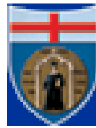
Università degli Studi di Genova

Facoltà di Scienze Matematiche Fisiche e Naturali

Laurea Triennale in Informatica

Prova Finale

WebFormKit: un nuovo modo di sviluppare applicazioni Web



CANDIDATO

Antonio Cuni

RELATORE

Prof. Marina Ribaudo

Anno Accademico 2003/2004

Alla mia famiglia

A mio zio

Contents

1 Introduction **1**

1.1 What is a Web Application? 1

1.2 A big issue: HTTP is stateless 1

1.3 Web Application in the old style 2

1.4 Another problem: business logic is mixed with presentation logic 2

2 *WebFormKit* for dummies **4**

2.1 *WebFormKit* key concepts 4

2.1.1 Why Python? 5

2.2 *Hello World* with *WebFormKit* 5

2.3 *WebFormKit*'s DOM 8

2.4 The extension mechanism 9

2.5 Attributes and namespaces 10

2.6 Events and Event handlers 11

3 WebFormKitPage and documents stack **13**

3.1 The WebFormKitPage class 13

3.2 *Documents* and *documents stacks* 13

3.3 A real life problem 14

3.4 A simple example 15

4 The *actions* **19**

4.1 Introduction 19

4.2 *Action providers* 19

4.3 *Actions* with parameters 20
4.4 *Actions* in action 21

5 A complete example: the *multiplication table* **23**
5.1 Description of the project 23
5.2 The web:textbox and web:table custom controls 23
5.3 Validating forms 24
5.4 Code analysis 24

Chapter 1

Introduction

1.1 What is a Web Application?

There are two kinds of Web sites, those that behave like magazines, places where you read stuff; and sites that behave like software, places where you go to do stuff.

Examples of magazines include the NY Times and Newsweek sites, Weblogs, the Encyclopedia Britannica and the Van Gogh Museum. These are published sites, created by writers, designers and graphics people.

Examples of applications include Yahoo's calendar and Hotmail's email tools. These are software tools and utilities that run on a server and interface through a Web browser like MS Internet Explorer and Netscape navigator. The people who create Web applications are programmers and user interface designers, experts in creating software.

There has been a lot of new activity in Web applications for a simple reason: it takes time to figure a new environment out. The Web browser is one of the toughest user interface environments we have ever had to develop for. There are a lot of Web application toolkits that simplify programmers' work: *WebFormKit* is one of these and tries to be better that its predecessors.

1.2 A big issue: HTTP is stateless

One of the reasons for the huge popularity of the HTTP protocol is that it is stateless. Other protocols typically require to establish at least one TCP/IP connection and then

communicate with the server over it. With HTTP, every request is independent; no matter how many requests you make the server always looks at each request on its own.

This worked fine while requirements were simple (such as static web sites) but it became a nightmare for web applications. To solve the problem, application authors had to maintain the state themselves: usually this is automatically done by Web applications toolkits, as explained in the next section.

1.3 Web Application in the old style

A possible solution for the lack of state is to maintain state at "session" granularity. This means that web applications can remember some information between subsequent requests, such as the login name of the current user.

PHP [4] and ASP [1] provide good examples: they both allow programmers to mark a variable as a *session variable*, meaning that its value is automatically remembered between requests.

Unfortunately for complex applications often this is not enough: programmers still have to manually track information about *page*'s state, such as input field values and so on. As we will see in section 2.1, *WebFormKit* fills this lack.

1.4 Another problem: business logic is mixed with presentation logic

When we talk about business logic, we mean the following kind of logic:

- how we model real world business objects such as accounts, loans, travel itineraries etc;
- how these objects are stored;
- how these objects interact with each other - e.g. a bank account must have an owner and a bank holders portfolio is the sum of his accounts;
- who can access and update these objects.

By contrast when we speak about Presentation Logic we mean how we display these objects to a user. E.g., do we have a drop down list or a pop-up screen? Do we display accounts in a list format and have the user pick which one to edit? etc.

There are several reasons why separating business logic from presentation logic is desired.

The kind of programming used for each part and sometimes the language used are different. Separating the two allows one to use the best tools for that particular goal. For example consider the caso of programming a user interface screen - the kinds of things we are concerned with are paging - e.g. how many records to show on a page, widths of forms, colors readability etc. There is also the need to see the screen during the development and to position the items on it. On the business side we are not concerned about rendering of information but we are interested in getting the data from "somewhere" and controlling which data are loaded, and how that data are updated. In case of change of the programming languages, it is much easier to reuse old code when it is separated from the presentation that would be mostly HTML or XML code which can be easily refitted within a different model.

