WebDSL language reference

The WebDSL Team

30th October 2008

Contents

Part I

Introduction to WebDSL

Introduction to WebDSL

1

1.1 AN INTRODUCTION TO DOMAIN SPECIFIC LANGUAGES

Software Engineering can be summarized as the struggle for automating or optimizing the software development process. Main goals of software engineering are the improvement of development time, reducing the efforts in software maintanaince and increasing software quality. Since the early days of computer science abstractions have been build on top of other abstractions, leading away from the details of processor instructions, memory management etc. etc., in an attempt to reduce the number of concerns a software developer has to deal with. Those abstraction have led to the huge amount of GPL languages nowadays exist in industry, like Pascal, C, Python, Java and many many more.

To improve the improve the capabilities all GPL languages support notions like 'libaries' and 'API's', to enable reuse of specific functionality. However a GPL languages have a set of restrictions, which restrain to easily abstract from the GPL, leaving the developer with lots of boilerplate code that has to be written (like initializing database transactions, write conversion mechanims etc). The restrictions created by the nature of GPL's can be summarized as follow:

The inexstensibilty of syntax Libraries offer the capability to automate specific semantic behaviour, like querying a database. However, no syntactic extensions or embeddings are possible, to be able to define SQL queries in an intuitive manner. One cannot avoid the usage of strings or other kinds of code-sugar.

The absence of domain restrictions In a GPL one can express in every function or class everything that can be expressed in the GPL as a whole. This makes GPL's very expressive in general, it always enables unstructured, semantic incorrect or very indepent code.

The huge amount of code needed for general patterns Within a certain domain, certain actions or patterns have to be executed frequently. Little abstractions can be made using helper or library functions, inheritance or even macro's to reuse as much code as possible. However the ways abstractions can be made are limited, and not every abstraction can be expressed in the language

The huge depencens on the target GPL Once a application has been developed in a certain GPL, it takes much effort to rebuild the application in another GPL, even when the language abstractions, semantics and syntax are very similir (e.g. Java and C#). This show how much code relies on the

> language it is written for, even when the abstractions and semantics required by that language are very similar to the ones made available with another GPL.

To deal with the probles stated above, much effort has been put in Model Driven Engineering. Goal is to have a model describing an application, abstraction from the details of programming languages. Domain Specific Languages provide an interface to a developer where only behaviour related to the domain should be specificied, while the compiler, code generator or model interpreter takes care of the boilerplate code needed for a fully working application, query or whatever the target of a DSL. In [dezelfde paper dus] a DSL is defined as

> A domain-specific language (DSL) is a high-level software implementa tion language that supports concepts and abstractions that are related to a particular (application) domain.[1]

1.2 PURPOSE OF WEBDSL

WebDSL is an attempt to create a domain specific language for web-applications. Things like peristency, request handling and GPL/ Markup code generation will be taken care of by the language compiler. The responsiblility of the user is to think about the data model, navigation, page layout and such issues. No boilerplate code should be needed to be written by the user, and the way to define te model (textual in this case) should feel intuitive. The purpose of WebDSL is stated at WebDSL.org as follows

> WebDSL is a domain-specific language for developing dynamic web applications with a rich data model. [2]

The language supports modeling of

- the applications domain in data entities
- the applications user interface
- the page flow
- security and access restrictions
- dynamic behaviour, commonly refered to as Ajax.
- styling

Last but not least, webDSL applications can be targeted to different front- and backends, including a Tomcat servlet, Google Apps Python backend and a Ajax enabled (dynamic pages) or Ajax-disabled (static pages) front-end.

[1] TODO
[2] TODO

1.3 SCIENTIFIC WORK

lijst met verschenen papers

Installing WebDSL for WebDSL users

2

This section describes how to install WebDSL if it is only to be used to compile WebDSL applications. Currently the compiler is only available for Unix based systems, like Linux and Apple's OS-X. The required packages depend on the target platform of WebDSL applications. The WebDSL compiler itself has the following dependencies:

- The *gcc* C compiler

2.1 A NOTE ON NIX DEPLOYMENT SYSTEM

WebDSL and Stratego/XT (one of the depencies when extending WebDSL) can be installed manually or using the *Nix*[1] deployment system, which is highly recommended. Nix takes care of depencies and provides the functionality to easily obtain the latest version from the buildfarms.

