Emerging Technologies for the Enterprise

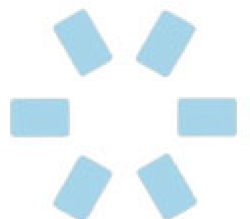
March 28-29, 2007 Drexel University, Philadelphia, PA

CXF for the Enterprise and Web

Dan Diephouse

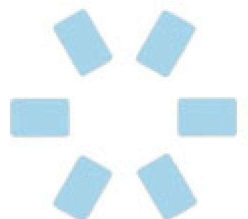
Today

- Our dilemma
- CXF? What's that?
- The Customer Service
- RESTful rendition
- SOAP rendition
- Conclusions

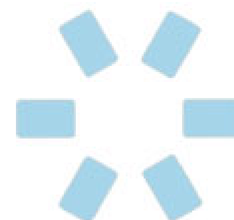


Our Dilemma

SOAP	REST & HTTP
Java & .NET	Dynamic languages
WSDL & WS-*	HTTP, SSL, URLs
Middleware Integration	Web Integration
Messages	Resources

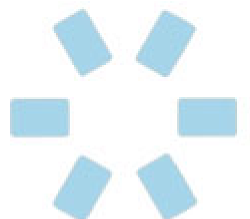


Survey!



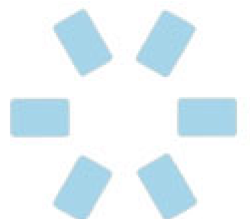
What is CXF?

- Services Framework
- SOAP 1.1, 1.2
- Basic Profile
- WSDL 1.1
- WS-ReliableMessaging, WS-Policy, WS-Security
- JAX-WS 2.1
- RESTful services
- JSON & XML support
- Asynchronous
- Spring 2.0 support



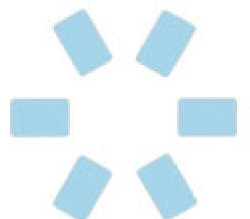
History

- Merger of Celtix and XFire communities
- Apache Incubator project
- Started ~8 months ago
- Improvements over XFire
  - Cleaner architecture
  - Focus on supporting more of WS-*
  - Improved JMS support
  - Spring 2.0
  - REST
  - JSON
  - Asynchronous
  - Much more!



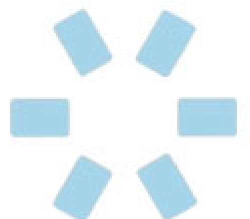
The Customer Service

- Customer Service
- We need to:
  - Add
  - Remove
  - Get
  - Delete



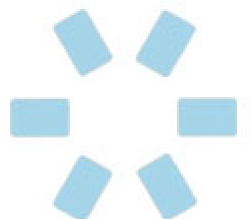
The Customer Service

```
public interface CustomerService {
   Customers getCustomers();
   Customer getCustomer(long id);
   void updateCustomer(Customer c);
   long addCustomer(Customer c);
   void deleteCustomer(long id);
}
```



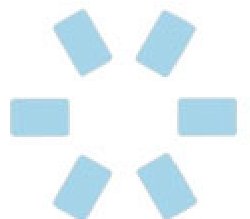
JAX-WS

- Annotations and APIs for building and consuming web services
- Server:
  - @WebService, @WebMethod, @WebParam, @WebResult..
  - Provider
  - wsdl2java generated services
- Client
  - Dispatch
  - wsdl2java generated clients



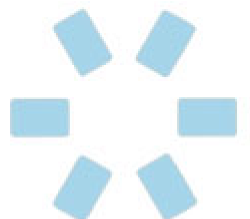
JAX-WS

```
@WebService
public interface CustomerService {
  @WebResult(name="Customer")
  Customer getCustomer(
    @WebParam(name="id") long id);

…
}
```



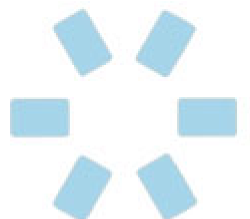
JAX-WS

```
@WebService( endpointInterface =
    "CustomerService" )
public class CustomerServiceImpl {
  Customer getCustomer(long id) {
   …
  }
…
}
```

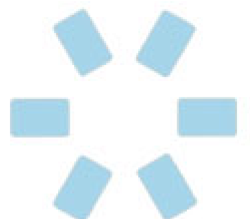
JAXB

- Standard Java data-binding
- Uses annotations to map classes to XML



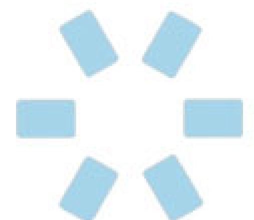
JAXB

```
@XmlRootElement
public class Customer {
   private long id;

   public long getId() { … }
   public void setId(long id) { … }
…
}
```



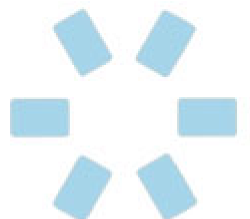
SOAP Services

- Everything is a message
- Point to point
- Headers provide extensibility
- Described by WSDL



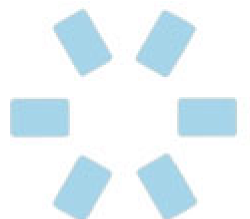
Building a SOAP service with CXF

```
JaxWsServerFactoryBean sf =
  new JaxWsServerFactoryBean();

sf.setServiceClass(CustomerServiceImpl.class);
sf.setAddress("http://localhost:8080/soap");
sf.getServiceFactory().setInvoker(
  new BeanInvoker(new CustomerServiceImpl()));

sf.create();
```



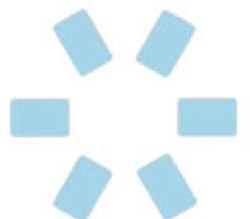
Spring 2 too…

```
<beans ..>

  <jaxws:endpoint
    implementor="com.foo.CustomerServiceImpl"
    address="http://host/CustomerService">

</beans>
```



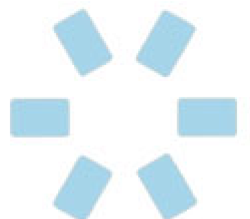
Advantages

- Integrates seamlessly with .NET and other Java appservers
- Works well with JMS & HTTP
- Published WSDL describes your operations to others



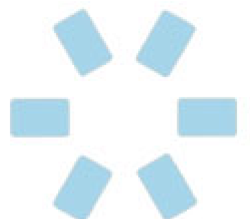
RESTful Services

- REpresentational State Transfer
- Instead of messages, we have resources
- Resources are accessed via verbs
- URIs are accessed via GET, POST, PUT, DELETE
- HTTP is *the* example.



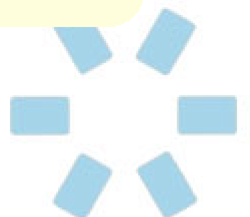
Why “Just use HTTP”?

- Linkability: EVERYTHING is accessible and addressable via a simple URL
- Simplicity
  - Uniform interface
  - Testable via a browser
  - No WSDL
- Works well with caches
- Integrates well with dynamic languages such as Javascript or Ruby



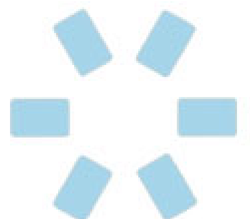
HTTP Methods

Method	Properties
GET	• Cacheable • SAFE – no side effects
POST	• Unsafe operations, which can't be repeated
PUT	• Idempotent
DELETE	• Idempotent



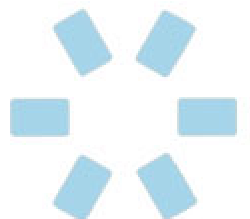
The CXF HTTP Binding

- Allows you to map operations to different URI & Verb combinations
- Convention based mapping
- Annotation based mapping
- You can also supply your own class to do the mapping



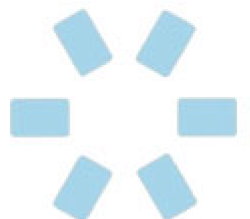
Example using conventions

- getCustomers() -> GET + /customers
- getCustomer(id) -> GET + /customers/{id}
- addCustomer(Customer) -> POST + /customers
- updateCustomer(Customer) ->
  PUT + /customers/{id}
- deleteCustomer(id) -> DELETE + /customers/{id}



Example using annotations

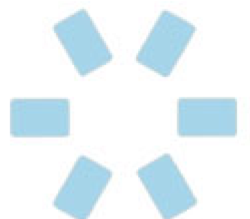
```
@Get
@HttpResource("/customers/{id}")
Customer getCustomer(String id);

@Post
@HttpResource("/customers")
String addCustomer(Customer c)
```

Schema -> Parameter Matching

```
<element name="getCustomer">
 <complexType>
  <sequence>
    <element name="id" type="long"/>
  </sequence
 </complexType>
</element>
```

Matches the {id} in the URI template



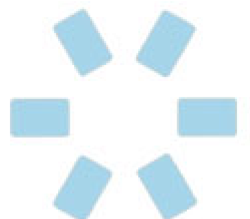
"Unwrapped" mode

```
@Get
@HttpResource("/customers/{id}")
Customer getCustomer(GetCustomers request);
```

Avoids returning:

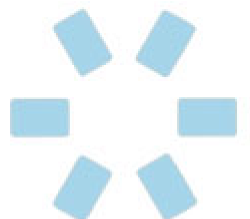
```
<getCustomersResponse>
  <customers>..</customers>
</getCustomersResponse>
```

Returns: <customers>…</customers>



Creating the Service

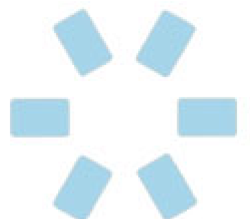
```
JaxWsServerFactoryBean sf =
  new JaxWsServerFactoryBean();

sf.setServiceBean(new CustomerService());

sf.setBindingFactory(
  new HttpBindingInfoFactoryBean());

sf.setAddress("http://localhost:8080/xml");
sf.getServiceFactory().setWrapped(false);
sf.create();
```



Accessing the Service

- To the web browser and command line!

