



**ORACLE<sup>®</sup>**

## **Dealing with Asynchronicity in Java™ Technology-Based Web Services**

Gerard M Davison, Senior Principal Software Engineer

Oracle JDeveloper

[gerard.davison@oracle.com](mailto:gerard.davison@oracle.com)



Learn how to invoke and implement  
Asynchronous Services using JAX-\*

(No not AJAX, Comet etc)

# Agenda

- Introduction
- Client side asynchrony
- Server side asynchrony
- Implementation of clients and services
- Declarative asynchronous services
- Other improvements
- Q & A



# Introduction

# Introduction

## Synchronous Call

```
Service service = ...;  
StockQuote quoteService =  
    service.getStockQuote();  
  
float quote =  
    quoteService.getPrice(ticker);
```

# Introduction

## Asynchronous Call

```
Service service = ...;  
StockQuote quoteService =  
    service.getStockQuote();  
  
quoteService.getPriceAsync(ticker);  
  
// Do some other work  
  
// How and when do we get the response?
```

# Introduction

## Why Asynchronous?

- The world is fundamentally asynchronous
- Improvement in ..
  - User's experience
  - Reliability in distributed applications
  - Scalability
- Only way to go in long running processes
- Implementation
  - Client side asynchrony
  - Server side asynchrony

# Introduction

## Alternatives to JAX- \*

- BPEL
  - Integration of other operations
  - Can be re-exposed as an asynchronous web service
- Asynchronous 3.1 EJB™ architecture
  - Java only
- JMS, MQueue, Native Database Queues
  - Generally platform specific
- SMTP
  - Golden oldie
  - Can be used with web services
- SCA



# Client Side Asynchrony

# Client Side Asynchrony

## JAXWS 2.x API

- Calling synchronous services in asynchronous way
- Not really synchronous though
- WSDL to Java customization option
  - Synchronous
  - Asynchronous
- Asynchronous Interface
  - Polling
  - Callback
- No global control over thread pooling, can set per Service instance.

# Client Side Asynchrony

## JAXWS 2.x API

- Enable asynchronous interface generation by
  - `<jaxws:enableAsyncMapping> true</jaxws:enableAsyncMapping>`
- Enable for
  - Definition
  - PortType
  - Operation
- `enableAsyncMapping` goes in ..
  - WSDL file
  - External customization file

# Client Side Asynchrony

## Synchronous Interface

```
@WebService
```

```
public interface StockQuote {  
    float getPrice(String ticker);  
}
```

# Client Side Asynchrony

## Asynchronous Interface

```
@WebService
```

```
public interface StockQuote {  
    float getPrice(String ticker);
```

```
    Response<Float>
```

```
        getPriceAsync(String ticker);
```

```
}
```

# Client Side Asynchrony

## Polling Style

```
StockQuote quoteService =  
    (StockQuote)service.getPort(portName);  
  
Response<Float> response =  
    quoteService.getPriceAsync(ticker);  
  
// do something  
while (!response.isDone()) {  
    // do something while we wait  
}  
float quote = response.get();
```

# Client Side Asynchrony Response

- Package: `javax.xml.ws`
- Extends: `java.util.concurrent.Future < V >`
- Methods on Response:
  - `Map<String, Object> getContext();`
- Methods on Future:
  - `boolean isDone();`
  - `V get();`
  - `V get(long timeout, TimeUnit unit)`
  - `boolean cancel(boolean mayInterruptIfRunning)`
  - `boolean isCancelled();`

# Client Side Asynchrony

## Asynchronous Interface

```
@WebService
```

```
public interface StockQuote {  
    float getPrice(String ticker);
```

```
    Response<Float> getPriceAsync(String  
        ticker);
```

```
    Future<?> getPriceAsync(String ticker,  
        AsyncHandler<Float>  
        responseReceiver);
```

```
}
```

# Client Side Asynchrony Callback Style

```
PriceReceiver priceReceiver = new
    PriceReceiver();
quoteService.getPriceAsync(ticker,
    priceReceiver);
..
class PriceReceiver implements
    AsyncHandler<Float> {
    public void handleResponse(Response<Float>
        response) {
        Float price = response.get();
        // do something with the result
    }
}
```



