Microservices
Technology Enabler
from Oracle

iJUG / Oracle Roadshow 2015

Peter Doschkinow
Michael Bräuer
November 2015
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

• Microservices
• Jersey features for microservices
• Demos
Characteristics of Existing Monolith Architecture

The status quo has served us well but there are new alternatives

- Three tiers
- Scale by cloning behind load balancer (X-axis scaling)
- One programming language
- Everything centralized – messaging, storage, database, etc

One large archive, including UI(s) and application code

Feature-rich – support large, complicated applications, many use cases

Provide 100% isolation between tenants

Procured and manually set up
Existing Monolith Architecture Has its Limits

**Too Complex**
Apps get too big and complicated for a developer to understand over time. Shared layers (ORM, messaging, etc.) have to handle 100% of use cases – no point solutions.

**Too Slow**
Teams split up by function – UI, application, middleware, database, etc. Takes forever to get anything done due to cross-ticketing.

**Too Fragile**
A bug will quickly bring down an entire application. Little resiliency.

**No Specialization**
Different parts of applications have different needs – more CPU, more memory, faster network, etc.. Can not evolve at a different pace.

**No Ownership**
Code falls victim to “tragedy of the commons” – when there’s little ownership, you see neglect.

**Inefficient Testing**
Each time you touch the application, you have to re-test the whole thing. Hard to support continuous delivery.
What Are Microservices?

Minimal function services that are deployed separately but can interact together to achieve a broader use-case

<table>
<thead>
<tr>
<th>Status Quo</th>
<th>Microservices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single, Monolithic App</td>
<td>Many, Smaller Minimal Function Microservices</td>
</tr>
<tr>
<td>Must Test/Deploy/Scale Entire App</td>
<td>Can Test/Deploy/Scale Each Microservice Independently</td>
</tr>
<tr>
<td>One Database for Entire App</td>
<td>Each Microservice Has Its Own Datastore</td>
</tr>
<tr>
<td>In-process Calls Locally, SOAPExternally</td>
<td>REST Calls Over HTTP, Messaging, or Binary</td>
</tr>
<tr>
<td>Organized Around Technology Layers</td>
<td>Organized Around Business Capabilities</td>
</tr>
<tr>
<td>One Technology Stack for Entire App</td>
<td>Choice of Technology for Each Microservice</td>
</tr>
<tr>
<td>Developers Don’t Do Ops</td>
<td>Developers + Ops Support Production in Perpetuity</td>
</tr>
</tbody>
</table>
Benefits of Microservices Come With Costs

**Benefits**

**Strong Module Boundaries**
*Forces boundaries because each module is deployed separately*

**Independent Deployment**
*Each team is free to deploy what/when they want*

**Ability to Pick Different Technology**
*Each team can pick the best technologies for each microservice*

**Costs**

**Distributed Computing**
*Microservice deployed separately, with latency separating each service*

**Eventual Consistency**
*System as a whole is eventually consistent because data is fragmented*

**Operational Complexity**
*Need mature DevOps team, with very high skills*
Microservices: Reality Check

• The name “Microservices” is incredibly vague
  – Big hurdle to practical adoption by average Joe developer
  – Already hijacked and overloaded by commercial interests

• Simple concept with a long history
  – UNIX, CORBA, Jini, RMI, EJB 1/2, COM/DCOM, OSGi, SOAP/ESB
  – A SOA with some special characteristics

• Decomposing larger systems into smaller independently deployable parts
  – Purists distance themselves from SOAP, ESB
  – Purists embrace mostly REST and messaging
  – Purists take for granted testing, DevOps, continuous delivery
  – Purists focus on (ridiculously) fine grained services
  – Purists consider the implementation of non-functional requirements to be part of the service
Microservices: The Bottom Line

– Majority of systems just fine as “monoliths”
– Majority of systems needing microservices could evolve into “hybrids”
– Few practical enterprise systems can or need to achieve microservices nirvana

... don’t even consider microservices unless you have a system that’s too complex to manage as a monolith.
The majority of software systems should be built as a single monolithic application. 
Do pay attention to good modularity within that monolith, but don’t try to separate it into separate services 

http://martinfowler.com/bliki/MicroservicePremium.html
Microservices Related Technologies

• Frameworks: fat jars, “containerless”
  – Vert.x, Spring Boot, Dropwizard
  – WildFly Swarm, Payara Micro/Embedded GlassFish, TomEE Embedded
  – Grizzly + Jersey + WebSocket + ...