The latest release of Nix can be found at its website *http://www.nixos.org*. Use the *configure*, *make* and *make install* commands to install the deployment system. As a final step a autostart instruction needs to be added to your *.profile* file:

```
. /nix/etc/profile.d/nix.sh
```

2.2 INSTALLING THE WEBDSL COMPILER

Two ways exist to obtain WebDSL, either from the Nix channel, or downloading the sources directly. The latest source can be downloaded from *http://buildfarm.st.ewi.tudelft.nl/releases/strategoxt/channels-v3/webdsls-unstable/index-webdsls.html#webdsls-Unstable*.

Using Nix, one can subscribe to the WebDSL channel using the following code:

```
nix-channel --add http://buildfarm.st.ewi.tudelft.nl/
    releases/strategoxt/channels-v3/webdsls-unstable
nix-channel --update
nix-env -i webdsl-8.3pre1057
```

After subscribing to the channel updating to the latest version can be done simply using the following commands:

```
nix-channel --update
nix-env -u webdsls
```

[1] http://www.nixos.org

After obtaining the sources, executing the well-known commands *configure*, *make* and *make install* should do the job. Test your installation by building a first WebDSL application, as described in 4. Note that your test application cannot be deployed until the instructions in the next section are executed.

2.2.1 *Installing the WebDSL backends*

Currently, webdsl applications can be compiled to Java Servlets hosted by a Tomcat server[2] or to a Python script which can be hosted by Google's AppEngine[3].

Installing the Tomcat backend

Installing the Python backend

[2] http://tomcat.apache.org
[3] http://code.google.com/appengine/

Installing WebDSL for WebDSL developers

3

To build your own version of WebDSL, it takes some extra dependencies that needs to be satifisfied before being able to build the compiler. WebDSL has ben beeld on the program transformation toolset *Stratego/XT*[1]. First off all, make sure the depencies of Stratego/XT are satisfied. The following linux packages are needed:

- install
- curl
- m4
- autoconf
- automake
- pkgconfig
- libtool
- subversion

Stratego/XT can be installed using the Nix distributed system. Make sure Nix is installed as described in 2.1. To install Stratego/XT open a console and execute the following commands:

```
nix-channel --add http://releases.strategoxt.org/
    strategoxt-packages/channels/strategoxt-packages-
    stable
nix-channel --add http://releases.strategoxt.org/
    strategoxt/channels/strategoxt-unstable
nix-channel --update
```

Then install the stratego packages:

```
nix-env -i aterm java-front sdf2-bundle stratego-shell
     strategoxt strategoxt-utils
```

Finally, update your */.profile* again and add:

```
export PKG_CONFIG_PATH=~/.nix-profile/lib/pkgconfig
```

[1] http://www.strategoxt.org

The full sources of WebDSL can be retreived from the svn repository at *https://svn.strategoxt.org/repos/WebDSL/webdsls*. Use the following command to obtain the sources:

```
svn co  https://svn.strategoxt.org/repos/WebDSL/
    webdsls/trunk
```

Finally, you can compile your own webdsl compiler using

```
cd webdsls
./bootstrap
./configure --disable-shared --prefix=/usr/local
make
make install
```

Install any required backends as described in **??** and check your installation as described in 4

3.0.2 *A note on Mac Users*

Macintosh users require to install some applications already availalble in most linux environments. First install Apple *XCode* from your DVD. Secondly download and install *DarwinPorts*. Follow the instructions and then install the required ports:

```
port install curl m4 autoconf automake pkgconfig
    libtool subversion
```

Running your first application

4

Compiling this documentation

5

The sources of this documentation can be found in the ./doc/langref/ directory of the WebDSL sources. Compiling the document requires Latex[TODO] and a number of other packages installed, among them *Rubber* and *TexLive* to be installed. After updating the WebDSL source to the most recent version, you may compile your own documentation postscript with the command.

```
rubber -f -s --inplace -d index.tex
```

Part II

Core Language Reference

WebDSL applications and their organization

6

6.1 THE STRUCTURE OF A WEBDSL APPLICATION

WebDSL application are organized in **.app* files. Each .app file has a header, that either declares the name of a *module* or the name an *application*. The declared name should be identical to the filename. Each *application* needs an .app file that declares the name of the application. This is the name refered to in the application.ini file (see below).

