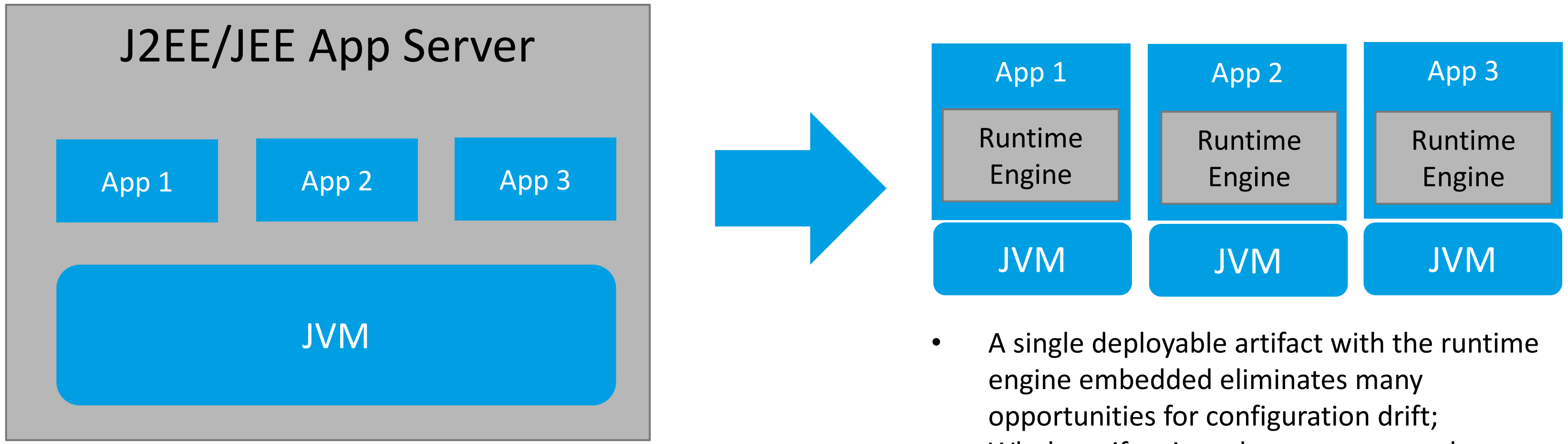
Deploy	Stand Alone Monolithic Application	Application(s) in App Server (cluster)	Microservice – built in runtime engine
Host Server(s)			
VMs on Server(s)			
Containers*			✔
Function Fabric (e.g. AWS Lambda, MS fabric)	Not	Possible	

* Not OSGI, not app servers, not tomcat or NGINX….

Deployment Patterns

- Multiple Service Instances per Host, Process or Process Group Pattern
- Service Instance per Host Pattern
- Service Instance per Virtual Machine Pattern
- Service Instance per Container Pattern
- Application Runtime Engine within Service
- Server-less (Function Fabric) Service Pattern, e.g. AWS Lambda, Azure Service Fabric.