# Server Side Asynchrony

# Server Side Asynchrony

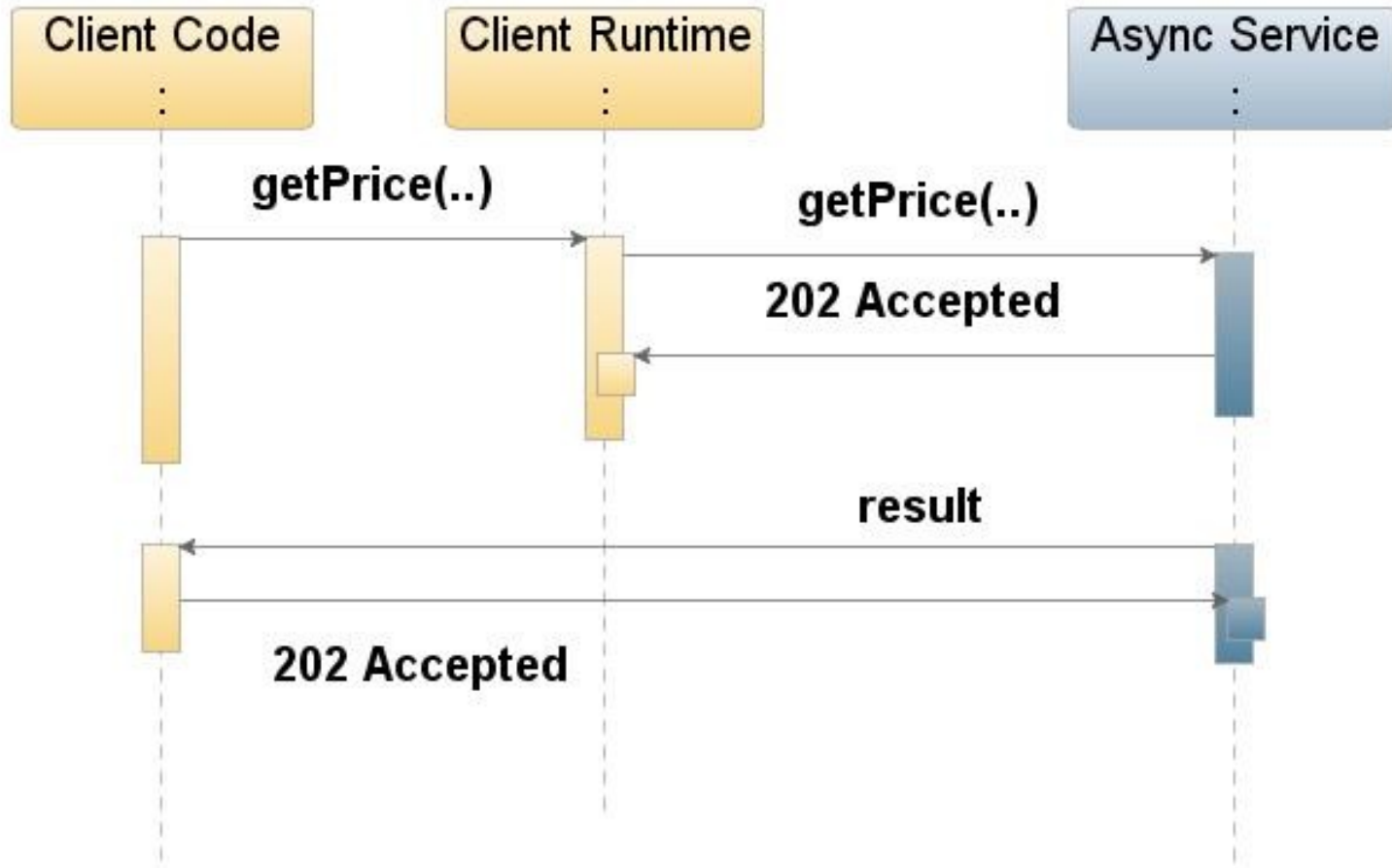
- Invocation and response separated
- A new connection for the call and the response
- The invoker and the receiver don't have to be on the same machine
- How it is implemented depends on the binding used

