ORACLE

Reactive Java EE - Let Me Count the Ways!

Reza Rahman
Java EE/GlassFish Evangelist
Reza.Rahman@Oracle.com
@reza_rahman

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

- What Exactly is Reactive?
- Touring Reactive in Java EE
- Bearable Reactive with Java SE 8?

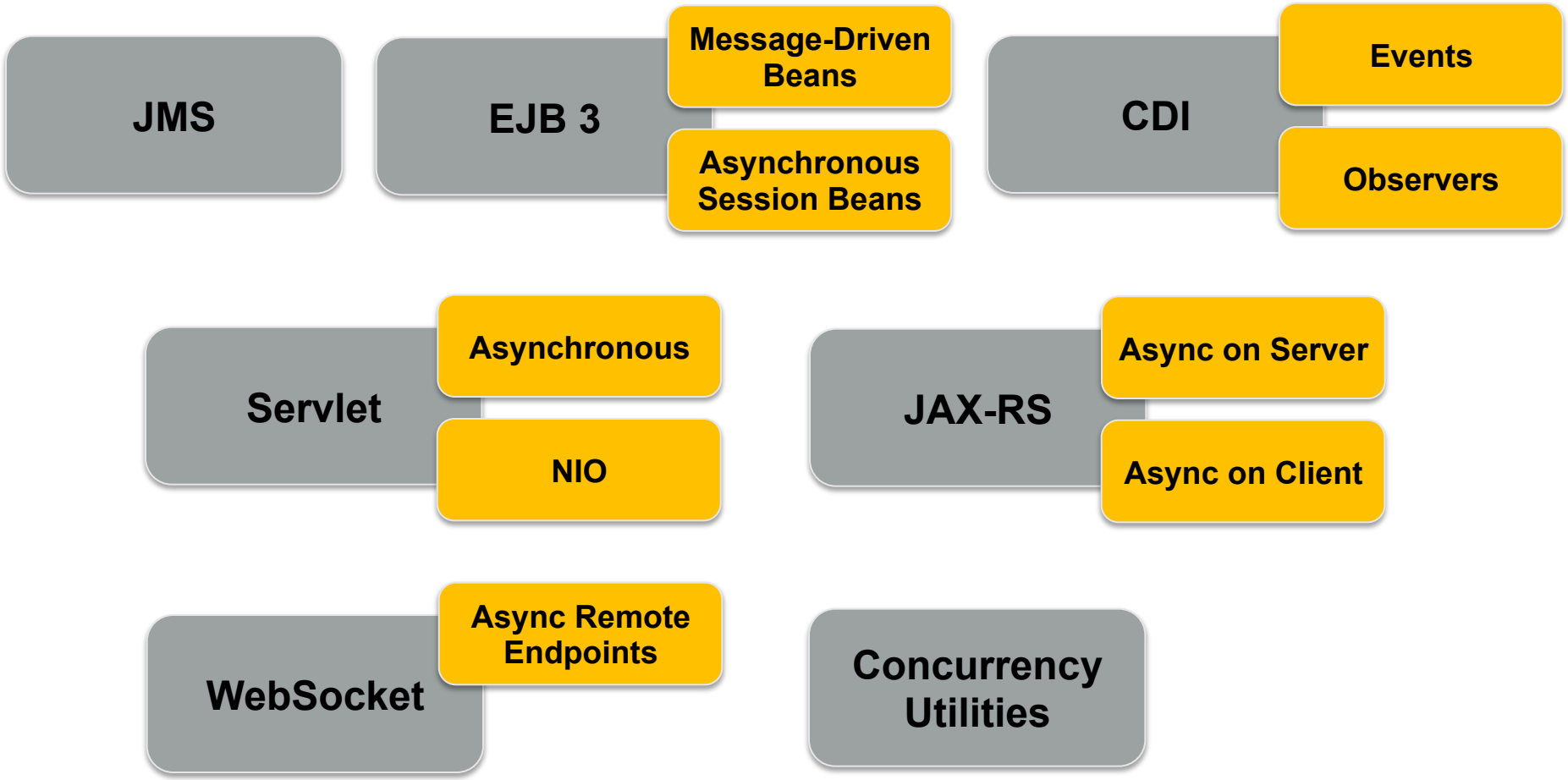
Reactive: What's in a Name?