Moreover, it is easier to read business logic that is not intermingled with presentation logic. Similarly it is easier to look at the presentation of a screen or read HTML code without having to go through the database and security logic.

Unfortunately most web application toolkits, such as PHP and ASP, do not help programmers in this task: source code often contains HTML tags as well as programming language constructs, i.e. presentation logic mixed to business logic.

Chapter 2

WebFormKit for dummies

2.1 *WebFormKit* key concepts

The primary goal of *WebFormKit* is solving the problems analyzed in the previous chapter in order to allow web developers to easily and quickly write their applications.

WebFormKit's documents are composed by two files: an XML file that contains the structured content which will be used to produce an output file and a Python file that contains the business logic underlying the document itself.

When a document is opened for the first time *WebFormKit* parses the XML file and builds a corresponding *DOM* (Document Object Model): Python code can control the output by interacting with the DOM, e.g. by hiding a particular portion of the document or by changing the text to display.

The basic idea of using an XML file to separate the business logic from the presentation logic is not new: other projects, such as *ASP.NET* [2] and *Nevow* [3], have already developed it. In particular, *WebFormKit* takes a lot of ideas from *ASP.NET*, trying to improve the good ones and to corrects the bad ones.

In our point of view one of the major problem with *ASP.NET* is a poor maintenance of the state at "page" granularity: *WebFormKit* improves this feature by creating *persistent objects* whose life's cycle lasts as long as the page's one; this means that during successive requests to the same page, developers interact with the very same object, allowing them to store informations between subsequent requests.

WebFormKit's applications are event driven: this paradigm is an abstraction over the classical request-response cycle that used to characterize most web applications. Considering the object persistence just discussed, this mean that *WebFormKit* application development is in a way similar to classical GUI application development.

2.1.1 Why Python?

WebFormKit is entirely written in Python [5]: we choose this programming language after a deep comparison with other languages such *PHP*, *Java* and *C#*. Python resulted to be the simplest and most powerful tool for our purposes.

In addition, Python is completely portable between *Windows* and *Unix like* platforms, meaning that we can develop an application under, say, *Linux + Apache* and deploy it under, say, *Windows + IIS* without changing a single line of code!

WebFormKit is written as a *Webware* [7] plug-in. *Webware* is a powerful application server written in Python. This choice allowed us to exploit a consistent amount of ready-to-work code without the need to rewrite it from scratch.

If we take a deeper look to *WebFormKit*, we can see that it is split into two packages, the *WebFormLib* and the *WebFormKit*, described below:

- *WebFormLib* is the core of the library and provides most of the features of the entire toolkit, such as the DOM construction and the event management;
- *WebFormKit* is the bridge between *WebFormLib* and *Webware* and gives its name to the entire project.

We decided to split the project into two parts for improving portability between different application servers: today we use *Webware* but in the future we might want to migrate to another tool. By keeping the non *Webware* specific features in the *WebFormLib* package the porting is simplified.

2.2 *Hello World* with *WebFormKit*

The unique feature shared by all programming languages and tools is the famous *Hello World*: all beginners start with this example. We don't want to make an exception, so let's

go greeting our planet.

As we discussed in section 2.1, a *WebFormKit*'s document is composed by at least two files: the XML one and the Python one.

The *Hello.xml* and *Hello.py* files are shown in figures 2.1 and 2.2, respectively.

```
<?xml version="1.0" ?>
<web:document xmlns:web="http://www.disi.unige.it/WebFormLib.Web"
              web:code="Hello.HelloDocument">

  <h2 web:id="title">My first <i>WebFormKit</i> page</h2>

  <h1><web:label web:id="label"/></h1>

  <web:form>
    <web:button web:id="show" web:enabled="false">Show title</web:button>
    <web:button web:id="hide">Hide title</web:button>
    <web:button web:id="random">Change greeting</web:button>
  </web:form>

  <p>This page has been rendered <web:label web:id="renderCount"/> times</p>

</web:document>
```

Figure 2.1: The *Hello.xml* file

We do not explain all the details of these files for the moment, but we focus on *WebFormKit*'s key concepts.

First of all, notice the HelloDocument class: it contains the *handlers* used to react to various *events*.

The first handler, on_open, is called the first time the page is opened and produces some initialization. The count member stores the number of times the page has been rendered. Notice that we can reference all objects with an associated web:id attribute, such as label, title or show, by simply using their corresponding identifier.

Next handler is named on_render and it is called each time the page needs to be rendered to be sent to the browser. In such a handler we update the count member and the related renderCount label.

The last three handlers are invoked when the user clicks on the corresponding button: the random button randomly select a greeting in a different language from the current one while show and hide buttons shows the semantics of the web.visible attribute. Setting it to False implies that the element will not be sent to the browser and therefore the user will

```python
import random

from WebFormLib import EventHandler
from WebFormLib import StopEventProcessing
from WebFormKit import WebFormKitPage

# declare the Webware Servlet
class Hello(WebFormKitPage):
    defaultDocument = 'Hello.xml'

# declare the event handlers for the 'Hello.xml' document
class HelloDocument(EventHandler):
    greetings = [
        'Hello world!',
        'Ciao mondo!',
        'Salut le monde!',
        'Ola mundo!',
        '!Hola mundo!',
        'Hallo Welt!'
        ]

    def on_open(self):
        self.count = 0
        self.label.web.text = self.greetings[0]

    def on_render(self):
        self.count += 1
        self.renderCount.web.text = self.count

    def on_random_click(self):
        new = old = self.label.web.text
        while new == old:
            new = random.choice(self.greetings)
        self.label.web.text = new

    def on_show_click(self):
        self.title.web.visible = True
        self.show.web.enabled = False
        self.hide.web.enabled = True

    def on_hide_click(self):
        self.title.web.visible = False
        self.show.web.enabled = True
        self.hide.web.enabled = False
```

Figure 2.2: The *Hello.py* file

Figure 2.3: The page before the user clicks on the Hide button

Figure 2.4: The page after the user clicked on the Hide button

not be able to see it. Note that the web.visible attribute works correctly even with objects associated with standard *HTML* tags such as *h1*.

Figures 2.3 and 2.4 show the page in a web browser before and after the user clicked on the Hide title button, respectively.

2.3 *WebFormKit*'s DOM

In section 2.1 we said that *WebFormKit* uses XML files to construct a DOM: but, what is a DOM? We can think of it as a tree representing our page: the nodes of the tree are either XML tags or blocks of text.

When a browser requests a page, the DOM is *rendered* and sent back to the user. The rendering is done according to the specific type of node involved in the transformation.

We can distinguish among three types of nodes:

TextNode These nodes are created whenever the parser encounters a sequence of *parsed characters data* (i.e., the "text between tags"). They render as they are in the XML source file.

LiteralTag These nodes are created whenever the parser encounters a generic XML tag without an associated namespace: they renders as they are in the XML source file.

Since (X)HTML tags have no namespace they are all represented as LiteralTags: this means that the output file will include them as they appear in the XML source file.

Custom controls They are one key concept of *WebFormKit*: when the parser encounters a specific set of tags it searches for the associated *Python class* and instantiates it (see section 2.4 for more details). In the *Hello world* example the web:document, web:label, web:form and web:button tags are all *custom controls*.

Since *Custom controls* are instances of arbitrary Python classes, they can render themselves as they want: for example, their output can depend by the value of a specific attribute that the user can modify, such as the web.text attribute of the label object.

2.4 The extension mechanism

WebFormKit is designed to be as much extensible and customizable as possible: the preferred way to add features to the toolkit is by writing or using *extension modules*. These are normal Python modules respecting some conventions that we do not discuss here.

Developers can tell *WebFormKit* to load extension modules by inserting a namespace declaration in the XML file. The Python module to be loaded is determined by inspecting the *namespace URI*: in particular, *WebFormKit* tries to import a module named as specified in the *path* component of the URI, as described in [6] (i.e., the "name after the last slash").

When the parser encounters a tag with an associated XML namespace, it searches the appropriate extension module for a *Python class* suitable for that tag and instantiates it.

After the class has been instantiated, the parser can take the more appropriate action according to its type:

- if the class is declared as a *custom control*, the parser inserts the instance into the appropriate place of the DOM, just as if it were a LiteralTag (see section 2.3);

- if the class is declared as a *directive*, the parser *executes* it. By being executed, directives can modify the attributes and the behaviour of their enclosing parent: directives are a convenient way to declaratively specify a particular behaviour without the need to write Python code. For some examples of directives, see chapter 4.

2.5 Attributes and namespaces

If you paid attention to the source code of the *Hello world* example, you should have noticed that custom controls' attributes are prefixed by the web prefix either in the XML and the Python file. As for custom controls, attributes are grouped into XML namespaces as well.

When writing Python code, we can access objects' attributes via their attrs member: attrs groups together the various XML namespaces, that can be accessed either via the *subscription* syntax (e.g. attrs['web']) or via the *dot* syntax (e.g attrs.web).

Each attr's member contains the attributes of a particular XML namespace: single attributes can be accessed with one of the two syntaxes just introduced. Attributes that do not belong to any particular namespace are stored in a pseudo-namespace denoted as "_".

Now suppose we have an object named obj: next example shows how to use the different syntaxes to access its attributes.

```
# the following lines all access to the same attribute
obj.attrs.web.text = 'hello'
obj.attrs.web['text'] = 'hello'
obj.attrs['web'].text = 'hello'
obj.attrs['web']['text'] = 'hello'

# attributes without namespace can be accessed via the '_'
# pseudo-namespace
obj.attrs._.bgcolor = 'red'
obj.attrs['_']['bgcolor'] = 'red'

# there are a couple of special cases in which we can't use the dot
# notation; e.g., 'class' is a Python keyword, so we can't use it to
# specify an attribute
obj.attrs._.class = 'myCssClass'    # SyntaxError !!!
obj.attrs._['class'] = 'myCssClass' # OK
```

Typing attrs again and again is annoying and, we know, programmers do not like boring tools: *WebFormKit* saves developer's feelings by providing a handy shortcut that works in most case. The namespace's name does not conflict with other object's members we can omit the attrs part. It is worth noting that this shortcut works with the *dot* notation only:

```
obj.web.text = 'hello'
obj.web['text'] = 'hello'
obj['web'].text = 'hello' # WRONG!

# let's break the toy: create a 'web' member on 'obj'
obj.web = 'This is a standard member of obj'

# prints 'This is a standard member of obj'
print obj.web

# AttributeError: obj.web is a string, not a namespace
print obj.web.text

# in such cases, we can use the old 'attrs' notation
print obj.attrs.web.text # OK
```

2.6 Events and Event handlers

In section 2.1 we said that *WebFormKit* is an *event driven* toolkit: events represent a particular interesting happening that developers might want to be notified of. Programmers declare to be interested in particular events by writing their corresponding *handlers*.

To handle events is necessary to perform the next two tasks:

- write the handlers;
- tell *WebFormKit* where the handlers are stored.

Event handlers are easy to write: they are stored in Python classes derived by the EventHandler class. Each handler is just a method with a special name: the convention for handling an event myEvent raised by the object myObject is to write a method named on_myObject_myEvent. If we are interested in events raised by the main document (which has no name) we can simply specify on_myEvent.

After writing the handlers, we must associate them to the appropriate objects: this is done by using the web:code attribute of the web:document custom control.

There are two possibilities:

- if web:code contains a dot (.) it will be considered a fully qualified class name, where the word after the last dot represents the class name and those preceding the last dot represent the module name (optionally including package and subpackages specification);

- else, if web:code does not contains dots, *WebFormKit* searches for a module named as the attribute value and then searches for a class with the same name.

Now, let's look again at the *Hello World* example. The HelloDocument class shown in figure 2.2 collects the events handlers: it has methods handling the open and render events of the main document and the click event raised by the three buttons (see figures 2.3 and 2.4).

The Hello.xml file, shown in figure 2.1, contains the event handler declaration: the web:code attribute is set to Hello.HelloDocument, meaning that *WebFormKit* should search for the HelloDocument class in the Hello module.

Chapter 3

WebFormKitPage and documents stack

3.1 The WebFormKitPage class

If we look again to the Hello.py file shown in figure 2.2 we can notice that it contains two classes: since in previous sections we explained how the HelloDocument class works, now it's time to understand what the Hello class does.

Hello is a *subclass* of WebFormKitPage class, which in turn is a subclass of the standard *Webware*'s Page class: this means that we can modify its behaviour by simply overriding methods such writeBodyParts or writeContent, just as we always did with *Webware*.

Since WebFormKitPage extends its base class, developers can access a new set of higher level features provided by this class: the most important example is surely the concept of *documents stacks*, which is described in the next section.

3.2 *Documents* and *documents stacks*

We can think of a *documents* as the join between a *WebFormKit*'s XML file and the corresponding EventHandler subclass: in the *Hello world* example there is only one document, composed by the Hello.xml file and the HelloDocument class.

Generally speaking *WebFormKit*'s pages will consist of only one document, but if necessary they can *pop-up* new ones: newly opened documents will be put on top of the page's *document stack*. At the time of rendering the page will consider only the document on the top of the stack: the user will not be able see the other documents on the stack because they

are “hidden” by the one on the top.

When the user decides *close* the current document it is removed from the top of the stack and the one immediately below is rendered. Since *WebFormKit*'s documents are *persistent* (see section 2.1) that document will be in the same state it was when the user decided to open the new one.

The document shown the first time the page is entered is taken from the defaultDocument class member.

The documents stack can be manipulated using methods provided by the WebFormKit-Page class:

popupDocument Open a new document and push it on the top of the stack.

openDocument Clear the stack and push a newly opened document.

closeDocument Close and pop the document from the top of the stack.

3.3 A real life problem

Let us consider a real life example to better understand the power of *document stacks* are powerful: suppose we have a page with a table showing a set of records taken from a database; since there are many records the user can select the column to order by by simply clicking on the corresponding table header.

Moreover, the user can click on an Edit link that opens a page developed to modify record details: when the user close the Edit page it returns to the main page containing the table. Once back it would be great if the table is still ordered as the user previously selected, but how to do this?

Old style web applications have no chance: they must manually store the column selected in some persistent place like a cookie or a session variable, then the developer must remember to load those values when the page is reopened after the record has been edited. Imagine what happens if the page state is fairly complex or if the same page can be reached by more than one path: it can be a nightmare (and often it was!).

Using *WebFormKit*'s documents stack the solution to the problem is straightforward: it's enough to put a new Edit document on the stack when the user clicks on the Edit link and to

close that document when the user finishes editing. At this point the page will automatically render the previous document exactly as it was before.

3.4 A simple example

Section 3.2 explained what a *documents stack* is: let us now analyze an example showing how to use it.

Our example is composed by two documents split into three files: PopupDoc.xml and CloseDoc.xml contain the structure of the two documents, while Stack.py contains the *Webware*'s page and the event handlers for the documents. Figures 3.1, 3.2 and 3.3 shows these files.

```
import random

from WebFormLib import EventHandler
from WebFormKit import WebFormKitPage

# declare the Webware Servlet
class Stack(WebFormKitPage):
    defaultDocument = 'PopupDoc.xml'

class PopupDoc(EventHandler):
    def on_open(self):
        self.count = 0
        self.random.web.text = random.randint(1, 100)

    def on_render(self):
        self.count += 1
        self.renderCount.web.text = self.count

    def on_popup_click(self):
        self.page.popupDocument('CloseDoc.xml')

class CloseDoc(EventHandler):
    def on_close_click(self):
        self.page.closeDocument()
```

Figure 3.1: The *Stack.py* file

Let us examine the PopupDoc document first: there is a label named random that stores a random number generated in the on_open event and a renderCount label that displays the number of times the page has been rendered, just as in the *Hello world* example. Moreover, there is a popup button whose handler calls page's popupDocument method: when the user

```
<?xml version="1.0" ?>
<web:document xmlns:web="http://www.disi.unige.it/WebFormLib.Web"
              web:code="Stack.PopupDoc">

  <h2>Documents stack demo</h2>

  <p>This page demonstrates the use of documents stacks.</p>

  <p>Press the button below to popup a new document.</p>

  <p>Random number generated: <web:label web:id="random"/></p>

  <web:form>
    <web:button web:id="popup">Popup document</web:button>
  </web:form>

  <p>This page has been rendered <web:label web:id="renderCount"/> times</p>

</web:document>
```

Figure 3.2: The *PopupDoc.xml* file

```
<?xml version="1.0" ?>
<web:document xmlns:web="http://www.disi.unige.it/WebFormLib.Web"
              web:code="Stack.CloseDoc">

  <h2>Documents stack demo</h2>

  <p>This document is popped up by the <b>PopupPage</b> document.</p>

  <p>
    Press the button below to close this document and return to the
    previous.
  </p>

  <web:form>
    <web:button web:id="close">Close document</web:button>
  </web:form>

</web:document>
```

Figure 3.3: The *CloseDoc.xml* file

clicks on this button the CloseDoc document is pushed on the top of the stack.

The CloseDoc file is even simpler: it just contains a button whose handler calls page's closeDocument method.

Figures 3.4, 3.5 and 3.6 show the result of a simple test. In figure 3.4 we can see that the random number generated is *45* and this is the first time the document is rendered; as we can expect in figure 3.6 the random number hasn't been changed and the bottom line tells us this is the *second* time the document is rendered: this means that the document effectively remembered its state.

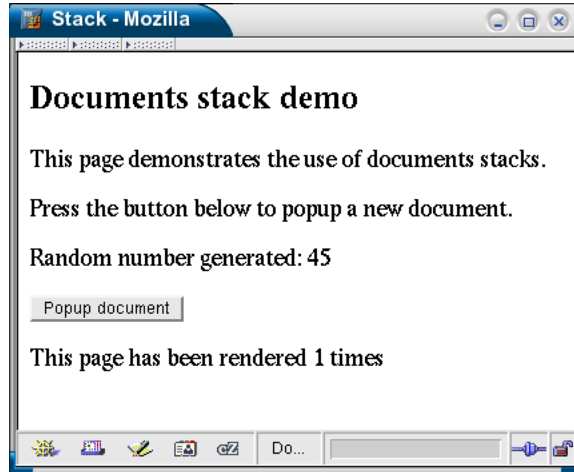


Figure 3.4: The page as it appears the first time is opened

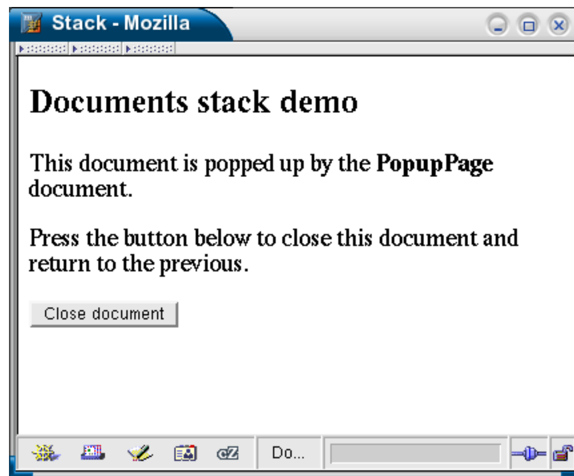


Figure 3.5: The page after the user clicked on the Popup document button

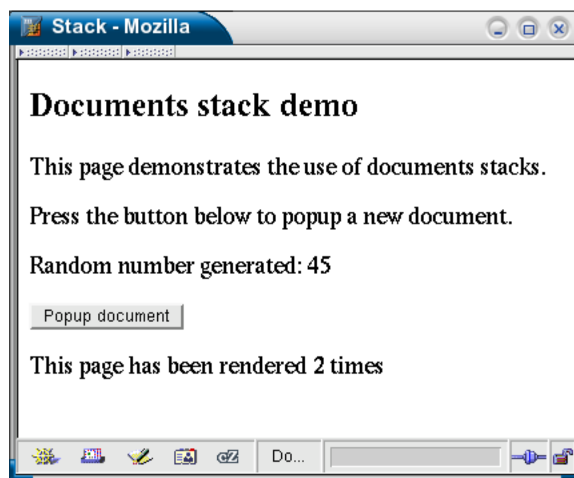


Figure 3.6: The page after the user clicked on the Close document button

Chapter 4

The *actions*

4.1 Introduction

Actions provide a great example to show *WebFormKit*'s extensibility capabilities: they represent a high level concept built on top of lower level *WebFormKit*'s core APIs.

Each single action represent a standard, simple behaviour that can be associated with an event: we can think of them as an alternative way to specify an event handler.

Actions can be very convenient since they can be specified in a *declarative* way, i.e. we can bind handlers to events directly in the XML file with no need to touch the Python one.

4.2 *Action providers*

Before we can use an action there must be some place where it is defined: usually actions are simply methods of the Python class associated to a *container* tag, i.e. a tag which can have children like web:document and web:form. Such containers are named *action providers* and they must define an attribute name to be used by children for specifying the action they want to be bound. As an example, the web:document tag provides three actions:

popup Open a new document and push it on the top of the page's stack.

open Clear the page's stack and push a newly opened document.

close Close and pop the document from the top of the page's stack.

Careful readers should have noticed that these actions have exactly the same behaviour as WebFormKitPage's methods described in section 3.2: in fact, they are really implemented by forwarding to those methods.

To bind their *default event* to web:document's action, controls can use the web:action attribute. For example, the following code binds the button's click event to the standard action close.

```
<?xml version="1.0" ?>
<web:document xmlns:web="http://www.disi.unige.it/WebFormLib.Web">
  <web:form>
    <web:button web:action="close" />
  </web:form>
</web:document>
```

4.3 *Actions* with parameters

The code shown in the previous section works properly just because the close action takes no parameters so that we can bind it directly to the button's click event.

When using actions with parameters the previous approach does not work and we need to load the Action extension, whose URI is http://www.disi.unige.it/WebFormLib.Action.

The Action extension provides a powerful directive (see section 2.4) named simple. Such a directive binds the default event of its parent with a special handler which collects its XML attributes and passes them as arguments to the action. Let us better explain this behaviour by the following example:

```
<?xml version="1.0" ?>
<web:document xmlns:web="http://www.disi.unige.it/WebFormLib.Web"
              xmlns:action="http://www.disi.unige.it/WebFormLib.Action">
  <web:form>
    <web:button>
      <action:simple web:action="popup" document="foo.xml" />
    </web:button>
  </web:form>
</web:document>
```

The action:simple in the code fragment above binds the button's click event to an handler calling the action popup with the document parameter set to foo.xml. The arguments are

passed *by name*: the XML attribute name must be exactly the same used during action's declaration.

4.4 *Actions* in action

Let us go back to the *Stack* example shown in section 3.4, where we can notice that on_popup_click and on_close_click handlers can be easily implemented using actions.

Figures 4.1, 4.2 and 4.3 shows the modified example.

```
import random

from WebFormLib import EventHandler
from WebFormKit import WebFormKitPage

# declare the Webware Servlet
class ActionExample(WebFormKitPage):
    defaultDocument = 'PopupDocAction.xml'

class PopupDoc(EventHandler):
    def on_open(self):
        self.count = 0
        self.random.web.text = random.randint(1, 100)

    def on_render(self):
        self.count += 1
        self.renderCount.web.text = self.count
```

Figure 4.1: The *ActionExample.py* file

```
<?xml version="1.0" ?>
<web:document xmlns:web="http://www.disi.unige.it/WebFormLib.Web"
              xmlns:action="http://www.disi.unige.it/WebFormLib.Action"
              web:code="ActionExample.PopupDoc">

  <h2>Actions demo</h2>
  <p>This page demonstrates the use of actions.</p>
  <p>Press the button below to popup a new document.</p>
  <p>Random number generated: <web:label web:id="random"/></p>

  <web:form>
    <web:button web:id="popup" web:text="Popup document">
      <action:simple web:action="popup" document="CloseDocAction.xml"/>
    </web:button>
  </web:form>

  <p>This page has been rendered <web:label web:id="renderCount"/> times</p>
</web:document>
```

Figure 4.2: The *PopupDocAction.xml* file

```
<?xml version="1.0" ?>
<web:document xmlns:web="http://www.disi.unige.it/WebFormLib.Web">

  <h2>Actions demo</h2>

  <p>This document is popped up by the <b>PopupPageAction</b> document.</p>

  <p>
    Press the button below to close this document and return to the
    previous.
  </p>

  <web:form>
    <web:button web:id="close" web:action="close">Close document</web:button>
  </web:form>

</web:document>
```

Figure 4.3: The *CloseDocAction.xml* file

Chapter 5

A complete example: the *multiplication table*

5.1 Description of the project

In this chapter we develop step-by-step a very simple web application using *WebFormKit*. The purpose of the application is to generate a *multiplication table* based on user's input: users will be able to change endings of either rows and columns.

To develop the application we need a couple of *WebFormKit*'s constructs we have not seen yet: these constructs will be explained in next sections.

5.2 The web:textbox and web:table custom controls

Until now the only custom controls we have seen are web:button and web:label. In our *multiplication table* we will use two other custom controls from the Web standard extension:

web:textbox This custom control is rendered using the standard <input type="text"> HTML tag, meaning that the user will see a text input box which he/she can type into. Python's code can read or modify textbox content using the value property.

web:table This custom control is in some way similar to web:label; it does not provide behaviour, only presentation layout: the difference is that while web:label is tailored for outputting simple strings of text, web:table is ideal when we want to output data in

a tabular form. The appendRow method adds new row to the table, while the clear method removes all the rows inserted before.