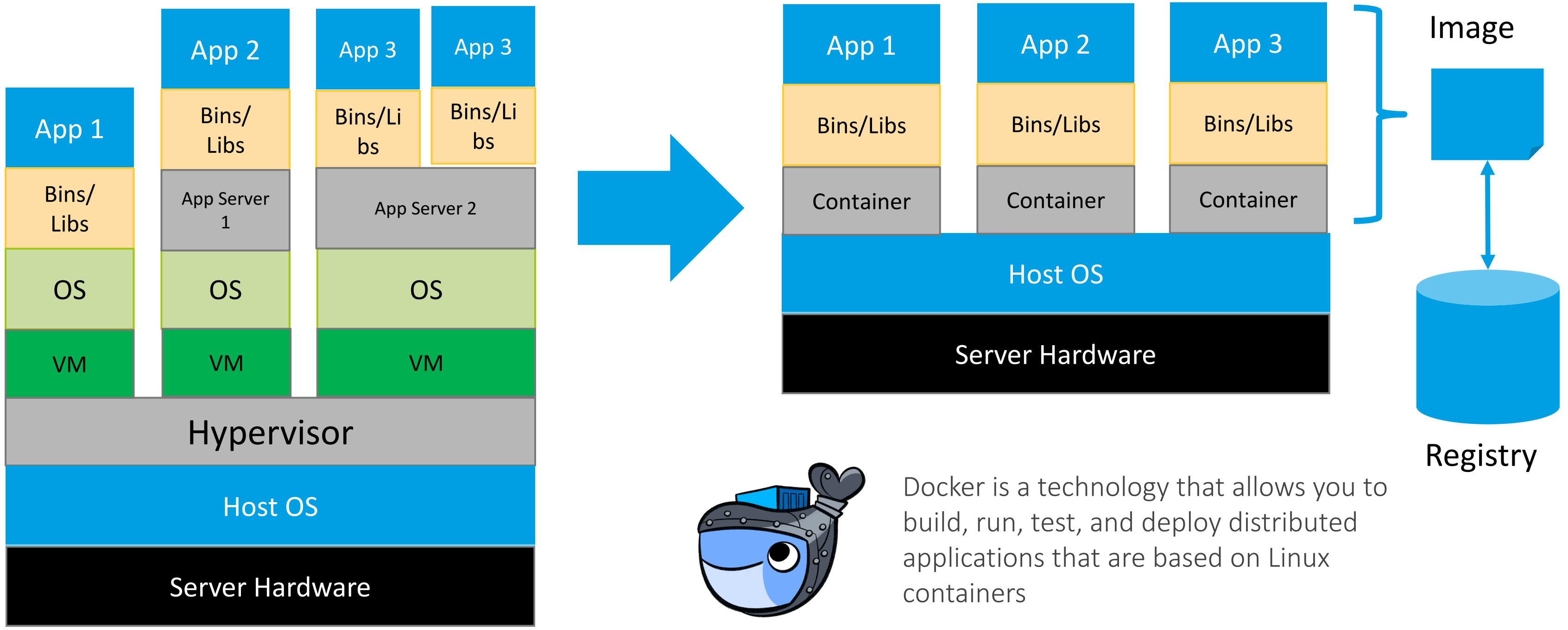
Application Runtime Engine within Code

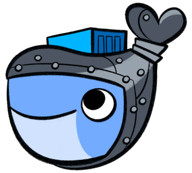


- App server is configured, patched and managed independently, usually not under configuration mngt;
- Configuration drift creep and server failures introduced.
- App Servers create dependencies, that can cripple systems and stop them from being repaired and upgraded.

- A single deployable artifact with the runtime engine embedded eliminates many opportunities for configuration drift;
- Whole artifact is under source control;
- Allows the application team to be able to better reason through how their application is built and deployed.

VMs + App Server Vs Containers



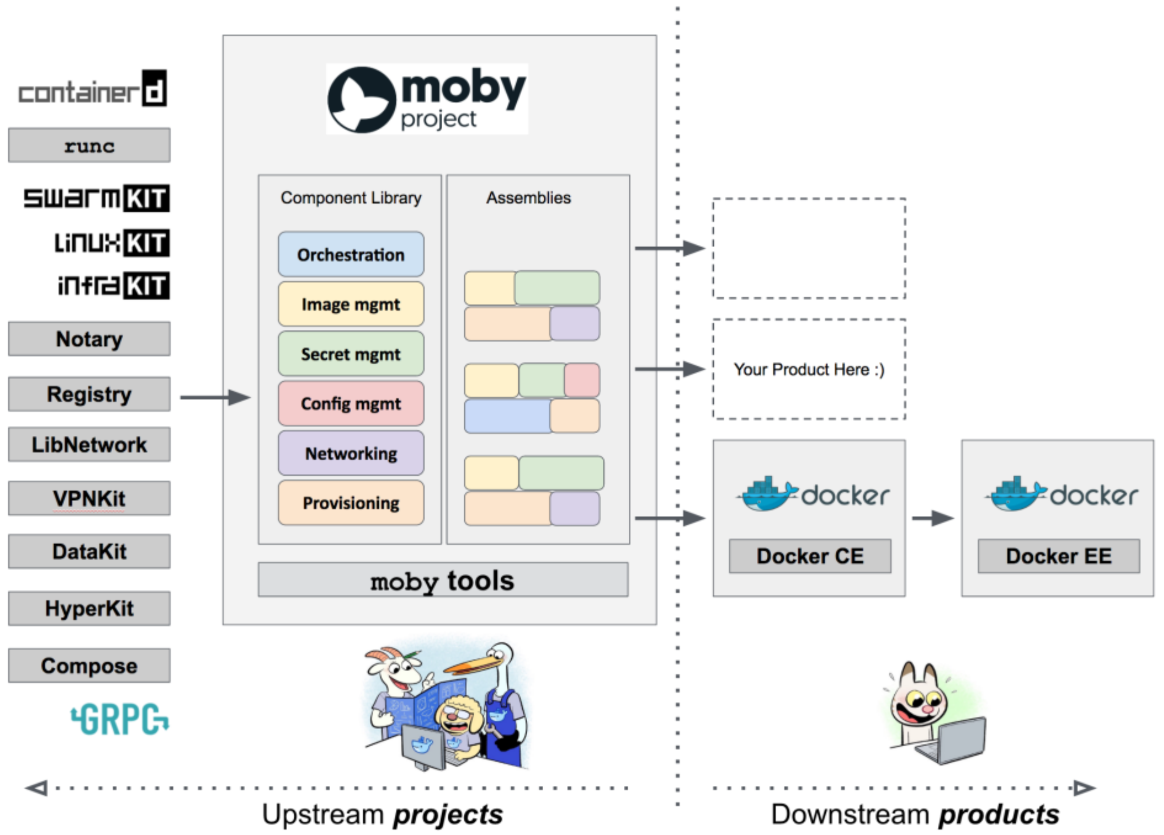


Docker is a technology that allows you to build, run, test, and deploy distributed applications that are based on Linux containers