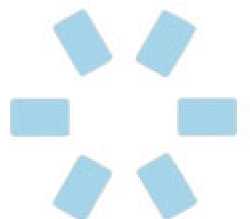
- wget http://localhost/xml/customers
- wget --post-file customer.xml http://localhost/xml/customers



JSON

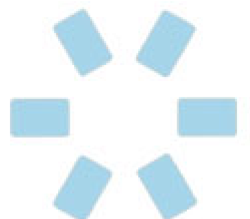
- JSON = Javascript Object Notation
- The data format **IS** Javsacript
- This enables easy integration with web pages
- Example

```
{ “customer” : {
      “name” : “Jim Bob”,
      “id” : “12345”
} }
```

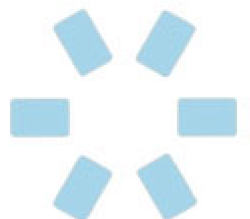
Javascript

```
var res =
   '(' + xmlhttp.responseText + ')';
var customers = eval(res);

customers[0].customer.name
```

Introducing Jettison

- Jettison is a StAX implementation which reads and writes JSON instead of XML
  - http://jettison.codehaus.org
- Two variations on how to map XML to JSON
  - Mapped convention
  - BadgerFish
- Mapped Convention allows you to map an xml element to a JSON



Mapped Convention Example

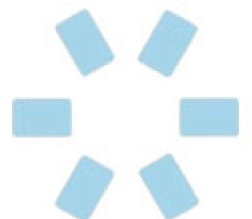
```
<customer>
  <name>Jim Bob</name>
</customer>
```

```
{ "customer" {
   "name" : "Jim Bob"
}}
```

Becomes:

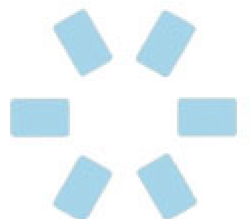
```
<c:customer
  xmlns:c="urn:acme:customer">
  <c:name>Jim Bob</c:name>
</c:customer>
```

```
{ "acme.customer" {
   "acme.name" : "Jim Bob"
}}
```



Enabling Jettison on your endpoint

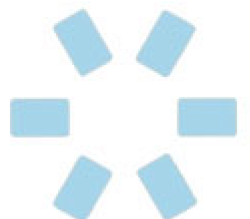
```
Map nstojns = new HashMap();
nstojns.put("http://customer.acme.com", "acme");

MappedXMLInputFactory xif = new
   MappedXMLInputFactory(nstojns);

Map properties = new HashMap();
properties.put(XMLInputFactory.class.getName(),
   xif);
serverFactory.setProperties(properties);
```



Using the Service

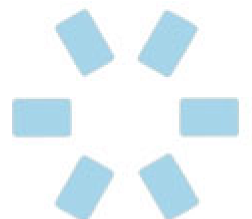
- To the web browser and command line!

- wget http://localhost/json/customers
- wget --post-file customer.json http://localhost/json/customers



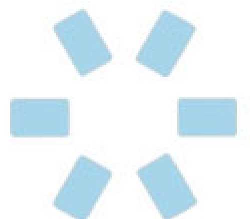
Writing a Client

```
JaxWsProxyFactoryBean sf =
  new JaxWsProxyFactoryBean();

sf.setServiceClass(CustomerService.class);
sf.getServiceFactory().setWrapped(false);
sf.getClientFactoryBean().setBindingFactory(new
   HttpBindingInfoFactoryBean());
sf.setAddress("http://localhost:8080/xml/");

CustomerService cs = (CustomerService)
   sf.create();
```



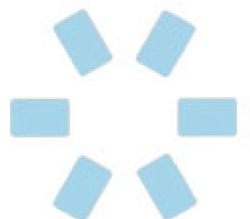
Advantages

- Easy mapping to HTTP via conventions or annotations
- Support for full set of HTTP verbs
- Support for JSON via Jettison allows easy integration with Javascript



Conclusions

- Its possible to build both a SOAP and RESTful endpoint with the same data objects and business logic
- Allows you to cater to different languages and different integration styles easily
- RESTful services are very powerful



Questions?

- Blog: http://netzooid.com
- CXF: http://incubator.apache.org/cxf/
- Jettison: http://jettison.codehaus.org
- Corporate: http://envoisolutions.com
- Email: dan@envoisolutions.com