An application can be organized in different *modules*. In a typical .app file the header is followed by a serie of import statements, which contain a path to other modules (without extension). In this way your application can be seperated over several files, and modules can be reused.

Within .app files one can define sections. A section is merely a label to indentify the structure of a file. Most sectionnames have no influence on the program itself, some have however, for example in styling definitions.

The real contents of a .app file are a serie of definitions. This might be page-, template-, action- or entity definitions. Other kinds of definitions might be introduced by WebDSL modules[1]. Those definitions will be examined in detail in the next chapters.

A very simple application might look like:

Listing 6.1 Helloworld.app

```
application HelloWorld                          1
imports MyFirstImport                           2
section pages                                   3
define page home () {                           4
    "hello world" IAmImported()                 5
}                                               6
```

Listing 6.2 MyFirstImport.app

```
module MyFirstImport                            1
section templates                               2
define IAmImported() {                          3
    spacer "I am imported from a module file"   4
}                                               5
```

[1] A module might either refer to a module of an WebDSL application, or to an module of the WebDSL compiler itself. In this case the latter one is reffered to.

6.2 APPLICATION CONFIGURATION

In the *application.ini* file compile-, database- and deployment information is stored. Executing the *webdsl* command in a certain directory will look for a *application.ini* file to obtain compilation information. If no such file was found, it will start a simple wizard to create one.

The application.ini file contains the following information

Appname The name of the application compile. The compiler will look for a .app file named likewise and start compiling it

Backend The target type of application. This can be either *servlet* or *python*. Older versions of WebDSL might also support *seam* which generates JSF/Seam/JBoss servlets.

Tomcatpath or JBosspath Depending on the choice for backend, this field should contain the directory the compiled applications needs to be deployed or the proper backend is stored. For example */usr/bin/apache-tomcat*.

DBMode This field indicates if the compiler should try to create a new database, or try to sync it with the new application to avoid loss of data. Valid values are *create-drop* and *update*. The latter one might lead to unpredictable results however if data model is changed to much.

Furthermore there is a list of items needed to configure the database and email functionality. Those items speak for themselves. They are shown in the following example configurationfile:
application.ini:

```
export BACKEND=servlet
export TOMCATPATH=/opt/tomcat
export JBOSSPATH=
export APPNAME=HelloWorld
export DBSERVER=localhost
export DBUSER=michel
export DBPASSWORD=youdidliketoknowdontyou
export DBNAME=webdsldb
export DBMODE=create-drop
export SMTPHOST=localhost
export SMTPPORT=25
export SMTPUSER=
export SMTPPASS=
export SMTPSSL=false
export SMTPTLS=false
```

Page and template definition

7

WebDSL applications can be viewed as a set of defitions. In this chapter we will look into the *page* and *template* definitions. Page and templates definitions define directly the contents shown to the end-user and the behaviour he will expierence. Those definitions define the looks and the flow of your application.

Pages and templates have a very similar structure. The semantics however are slightly different. First of all, *pages* are directly refereable to with a URL. Second, a user is always at one page at a time. Navigation or program flow is achieved by moving from one page to another[1]. A page can include several templates, as can templates themselves. However pages cannot be loaded inside other pages.

Templates on the other hand merely function as container to gather a set of *elements* (see next chapter). Templates can call each other to enable reuse as much as possible in WebDSL applications.

