
ORACLE®

JavaOne™

ORACLE®

Keep Learning with Oracle University

ORACLE UNIVERSITY

Classroom Training

Learning Subscription

Live Virtual Class

Training On Demand

Cloud

Technology

Applications

Industries

education.oracle.com

Session Surveys

Help us help you!!

- Oracle would like to invite you to take a moment to give us your session feedback. Your feedback will help us to improve your conference.
- Please be sure to add your feedback for your attended sessions by using the Mobile Survey or in Schedule Builder.

RESTful Microservices

Petr Janouch
Software Developer
Oracle, Application Server Group
October 28, 2015

JavaOne
ORACLE

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.

Goal of The Presentation

- To show how Jersey as JAX-RS 2.0 could be used outside a Java EE container in a light-weight fashion to implement RESTful micro-services in Java

Agenda

1. Microservices Primer
2. JAX-RS/Jersey Primer
3. Jersey features to support microservices development

Agenda

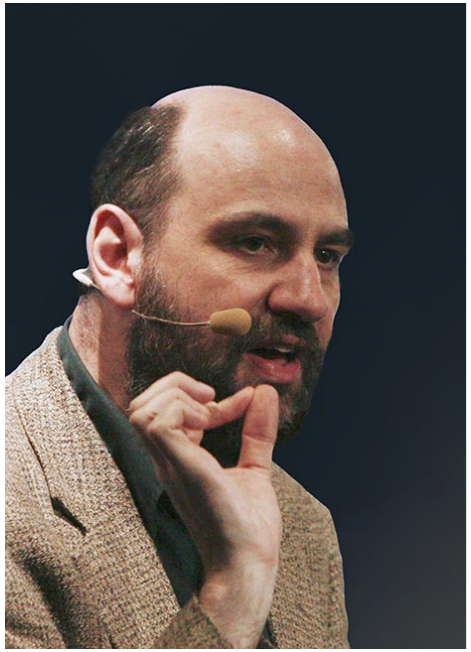
1. Microservices Primer
2. JAX-RS/Jersey Primer
3. Jersey features to support microservices development

Properties of large monolith applications

- Large applications too complex and fragile
- Deployments slow and expensive
  - entire application must be tested
  - hard to deploy/test any module in isolation
- Problem tracking and isolation
- Scaling only the entire application

Dream of a microservice

- working the unix way
  - narrow your scope
  - do one thing but do it well
- decoupling
  - simpler
  - easier
  - cheaper
  - faster to develop



"Small services,
each running in its own process and communicating with lightweight mechanisms."

— *Martin Fowler, ThoughtWorks*

Properties of Microservice Architecture

- Isolated impact of changes
  - Can be rewritten rather than maintained
  - Easier to upgrade technologies
- Isolated scope
  - reflect one business capability
  - small enough to fit in your head
- Container-less deployment
  - Self-contained
  - Single OS process
- Smart endpoints & dump pipes
  - Communicate using standardised application protocols and message semantics
- Cloud-friendly
  - Auto-scaling and designed for failures
- Enforces modularity
- “You build it, you run it” model
  - stronger customer focus

Microservices -The Hard Stuff

- Provisioning
  - How do I deploy a service?
  - How do I upgrade a service to a new version?
  - How do I upgrade multiple services to a new version?
  - How do I ensure consistent configuration?
- Integration & discovery
  - Where can I find the service I need to interact with?
  - How do I interact with a service?
- Testing & Troubleshooting
  - How do I test and debug in a distributed service environment?
- Consistency
  - How do I ensure that all services expose consisted API?
- Fault tolerance and isolation

Application interface

- Be of the web not on the web!
- HTTP and universal media types can be consumed by different clients
- Looks familiar? You are right, this is REST

Agenda

1. Microservices Primer
2. JAX-RS/Jersey Primer
3. Jersey features to support microservices development

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 many non-standard features

Notable features

- Integration with various HTTP containers and client transports
- Support for SSE
- MVC view templates
- Reactive/Async Client
- Security (SSL, OAuth, …)
- Test Framework
- Monitoring and Tracing
- Various data bindings

One slide Jersey application

```
HttpServer httpServer = GrizzlyHttpServerFactory.createHttpServer(myUri, new
MyApp(), false);
httpServer.start();

public MyApp() extends ResourceConfig {
    super(HelloResource.class);
}

@Path(“hello”)
public class HelloResource {

    @GET
    public String sayHello() {
        return “Hello”;
    }
}
```

