Enterprise Java for the Cloud

Rajiv Mordani
Josh Dorr
Dhiraj Mutreja

September, 2016

Safe Harbor Statement

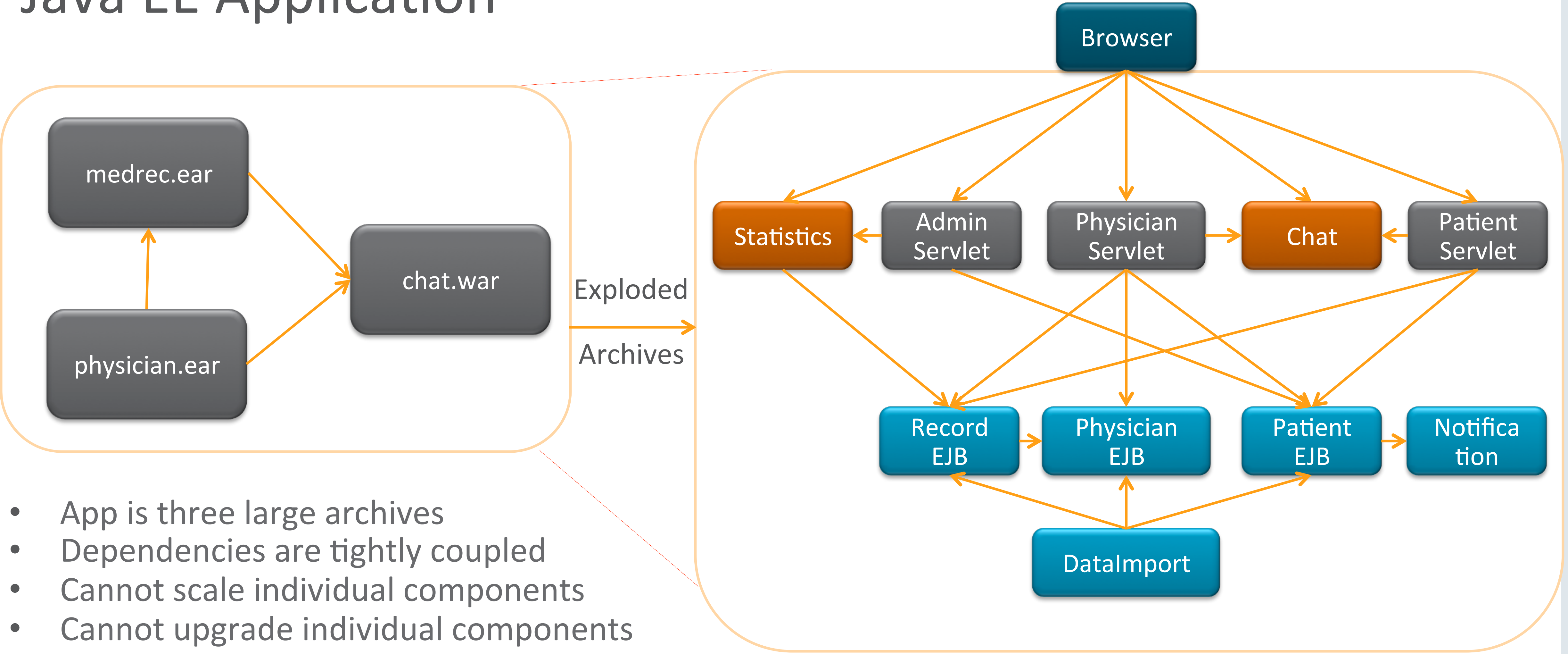
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Agenda

1. Overview
2. Programming Model
3. State
4. Configuration
5. Multi-Tenancy
6. Security
7. Packaging and Orchestration
8. Summary

Application Development is Changing

Java EE Application



- App is three large archives
- Dependencies are tightly coupled
- Cannot scale individual components
- Cannot upgrade individual components

Rapid Changes Over Past Few Years

Driven by increasing business needs

Microservices

Apps divided into many small pieces

Distributed Computing

Many data centers, AZs, regions, etc.

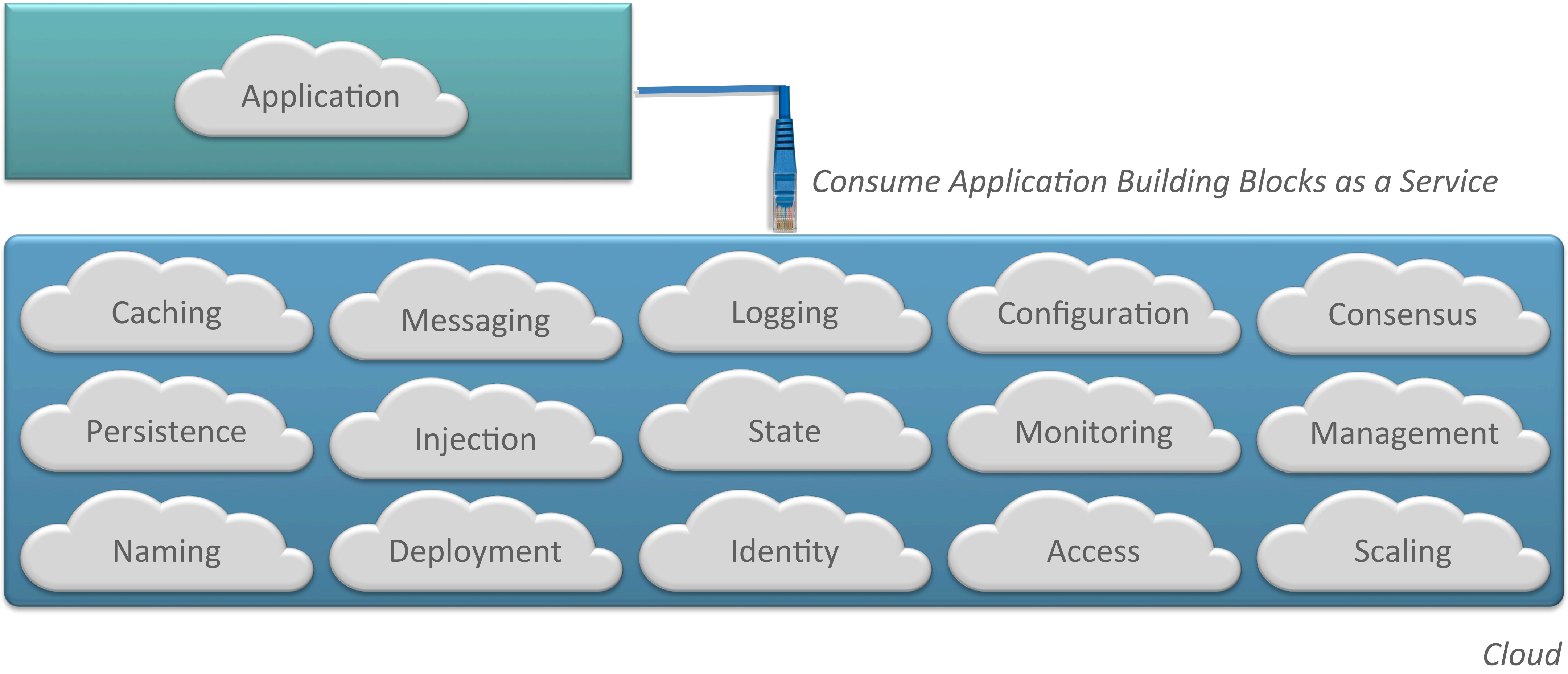
New Technology Trends

Docker, Cloud, DevOps, etc.

The Twelve Factors

1. Codebase
   – One codebase tracked in revision control, many deploys
2. Dependencies
   – Explicitly declare and isolate dependencies
3. Configuration
   – Store configuration in the environment
4. Backing services
   – Treat backing services as attached resources
5. Build, release, run
   – Strictly separate build and run stages
6. Processes
   – Execute the app as one or more stateless processes
7. Port binding
   – Export services via port binding
8. Concurrency
   – Scale out via the process model
9. Disposability
   – Maximize robustness with fast startup and graceful shutdown
10. Dev/prod parity
    – Keep development, staging, and production as similar as possible
11. Logs
    – Treat logs as event streams
12. Admin processes
    – Run admin/management tasks as one-off processes

