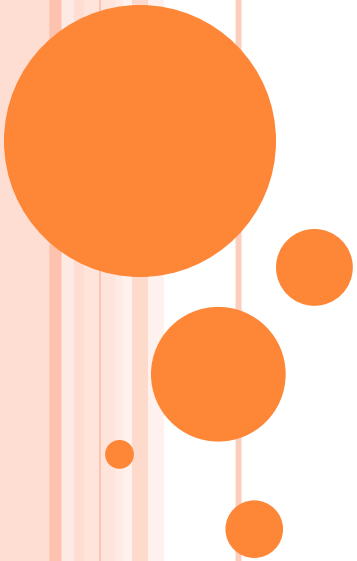


Web Services

COMP 30220: Distributed Systems

Lecturer: Rem Collier

Email: rem.collier@ucd.ie

Recap: Service-Oriented Architecture

- A way of building software in which the system is broken down into reusable components (called services) that can be combined to implement business processes.
  - It is supported through the use of **standards**.
  - Adheres to a **develop**, **deploy**, **use** philosophy
  - Deployments often require some form of "run-time" that provides some form of service registry that provides an infrastructure to support the registration, discovery and lookup of services.

- Simple Example: A University
  - **Services**: Student Registration, Programme Enrolment, Module Selection, Module Class Lists, Fees Payment, …
  - **Applications**:
    - A student coming to UCD to study Computer Science.
    - A lecturer teaching a module.

What is a Web Service?

- In the last lecture, we saw that services are discrete business functions that operate on data in a consistent and predictable way.

- Simply put, a **web service** is a service that is **accessible through the web**.
  - A web service is a **web page** that is designed to be **consumed by a program**.
  - Web services are an evolution of earlier techniques: RPC, RMI, CORBA, …
  - Key design objectives are: interoperability, firewall traversal, and complexity.
  - Web services are designed for standardised (interoperability) access over established protocols, such as HTTP or SMTP (firewall friendly), in a way this is simple to use (complexity).

What is a Web Service?

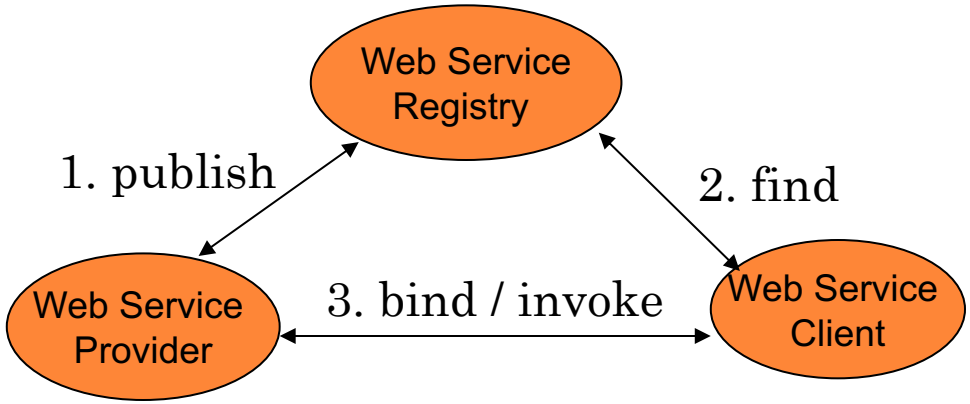
- A web service is an application component that:
  - Communicates via open protocols (HTTP, SMTP, …)
    - Well known protocols offered on standard ports.
  - Processes XML messages framed using SOAP
    - XML – eXtensible Markup Language (HTML for machines)
    - SOAP – Simple Object Access Protocol (a standard way to communicate built on XML)
  - Describes its messages using XML Schema
    - XML Schema – defines what is valid XML content.
  - Provides an endpoint description using WSDL
    - WSDL – Web Services Description Language (more XML to define what services are available)
  - Can be discovered using UDDI
    - Universal Description Discovery and Integration Specification – a service registry for web services.