Agenda

1. Microservices Primer
2. JAX-RS/Jersey Primer
3. Jersey features to support microservices development

Selected Jersey features

- Grizzly HTTP server support
- Application monitoring and tracing
- Powerful client

Supported server containers

- **Grizzly HTTP server**
- Servlet 2.4-3.1
- Jetty HTTP Container (Jetty Server Handler)
- Java SE HTTP Server (HttpHandler)
- Other containers could be plugged in via ContainerProvider SPI

Grizzly HTTP server

- Lightweight HTTP server
- High performance
- Powers Glassfish AS
- HTTP 2, Websockets, Comet
- Secure
- Optional Servlet API
- Serves static resources

Grizzly configuration example

```
HttpServer httpServer = GrizzlyHttpServerFactory
    .createHttpServer(myUri, new MyApp(), false);
HttpHandler httpHandler = new
CLStaticHttpHandler(HttpServer.class.getClassLoader(), "/static/");
httpServer.getServerConfiguration().addHttpHandler(httpHandler, "/");
httpServer.getServerConfiguration().setSessionTimeoutSeconds( . . .);
NetworkListener listener = httpServer.getListener("grizzly");
listener.getTransport().setSelectorRunnersCount(4);
listener.getTransport().setWorkerThreadPoolConfig(
ThreadPoolConfig.defaultConfig().setCorePoolSize(8).setMaxPoolSize(16));
listener.setDefaultErrorPageGenerator(. . .);
listener.getFileCache().setMaxCacheEntries(. . .);
listener.getCompressionConfig().setCompressionMode( . . .);
```

Monitoring support

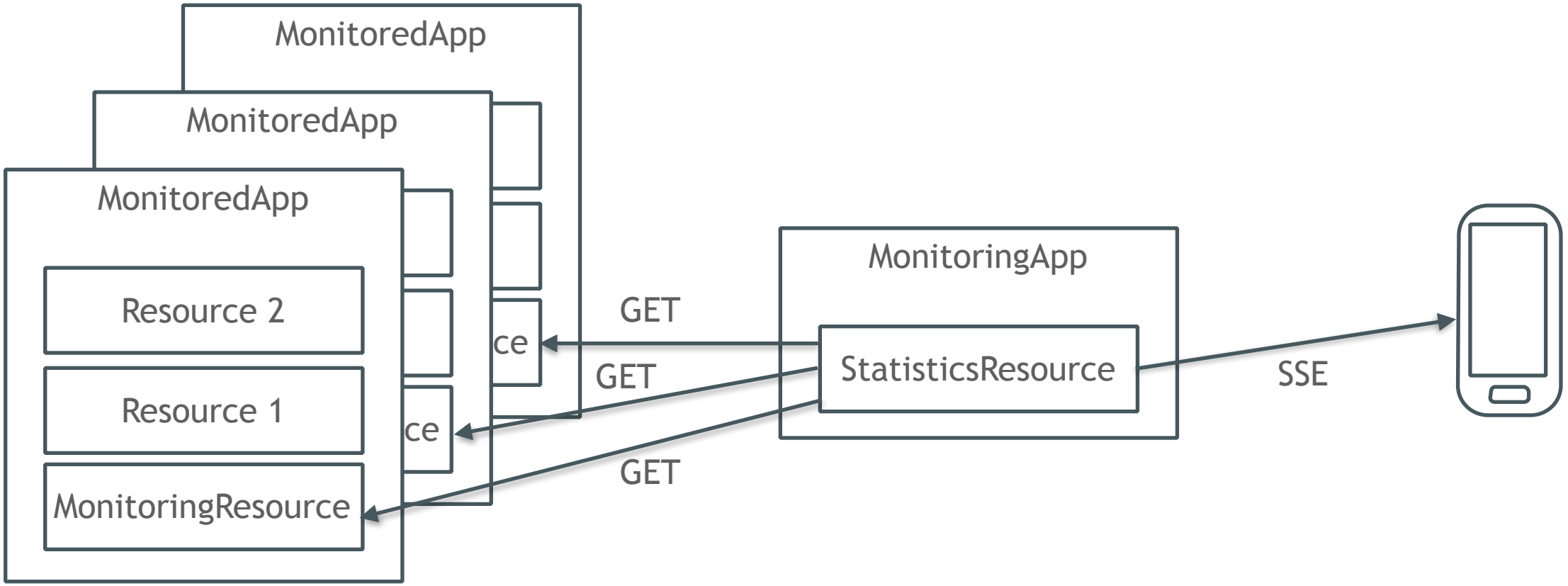
- Powerful monitoring API
- Basic statistics collected
- Custom event listeners can be created
- MBean and programmatic API
- Statistics can be injected into a resource:

```
@Inject
private Provider<MonitoringStatistics> statistics
```

Custom monitoring event listeners

```
public class MyRequestEventListener implements RequestEventListener {
    private final long startTime = System.currentTimeMillis()

    @Override
    public void onEvent(RequestEvent event) {
        switch (event.getType()) {
            case RESOURCE_METHOD_START:
                System.out.println("Resource method "
                    + event.getUriInfo().getMatchedResourceMethod()
                        .getHttpMethod()
                    + " started for request " + requestNumber);
                break;
            case FINISHED:
                System.out.println("Request " + requestNumber
                    + " finished. Processing time "
                    + (System.currentTimeMillis() - startTime) + " ms.");
                break;
        }
    }
}
```

Grizzly and monitoring demo



Grizzly and monitoring demo

```
@Path("/resource1")
public class MonitoredResource1 {

    @GET
    public String getHello() {return "Hello from resource 1";}
}

@Path("/resource2")
public class MonitoredResource2 {

    @GET
    public String getHello() {return "Hello from resource 2";}
}
```

Grizzly and monitoring demo

```
@Path("monitoring")
public class MonitoringResource {

    @Inject private Provider<MonitoringStatistics> statistics;

    @Produces(MediaType.APPLICATION_JSON)
    @GET
    public MonitoringData get() {
        MonitoringData monitoringData = new MonitoringData();
        Map<String, Long> rr = statistics.get()
                .getResourceClassStatistics()
                .entrySet()
                .stream()
                .collect(Collectors.toMap(
                        e -> e.getKey().getSimpleName(),
                        e -> e.getValue()
                                .getRequestExecutionStatistics()
                                .getTimeWindowStatistics()
                                .get(1000L)
                                .getRequestCount()));
        monitoringData.setRequestsPerResource(rr);
        return monitoringData;
    }
}
```

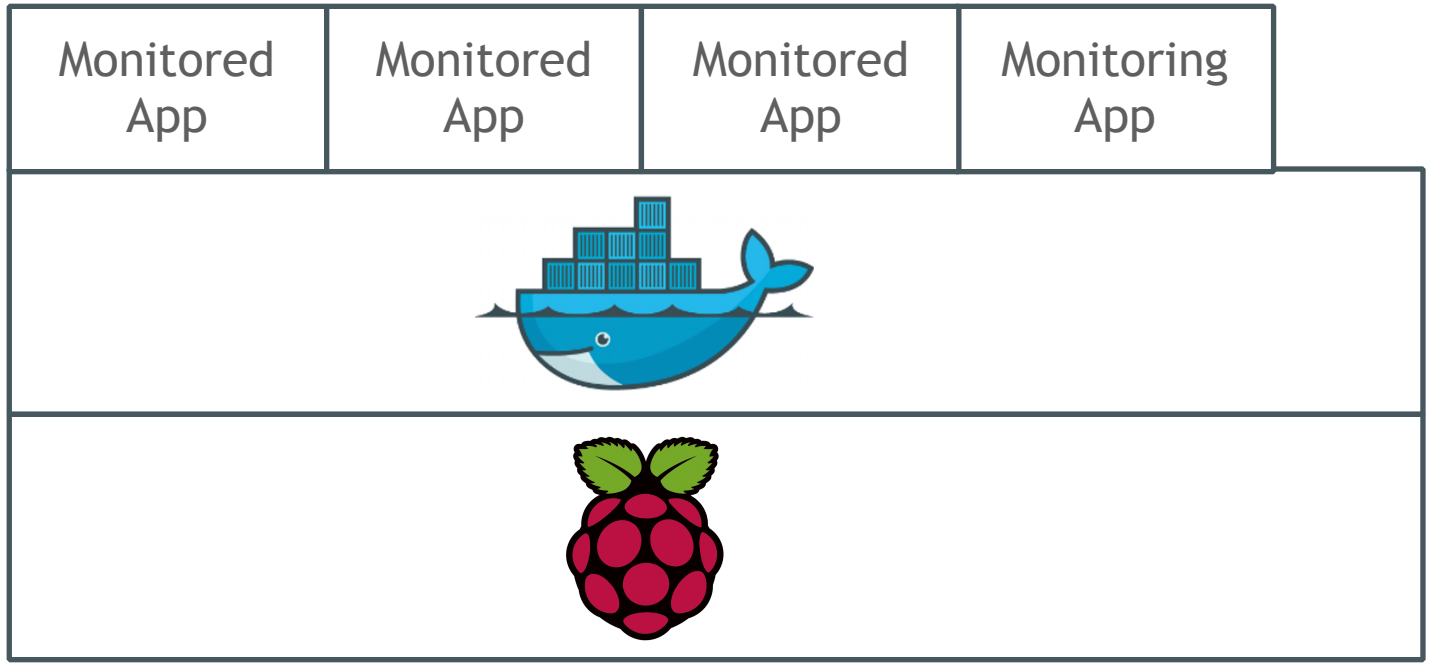
Grizzly and monitoring demo

```
@Path("statistics")
public class StatisticsResource {

    private static final SseBroadcaster broadcaster = new SseBroadcaster();
    private static final ScheduledExecutorService scheduler = ...
    private static final Client client = ClientBuilder.newClient();

    @Inject
    private MonitoringApp monitoringApp;

    @GET
    @Produces(SseFeature.SERVER_SENT_EVENTS)
    public EventOutput get() {
        EventOutput output = new EventOutput();
        broadcaster.add(output);
        scheduler.scheduleAtFixedRate(this::broadcastStatistics, 0, 1, TimeUnit.SECONDS);
        return output;
    }

   . . .
}
```

Grizzly and monitoring demo

```
private void broadcastStatistics() {
        List<URI> monitoringEndpoints = monitoringApp.getMonitoredApps();
        List<MonitoringData> monitoringData = monitoringEndpoints
                .stream()
                .map((endpointUri) -> {
            Response response = client.
                    target(endpointUri)
                    .path("monitoring").request()
                    .get();
            MonitoringData data = response.readEntity(MonitoringData.class);
            data.setNode(endpointUri.getHost() + ":" + endpointUri.getPort());
            return data;
        }).collect(Collectors.toList());

        OutboundEvent event = new OutboundEvent.Builder()
                .mediaType(MediaType.APPLICATION_JSON_TYPE)
                .data(monitoringData).build();
        broadcaster.broadcast(event);
    }
```

Demo deployment



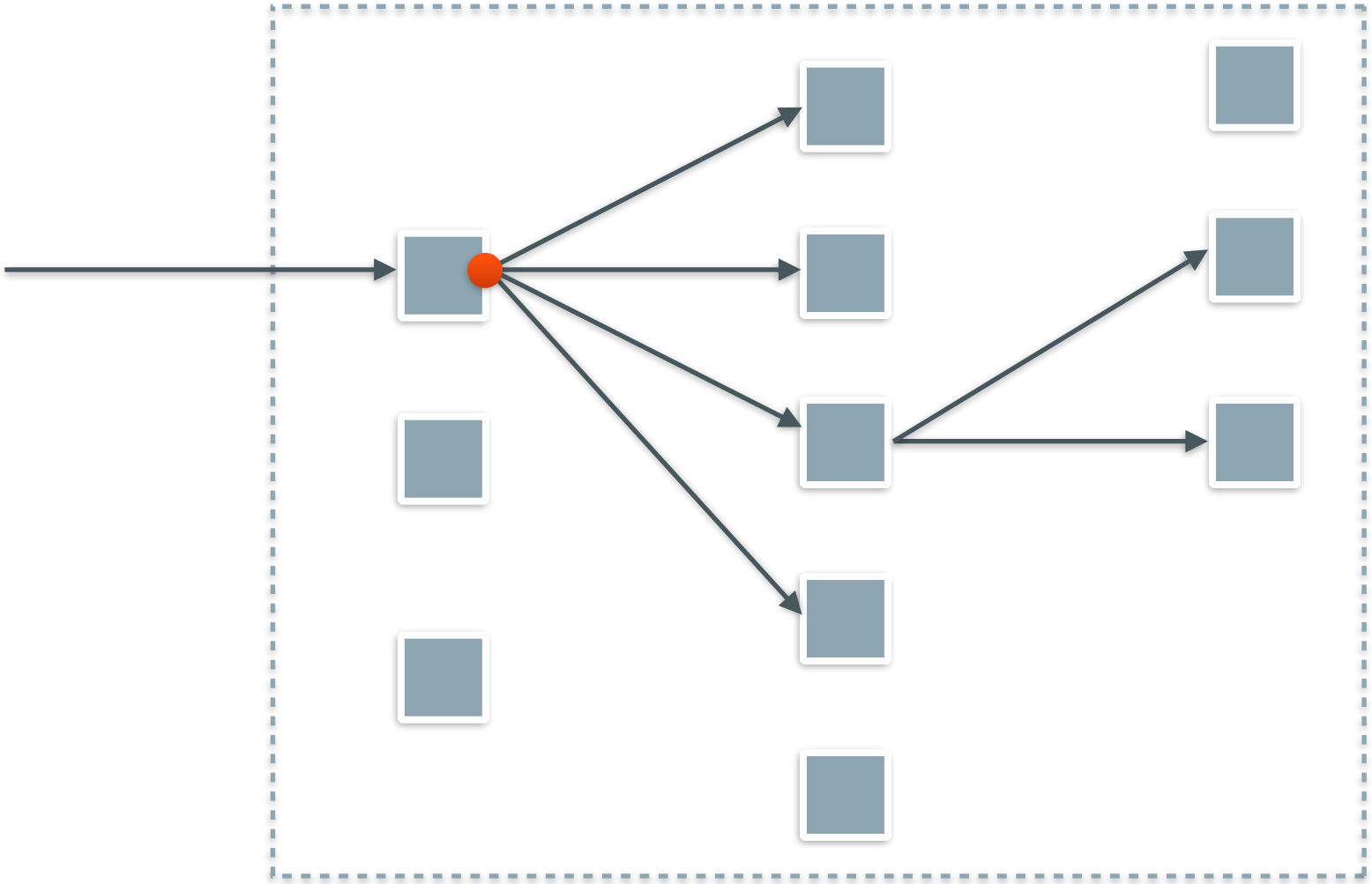

```
# create arm/java image
FROM resin/rpi-raspbian:wheezy
COPY jre /data/jre
ENV PATH /data/jre/bin:$PATH
CMD ["java", “-version"]
```

```
# create monitoredApp image
FROM arm/java8
COPY Monitored-app.jar /data/Monitored-app.jar
CMD ["java", "-jar", "/data/Monitored-app.jar"
```

Grizzly and monitoring demo summary

- https://github.com/PetrJanouch/JavaOne2015-Monitoring-Demo

Client in the microservice world



Jersey client primer

```
Client client = ClientBuilder.newClient(new ClientConfig()
            .register(MyClientResponseFilter.class)
            .register(new AnotherClientFilter()));

String entity = client.target("http://example.com/rest")
            .register(FilterForExampleCom.class)
            .path("resource/helloworld")
            .queryParam("greeting", "Hi World!")
            .request(MediaType.TEXT_PLAIN_TYPE)
            .header("some-header", "true")
            .get(String.class);
```