- "Reactive" fairly old but incredibly vague term
  - A big hurdle to broad adoption by average developers
- Sound core principals perhaps co-opted by marketing concerns?
  - Event driven
  - Asynchronous
  - Non-blocking
  - Message driven
- Some overloaded concerns to very simple core principals attempted to be added on more recently
  - Responsive, resilient, elastic, adaptive, fault-tolerant, scalable, etc
  - These really don't have that much to do with Reactive concepts proper
  - Long met by Java EE runtimes

What's the Big Deal?

- Reactive has always been an important software engineering technique!
  - More responsive user experience
  - High throughput, optimal hardware/CPU utilization
  - Loose coupling, complex event processing
- Will potentially become even more important
  - Internet of Things (IoT), device-to-device communication
  - Mobile, large global concurrent user bases, more chatty applications
- Not necessarily a panacea
  - Asynchronous, event driven code is always harder to write, maintain than synchronous, blocking code
  - Horizontal/hardware scalability can be a cheaper/more maintainable answer

Reactive Java EE



JMS and Message Driven Beans

- JMS one of the oldest APIs in Java EE, strongly aligned with Reactive principles
  - Message oriented middleware
  - Message/event driven, asynchronous, loosely coupled, reliable, transactional, durable, fault tolerant, error tolerant, clustered
- Message Driven Beans primary vehicle for JMS message handling in Java EE
  - Just POJOs with annotations (metadata)
  - Transactional, thread-safe, pooled, reliable, load-balanced, fault-tolerant, error-tolerant

JMS Send

```
@Inject JMSContext jmsContext;
@Resource(lookup = "jms/HandlingEventRegistrationAttemptQueue")
Destination handlingEventQueue;
...
public void receivedHandlingEventRegistrationAttempt(
    HandlingEventRegistrationAttempt attempt) {
  ...
  jmsContext.createProducer()
      .setDeliveryMode(DeliveryMode.PERSISTENT) // The default :-)
      .setPriority(LOW_PRIORITY)
      .setDisableMessageID(true)
      .setDisableMessageTimestamp(true)
      .setStringProperty("source", source)
      .send(handlingEventQueue, attempt);
}
```

JMS MDB

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(propertyName = "destinationType",
    propertyValue = "javax.jms.Queue"),
  @ActivationConfigProperty(propertyName = "destinationLookup",
    propertyValue = "jms/HandlingEventRegistrationAttemptQueue"),
  @ActivationConfigProperty(propertyName = "messageSelector",
    propertyValue = "source = 'mobile'")})
public class HandlingEventRegistrationAttemptConsumer
    implements MessageListener {
  ...
  public void onMessage(Message message) {
     ...
     HandlingEventRegistrationAttempt attempt
         = message.getBody(HandlingEventRegistrationAttempt.class);
     ...
  }
}
```

Even Better with JMS 2.1?

```
@ApplicationScoped
@MaxConcurrency(10)
public class HandlingEventRegistrationAttemptConsumer {
  @JmsListener(
     destinationLookup="jms/HandlingEventRegistrationAttemptQueue",
     selector="source = 'mobile'",
     batchSize=10, retry=5, retryDelay=7000,
     orderBy=TIMESTAMP)
  public void onEventRegistrationAttempt(
      HandlingEventRegistrationAttempt... attempts) {
    ...
  }
}
```

Asynchronous Session Beans

- Dead simple asynchrony at the component level
  - Just an annotation on a POJO
  - Simple return type: void (fire-and-forget) or Future<V> (client processes asynchronous results)
- Great when all that is required is greater throughput or responsiveness
  - Still transactional, thread-safe, pooled
  - Not loosely coupled, persistent, fault tolerant or error tolerant (client must explicitly handle errors)

Asynchronous Session Bean