Containers – the new term ‘CaaS’

- Provide Logical Process Isolation (Business change, Security, Fault Tolerance….);
- Good to isolate COTS products, e.g. dbms, Kafak + Zookeeper;
- Abstract the Infrastructure and Virtualise the OS;
- Deployment Images – Easy to manage what runs:
  - Image is a versioned artefact, includes the app/service;
  - Portable - run same image (and management scripts) in any environment;
  - A new Image can extend an old Image; no duplication of resources;
  - One Image, One Version, One Artefact;
- Automation – deploy app/service automatically, quickly;
  - Schedulable, clusterable (Swarms), auto-scalable;
- Flexible – create clean and reproducible app/service;
- Immutable - make the system read only, remove configuration creep;
- Fast – OS is already running;
- Efficient – memory, resources, CPU better used than VM. No hypervisor required;
- **Ideal for microservices, break down the monolith, deploy separately;**

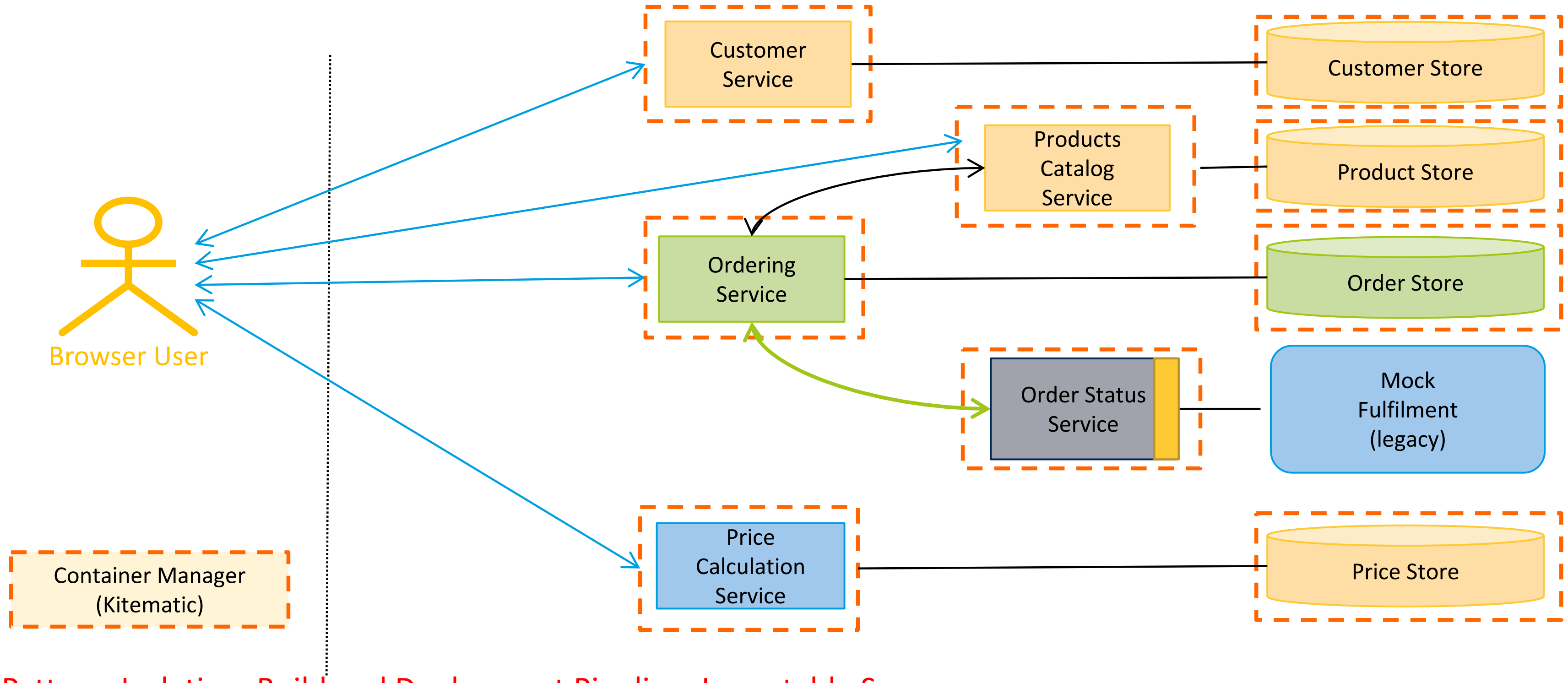
Moby - Roll you own Open Source Container



