Jan Steinman Barbara Yates

SmallDoc Web Serving

In our June and September 1995 columns, we introduced a hyper–literate programming system we call "SmallDoc." In the last two issues, we sketched out how to turn SmallDoc into HTML, and how to build a generic TCP/IP server framework. This issue ties it all together so you can begin serving your Smalltalk project documentation to anyone with a Web browser.

The generic TCP/IP server described earlier needs only one or two blocks of Smalltalk code in order to implement a complete server. A message is sent to the class that associates a "service" block with a port number, and a second, optional (but strongly encouraged!) message is sent to the class to associate an "exception" block with the same port number.

For example, a simple hypertext service can be implemented by adding the following method to the *TcpServer* class that we presented last month:

```
TcpServer class:
        initializeForHttpd
        "Set up a service and exception handler suitable for
servicing World Wide Web requests."

        selfdefaultHandlerFor: 80 is: [:exception :stream|
                stream httpChattyHandle: exception];
            defaultServiceFor: 80 is:[:stream |
                stream htmlForSmallDocRequest]

        httpd
        "Answer the default hypertext transport protocol server."

        ^self onPort: 80
```

Now, to start up a Hypertext Transfer Protocol server all you need to do is evaluate "TcpServer httpd." Of course, if you do that right now it will crash, because we haven't really written the handler or service blocks yet.

If you are on a UNIX machine, remember that port 80 is privileged — you will have to run Smalltalk as root to run this service. If that is not possible, choose some other port above 1024, such as 8080.

When an error is encountered in the server, it should alert the client so that things can be fixed. Our httpChattyHandle:for: method assumes the person using the Web browser might know something about Smalltalk, and so it sends contextual information back to the client. For non-developers, you might want to simplify this by reporting only that an error occurred.

```
PositionableStream
        httpChattyHandle: exception
        "Upon trouble with the request, attempt to send
back a contextual information from <exception> in HTML
format on <stream>."

        | ctx |
        self
            cr; nextPutAll: 'HTTP/1.0 500'; cr;
            nextPutAll: 'Server: '; nextPutAll: (TcpServer
signatureIn: TcpServer controller); cr; cr;
            nextPutAll:'<HTML><HEAD><TITLE>Unhandled
exception!</TITLE></HEAD><BODY><H1>';
            nextPutAll: exception errorString;
            nextPutAll: '</H1><P>Your request had a
problem. Please copy the following stack and mail it to
the<A HREF="mailto:'
            nextPutAll: (EmUser called:'Supervisor')
networkName;
            nextPutAll: '">ENVY Library
Supervisor</A>.</P>'.
            ctx := exception thisContext.
            5 timesRepeat:
                [ctx == nil ifFalse:
                    [self print: ctx;cr]].
            self nextPutAll:'</BODY></HTML>'; cr; flush
```

Also, this example relies on ENVY repository information to report the server version information, and to obtain the email address of the repository supervisor. You should use suitable substitutes if you use a different code management system.

SERVICING REQUESTS

Now that we can handle failed requests, we should think

Jan Steinman and Barbara Yates are co–founders of Bytesmiths, a technical services company that has been helping companies adopt Smalltalk since 1987. Between them, they have over 22 years Smalltalk experience. They can be reached at Barbara@Bytesmiths.com orJan@Bytesmiths.com, or at their website at http://www.bytesmiths.com.

about servicing *real* requests! When the *TcpServer* sends htmlForSmallDocRequest to the socket stream, the stream may contain "GET" a space, and the URL the user entered or clicked.

There is much more information in the typical HTTP request, and this method can easily become complex. If you want to process more of the request information, be sure to factor this method into smaller methods that handle particular request information.

In particular, this method only attempts to deal with "GET" requests, which is the normal way a Web browser passively requests a Web page. This method does not handle "POST" requests, which is how a Web browser passes information entered by the remote user.

```
PositionableStream
    htmlForSmallDocRequest
    "Assume I am a bi-directional stream on a socket
that is connected to a web browser. Process an incoming
SmallDoc GET request."

    | line path |
    [(line := self nextLine) size > 0]whileTrue:
        [('GET ' occursIn: line at: 1) ifTrue:
            [path := line copyFrom: 5 to: line size]].

    self nextPutAll: 'HTTP/1.0 200';
        nextPutAll: 'Server: ';nextPutAll: (TcpServer
signatureIn: TcpServer controller); cr; cr;
    (path size = 0 or: [path = '/'])
        ifTrue: [self httpHomePage]
        ifFalse: [self htmlSmallDocGet: path]
```