Typical Web Service Scenario

- A buyer (which might be a simple client) is ordering goods from a seller service.
- The buyer finds the seller service by searching the UDDI directory.
- The seller service is a Web Service whose interface is defined using Web Services Description Language (WSDL).
- The buyer is invokes the order method on the seller service using a combination of Simple Object Access Protocol (SOAP) and the WSDL definition for the seller service.
- The buyer knows what to expect in the SOAP reply message because this is defined in the WSDL definition for the seller service.

The Web Services Model

- The Web Services architecture is based upon the interactions between three roles:
  - Service provider
  - Service registry
  - Service requestor
- The interactions involve the:
  - Publish operations
  - Find operation
  - Bind operations.



Insert: XML

- XML - eXtensible Markup Language.
  - A markup language much like HTML, but designed to describe data.
  - Unlike HTML, you **define your own tags**.

HTML	XML
<html> <body> John Doe 2 Backroads Lane New York 045935435 john.doe@gmail.com </body> </html>	<?xml version=1.0?> <contact> <name>John Doe</name> <address>2 Backroads Lane</address> <county>New York</county> <phone>045935435</phone> <email>john.doe@gmail.com</email> </contact>

- HTML is about presentation while XML is about representation!

Insert: XML

- The structure of an XML document is normally defined in an **XML Schema**.
  - Defines what tags can be used; how the can be used; and what types of values are permitted.
  - Clashes can be used by associating schema with **namespaces.**

```
<xs:element name="Customer">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Dob" type="xs:date" />
      <xs:element name="Address" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<Customer>
  <Dob>2000-01-12T12:13:14Z</Dob>
  <Address>
    34 thingy street, someplace,
    sometown, w1w8uu
  </Address>
</Customer>
```

```
<xs:element name="Supplier">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Phone‟ type="xs:integer"/>
      <xs:element name="Address" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<Supplier>
  <Phone>0123987654</Phone>
  <Address>
    22 whatever place, someplace,
    sometown, ss1 6gy
  </Address>
</Supplier>
```

SOAP

- SOAP stands for "Simple Object Access Protocol" .
  - Web Services expose useful functionality to Web users through a standard Web protocol called SOAP.
  - SOAP is an XML Schema that enable programs on separate computers to interact across any network.
  - SOAP uses mainly HTTP as a transport protocol (i.e. SOAP is transmitted as the payload of a HTTP message).

- SOAP has three major characteristics:
  - **Extensibility**: WS-security and WS-routing are among the extensions under development.
  - **Neutrality**: SOAP can be used over any transport protocol such as HTTP, SMTP or even TCP.
  - **Independence**: SOAP allows for any programming model .

SOAP Building Blocks

- A SOAP message is a XML document containing the following elements:
  - A **required** Envelope element that identifies the XML document as a SOAP message.
  - An **optional** Header element that can be used to transmit application specific information (e.g. authentication tokens, payment information, …)
  - A **required** Body element that contains call and response information.
  - An **optional** Fault element that provides information about errors that occurred while processing the message.

- In terms of usage. The envelope should wrap the header, body, and fault elements.

Example SOAP Request

```
POST /InStock HTTP/1.1
Host: www.stock.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 150

<?xml version=1.0 ?>
<soap:Envelope
    xmlns="http://www.w3c.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3c.org/2001/12/soap-encoding">

    <soap:Body xmlns:m="http://www.stock.org/stock">
        <m:GetStockPriceRequest>
            <m:StockName>IBM</m:StockName>
        </m:GetStockPriceRequest>
    </soap:Body>
</soap:Envelope>
```

HTTP Request Header

SOAP Message

Example SOAP Response

```
HTTP/1.1 200 OK
Content-Type: application/soap; charset=utf-8
Content-Length: 126

<?xml version=1.0 ?>
<soap:Envelope
    xmlns="http://www.w3c.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3c.org/2001/12/soap-encoding">

    <soap:Body xmlns:m="http://www.stock.org/stock">
        <m:GetStockPriceResponse>
            <m:Price>102.5</m:Price>
        </m:GetStockPriceResponse>
    </soap:Body>
</soap:Envelope>
```

