

Chapter 13

Water Protocol: Creating Custom Protocols – SOAP

IN THIS CHAPTER

- ◆ Describing a protocol and its Stages
- ◆ Customizing a protocol

A **PROTOCOL** IS THE PROCESS AND FORMAT for calling resources. Every type of resource requires a protocol to communicate with it. A protocol for local resources might access files and databases. A remote protocol is used to access remote resources. The most common protocols for accessing Web resources are HTTP and SMTP. Other standard Web protocols are FTP and SOAP.

Water Protocol describes how to define new protocols and customize standard protocols.

Looking at a First Example

HTTP is by far the most common protocol for making a remote resource request. The following example first creates a new Web resource that uses the HTTP protocol. A request is made to that remote resource by using HTTP.

```
<set water_lang_home=  
    www.<web "http://www.waterlang.org/" />  
/>  
water_lang_home.<request/>
```

The return value is a `request_response` instance that contains the HTML page in the `a_response.body_string` field.



Protocol is the data format and process for interacting with a remote resource. Every protocol has a unique name. The protocol name is used to look up the protocol handler method.



A *Web protocol* is a protocol for accessing Web-based resources. Examples include HTTP, FTP, and SMTP.



A *request* gets the value of a remote resource, or calls the remote resource with arguments.

Overview of a Protocol

A protocol requires both client-side and server-side components. The protocol on the client-side knows how to send a request to the server by using some protocol and also how to process the response from the server. The protocol on the server-side knows how to handle a request coming from a client, process the request, and return a response.

The server-side protocol handling is described in Chapter 14 on Water Server. This chapter explains the client-side protocol processing.

When a request is called on a Web resource, a `request_response` instance is created. A `request_response` has two fields: `a_request` and `a_response`. The basic contract is as follows:

```
<defclass request_response a_request a_response />
```

A client processes a remote request by calling `request` on a specific protocol class such as `http_protocol`.

The `request` can be any code, but in most cases it validates the request to make sure it obeys the contract for the remote resource. If so, then it creates a `request_response` with a valid request and an empty response and calls `protocol_pipeline`.

The `protocol_pipeline` invokes a series of methods that process the `request_response` and return a result. These processing stages of `protocol_pipeline` can be individually customized. Customizing the pipeline is described further in the chapter.

```
web.http_protocol.  
<request_response  
  a_request=<uri> http://localhost/boston </uri>  
  a_response=<response content_type=null result=null/>  
>.<protocol_pipeline/>
```

The following code shows the outline for the `http_protocol` class. The class defines a `request_response` subclass, and the top-level `request` method.

```
web.  
<defclass http_protocol>  
  <defclass response content_type=null result=null/>  
  
  <defclass request_response>  
    a_request=required=uri  
    a_response=required=response  
  </defclass>  
  
  <defmethod request args=null>  
    subject.<request_response  
      a_request=<if> args.<is null/> .a_uri  
        else .a_uri.<uri query=args/>  
      </if>  
      a_response=<response content_type=null result=null/>  
    />.<protocol_pipeline/>  
  </defmethod>  
  
  <defmethod protocol_pipeline>  
    subject.<encode/>.<make_request/>.<decode/>  
  </defmethod>  
  
  <defmethod encode/>  
  <defmethod make_request/>  
  <defmethod decode/>  
  
</defclass>
```

Stages of a Web Protocol

The `protocol_pipeline` invokes a series of methods that process the `request_response`. The standard methods are `encode`, `make_request`, and `decode`. The output from the last stage of the pipeline is returned to the `request` method. The standard `protocol_pipeline` is as follows:

```
<defmethod protocol_pipeline>  
  subject.<encode/>.<make_request/>.<decode/>  
</defmethod>
```

The `subject` is an instance of `request_response`. The `request_response` flows through the pipeline stages and returns a `request_response` that contains

`a_response` field containing a response instance. An example of a completed `request_response` is as follows:

```
web.http_protocol.
<request_response
  a_request=<uri> http://localhost/boston </uri>
  a_response=<response content_type="text/html"
                    body_string="<HTML> ... </HTML>"
                    />
/>
```

Custom Protocol Pipeline

Water Protocol makes it possible to customize the standard pipeline processing. This is useful in many cases where a request to a Web resource should return an object that is not a `request_response` object. For example, when calling a temperature service, you might want to get back an integer.

```
<current_temperature "boston"/> → 60
```

By default, though, a full `request_response` object is returned that contains a response with the temperature encoded in a string.

```
<current_temperature "boston"/> →
<request_response
  a_request=<uri "http://some_temperature_service/" />
  a_response=.<response
                content_type="text/xml"
                body_string="<do>60</do>"
                />
/>
```

The `body_string` field of response is the string "`<do>60</do>`" \, which is a valid XML 1.0 representation for the integer 60. The goal is to return the integer 60 instead of a `request_response` object. This can be accomplished by customizing the `decode` method of the protocol.

The default `decode` method just returns the subject, which is the `request_response` object.

```
web.http_protocol.
<defmethod decode>
  _subject
</defmethod>
```

To return an integer, a new `decode` method needs to be created that takes a `request_response` as a subject and returns an integer.

The `a_response.body_string` in the `request_response` is a string of XML, "`<do>60</do>`". That string needs to be converted into an integer. The following example calls `execute_string` on an XML string and returns a `Water` object.

```
"<do>60</do>".<execute_string/> → 60
```

The next example uses `execute_string` in a customized version of `decode`.

```
web.http_protocol.  
  <defmethod decode>  
    subject.a_response.body_string.<execute_string/>  
  </defmethod>
```

That is all that is required to customize the protocol pipeline. Another use for customizing the `decode` method is to convert an HTML page containing a table into a vector of `Water` objects. For example, a search might return a list of books formatted in an HTML table. The `decode` needs to take the HTML page and return a vector of book instances. Each book instance was created from a row of the HTML table.

The `encode` method might change if you want to check the local data cache before making an expensive remote call. Another example is converting arguments into a SOAP envelope and body.

The entire `protocol_pipeline` method can also be replaced. This is useful for supporting new protocols – for example, a protocol to support asynchronous calls or requests that request multiple handshakes for authentication.

Summary

Each stage of a protocol can be modified, as well as the entire pipeline. With the flexibility in Water Protocol, you could create a protocol that sends a request over SMTP and gets the result asynchronously via FTP. The ability to create variations of standard protocols will open up new ways for businesses to share data and processes.

This chapter covered Water Protocol and described how to easily create new protocols. The `http_protocol` method is automatically called when a URI starts with `http:.` Each stage of a protocol (`encode`, `make_request`, `decode`) can be changed, and the entire pipeline can also be customized by changing the `protocol_pipeline` method and the `request_response` class. With the flexibility in Water Protocol, you could create a protocol that sends a request over SMTP and gets the result asynchronously via FTP. The ability to create variations of standard protocols will open up new ways for businesses to share data and processes.