Both pages and templates can take a list of *arguments* that the caller of a page/ template needs to provide. Overloading parameters is allowed. The syntax of pages and templates is as follows

Listing 7.1 Definition syntax

```
defintion   := "define" modifier* name "(" formalargs ")"   1
    "{" elements* "}"
formalargs  := ( formalarg "," )*                           2
formalarg       := name ":" type                            3
```

Types and elements will be explained in the next chapters. Names are identifiers as we known them in programming languages like C en Java. Modifiers influence the type of object we are defining. This chapter introduced the modifiers *page* and *template*. If the modifier is omitted, *template* is assumed. Other valid modifiers are *email*, *feed* and *local*. They will not be explained here however. A simple page definitions might look like this:

Listing 7.2 Simple page definition

```
define page user(u : User) {                                1
    "The name of this user is " u.name                      2
}                                                           3
```

[1] However enabling Ajax in your applications might change this slightly

7.1 REDEFINITIONS

Furthermore it is allowed for a page or template to redefine a template call. This allows to reuse a template while redefining a small part of it. This mechanic is shown in the example below. Both the *home* and the *publication* page call the *main* template. This main template introduces a menu, a sidebar and a (not necessarily) empty body. This way both *home* and *publication* get those elements for free with the body at the proper place. However they still are able to define their own body.

Listing 7.3 Redefinition example

```
application redefine
section home page

define page home() {
  main()
  define body() {
    //some other elements
  }
}

define page publication(p : Publication) {
  main()
  define body() {
    //some completely other elements
  }
}

section templates

define main()
{
  sidebar() body() manageMenu()
}

define body() { //empty }
define manageMenu() { "menu" }
define sidebar() {"sidebar"}
```

User Interface elements

8

The contents of pages and templates are defined by templates elements. Roughly different kind of template elements exist. Control-flow elements (such as loops and variable declarations), User interface elements, actions and redefinitons. Redefinitions are mentioned in the previous chapter, control-flow elements and actions will be introduced in a later chapter. In this chapter we will focus on the variable declarations and user interface elements.

User interface are some of the most concrete elements in WebDSL, as they closely resemble to the nature of HTML. Altough most elements can directly be mapped to HTML, some elements have extra semantics, such as headers. Furthermore the elements abstract from HTML, so in future it might be possible that other UI standards might be targeted (such as XUL, Gtk or Windows forms).

8.1 GENERAL STRUCTURE

In general an interface elements consists of four parts. An element indication, a set of parameters, a set of attributes and a set of children. All currently available elements in WebDSL will be investigated in the next section. Attributes are static configuration properties, like an id declaration (so the element is uniquely refererable trough the whole application), a class declaration (to override style properties) and such things. As those attributes are particulary useful in an dynamic application, they will be explained in the *modules - ajax* **??** chapter. However some very useful properties will be mentioned already in this chapter. In general, attributes are not limited to a single kind of element, but are valid for different kind of objects.

Parameters are actual WebDSL objects which stronly influence the behaviour of an element. For example *input* takes a data object to generate an input for, and *navigate* takes a pagecall to generate a link.

Some elements accept a group of children, which will be embedded in the object. However, not all elements take a group of children. Note that attributes and arguments are seperated from each other using comma's, but elements are seperated by just using whitespace. The general syntax of a interface element is:

Listing 8.1 WebDSL element syntax

```
element := elementtype arguments? atributes? children?      1
arguments   := "(" argument ")"                             2
argument    := expression ( "," argument )?                 3
attributes  := "[" attribute "]"                            4
```

```
attribute := attributename ":=" attributevalue ( "," attribute )?
children    := "{" element* "}"
```

Expressions will be explained in a later chapter, valid attributenames and values will be explained whenever applicatble. In short this is what a set of UI elements look like:

Listing 8.2 Attribute example

```
table[class:=maxwidth] {
  row {
    column {
      group("News") {
                //some newsitems
           }
       }
    }
}
```

8.2 INTERFACE ELEMENTS

8.2.1 *Text*

To output text, quoting it between '"' is enough. Escaping is done using 'ás is common in most languages. Note that due to the nature of HTML ' n' and other whitespace escapes have no meaning, except when used inside a *pre* elements, which outputs text literally.

