Portable Cloud Applications with Java EE

Josh Dorr
Rajiv Mordani
Joe Di Pol
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. Packaging
3. Provisioning
4. Clustering
5. Availability
6. Versioning
7. Demo
Application Development Is Changing
Rapid Changes Over Past Few Years

Driven by increasing business needs

- Many small blocks of code, each developed and deployed independently
- Focus on Business Capabilities
- Better adhere to Continuous Delivery principles
- Distributed and decentralized architectures are inherently more resilient
- Each microservice is easy to develop but hard to deploy

[Graph showing Complexity Over Time with Microservice and Monolith lines crossing]
Microservice Trends

• Distributed Single Purpose Services
  – Results in a lot of remote calls

• Interact via REST / JSON making remote calls asynchronously

• Eventual consistency for data persistence as well as across service calls

• Reactive style programming
  – Eventing

• Built in resiliency in the runtime utilizing health check, circuit breaker and bulkhead patterns
## Technical Focus Areas

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

### Key Value/Doc Store
- Persistence and query interface for key value and document DB

### Eventual Consistency
- Automatically event out changes to observed data structures

### Configuration
- Externalize configuration
- Unified API for accessing configuration

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

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

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

### Multitenancy
- Increased density
- Tenant-aware routing and deployment

### Security
- Secret management
- OAuth
- OpenID

### State
- API to store externalized state
Proposed Platform Architecture

Java EE Packaging, Serverless, Multitenancy

Load Balancer
API Gateway
Service Discovery

Java EE Runtime

Java SE Runtime

Container Runtime

OS / Hypervisor

JSON Binding
REST API
Event API
HTTP/2
Event API
Resiliency
Security API
Eventual Consistency
State API
Config API
Key Value Store API

Reliability, Monitoring
Management and Orchestration
Scheduling and Elastic Scaling

RDBMS
NoSQL
Logging
Config
State
Security
Notification

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.
Application Portability
Portable Java EE 9 Microservice
Common Application Requirements Across Different Java EE 9 Environments

Define Provisioning Details
• Specify Resource requirements?
• Describe application attributes?

Influence Service Placement
• Group Services on Host?

Service Discovery
• Connection to my other services?
• Utilize Cloud Vendor Services?

Ensure Availability
• Declare Health Checks?
• Declare Performance Metrics?

- Cloud Infrastructure **NOT** Considered for Java EE 9
- Suggestion for Java EE 9

Java EE 9

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.
Walking Through a Microservice Lifecycle

Package the application code

Packaging
- Portable Archive
- Container

Versioning

Provisioning

Availability

Clustering
A Next Generation Application Runtime

Your Code -> Packager -> YourApp.jar

Key Java EE APIs
Runtime
Java SE

Any JVM
Cloud

Java SE
Java EE APIs
Your Code
Intelligent Platform for Portable Applications

- Our Goal is to make Microservices easy, fast, portable
- One archive with proper meta data could be deployed to any Java EE 9 Environment
- Environment can provide services that a Microservice can be automatically wired to
Technology Advances Have Driven the Platform to Cloud

Today’s infrastructure is completely different

- Containers
  - De-facto packaging mechanism
  - Immutable
  - Dynamically scheduled
  - No fixed host names, IPs, ports, …
- ‘You build it, you own it’
- Workloads distributed across DCs, regions, …
- Infrastructure is disposable
Walking Through a Microservice Lifecycle

Provision the Microservice instance in the cloud

- Packaging
- Versioning
- Availability
- Provisioning
  - Metadata
  - Grouping
- Clustering
Microservice Location in an Abstract Environment
Information Required for Provisioning

Common metadata worth standardizing

• Various Formats
  – YAML, JSON, Annotations, Properties

• Common Attributes
  – Name, Version
  – CPUS, Memory, Resource
  Dependencies
  – Grouping

```
{
  "id": "basic-3",
  "cmd": "python3 -m http.server 8080",
  "cpus": 0.5,
  "mem": 32.0,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "python",
      "network": "BRIDGE"
    }
  }
}
```

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
          - containerPort: 80
```
Instance Location In Cloud Environment

Naively placed instances versus grouping

• Typical Cloud Distribution
  – Services run anywhere in the cluster

• Service Grouping
  – Select services can be collocated
  – Reduce network traffic
Microservice Grouping Strategies

• Tight Groups (Collocated)
  – Shared Host
  – Scale entire group
  – New instances form new group

• Benefits of Tight Groups
  – Low latency
  – Shared Disk Space
  – Reduce Network Traffic

• Loose Groups / Tags
  – Affinity Level
  – Request low latency between them
  – Replicate and Scale at service level
  – New services included into group

• Benefits of Loose Groups
  – Health and Performance Monitoring
  – Log Consolidation
Microservice Grouping Examples

