# INFO-H-511: Web Services
# TP 4 - Services Composition

Lecturer: Stijn Vansummeren
Teaching Assistant: Francois Picalausa
http://cs.ulb.ac.be/public/teaching/infoh511

2011–2012

---

## Part I: Asynchronous calls

Asynchrony is often used to improve application performance. For instance, web browser rely on being able to load asynchronously different parts of a web page (images, scripts, and style sheets) to provide a fast experience to the user. Asynchrony is also used when a requested task cannot be processed immediately. In this first part, We will see how asynchronous SOAP services can be implemented.

Download the Deep Thought service from the labs webpage. This service implements the `DeepThought` interface that contains a `computeAnswer` method which takes a long time to complete. Unless specified otherwise, the `ServiceServer` class will be used to provide this service in the following exercises.

Run each of the following consumers and describe the connection(s) they establish with the service, and the messages they exchange. Also answer the questions for each consumers to understand how they interact with the service.

**Synchronous call** Run the `SyncServiceClient`.

Identify in the code

- where the consumer sends its messages,
- where the service processes the message, and
- where the consumer waits for a response.

**Consumer-side Asynchronous call** Run the `AsyncServiceClient`. Notice that the consumer does some work while waiting for the answer.

JAX-WS relies on `Future`s to call web services asynchronously. Describe in a few words what `Future`s[1] are, based on Java documentation.

Identify in the code:

- where the consumer sends its message,
- where the service processes messages,
- where the consumer processes the answer, and
- how `Future`s are defined and used.

**Message Oriented Middleware** Run the `JMSServiceClient`. Note that is client relies on the `JMSServiceServer`, which uses JMS as a message oriented middelware (MoM) instead of SOAP.

Identify in the code:

- where the consumer and the service connect to the MoM,

[1] `http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Future.html`

- where the consumer sends its message,
- where the service processes the message, and
- where the consumer processes the answer.

**WS-Addressing replies** Run the `WSAServiceClient`. Notice that the console indicates that a new service is created inside the consumer.

Identify in the code:

- where the consumer connects to the service,
- where the consumer creates its own service and sends its message,
- where the service processes the message and how it dispatches the answer, and
- where the consumer processes the answer.

**Comparison** Compare the various approaches in terms of their requirements, their benefits, and their disadvantages.

**(Optional)** Read the Abstract, and sections 3, 7.1, and 7.3 of the PubSubHubbub protocol working draft[2]. How does this compare with the above methods?

## Part II: Messages Routing

In this second part, we will build a web service capable of routing its messages to different implementation. This kind of routing is useful to enable flexible maintenance of services (e.g. switching to new implementation of a service transparently), to enable different quality of services (e.g. redirect to the fast server for clients with a premium account), etc.

In the code skeleton available on the labs webpage, three implementations of a service similar to the Deep Thought service are available, namely `DeepThoughtV1`, `DeepThoughtV2`, and `DeepThoughtDummy`. In our setting, we want to replace `DeepThoughtV1`, with `DeepThoughtV2` transparently, while enabling consumers that specify a certain version in their SOAP header to continue using that version. `DeepThoughtDummy` will be used as an implementation detail for the router. The code skeleton already provides a client implementation that you can use to test your router at each step.

**Installing the router** The skeleton code currently only instanciates `DeepThoughtV1`. Move `DeepThoughtV1` to a new address, and instanciate `DeepThoughtDummy` in its stead. The client should now display "null" as the answer.

**Intercepting messages** The `RoutingInterceptor` class aim is to intercepts incoming messages, and route them to the appropriate service. Register the `RoutingInterceptor` with the `Endpoint` that serves `DeepThoughtDummy`. Complete its implementation by calling `redirect` in the `onMessage` method. Explain why we use `Phase.POST_STREAM` in the constructor of our interceptor.

**Route based on headers** Adapt the `onMessage` to parse the message headers with `getServiceVersion`. Explain why you need to call `bufferMessageStream` before processing the headers.

## Part III: WS-BEPL

WS-BEPL is language for specifying executable web-services-based workflows. In this exercise, we will use Eclipse's WS-BEPL designer to design a few simple workflows.

[2] `http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html`

**Installing Apache ODE** Apache ODE is a system for executing WS-BEPL processes. You will first download the Apache Tomcat application server, and install ODE on it. A guide explaining the different steps involved is available at `http://www-inf.it-sudparis.eu/~nguyen_n/teaching_assistant/bpel`.

**Hello World** Create a new BEPL project containing a single BEPL process. This process will receive a *name* parameter as its input, and will output "Hello *name*". For this purpose, add an "Assign" action to your workflow using the `concat` function of XPath.

**Compositing Services** Start the Deep Thought service defined earlier, after making sure you change the port from 8080 to e.g. 8081 to avoid collisions with Tomcat. Download its WSDL file into your BEPL project and create a new process that:

- Receives a *name* parameter
- Invokes the Hello World process with this *name*
- Invokes Deep Thought
- Outputs "Hello *name*, the answer is: *42*", where *42* is the answer generated by Deep Thought.