```
@Stateless
public class ReportGeneratorService {
  @Asynchronous
  public Future<Report> generateReport(ReportParameters params){
    try{
      Report report = renderReport(params);
      return new AsyncResult(report);
    } catch(ReportGenerationException e) {
      return new AsyncResult(new ErrorReport(e));
    }
```

```
@Asynchronous
public void processPayment(Payment payment){
  // CPU/IO heavy tasks to process a payment
}
```

Asynchronous Session Bean Client

```
@Inject ReportGeneratorService reportGeneratorService;
...
Future<Report> future =
    reportGeneratorService.generateReport(parameters);
...
if (future.isDone()) {
    Report report = future.get();
    ...
}
...
future.cancel(true);
```

@Asynchronous + CompletableFuture?

```
@Asynchronous
public CompletableFuture<Confirmation> processPayment(
    Order order) {
  ...
  Confirmation status = ...;
  return
    CompletableFuture<Confirmation>.completedFuture(status);
}
```

```
paymentService
    .processPayment(order)
    .thenAccept(
        confirmation -> System.out.println(confirmation));
```

CDI Events/Observers

- Compact, simple, elegant, type-safe events
  - Essentially the observer pattern formalized via a DI framework and annotations
- Offers excellent solution to loose-coupling and type-safe filtering/ chaining, but not much else including asynchrony (not yet anyway)
  - Can be one-to-one or one-to-many

CDI Events

```
@Inject @CargoInspected Event<Cargo> cargoInspected;
...
public void inspectCargo(TrackingId trackingId) {
   ...
   cargoInspected.fire(cargo);
}
```

```
public void onCargoInspected(
     @Observes @CargoInspected Cargo cargo) {
```

```
@Qualifier
@Retention(RUNTIME) @Target({FIELD, PARAMETER})
public @interface CargoInspected {}
```

Asynchronous CDI Events?

```
@Inject @CargoInspected Event<Cargo> cargoInspected;
...
public void inspectCargo(TrackingId trackingId) {
   ...
   cargoInspected.fireAsync(cargo);
}
```

```
public void onCargoInspected(
     @Observes(async=true) @CargoInspected Cargo cargo) {
```

Asynchronous Servlets and NIO

- Asynchronous Servlets maximize throughput/thread utilization
  - Decouple connection from request thread
  - Return request thread back to pool
  - Handle IO/CPU heavy work on separate backend thread
  - Close cached connection when done
- NIO removes possible thread blocks during slow read/write
  - Get notified when the IO channel might be ready
  - Only read/write when IO channel is ready
  - Obvious need when Servlet IO is particularly heavy, otherwise a very complex solution

Asynchronous Servlet

```
@WebServlet(urlPatterns={"/report"}, asyncSupported=true)
public class AsyncServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
      HttpServletResponse response) {
    ...
    final AsyncContext asyncContext = request.startAsync();
    asyncContext.start(() -> {
      ReportParameters parameters =
          parseReportParameters(asyncContext.getRequest());
      Report report =  generateReport(parameters);
      printReport(report, asyncContext.getResponse());
      asyncContext.complete();
    });
  }
}
```

Asynchronous Servlet NIO (Output Stream)

```
private void printReport(Report report,
    AsyncContext context) {
  ServletOutputStream output =
      context.getResponse().getOutputStream();
  WriteListener writeListener = new ReportWriteListener(
      output, report, context);
  output.setWriteListener(writeListener);
}
```

Asynchronous Servlet NIO (Write Listener)

```
class ReportWriteListener implements WriteListener {
  private ServletOutputStream output = null;
  private InputStream input = null;
  private AsyncContext context = null;

  ReportWriteListener(ServletOutputStream output, Report report,
      AsyncContext context) {
    this.output = output;
    this.input = report.asPdfStream();
    this.context = context;
  }
  ...
```

Asynchronous Servlet NIO (Write Listener)