Cloud Has Become the Platform



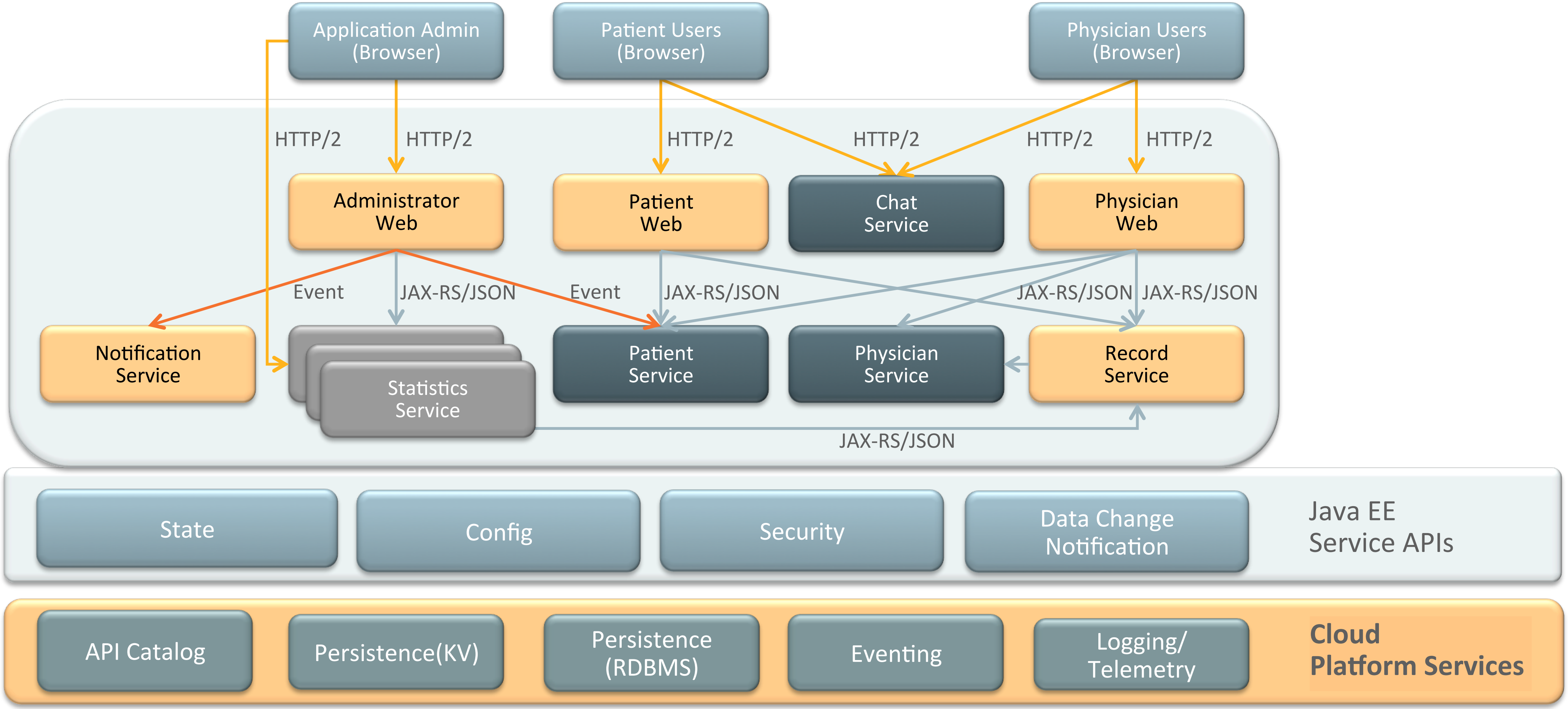
It’s Confusing!

Too many choices....
Which components?
Overall architecture?
Standards?
Vendor commitment?

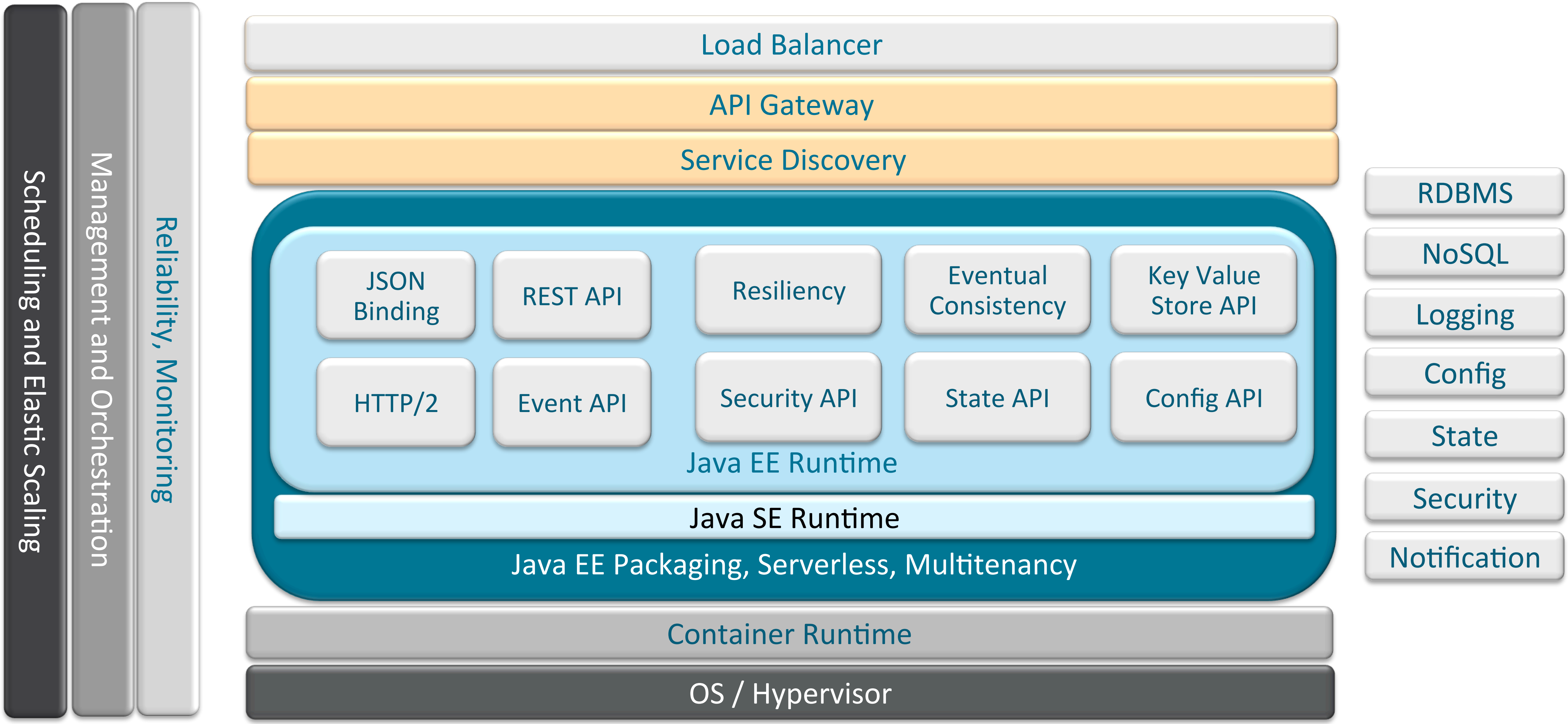
Enter Java EE 9

- Java EE has provided the standard infrastructure for building enterprise applications
- With the shift to cloud the type of applications and the requirements for these applications have changed
- Applications are becoming more Microservice oriented
- Java EE 9 provides an opportunity to create a standard for applications deployed to the cloud to simplify development and maximize portability

Java EE Application as Independent Services



Proposed Platform Architecture



Scheduling and Elastic Scaling

Management and Orchestration

Reliability, Monitoring

Load Balancer

API Gateway

Service Discovery

JSON Binding

REST API

Resiliency

Eventual Consistency

Key Value Store API

HTTP/2

Event API

Security API

State API

Config API

Java EE Runtime

Java SE Runtime

Java EE Packaging, Serverless, Multitenancy

Container Runtime

OS / Hypervisor

RDBMS

NoSQL

Logging

Config

State

Security

Notification

Technical Focus Areas

Programming Model

- Extend for reactive programming
- Unified event model
- Event messaging API
- JAX-RS, HTTP/2, Lambda, JSON-B, ...

Packaging

- Package applications, runtimes into services
- Standalone immutable executable binary
- Multi-artifact archives

Key Value/Doc Store

- Persistence and query interface for key value and document DB

Eventual Consistency

- Automatically event out changes to observed data structures

Serverless

- New spec – interfaces, packaging format, manifest
- Ephemeral instantiation

Configuration

- Externalize configuration
- Unified API for accessing configuration

Multitenancy

- Increased density
- Tenant-aware routing and deployment

State

- API to store externalized state

Resiliency

- Extension to support client-side circuit breakers
- Resilient commands
- Standardize on client-side format for reporting health