Listing 8.3 Simple Text usage

```
define simpleText {
    "This is some text.." "... and here is even more \"
       quoted\" text"
}
```

To output variables one can use the generic *output(Expression)* construction or the *text(String-Expression)* construction. To assign styling to text, one might use *container* element discussed below.

8.2.2 *Basic elements*

title Takes no arguments, but its contents is used to set the current page title.

header Generates a text header. The children of header are used as caption. The header depth is determined automatically, and so is the style. So a header element will result in a HTML *h1* element, but if used inside a

section that already contains a header, it wll result in a HTML *h2* element etc.

spacer or horizontalspacer results in a horizontal line (HTML: *hr*).

image(string) displays an image. The first argument is used as URL to the image, or can be a relative path from the applications directory to the image. Useful attributes for an image are *height* and *width*. Example: *image("/images/logo.png")[height:= 100px, width:=20%]*

pre is used to display preformatted text. The text inside will be literally shown by the browser, using an alternative font and maintaining whitespace. Especially useful to display sourcecode.

label(string) is used to render a HTML *label*. The caption is determined by the first argument. If the label has any children, the label will link to the first child (clicking the label will give the focus to that child). If used inside a *row* or *group* element, the label and its children will be rendered in different columns. This way it is very easy to align different input elements and their captions. Example (notice that a label should *not* be put inside a *column* as columns will be generated automatically for labels:

Listing 8.4 Group example

```
group("edit "+user.name) {                                    1
    groupitem{label("name")          { input(user.name        2
        )} }
    groupitem{label("password)   { input(user.password        3
        ) } }
}                                                             4
```

8.2.3 *Tables*

table defines a tabular structure. Tables are useful to display a set of simalar data (records), or to achieve a column-based layout. Table accepts any kind of element as child but every child that isn't a row will automatically wrapped by one. In other words putting elements directly in a table will order them below each other. It is advisable to explicitly use *row* elements to avoid this implicit behaviour. A common usage pattern is to open a *table* element, open a *for* loop (see next chapter) to iterate over some data and display that data inside a *row* element. See the example below.

row starts a new row. Is only useful in the context of a table, a control-flow element or as root element of a template (which can be called inside a table). Every element inside a row will be wrapped automatically with a column element if it isn't already one.

column indicates a new column or *cell*. Every row inside a table should have an equal number of columns to have no unpredicatble output. Use the rowspan or colspan attributes merge multiple cells.

header results in a HTML *th* element, indicating an alternative layout. Using *rowheader...* or *rowcolumnheader..* both results in a single cell acting as header.

The following code fragment displays a list of users. Note that all column elements could be omitted to achieve the same effect. However to force multiple elements to be displayed in one cell, the column element is necessary.

Listing 8.5 Table example

```
table {                                                                    1
    row { column { header { "name" } } column { header {                   2
        "nickname" } } }
    for(u: User) {                                                         3
        row {                                                              4
            column { output(u.name) }                                      5
            column { output(u.nickname) }                                  6
        }                                                                  7
    }                                                                      8
}                                                                          9
```

8.2.4 *Grouping elements*

Besides the table structure, WebDSL has a number of other elements to group elements.

block(string? string?) is used to group a set of elements. Blocks are typically used to position elements using classes (see *modules - styling and layout*). Beside that, blocks are useful to replace or update pages partially (see *modules - ajax*). For that reason in practice the *id* attribute is set on many blocks. A block takes zero, one or two arguments. All arguments should be strings and are used to assign styles to a block. All blocks result in HTML *div* elements.

section result in a HTML *span* elements, just as plain text does. However section can be used to group multiple items inside a single span. This might be useful to assign a class attribute to multiple items. In contrast to blocks, sections are inlined. Furthermore sections keep track of the nesting depth, this is very useful in combination with the *header* element.

container result in a HTML *span* element too, but without the nesting semantics of sections.

par is used to create text paragraphs, and result in a HTML *p* element.

list and listitem are used to display lists. By default each item receive a bullet and all items are displayed below each other. This is completely configurable however using styling. To achieve predictable results, it is recommended to use only listitems as ui-children in list, and use only text based ui-elements in listitems (e.g. text, navigate, actionLink etc).

group(string) and groupitem are very useful to group a set of elements visually, so the end-user notice a natural grouping of a set elements concerning the same issue. By default a group element renders a border and the first argument is used as caption. However it is allowed to provide no caption. The children of a group should be groupitems, otherwise each element will be wrapped by one. If a groupitem contains multiple children, each child results in a column, similar to the table structure.