5.3 Validating forms

Often developers need to *validate* web forms, i.e. check that the user inserted values respecting some constraints: *WebFormKit* developers can use web:form's validate event to do their works.

The validate event is raised whenever the user clicks a button unless its causesValidation attribute is False.

In the event handler developers can check values and eventually show error messages, e.g. by setting web:label contents. In case of serious errors programmers can decide to interrupt event handling by raising a StopEvenProcessing exception. Since the validate event is always processed before web:button's click events this is a convenient way to ensure that the latter are executed only if user's input is valid.

5.4 Code analysis

Figure 5.1 shows the XML code of the document: it is composed by a web:table control that will shows the multiplication table and a web:form control which user can fill to modify table's layout. Lastly, there is a web:label that can be used to display validation errors

In figure 5.2 we can see the Python code that does the real work: the on_open handler makes the default table by forwarding to the _makeTable method, which firstly clears the table, then re-fills it with new values.

The on_updateForm_validate method is interesting; it does two checks:

- it tries to convert all textboxes into integer numbers;
- it checks that rows' and columns' end-value are greater than the corresponding start-value.

If one of the checks fails, this method sets an appropriate error message and stops the processing of subsequent events.

```
<?xml version="1.0" ?>
<web:document xmlns:web="http://www.disi.unige.it/WebFormLib.Web"
              web:code="MultiplicationTable.Table">

  <h2>The multiplication table</h2>

  <web:table web:id="table" style="border: 1px solid black"/>

  <br />

  <web:form web:id="tableForm">
    <i>Number of rows:</i> <br/>
    From: <web:textbox web:id="txtRowStart" size="3"/>
    To: <web:textbox web:id="txtRowEnd" size="3"/>

    <br/>

    <i>Number of columns:</i> <br/>
    From: <web:textbox web:id="txtColStart" size="3"/>
    To: <web:textbox web:id="txtColEnd" size="3"/>

    <br/>

    <web:button web:id="cmdUpdateTable">Update table</web:button>

    <br/>
    <web:label web:id="error" />
   </web:form>

</web:document>
```

Figure 5.1: The *Table.xml* file

```
from WebFormLib import EventHandler
from WebFormLib import StopEventProcessing
from WebFormKit import WebFormKitPage

# declare the Webware Servlet
class MultiplicationTable(WebFormKitPage):
    defaultDocument = 'Table.xml'

# declare the event handlers for the 'Table.xml' document
class Table(EventHandler):
    def on_open(self):
        # make the default table
        self.rowStart = 1
        self.rowEnd = 5
        self.colStart = 1
        self.colEnd = 10
        self._makeTable()

    def on_tableForm_validate(self):
        self.error.web.text = ''

        try:
            self.rowStart = int(self.txtRowStart.value)
            self.rowEnd = int(self.txtRowEnd.value)
            self.colStart = int(self.txtColStart.value)
            self.colEnd = int(self.txtColEnd.value)
        except ValueError:
            self.error.web.text = 'Invalid number'
            raise StopEventProcessing

        if self.rowEnd < self.rowStart:
            self.error.web.text = 'Row end must be greater than row start'
            raise StopEventProcessing

        if self.colEnd < self.colStart:
            self.error.web.text = 'Column end must be greater that column
                start'
            raise StopEventProcessing

    def on_cmdUpdateTable_click(self):
        self._makeTable()

    def _makeTable(self):
        self.table.clear()
        for i in xrange(self.rowStart, self.rowEnd+1):
            row = [i*j for j in xrange(self.colStart, self.colEnd+1)]
            self.table.appendRow(row)
```

Figure 5.2: The *MultiplicationTable.py* file

Bibliography

[1] Active Server Pages. msdn.microsoft.com/asp/.

[2] ASP.NET Web: the Official Microsoft ASP.NET Site. http://www.asp.net.

[3] Nevow: A Web Application Construction Kit. http://www.nevow.com.

[4] PHP: Hypertext Preprocessor. http://www.php.net.

[5] Python Programming Language. http://www.python.org.

[6] RFC 1738, Uniform Resource Locators (URL). http://www.faqs.org/rfcs/rfc1738.html.

[7] Webware For Python. http://webware.sourceforge.net.