Tutorial: Designing Custom Template Processing Languages Using XML

Christophe Chardonnet

Technical Consultant, OmniMark

Outline

- Introduction
  - Template based approach
  - Programming based approach
  - Mixed approach: custom template language
- Technical environment
  - OmniMark language
  - CGI programs
  - Server daemons

Outline

- Practical example:
  - Programming approach
  - Template: Basic substitution
  - Centralized processing
  - Using referents
- CCL: Content Control Language

Introduction

Why talk about template languages ?

- Because our professional service team has found them to be an ideal solution in several major projects
- Because they illustrate the power of the streaming programming model

Why talk about template languages ?

- Because they are an effective solution you probably would not even contemplate in another language
- Let programmers program and designers design

Building interactive web applications

- Two approaches to building interactive web applications
  - Use page templates with embedded code
    - ASP
    - Cold Fusion
  - Write CGI or servlet programs
    - OmniMark
    - Perl
    - Java

Template based approaches

- Quick and easy to start
- Non-programmers can do a lot of the work

- But:
  - Complex operations are difficult to code
  - Code base gets scattered all over your web site, creating maintenance nightmare

Programming based approaches

- Provide maximum processing power
- Keep your code organized

- But:
  - Lack flexibility
  - Require programmer intervention to change even simple elements
    - Designers and content providers can't get their work done
    - Programmers are bogged down in boring maintenance work

Mixed approach

- Build template processing into your applications
- Give designers and content providers the flexibility they need, without writing the whole application in template pages

Mixed approach

- Keep control over your code base
- Design your template functionality to suit:
  - your business practices
  - division of responsibilities in your organization
  - skill level of your colleagues

Tutorial: Technical environment

Tutorial: Practical application

- Let ’s design some custom template language
- Use of the OmniMark language
- CGI or Web daemon programs

OmniMark language

OmniMark language

- XML and text programming language
  - filters, batch conversions
  - CGI, servers
- Streaming programming model
- Rule-based program structure
- Integrated XML and SGML parsers

Streaming model

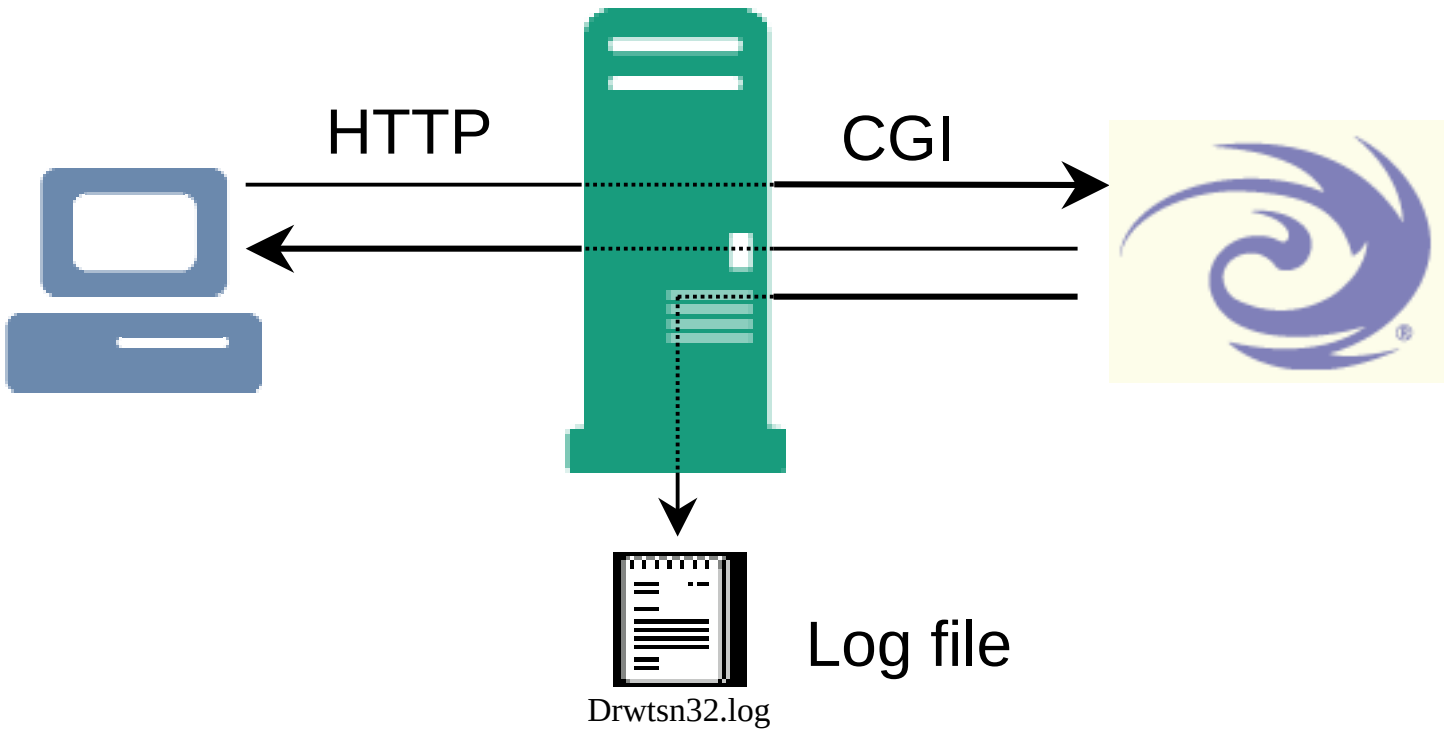
- Data is streamed from a source
- Working process as it flows
- Data streams directly to output
- No buffering of input or output

Rule based

- Program execution begins in a `process` rule
- Invoke the markup processor (XML or SGML)
  - do xml-parse ... or do sgml-parse …
  - write `element` rules
  - hierarchical streaming parsing model
- Invoke the pattern processor
  - submit file …
  - write `find` rules

CGI programs

OmniMark CGI program



From browser to browser …

- Browser accesses a URL that points to a CGI program
- Server activates the program and uses the CGI protocol to pass the data sent by the client to the CGI program
- CGI program acts on the data and does the required processing
- CGI program returns a response to the web server using the CGI protocol
- Web server sends the response to the client using HTTP

HTTP

- Hypertext Transfer Protocol
  - Communication standard between browsers and web servers
  - Stateless information retrieval network protocol
  - Based on TCP/IP (Transmission Control Protocol/Internet Protocol) connections

Common Gateway Interface

- The Common Gateway Interface is a protocol that allows web servers to communicate with other programs
  - Web server invokes the program and sends input data to the program
  - The program runs and output data is sent to the web server

HTTP communication

- That's the job of the web server!
- But you need to know …
  - How to specify and analyze requests (URL, HTML forms, decoding data …) from a browser
  - What the web server needs to send to the browser

CGI communication

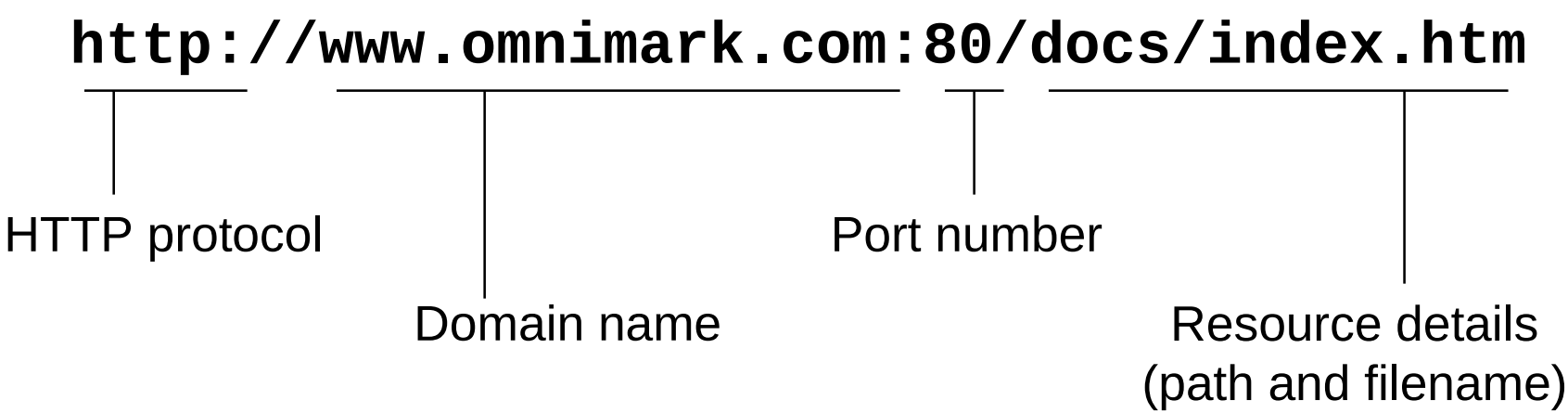
- The web server should invoke OmniMark whenever an OmniMark CGI program is requested
  - Implies setting up the web server
- Passing data is standardized
  - Know how web servers pass input data
  - Know what web servers expect from CGI

The browser can request …

- An OmniMark program directly
  - Conventional extension is "`*.xom`"
- An OmniMark arguments file
  - Conventional extension is "`*.xar`"
  - Tells where to find the OmniMark program
  - Can contain many other options
  - Is the preferred method because it is more flexible

URL

- Uniform Resource Locator
  - An address scheme for specifying Internet resources



URL

- URL components
  - Protocol: HTTP, FTP, etc
  - Domain name: the site on which the server is running
  - Port number: port at which the server is listening
  - Resource details:
    - file
    - gateway program
    - …

URL to invoke a CGI

- Invoking an OmniMark file

```
http://www.omnimark.com/scripts/hello.xom
```

- Invoking an OmniMark arguments file

```
http://www.omnimark.com/scripts/hello.xar
```

Web server invokes OmniMark because …

- On UNIX and Linux systems
  - The `#!` directive appears on the first line of the CGI program and tells the web server where to find the OmniMark executable:
    - OmniMark program

```
#!/usr/bin/omnimark/omnimark -sb
```

    - OmniMark arguments file

```
#!/usr/bin/omnimark/omnimark -f
```

Web server invokes OmniMark because…

- On Windows operating systems
  - Depending on the requested file extension, you need to set up the web server to invoke a program
  - Set up the web server by associating file extensions with the OmniMark executable:

```
".xom" = "omnimark.exe -sb %s"
".xar" = "omnimark.exe -f %s"
```

Command line

`omnimark` is OmniMark C/VM (compiler/ virtual machine)

`-sb program.xom`

- Run the program "`program.xom`"

`-f arg.xar`

- Pass parameters through `arg.xar` arguments file. This file contains command-line options including

  `-sb program.xom`

Using OmniMark VM

- You can produce a compiled program from your source code using OmniMark C, and use OmniMark VM to invoke it

```
omnivm -load prog.xvc
```

- That way, your code is protected

Example: setting up IIS

- It depends on the version of IIS, but for mine...
  - Open the “IIS Management Console”
  - Select “Default Web Site”
  - Right-click on “Properties”
  - Select “Home Directory” and then “Configuration”
  - …

Example: Setting up IIS … continued

- You will see a list of associations between extensions and executables
- Add one for “*.xom” files

```
*.xom = omnimark -sb %s
```

- Add one for “*.xar” files

```
*.xar = omnimark -f %s
```

Directories and virtual directories

- Home directory: is directly mapped to your domain name
- Virtual directory: choose a mapping between a physical directory and an alias name
- Directory permission: read, execute, and script

File and directory permissions

- Invoked file must be in a web server directory with “script” permissions
- Invoking an arguments file
  - Web server must be able to execute and read the arguments file
  - OmniMark program must be readable
- Invoking a program directly
  - Web server must be able to execute and read the OmniMark CGI program

Giving permissions to a web directory

- By convention the “script” or “cgi” directories are set up with script permission
- You can choose permissions for each directory
- Subdirectories inherit permissions
  - You can organize different projects in different directories

Let's invoke an OmniMark program!

- First OmniMark CGI program:

```
#!omnimark -sb
process
output "Content-type: text/plain"
    || "%13#%10#" ||*2
    || "Hello you!"
```

Let's invoke an arguments file!

- Arguments file `hello.xar`

```
#!omnimark.exe -f
-sb hello.xom
```

- OmniMark program `hello.xom`

```
process
output "Content-type: text/plain"
    || "%13#%10#" ||*2
    || "Hello you!"
```

Error messages

- Handling CGI program error messages
  - If your program has an error, the result shown in the browser is not always helpful
  - Often displays nothing more than:

"CGI program returned an incomplete set of HTTP headers"

Using “-log” or “-alog”

- Causes all OmniMark error messages to be written to the specified file
- Use arguments files!

```
#!omnimark.exe -f
-sb hello.xom
-log d:\inetPub\logs\omni\hello.log
```

Create arguments files

- You can use the “project files” the IDE creates
  - Save them as “`*.xar`”
  - Modify them to add more options such as the `-log` option

CGI directories organization

- READ access for static pages
- “root” directory for static pages

  `C:\InetPub\wwwroot\cgiclass`

Access it with

`“http://localhost/cgiclass/file.htm”`

or

`“http://127.0.0.1/cgiclass/file.htm”`

SCRIPT access for CGI

- “script” directory for “*.xar” files
  `C:\InetPub\cgi\cgiclass`
- Access it with
  `http://localhost/cgi/cgiclass/file.xar`

or

`http://127.0.0.1 /cgi/cgiclass/file.xar`

Sending output data

- Any data your CGI program sends to standard output is sent to the browser
- `output "foo"` writes `"foo"` to `#current-output` which is, by default, connected to the standard output

Standard output and CGI

- `#process-output` should be set to binary-mode, and can also be unbuffered

```
declare #process-output has binary-mode
declare #process-output has unbuffered
```

What about HTTP?

- The web server should understand data sent by your program
  - Should be HTTP compliant

HTTP response

- HTTP header tells the browser how to interpret incoming data
- A blank line following the HTTP header tells the browser that the header is complete
- All other data sent to standard output will be interpreted by the browser — this is the data you want to display

Basic HTTP headers

- HTTP header examples
  - `Content-type: text/plain`
  - `Content-type: text/html`
  - `Content-type: image/gif`
  - Cookies ...

```
output “Content-type: text/plain”
     || “%13#%10#” ||* 2
```

OmniMark CGI example

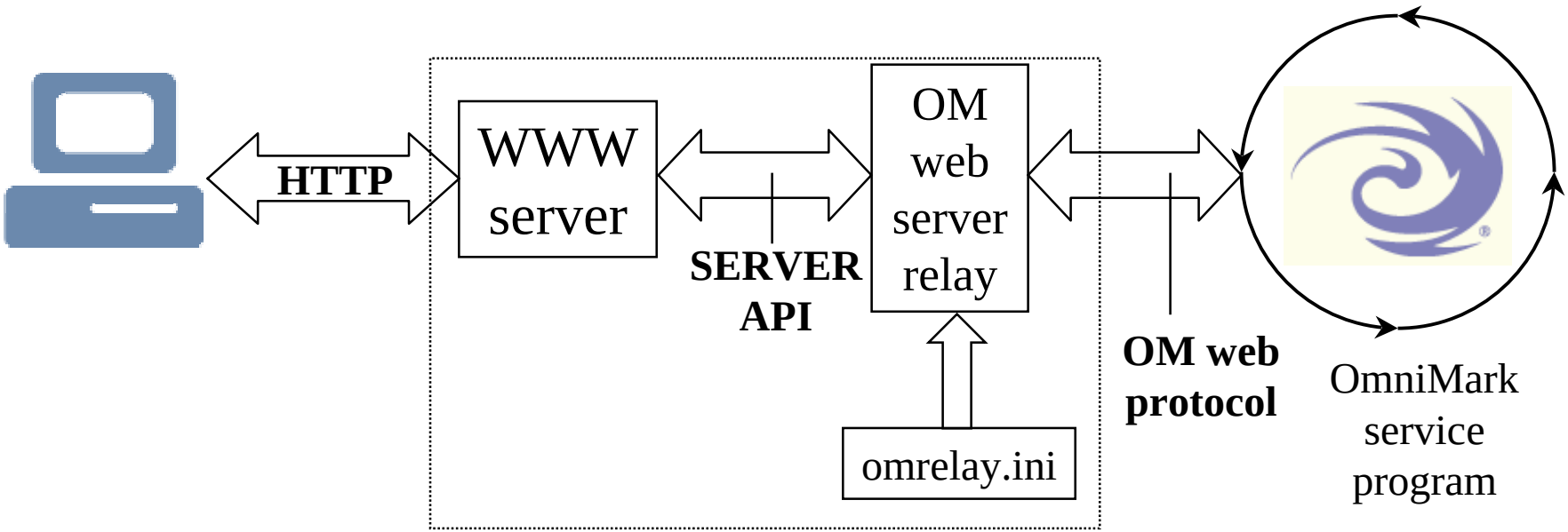
```
declare #process-output has binary-mode

macro CRLF is “%13#%10#” macro-end

process
   output “Content-type: text/plain”
        || CRLF ||* 2
        ||“Hello OmniMark Developers!”
```

OmniMark service daemons

CGI request

- Shortcomings:
  - Gateway programs are server-side programs that must run on the same machine as the web server. Therefore, they do not allow for distributed processing.
  - Gateway programs are not scalable — there is a limit on how many of these programs can run simultaneously
  - Gateway programs start and stop each time a new request arrives

OmniMark service programs in a web environment



OmniMark service programs …

- … are server programs listening 24/7 for a request on a TCP/IP port
- When a request arrives, the server accepts it, processes it, and sends the response
- OmniMark service program is then ready again to accept a new request
- Communication (request/response) is compliant with the OmniMark web protocol

OmniMark web server relays

- OmniMark web server relays handle communication between the web server and OmniMark service programs
  - `OMCGIR` for any CGI-compliant web servers

OmniMark web protocol

- The OmniMark web protocol is a client/server protocol that describes the mechanism the OmniMark web server relay uses to communicate with OmniMark service programs

OmniMark service URL

The general format for an OmniMark service URL is:

```
http://host:port/om-webserver-relay/omnimark-service-name?data
```

For example:

```
http://www.omnimark.com/omcgir.exe/omecho?first=Billy&last=Jones
```

Passing parameters: forms

- See the `action` and `method` parameters

```
<FORM METHOD=post ACTION="/bin/omcgir.exe/omecho?">
<P>You are:<BR>
    <SELECT NAME= "who">
     <OPTION VALUE="Joe">Joe
     <OPTION VALUE= "William">William
     <OPTION VALUE= "Jack">Jack
     <OPTION VALUE= "Averell">Averell
    </SELECT>
</FORM>
```

CGI vs OmniMark web service

CGI Program

- Standalone program
- Expensive process (CPU, memory)
- Limit on the number of CGI programs that can run
- CGI programs run on the same machine as the web server
- No opportunity for distributed computation
- Buckles under peak load
- Quick to write

OM Web Service

- Web server extension
- Lightweight thread
- OM servers can run on the network on many machines
- Allows for extensively distributed computing
- Gracefully handles peak load
- Do initialization only once
- Keep other connection alive

CGI vs OmniMark web service

- CGI is good for quick applications that are not invoked very often or don't need a lot of resources
- OmniMark web services are good for crucial applications that need a quick answer, run 24/7, or may need a lot of initialization

omasf

- `omasf.xin` is an OmniMark program template used to write web server programs communicating with OmniMark web server relays
- To define your own logic, just implement three functions in `omasf.xom`
  - `ServiceInitiate`
  - `ServiceTerminate`
  - `ServiceMain`