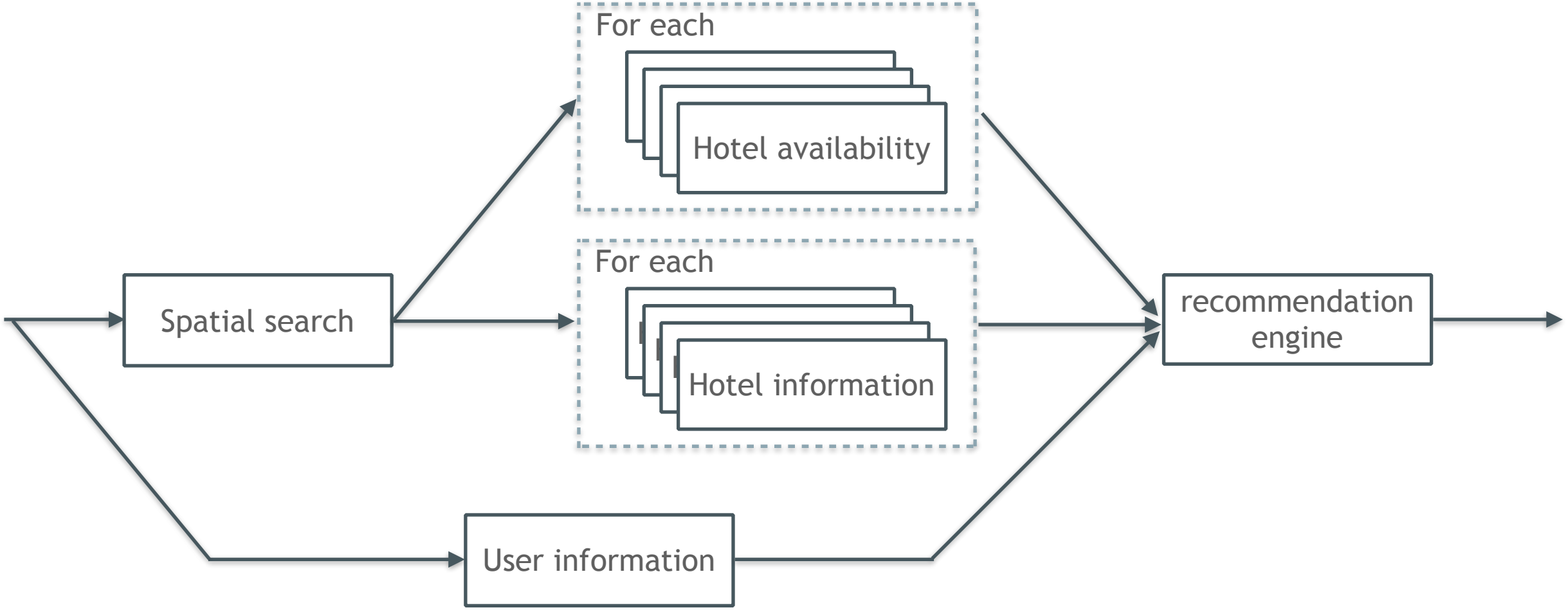
Jersey client - features

- Fluent API
- Many connectors (Grizzly, Jetty, Apache, …)
- Secure (SSL, Digest, Basic, OAuth, …)
- Various data bindings
- Filters
- **Reactive extensions**

Hotel booking example

- Get free hotels close to a specified location
  - find hotels within 5 kilometre radius
  - check hotel availability
  - get stored information about the hotel
  - get stored information about the user
  - return personalised list of available hotels
- Each call to a service takes 100ms

Hotel search service call dependency



Hotel search - synchronous client

- Easy and straightforward
- Latency for 10 results:
  - 100 + 10x100 + 10x100 + 100 + 100 = 2300 ms
- Executors to the rescue?

```
executorService.submit(()-> {
            Response searchResult = client.target("search").request().get();
            ...
        });
```

  - up to 21 threads handling 1 hotel search request
  - a lot of synchronisation required

Hotel search - asynchronous client

```
client.target("search").request().async()
    .get(new InvocationCallback<List<String>>() {
        public void completed(List<String> hotels) {
            for (String hotel : hotels) {
                client.target("hotelDetail").path(hotel).request().async()
                    .get(new InvocationCallback<Hotel>() {

                        public void completed(Hotel hotel) {
                            ...
                        }

                        public void failed(Throwable throwable) {