HTTP Response Header

SOAP Message

WSDL

- WSDL stands for Web Services Description Language.
  - WSDL is an XML Schema for Web services, that allows developers to describe Web Services and their capabilities, in a standard manner.
  - WSDL is a contract that specifies unambiguously what a request message must contain and what the response message will look like.
  - Additionally, WSDL defines where the service is available and what communications protocol is used to talk to the service.
- A WSDL document defines a web service using 4 elements:
  - **port type** - The operations performed by the web service.
  - **message -** The messages used by the web service.
  - **types -** The data types used by the web service.
  - **binding -** The communication protocols used by the web service.
  - **port** – an actual endpoint (i.e. an instance of the service)
  - **service** – a set of endpoints (ports)

(Simple) Example WSDL Document

- Consider a simple Web Service that wants to provide the current price of a stock (given a stock code):

```
public double GetStockPrice(String StockName);
```

- What would the WSDL document for this method look like?

(Simple) Example WSDL Document

- The first part of the document decomposes the method into two messages:
  - The request message:

```
<message name="GetStockPriceRequest">
    <part name="StockName" type="xs:string"/>
</message>
```

  - The response message:

```
<message name="GetStockPriceResponse">
    <part name="Price" type="xs:double"/>
</message>
```

(Simple) Example WSDL Document

- These two messages are then combined to form a portType, which represents the operation(s) supported by the web service:

```
<portType name="StockQuotePortType">
    <operation name="GetStockPrice">
        <input message="GetStockPriceRequest">
        <output message="GetStockPriceResponse">
    </operation>
</portType>
```

- The operation becomes the WSDL equivalent of the original method signature.

(Simple) Example WSDL Document

- Next, the binding is used to specify how the service will be interacted with.

- It specifies the following information:
  - **Transport**: the protocol used – e.g. HTTP, SMTP, FTP, …
  - **Binding**: how the body of the SOAP messages are encoded
    - ***RPC***: The body of the message is derived from the method being invoked.
    - ***Document***: The body of the message is defined using an XML Schema.
  - **Encoding**: how the values passed are encoded
    - ***Encoded***: type information is provided for each parameter / return value.
    - ***Literal***: no type information is provided.

(Simple) Example WSDL Document

- This is a RPC/literal binding for a HTTP-based web service:

```
<binding name="StockQuoteBinding" type="tns:StockQuotePortType">
    <soap:binding
        style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>

    <operation name="GetStockPrice">
        <soap:operation soapAction="http://stock.org/GetStockPrice"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
   </operation>
</binding>
```

(Simple) Example WSDL Document

- Finally, the service part defines the actual deployment of the service (how to access the service and what it does):

```
<service name="StockQuoteService">
    <documentation>
        My really cool first web service
    </documentation>

    <port
        name="StockQuotePort"
        binding="tns:StockQuoteBinding">

        <soap:address location="http://stock.org/stock"/>
    </port>
</service>
```

The Result

- So, if I invoke the method GetStockQuote("IBM"), expecting the price 123.5 to be returned, what messages are generated:
  - The SOAP Request:

```
<soap:envelope>
    <soap:body>
        <GetStockQuote>
            <StockName>IBM</StockName>
        </GetStockQuote>
    </soap:body>
</soap:envelope>
```

  - The SOAP Response:

```
<soap:envelope>
    <soap:body>
        <GetStockQuote>
            <Price>123.5</Price>
        </GetStockQuote>
    </soap:body>
</soap:envelope>
```

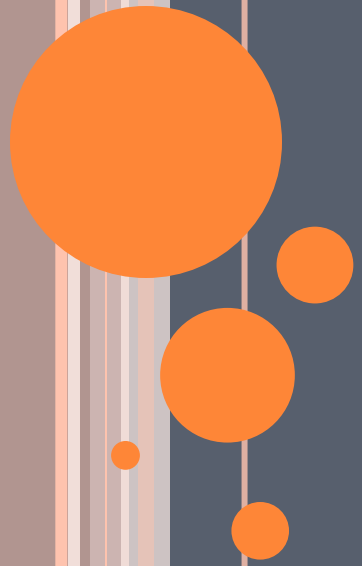
UDDI

- UDDI stands for Universal Description, Discovery and Integration.
  - It is a directory for storing information about web services, like yellow pages (i.e. it is a directory of web service interfaces described by WSDL).