9 Containers – Isolate Processes

BP

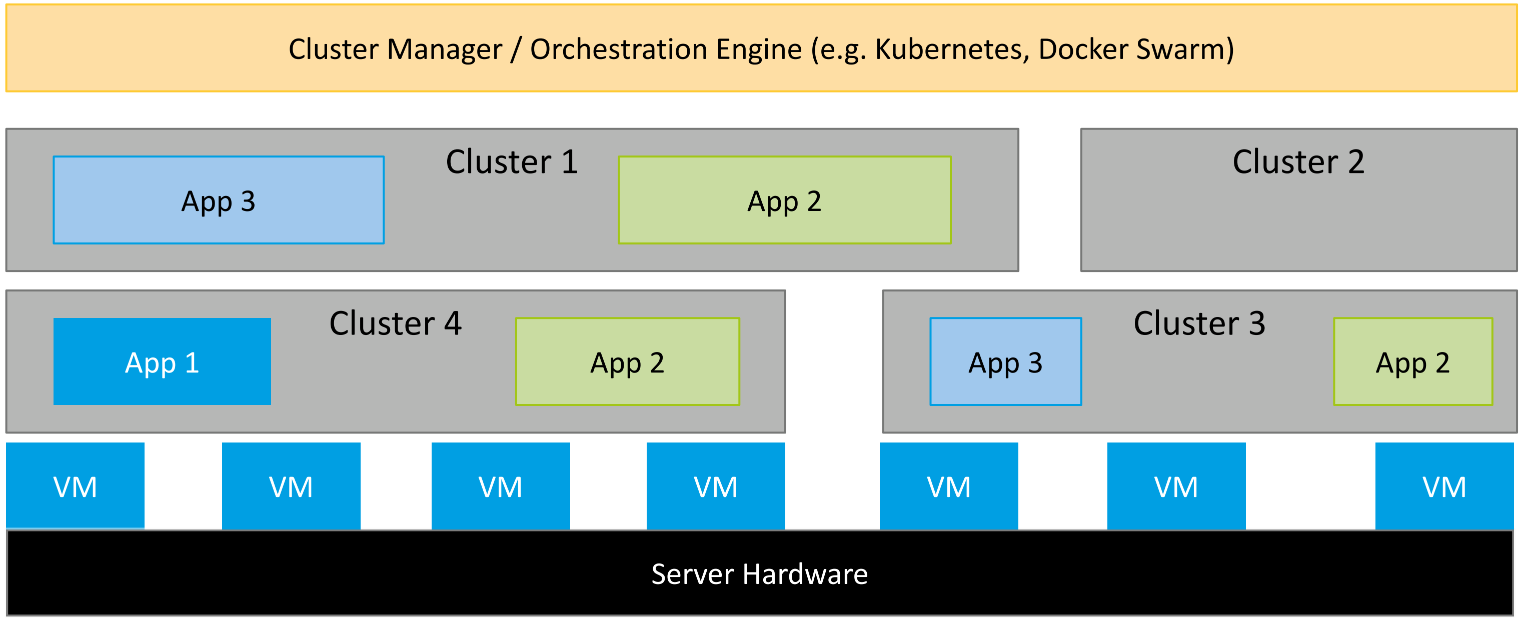
Docker Containers – create and start all services with single ‘Docker-Compose’ script



Pattern: Isolation, Build and Deployment Pipeline, Immutable Servers

Data Center as the Computer

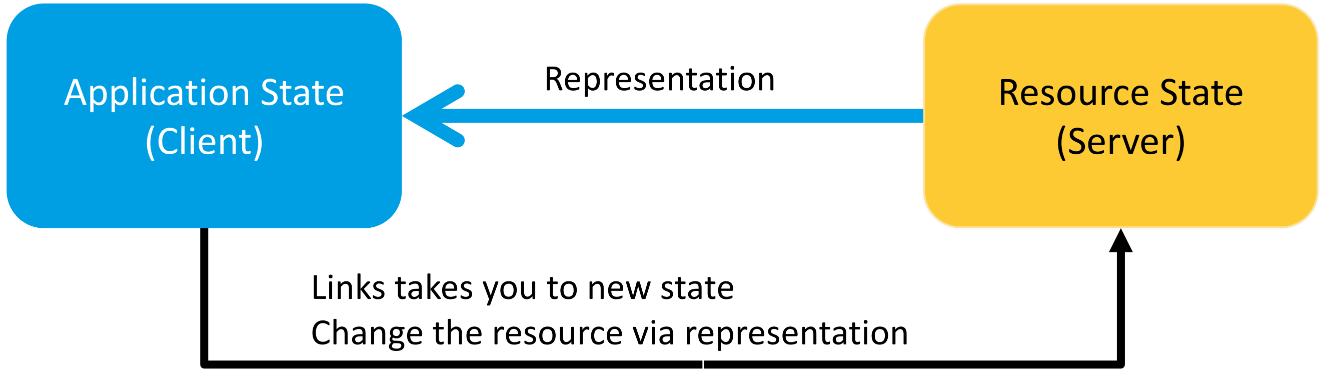
How to manage distributed containerised components.



REST

REST - Representational State Transfer

- Client – Server: Application State - Resource State
- A "representation" of a resource is used to transfer resource state which lives on the server to application state on the client.
- The representation contains hyper-links that can be used to change the state. The new state is then transferred.
- Representation and protocols not specified; any media type: html, image, text, XML, JSON. Commonly used over HTTP.


Application State
(Client)
Representation
Resource State
(Server)
Links takes you to new state
Change the resource via representation

REST Formal Constraints*, Part - 1

- **Client Server:** A uniform interface separates clients from servers.
    - Consumers pull representations

Stateless: The client–server communication is further constrained by no client context being stored on the server between requests. This makes it highly scalable.

- **Cacheable:** As on the World Wide Web, clients can cache responses.
    - Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not.
- **Layered system:** A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.
    - Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches. They may also enforce security policies.
- **Code on demand (optional):** Servers can temporarily extend or customize the functionality of a client by the transfer of executable code. e.g. Javascript, Angular, one page apps.

* From Ray Fielding

REST Formal Constraints, Part - 2

- **Uniform Interface:** The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently. All resources are accessed with a generic interface (e.g. HTTP: GET, POST, PUT, DELETE).

- The four guiding principles of this interface are:
    - **Identification of resources** - Individual resources are identified in requests, for example using URIs in web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client.
    - **Manipulation of resources through these representations.** When a client holds a representation of a resource, it has enough information to modify or delete the resource.
    - **Self-descriptive messages** - Each message includes enough information to describe how to process the message.
    - **HATEOAS -** Hypermedia As The Engine Of Application State.

Uniform Interface: HTTP Methods

Method	CRUD	Description	Idempotent	Safe
GET	R	Retrieve a resource representation; use response links for pagination, sorting, filtering for long lists.	Yes	Yes
POST	C	Submit a new resource. Return link to item with new ID (in Location header).	No	No
PUT	U	Update or Replace an existing resource. Supply ID in Request URL.	Yes	No
DELETE	D	Request that a resource be removed. Return deleted Item	Yes – but Error 2nd time	No
PATCH		Provide instructions to atomically update parts of resource. (not originally REST)	No	No

HTTP Return Codes - Async Scenario

CODE	Type	Description
200	Success	**OK**
201	Success	**Created** - Spec says it must create the resource before returning
202	Success	**Accepted** - but not processed, or busy. Use for Async. Return location link of status resource. This resource may display completion time.
204	Success	**No Content**
303	Redirection	**See Other** – this is not the right resource, see elsewhere. Provide link to correct resource. E.g. we try to get status but process is complete and now provide link to actual resource.
400	Client Error	**Bad Request**
401	Client Error	**Un-authorised**
403	Client Error	**Forbidden**
404	Client Error	**Not Found** – ID not found or invalid
422	Client Error	**Unprocessed Entity** – request syntax is OK but semantically incorrect
500	Server Error	The Web server encountered an unexpected condition that prevented it from fulfilling the request by the client

Async via HTTP

- URL Request - Order Submit:
  - Reply: 202 – not processed yet, but return a link to status resource;
- URL Request - Status:
  - Reply: 200 – Status In Progress, completion time 1 Hour.
  - Keep trying until….
- URL Request - Status:
  - Reply: 303 – No longer n progress, wrong resource. Provide a Link to the now created Order.
- URL Request - Order :
  - Reply: 200 – get the new order.

HATEOAS

HATEOAS distinguishes REST from most other network application architectures. A client interacts with a network application **entirely** through hypermedia provided dynamically by services.

