
ORACLE®

JavaOne™

ORACLE®

Client Orchestration and Reactive Programming in JAX-RS Applications

CREATE THE FUTURE

Michal Gajdos
Software Developer, Oracle

September, 2014

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.

Program Agenda

1. The Problem
2. The Why
3. The How
4. Beyond

The Problem

The Problem

Description

- Robust, Non-Uniform infrastructure of Services
  - Use different technologies – EJB, JMS, WS
  - Custom interfaces and data formats
    - Return data tailored for communicating inside the internal infrastructure
  - Not all are accessible from outside of the internal network
- WebApp that allows customers to communicate
  - Has access to the internal network, can communicate with desired services
- Goal: Expand to other platforms

The Problem

Example Application: Travel Agency

- 3 separate services inside the internal network
  - List visited destinations and recommend new places to visit
  - Obtain weather forecast for a destination
  - Obtain price calculation for a destination
- Data formats: JSON, XML
- Create an orchestration layer for an (mobile) application
  - list visited places and recommend new places for a user

The Why

The Why

Building an Orchestration Layer

- 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

The Why

Building an Orchestration Layer

- Flexible “Layer App” updates
- “Similar” user experience
  - Even for the older devices
- Security
  - No need to expose all services

The How

JAX-RS 2.0 and Jersey 2

JAX-RS Resources and Resource methods – Synchronous

```
@Path("agent/sync")
@Produces("application/json")
public class SyncAgentResource {

    @Uri("remote/destination")
    private WebTarget destination;

    @Uri("remote/calculation/from/{from}/to/{to}")
    private WebTarget calculation;

    @Uri("remote/forecast/{destination}")
    private WebTarget forecast;

    @GET
    public AgentResponse sync() {
        // ...
    }
}
```

JAX-RS Resources and Resource methods – Asynchronous

```
@Path("agent/completion")
@Produces("application/json")
public class CompletionStageAgentResource {

    @Uri("remote/destination")
    private WebTarget destination;

    @Uri("remote/calculation/from/{from}/to/{to}")
    private WebTarget calculation;

    @Uri("remote/forecast/{destination}")
    private WebTarget forecast;

    @GET
    public void completion(@Suspended final AsyncResponse async) {
        // ...
    }
}
```

JAX-RS Application

```
@ApplicationPath("/rx")
public class RxApplication extends javax.ws.rs.core.Application {

    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<>();
        classes.add(SyncAgentResource.class);
        // ...
        return classes;
    }

    @Override
    public Set<Object> getSingletons() { ... }

    @Override
    public Map<String, Object> getProperties() { ... }
}
```

Jersey Application

```
@ApplicationPath("/rx")
public class RxApplication extends ResourceConfig {

    public RxApplication() {
        // Agent Resources.
        packages("org.glassfish.jersey.examples.rx.agent");

        // Providers.
        register(JacksonFeature.class);
        register(ObjectMapperContextResolver.class);
    }
}
```

RxApplication

Exposed resources

- HTTP GET – **rx/agent/sync**
  - Response – application/json content type
- HTTP GET – **rx/agent/completion**
  - Response – application/json content type

JAX-RS 2.0 Client – Synchronous

```
Client client = ClientBuilder.newClient();

WebTarget rx = client.target("http://example.com/rx").register(JacksonFeature.class);

WebTarget forecasts = rx.path("remote/forecast/{destination}");

Forecast forecast = forecasts.resolveTemplate("destination", dest.getDestination())
                             .request("application/json")
                             .get(Forecast.class);
```

JAX-RS 2.0 Client – Asynchronous

```
Future<Forecast> forecast = forecasts.resolveTemplate("destination", d.getDestination())
                                     .request()
                                     .async()
                                     .get(new InvocationCallback<Forecast>() {
                                         @Override
                                         public void completed(Forecast forecast) {
                                             // Do Something.
                                         }

                                        @Override
                                        public void failed(Throwable throwable) {
                                            // Do Something else.
                                        }
                                     });
while (!forecast.isDone()) {
    // Do Something.
}
System.out.println(forecast.get());
```

Jersey 2

- JAX-RS 2.0 Reference Implementation
- Glassfish 4, WebLogic 12.1.3 (Jersey 1.18.1, Jersey 2.5.1), Standalone
- Other JAX-RS 2.0 implementations
  - RESTEasy – JBoss, WildFly
  - CXF

Orchestration Layer with JAX-RS

Example: Travel Agency

- REST API exposed using JAX-RS Resources and Resource Methods (Agent)
  - serving visited and recommended destinations
    - Including weather forecasts and price calculations for recommended places
  - JSON
- JAX-RS Clients obtaining all relevant data
  - Sync, Async, Rx
  - Combine the retrieved data to create a response

DEMO

Client - Synchronous Approach

- Easy to read, understand and debug
  - Simple requests
  - Composed requests
- Slow
  - Sequential processing even for independent requests
- Wasting resources
  - Waiting threads

DEMO

Client – Asynchronous Approach

Futures

- Return immediately after submitting a request
  - Future
- Harder to understand, debug
  - Especially when dealing with multiple futures
- Fast
  - Each request can run on a separate thread
  - Need to actively check for completion event (future.isDone()) or block (slow)