- The vision was always that trusted 3rd parties would provide UDDI services to allow companies to advertise their web services…
  - E.g. http://uddi.xml.org/public-uddi-registry
  - Companies would be able to compose applications by seamlessly combining internal and external web services.
  - Business Workflow languages (e.g BPEL) could be used to specify the architecture – removing the need for the developer!
  - Potentially this could be done dynamically through service orchestration applications (read Classical AI PLANNERs).
  - What about security, trust, provenance of data, …

Its Simple! Right?

- The goal of web services was to provide an architecture to allow machine-2-machine interoperability.
- The intention was always to develop tool support to help developers to build these systems.
  - **Spring** – a Java-based framework for building applications includes a web services dependency that allows you to "easily" deploy web services (you have to learn spring first).
  - **Jax-WS** –A lightweight web services framework that has been a part of Java since version 1.6
- In practice, developers will often "look under the hood".
  - Web Services have got a reputation for being heavy-weight and slow due to their reliance of XML.
  - Few developers use the advanced features, such as UDDI or service orchestration.



JAX-WS

Introduction

- Jax-WS is a fully functional Java API for implementing web services.
- It also includes a set of tools for helping to manage interoperability with web services that were not built using Jax-WS.
  - Jax-WS does not provide full support for web services, for example RPC/Encoding binding types are NOT supported.
- We will cover:
  - Creating and Deploying a Web Service
  - Connecting to a Web Service (built using Jax-WS)
  - Creating to other Web Services.

Creating A Web Service

- We will reuse the stock quotation example from before.
- Similarily to OSGi, to define a Web Service, we must create an interface and implementation.
  - The interface will need to be shared between the service (server) and the service user (client).
- Annotations are used to indicate that interface and class implement a web service.
  - `@WebService` is applied at the class/interface level (define a service)
  - `@WebMethod` is applied at the method level (define operations provided by the service).

Creating a Web Service

```
package quote;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public interface StockService {
    @WebMethod public double GetStockQuote(String StockName);
}
```

Creating a Web Service

- Next, we implement the service (we will return a random number between 100 and 200).
  - Again, we annotate the class using @WebService
  - Additional annotations can be used to customise the deployment:
    - For example, to specify RPC/Literal type message bindings, you should use:

```
@SOAPBinding(style = Style.RPC, use=Use.LITERAL)
```

  - Finally, to run the service, you simply publish an endpoint.

Creating a Web Service

```
package quote;

@WebService(name="StockService")
@SOAPBinding(style = Style.RPC, use=Use.LITERAL)
public class StockImpl implements StockService {
    public static void main(String args[]) throws Exception {
        Endpoint.publish("http://localhost:9000/StockService/GetStockQuote", new StockImpl());
    }

    private Random random = new Random();

    @Override
    public double GetStockQuote(String StockName) {
        double val = 100+random.nextDouble()*100;
        System.out.println("Processed GetStockQuote(" + StockName + ")="+val);
        return val;
    }
}
```

Creating a Web Service: WSDL

- WSDL Url: http://localhost:9000/StockService/GetStockQuote?WSDL
- Have a look – how close is it to the example we did earlier?
- In the StockImpl class, change the SOAP Binding style to `Style.DOCUMENT`.
  - What happens to the WSDL (reload the above url)?
  - Notice the <types> section at the top of the WSDL document and the <xsd:import …> child - copy and paste the url in the `schemaLocation` property to see the XML schema.

