Writing a JAX-WS Client

- Types of JAX-WS Clients
- Proxy-based Clients
- Dispatch Clients

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Types of JAX-WS Clients

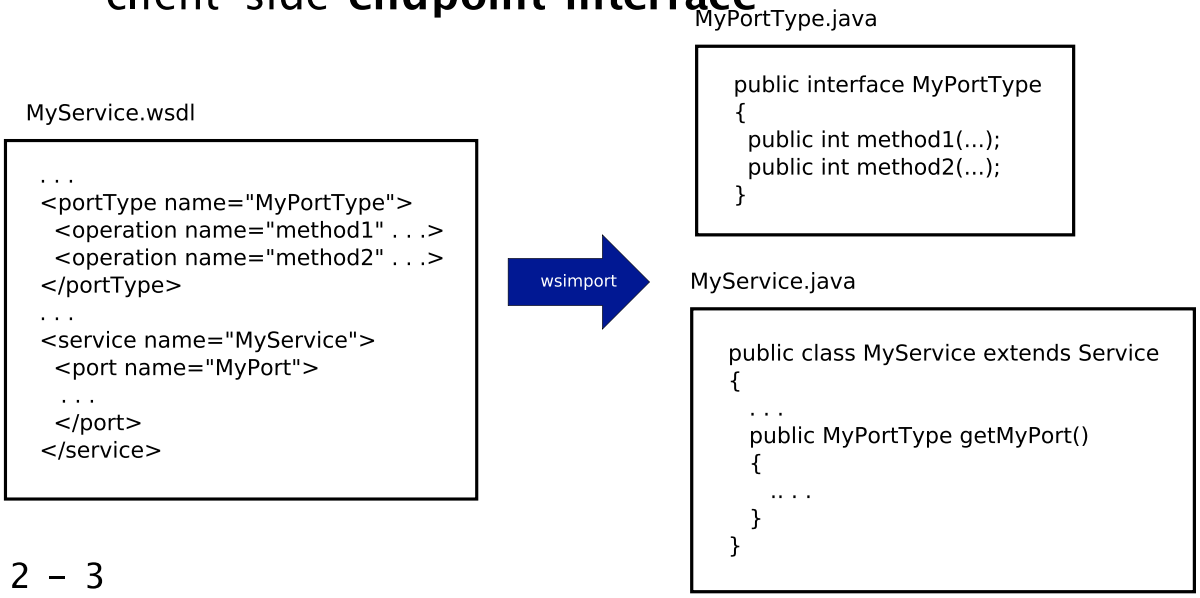
- JAX-WS provides for two basic client programming models:
  - **Proxy (stub)** - High-level access with automatic Java/XML mapping performed by JAX-B
  - **Dispatch** - Low level access to the "raw" message
- We will cover Dispatch clients in later chapters

The Dispatch approach replaces the Dynamic Invocation Interface (DII) from JAX-RPC.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Proxy-Based Clients

- Proxy-based clients usually rely on Java artifacts generated by a tool like **wsimport** that process WSDL
- The tool generates a **Service** subclass and a client-side **endpoint interface**



You can actually use the proxy approach even if you don't have a tool like wsimport, but then you need to write the SEI and any other required Java files manually, or use some other generation tool.

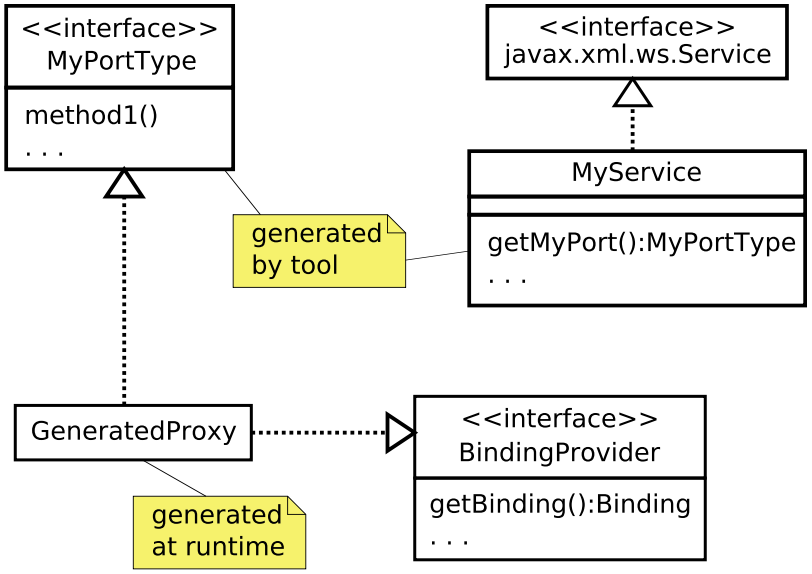
wsimport is included with JDK 6 and later.