```
  ...
  public void onWritePossible() throws IOException {
    byte[] chunk = new byte[256];
    int read = 0;
    while (output.isReady() && (read = input.read(chunk)) != -1)
      output.write(chunk, 0, read);

    if (read == -1)
      context.complete();
  }

  public void onError(Throwable t) {
    context.complete();
    t.printStackTrace();
  }
}
```

Asynchronous JAX-RS

- Asynchronous capabilities newly added to JAX-RS 2/Java EE 7
  - Both on the server and client side
- Server-side essentially identical to Servlet 3 async
  - Nicer declarative syntax
- Client API async capabilities very symmetric to synchronous API
  - Both Futures and callbacks supported
- No NIO yet, but could be included in Java EE 8

Asynchronous JAX-RS Resource

```
@Stateless
@Path("/reports")
public class ReportsResource {
   ...
   @Path("{id}")
   @GET
   @Produces({"application/pdf"})
   @Asynchronous
   public void generateReport(
       @PathParam("id") Long id,
       @Suspended AsyncResponse response) {
     ResponseBuilder builder =
Response.ok(renderReport(id));
     builder.header("Content-Disposition",
         "attachment; filename=report.pdf");
      response.resume(builder.build());
   }
}
```

Asynchronous JAX-RS Client

```
WebTarget target = client.target("http://.../balance")...

Future<Double> future = target.request()
                              .async().get(Double.class));
...
Double balance = future.get();
```

```
WebTarget target = client.target("http://.../balance")...

target.request().async().get(
    new InvocationCallback<Double>() {
        public void complete(Double balance) {
          // Process balance
        }
        public void failed(InvocationException e) {
          // Process error
        }
    });
```

Asynchrony/NIO in WebSocket

- WebSocket endpoints are inherently non-blocking/event-driven!
  - There's no thread-connection association in the first place
  - True for server and client side
- Writes/sends can be made asynchronously for better throughput
  - Very symmetric API for both sync and async
  - Futures or callbacks supported
  - Good idea to use asynchronous send in most cases!

Asynchronous Remote WebSocket Endpoint

```
@ServerEndpoint(value = "/chat"...)
@Singleton
public class ChatServer {
  private Set<Session> peers = new HashSet<>();

  @OnOpen
  public void onOpen(Session peer) {
    peers.add(peer);
  }

  @OnClose
  public void onClose(Session peer) {
    peers.remove(peer);
  }

  @OnMessage
  public void onMessage(ChatMessage message) {
    for (Session peer : peers) {
      ...peer.getAsyncRemote().sendObject(message)...
    }
  }
}
```

Future vs Callback

```
Future<Void> future =
    peer.getAsyncRemote().sendObject(message);
```

```
peer.getAsyncRemote().sendObject(message,
    (SendResult result) -> {
      ...
      if (!result.isOK()) {
        ...result.getException()...
      }
      ...
    });
```

Java EE Concurrency Utilities

- Allows for lower-level threading/asynchronous capabilities in Java EE in a safe, reliable, managed fashion
  - Very specialized code, custom workloads
- Fairly small extension of Java SE Concurrency Utilities
  - ManagedExecutorService
  - ManagedThreadFactory

Managed Executor Service

```
@Path("/reports")
public class ReportsResource {
  @Resource ManagedExecutorService executor;
  ...
  @Path("{id}")
  @GET
  @Produces({"application/pdf"})
  public void generateReport(
      @PathParam("id") Long id,
      @Suspended AsyncResponse response) {
    executor.execute(() -> {
      ResponseBuilder builder = Response.ok(renderReport(id));
      builder.header("Content-Disposition",
          "attachment; filename=report.pdf");
      response.resume(builder.build());
    }
  }
}
```

Executor Service API

```
public interface ManagedExecutorService
    extends ExecutorService {
  public void execute(Runnable command);

  public <T> Future<T> submit(Callable<T> task);
  public Future<?> submit(Runnable task);
  public <T> Future<T> submit(Runnable task, T result);
  ...
}
```

Java SE 8 Completable Future

- Futures and callbacks both have serious flaws
  - Especially when it comes to significantly Reactive code
- Java SE 8 CompletableFuture significantly better for Reactive programming
  - Non-blocking, event-driven, composable and functional (via lambdas)
- Easy to integrate with Java EE 7 managed executors
  - Should Java EE (and Java SE) embrace CompletableFuture more uniformly?

Looks are Deceiving…