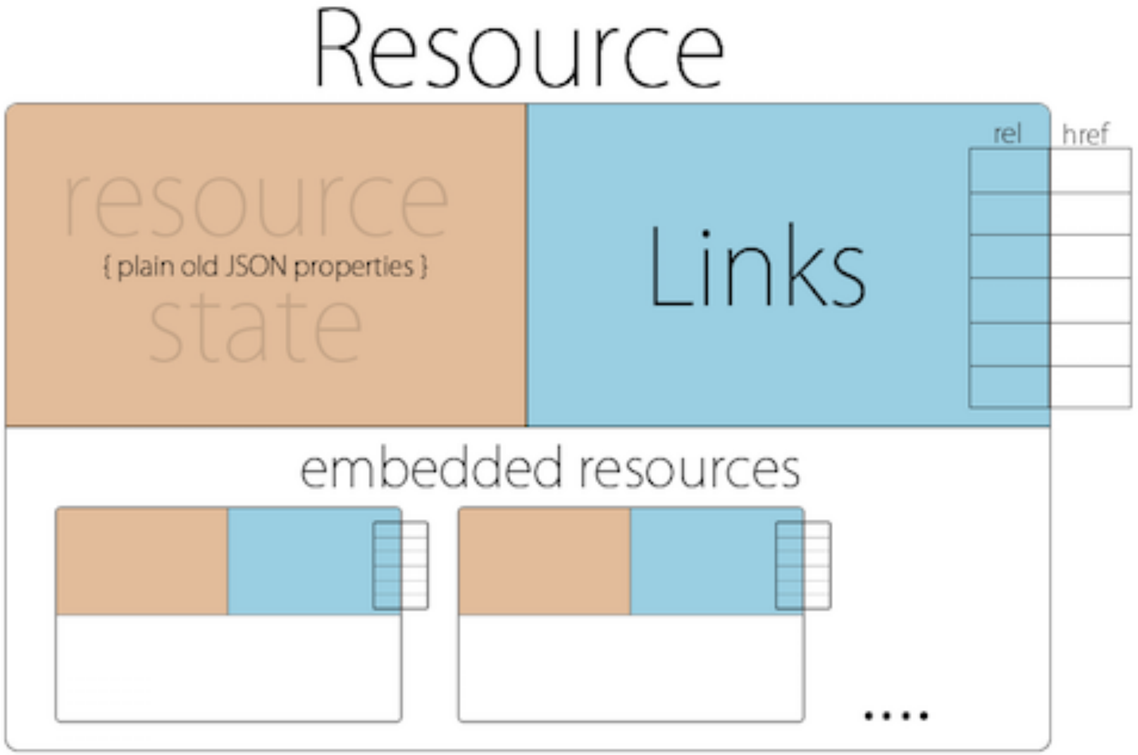
Hypertext simultaneously presents information and controls (links). Information is the affordance through which the user obtains choices, used to select actions.

A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

- By contrast, most SOA, clients and servers interact through a fixed interface shared through documentation or an IDL.

A hypermedia-driven site provides information to navigate the site's interfaces dynamically by including hypermedia links with the responses.

HAL Model


Resource
resource
{ plain old JSON properties }
state
Links
rel
href
embedded resources
....

HATEOAS – Descriptive State via Links

```
HTTP/1.1 200 OK
<?xml version="1.0"?>
<account>
   <account_number>12345</account_number>
   <balance currency="usd">100.00</balance>
   <link rel="deposit" href="/account/12345/deposit" />
   <link rel="withdraw" href="/account/12345/withdraw" />
   <link rel="transfer" href="/account/12345/transfer" />
   <link rel="close" href="/account/12345/close" />
</account>
```

When our account is in credit there are 4 options.

Each message includes enough information to describe how to process the message.

When our account is in debit, only one option.

```
HTTP/1.1 200 OK
<?xml version="1.0"?>
<account>
   <account_number>12345</account_number>
   <balance currency="usd">-25.00</balance>
   <link rel="deposit" href="/account/12345/deposit" />
</account>
```

HAL - Hypertext Application Language

HAL provides a set of conventions for expressing hyperlinks in either JSON or XML. Provides ’real’ REST, with HATEOAS:

- Good for M2M comms;
- Links to Documentation too;
- Described in RFC4627;
- Media type "application/hal+json”;
- Root object MUST be a Resource Object;
- Supports 50+ Languages;
- HAL Browser – way to explore the API, thus helps document the API.

```
GET /orders/523 HTTP/1.1
  Host: example.org
  Accept: application/hal+json

  HTTP/1.1 200 OK
  Content-Type: application/hal+json

    {
   "_links": {
    "self": { "href": "/orders/523" },
    "warehouse": { "href": "/warehouse/56" },
    "invoice": { "href": "/invoices/873" }
       },
   "currency": "USD",
   "status": "shipped",
   "total": 10.20
    }
```

REST must be Hypertext Driven

"I am getting frustrated by the number of people calling any HTTP-based interface a REST API.

Today's example is the SocialSite REST API. That is RPC. It screams RPC… .**What needs to be done to make the REST architectural style clear on the notion that hypertext is a constraint?** In other words, if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API. Period. Is there some broken manual somewhere that needs to be fixed?"