Security

- Secret management
- OAuth
- OpenID

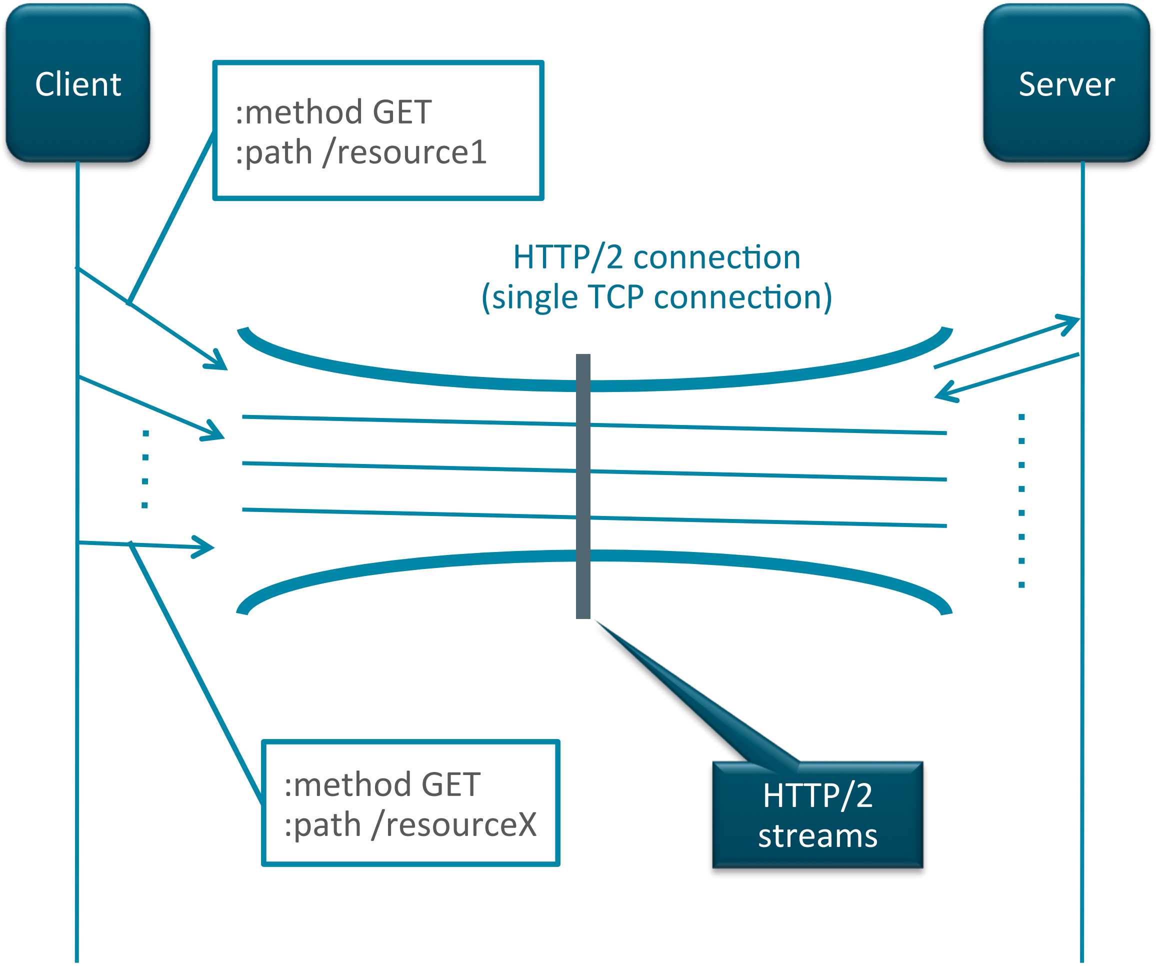
Programming Model Trends

- Programming model needs to be enhanced to support
  - Distributed smaller services
- Interact via REST / JSON making remote calls asynchronously
  - Results in a lot of remote calls
  - Need to be resilient to latency and other network failures
  - Need to support asynchronous calls
- Need to support eventual consistency for data persistence as well as across service calls
- Reactive style programming
  - Event based asynchronous application programming model
- Built in resiliency in the runtime utilizing health check, circuit breaker and bulkhead patterns
- Support security standards like OAuth, Open ID Connect that are more relevant for cloud native applications

HTTP/2, REST, JSON

HTTP/2

- Same semantics as HTTP/1.1
- Binary protocol
- Multiplexed communication
  - Single TCP connection to single origin, shared for consequent/parallel requests
- Compressed headers
  - HTTP/2 introduces HPACK (compression algorithm)
- Server Push
  - Server can push (cacheable) content to the client before client asks



Java EE 9 Proposal

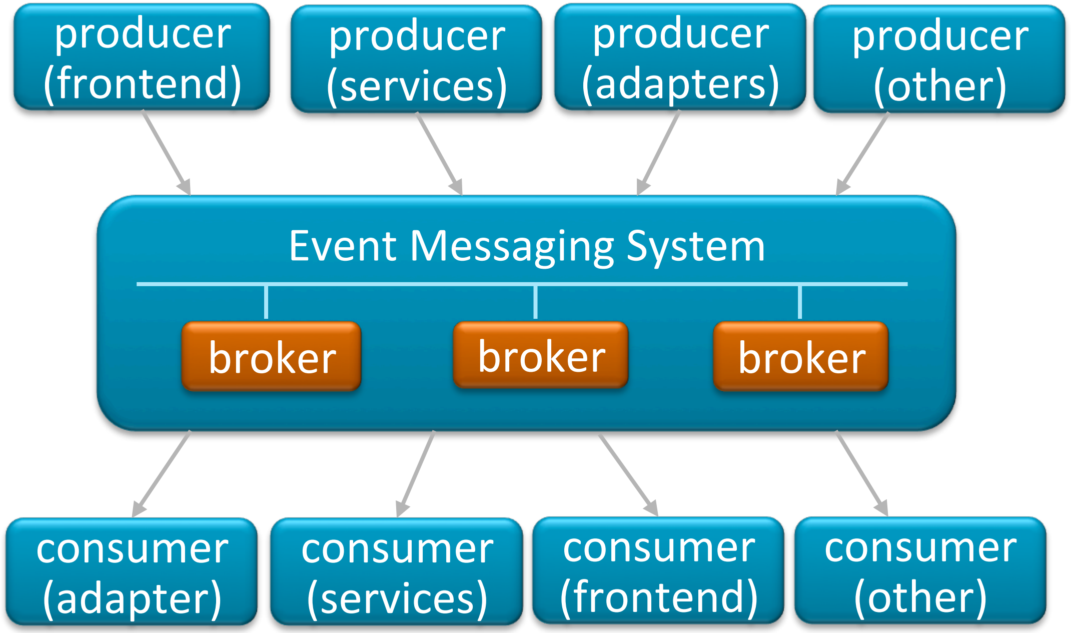
- Servlet already supports asynchronous programming (introduced in Servlet 3.0)
- Servlet 4.0 adding some support for HTTP/2
- Considerations for Java EE 9
  - Provide asynchronous, non-blocking HTTP/2 programming API which can fully leverage features like server push, stream prioritization, flow control etc.
  - Provide unified reactive HTTP programming API which can support HTTP/1.x, HTTP/2, WebSocket, SSE, etc.

- JAX-RS provides REST server and client side support
- Proposed to be enhanced to support
  - Non-blocking IO
  - Security standards
  - Server Sent Events
- Client enhancements
  - Circuit breakers
  - Reactive client APIs
- First class support for JSON in the platform for processing and binding
  - JSON-P
  - JSON-B

Eventing

Use Cases for Eventing API for Cloud

- Handle very large quantities of messages driven by events, throughput is the dominant concern
- Use Cases:
  - Website activity tracking
  - Metrics, Log data aggregation
  - Gaming data feed
  - Etc.
- New Java EE API is needed for eventing in cloud



Comparisons of Eventing Systems Used in Cloud

	Kafka	Amazon Kinesis	Azure Event Hub
HA and Fault Tolerance	Replication between cluster nodes. Support zero downtime upgrades	Synchronously replicates your streaming data across three facilities in an AWS Region	Geo-Redundant Storage Availability Sets to achieve HA and Fault Tolerance
Scalability	Increase partition count per topic OR number of downstream consumer threads to increase throughput.	Data records are segregated into different shards, throughput can be dynamically adjusted via re-sharding	Scalable depending on the number of throughput units
Delivery semantics	At least Once	At least Once	At least Once
Throttling	?	Yes	Yes
Transaction	No	No	No
On-premises Support	Yes	No (cloud-based service)	No（managed service）
Security	?	Yes (HTTPS for all operations)	Yes (SAS tokens)
Retention	Unlimited	Up to number of days	Up to number of days

Existing Java EE Technologies for Messaging, Eventing

- JMS
  - Designed for enterprise messaging
  - Although provide varied QoS, must meet highest requirements as a Java EE conformant JMS provider
- CDI Event
  - Designed for within application same JVM
  - Producer and consumer rendezvous by Object type and qualifiers
- Java API for WebSocket
  - Designed for integrating WebSockets into applications
- JAX-RS
  - Designed for creating REST web services

Event API Proposal for Java EE 9

- A Simple Event API
  - Producer and Consumer as top level injectable resources, for example

```
@Inject EventPublisher(“mytopic”) publisher;
@Inject EventConsumer(“mytopic”) consumer
```

  - declarative message listeners - any POJO as event listener, for example

```
@EventListener(“mytopic”)
public void onMyEvent(MyEvent event) {  //do something }
```

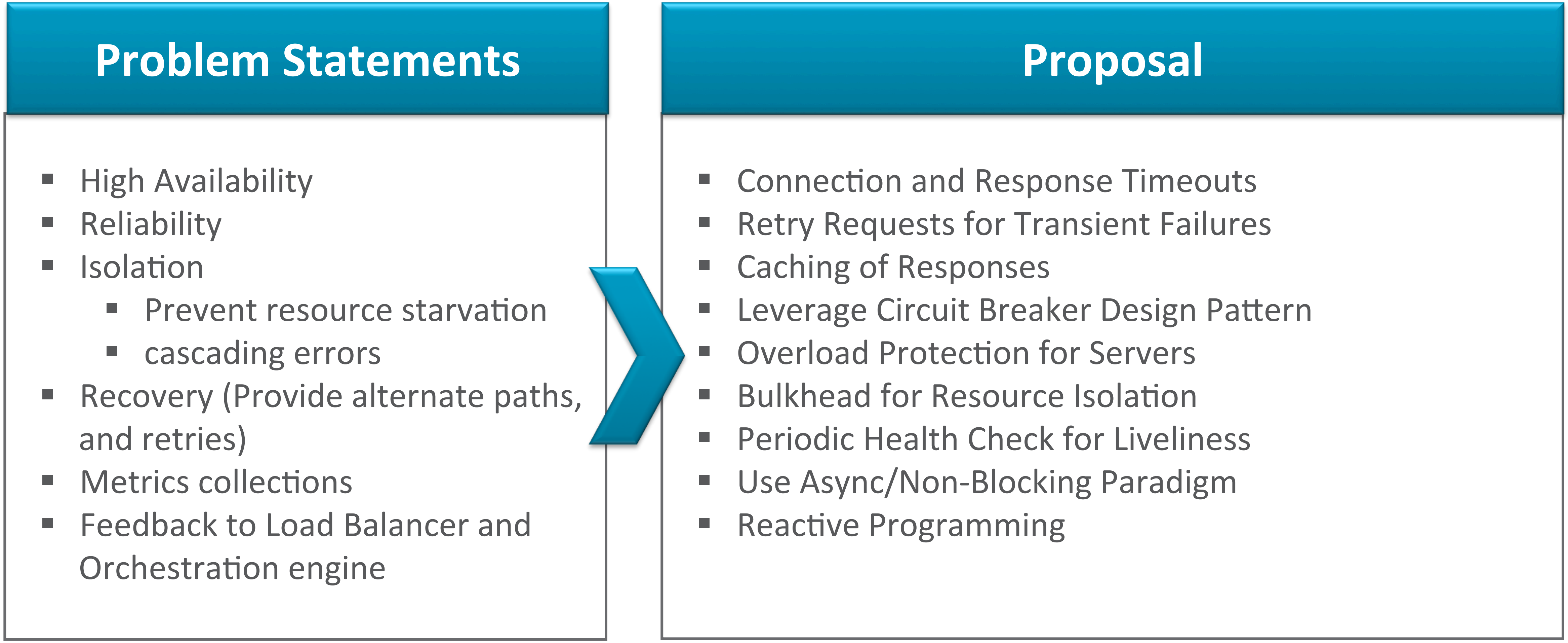
- Reactive style for async eventing using Java 9 Flow, for example

```
public java.util.concurrent.Flow.Publisher<Status> sendAsync(List<EventMessage> events)
```

- Able to plugin different cloud messaging systems in Java EE for eventing

Resiliency

Proposal for Resiliency



Problem Statements

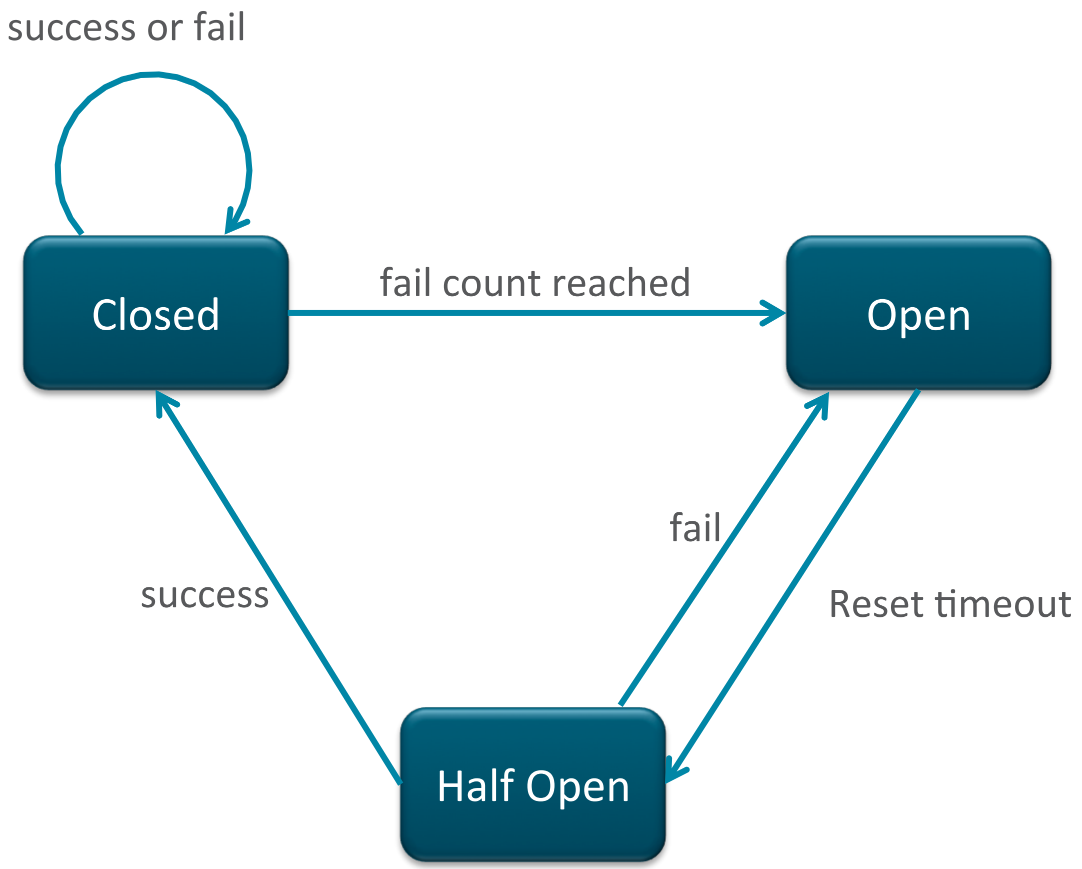
- High Availability
- Reliability
- Isolation
  - Prevent resource starvation
  - cascading errors
- Recovery (Provide alternate paths, and retries)
- Metrics collections
- Feedback to Load Balancer and Orchestration engine

Proposal

- Connection and Response Timeouts
- Retry Requests for Transient Failures
- Caching of Responses
- Leverage Circuit Breaker Design Pattern
- Overload Protection for Servers
- Bulkhead for Resource Isolation
- Periodic Health Check for Liveliness
- Use Async/Non-Blocking Paradigm
- Reactive Programming

Circuit Breakers

- Generic way how to deal with failures in remote service invocation process
- Protecting system resources by monitoring calls to remote service
  - If some certain number of failures is reached, no further calls are made and the error is returned immediately
  - When a circuit is “open”, error supplier might provide replacement answer
    - Could be completely different (empty) answer, or cached value from previous successful invocation
- Several HTTP properties which could trigger failure
  - TCP level: connect error, connect timeout, connection timeout, ..
  - HTTP level: status code, …



Comparison of Selected Circuit Breaker Implementations

	Hystrix	Failsafe	Akka CircuitBreaker
Creation	new HystrixCommand() or HystrixObservableCommand() Call to be protected to be put in run() method.	new CircuitBreaker() Call to be protected as argument Failsafe.with(circuitBreaker).run() etc.	new akka.pattern.CircuitBreaker() Call to be protected as argument to CircuitBreaker call
sync/async support	Yes	Yes	Yes
Reactive model support	Yes, through API that returns Observable	No direct API support. But can be used with reactive framework such as rx.Observable	Yes
Configuration	Many configuration properties supported, through Netflix Archaius.	Many configuration properties supported through CircuitBreaker and FailSafe APIs.	Only a few configuration properties supported, through CircuitBreaker constructor
Threadpool for execution	Managed thread pools internally	Caller to provide thread pool	Caller to provide Scheduler

Resiliency – Proposal for Java EE 9

- Annotation for resiliency policies
- Real-time monitoring and dynamic configurations
- Support for reactive programming
- Request/Response caching
- Graphical Dashboard showing service dependencies and their runtime stats

```
public class BookService {
 ...
 @RetryPolicy(delayPeriod=10,
unit=SECONDS, numRetries=1)

@CircuitBreaker(fallbackMethod=“getBook
sByAuthorFallBack”)
 @BulkHeadPolicy(threadCount=5)
 public Collection<Book>
getBooksByAuthor(String authorName) {
  ...
 }
 public Collection<Book>
getBooksByAuthorsFallBack() {...}
        }
```

Reactive Programming

Existing Standard for Reactive Programming

- Reactive Streams provides “a standard for asynchronous stream processing with non-blocking *back-pressure*”
- Core concern is handling *back-pressure*
- Several frameworks, tools, libraries are emerging to develop reactive applications
  - RxJava
  - Akka
  - Reactor
  - Spring Framework
- Implementations can interoperate as they use a standard API
- Java SE 9 introduces Reactive Streams interfaces through Flow APIs

Popular Implementations and Comparison of Reactive Streams

	RxJava	Reactor	Akka Stream	Java SE 9 Flow
Architecture	Event driven	Event driven	Actor based	Event driven
Back-pressure	Yes	Yes	Yes	Yes
Concurrency	Default single threaded	Default single threaded Schedulers.parallel()	Default runs parallel	Multi threaded
Clustering		No	Yes	No

Publisher	Single	Mono (0 or 1)
		Flux(N)
			Source.single(0 or 1)
			Source.from(N)
				SubmissionPublisher (1 by default)

DataFlow	Synchronous Asynchronous	Synchronous Asynchronous	Synchronous Asynchronous	Asynchronous(it provides only SubmissionPublisher which is async by default)

Proposal for Standardizing Reactive

- Reactive Streams does not provide comprehensive set of APIs for cloud native application development
- In order to provide a comprehensive set of APIs the proposal is to standardize
  - Publisher / Subscriber APIs
  - Tie Publisher to existing data structures (e.g. Iterable, Arrays, etc.)
  - Provide operators to process stream of events
  - Add high level APIs to handle back-pressure
  - Support good Error handling mechanism
  - Interoperability of the stream of events
- Build on JDK 9 Flow APIs
- Allow plugging in of different implementations