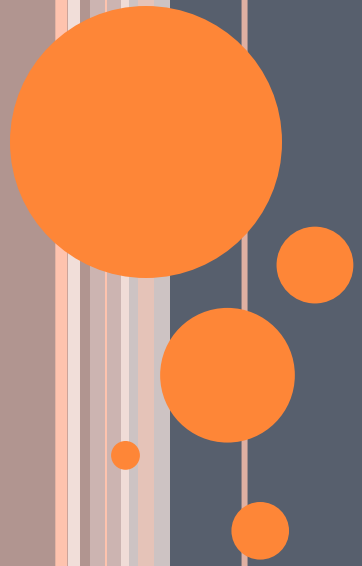
Using a Web Service

- To use a Jax-WS web service, we simply use the service interface to access the client.

- To create the client, you need three pieces of information:
  - The URL of the WSDL file
  - The target namespace – from WSDL file
  - The name of the web service – from WSDL file

Using a Web Service

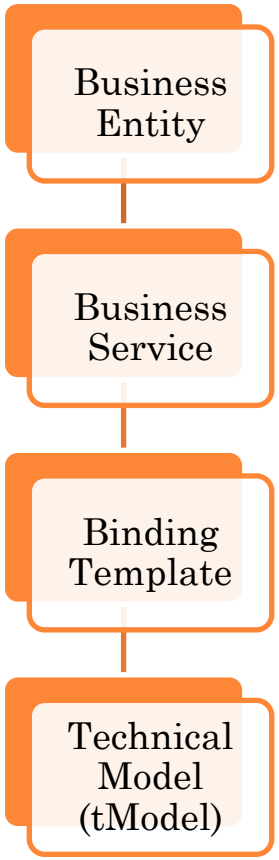
```
package quote;

public class StockClient {
    public static void main(String[] args) throws Exception {
        URL wsdlUrl = new
            URL("http://localhost:9000/StockService/GetStockQuote?wsdl");

        QName qname = new QName("http://quote/", "StockImplService");

        Service service = Service.create(wsdlUrl, qname);

        StockService stockService = service.getPort(StockService.class);

        System.out.println(stockService.GetStockQuote("IBM"));
    }
}
```



jUDDI: A Java UDDI Registry

UDDI: Universal Description, Discovery and Integration

- The registry service for Web Services.
  - Organised around a set of businesses that offer services.
  - Binding templates are used to define how those services are accessed (e.g. WSDL/SOAP bindings).
  - Clarification of how those services are delivered is done through tModels, a type system for UDDI.
    - Many tModel types are specified by default (e.g. SOAP, HTTP)
    - Additional tModels can be specified for new components (e.g. a new port type).
    - tModels are then referenced in a binding template, introducing a constraint on the implementation of that service.
  - UDDI Entries are stored in an XML-based registry.