This method looks for a line in the socket stream that begins with GET, and saves the rest of the line as the URL to fetch. If the URL is empty or if it is a single slash, then some form of home page information should be sent back to the Web browser, followed by closing the stream, which lets the Web browser know the request is complete.

```
PositionableStream
    httpHomePage
    "An empty GET request is received,so give a hearty
welcome."

    self
        nextPutAll: 'This is an exercise for the reader.
Put some literal HTML here (or some Text asHtml!)that
explains how to navigate through your Smalltalk
documentation repository.';
        close
```

If the URL is *not* empty, there's more work to do. We follow ENVY's existing structure for navigation; if you are using some other source code management system, you will have to implement a navigation strategy for your repository.

We expect the first component of the URL to be a *naming root* that serves as a dispatcher for the remainder of the URL.

```
PositionableStream
    htmlSmallDocGet: pathString
    "Place on myself valid top-level HTML for the given
<pathString>, which must begin with a slash ($/), and
therefore must have a size greater than zero, and must
consist of URLized path from some naming root, separated
by slashes.
Valid roots are:
        1) Smalltalk,
        2) EmUser,
        3) EmConfigurationMap,
        4) Application, or
        5) SubApplication.
    The second component of the path is always one of
the names that a root knows about. What follows is
dependent on processing by the root, which is sent the
rest of the path to play with.
    A new root must either be handled by this method,
or it must be a global, and it must supply the methods
#htmlAsRootOn:, and #htmlForPath:on:.
    This does minimal error checking —it assumes a
handler will catch exceptions."

    | path root |
    "Parse the path, keeping the result."
    path := pathRequest splitOn: $/.
    root := Smalltalk at: path first asSymbol.
    1 = path size
        ifTrue:
            [self nextPutAll: 'Pragma: no-cache'; cr; cr.
            root htmlAsRootOn: self]
        ifFalse:
            [root htmlForPath: path on: self]
```

Now we have a "naming root" that can be used for navigation, and all that is left to do is implement htmlAsRootOn: and htmlForPath:on: so that any global can serve Web information. For example, a simple inspector can be implemented by making "Smalltalk" a naming root by implementing htmlAsRootOn:.

```
SystemDictionary
    htmlAsRootOn: stream
    "Place on the given <stream> HTML links for my
distinguished instances."

    stream htmlTitleAndH1: 'Smalltalk Globals'.
    ^(self keys asSortedCollection
        inject: stream
        into: [:stream :globalName | |global |
            global := Smalltalk at: global Name.
            stream
                nextPutAll: '<A HREF="/Smalltalk/';
nextPutAll: globalName; nextPutAll: '">';
                nextPutAll: globalName; nextPutAll: '</A> '.
            global class isMeta
                ifTrue:
                    [stream nextPutAll: '(a class'.
```

```
                    (global class instSize > Object class
instSize or: [global classPool size > 0])ifTrue:
                        [stream nextPutAll: ' with state'].
                    stream nextPutAll: ')<BR>']
                ifFalse: [stream nextPutAll: '(an instance
of '; print: global class; nextPutAll:')<BR>'].
            stream]) htmlCloseBody
```

Now if htmlForPath:on: is implemented in *Object*, you can inspect arbitrary objects from a Web browser. This method uses a number of stream utility methods that make the task easier, by providing pre-assembled snippets of commonly used HTML.

```
PositionableStream
    htmlBody: anObject
        "Place on myself the proper HTML to make
<anObject> appear as body text. This must be preceded by
a 'title' statement. Answer myself."

        self nextPutAll: '<BODY>'; htmlFor:an Object;
htmlCloseBody

    htmlCloseBody
        "Place on myself the proper HTML to close off a
'body' statement. Answer myself."

        self nextPutAll: '</BODY></HTML>'

    htmlTitle: string
        "Place on myself the proper HTML to make <string>
a title. This must be followed by a 'body' statement.
Answer myself."

        self nextPutAll:'<HTML><HEAD><TITLE>';
nextPutAll: string; nextPutAll:'</TITLE></HEAD>'

    htmlTitleAndH1: string
        "Place on myself the proper HTML to make <string>
a title, followed by a 'body' statement and <string> as a
top-level heading. Answer myself."

        self
            htmlTitle: string;
            nextPutAll: '<BODY><H1>';
            nextPutAll: string;
            nextPutAll: '</H1>'
```