```
Person p = ...
Assets assets = getAssets(p);
Liabilities liabilities = getLiabilities(p);
Credit credit = calculateCreditScore(assets, liabilities);

History history = getHealthHistory(p);
Health health = calculateHeathScore(history);

Coverage coverage = underwrite(credit, health);
```

The Problem with Futures (and Callbacks)

```
Person p = ...
Future<Assets> f1 = executor.submit(() -> getAssets(p));
Future<Liabilities> f2 = executor.submit(
    () -> getLiabilities(p));
Future<Credit> f3 = executor.submit(
    () -> calculateCreditScore(f1.get(), f2.get()));

// The unrelated calls below are now blocked for no reason.
Future<History> f4 = executor.submit(
    () -> getHealthHistory(p));
Future<Health> f5 = executor.submit(
    () -> calculateHeathScore(f4.get()));


// Unrelated paths join below.
Future<Coverage> f6 = executor.submit(
    () -> underwrite(f3.get(), f5.get()));
```

Callbacks don't block, but introduce callback hell…

https://github.com/m-reza-rahman/reactive_javaee/blob/master/CallbackHell.java

CompletableFuture Basics

```
@Resource ManagedExecutorService executor;
...
public CompletableFuture<Confirmation> processPayment(Order order) {
  CompletableFuture<Confirmation> future = new CompletableFuture<>();
  executor.execute(() -> {
    Confirmation status = ...
    future.complete(status);
  });
  return future;
}
```

```
paymentService
    .processPayment(order)
    .thenAccept(
        confirmation -> System.out.println(confirmation));
```

Functional Reactive to the Rescue?

```
CompletableFuture<Assets> getAssets =
    CompletableFuture.supplyAsync(
        () -> getAssets(person), executor);
CompletableFuture<Liabilities> getLiabilities =
    CompletableFuture.supplyAsync(
        () -> getLiabilities(person), executor);
CompletableFuture<Credit> calculateCreditScore =
    getAssets.thenCombineAsync(getLiabilities,
        (assets, liabilities) ->
            calculateCreditScore(assets, liabilities), executor);

CompletableFuture<Health> calculateHeathScore =
    CompletableFuture.supplyAsync(
        () -> getHealthHistory(person), executor)
            .thenApplyAsync(
                history -> calculateHeathScore(history), executor);

Coverage coverage =
    calculateCreditScore.thenCombineAsync(calculateHeathScore,
        (credit, health) -> underwrite(credit, health), executor)
            .join();
```

More Possibilities…

- Reactive JPA?
  - Last major reactive frontier for Java EE?
  - Async/NIO support in underlying database driver/JDBC
  - A specialized thread pool might help in the meanwhile?

```
CompletableFuture<List<Country>> countries =
    em.createQuery("SELECT c FROM Country c",
Country.class)
        .async().getResultList();
```

- Reactive MVC?
  - Similar to basic model in JAX-RS/Servlet
  - EJB async another possible model
  - Reactive JSF conceptually tough

Summary

- Reactive programming well established technique, may be more important in the near future
- Java EE has long had rich support for the Reactive model
- Things could be improved even more with Java EE 8
- Java SE 8 helps quite a bit to make the programming model easier
- Beyond Java EE application servers provide clustering, load-balancing, replication, failover, bandwidth throttling, resource pooling, thread pooling, caching
- Be careful – Reactive is not an easy approach to take

Resources

- Java EE Tutorials
  - http://docs.oracle.com/javaee/7/tutorial/doc/home.htm
- Java SE Tutorials
  - http://docs.oracle.com/javase/tutorial/
- Digging Deeper
  - http://docs.oracle.com/javaee/7/firstcup/doc/home.htm
  - https://glassfish.java.net/hol/
  - http://cargotracker.java.net
- Java EE Transparent Expert Groups
  - http://javaee-spec.java.net
- Java EE Reference Implementation
  - http://glassfish.org
- The Aquarium
  - http://blogs.oracle.com/theaquarium


Java™