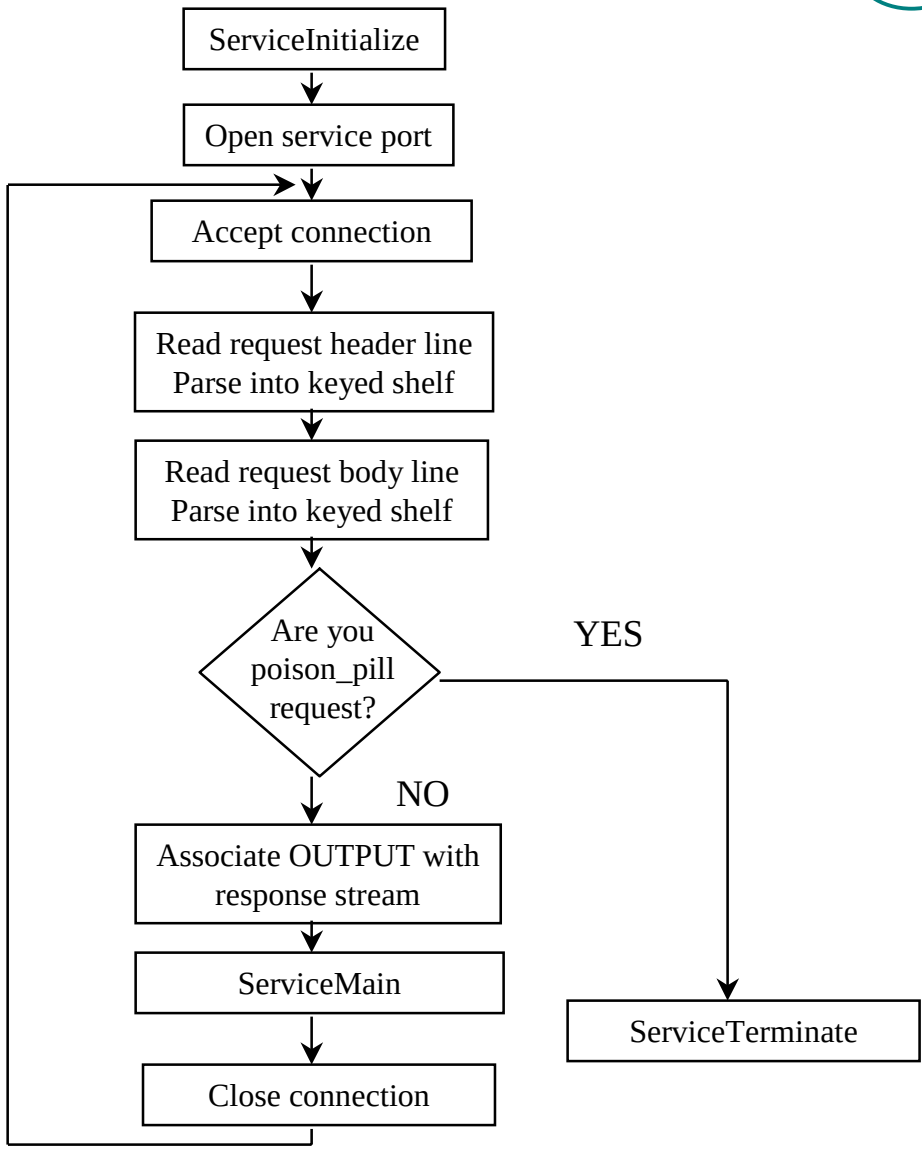
omasf

- The `omasf.xin` template uses the omtcp library to communicate with the web server relay using TCP/IP
- `omasf.xin` implements the server loop
- `omasf.xin` also closes the connection

omasf

- The framework parses encoded requests into two keyed shelves which are passed to the `ServiceMain` function:
  - `requestHeader`
  - `requestBody`
- `omasf.xin` also associates the current output with the response stream, meaning that anything output in `ServiceMain` is automatically transmitted back to the client for you

omasf.xin Flowchart



Using omasf.xin

- When you want to use the server framework, in `omasf.xom`…

```
declare no-default-io

global counter  ListenPort           initial {5700}
global stream   UserPoisonPillKey    initial {"POISON-PILL"}
global stream   UserPoisonPillValue  initial {"die"}
global stream   ServiceName          initial {”servicename"}

include 'omasf.xin'
```

Using omasf

```
define function ServiceInitialize
as
 ; add code needed at server start up

define function ServiceTerminate
as
 ; add code needed at server shut down

define function ServiceMain
   (  read-only stream requestHeader,
      read-only stream requestBody )
as
  ; business logic starts here…, output Content-type, and the
  ; response stream
```

Passing information: Web Server Relay

- General form:

  `http://host:port/`**`om-webserver-relay/omnimark-service-name?data`**

- Example:

  `http:/localhost/bin/`**`omcgir.exe/omasf?data=in.txt`**

- Link:

  `<A HREF="/bin/`**`omcgir.exe/disp?id=1&acces=all`**`"> Click Here </A>`

Getting the parameters

- The `omasf.xin` framework populates 2 keyed stream shelves: `requestHeader` and `requestBody`
- Key = name of the parameter
- Value = value of the parameter
- Access them and build your business logic accordingly!