# Server Side Asynchrony

## Web Service Bindings

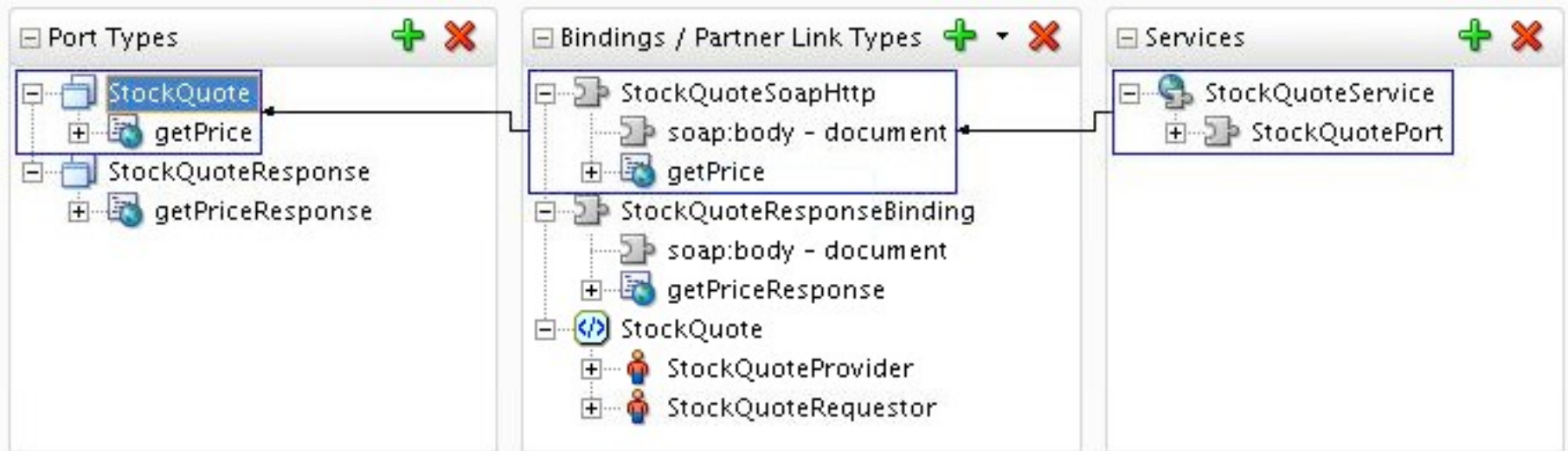
- Using HTTP protocol
  - Can do polling be inefficient
  - Call and response are separated
  - But we need to layer something on top, WS-\*
- Using JMS
  - Natively designed for asynchrony
  - Problems with firewalls
  - Only works for Java-2-Java
- Using SMTP
  - Useful in disconnected cases
  - Use in conjunction with WS-RM
  - Implementation for Java environment and .NET

# Server Side Asynchrony HTTP



# Server Side Asynchrony HTTP

- Defined with two port types, two binding; but one service



- Can use partnerLinkType to relate the two different ports
- Part of the BPEL specification; but an extension to WSDL

# Server Side Asynchrony HTTP WS- Addressing

- Transport agnostic SOAP message delivery
- Endpoints references and message information headers
- Message information headers
  - Allow you to relate the message to a particular instance
- Endpoint references
  - Tell you where to send the message
- They can tell you where to reply to and send faults to, important for asynchronous services

# Server Side Asynchrony HTTP WS- Addressing Request

```
<soap:Envelope>  
  <soap:Header>  
    <wsa:MessageID>uuid:35f19ca8-c9fe</wsa:MessageID>  
    <wsa:Action>http://lh:80/request/...</wsa:Action>  
    <wsa:ReplyTo>  
      <wsa:Address>http://lh:77/response</wsa:Address>  
    </wsa:ReplyTo>  
    <wsa:To>http://lh:80/request</wsa:To>  
  </soap:Header>  
  <soap:Body>...</soap:Body>  
</soap:Envelope>
```

# Server Side Asynchrony HTTP WS- Addressing Response

```
<soap:Envelope>
  <soap:Header>
    <wsa:To>http://lh:77/response</wsa:To>
    <wsa:Action>urn:response</wsa:Action>
    <wsa:MessageID>uuid:cb383139-cdf2</wsa:MessageID>
    <wsa:RelatesTo>uuid:35f19ca8-c9fe</wsa:RelatesTo>
  </soap:Header>
  <soap:Body>...</soap:Body>
</soap:Envelope>
```



