

Commonly, an HTTP-based SOAP web service is just a set of operations a client can perform. Each operation has predefined input and output format, and all operation are described in WSDL for discovery purposes. Let's say we have a system that can give us the current weather conditions for any zip-code we provide it with. We want to turn this into a SOAP web service so that others can use the weather information in their applications. This requires an operation that takes zip code as an input and returns weather information as an output.

The first step would be to define its input and output formats. This is done through extending `ComplexType` class and adding in the attributes you need. For the purposes of this example, I'm going to return an array of objects, each of which represents weather data for a particular date.

```
public class WeatherRequest extends ComplexType{ //input
    int zip;
}
```

```
public class WeatherResponse extends ComplexType{ //output
    WeatherInfo[] forecasts;
}
```

```
public class WeatherInfo extends ComplexType{ //part of output
    int forZip;
    String date;
    int temp;
    @Optional String wind; //wind direction can be skipped
}
```

The toolkit supports Java primitives, nested `ComplexType`'s and one-dimensional arrays of either of those. When `MicroWSS` receives a request from the client, it instantiates the request object and fills it with data taken from SOAP message. The response works the same way, just in reverse.

The next step would be to define the logic of the operation. This is done by extending the class `Operation` and implementing its abstract methods:

```
public static class WeatherForecast extends Operation{
    protected WeatherRequest request = new WeatherRequest();
    protected WeatherResponse response = new WeatherResponse();

    public ComplexType getRequest(){
        return request;
    }

    public ComplexType getResponse(){
        return response;
    }

    public void translate(){
        response.forecasts = new WeatherInfo[2];
        response.forecasts[0] = new WeatherInfo();
        response.forecasts[1] = new WeatherInfo();

        response.forecasts[0].forZip = request.zip;
        response.forecasts[0].date = "2009-01-01";
    }
}
```

```

        response.forecasts[0].temp = 20;
        response.forecasts[0].wind = "North-West";

        response.forecasts[1].forZip = request.zip;
        response.forecasts[1].date = "2009-01-02";
        response.forecasts[1].temp = 15;
    }
}

```

Methods `getRequest` and `getResponse` are semi-boilerplate. They map request and response types to the particular operation. It's important to return references to the objects that can be later accessed by the same instance of the operation. The simplest way to do it is to return references to attributes of the operation's subclass itself.

The `translate` method is where application logic takes place. In real life you would replace static data with calls to other parts of the system that actually handles weather information.

Finally, you need to register the operation with a server.

```

Server wsServer = new Server();
wsServer.setTargetNs("http://example.com/targetNamespace");
wsServer.register(WeatherForecast.class);

wsServer.getWsdL(location); //get WSDL, location is a string with server's
address
String responseStr = wsServer.serve(requestStr); //serve a request

```

Technically, this is it. However, in real life you would need to plug the code into something that handles HTTP transactions:

```

public class DemoService extends HttpServlet {

    protected Server wsServer = new Server(); //This is MicroWSS "server"

    @Override
    public void init() throws ServletException {
        super.init();
        // Setting a reasonable namespace is a good idea (though not
mandatory)
        wsServer.setTargetNs("http://example.com/targetNamespace");
        // Need to register all operations before they can be seen/used
        wsServer.register(WeatherForecast.class);
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {

        String location = request.getRequestURL().toString();
        response.setContentType("text/xml");
        try {

```

```

        response.getWriter().write(wsServer.getWsdL(location)); //serve
WSDL
        response.getWriter().flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse
response)
throws ServletException, IOException {
    try {
        String requestStr = slurp(request.getInputStream());
        String responseStr = wsServer.serve(requestStr);
        response.setContentType("application/soap+xml; charset=utf-8");
        response.getWriter().write(responseStr); //serve SOAP response
        response.getWriter().flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static String slurp(InputStream in) throws IOException {
    StringBuilder out = new StringBuilder();
    byte[] bytes = new byte[4096];
    for (int n; (n = in.read(bytes)) != -1;) {
        out.append(new String(bytes, 0, n));
    }
    return out.toString();
}

@Override
public String getServletInfo() {
    return "MicroWSS Demo";
}
}

```

In this example, the servlet would respond with WSDL to all GET requests, and handle SOAP messages via POST. Naturally, you can register many operations with one MicroWSS “server.” You don’t have to register them directly in the servlet code either. Normally, you would write something similar this servlet first, and then load new operations programmatically.