Practical example

Programming based approach

Send HTML: programming approach

```
declare #process-output has binary-mode
macro CRLF is “%13#%10#” macro-end
process
   output “Content-type: text/html”
          || CRLF ||* 2
          || “<HTML><HEADER><TITLE>Hello</TITLE>”
         || “</HEADER>”
         || “<BODY>”
         || “<H1>Hello OmniMark Developers!</H1>”
         || “</BODY></HTML>”
```

Practical example

Template: basic substitution

Basic substitution

- Simplest form of template is value substitution
- Choose a tag style
  - Easy to pick out in the template
  - Unlikely to interfere with text of template
- I like "<<<tag>>>"
  - "<<<" stands out
  - In HTML it would be escaped as "< < <"

Example

- Template ‘web-page.tpl’

```
<HTML>
  <BODY>
    <H1>
      Happy birthday <<<NAME>>>
    </H1>
  </BODY>
</HTML>
```

Basic substitution

- Process by scanning the template:

```
global stream form-data variable
process
    CGIGetData into form-data
    submit file 'web-page.tpl'

find "<<<NAME>>>"
    output form-data{"name"}
```

Scanning process

- Invoke the pattern processor
  - `submit file ‘ filename ’`
- Write find rules
  - `find ‘ <<<NAME>>> ’`

Other scanning process

```
repeat scan file new-template
      match any++ => stuff
       lookahead ("<<<" | =|)
         output stuff
      match "<<<Word>>>"
         output form-data {"new_word"}
      match "<<<Action>>>"
         output cgi-data{"SCRIPT_NAME"}
             || "/login"
   again
```

Template: tag substitution

Centralized processing

Centralized processing

- If you use multiple templates, centralize processing

```
define function process-template
    value stream template-file
    using read-only stream substitutes
    as
    repeat scan file template-file
```

Centralized processing

```
match any++ => stuff lookahead ("<<<" | =|)
   output stuff
match "<<<" letter+ => placeholder ">>>"
   do when substitutes has key placeholder
      output substitutes{placeholder}
   else
      throw invalid-template
   done
match any ;this should never fire
   throw invalid-template
```

Centralized processing

Process application data, then process template

```
set new substitutes {"Word"}
 to form-data{"new_word"}

set new substitutes{"Action"}
 to cgi-data{"SCRIPT_NAME"} || "/login"

process-template unoriginal-new-word-template
 using substitutes
```

Template: tag substitution

Using referents

Alternative method

- Process template, then process application data
- Template tags are placeholders
- Referents are placeholders!
- Output a referent for each template tag
- Supply the value as you come to it

When do you use a referent in OmniMark?

- When you need to write something out, but you don't necessarily know what it is yet
  - Referents solve problems that would require multi-pass programs in other languages
  - Examples: cross references, table of contents, …

Referent mechanism

- Write referents:
  - Name the place where you are not sure what value will be displayed
  - Set the value when you know it
- … and let OmniMark do the rest!

Referents syntax

- Write out a referent instead of a string

```
output referent referent-name
output referent “ref-1”
```

- At some time during processing, bind the referent to a string value

```
set referent referent-name to StringValue
set referent “ref-1” to “Refer to Chapter 2”
```

- Two separate actions
  - You can set a referent and not output it, and you can output a referent and not set it

Quick example

```
global integer letter-count initial {0}

process
  output referent “final-count”
  submit “this is the text to be processed by the find rules”
  set referent “final-count” to “There are %d(letter-count)
   letters in the following text:%n”


find letter => let
  increment letter-count
  output let
```

Other things to know about referents

- Resolution occurs at end of referent scope (end of program by default)
- Last value stored in referent is the one displayed
- Referent names are string expressions and are case sensitive
  - Use dynamic names and names that make sense!

Using referents for template processing

```
repeat scan file template-file-name
     match ([\"<"]+ or
        "<" lookahead not "<<")+ => stuff
        output stuff
     match "<<<" letter+ => placeholder ">>>"
        output referent placeholder
     match any
        throw template-error
  again
```

Using referents for template processing

```
set referent "OrderTable"
 to "<P>No items ordered."

set referent "order cookie"
 to 'Set-Cookie: order=; '
 || 'expires=sat, 01-Jan-2000 12:12:12 GMT'
```

CCL

Content Control Language

Add intelligence to a template language

- Simple substitution does not always give the designer/content provider sufficient control
- But template languages like Cold Fusion and ASP may be too complex and require a programmer

Add intelligence to a template language

- Create a template language with just enough intelligence to do what the designer/content provider needs
- Leave the heavy lifting to programs written in a full programming language

Example

- OmniMark professional services did a project for a major retailer
- Some items are on sale
- For these items we need to show sale price as well as regular price
- Template includes special presentation features for sale items
- We need to suppress those elements if the item is not on sale

Add intelligence to a template language

- A good balance is to enable the template language to make basic decisions about what to include and exclude
- Should be able to include and exclude sections of the template as well as the data

Add intelligence to a template language

- Need to be able to do loops for tabular data
- OmniMark is an ideal language for writing template processing code
  - Scoping perfect for implementing these features

Content Control Language

- The rest of the presentation will cover CCL (Content Control Language)
- CCL is intended as a didactic device for teaching template processing techniques
- It may also be useful as a template for developing your own template languages
- You should design a template language to fit your particular business needs