Client – Asynchronous Approach

The Callback Hell

- "Don't call us, we'll call you"
- Harder to read, understand and debug
  - Especially for composed calls (dependent)
- 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

DEMO

Beyond The Callback Hell

Rx JAX-RS Client

Client – Reactive Approach

- Data-Flows
  - Execution model propagates changes through the flow
- Asynchronous
  - Preferably, Speed
- Event-based
  - Notify user code or another item in flow about continuation, error, completion
- Composable
  - Compose/Transform multiple flows into the resulting one

Client – Reactive Approach

Libraries

- RxJava – Observable
- Java 8 – CompletionStage and CompletableFuture
- JSR166e – CompletableFuture (Java SE6, Java SE7)
- Guava – ListenableFuture and Futures

An Observable<Response> Example

```
Observable<Response> response = // ...

List<String> visited = new ArrayList<>(5);

// Read a list of destinations from JAX-RS response.
response.map(resp -> resp.readEntity(new GenericType<List<Destination>>() {}))
        // If an exception is thrown, continue with an empty list.
        .onErrorReturn(throwable -> Collections.emptyList())
        // Emit list of destinations one-by-one (as a new Observable).
        .flatMap(Observable::from)
        // Take the next 5 destinations.
        .take(5)
        // Obtain a string representation of a destination.
        .map(Destination::getDestination)
        // Observe on a separate thread.
        .observeOn(Schedulers.io())
        // Subscribe to callbacks – OnNext, OnError, OnComplete.
        .subscribe(visited::add, async::resume, () -> async.resume(visited));
```

Jersey Rx Client

Extension of JAX-RS Client

- Remember `#request()` and `#request().async()` ?
  - request() returns **Invocation.Builder**; **SyncInvoker** – sync HTTP methods
  - request().async() returns **AsyncInvoker** – async HTTP methods
- **`#rx()`** and **`#rx(ExecutorService)`**
  - Return an extension of RxInvoker
- Why an extension?

SyncInvoker and AsyncInvoker

```
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

```
@Beta public interface RxInvoker<T> {
    T get();
    <R> T get(Class<R> responseType);
    <R> T get(GenericType<R> responseType);
    // ...
}
@Beta public interface RxObservableInvoker extends RxInvoker<Observable> {
    Observable<Response> get();
    <T> Observable<T> get(Class<T> responseType);
    <T> Observable<T> get(GenericType<T> responseType);
    // ...
}
```

Jersey Rx Client – contd

Extension of JAX-RS Client

- Affected JAX-RS interfaces
  - RxClient<RX extends RxInvoker> extends Client
  - RxWebTarget<RX extends RxInvoker> extends WebTarget
  - RxInvocationBuilder<RX extends RxInvoker> extends Invocation.Builder
- Rx class
  - RxObservable
  - RxCompletionStage
  - RxListenableFuture
  - RxCompletableFuture (JSR 166e)

How to create Jersey Rx Client

Rx – Static Helper Methods returning RxClient<RX extends RxInvoker>

- newClient(Class<RX> invokerType)
- newClient(Class<RX> invokerType, ExecutorService executor)
- from(Client client, Class<RX> invokerType)
- from(Client client, Class<RX> invokerType, ExecutorService executor)
- from(WebTarget target, Class<RX> invokerType)
- from(WebTarget target, Class<RX> invokerType, ExecutorService executor)

How to create Jersey Rx Client

RxObservable – Static Helper Methods returning RxClient<RxObservableInvoker>

- newClient()
- newClient(ExecutorService executor)
- from(Client client)
- from(Client client, ExecutorService executor)
- from(WebTarget target)
- from(WebTarget target, ExecutorService executor)

DEMO

Resources

JAX-RS and Jersey

- JAX-RS Client API
  - https://jax-rs-spec.java.net/nonav/2.0/apidocs/overview-summary.html
  - https://jersey.java.net/documentation/latest/client.html
- Jersey Rx Client
  - https://github.com/jersey/jersey/tree/master/incubator/rx/rx-client
  - https://github.com/jersey/jersey/tree/master/incubator/rx/rx-client-guava
  - https://github.com/jersey/jersey/tree/master/incubator/rx/rx-client-java8
  - https://github.com/jersey/jersey/tree/master/incubator/rx/rx-client-jsr166e
  - https://github.com/jersey/jersey/tree/master/incubator/rx/rx-client-rxjava

Resources

Example and Libraries

- 3rd party libraries
  - https://code.google.com/p/guava-libraries/
  - https://github.com/ReactiveX/RxJava
  - http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
  - http://gee.cs.oswego.edu/dl/concurrency-interest/index.html
- Example (JDK7)
  - https://github.com/jersey/jersey/tree/master/examples/rx-client-webapp

Related Talks

- New and Noteworthy in Jersey 2 [CON3782]
  - Wednesday, Oct 1, 11:30 AM - 12:30 PM - Parc 55 - Mission

Questions & Answers

Safe Harbor Statement

The preceding 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.

Hardware and Software
Engineered to Work Together

JavaOne™

ORACLE®


ORACLE®