• Java libraries for reactive programming
  – RxJava, Hystrix

• Virtualization
  – Docker, Rocket

• Cloud
  – IaaS, PaaS
WebLogic Multitenant Microcontainer for Microservices

Similar to Oracle Database pluggable/container databases

• Each microservice instance can have its own light-weight WebLogic container-like partition
• Partition isolation inside the JVM
• Easily move partitions between WebLogic hosts
• Each partition is exceptionally light
• Each WebLogic host can support hundreds of partitions
JAX-RS/Jersey primer

• JAX-RS 2.0
  – part of Java EE 7 (2013)
  – defines a standard API for
    • Implementing RESTful web services in Java
    • REST client API

• Jersey 2.0
  – provides production ready JAX-RS 2.0 reference implementation
  – brings several non-standard features
  – Current version is 2.22.1
Agenda

• Microservices
• Jersey features for microservices
• Demos
Jersey for Microservices

- Integration with various HTTP containers and client transports
- Reactive/Async Client
- Test Framework, Monitoring and Tracing
- Support for SSE
- Dynamic reloading
- Various data bindings
- Security
- MVC view templates
- Weld (CDI) support
Supported server containers

- Grizzly HTTP server
- Jetty HTTP Container (Jetty Server Handler)
- Servlet 2.4-3.1
- Java SE HTTP Server (HttpHandler)
- Other containers could be plugged in via ContainerProvider SPI
Grizzly Lightweight HTTP Server: High Performance I/O

Great for inter-process communication

- Oracle sponsored open source
- Brings non-blocking sockets to the protocol processing layer
  - Support for non-blocking I/O and HTTP processing
- HTTP/2, WebSocket, Comet Support
- Serves static resources
- Endless configuration possibilities
Grizzly HTTP server support and configuration

HttpServer httpServer =
   GrizzlyHttpServerFactory.createHttpServer(AppURI, new JaxRsApplication(), false);
httpServer.getServerConfiguration().setSessionTimeoutSeconds( . . .);
NetworkListener grizzlyListener = httpServer.getListener("grizzly");
grizzlyListener.getTransport().setSelectorRunnersCount(4);
grizzlyListener.getTransport().setWorkerThreadPoolConfig(
   ThreadPoolConfig.defaultConfig().setCorePoolSize(16).setMaxPoolSize(16));
listener.setDefaultErrorPageGenerator( . . .);
listener.getFileCache().setMaxCacheEntries( . . .);
listener.getCompressionConfig().setCompressionMode( . . .);
httpServer.start();
HTML5 App with Jersey+Tyrus+Grizzly: Drawing Board Demo

• Collaborative drawing
• Two-page application
  – List of drawings
  – Drawing
• Demonstrating
  – Server-side
    • Java EE 7: JAX-RS, JSON, WebSocket
    • Jersey specific: SSE, JSON-B
    • Lightweight integration Jersey+Tyrus+Grizzly – only 10 MB footprint!
  – Client-side: AngularJS or JavaFX
JAX-RS based Microservices Orchestration

Travel Agency Demo Application

• Remote
  – Destinations, weather, quoting
  – application/json, application/xml
  – Delays are simulated

• Travel agency client
  – application/json
  – Dependent calls

https://github.com/jersey/jersey/tree/master/examples/rx-client-java8-webapp
https://jersey.java.net/documentation/latest/user-guide.html#rx-client
Orchestration Layer Benefits

• Client specific API
  – Different needs for various devices: screen size, payment methods, ...

• Single Entry Point
  – No need to communicate with multiple services

• Thinner client
  – No need to consume different formats of data

• Less frequent client updates
  – Doesn’t matter if one service is removed in favor of another service
Implementing the Service
A Naïve Approach

1. Get Customer Details
2. Get a list of 10 Recommended Destinations
3. For each Destination:
   a. Get Quote for the Customer: 170 ms
   b. Get Weather Forecast: 330 ms

Total Time: 5 400 ms
Client – Synchronous Approach

• Easy to read, understand and debug
  – Simple requests, Composed requests

• Slow
  – Sequential processing even for independent requests

• Wasting resources
  – Waiting threads

• Suitable for
  – Lower number of requests
  – Single request that depends on the result of previous operation
Implementing the Service

Optimized Approach

- Get Customer Details
- Get a list of 10 Recommended Destinations
  - for each Destination
    - Async Get Quote for the Customer
    - Async Get Weather Forecast

Total Time: 730 ms
Client – Asynchronous Approach

Futures

• Returns immediately after submitting a request
  – Future

• Harder to read, understand and debug
  – Especially when dealing with multiple futures and composed, dependent calls

• Need to find out when all Async requests finished
  – Relevant only for 2 or more requests (CountDownLatch)

• Fast
  – Each request can run on a separate thread

• Suitable for many independent calls
Jersey Client Features

• Fluent API for sync and async calls
• Reactive extensions
• Many connectors (Grizzly, Jetty, Apache, …)
  – Alternatives to the Jersey default transport, based on HttpURLConnection
• Secure (SSL, Digest, Basic, OAuth, …)
• Various data bindings
• Filters
Reactive Jersey Client API