8.2.5 *Template calls*

Templates (see previous chapter) can be called as being an ordinary element. We saw this already in the chapter about pages and templates. Templatecalls take arguments (the parenthesis are mandatory) but take no children or attributes. Using templates we can rewrite the table listing earlier in this chapter to the following:

Listing 8.6 Template call example

```
define template showUser(u: User) {
    column { output(u.name) }
    column { output(u.nickname) }
}

define page home() {
    table {
        row { column { header { "name" } } column {
            header { "nickname" } } }
        for(u: User) {
            row { showUser(u) }
        }
    }
}
```

8.2.6 *Navigation elements*

WebDSL knows two types of elements to define how an user is able to navigate trough an static application. Navigates an actions. Navigates replaces the current page with another page onces selected, action links to an action, which should return page that will be displayed after execution of the action (see next section). For dynamic applications more ways exist to navigate trough an application. See *modules - Ajax*.

navigate(target) creates a link (or *anchor (a)* in HTML terms). A navigate takes zero or one argument. The argument can either be a string containing an URL or a pagecall (for example *navigate(home())"click me!"*. Clicking on one of the children of a navigate will result in navigating to the provided location. The navigate without argument is only useful in combination with Ajax.

navigatebutton(target, string) has the same semantics as a navigate, but takes two arguments and displays a button instead of a random element. The first argument again is the target location and the second argument is a string used as caption for the button.

8.2.7 *Forms*

form is used to group a set of user inputs that should be submitted simultaniously to the server. Submitting usually done by creating a button that executes an action containing an submit. An example of a form is shown below. Inputs are created using the input element, explained in the next section.

action(string, actioncall?) shows a button which executes an action defined on the same page or template. The first argument is the caption of the button, the optional second argument is the call to the action to be executed.

actionLink(string, actioncall?) takes the same arguments as *action* and has the same behaviour, but shows a link rather than a button.

A simple form might look like the following listing. Note that there is a distinction between the action element (*action("remove", remove())*) and the action definition (*action remove() ...*). Action definitions will be explained in a later chapter.

Listing 8.7 Simple form example

```
define editFolder(f: Folder) {
  form {
    group("editing folder "+f.name) {
      groupitem{
        label("name ") {input(f.name) }
      }
      groupitem{
        label("description") {input(f.description) }
      }
      groupitem {
        action("remove", remove())
        action("save", save())
      }
    }
```

```
  }                                                    15
  action save() {                                      16
    f.save();                                          17
    return showfoldercontents(f);                      18
  }                                                    19
  action remove() {                                    20
    f.delete();                                        21
    return showfolderlist();                           22
  }                                                    23
}                                                      24
```

8.2.8 *Generitave elements input and output*

There are two basic constructions to input and output data: *input(object)* and *output(object)*. Output is comparable to Java's *toString()* method. The type of output is determined compile-time on the type of the variable provided. For example, a String variable results in a plain string being outputted, but a Image object results in a *image* element being created. The same holds for the input element. Providing a String variable as input results in a simple textbox, providing an Image variable results in an uploadbox being created. It is possible to write your own input or output overload for a specific type by defining a template named *input* or *output* that accepts the proper type as first parameter. For example

Listing 8.8 Redifine output example

```
define output(users: List<User>) {                     1
    list {                                             2
        for(u: users) {                                3
            listitem {                                 4
                output(u.nickname) ":" output(u.name)  5
            }                                          6
        }                                              7
    }                                                  8
}                                                      9
```