# **Server Side Asynchrony Client Implementation**

# Server Side Asynchrony Client Implementation

- Create a proxy for the service
- Implement the callback binding to receive the response
- Deploy the callback web service
  
- Instantiate a proxy for the initiation endpoint.
- Set the **MessageId** property to a new unique value
- Set the **To**; **ReplyTo**; and **FaultTo** URI
- Invoke the web service
  
- In the callback web service use the “**RelatesTo**” header to correlate the response

# Server Side Asynchrony Initiator Implementation

```
StockQuote sq = service.getStockQuote();

// Populate headers

AddressingVersion av = AddressingVersion.W3C;
WSBindingProvider wsbp = (WSBindingProvider)sq;
WSEndpointReference replyTo = new WSEndpointReference(
    "http://lh:77/response", av);
String uuid = "uuid:" + UUID.randomUUID();

wsbp.setOutboundHeaders(
    new StringHeader(av.messageIDTag, uuid),
    new StringHeader(av.toTag, "http://lh:88/..."),
    new StringHeader(av.toAction, "..."),
    replyTo.createHeader(av.replyToTag);

sq.getPrice("ORCL");
```

# Server Side Asynchrony Callback Implementation

```
// Deal with response
```

```
@Resource WebServiceContext wsContext;
```

```
public void getPriceResponse(String _return)
```

```
{  
    HeaderList hl = wsContext.getMessageContext().get(  
        JAXWSProperties.INBOUND_HEADER_LIST_PROPERTY);  
    Header h = hl.get(AddressingVersion.W3C.relatesToTag());  
    String relatesToMessageId = h.getStringContents();
```

```
    ...  
}
```

# Server Side Asynchrony FaultTo Implementation

- This should be easy right?
- Just create a `@WebServiceProvider`?
- Sadly there is a bug in JAX-WS RI (Issue 585)
- For the moment you have use a Servlet.

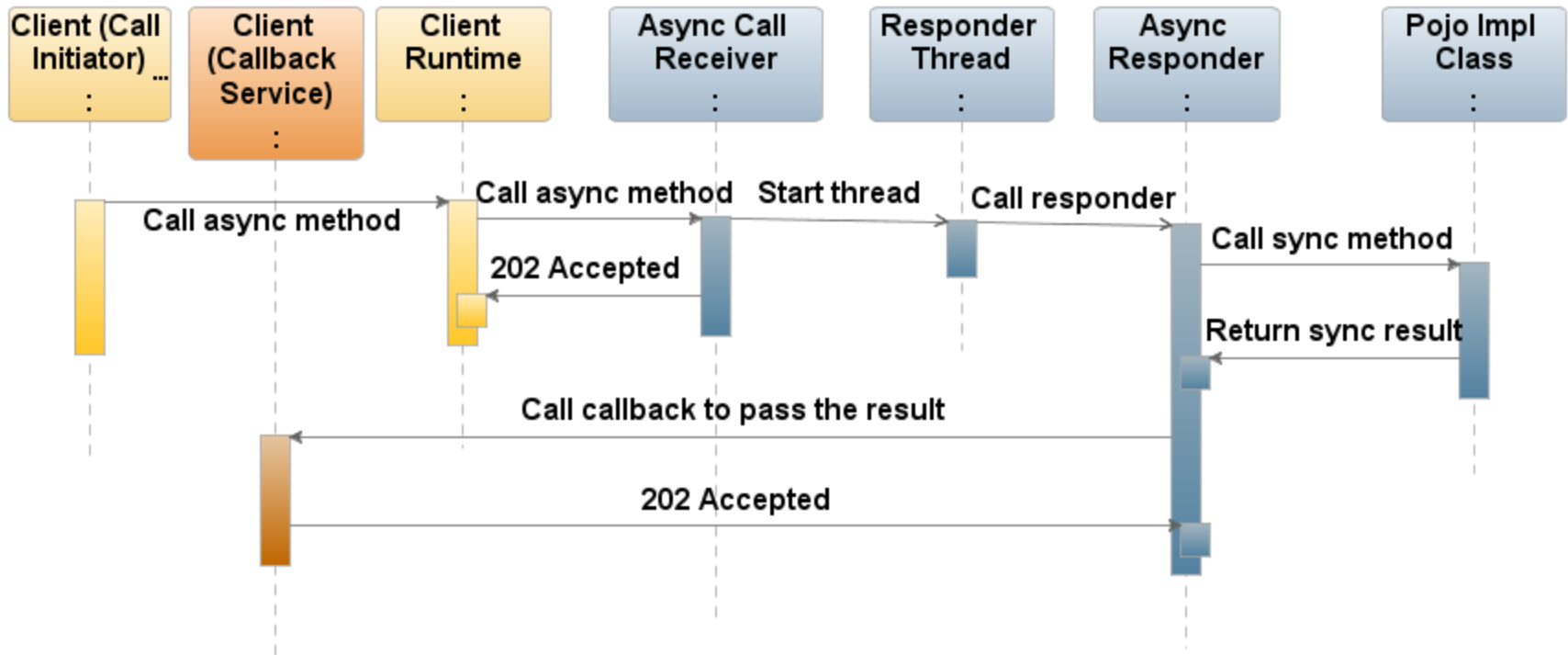


# **Server Side Asynchrony Service Implementation**

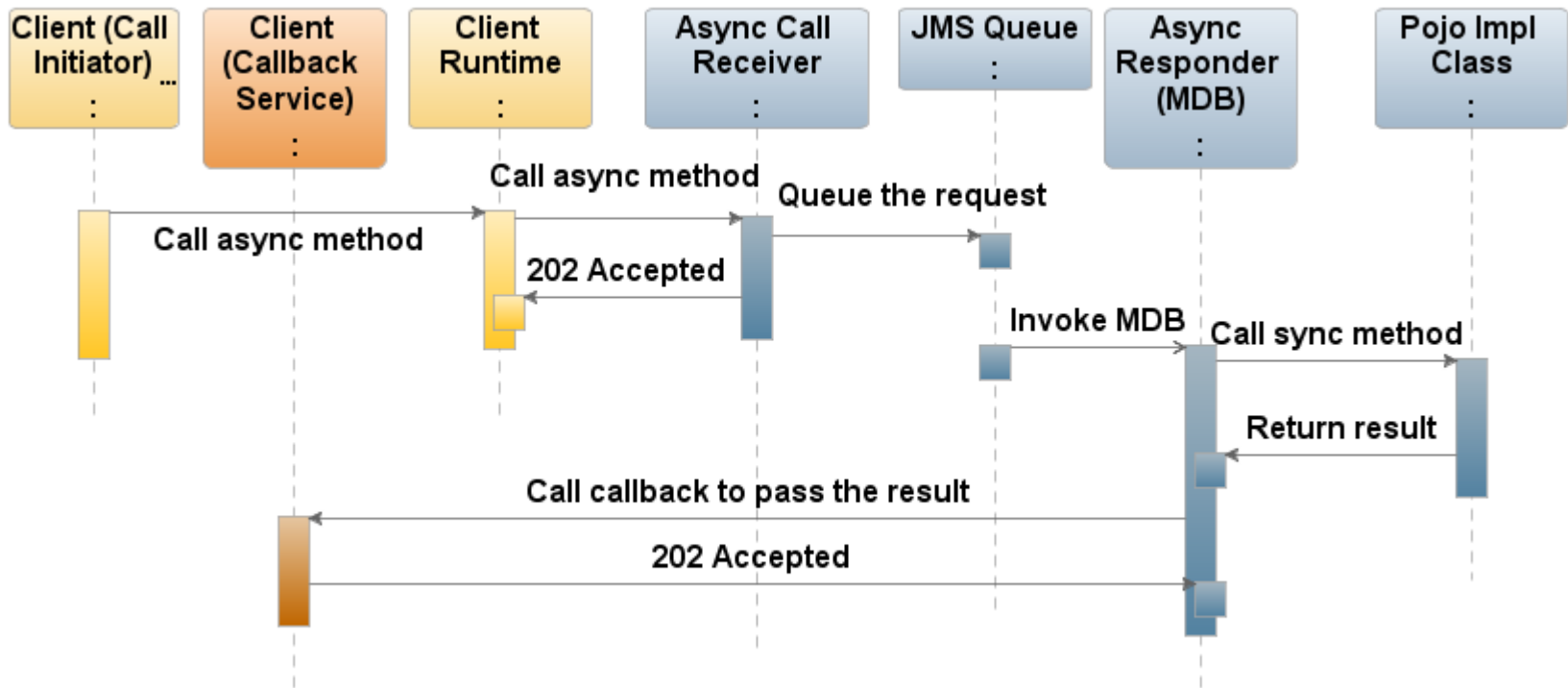
# Server Side Asynchrony HTTP Service Implementation

- Accept the asynchronous request and save it (where?)
- Return 202 Accepted
- Pick the request and process it
- Invoke the callback service and pass the result
- JAX-WS RI AsyncProvider doesn't solve all of the issues

# Implementation of asynchronous services Using Threads



# Implementation of asynchronous services Using JMS



# Implementation of asynchronous services

## Implementing the callback

```
// Use the servlet context to reference the wsdl  
// then create a new service/port dynamically
```

```
ServletContext sc = _context.getMessageContext()  
    .get("javax.xml.ws.servlet.context");
```

```
Service s = Service.create(  
    sc.getResource("/wsdl/StockQuote.wsdl")  
    new QName("http://com.stock/",  
        "StockQuoteResponseService"));
```

```
StockQuoteResponse sqr = s.getPort(  
    new QName("http://com.stock/",  
        "StockQuoteResponse"),  
    StockQuoteResponse.class);
```

# Implementation of asynchronous services

## Implementing the callback

```
// Change the endpoint address to the replyTo

Map requestProperties = sgr.getRequestContext();
requestProperties.put(
    ENDPOINT_ADDRESS_PROPERTY, replyTo);

// Populate WS-Addressing headers, relates to etc

((WSBindingProvider)sgr).setOutboundHeaders(
    new StringHeader(av.messageIDTag, "uuid:" + UUID.ran...
    ()),
    new StringHeader(av.toTag, replyTo),
    new StringHeader(av.toAction, "..."),
    new StringHeader(av.relatesTo, incomingMessageId));

sgr.getPriceResponse(stockPrice);
```



# **Server Side Asynchrony Declarative Proposal**

# Declarative Asynchronous Services

- Implementing an asynchronous service is not trivial
- But the basic concepts are....
- .... so how about a declarative model?
- We have been working on something in Oracle iAS 11, based on the JMS model
- We would like propose this for the JAX-WS standard

# Declarative Asynchronous Services

## Simple Return Case

```
package com.stock;

import ...;

@WebService
@AsyncWebService
@Addressing
public class StockQuote {
    public float getPrice(String ticker) {
        return 100;
    }
}
```

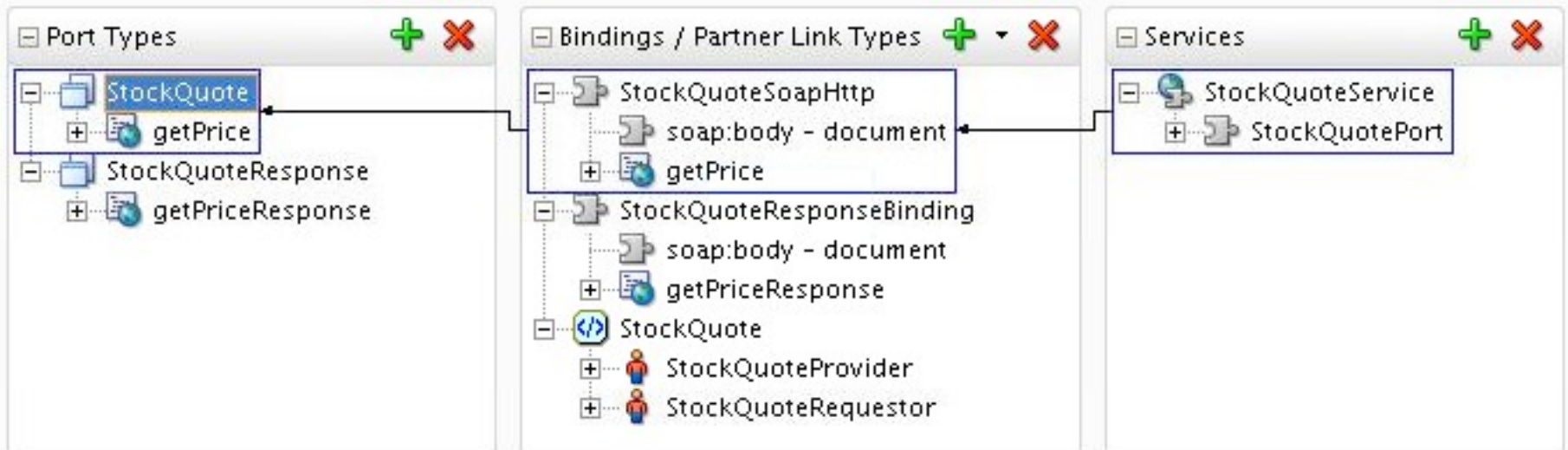


Yes Just One  
Annotation

# Declarative Asynchronous Services

## Simple Return Case

- The resultant WSDL, as before



# Declarative Asynchronous Services Callback Case

```
@WebService
@AsyncWebService
@Addressing
@CallbackInterface(StockQuoteResponse.class)
public class StockQuote {

    @CallbackRef StockQuoteResponse response;

    @OneWay public void getPrice(String ticker) {
        if ("ORCL".equals(ticker))
            response.stockPrice(100);
        else
            response.alternatives(new String[] {"ORCL"});
    }
}
```

# Declarative Asynchronous Services Callback Case

```
package com.stock;

import ...;

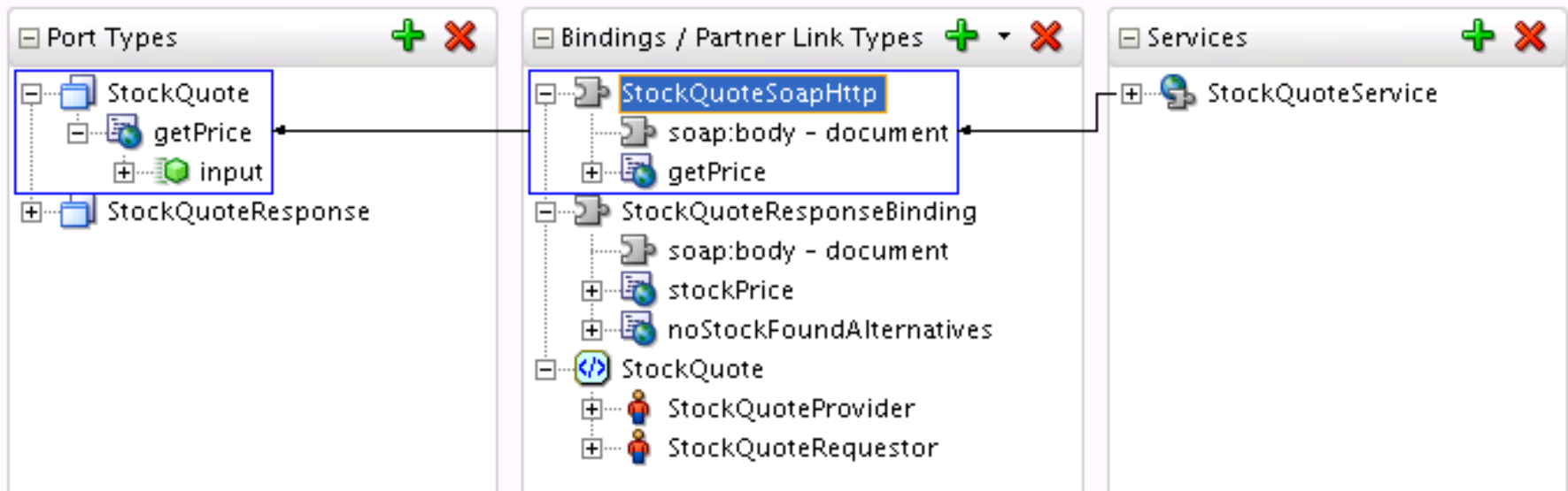
@WebService
public interface StockQuoteResponse {

    public void stockPrice(float price);
    public void alternatives(String alternatives[]);

}
```