This is similar to JAX-RPC, but completely implementation portable (JAX-RPC implementations generate vendor-specific artifacts).

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JAX-WS Proxy API

- The **Service** interface provides **factory** methods so that clients can create proxies
- The **BindingProvider** interface provides access to the service's protocol binding (e.g. the endpoint URL)



The tool-generated Service subclass will have factory methods customized for the specific service described by the WSDL, in this case getMyPort().

When the client calls getMyPort() on the MyService, the latter object uses the JSE dynamic proxy API to generate a class and an object "on the fly". For more information on the JSE dynamic proxy API (part of the reflection API), see the JSE JavaDoc.

The dynamically generated proxy object (and class) implements both the SEI and BindingProvider interfaces.

Note that the tool will also generated any other needed Java artifacts. For example, if a WSDL operation returns an XML complex type, JAX-WS will delegate to JAXB to generate a Java class that maps to the complex type.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using the Service as a Factory

- A client can create an instance of the Service subclass and then use it to retrieve a proxy object
- At runtime, the JAX-WS runtime uses the JSE **dynamic proxy** facility to generate a proxy class on the fly and create a proxy object

```
1    MyService svc = new MyService();
2
3    MyEndpoint proxy = svc.getMyPort();
```


Unlike the equivalent JAX-RPC code, this JAX-WS use of proxies is completely portable across implementations.

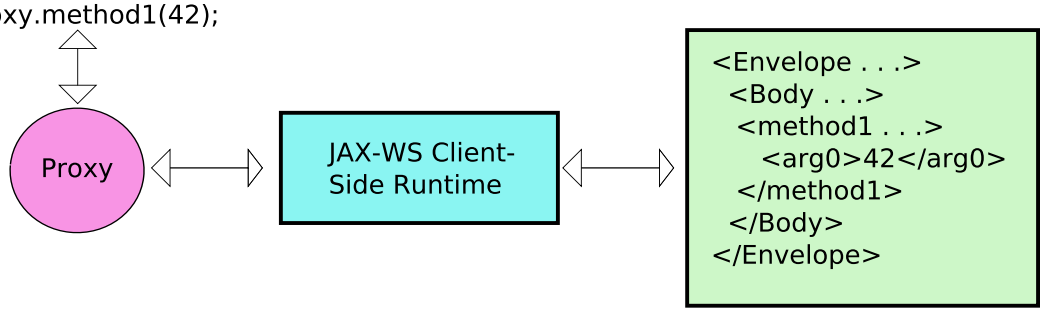
Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Calling SEI Methods on the Proxy

- The proxy and the client-side JAX-WS runtime generate a SOAP messages and send it to the remote Web service, performing Java-to-XML mapping
- The proxy waits for the response and then performs XML-to-Java mapping to return any results back to the waiting client

```
MyEndpoint proxy = . . .

int ret = proxy.method1(42);
```