Specialized input and output elements

Every *input* and *output* element will be rewritten to a more specialized input or output element. Those elements can be called directly instead of using *input* or *output*. This way the default inputfield for example can be replaced by a textarea. For list input possiblities, it is important to take a look at the *control-flow* elements in the next chapter. This are all the input and output elements available by default:

inputBool(boolean) results in a checkbox being created.

outputBool(boolean) results in a disabled checkbox (with the proper state as one might expect)

inputString, outputString are used in combination with objects of type *String*. The input displays a simple textfield. The output can also be used in combination with objects that are not of type String, but results might be unpredictable.

inputInt, outputInt are used in combination with objects of type *Int*. The input displays a simple inputbox.

inputFloat, outputFloat are not implemented yet

inputDate, inputTime, inputDateTime, outputDate, outputTime and outputDateTime all take one argument, a expression of the primitive type Date, Time or DateTime and act as one might expect. *Not implemented in the current version of WebDSL*

inputSecret, outputSecret display a password inputfield, which do not show the charachters to the users, but asterixes. outputSecret display a fixed number of asterixes. The provided object should be of type *String* or *Secret*

inputUrl is used in combination with the *Url* type, but works similar to inputString.

url displays an URL rather than creating a link.

inputText creates a multiline textarea in which a String type can be inputted.

outputText Outputs text, not literally but using a *Markdown processor*[1] to process the text to be displayed.

inputEmail, outputEmail are not implemented yet

inputFile, outputFile create either an uploadbox or a download link, to send or retreive a file. Both require a single argument of primitive type *File*. However, downloading can also be achieved by using actions too, see the primitive types chapter. This way access control rules can be applied.

inputImage, outputImage are used in combination with objects of the type *Image*. The input creates an uploadbox, but the output displays the image. To download the image use a action that calls the *download()* method on the image variable.

inputWikiText, outputWikiText are used to create Wiki like pages. The input again is a textarea, the output uses a Wiki text parser te generate output. WikiText elemens can be used on *String* and *WikiText* objects.

[1] A markdown processor transforms plain text to rich XHTML text. Users can identify headings etc. intuitively. For more information see for example http://daringfireball.net/projects/markdown/

captcha element can be used to display an image to an user which much be copied in a textfield, so the server know it has to do with a real human, not a script. Used by many popular sites in signup forms.

8.2.9 *Menu's*

menubar(direction?) is used to start a menu. The parameter should contain *"horizontal"*, *"vertical"* or be omitted. Horizontal is the default. As children it is adviced to use *menu* items, others are allowed but might break styling.

menu is used to define one menu in a menubar. It should contain at least one element (menuheader) and optionally more menuitems in which case it will unfold if selected.

menuheader indicates the first element in a menu. It is advised to use only text or *navigate* elements inside it.

menuitem represents a single item inside a menu. It is advised to use only text or *navigate* elements inside it.

menuspacer draws a line inside the current menu, to enable some grouping inside menu's.

8.2.10 *Markdown notation*

A number of usual WebDSL element constructions have shorthand notation, a bit comparable to the *Markdown* or *Wiki* notation of *HTML* elements. The constructions below are available to make your code shorter (but a bit less readable). The *attrs* mentioned are the same as explained in the general structure, and are optional for each element.

* *attrs? element* shorthand for a *listitem* with one child

−− *attrs?* shorthand for *spacer*

= *attrs? element* = shorthand for *header* with one child

#[*attrs? element**] shorthand for a *block* definition, with multiple children

< *attrs? (*| *element*)** > shorthand for *row* with *column*'s (seperated by "|") in it. Can also be used instead of *groupitem* in groups. It is possible to specify no columns, in that case every single child element will be treated as being a seperate *column*. Use this in combination with *label*.

Listing 8.9 Markdown example

```
template t() {                                  1
    group("a group") { //or table               2
        <|"1" | "2">                            3
        <|"3" | "4">                            4
```

```
    }
    --
    = "below, a list" =
    list {
        * "1"
        * "2"
        * "3"
    }
}
```

Control flow elements

9

WebDSL has a small number of elements to influence the flow of the rendering of a page or template, for example to make certain parts of the interface optional, or to iterate over a set of elements. Note that those control flow elements can be used as *statements* too.

