Middleware
What is Middleware?

Middleware is a class of software technologies designed to help (I) manage the complexity and (II) heterogeneity inherent in distributed systems.
Middleware as a Programming Abstraction

- A software layer **above the operating system** and **below the application program** that provides a common programming abstraction across a distributed system

- A higher-level building block than APIs provided by the OS (such as sockets)
Middleware as a Programming Abstraction

Programming abstractions offered by middleware mask some of the heterogeneity and handles some of the complexity programmers of a distributed application must deal with:

- Middleware always mask heterogeneity of the underlying networks, hardware
- Most middleware mask heterogeneity of operating systems and/or programming languages
- Some middleware even mask heterogeneity among different vendor implementations of the “same” middleware standard

e.g., CORBAIDL
Middleware as a Programming Abstraction

- An operating system (OS) is the software that makes the hardware useable (A bare computer without an OS could be programmed with great difficulty.)

- Middleware is the software that makes a distributed system (DS) programmable (Programming a DS is much more difficult without middleware.)
Middleware as Infrastructure

- Behind programming abstractions there is a complex infrastructure that implements those abstractions
  (Middleware platforms are very complex software systems.)

- As programming abstractions reach higher and higher levels, the underlying infrastructure implementing the abstractions must grow accordingly
  - Additional functionality is almost always implemented through additional software layers
  - The additional software layers increase the size and complexity of the infrastructure necessary to use the new abstractions

E.g. to handle errors we can have “transactional RPCs”
The infrastructure is also intended to support additional functionality that makes development, maintenance, and monitoring easier and less costly:

- logging,
- recovery,
- advanced transaction models (e.g. transactional RPC),
- language primitives for transactional demarcation,
- transactional file system,
- etc.
Middleware as Infrastructure

The **infrastructure also takes care of (all) the non-functional properties** typically ignored by data models, programming models, and programming languages:

- performance,
- availability,
- resource management,
- reliability,
- etc.
Middleware as Infrastructure

Conceptual Model

[Alonso; Web services: Concepts, Architectures and Applications; Springer, 2004]
Web Services
Point-to-point integration across companies

Motivation

Each company uses its own “concrete” message broker - if we want to communicate with multiple companies we have to implement and maintain several adapters / solutions.

[Web Services - Concepts, Architectures and Applications; Alonso et al.; Springer 2004]
What is a Web Service?

[...] self-contained, modular business applications that have open, internet-oriented, standards-based interfaces.

Definition by the UDDI consortium
What is a Web Service?

Interactions based on protocols redesigned for peer to peer and B2B settings.

Internal functionality made available as a service.

Customer

Web service

Internal procurement requests

Internal infrastructure

Supplier

Web service

Internal infrastructure

Warehouse

Web service

Internal infrastructure

Languages and protocols standardized, eliminating need for many different middleware infrastructures (need only the web services middleware).

(Web Services; Springer 2004)
What is a Web Service?

Company A (provider)

internal service

middleware

Web service

Web service

Web service

Company B (client)

internal service

internal service

wide area network (Internet)

client

middleware

internal service

internal service
What is a Web Service?

Core Elements

- The components:
  - Service requester: The potential user of a service
  - Service provider: The entity that implements the service and offers to carry it out on behalf of the requester
  - Service registry: A place where available services are listed
“Classical”
Web Services
(using SOAP and WSDL)
What is a Web Service?

A Web service is a **software system identified by a URI**, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.
“Classical” Web Services

What is a Web Service?

Web Services are:

- A standardized way of integrating web-based applications using the XML, SOAP, WSDL and UDDI open standards over an Internet protocol backbone.
- XML is used to tag the data, SOAP is used to transfer the data, WSDL is used for describing the services available and UDDI is used for listing what services are available.
What is a Web Service?

Core Standards

- Simple Object Access Protocol (SOAP)
  Web service message format

- Web Services Description Language (WSDL)
  Web service interface description

- Universal Description, Discovery, and Integration (UDDI)
  Web service discovery
Minimalist Infrastructure for Web Services

```
<operation name="orderGoods">
  <input message = "OrderMsg"/>
</operation>
```

[Web Services; Springer 2004]
Minimalist Infrastructure for Web Services

[Web Services; Springer 2004]
Web Services Protocol Stack

Basic Protocol Stack
- XSD, WSDL, UDDI, Policy, MetadataExchange
- XML, SOAP, Addressing
- HTTP, HTTPS, SMTP

Extended Protocol Stack
- Security
- Reliable Messaging
- Transactions

Basic Protocol Stack
- BPEL4WS

Extended Protocol Stack
- WS - Security
- WS-Reliability
- WS-AtomicTransaction
- WS-BusinessActivity

Service Composition
- Composable Service Assurances
- Description
- Messaging
- Transports
“Classical” Web Services
RESTful Web Services

Some of the following slides are based upon the presentation:
RESTful Web Services; John Cowan; cowan@ccil.org
http://www.ccil.org/~cowan
What’s a Web Service?
(In the context of restful web services.)

- A web service is just a web page meant for a computer to request and process
- More precisely, a Web Service is a “web page” that’s meant to be consumed by an autonomous program as opposed to a web browser or similar UI tool
What’s a Web Service?
(In the context of restful web services.)

- Web Services require an architectural style to make sense of them, because there’s no smart human being on the client end to keep track.
- The pre-web techniques of computer interaction don't scale on the Internet. They were designed for small scales and single trust domains.

**Architectural Style**

- “...a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem.”
- Some common examples of architectural styles include:
  - Pipes and Filter
  - Layered Architecture
  - ...
REST Style

Basically, a set of design principles to judge architectures.

- Client-server
- Stateless
- Cached
- Uniform interface
- Layered system
- (Code on demand)
RESTful?

- REST = Representational State Transfer
  (Basically, a set of design principles to judge architectures)
  - Resources are identified by uniform resource identifiers
  - Resources are manipulated through their representations
  - Messages are self-descriptive and stateless
  - Multiple representations are accepted or sent
  - Hypertext is the engine of application state

- One possible architecture is “the” resource-oriented architecture (ROA)
  The ROA qualifies as being RESTful.
  - Method information goes into the HTTP method
  - Scoping information goes into the URI/IRI

Here, hypertext basically means that a message/representations contains the necessary links to other resources.
RESTful Web Services

Foundations

- **HTTP**
  - … as the underlying transport protocol; *important property: stateless*

- **URI** (or IRIs if you wish)
  - … to locate resources

- **XML or JSON (not necessarily, but in most cases) (Alternative XHTML)**
  - … to get a representation

(The Web already supports machine-to-machine integration. What's not machine-processable about the current Web (i.e. HTML) isn't the protocol, it's the content.)
Surfing the Web:

- To fetch a web page, the browser does a **GET** on some URI and retrieves a representation (HTML, plain text, JPEG, or whatever) of the resource identified by that URI.

- **GET** is fundamental to browsers because mostly they just browse.

- **REST requires a few more verbs to allow taking actions**
  (However, in REST we use universal verbs only.)
“Universal Verbs”

- GET
  … to retrieve information
- POST
  … to add new information, showing its relation to old information
- PUT
  … to update information
- DELETE
  … to discard information
Web Pages as Resources

Nouns...

- A web page is a representation of a resource
- **Resources are just concepts**
- URIs tell a client that there's a concept somewhere
- **Clients can then request a specific representation** of the concept from the representations the server makes available
Two Types of State

First: Application(/Session) State

- “State” means **application/session state**
  Application state is the information necessary to understand the context of an interaction
  (Authorization and authentication information are examples of application state.)

- Maintained as part of the content transferred from client to server and back to client

- Thus any server can potentially continue a transaction from the point where the transaction was left off
Two Types of State
Second: Resource State

- **Resource state** is the kind of state that the S in REST refers to.
- The "stateless" constraint means that all messages must include all application state.
  (Basically, that we don’t have sessions.)
(Multiple) Representations

- Most resources have only a single representation
  REST can support any media type, but XML/JSON is expected to be the most popular transport for structured information. (HTTP supports content negotiation.)

- XML makes it possible to have as many representations as you need
  - You can even view them in a clever way, thanks to the magic of XSLT and CSS

- Links can be embedded
  Links mirror the structure of how a user makes progress through an application. The user is in control, thanks to the back button and other non-local actions. In a web service, the client should be in control in the same sense.
Test of RESTfulness

- Can I do a **GET** on the URLs that I **POST** to?
- If so, do I get something that in some way represents the state of what I've been building up with the **POSTs**? (HTML forms almost always fail miserably.)
- Would the client notice if the server were to be...
  - restarted at any point between requests,
  - re-initialized by the time the client made the next request.
Is REST Enough?

Is a “uniform” interface enough?

- What happens when you think you need application semantics that don't fit into the GET / PUT / POST / DELETE generic interfaces and representational state model?
  - People tend to assume that the REST answer is:
    - If the problem doesn't fit HTTP, build another protocol
    - Extend HTTP by adding new HTTP methods (e.g. WEBDav, WEBCal,..).
  - However, the four standard verbs are considered sufficiently general!
RESTful Web Services
a first Example

http://www.peej.co.uk/articles/restfully-delicious.html
Example (Based Upon del.icio.us)

- Lets start with describing what it is the Web Service does
- It allows us to:
  - Get a list of all our bookmarks and to filter that list by tag or date or limit by number
  - Get the number of bookmarks created on different dates
  - Get the last time we updated our bookmarks
  - Get a list of all our tags
  - Add a bookmark
  - Edit a bookmark
  - Delete a bookmark
  - Rename a tag
Resources
Example (based upon del.icio.us)

› Bookmarks
  http://del.icio.us/api/[username]/bookmarks

› Tags
  http://del.icio.us/api/[username]/tags

› [username] is the username of the user’s bookmarks we’re interested in

Beware - URI design is important, but not the most important aspect!
### Resource Representations

Example (based upon del.icio.us)

- We define some XML document formats and mimetypes to identify them:

<table>
<thead>
<tr>
<th>Mimetype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delicious/bookmarks+xml</td>
<td>A list of bookmarks</td>
</tr>
<tr>
<td>delicious/bookmark+xml</td>
<td>A bookmark</td>
</tr>
<tr>
<td>delicious/bookmarkcount+xml</td>
<td>Count of bookmarks per date</td>
</tr>
<tr>
<td>delicious/update+xml</td>
<td>When the bookmarks were last updated</td>
</tr>
<tr>
<td>delicious/tags+xml</td>
<td>A list of tags</td>
</tr>
<tr>
<td>delicious/tag+xml</td>
<td>A tag</td>
</tr>
</tbody>
</table>
## Getting Bookmarks

<table>
<thead>
<tr>
<th>URL</th>
<th><a href="http://del.icio.us/api/%5Busername%5D/bookmarks/">http://del.icio.us/api/[username]/bookmarks/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>GET</td>
</tr>
<tr>
<td>Querystring</td>
<td>tag= Filter by tag</td>
</tr>
<tr>
<td></td>
<td>dt= Filter by date</td>
</tr>
<tr>
<td></td>
<td>start= The number of the first bookmark to return</td>
</tr>
<tr>
<td></td>
<td>end= The number of the last bookmark to return</td>
</tr>
<tr>
<td>Returns</td>
<td>200 OK &amp; XML (delicious/bookmarks+xml)</td>
</tr>
<tr>
<td></td>
<td>401 Unauthorized</td>
</tr>
<tr>
<td></td>
<td>404 Not Found</td>
</tr>
</tbody>
</table>
GET http://del.icio.us/api/peej/bookmarks/?start=1&end=2

<?xml version="1.0"?>
<bookmarks start="1" end="2"
    next="http://del.icio.us/api/peej/bookmarks?start=3&amp;end=4">
    <bookmark url="http://www.example.org/one" tags="example,test"
        href="http://del.icio.us/api/peej/bookmarks/a211528fb5108cddaa4b0d3aeccdbdcf"
        time="2005-10-21T19:07:30Z">
        Example of a Delicious bookmark
    </bookmark>
    <bookmark url="http://www.example.org/two" tags="example,test"
        href="http://del.icio.us/api/peej/bookmarks/e47d06a59309774edab56813438bd3ce"
        time="2005-10-21T19:34:16Z">
        Another example of a Delicious bookmark
    </bookmark>
</bookmarks>
## Get Information on a Specific Bookmark

<table>
<thead>
<tr>
<th>URL</th>
<th><a href="http://del.icio.us/api/%5Busername%5D/bookmarks/%5Bhash">http://del.icio.us/api/[username]/bookmarks/[hash</a>]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>GET</td>
</tr>
<tr>
<td>Returns</td>
<td>200 OK &amp; XML (delicious/bookmark+xml)</td>
</tr>
<tr>
<td></td>
<td>401 Unauthorized</td>
</tr>
<tr>
<td></td>
<td>404 Not Found</td>
</tr>
</tbody>
</table>
GET http://del.icio.us/api/peej/bookmarks/a211528fb5108cddaa4b0d3aeccdbdcf

<?xml version="1.0"?>
<bookmark url="http://www.example.org/one" time="2005-10-21T19:07:30Z">
    <description>
        Example of a Delicious bookmark
    </description>
    <tags count="2">
        <tag name="example" href="http://del.icio.us/api/peej/tags/example"/>
        <tag name="test" href="http://del.icio.us/api/peej/tags/test"/>
    </tags>
</bookmark>
# Get Count of Posts per Date

<table>
<thead>
<tr>
<th>URL</th>
<th><a href="http://del.icio.us/api/%5Busername%5D/bookmarks/count">http://del.icio.us/api/[username]/bookmarks/count</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>GET</td>
</tr>
<tr>
<td>Querystring</td>
<td>tag= filter by tag</td>
</tr>
<tr>
<td>Returns</td>
<td>200 OK &amp; XML (delicious/bookmark+xml)</td>
</tr>
<tr>
<td></td>
<td>401 Unauthorized</td>
</tr>
<tr>
<td></td>
<td>404 Not Found</td>
</tr>
</tbody>
</table>
### Get Last Update Time for the User

<table>
<thead>
<tr>
<th>URL</th>
<th><a href="http://del.icio.us/api/%5Busername%5D/bookmarks/update">http://del.icio.us/api/[username]/bookmarks/update</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>GET</td>
</tr>
<tr>
<td>Returns</td>
<td>200 OK &amp; XML (delicious/bookmark+xml)</td>
</tr>
<tr>
<td></td>
<td>401 Unauthorized</td>
</tr>
<tr>
<td></td>
<td>404 Not Found</td>
</tr>
</tbody>
</table>
## Add a Bookmark

<table>
<thead>
<tr>
<th><strong>URL</strong></th>
<th><a href="http://del.icio.us/api/%5Busername%5D/bookmarks/">http://del.icio.us/api/[username]/bookmarks/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Method</strong></td>
<td><strong>POST</strong></td>
</tr>
<tr>
<td><strong>Request Body</strong></td>
<td>XML (delicious/bookmark+xml)</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>201 Created &amp; Location</td>
</tr>
<tr>
<td></td>
<td>401 Unauthorized</td>
</tr>
<tr>
<td></td>
<td>415 Unsupported Media Type (if the send document is not valid)</td>
</tr>
</tbody>
</table>
Add a Bookmark

Example delicious/bookmarks+xml document

POST http://del.icio.us/api/peej/bookmarks/

```xml
<?xml version="1.0"?>
<bookmark url="http://www.example.org/one"
  time="2005-10-21T19:07:30Z">
  <description>Example of a Delicious bookmark</description>
  <tags>
    <tag name="example" />
    <tag name="test" />
  </tags>
</bookmark>
```
## Modify a Bookmark

<table>
<thead>
<tr>
<th>URL</th>
<th><a href="http://del.icio.us/api/%5Busername%5D/bookmarks/%5Bhash">http://del.icio.us/api/[username]/bookmarks/[hash</a>]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td><strong>PUT</strong></td>
</tr>
<tr>
<td>Request Body</td>
<td>XML (delicious/bookmark+xml)</td>
</tr>
<tr>
<td>Returns</td>
<td>201 Created &amp; Location</td>
</tr>
<tr>
<td></td>
<td>401 Unauthorized</td>
</tr>
</tbody>
</table>
|         | 404 Not Found  
  (new bookmarks cannot be created using put!) |
|         | 415 Unsupported Media Type  
  (if the send document is not valid) |
## Delete a Bookmark