Example UDDI Registry Entry

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<businessEntity
        businessKey="uddi:juddi.apache.org:ebcf8557-a924-4ded-93bb-f4bc7517ab67"
        xmlns="urn:uddi-org:api_v3" xmlns:ns2="http://www.w3.org/2000/09/xmldsig#">
    <name>Rem's Bright and Shiny WS Emporium</name>
    <businessServices>
        <businessService
                serviceKey="uddi:juddi.apache.org:a78c9355-5a18-4b93-9b13-fbc2fe6f5603"
                businessKey="uddi:juddi.apache.org:ebcf8557-a924-4ded-93bb-f4bc7517ab67">
            <name>HelloWorld Service</name>
            <bindingTemplates>
                <bindingTemplate
                        bindingKey="uddi:juddi.apache.org:9fa8f213-5ae8-4741-8cb4-c9c94b21dc70"
                        serviceKey="uddi:juddi.apache.org:a78c9355-5a18-4b93-9b13-fbc2fe6f5603">
                    <accessPoint useType="wsdlDeployment">http://localhost:9000/HelloWorld/HelloWorld</accessPoint>
                    <tModelInstanceDetails>
                        <tModelInstanceInfo tModelKey="uddi:uddi.org:protocol:soap" />
                        <tModelInstanceInfo tModelKey="uddi:uddi.org:transport:http" />
                    </tModelInstanceDetails>
                    <categoryBag>
                        <keyedReference tModelKey="uddi:uddi.org:categorization:types"
                                keyName="uddi-org:types:wsdl" keyValue="wsdlDeployment" />
                    </categoryBag>
                </bindingTemplate>
            </bindingTemplates>
        </businessService>
    </businessServices>
</businessEntity>
```

jUDDI: A Java UDDI Framework

- jUDDI implements the OASIS specification for UDDI.
  - It is implemented in Java and combines both a fully functional registry and a client based on Apache Scout.
  - Both parts of jUDDI 3.3.4 can be downloaded from moodle.
    - The distro ZIP file contains the server
    - The example-client ZIP file contains some example code & the required uddi-client jar file.
  - Details of jUDDI can be found at https://juddi.apache.org/
    - This includes comprehensive guides on using them
  - To start jUDDI (on windows):
    - Extract the ZIP file
    - Go to juddi-tomcat.. / bin
    - Run startup.bat (.sh for linux/mac)
  - You can access the web-based GUI via: http://localhost:8080/
    - Click on "View the jUDDI User Interface"

Registering a Service (Programmatically)

- To register a service, you must create a UDDIClerk object.

```
UDDIClerk clerk = null;
try {
    UDDIClient uddiClient = new UDDIClient("META-INF/uddi.xml");
    clerk = uddiClient.getClerk("default");
    if (clerk == null)
        throw new Exception("the clerk wasn't found, check the config file!");
} catch (Exception e) {
    e.printStackTrace();
}
```

- The config file “META-INF/uddi.xml” identifies:
  - the instance of jUDDI you are interacting with,
  - what URLs to use to access primary services, and
  - The username / password of the service publisher

Registering a Service (Programmatically)

- Next, you (may) need to create a business entry to create a business key:

```
public static String createBusiness(String businessName, UDDIClerk clerk) {
    // Step 1: Creating the parent business entity that will contain our
    // service.
    BusinessEntity myBusEntity = new BusinessEntity();
    Name myBusName = new Name();
    myBusName.setValue(businessName);
    myBusEntity.getName().add(myBusName);

    // Adding the business entity to the "save" structure, using our
    // publisher's authentication info and saving away.
    BusinessEntity register = clerk.register(myBusEntity);
    if (register == null) {
        System.out.println("Save failed!");
        System.exit(1);
    }
    return register.getBusinessKey();
}
```

- You can skip this if you already have a business key.

Registering a Service (Programmatically)

- Next, you create your service entry:

```
public static BusinessService createWSDLService(String serviceName,
                                                 String myBusKey, String endpointUrl) {
    // Create a new business service
    BusinessService myService = new BusinessService();
    myService.setBusinessKey(myBusKey);
    Name myServName = new Name();
    myServName.setValue(serviceName);
    myService.getName().add(myServName);

    BindingTemplates myBindingTemplates = new BindingTemplates();

    // Create a WSDL/SOAP binding Template
    BindingTemplate myBindingTemplate = new BindingTemplate();
    AccessPoint accessPoint = new AccessPoint();
    accessPoint.setUseType(AccessPointType.WSDL_DEPLOYMENT.toString());
    accessPoint.setValue(endpointUrl);
    myBindingTemplate.setAccessPoint(accessPoint);
    myBindingTemplate = UDDIClient.addSOAPtModels(myBindingTemplate);
    myBindingTemplates.getBindingTemplate().add(myBindingTemplate);

    // Add it to the set of binding templates for the service
    myService.setBindingTemplates(myBindingTemplates);
    return myService;
}
```

Registering a Service (Programmatically)

- Finally, you publish the service to jUDDI:

```
public void publish() {
    try {
        // Optional first step (can replace with literal)
        String myBusKey =
                    WebServicesHelper.createBusiness("My WS Emporium", clerk);

        BusinessService myService = WebServicesHelper.createWSDLService(
                   "HelloWorld Service", myBusKey, HelloWorldImpl.ENDPOINT_URL);
        BusinessService svc = clerk.register(myService);
        if (svc == null) {
            System.out.println("Save failed!");
            System.exit(1);
        }

        String myServKey = svc.getServiceKey();

        clerk.discardAuthToken();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Discovering a Service (Programmatically)

- As with service registration, you must first connect to the jUDDI server:

```
UDDISecurityPortType security = null;
UDDIInquiryPortType inquiry = null;

try {
    UDDIClient client = new UDDIClient("META-INF/simple-browse-uddi.xml");

    Transport transport = client.getTransport("default");

    security = transport.getUDDISecurityService();
    inquiry = transport.getUDDIInquiryService();
} catch (Exception e) {
    e.printStackTrace();
}
```

- For a client, the jUDDI interface consists of a security service (for authentication) and an inquiry service (for service discovery).

Discovering a Service (Programmatically)

- Before you can access the discovery service, you need an authentication token:

```
public static String getAuthKey(UDDISecurityPortType security,
                                 String username, String password) {
    try {
        GetAuthToken getAuthTokenRoot = new GetAuthToken();
        getAuthTokenRoot.setUserID(username);
        getAuthTokenRoot.setCred(password);

        AuthToken rootAuthToken = security.getAuthToken(getAuthTokenRoot);
        return rootAuthToken.getAuthInfo();
    } catch (Exception ex) {
        System.out.println(
                "Could not authenticate with the provided credentials " +
                ex.getMessage());
    }
    return null;
}
```

Discovering a Service (Programmatically)

- Next, you need to find the Businesses you are interested in:

```
public static BusinessList partialBusinessNameSearch(
            UDDIInquiryPortType inquiry,
            String token, String partialName) throws Exception {
    FindQualifiers fq = new FindQualifiers();
    fq.getFindQualifier().add(UDDIConstants.APPROXIMATE_MATCH);

    FindBusiness fb = new FindBusiness();
    fb.setAuthInfo(token);
    fb.setFindQualifiers(fq);

    Name searchname = new Name();
    searchname.setValue(partialName);
    fb.getName().add(searchname);
    return inquiry.findBusiness(fb);
}
```

- This returns a list of matching business entities.

Discovering a Service (Programmatically)

- Now, you can retrieve service information for each business you are interested in:

```
public static ServiceDetail getServiceDetail(
                UDDIInquiryPortType inquiry,
                String token, BusinessInfo info) throws Exception {
    GetServiceDetail gsd = new GetServiceDetail();
    for (int k = 0; k < info.getServiceInfos().getServiceInfo().size(); k++) {
        gsd.getServiceKey().add(
            info.getServiceInfos().getServiceInfo().get(k).getServiceKey()
        );
    }
    gsd.setAuthInfo(token);
    return inquiry.getServiceDetail(gsd);
}
```

- This method constructs a get service detail request for all service keys associated with a given business.
  - The reply contains a list of services.

Discovering a Service (Programmatically)

- For each service description, we can invoke the associated service:

```
for (int k = 0; k < serviceDetail.getBusinessService().size(); k++) {
    BindingTemplate bindingTemplate =
        serviceDetail.getBusinessService().get(k).getBindingTemplates().
            getBindingTemplate().get(0);

    System.out.println("Access: " + bindingTemplate.getBindingKey());

    // Invoke the service…
    URL wsdlUrl = new URL(bindingTemplate.getAccessPoint().getValue());
    QName qname = new QName("http://helloworld/", "HelloWorld");
    Service service = Service.create(wsdlUrl, qname);
    HelloWorld helloWorld = service.getPort(HelloWorld.class);
    System.out.println(helloWorld.sayHi("It's Meee!!!"));
}
```