State

Java EE support for NoSQL

Proposal for Managing NoSQL Databases

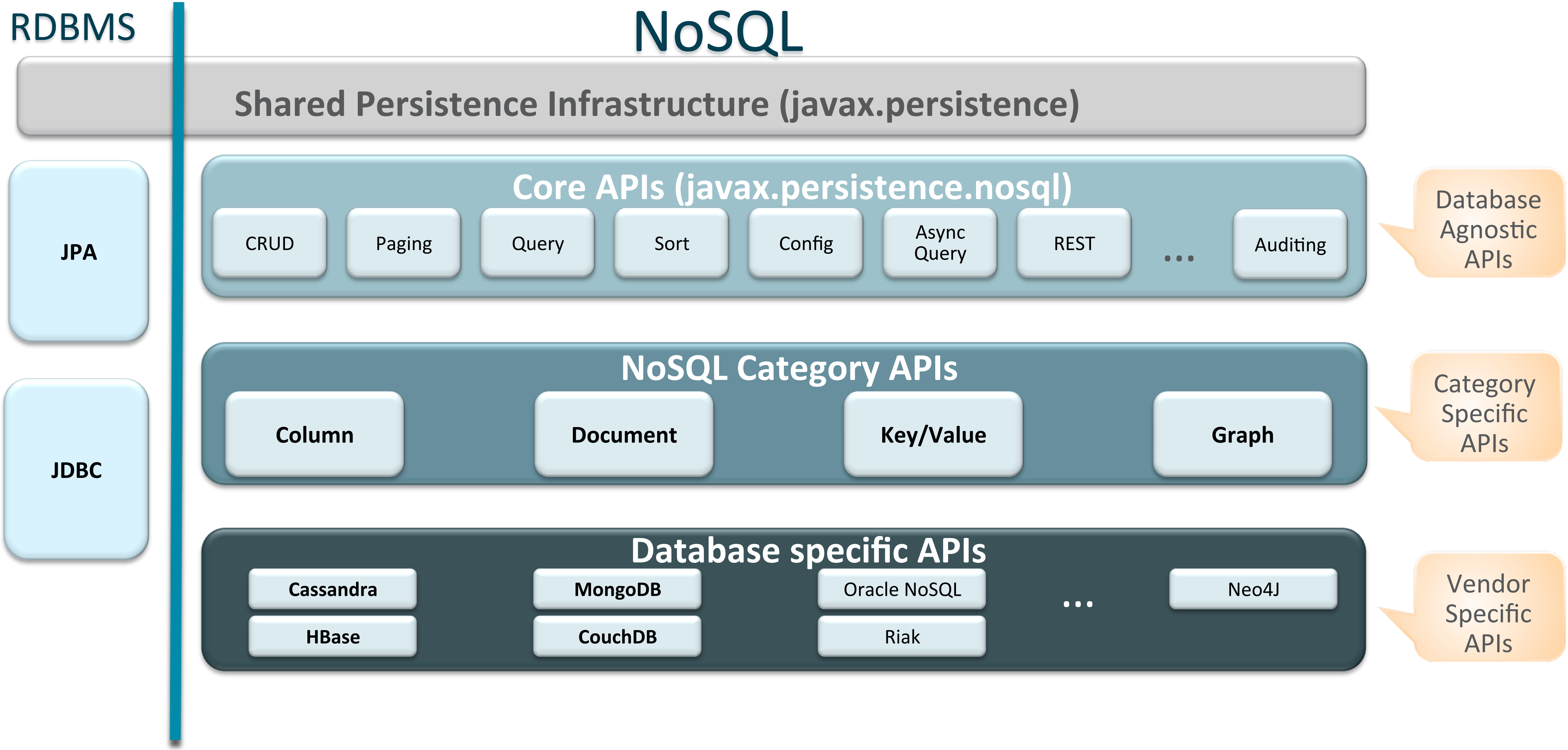
Problem Statement

- Java EE Standards are focused on RDBMS.
  - JPA was not designed with NoSQL in mind
- A single set of APIs or annotations isn't adequate for all database types
- JPA over NoSQL implies inconsistent use of Annotations.
- Diverse categories of NoSQL providers

Proposal

- Provide a consistent programming model
- Provide common abstractions for CRUD operations and additional support for the most common flavors of NoSQL databases
- Allow for direct access to Vendor Specific Functionality
- Simplified Querying:
  - Query inferences based on method names
  - Vendor specific query annotations
- Annotations grouped by category of functionality

JavaOne™ ORACLE



RDBMS

NoSQL

Shared Persistence Infrastructure (javax.persistence)

JPA

JDBC

Core APIs (javax.persistence.nosql)

CRUD

Paging

Query

Sort

Config

Async Query

REST

…

Auditing

Database Agnostic APIs

NoSQL Category APIs

Column

Document

Key/Value

Graph

Category Specific APIs

Database specific APIs

Cassandra

HBase

MongoDB

CouchDB

Oracle NoSQL

Riak

…

Neo4J

Vendor Specific APIs

Basic NoSQL CRUD APIs

```java
package javax.persistence.nosql;

import java.util.Iterator;

/**
 * Basic CRUD operations on a NoSQL store.
 *
 * @param <K> Primary Key for the Object
 * @param <V> Store Data
 */

public interface CRUDStore<K extends ID, V>
        extends BaseStore<K, V> {

  /**
   * Find all items in the store.
   *
   * @return the iterator for all items in the store.
   */
  Iterator<V> findAll();

  /**
   * Find an item based on a specific key or index.
   *
   * @return the iterator for all items in the store.
   */
  V find(K key);


/**
   * Saves a given item. Returns the current value of the object.
   * This may not reflect the "actual" value of the item in an
   * eventually consistent system.
   *
   * @param value
   * @return the current entity
   * @throws IllegalArgumentException if the item is null
   */
  V persist(V value);

  /**
   * Deletes am item with the specific key.
   *
   * @param key
   */
  void remove(K key);

  /**
   * Deletes am item which matches the specific value.
   *
   * @param value
   */
  void remove(V value);

}
```

Example of Category and Provider Specific APIs

Category Specific (e.g.Key/Value):

```java
/**
 * Basic Key/Value Store. The Key is composed of a set of one
or more
 * strings.
 */
public interface KVStore<V> extends CRUDStore<ID<String>, V> {
  /**
   * Store the item based on its key.
   */
  void persist(ID<String>key, V value);
}

/**
 * Store with methods specific to key/value caches.
 */
public interface KVCacheStore<V> extends KVStore<V> {
  /**
   * Persist with an expiration time.
   */
  void persist(ID<String> key, V value, long expires);

  /**
   * Set or change the expiration time on an object.
   */
  void expire(ID<String>key, long expires);
}
```

Provider Specific:

```java
public interface VoldemortStore<V> extends KVStore<V> {

  void get(ID<String> key, Transform<V> transform );
  void get(ID<String> key, Versioned<V> value );
  void get(ID<String> key, Versioned<V> value,
           Transform<V> transform );

  void store(ID<String> key, Transform<V> transform);
  void store(ID<String> key, Versioned<V> value);
  void store(ID<String> key, Versioned<V> value,
          Transform<V> transform);

  void delete(ID<String> key, Versioned<V> versioned);
}
```

NoSQL APIs in Action

Application Store Definition

```
public interface UserStore
   extends MongoStore<String,User> {

    /*
     * This query is inferred (generated) by its name.
     * The query looks for all documents where the
     * field “name” starts with “regex”
     */
    List<User> findByNameStartingWith(
           String regexp);

    /*
     * This query is inferred (generated) by its name.
     * The query looks for all documents where the
     * field “lastname” ends with with “regex”
     */
    List<User> findByLastnameEndingWith(
           String regexp);

    /*
     * This query is defined by the annotation
     */
    @Query("{ 'age' : { $gt: ?0, $lt: ?1 } }")
    List<User> findUsersByAgeBetween(
           int ageGT, int ageLT);
}
```

Application Store Usage

```
public class UserStoreIntegrationTest {

  @Inject
  private UserStore userStore;


  public void insertUser() {
    final User user = new User();
    user.setName("Jon");
    userStore.persist(user);
    List<User> users =
       userStore.findUsersByAgeBetween(5,10);
  }
}
```

Proposal for State Management API

Problem Statements

- No standard API to access state
  - JDBC and JPA are not enough
  - Non-relational data sources are very popular in the cloud
- Most existing APIs are blocking
  - Less than ideal for microservices
- Transient and persistent state are managed differently
- State management is too tightly coupled with persistence
  - Limits scalability

Proposal

- Define higher-level State Management API that supports:
  - Primary key-based reads and writes
  - Queries and aggregations
  - Data events and in-place processing
- Provide blocking (synchronous), as well as non-blocking (asynchronous and reactive) APIs
- Allow implementations for different kinds of data tiers
  - E.g. In-Memory Grid, Cache, RDBMS, K/V Stores
- Manage transient and persistent state the same way
  - Policy defined per entity type
- Decouple state management and persistence aspects
- Provide in-memory RI that can be used for dev and testing

Additional pattern for State Management

- **C**ommand
- **Q**uery
- **R**esponsibility
- **S**egregation

CRUD vs. CQRS

UserService

```
public interface UserService {

  void addUser(User user);
  void makeUserPreferred(UserId id);
  User getUser(UserId id);
  Set<User> getPreferredUsers();
  void removeUser(UserId id);

}
```

Same service performs read and write operations

UserReadService

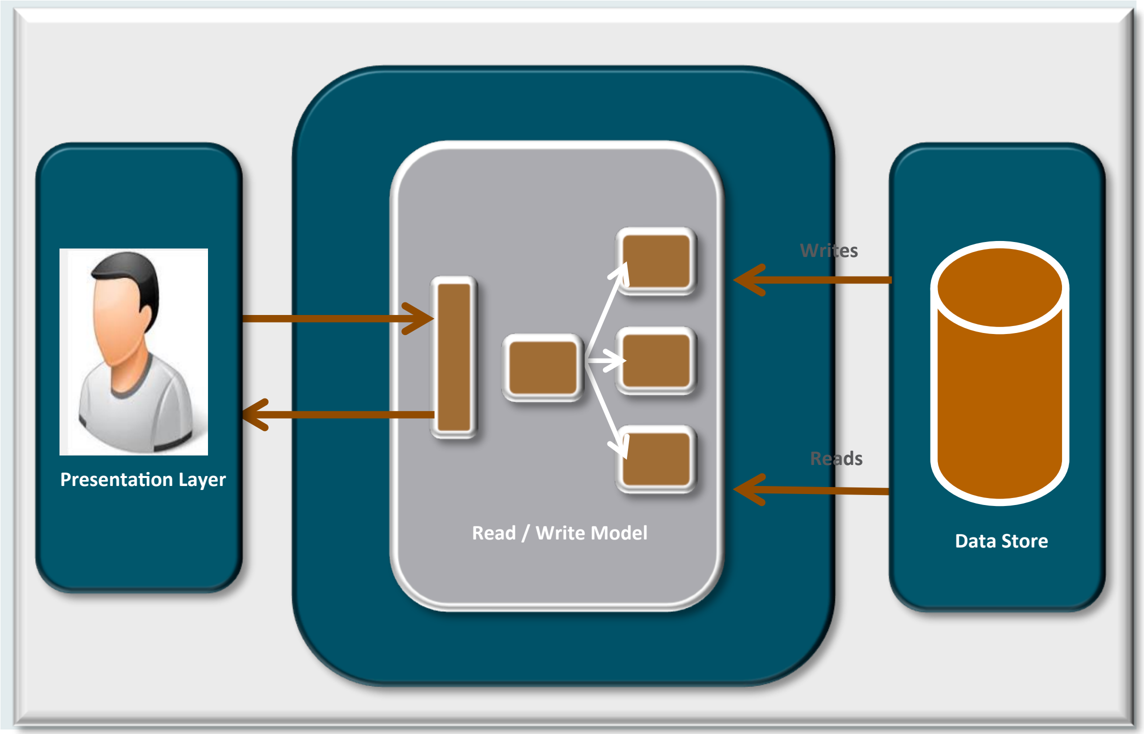
```
public interface UserReadService {
   User getUser(UserId id);
   Set<User> getPreferredUsers();
}
```