```json
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "",
    "labels": {
      "name": ""
    },
    "generateName": "",
    "namespace": "",
    "annotations": []
  },
  "spec": {
    "// See 'The spec schema' for details.
  }
}
```

```json
{
  "id": "/product",
  "groups": [
    {
      "id": "/product/database",
      "apps": [
        { "id": "/product/mongo", ... },
        { "id": "/product/mysql", ... }
      ]
    },
    {
      "id": "/product/service",
      "dependencies": ["/product/database"],
      "apps": [
        { "id": "/product/rails-app", ... },
        { "id": "/product/play-app", ... }
      ]
    }
  ]
}
```
Walking Through a Microservice Lifecycle

Microservice clustering

- Packaging
- Versioning
- Availability
- Provisioning
- Clustering
  - API Gateway
  - Service Discovery
High Level Architecture
Key Microservices technology – API gateway

• API Gateway is single entry point for all clients
  – Access control, protocol translation, content aggregation, response caching, logging
Service Registration and Discovery Issue

• How does my service find other back end services efficiently?
• Dynamically assigned and changing endpoints
• Requests should be load balanced

![Diagram showing service client connecting to service containers with different IP addresses and ports]

10.1.2.3:5000
10.1.2.99:5432
10.1.2.6:4545

Client Side Service Discovery

• Registry client caches queries from a dedicated Service Registry
• Client side code responsible for load balancing
• Example: Netflix Eureka, Ribbon
Server Side Service Discovery

• Discovery and Load Balancing abstracted away
• Router queries service registry
• Client sends requests through service router
• Example: AWS Elastic Load Balancer
Application Impact of Service Discovery

Service Discovery Client Examples

HttpClient.connect("http://localhost:8001");

HttpClient.connect("http://service_dns.example.com");

Springframework DiscoveryClient getInstances("SERVICE-NAME");

Application Developer Needs a Standard

– Service name + version + namespace
Supporting Service Registration and Discovery

Metadata for services suggested for Java EE 9

- ServiceName
- Compatibility Version(s)
- Implementation Version
- Dependencies
  - (service/version pairs)
- Internal vs. External

```java
@InternalService
@Service(name="userProfiles", compatibilityVersions="1,2")
@Depends(services={"userAvatars:2","userAddresses:5"})
...
public ProfileData getUserProfile(ID userId) {
  ...
}
Dependency Graphs

• It may not be obvious whether a service can be safely upgraded unless dependencies between services are tracked.
Connecting to Vendor Cloud Services

- Cloud services, Config service, eventing, state/cache/DB, can be provided as service.
- With a standard API in front of each type of platform service, Vendor can inject its specific implementation.
- Annotation or Meta-Data driven.
public class MyService {
    @EventService()
    EventService eventService;

    public MyService() {
        eventService.subscribe("StateChanges")
    }
}

public class MyService {
    @ConfigService(namespace="MyService")
    ConfigService configService;

    }
}
Walking Through a Microservice Lifecycle

Ensuring service availability

- Packaging
- Versioning
- Provisioning
- Clustering
- Availability
  - Health Checks
  - Auto Scaling
High Availability Through Service Replacement

Stateless ephemeral services
High Availability Through Service Replacement

Health check monitoring

- HTTP Health Checks with simple response results 200-399
- Container Exec for script or process invocation expecting status 0
- TCP Port Check
- Build on Java EE 8 Health Checking to support options

```json
{
  "portIndex": 0,
  "protocol": "TCP",
  "gracePeriodSeconds": 300,
  "intervalSeconds": 60,
  "timeoutSeconds": 20,
  "maxConsecutiveFailures": 0
}
```

```json
livenessProbe:
  exec:
    command:
      - cat
      - /tmp/health
    initialDelaySeconds: 15
    timeoutSeconds: 1
```
HealthCheck For Java EE 8

• Define an EE specific proposal
  – REST API which allows callers to list and execute health checks

• Declared bootstrapping end-point (/management/endpoints)
  – management/endpoints will return endpoints for health (i.e. /management/health)
  – Enable a generic client that is EE aware to discover URLs for health endpoints defined
  – Enables implementations to own forming actual URLs for health endpoints defined

• New annotations and descriptors to specify health endpoints

• Helper classes to assist with health report structure

• Implementations map from the annotations/descriptors marking health endpoints to the actual URLs
HealthCheck For Java EE 8

• "Scope" is defined by the implementation, for example:
  – One implementation may limit available health checks to only what can be specified with the EE annotations/descriptors
  – Another implementation may include scoping levels which provide the health of implementation specific concepts (domain, servers, clusters, partitions/tenants, load balancers, micro-services, etc...)