CCL: language specification

- Key points
  - Provide control for displaying or hiding information
  - All content comes from external services
    - single value
    - records
  - String and Boolean variables for process control
  - XML style syntax, with attributes

CCL: control mechanism

- Basic control mechanism is a "try" block
- If any operation inside a try block fails, output of everything in try block is suppressed
- Can test values in a try block to force a failure

CCL: control mechanism

- Can use variables to make success of one try block depend on the failure of another, or vice versa
- ccl-action-failed state applies to each try block

CCL: the tags

- ccl-value
- ccl-output
- ccl-test
- ccl-assert
- ccl-record
- ccl-field
- ccl-try
- ccl-cgi
- ccl-form

CCL: ccl-value tag

- example 1: <ccl-value source="time" zone="gmt">
- example 2: <ccl-value source="time" zone="gmt" set="time">
- Receives a value from a value service
- Extra attributes are passed to the service
- If "set" attribute specified, value assigned to named variable
- ccl-action-failed is set if no value received

CCL: ccl-output tag

- example: <ccl-output variable="time">
- Outputs the value of the variable specified by the "variable" attribute

CCL: ccl-test tag

- example 1: <ccl-test variable="time" comparison="<" compare-to="12:00:00">
- example 2: <ccl-test variable="time" comparison="<" compare-to="12:00:00" set="before-noon">
- Compares the value of the string variable specified by the "variable" attribute with the string specified by the "compare-to" attribute using the comparison specified by the "comparison" attribute.

CCL: ccl-test comparisons

- - "=" equals
- - "<" less than
- - ">" greater than
- - "in" variable is in the compare-to value

CCL: ccl-test comparisons

- - "contains" variable contains the compare-to value
- - "begins" variable starts the compare-to value
- - "begins-with" variable starts with the compare-to value

CCL: ccl-assert tag

- example: <ccl-assert true="before-noon">
- example: <ccl-assert false="before-noon">
- Tests the Boolean variable specified by the "true" or "false" attribute. Sets ccl-action-failed if the assertion fails.
- ccl-assert is used within a try block to test a condition occurring earlier in the template.

CCL: ccl-record tag

- example: <ccl-record source="customer" id="%value(customer-id)"> ... </ccl-record>
- Requests one or more records from a service
- Field values are returned by ccl-field tags
- If the service returns more than one record, the block defined by the start and end ccl-record tags is repeated for each record.
- Record tags can be nested

CCL: ccl-field tag

- example 1: <ccl-field name="city">
- example 2: <ccl-field name="city" set="city-name">
- Retrieves the value of a field in the current record
- If the "set" attribute is not specified, outputs the value
- If the "set" attribute is specified, assigns the value to the string variable named by the "set" attribute

CCL: ccl-try tag

- example 1: <ccl-field name="city">
- example 2: <ccl-field name="city" set="city-name">
- Retrieves the value of a field in the current record. If the "set" attribute is not specified, outputs the value. If the "set" attribute is specified, assigns the value to the string variable named by the "set" attribute.

CCL: ccl-try tag

- example: "<ccl-try>...</ccl-try>"
- example: "<ccl-try set="succeeded">...</ccl-try>"
- Defines a try block
- You can specify a Boolean variable in the optional "set" parameter
- The variable is set to true if the try block succeeds and to false if it fails
- Try blocks can be nested

CCL: ccl-cgi tag

- example 1: <ccl-cgi name="QUERY_STRING">
- example 2: <ccl-cgi name="QUERY_STRING" set="query">
- Outputs the value of the CGI environment variable specified by the "name" attribute.
- If the cgi variable is not found, ccl-action-failed is set to true.

CCL: ccl-form tag

- example 1: <ccl-form name="name">
- example 2: <ccl-form name="name" set="name">
- Outputs the value of the form variable specified by the "name" attribute.
- If the form variable is not found, ccl-action-failed is set to true.

Configure web server to run CCL

- Configuring IIS to run OmniMark CGI programs
  - **C:\OmniMark\omnimark.exe -sb %s**
- Configuring IIS to run CCL programs
  - **C:\OmniMark\omnimark.exe -sb ccl.xom %s**

Implementation: Processing the ccl template

- Submit the template named on the command line:

```
process
    CGIGetEnv into cgi-data
    CGIGetQuery into form-data
    output "Content-type: text/html"
        || crlf
        || crlf
    submit file #args[1]
```

Implementation: Find tags and arguments

- Find ccl tags

```
find "<ccl-" letter+ => command
```

- interior of ccl tag is a nested context so…

```
repeat scan #current-input
```

Why not use XML parser?