UserWriteService

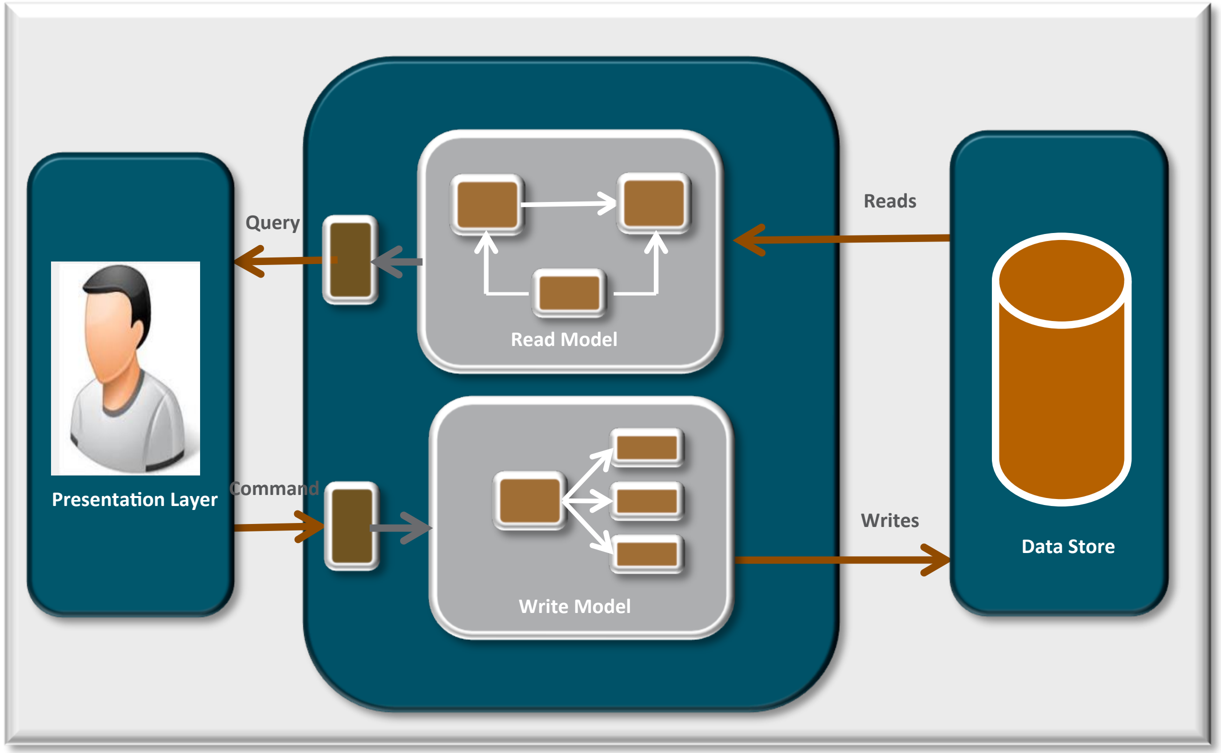
```
public interface UserWriteService {
  void addUser(User user);
  void makeUserPreferred(UserId id);
  void removeUser(UserId id);
}
```

Different services perform read and write operations

CRUD vs. CQRS



Read and Write operations performed on the same model



Read and Write operations are segregated

CQRS pattern overview

- Reads and writes may be performed on separate models
- Typically used in conjunction with Event Sourcing via
  - Commands
  - Domain Events
  - Event Store

CQRS pattern overview

- Commands
  - Issued to a service to update the write model
- Domain Events
  - Updates are recorded as immutable events to an Event store
- Event store
  - Ordered record of events for answering queries in the read model
  - Can be used for providing other materialized views of data
- The pattern can be useful for portions of a system (“bounded contexts” in DDD terminology)

What can we do in Java EE9

- Evolve the platform to facilitate CQRS implementation
- Explore with expert group to natively support
  - Commands
  - Domain Events
  - Domain Event Handlers
  - Event Store

Eventual Consistency

Eventual Consistency for Object State

Microservice instances may have a need to share state of an object

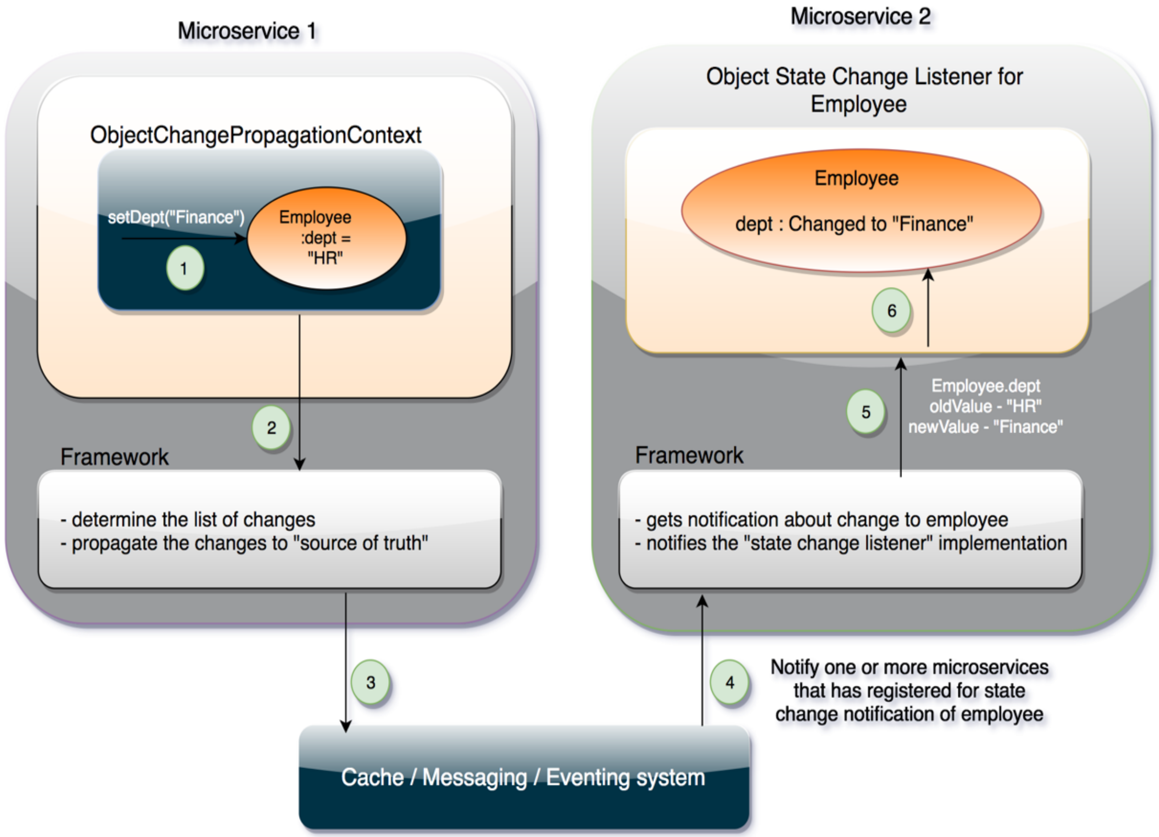
- Same object (of same identity) may be simultaneously used by them
- Changes made by one service need to be propagated to other(s)
- Multiple services may update the object simultaneously in their environment resulting in conflicts
- State sharing across micro-services could be done using multiple technologies
  - Cache (remote/distributed) based systems
  - Message oriented systems (publisher, subscriber)
  - Database based systems (push, pull)
  - Custom mechanisms
- Application code needs to make use of above vendor/technology specific APIs to achieve state sharing

Application's Responsibilities Using Various Technologies

	Caching	Messaging/Eventing	JPA+ Database
Source of truth (For data consistency)	Cache (distributed, partitioned, replicated etc.,)	Messaging provider's persistent store	Data-store
Creating an Object	"add/put" the object in the cache with an identity ("key")	Send a special message to represent creation of object by specific identity.	"persist" the Entity in the database through JPA
Updating an Object	Put ('replace") the object in the cache	Send an event/message with changes done to the object	Start a transaction and update the entity (object) in the database.
Listening to Object changes	• Add a listener to cache entry in the cache so as to be notified of object changes. • On notification, compute the difference i.e., changed attributes and refresh the object state	Receive the message/event having changes to attributes, refresh the object state	Through vendor specific means, listen to changes to a "row" in database and call EntityManager.refresh("entity") to refresh object state
Deleting an Object	"remove" the object from the cache	Send a special message to represent deletion of object by specific identity.	Start a transaction, call EntityManager.remove("entity") to delete the object from the database
Managing Conflicts in case of multiple sources updating an Object	Custom conflict resolver need to be implemented by application	No support from messaging provider. Each application instance need to detect and resolve conflict	Usually, database locks are used to avoid conflicts.
Complete POJO/Object based solution	Partial (no change notification support)	No	Partial (no change notification support, conflict resolution, need transactions)

Eventual Consistency

Listening to changes



Microservice 1

ObjectChangePropagationContext

setDept("Finance")

Employee :dept = "HR"

1

2

Framework

- determine the list of changes
- propagate the changes to "source of truth"

3

Cache / Messaging / Eventing system

4

Notify one or more microservices that has registered for state change notification of employee

Microservice 2

Object State Change Listener for Employee

Employee

dept : Changed to "Finance"