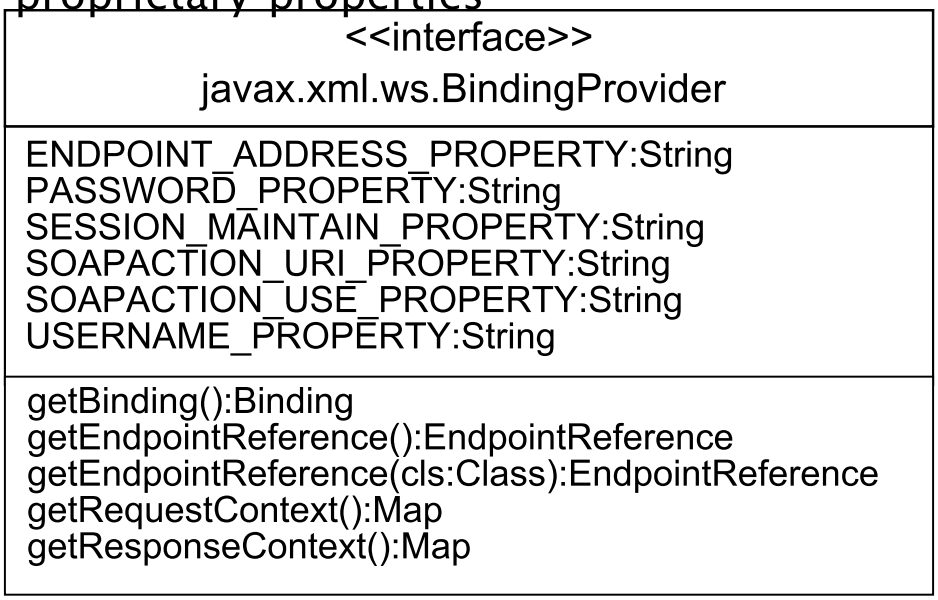


This is an example of the "Proxy" Design Pattern described in the famous Design Patterns book by Gamma, et al.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The BindingProvider Interface

- The **BindingProvider** interface gives a client access to several **standard properties** and implementations can provide additional proprietary properties



This page documents the standard properties - implementations can support additional proprietary properties.

The property names shown here are all Strings, but the property types themselves need not be. For example, the value of the SOAPACTION_USE_PROPERTY is java.lang.Boolean.

Note that the SOAPACTION_* properties don't actually let the client change the SOAP action - they just influence how SOAP action works with any WS-Addressing runtime.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using the BindingProvider Interface

- Since the generated stub object implements both the SEI and BindingProvider, you can safely cast references and then access properties

```
1    MyEndpointService svc = new MyEndpointService();
2
3    MySEI proxy = svc.getMyEndpointPort();
4
5    BindingProvider bp = (BindingProvider) proxy;
6    Map<String, Object> rc = bp.getRequestContext();
7
8    rc.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
9      "http://mynewendpoint.com");
10   rc.put(BindingProvider.USERNAME_PROPERTY,
11     "sam");
12   rc.put(BindingProvider.PASSWORD_PROPERTY,
13     "1234");
14
15   int result = proxy.m1("hh");
```

Here we show using the BindingProvider and its associated RequestContext to change the endpoint and login credentials for a service request.

Changing the endpoint is useful if you want to use a TCP/IP monitor, or if the client looks up the endpoint address dynamically, perhaps via a UDDI registry.

Providing a username and password is useful either if the service uses HTTP BASIC Authentication, or if the client is behind an authenticating proxy server. In the later case, the client will also need to configure the proxy host and port as System properties, for example:

```
System.setProperty("http.proxyHost", "myProxyServer.com");
System.setProperty("http.proxyPort", "80");
```

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Managed Clients

- Client programs that themselves run in a **JEE 5 container** can use **dependency injection** to retrieve a reference to **Service**

```
1    public class MyServlet extends HttpServlet
2    {
3      @WebServiceRef
4      private MyEndpointService myService;
5
6      protected void doGet(. . .)
7      {
8        . . .
9        MySEI proxy = myService.getMyEndpointPort();
10
11       out.println(proxy.m1("Hello"));
12     }
13   }
```

This follows the general trend in JEE 5 and later where programs don't explicitly look up or create resources, instead they request the resource to be injected by the application server.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The @WebServiceRef Annotation

- This annotation lets you inject a service reference

Element	Description
name	JNDI name of resource (optional)
value	Service class type (optional)
wsdlLocation	WSDL URL (optional)

The JNDI name is useful if you've defined a reference to the service in the JNDI namespace, for example by a local service-ref element in web.xml. It provides a level of indirection so that if the endpoint address changes, you don't need to modify client code, only the web.xml deployment descriptor.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- The fundamentals of JAX-WS clients
- How to write proxy-based clients

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum