

## Chapter 14

# Water Server: Creating Application Servers

### IN THIS CHAPTER

- ◆ Installing and configuring a Water server
- ◆ Calling a remote method through a URI
- ◆ Requesting an object through a URI
- ◆ Creating a new instance through a URI
- ◆ Customizing a Water server pipeline

**WATER SERVER IS THE WATER STANDARD** for creating and configuring custom servers for applications and Web services. Water methods are used for all server operations.



A *server* responds to network requests and can be considered a method that resides on another machine.

---

When a user clicks on a hypertext link or clicks on a button to submit a form, the Web browser makes a network request to contact a remote server. The remote server has an active process that is listening for requests on a particular port. When it receives a request, it processes the request and returns a result. A client machine calls the remote method and the remote method returns a result.

## Creating a Water Server

First, create a new HTTP server that listens on port 80 and serves the `thing` object.

```
<server thing/>
```

Second, type the following address in any Web browser to return the default page from the server.

```
http://localhost/ → No default_result.
```

The result "No default\_result" is returned because the URI (also known as a URL) did not call a specific method or access a particular field. The server can be created with any default result by providing a value for the argument `default_result`.

```
<server default_result=<H1>Some other default result</> />
```

Create a new method that can be called through the Water server.

```
<defmethod double x>  
  x.<times 2/>  
</defmethod>
```

The method `double` takes a single argument `x` and returns `x times 2`. The method could be called locally as in the following example:

```
<double 10/> → 20
```

Because every Water method is a Web service, no additional steps are required to call this method remotely. From a Web browser, type the following address:

```
http://localhost/double?x=10 → 20
```

The browser window should show 20. The address `http://localhost/double?x=10` has three main parts for the remote call: the protocol `http`, the machine `localhost`, and the specific resource to request `double?x=10`. The question mark means that the request is a method call. The part after the question mark represents the arguments to the call. The following example is the form for calling a Web service method with a URI.

```
http://domain_or_ip_address/method?arg_1_key=arg_1_value&arg_2_key=arg_2_value
```

A call without any arguments will have a URI that ends in a question mark.

```
http://localhost/some_method?
```

Creating a new Web service and configuring a Web server will often take hours using other technologies. One great property of Water is that simple things really are simple.

## Serving XML Object

A URI represents an identifier to a remote resource. A request can be made to a remote resource and a single value is returned. The next group of examples shows how to create a `Water` object, make it available as a Web service, and call it from a browser.

First, create a vector of three string objects, "Water", "SmallTalk", and "Java". Second, assign the vector to a field named `oo_languages`.

```
thing.<set oo_languages=<vector "Water" "SmallTalk" "Java"/> />
```

`oo_languages` is now a field in `thing`. To return the value of the object, just type the key of the field or variable. This is the local Water Identifier or symbol. In the preceding example, the symbol is `oo_languages`. Executing that symbol will return the value in that field.

```
thing.oo_languages → <vector "Water" "SmallTalk" "Java"/>
```

To return the value of the object by using a remote HTTP call, simply request the symbol by using a URI.

```
http://localhost/oo_languages
```

**Output as XML 1.0:**

```
<vector>
  <attributes> "Water" "SmallTalk" "Java" </attributes>
</vector>
```

The preceding output is an XML 1.0 representation for a `Water` vector. The use of `attributes` is described in Chapter 2 on ConciseXML.

For objects nested within other objects, use a remote Water Identifier in the URI. The following example creates the nested object:

```
thing.<set a_object=
  <thing oo_languages=
    <vector "Water" "SmallTalk" "Java"/>
  />
/>
```

The object can be returned over HTTP by typing the following address into a Web browser. The path is relative to the `root_object` of the server, which was set to `thing`. The first item in the URI path is the field `a_object` in `thing`.

```
http://localhost/a_object/oo_languages →
<vector>
  <attributes> "Water" "SmallTalk" "Java" </attributes>
</vector>
```

In the preceding example, the `root_object` is `thing`, `a_object` is a field in `thing`, and `oo_languages` is a field in the `a_object` that contains a vector.

## Creating a New Instance

A Water call is either a method call or a constructor call. You will not be able to tell what a call represents based on the syntax, as a method call and constructor call have the same structure and syntax. In the following example, the first line creates a method named `foo_method`. The second line calls the method with the argument `x="test"`. The method call executes the method implementation and returns the result.

```
<defmethod foo_method x> x </>
<foo_method x="test"/> → 10
```

Compare the preceding call method example to the following call constructor example and notice that the second lines of each example have exactly the same structure. The first line creates a class object named `foo_object`. The second line calls the default initialization constructor (the `init` method) for `foo_object` with an argument `x="test"`. The constructor call executes the `init` method, which returns the new instance.

```
<defclass foo_object x/>
<foo_object x="test"/> → <foo_object x="test"/>
```

By default, calling the `init` method returns a new instance where all the arguments of the call become fields in the new instance.

Because calling a method `<foo_method x="test"/>` and calling a constructor `<foo_object x="test"/>` have the same structure, the description for how to call a method using a URI that was discussed in the previous section also applies to calling a remote constructor.

The following URI is a remote request that creates a new instance of `foo_object`

```
http://localhost/foo_object?x=test
```

The following line also creates a new instance of `foo_object`, but it uses a local Water call expression.

```
<foo_object x="test"/>
```

### Accessing Instances from a URI

A common Water design pattern is for an `init` constructor method to assign the new instance to a field within the class. This lets you access all instances of a class through the class object. All instances are stored in the field named `of` in the class object.

```
<customer id=101 name="Acme Co"/>
<customer id=102 name="Water Works"/>
```

The preceding code creates two instances of `customer`. The following examples are how you want to reference those instances. Defining the `init` method for `customer` in the following example is required for these references to work.

```
customer.of.101 → <customer id=101 name="Acme Co"/>
customer.of.102.name → "Water Works"
<defclass customer id name>
  this.<set of=<thing/> />
  <defmethod init>
    customer.of.<set_value .id this/>
    this
  </>
</defclass>
```

Every time an instance of `customer` is created, the instance is stored in a field within the `customer` class – specifically within the `of` field of the class. The name of the field is the `id` field of the instance. The `init` method could use any field, but the simple implementation above just uses the `id` field.

This convention makes it straightforward for accessing all instances of a class from a URI.

```
http://localhost/customer?id=200&name=Steam+Cleaners
"http://localhost/customer.of.200" → <customer id=103
name="Steam Cleaners"/>
```

## Configuring a Water Server

A server can be configured with several arguments. The most common ones are shown with their defaults as follows:

```
<server root_object=required
  root_folder=null
  port=80
  default_result="No default_result"
/>
```

## Port

A server listens on the port specified in the `port` argument. By default, a server listens on port 80. The following example creates a new server that listens on port 8080.

```
<server thing port=8080/>
```

An HTTP request specifies the port number in the URI.

```
http://localhost:8080/
```

## root\_object

The `root_object` specifies the object to which all incoming requests are relative. Many Web servers have the concept of a root folder where all file requests are relative to that folder. `root_object` is similar in concept, except that instead of a base folder, there is a base object. All requests will be relative to that object. For example, if the `root_object` is `app1`, and the URI is `http://localhost/page1`, then `page1` is a field of `app1`. You will never see an explicit reference to `root_object` in the URI.

The `root_object` on a server could be assigned to either a method object or any other object. If `root_object` is a method, then the only thing the server can do is execute that one method. If `root_object` is a non-method object, then only fields and subfields of that object can be accessed. If the object is an application, the fields of the application object might be subapplications or methods.

The following example creates a method `my_service`.

```
<defmethod my_service x> x.<concat " test"/> </defmethod>
```

Create a new server in which the `root_object` is the `my_service` method defined above.

```
<server my_service/>
```

The method can now be called on the server with a URI.

```
http://localhost/?x=10 →  
10 test
```



Regarding security: Access can be limited to a specific method. The preceding URI does not have a path because the `root_object` of the server specifies the method to call—only the arguments need to be given. This is particularly useful for high security systems because a single service can reside on a single port. The firewall can be configured to allow only specific requests to be sent to the server.

---

If the server was created with `thing` as the `root_object`, then you would need to specify `my_service` in the URI.

```
http://localhost/my_service?x=10
```

## Creating an Application Server

`root_object` is also used to create a server for an individual application. The following code defines an application named `my_app`. The application has two pages named `first_page` and `second_page`. The `first_page` has a link that will jump to the `second_page`.

```
<defclass my_app
  first_page=<HTML><BODY>
    <H1>First Page</>
    <A href="/second_page?">Second Page</A>
  </BODY></HTML>
>
<defmethod second_page>
  <DIV> <H3>Second Page</>
    <A href="/first_page">Go to first page</A>
  </DIV>
</defmethod>
</defclass>
```

The following creates an application server for the `my_app`:

```
<server my_app/>
```

Any URI request to the server will only need to specify the field within `my_app` because every request is relative to the `root_object` of `my_app`.

```
http://localhost/first_page
```

You can create subapplications within `my_app`. For example, if `my_app` is the top-level application, other applications would be fields under `my_app`.

## Content Type and Format

Every response returned from the server will specify the type of content being returned. An HTML page, for example, will have `content_type="text/html"`, a GIF image will have a `content_type="image/gif"`, and an XML document will have `content_type="text/xml"`. The `content_type` is used by the client to select the program to render the content.

By default, Water will automatically set the `content_type` based on the type of object returned by the server.

The `content_type` can be explicitly set for the server so that all responses have a single `content_type`. This would be useful if you had a dedicated server that only returned a specific type of response.

## Custom Water Server Processing

When a Water server receives a request, the request is processed through four sequential stages. This process is implemented in a method called `server_process`. The standard processing has four stages:

1. *decode*: uses the URI to figure out what code to execute
2. *execute code*: executes the code and handles errors
3. *encode*: transforms the result object (optional)
4. *format*: converts result object into a string

Water Server lets you change any one of the stages or the entire process. This gives you incredible flexibility for creating custom servers.

A `request_response` object is created and passed from stage to stage for processing. Each processing method requires a `request_response` instance as input and output. It has the following definition:

```
<defclass request_response
  a_request=required=uri
  args=null
  a_response=null
  code='system_use_only'
/>
```



A `request_response` is an object with `a_request` and `a_response` fields that is passed through the server processing stages.

---

The first processing method is called `decode`. It looks at the `a_request` field and sets the `code` field of the `request_response` to the Water expression to execute.

```
server.
<defmethod decode>
```

```
_subject.<set code=.a_request.<to_expr_string/>.  
  <execute execution_kind='ekexpression' /> />  
_subject  
</defmethod>
```

The second processing method is `execute_code`. This method executes the Water expression in the `code` field.

```
server.  
<defmethod execute_code>  
  _subject.a_response.<set result=  
    <if> .code.<is null/>  
      .default_result  
    else  
      .code.<execute/>  
    </if>  
  />  
_subject  
</defmethod>
```

The third processing method is `encode`. It transforms the `a_response.result` field. It could, for example, wrap the result within another object or change the XML encoding of the return value.

```
server.  
<defmethod encode>  
  _subject  
</defmethod>
```

The last stage of processing is done by `format`. It takes `a_response.result` and converts it to a string to send back in the body of the response. `format` is also responsible for setting `content_type` for the response.

```
server.  
<defmethod format>  
  <set a_formatter=.<get_formatter/> />  
  .a_response.<set body_string=.<get_body_string a_formatter/> />  
  .a_response.<set content_type=.<get_content_type a_formatter/> />  
  _subject  
</defmethod>
```

## Creating a Custom Water Server

Any method in the Water Server processing pipeline can be replaced, as well as the whole server process pipeline named `server_process`. The default `server_process` method is

```
server.<defmethod server_process>
  _subject.<decode/>.<execute_code/>.<encode/>.<format/>
</defmethod>
```

The following example replaces the entire pipeline by a method that will inspect the transaction and return it without modifying any data.

```
<server thing server_process=<defmethod> .ii </> />
```

The following example creates a new class of server that incrementally builds a page over multiple transactions.

```
server.
  <defclass incremental_page
    server_encode=<defmethod add_to_page_and_display_page>
      .page.<insert .a_response.result/>
      .a_response.<set result=.page/>
      _subject
    </>
    page=<BODY/>
  ></defclass>
```

## Creating a Pure Water Server

A Pure Water Server accepts as input any Water expression in ConciseXML and returns a ConciseXML response. For example, instead of a URI of `http://localhost/foo?bar=10`, you would use `http://localhost/<foo bar=10/>`. Instead of a response in XML 1.0 syntax, `<do> 999 </do>`, the response would be in ConciseXML syntax, `999`.

The following code defines `server.pure_water`.

```
server.
  <defclass pure_water
    formatter="to_concise_xml"
  >
  <defmethod server_decode>
    .<set code=.a_request.path_string.<concat .a_request.query_string/>.
      <execute execution_kind='ekexpression'/>
```

```
/>  
  _subject  
</defmethod>  
</defclass>
```

To create a new server:

```
<server.pure_water root_object=thing/>
```

Now, a Web browser's address bar becomes a Water command-line interface to the Water Server.

## Summary

A Water Server makes an object, `root_object`, available through some protocol and port. The object can be anything, but is typically a class or method. A request to the Water Server is typically a URI. The URI is converted into a Water expression and then executed. The entire server process is controlled by the `server_process` method. The whole pipeline can be customized as well as any stage of the pipeline.

Water Server significantly reduces the complexity of creating a Web service server. Some tools and products may support a “one-click” approach to creating a new Web server, but the customization is severely restricted and the generated code is cumbersome to maintain and extend. Water Server eliminates the need for any code generation and makes creating a custom Web Server as easy as creating a Web page.