<table>
<thead>
<tr>
<th>URL</th>
<th><a href="http://del.icio.us/api/%5Busername%5D/bookmarks/%5Bhash">http://del.icio.us/api/[username]/bookmarks/[hash</a>]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>DELETE</td>
</tr>
<tr>
<td>Returns</td>
<td>204 No Content</td>
</tr>
<tr>
<td></td>
<td>401 Unauthorized</td>
</tr>
<tr>
<td></td>
<td>404 Not Found</td>
</tr>
</tbody>
</table>
Using the Service

Example scenario - navigating and deleting entries

- GET http://del.icio.us/api/
  Fetching the API’s homepage. We parse the response XML and find the tags URL of the user with the name "Peej"…

- GET http://del.icio.us/api/peej/tags/
  We parse the XML and see if the tag "test" is mentioned. If it isn't, then we GET the URL in the "next" attribute and try again. Eventually we'll get the URL http://del.icio.us/api/peej/tags/test...

- GET http://del.icio.us/api/peej/tags/test
  We've got the details for the tag "test", i.e. all bookmarks with the tag. Eventually we'll have a list of all bookmarks tagged with "test".

- DELETE http://del.icio.us/api/peej/bookmarks/[hash]
  Finally we can walk through our list of bookmarks and call the DELETE HTTP method on each of their URLs.
RESTful Web Services
a second Example using Jersey
(Developing RESTful Web Services using JAX-RS 1.0 (JSR 311))
https://jersey.dev.java.net/
A First Example

Implementation

```java
package samples.helloworld.resources;
import javax.ws.rs.*

// The Java class will be hosted at the URI path "/helloworld"
@Path("/helloworld")
public class HelloWorldResource {

    // The Java method will process HTTP GET requests
    @GET
    // The Java method will produce content identified by
    // the MIME Media type "text/plain"
    @Produces("text/plain")
    public String getClichedMessage() {
        return "Hello World";
    }

    @POST
    @Consumes("text/plain")
    public void postClichedMessage(String message) {
        // Store the message
    }
}
```

The `@Path` annotation specifies the relative URI path.

The `@GET` annotation is a request method designator, along with `@POST`, `@PUT`, `@DELETE`, and `@HEAD`, that is defined by JAX-RS, and which correspond to the similarly named HTTP methods.

The `@Produces` annotation is used to specify the MIME media types of representations a resource can produce and send back to the client.

The `@Consumes` annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.
The @Path Annotation and URI Path Templates

JAX RS 1.0 Mini-Tutorial

- The @Path annotation identifies the URI path template to which the resource responds, and is specified at the class level of a resource.
- The @Path annotation's value is a partial URI path template relative to the base URI of the server on which the resource is deployed.
- URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI.

**GIVEN:**
@Path("/users/{username}")

```java
public class UserResource {
    @GET @Produces("application/xml")
    public String getUser(@PathParam("username") String userName) { ... }
}
```

**THEN THE URL:** http://example.com/users/Galileo

WILL RESULT IN A CALL OF THE “getUser” METHOD WHERE THE “userName” is set to “Galileo”.
Using `@Consumes` and `@Produces`

Customize Requests and Responses - JAX RS 1.0 Mini-Tutorial

- The information sent to a resource and then passed back to the client is specified as a MIME media type in the headers of an HTTP request or response.

- The `@Produces` annotation is used to specify the MIME media types or representations a resource can produce and send back to the client.

```java
@GET
@Produces("text/html")
public String getHtml() {
    return "<html><body><h1>Hello World!</body></h1></html>";
}
```
Using `@Consumes` and `@Produces`

Customize Requests and Responses - JAX RS 1.0 Mini-Tutorial

- The `@Consumes` annotation is used to specify which MIME media types of representations a resource can accept, or consume, from the client. E.g. Consuming form data:
  - `@POST @Consumes("application/x-www-form-urlencoded")
    public void post(@FormParam("name") String name) {
        // Store the message
    }
    e.g. to make a HTTP “POST” call use curl:
  - `@POST @Consumes("text/plain")
    public void postClichedMessage(String message) {
        // Store the message
    }`