6

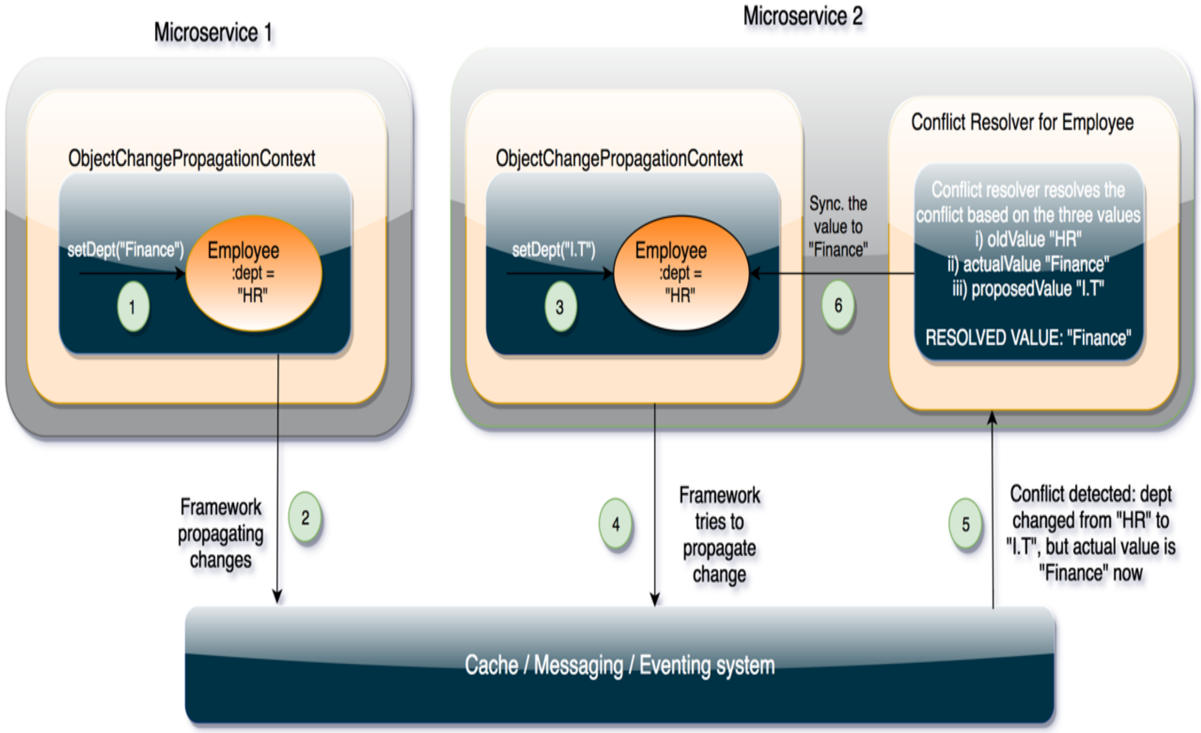
5

Employee.dept oldValue - "HR" newValue - "Finance"

Framework

- gets notification about change to employee
- notifies the "state change listener" implementation

Resolving conflicts



Microservice 1

ObjectChangePropagationContext

setDept("Finance")

Employee :dept = "HR"

1

Framework propagating changes

2

Microservice 2

ObjectChangePropagationContext

setDept("I.T")

Employee :dept = "HR"

3

Sync. the value to "Finance"

6

Conflict Resolver for Employee

Conflict resolver resolves the conflict based on the three values i) oldValue "HR" ii) actualValue "Finance" iii) proposedValue "I.T"

RESOLVED VALUE: "Finance"

4

Framework tries to propagate change

5

Conflict detected: dept changed from "HR" to "I.T", but actual value is "Finance" now

Cache / Messaging / Eventing system

Benefits

- Object based state sharing model.
- No dependency on specific technology or vendor for the micro-service code.
- Flexibility: A micro-service may decide
  - how to “refresh” the state, through auto-refresh or listen to fine grained changes and refresh
  - Whether to “lock” and then update or not
  - Whether to use “custom” conflict resolver or any framework provided conflict resolver

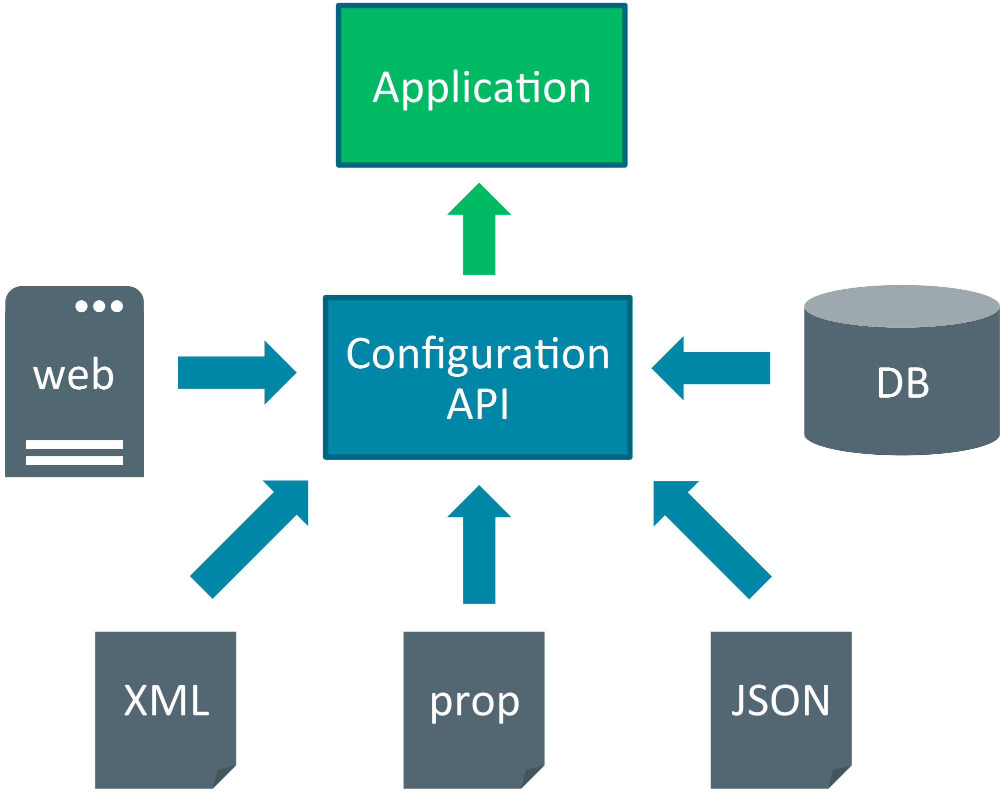
Configuration

Java API for Configuration

- A new JSR to standardize Java EE application configuration definition, access and management
- Inspired by
  - Apache Tamaya
  - Apache DeltaSpike
  - Netflix Archaius
  - Spring Configuration
- Proposed for JavaEE 8 and JavaEE 9
- Targeted for the cloud

Configuration API Main Features

- Unified API
- Properties, xml an json formats support out of the box
- Externalized configuration
- Support of multiple configuration sources
- Layering and overrides
- Optional configuration schema
- Polling and Dynamic Properties



Configuration API Sample

```java
Config config = ConfigProvider.getConfig();

// Returns "JavaOne"
String foo = config.getProperty("foo");

// Returns string "9"
String fooBar = config.getProperty("foo.bar");

// Returns null
String notExists = config.getProperty("not.exists");

// Returns string "default"
String notExistsDefault = config.getProperty("not.exists","default");

// Returns number 2016
Long fooBarBaz = config.getProperty("foo.bar.baz", Long.class);
```

```
foo=JavaOne
foo.bar=9
foo.bar.baz=2016
```

Multi-Tenancy

SaaS MultiTenancy – Use Cases

- Tenant specific UI customization
  - e.g. display tenant specific logo on the UI
  - JSF based UI composition at runtime
- Tenant specific data source
  - e.g. connect to tenant specific DB
- Tenant specific security

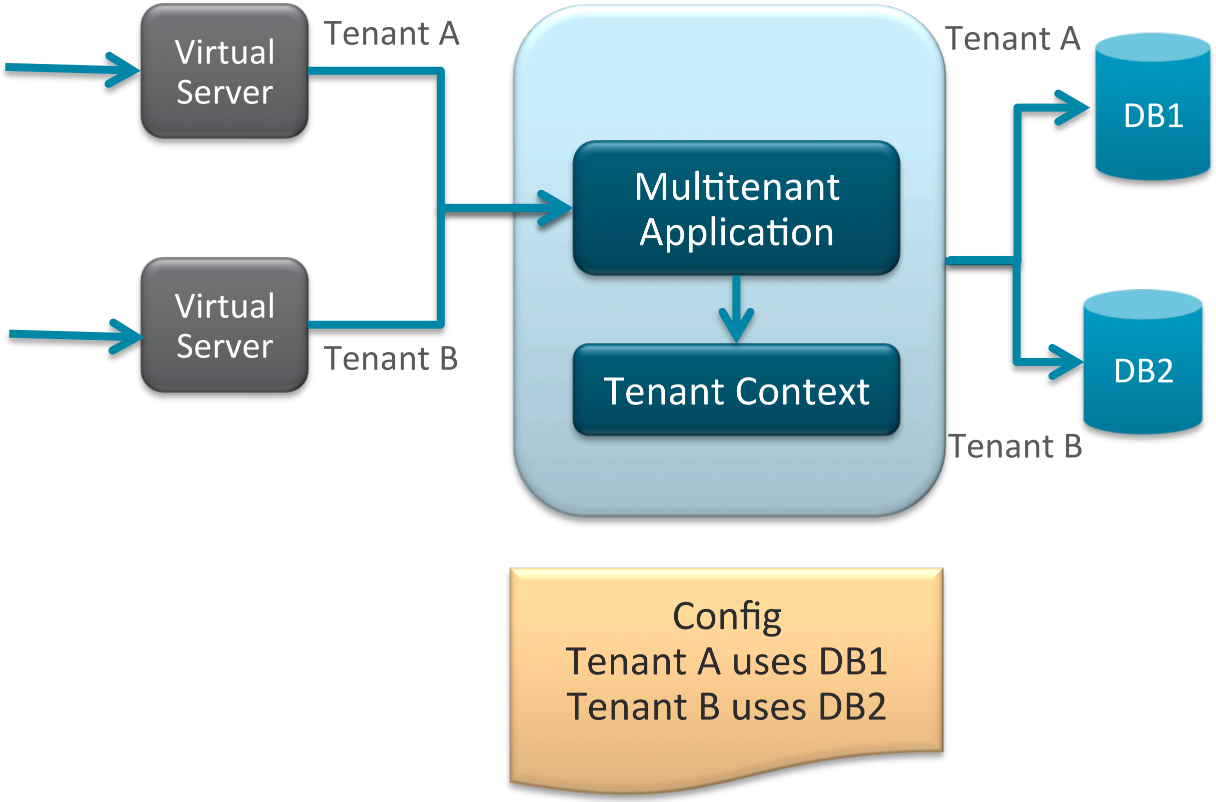
Tenant Context

- Container associates the inbound request to the Tenant and populate the TenantContext
  - e.g. use virtual server
- TenantContext holds information to identify the Tenant
  - e.g. TenantID, etc.
- Once populated, TenantContext can be used throughout by the application and the container to do tenant specific processing

```
public interface TenantContext {
  public String getTenantID();
  public String getTenantName();
  public void setProperty(String name,
String value);
  public String getProperty(String
name);
  public Map<String, String>
getProperties();
}
```