Reactive programming model

• Easier programming for asynchronous data streams
• Data flow
  – execution model propagates changes through the flow
• Event based
  – notify observers about new events, completion or error
• Composable
  – compose/ transform streams into a resulting stream
• Reactive client API to be introduced in JAX-RS 2.1

https://github.com/jersey/jersey/tree/master/ext/rx
Reactive Jersey Client API
Abstraction over different reactive libraries

• Java 8: CompletionStage, CompletableFuture
  – Native part of JDK
  – Fits the new Java Stream API programming model
  – JSR166e – Support for CompletableFuture on Java SE 6 and Java SE 7

• RXJava: Observable
  – Currently most advanced reactive API in Java
  – Contributed by Netflix – hardened & tested in production

• Guava: ListenableFuture, Futures
  – Similar to Java SE 8
SyncInvoker and AsyncInvoker

```java
public interface SyncInvoker {
    Response get();
    <T> T get(Class<T> responseType);
    <T> T get(GenericType<T> responseType);
    // ...
}

public interface AsyncInvoker {
    Future<Response> get();
    <T> Future<T> get(Class<T> responseType);
    <T> Future<T> get(GenericType<T> responseType);
    // ...
}
```
RxInvoker and an extension Example

```java
public interface RxInvoker<T> {
    // for now T can be
    // CompletionStage/Java8, Observable/RxJava, CompletableFuture/jsr166, ListenableFuture/Guava
    T get();
    <R> T get(Class<R> responseType);
    <R> T get(GenericType<R> responseType);
    // ...
}

public interface RxCompletionStageInvoker extends RxInvoker<CompletionStage> {
    CompletionStage<Response> get();
    <T> CompletionStage<T> get(Class<T> responseType);
    <T> CompletionStage<T> get(GenericType<T> responseType);
    // ...
}
```
Sync Client Example

SyncInvoker used

private WebTarget destination;
List<Destination> recommended = Collections.emptyList();
...
    recommended = destination.path("recommended").request()
        // Identify the user.
        .header("Rx-User", "Sync")
        // Return a list of destinations.
        .get(new GenericType<List<Destination>>() {});
...

Async Client Example

AsyncInvoker used

```java
private WebTarget destination;
List<Destination> recommended = Collections.emptyList();
...
recommended = destination.path("recommended").request()
    // Identify the user.
    .header("Rx-User", "Sync")
    // Async invoker.
    .async()
    // Return a list of destinations.
    .get(new InvocationCallback<List<Destination>>() {
        @Override
        public void completed(final List<Destination> recommended) {
            ...
        }
    });
```
RxObservableInvoker used

```java
private WebTarget destination;
List<Destination> recommended = Collections.emptyList();
...
final Observable<Destination> recommended = RxObservable.from(destination).path("recommended").request()
    .header("Rx-User", "RxJava")
    .rx()
    .get(new GenericType<List<Destination>>() {})
    .flatMap(Observable::from)
    .cache();
```
Jersey Test Framework

• Based on JUnit
• Support for TestNG available
• Multiple container support
  – Grizzly
  – In memory
  – Java SE Http Server
  – Jetty
  – External container support
Monitoring support

• Powerful monitoring API
  – Basic statistics collected

• Must be explicitly enabled
  – ServerProperties.MONITORING_STATISTICS_ENABLED
  – ServerProperties.MONITORING_STATISTICS_MBEANS_ENABLED
  – Register your own event listeners

• MonitoringStatistics could be injected into any resource and reused:
  – @Inject private Provider<MonitoringStatistics> statistics;
Grizzly and Jersey Monitoring Demo
https://github.com/PetrJanouch/JavaOne2015-Monitoring-Demo
Jersey 3.0

• Jersey 2.x branched off and 3.x on the master
• Based on JAX-RS 2.1
  – Non-blocking IO
  – SSE support
  – Support for reactive programming
• Java 8 friendly
• Backwards compatible with 2.x
Jersey 3.0 Non-Blocking I/O

• Extra performance boost
• Inspired by but not based on Servlet 3.1
• Beneficial for large and streamed entities
• A brand new client connector
  – Getting rid of HttpURLConnection
  – First version already in incubator
  – Much better performance than HttpURLConnection even in blocking mode
Summary

• Microservices are a valuable architectural technique, but:
  – not necessarily for everyone
  – not necessary always
  – not necessarily all-at-once

• Building microservices with Jersey is easier
  – Many microservices-related features in Jersey are going to be standardized
Integrated Cloud
Applications & Platform Services