9.1 VARIABLE DECLARATIONS

Variables can be used troughout a template to rembember and manipulate complex objects troughout the template. Variables in WebDSL are scoped as one might expect. Variables are declared with the following syntax:

```
VarDecl := "var" Id ":" Sort (":=" Exp)?
```

Note that the declaration is not ended with semilicon if used inside a template. If the declaration is used as statement (inside an action for example) one *must* end it with a semilicon. Inside the scope the variable is directly accessable using the given id, and it hides any parameters with the same name.

If an action is declared inside the scope were the variable was declared, it is not always necessary to pass the variable along as parameter, but it is recommended as the behaviour is defined more explicly. If the variable is changed during execution (for example due to reassignment by a for-loop) it is necessary to use it as a parameter.

Listing 9.1 optional but recommended variable passing

```
define t () {                                          1
    var s : String := "foo"                            2
    action("execute", doSomethingWithVar(s))           3
            //can be invoked without passing 's'       4
    action doSomethingWithVar(s: String) {             5
            //if this formal argument was omitted      6
        return v(s);                                   7
            //but now the outer 's' is hidden          8
    }                                                  9
}                                                      10
                                                       11
define u () {                                          12
            //this style is unrecommended              13
    var s : String := "foo"                            14
    action("execute", doSomethingWithVar())            15
    action doSomethingWithVar() {                      16
```

```
        return v(s);
            //s is available due to the scoping
    }
}

define page v (s2: String) {
    //...
}
```

9.2 IF-ELSE CONSTRUCTION

ıconditional template elements WebDSL has an *if-else* construction very similar to the ones available in GPL's like Java. The syntax is defined as

```
If  := "if" "(" Exp ")" "{" element* "}" ( "else" "{"
   element* "}" )?
```

Note that the curly braces are not optional in the case of a single statement, which is usual in other languages. The *else* part is optional. The expression needs to evaluate to a *boolean* value.

9.3 FOR-LOOPS

The for statement is quite powerful construction in WebDSL. It is able to fetch objects from the datastore automatically, and iterate over them. The behaviour is very similar to the *SQL Select* statement The syntax is defined as

```
For       := for "(" Id ":" Sort ("in" Exp)? Filter? "
   )" "{" element* "}" ( "seperated-by" "{" element*
   "}" )?
Filter      := ("where" Exp) ? ("order" "by" OrderExp)
   ? (Limit | Offset)?
OrderExp    := Exp ("asc" | "desc")?
Limit           := "limit" Exp Offset?
Offset      := "offset" Exp?
```

The most simple form of for, like *for(u: User)*, iterates over all objects of a certain type (or inheriting from) available in the application. To restrict the set of objects to be evaluated one can use the *in* construction. In requires an expression that returns a *list* of the provided *sort*. The *filter* and *order* parts need to be defined in terms of the iteration variable (identified by the first *id*).

The *seperated-by* part can be used to put elements *between* every of iteration of the for-loop. In other words, the elements are not inserted if the loop is executed zero or one times and inserted once (between the first and second element) if the loop is executed twice, etc. This is very useful for creating

comma-seperated lists without a comma to much at the end (or beginning) of the list.

Listing 9.2 For-loop advanced example

```
define t(persons : List<Person>) {                                    1
    for(p : Person in persons where p.lastName = "Foo"                2
        order by p.firstname asc limit 10 offset 0) {
         output(p.firstname)                                          3
  } seperated-by { "," }                                              4
}                                                                     5
```

Primitive Data Types

10

Defining Data Types

11

Actions and functions

12

Expressions

13

Part III

Module Reference

Access Control

14

15 Styling

WebWorkflow

16

Ajax

17

Part IV

Examples

Part V

A note about Stratego/XT

Introduction to Stratego/XT

Core transformations of Stratego

Useful commands

Part VI

Architecture of the WebDSL compiler

Part VII

Debugging guide to WebDSL applications and the compiler

Part VIII

Index

Listings

Index