# Declarative Asynchronous Services Callback Case

- The resultant WSDL



# Declarative Asynchronous Services

## Details

- **@CallbackMethod**, can be used to make a method synchronous and control response details
- **@AsyncWebServiceQueue**, override the default JMS queue
- Can have a mix of the return and callback styles
- Policy Annotations
  - Defined for the initiation service and for the callback invocation
  - Problem when creating ad-hoc relationships
- Weblogic has the **@Buffered/ @Callback** in RPC



# **Demo Declarative Asynchronous Service**

## **Oracle JDeveloper/ iAS**

# Declarative Asynchronous Services

## Partial RI implementation

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@WebServiceFeatureAnnotation(
    id=AsynchronousFeature.ID,
    bean=AsynchronousFeature.class)
@InstanceResolverAnnotation(
    AsynchronousInstanceResolver.class)
public @interface Asynchronous {
}
```

# Declarative Asynchronous Services

## Partial RI implementation

```
public class AsynchronousFeature extends
WebServiceFeature {

    String ID = ".../async";

    @FeatureConstructor
    public AsynchronousFeature() {
        this.enabled = true;
    }

    public String getID() {
        return ID;
    }
}
```

# Declarative Asynchronous Services

## Partial RI implementation

```
public class AsynchronousInstanceResolver<T>
    extends AbstractMultiInstanceResolver<T> {

    public AsynchronousInstanceResolver(Class<T> clazz)

    {
        super(clazz);
    }

    public T resolve(Packet request) {
        try {
            return clazz.newInstance(); // For injection
        } catch (Exception e) {
            return null;
        }
    }
}
```

# Declarative Asynchronous Services

## Partial RI implementation

```
public Invoker createInvoker() {
    return new Invoker() {
        public void start(...) { A...r.start(wsc, endpoint); }
        public void dispose() { A...r.dispose(); }

        public Object invoke(Packet p, Method m,
                               Object... args) {
            T t = resolve(p);
            try {
                // If one way post to queue
            } finally {
                postInvoke(p, t);
            }
        }
    };
}
```

# Declarative Asynchronous Services

## Partial RI implementation - TODO

- Not clear how to do the transform with the RI
  - Annotation processor?
- Need to perform injection on class?
- How to control WSDL generation?



# Other Improvements

# Other Improvements

## WSDL Improvements

- Use partnerLinkType to relate portType instances
- Request/ Response operation mapping
- Binding mapping
- Would allow better “Top Down” generation
- Complex message exchanges
  - One request – many responses
  - One request – one or more of possible responses
  - Interaction from the process point of view
- Fault as input in the response portType

# Other Improvements

## WSDL Improvements

```
<binding name="StockQuoteResponseBinding"
type="tns:StockQuoteResponse">
  <soap:binding style="document" transport="..." />
  <async:responseFor ref="tns:StockQuoteSoapHttp" />
  <operation name="getPriceResponse">
    <soap:operation soapAction="" />
    <async:responseFor ref="getPrice" />
    <input>
      <soap:body use="literal" />
    </input>
  </operation>
</binding>
```

## Other Improvements

### Asynchronous Service Better Client v 1

```
@WebService
```

```
@AsyncClient
```

```
public interface StockQuote {
```

```
    Response<Float> getPriceAsync(String  
    ticker);
```

```
    Future<?> getPriceAsync(String ticker,  
        AsyncHandler<Float> responseReceiver);
```

```
}
```

## Other Improvements

### Asynchronous Service Better Client v 2

```
public class Client
    implements StockQuoteReponse {

    @Component StockQuote;

    public void
        getStockQuoteResponse(float...)
    {
        .. Handle response
    }
}
```

# Summary

- Asynchronous APIs are **powerful** but harder to work with
- By using WS- Addressing you can create a **cross platform** asynchronous messaging system
- **Implementing** a service can be fiddly
- We put forward a proposal for **declaratively** making a POJO service asynchronous
- WSDL could be **extended** to make asynchronous relationships more explicit for tooling
- **Client model** could be better.

# Q & A

- See Also

- <http://jax-ws.dev.java.net>
- <http://www.oracle.com/technology/products/jdev/>
- <http://www.oracle.com/technology/products/ias/index.html>
- <http://edocs.bea.com/wls/docs/100/index.html>

- Blogs

- <http://kingsfleet.blogspot.com/>

- Email

- [gerard.davison@oracle.com](mailto:gerard.davison@oracle.com)