WebDSL language reference

The WebDSL Team

22nd October 2008

Contents

Part I

Introduction to WebDSL

Chapter 1

Introduction to WebDSL

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

Chapter 2

Installing WebDSL for WebDSL users

This section describes how to install WebDSL if it is only to be used to compile WebDSL applications. /nix/etc/profile.d/nix.sh . 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/ 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-
nix-channel --update
nix-env -i webdsl-8.3pre1057
```

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

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
```

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 **??**. Note that your test application cannot be deployed until the instructions in the next section are executed.

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/

Chapter 3

Installing WebDSL for WebDSL developers

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 **??**. To install Stratego/XT open a console and execute the following commands:

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

Then install the stratego packages:

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

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 **??**

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 subversio
```

Chapter 4

Running your first application

Chapter 5

Compiling this documentation

The sources of this documentation can be found in the ./doc/langref/ directory of the WebDSL sources. Compiling the document requires Latex[TODO] and Rubber[TODO] 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

Chapter 6

WebDSL applications and their organization

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:
HelloWorld.app:

```
application HelloWorld
imports MyFirstImport
section pages
define page home () { "hello world" IAmImported() }
```

MyFirstImport.app:

```
module MyFirstImport
section templates
define IAmImported() { spacer "I am imported from a module file" }
```

[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 JS-F/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
```

Chapter 7

Page and template definition

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

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

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:

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

[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.

```
    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"}
```

Chapter 8

Template elements

Chapter 9

Primitive Data Types

Chapter 10

Defining Data Types

Chapter 11

Actions and functions

Chapter 12

Expressions

Part III

Module Reference

Chapter 13

Access Control

Chapter 14

Styling

Styling sublanguage

This is a normal paragraph.

This is a preformatted code block.

This is code: a := b * (c + d);

This is verbatim: a := b * (c + d);

emph strong Overview of styling properties and their types: Property Type Element Value Example align Align

1. left 2. right 3. center

align := Align.center; background-color Color

* hexcolor * e.g. white, blue, ...

background-color := Color.red border-color Color see background-color border-color := f0f0f0; border-style BorderStyle

1. solid 2. dotted 3. dashed 4. double 5. none

border-style := BorderStyle.solid; border-width Length border-width := 1px; font Font e.g. Lucida, Arial, Verdana, e.g. font := Font.Lucida; font-color Color see background-color font-color := f4f4f4; font-line Line

1. under 2. over 3. through 4. none

font-line := Line.under; font-size Length e.g. 2px, 1em font-size := 1em; font-style FontStyle

1. italic 2. bold 3. normal

font-style := FontStyle.italic;

Chapter 15

WebWorkflow

Chapter 16

Ajax

Examples

A note about Stratego/XT

Introduction to Stratego/XT

Core transformations of Stratego

Useful commands

Architecture of the WebDSL compiler

Debuging guide to WebDSL applications and the compiler