It would be easy to slip into gratuitous serving of all sorts of objects over the Web at this point, but we'd neglect our primary purpose: to serve Smalltalk project documentation over the Web. To do this, we need to allow *SubApplication* to function as a naming root. (Since *Application* is a subclass of *SubApplication*, this also allows *Application* to serve as a naming root.)

```
SubApplication class
    htmlAsRootOn: stream
        "Place on the given <stream> HTML links for all
subapps or apps."

        stream htmlTitleAndH1: self name, 's'.
        ^(self allNames asSortedCollection
            inject: stream
            into: [:stream :appName |
                stream
                    nextPutAll: '<A HREF="/'; print: self;
nextPut: $/;
                    nextPutAll: appName; nextPutAll: '">';
                    nextPutAll: appName; nextPutAll:
'</A><BR>'.
                stream]) htmlCloseBody
```

Now when a URL with a naming root, such as `<http://yourhost/Application>`, is entered into a Web browser, a page is returned that lists all *Applications* in the repository, together with links that have the next path component filled in. When one of the listed *Applications* is clicked in the Web browser, the following method is sent in the SmallDoc server:

```
SubApplication class
    htmlForPath: path on: stream
        "Place HTML for my components described by
<path> on the <stream>."

        | component |
        2 = path size ifTrue:
            ["This is dynamic information — do not cache it
in the client."
            stream nextPutAll: 'Pragma:no-cache'; cr; cr.
            self htmlEditionsForName: path last on: stream]
ifFalse:
        [(path last conform: [:ch | ch isDigit]) ifTrue:
            [path at: path size put: (Integer readFrom: path
last readStream)].
        component := (SmallTalk classAt: path first)
hrefToLibraryComponentFor: path.
        component isVersion ifFalse:[stream nextPutAll:
'Pragma: no-cache'; cr].
        stream cr.
        3 = path size ifTrue:
            [stream htmlBody:
(component commentOrTemplateIn: component)] ifFalse:
        [4 = path size ifTrue:
            [stream htmlBody: (component
commentOrTemplateIn: component application)] ifFalse:
        [5 = path size ifTrue:
            [stream htmlBody: component comment] ifFalse:
        "path size > 5 ifTrue:"
            [stream error: 'bad URL']]]]
```

This method is a case statement. Only one of the "path size" cases will be evaluated in any given invocation. Also note that if the last component of the path consists of digits, it is converted to an *Integer*.

This sends two methods that we're going to have to

continued on page 36

continued from page 24

leave you to implement yourself, due to space constraints. The *SubApplication* method htmlEditionsForName:on: needs to obtain all the editions for the *SubApplication* named by the second part of the path, and render them into the proper HTML so they will appear as links in the Web browser. For example, if the Web browser user typed or clicked `<http://yourhost/Application/Kernel>`, the server should place links for each edition of *Kernel* on the socket stream.

The more interesting method to complete is hrefToLibraryComponentFor:, which takes a collection of component parts and fetches the proper component out of the repository. For example, the URL `<http://yourhost/Kernel/Object/at:put:/3016057369>` should cause the comment for the *Object* method at:put: with the edition time stamp of July 29, 19961:42:49 am to be placed on the socket stream.

As hinted by the code, our treatment of the URL depends on its number of path parts. For an individual repository component, such as an app, subapp, class, class extension, or method, the last part of the URL path is always a second count from the component's time stamp, thus allowing you to browse version history from a Web browser. These integers are meant to be "opaque references" — the user should never have to type them in; rather, they should be part of anchors that were generated from lists of editions.

Following the example of *SubApplication*, you can now easily add htmlAsRootOn: and htmlForPath:on: to *EmUser* and *EmConfigurationMap*, as well as any other global that you want to use as a "naming root" for serving arbitrary information from Smalltalk over the Web.

SATISFYING WITHOUT COMPROMISING

This completes our series on putting your Smalltalk project documentation on the Web. This series enables our principles of hyper-literate programming by ensuring that:

1) the documentation for a thing is on the same conceptual level as that thing;
2) the documentation for a thing constantly and accurately describes that thing;
3) the documentation for a thing is accessible by creators, their peers, re-users, reviewers, end-user documenters, and the merely curious; and
4) the documentation for a thing is measurable, quantitatively and especially qualitatively.

In addition, we hope we've shown you a few useful things about developing frameworks and automatically generating HTML.

Maintaining your documentation in your Smalltalk repository while exporting it "live" to Web browsers will satisfy the needs of external parties without compromising the efficiency of your development team.

Next month, we'll explore a topic close to our hearts — the use and abuse of Smalltalk mentors. **S**