                            ...
                        }
                    });
                ...
            }
        }

        public void failed(Throwable throwable) {
            ...
        }
    });
```

Reactive client

- As fast as an async client
- Data flows
  - execution model propagates changes through the flow
- Event based
  - notify user code or another item in the flow continuation, error, completion
- Composable
  - compose/ transform flows into a resulting flow

Jersey reactive client libraries

- Java 8
  - CompletionStage, CompletableFuture
- Guava
  - ListenableFuture, Futures
- RXJava
  - Observable
  - Contributed by Netflix
  - Complicated but powerful

Hotel search - RX client

```
Observable<Destination> recommended = RxObservable.from(client.target("search"))
    .request()
    .rx()
    .get(new GenericType<List<String>>() {})
    .onErrorReturn(throwable -> {
        . . .
    })
    .flatMap(hotelId -> {
        Observable<Hotel> i = RxObservable.from(client.target(“hotelInfo")) . . .
        Observable<Boolean> available = RxObservable.from(. . .
        return Observable.zip(i, available, . . .)
    }
    .take(10)
    .toList()
    . . .
```

Summary

- When writing microservices in Java, JAX-RS is a natural choice to implement REST interface
- Jersey brings several non-standard options that might be handy:
  - Lightweight container support
  - Monitoring features (auto-scaling)
  - Powerful client (+ reactive extensions)
- There is more to come in future Jersey versions

Q/A