Roy T. Fielding

http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Undisturbed REST

“ Surprisingly, while most APIs claim to be RESTful, they fall short of the requirements and constraints asserted by Dr. Roy Fielding. One of the most commonly missed constraints of REST is the utilization of hypermedia as the engine of application state, or HATEOAS.”

“Undisturbed REST” – Michael Stowe

REST is hard to Understand and Implement

“Most normal developers dropped (or never really understood) HATEOAS in the JSON context anyway, so it didn’t end up mattering too much. ***Except that, unfortunately, it was the most important and innovative aspect of REST!***

As the Wikipedia page says:
“HATEOAS is a constraint of the REST application architecture that distinguishes it from most other network application architectures.”

Without HATEOAS, REST is just a collection of (very) good ideas on client-server layout, but not a revolutionary architecture shift.”

http://intercoolerjs.org/2016/01/18/rescuing-rest.html

Roy Fielding’s Uniform Access Principle

From his blog 2008:

“A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace.”

“A REST API should never have “typed” resources that are significant to the client… The only types that are significant to a client are the current representation’s media type and standardized relation names…” That is to say: the client need only know how to work with ATOM/RSS, images, XLS, generic data encoded as JSON or XML, HTML, media, etc. No further knowledge about implementation types should be required.

The client should know about only one URI, the entry point (bookmark) URI. All other navigation should be discovered while interacting with the API. Navigation information is well conveyed using link entities, like this in XML:

<link rel ="users" href="http://127.0.0.1:8080/users" />   or this in JSON:

{ "rel" : "users", "href" : "http://127.0.0.1:8080/users" }.

DEMO

Example Demo Service Locations – Containers

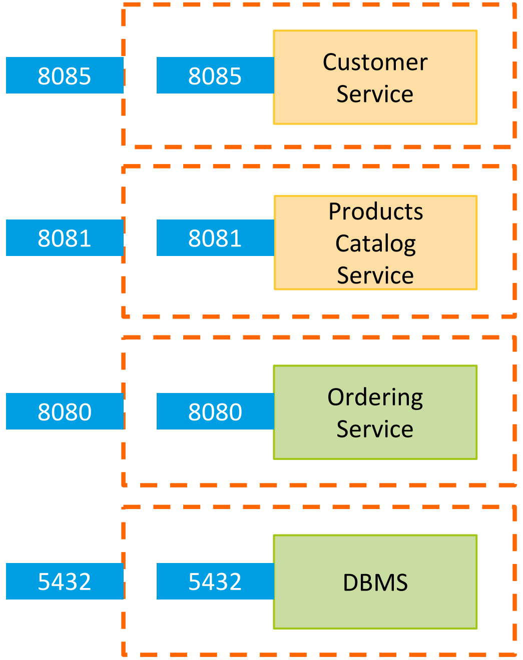
Containers configured on my LocalHost:

Customer Service – Port 8085

Product Service – Port 8081

Order Service – Port 8080

DataBase – Port 5432



Example Scenario - Order Product

- Check Customer Details;
- Look up the Product ID for a ‘CVC’;
- Create a draft Order;
- Add Order Items, ‘CVC’ Product, to Order;
- Submit the Order;
- Track the Status

Examples are JSON APIs.

JSON Customer Services (8085)

CustomerID

Step 1: Get Customer:

GET http://localhost:8085/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a

Add New Customer:

POST http://localhost:8085/v1/customers

Request Body:

No ID provided

```
{
  "name": "No Telus Co",
  "contactName": "Fred Smith",
  "contactEmail": "fred.smith@notelus.com",
  "contactPhone": "922-123-1212"
}
```

Response – Customer Representation with

"id": "g354f8c-c552-4ebe-a82a-e2fc1d1ff78a"

JSON Product Services (8081)

Step 2: Get a List of all Product:

GET http://localhost:8081/v1/products

Step 2: Get the 'Left Wing' Product:

GET http://localhost:8081/v1/products/f354f8c-c443-4ebe-a82a-e2fc1d1ff78a

Add a New Product:

POST http://localhost:8081/v1/products

Request Body:

```
{
  "name": "LBW",
  "description": "Little Bit Of Wire"
}
```

Response - Product with
"id": "2d652189-7a64-4528-aead-732c2b93fea8"

JSON Order Services (8080)

CustomerID

Get Customers Orders:

http://localhost:8080/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders

OrderID

Get Specific Order:

http://localhost:8080/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a

Get Order Items:

http://localhost:8080/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a/items

Get Specific Item:

http://localhost:8080/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a/items/e254f8c-1111-4ebe-a82a-e2fc1d1ff78a

Note: we will use the API Gateway to provide a nicer and less confusing Order URL later…

JSON Order Services (8080)

Step 3: Create a New Order:

POST http://localhost:8080/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders

Request Body:

```
{
  "customerId": "e254f8c-c442-4ebe-a82a-e2fc1d1ff78a",
  "status": "draft"
}
```

Response Order with
"id": ”8de26bcc-2a53-47c7-906d-025d8576af12”

Submit Order:

PUT with: "status": "submit"

JSON Add OrderItem to Order

Step 4: Add an OrderItem:

POST

http://localhost:8080/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a/items

Request Body:

```
{
  "orderId": "f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a",
  "productNumber": "f354f8c-c443-4ebe-a82a-e2fc1d1ff78a",
  "numberPurchased": 5
}
```

Response : OrderItem with
"id": "052fe221-bc72-4aa2-a118-5506e4f4649e"

JSON Order Services (8080)

Step 5: Submit the Order:

PUT http://localhost:8080/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders/8de26bcc-2a53-47c7-906d-025d8576af12

Request Body:

```
{
  "customerId": "e254f8c-c442-4ebe-a82a-e2fc1d1ff78a",
  "status": "submit"
}
```

Issues

These are JSON APIs, RPCs,

but NOT REST !

The only thing a client should need is the root URL

Issues

Request Response is only one approach.

- It is not as reliable as other approaches;
- It doesn't handle scaling as well as other methods;
- It doesn't make sense when a response is not required;
- Requires a consumer to poll for change happening at server side;
- Can't manage high throughput peaks well;

Asynchronous alternative will be discusses in later talks, such as Kafka Steam processing Platform;

- NFRs are essential to choose between these.

Generic REST APIs don’t cut it

“REST APIs are excellent at handling requests in a generic way, establishing a set of rules that allow a large number of known and unknown developers to easily consume the services that the API offers……

The potential shortcomings surface because this model assumes that a key goal of these APIs is to serve a large number of known and unknown developers. **The more I talk to people about APIs, however, the clearer it is that public APIs are waning in popularity and business opportunity and that the internal use case is the wave of the future.** There are books, articles and case studies cropping up almost daily supporting this view. And while my company, Netflix, may be an outlier because of the scale in which we operate, I believe that we are an interesting model of how things are evolving.”

Daniel Jacobson, director of engineering for the Netflix API.

Future Episodes and Prequels

Putting REST to Sleep

Many problems with XML, then SOAP, and now REST…………..

The number of different ways to do ‘REST’, and alternatives, is growing and not stopping:

- GraphQL
- HAL *
- JSON API
- JsonPatch
- Collection + JSON *
- AVRO
- gRPC and Protocol Buffers
- Thrift
- Atom
- Binary Formats
- IOT Messaging, MQTT

- JSON-LD
- Siren *
- Uber
- Mason
- Yahapi
- CPHL
- RESTdesc
- PubSubHubbub
- OData
- Etc

30 ways to represent a date in JSON

* Real HATEOAS REST

3 Main Competing HTTP API Specs

Need for a common format that describes the service model of an API, 3 Specs:

1. Open API Specification (OAS) by Open API Initiative (OAI) – based on Swagger. Supported by 3Scale, Apigee, Google, IBM, Microsoft, Paypal, etc…
   - Seems most open and winning at moment;
2. RAML with MuleSoft as a main contributor;
3. API Blueprint backed by Apiary (bought by Oracle).

Next Episode

Infrastructure Services to support Business Services:

- **Configuration Server** – set up the containers;
- **Gateway Server** – ‘friendly’ names for services, one point of interaction;
- **Registry Server** – find the services and instances;
- Demo – have example service available in GIT for all;

Later Episodes Include:

- Failure and Circuit Breakers, Bulkheads etc;
- Client Side Load balancing;
- EDA, Asynchronous, getting the order to Fulfilment;
- Legacy system integration and ESBs;
- And more….

Rest Vs SOAP

REST	SOAP
An architecture style using standard HTTP to provide a simple manner of connectivity. A standardized or enforced contract does not exist.	A protocol utilizing a service's interface to expose business logic in a strictly enforced WSDL contract.
Exposes named resources, building upon POST, GET, PUT, DELETE, and PATCH operations.	Exposes functions and processes using XML-based protocol.
Security is handled by the underlying infrastructure.	Supports WS-Security, which provides the ability to protect both data from a privacy and integrity perspective.
Caching can be utilized to yield performance improvements.	Caching is not an option for SOAP method calls.
Limited to HTTP and cannot perform two-phase commits across distributed systems.	Supports WS-Atomic Transaction and allows for the ability to perform two-phase commits.
Allows use of multiple data formats (JSON, XML, text, user-defined).	Supports only XML format.
Smaller learning curve based upon the simplicity of the architecture style.	Learning curve is higher, but is justified by the advantages of using the standardized protocol.