Multitenant Data Access

- Applications declare themselves as @MultiTenant
- Each tenant has its own data that is separated and protected from other tenants
- MultiTenant application uses TenantContext to connect to tenant specific DB
- Runtime uses TenantContext to connect to and return tenant specific DB by looking it up in a naming service
  - Data source APIs may be enhanced to support multitenancy via @MultiTenant to allow containers to connect to tenant specific data source automatically



Security

Proposal for Security

Problem Statements

- Identity could be from diverse Identity stores
- Authentication mechanism could change between deployment environments
- OpenIDConnect is emerging as the default authentication standard
- Who Authenticated the user?

Proposal

- Standard API for Identity Store Abstraction,
- Simple configuration to support changing Identity store
- Standard API for Authentication Mechanisms
- Extensible to support OpenIDConnect
- Security Context for Application to consistently determine how the user was authenticated, groups, roles

Java EE 9 Security

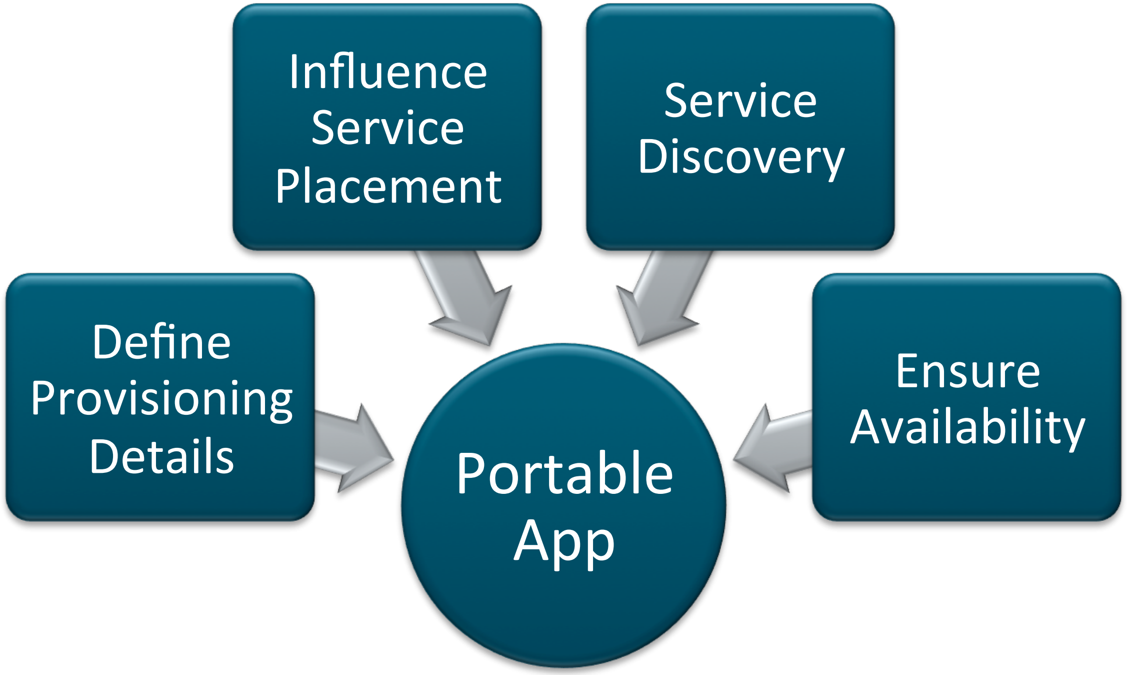
Areas for Exploration with EG

- Authorization Discover/publish OAuth Resources
  - OAuth Client registration
  - Authorization Interceptors
  - Authorization Rules EL
- Token representations
  - API to acquire tokens
  - API to validate tokens

Packaging and Orchestration

Portable Java EE 9 Microservice

Common Application Requirements Across Different Java EE 9 Environments



Influence Service Placement

Service Discovery

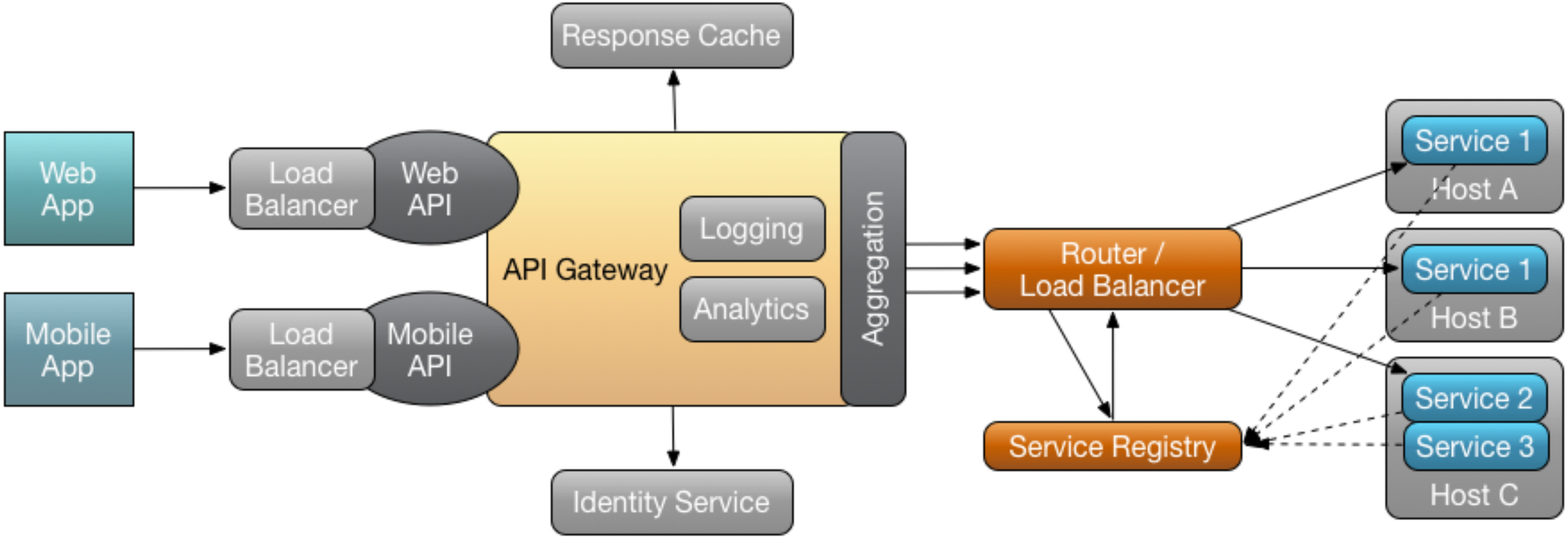
Define Provisioning Details

Ensure Availability

Portable App

High Level Architecture

Common cloud infrastructure



Java EE 9 Portable Application Requirements

Java
EE 9

Areas for exploration with EG for Spec drafts

Service Metadata

- Declare Required Resources (CPU, Memory, etc.)
- Describe Application Metadata
  - Versioning Information for Routing and Discovery
  - Dependency Information
- Service Grouping

Service Discovery

- Utilize consistent naming pattern to discover service dependencies
- Easily find Vendor Cloud Services with Injection/Auto Wiring

Availability

- Provide Health Check Method Through Metadata or Annotations
- Custom Service Performance Metrics Through Metadata or Annotations

Summary

- Java EE 9 to bring standards around microservices and developing for the cloud
  - Enables portability of applications across multiple vendors
- Want to work with existing solutions and vendors
- Standardize commonly faced problems for developers in the new environment

Next Steps

Give us your feedback

- Take the survey
  - http://glassfish.org/survey
- Send technical comments to
  - users@javaee-spec.java.net
- Join the JCP – come to Hackergarden in Java Hub
  - https://jcp.org/en/participation/membership_drive
- Join or track the JSRs as they progress
  - https://java.net/projects/javaee-spec/pages/Specifications
- Adopt-a-JSR
  - https://community.oracle.com/community/java/jcp/adopt-a-jsr

Where to Learn More at JavaOne

Session Number	Session Title	Day / Time
CON1558	What's New in the Java API for JSON Binding	Monday 5:30 p.m.
BOF7984	Java EE for the Cloud	Monday 7:00 p.m.
CON4022	CDI 2.0 Is Coming	Tuesday 11:00 a.m.
CON7983	JAX-RS 2.1 for Java EE 8	Tuesday 12:30 p.m.
CON8292	Portable Cloud Applications with Java EE	Tuesday 2:30 p.m.
CON7980	Servlet 4.0: Status Update and HTTP/2	Tuesday 4:00 p.m.
CON7978	Security for Java EE 8 and the Cloud	Tuesday 5:30 p.m.
CON7979	Configuration for Java EE 8 and the Cloud	Wednesday 11:30 a.m.
CON7977	Java EE Next – HTTP/2 and REST	Wednesday 1:00 p.m.
CON6077	The Illusion of Statelessness	Wednesday 4:30 p.m.
CON 7981	JSF 2.3	Thursday 11:30 a.m.

JavaYour
(Next)