- Templates don't have to be well formed XML
- It's just as easy to use find rules
- This method lets me illustrate streaming techniques
- The program needs to capture and re-scan markup between record tags

Select the command to execute

- Can't use do scan because it would shadow #current-input in the functions

```
do when command matches ul "test" =|
      ccl-test argument
   else when command matches ul "cgi" =|
      ccl-cgi argument
   else when command matches ul "form" =|
      ccl-form argument
```

The "cgi" command

```
; "cgi" command
define function ccl-cgi
 read-only stream argument
 as
   do when argument has key "set"
      new ccl-string{argument{"set"}}
       unless ccl-string has key argument{"set"}
      set ccl-string{argument{"set"}}
       to cgi-data{argument{"name"}}
   else
      output cgi-data{argument{"name"}}
   done
```

The "try" structure

- The try structure spans a section of the template
- It redirects output to a buffer by creating a new output scope and submitting #current-input
- Because try structures can be nested, we need a stack of buffers
- We also need a stack of ccl-action-failed variables

The "try" structure: push on the stack

- Push new conditional-output buffer and ccl-action-failed test onto try stack

```
new conditional-output
new ccl-action-failed
```

- Direct all output in the try block to conditional output buffer

```
open conditional-output as buffer
using output as conditional-output
    submit #current-input
```

The "try" structure: succeed or fail

- Catch end of try block scope

```
catch close-ccl-tag name
```

- Check for well-formed ccl markup

```
throw ccl-error unless name = "try"
```

- Output the conditional output if no errors occurred in the try block

```
close conditional-output
output conditional-output
 unless ccl-action-failed
```

The "try" structure: set the test variable

```
do when argument has key "set"
   do when ccl-switch has key argument{"set"}
      set ccl-switch{argument{"set"}}
       to ! ccl-action-failed
   else
       set new ccl-switch{argument{"set"}}
        to ! ccl-action-failed
   done
done
```

The "try" structure: pop it off the stack

- Simply remove the top item on the stack

```
remove conditional-output
remove ccl-action-failed
```

- Note how the stack mechanism works with the scoping mechanism
- Note how little coding is required to support nested try blocks

The "value" command

- Calls the ccl-value-service function for the data
- Creates new string variable, if "set" specified

```
do when argument has key "set"
    new ccl-string{argument{"set"}}
     unless ccl-string has key argument{"set"}
    set ccl-string{argument{"set"}}
     to ccl-value-service argument{"source"}
     parameters argument
```

The "value" command

- Otherwise, outputs the value

```
else
    output ccl-value-service
     argument{"source"}
     parameters argument
done
```

- Sets ccl-action-failed if there is an error

```
catch ccl-value-service-error
    set ccl-action-failed to true
```

The "ccl-value-service" function

- Looks up service in the service registry (loaded at program start)
- Uses "take" and "drop" to break up service address in form <machine name>:<port number>

How the services work

- Each service is provided by a daemon process
- One daemon may provide one or more services
- Services.txt file lists services:

  order-info=sonic:6789

How the services work

- Request in in XML format

  <service-request name="order-info">

  <parameters source="order-info" id="1001"/>

  </service-request>

- Programmers job: write services

"ccl-value-service" : sending the request

```
open request as tcpConnectionGetOutput connection
    protocol IOProtocolMultiPacket
   using output as request
    do
       output '<service-request name="'
            || service-selector
            || '">%n<parameters '
       repeat over argument
          output key of argument
              || '="' || argument  || '" '
       again
       output "/></service-request>"
    done
```

"ccl-value-service" : receiving the reply

- Stream connection to parser
- Stream processed XML to return-value

```
open return-value as buffer
  using output as return-value
   using group parse-value-service
   do xml-parse
    scan tcpConnectionGetSource
    connection protocol IOProtocolMultiPacket
     output "%c"
  done
  close return-value
  return return-value
```

The "ccl-record" command

- Must repeat the record block for each record returned
- Achieve repetition by grabbing markup of entire block and submitting once for each record

The "ccl-record" command

- Save ccl-current-record and ccl-record-values to support nesting
- Save is another way of implementing a stack for nested structures, but does not allow open streams -- so not suitable for the ccl-try construct

The "ccl-record" command

- Note that between the scope/stack mechanism used for try and the scope/stack mechanism used for record, try blocks and record blocks nest within each other to any depth
- Trys within a record will be instantiated once for each record and will be evaluated independently for each record

"ccl-record" command: grabbing the markup

- Grabbing the markup is an interesting problem
- ccl-record tags can be nested, so this won't work:
- any** => tag-markup "</ccl-record>"
- Need to match any number of nested records
- Calls for a recursive pattern matching function

"ccl-record" command: grabbing the markup

```
set record-markup
  to #current-input take
 ( (any**
  ( ccl-record-start-tag
    between-ccl-record-tags
    ccl-record-end-tag
  ) )?
  any** lookahead ccl-record-end-tag)
```

The " between-ccl-record-tags " function

- **define switch function between-ccl-record-tags as**

```
repeat scan #current-input
  match any** lookahead
   (ccl-record-start-tag |
    ccl-record-end-tag)
  match ccl-record-start-tag
    between-ccl-record-tags
    ccl-record-end-tag
  match value-end
    return false
again
return true
```

Recursive pattern matching functions

- Pattern matching functions return true or false to pattern
- Don't need to capture the data they match, the pattern itself does that
- Scan #current-input
- Call themselves when they spot start of a nested structure

CCL Demo

Questions?