• Security requirements are dictated by the implementation, for example:
  – One implementation may allow simple status checks to be unauthenticated, restricting access to checks with detailed reasons
  – Another implementation may restrict all checks, etc...
Metrics & Performance Monitoring for Java EE 8

- Define Performance Monitoring Configuration/Declaration
  - REST API which allows callers to identify and access metric information
- New annotations and descriptors to allow developers to specify custom metric endpoints and custom metric metadata
- Helper classes to assist with surfacing metrics
- Implementations map from the annotations/descriptors marking metric endpoints to the actual URLs
- SPI to allow vendors to provide implementations for in-process agents for harvesting metric collections into a centralized service
- Security requirements are dictated by the implementation
High Availability Through Horizontal Scaling

Common cloud infrastructure supported scaling

• Declaratively specify the number of instances for their service. This should automatically adjust the current number.

• Specify AutoScaling policies supporting
  – CPU consumption
  – Memory consumption
  – Request routing information
  – Custom metrics collecting summarized information from the app
  – Predictive scheduling

• Dynamic Service/Feature Configuration of Scaling Policies
Policy Based Scaling

• A standardized performance check could be used to create automatic scaling policies that can be applied to any service

• Example Policy:
  – Scale out if all instances report 75% utilization or higher for 3 minutes
  – Scale out if all instances report 90% utilization or higher
  – Scale back if all instances report 20% utilization or less for 15 minutes
  – Scale back if average utilization is 40% or less and number of instances > 6
  – Minimum and Maximum Bounds on number of instances
  – Policy can be schedule-based to accommodate anticipated peaks and troughs
Walking Through a Microservice Lifecycle

Independent service versioning

- Packaging
- Versioning
  - Rolling Replacement
  - Blue / Green Updates
- Provisioning
- Resiliency
- Clustering
Versioning and Upgrades Common Strategies

• Incompatible Upgrades
  – Deploy each new version as a unique service (Netflix)
  – Register the new version at a new location (different port, or version number in path)
  – Leverage auto-scaling to allow new version to scale up and older version to scale down as client base shifts

• Backward Compatible Upgrades
  – Each new service version is written to translate and emulate current and previous compatibility version (provides backwards compatibility)
  – Data Model conversion techniques
  – Blue / Green or Rolling Replacement
Incompatible Upgrades

- Service MetaData includes compatibility version
- If compatibility version of new service doesn’t include compatibility version of old service instance, then we need side by side deployment
- Auto Scaling will decrease the number of v1 instances over time as clients move to v2. For the same reason, it will increase the number of v2 instances
Side By Side Versions

- All client requests include a version (either in the path or name or metadata)
- The service registry is used to route the request to an instance that supports that version
- In this case a different instance is used for each version
Backwards Compatible Upgrades

- Service MetaData includes compatibility version
- If compatibility version of new service includes compatibility version of old service instance, then we can replace original instance
- Creating the new instance before removing the old ensures continuity of availability
Backwards Compatible Requests

- All client requests include a version (either in the path or name or metadata)
- The service registry is used to route the request to an instance that supports that version
- In this case the same instance can support both versions
Blue / Green Requests

- All client requests include a version (either in the path or name or metadata)
- The service registry is used to route the request to an instance that supports that version
- In this case a different instance is used for each version
Compatible Requests

Rolling Replacement
Java EE 9 Portable Application Requirements

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
Concepts Demo

Java EE 9 Service Group

Cloud Provider Tooling

Browser

Java EE Cloud Provider

Sparklines Group Instance

Mesos / Marathon

VM 1

VM 2

Sparklines Web App

Sparklines Resources

Sparklines Cache

Sparklines Web App

Sparklines Resources

Sparklines Cache

Browser

Sparkline Service Group

- sparklines-group.json
- sparklines-webapp.jar
- sparklines-resources.jar
- sparklines-cache.jar
Summary

• Java EE 9 enables **portability of applications** across multiple vendors
  – Bring standards around microservices and developing for the cloud
  – leverage the features that the cloud offers, like resiliency, scalability, and efficiency

• 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

• 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

<table>
<thead>
<tr>
<th>Session Number</th>
<th>Session Title</th>
<th>Day / Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CON7980</td>
<td>Servlet 4.0: Status Update and HTTP/2</td>
<td>Tuesday 4:00 p.m.</td>
</tr>
<tr>
<td>CON7978</td>
<td>Security for Java EE 8 and the Cloud</td>
<td>Tuesday 5:30 p.m.</td>
</tr>
<tr>
<td>CON7979</td>
<td>Configuration for Java EE 8 and the Cloud</td>
<td>Wednesday 11:30 a.m.</td>
</tr>
<tr>
<td>CON7977</td>
<td>Java EE Next – HTTP/2 and REST</td>
<td>Wednesday 1:00 p.m.</td>
</tr>
<tr>
<td>CON6077</td>
<td>The Illusion of Statelessness</td>
<td>Wednesday 4:30 p.m.</td>
</tr>
<tr>
<td>CON 7981</td>
<td>JSF 2.3</td>
<td>Thursday 11:30 a.m.</td>
</tr>
</tbody>
</table>