OpMath

DMLDoc:

Web Designer's Guide

D_2

Software Release: DMLDoc Standard Edition 02

Documentation Release: 01

---

http://www.opmath.com/projects/components/utilities/

Documentation Catalogue

DMLDoc: Installation Guide - Steps through the process of installing and running DMLDoc. The guide covers the most popular operating system, development environment and application server combinations in detail, but also provides generic information.

DMLDoc: Web Designer's Guide - this document.

DMLDoc: Programmers Guide - Describes the use of DMLDoc in Java server-side programming. Covers the architecture of the runtime environment and the structure of DMLDoc applications. The guide steps through the construction of DMLDoc applications, based on example code which can be got from:

http://www.opmath.com/projects/components/utilities/

Assumes some knowledge of Java programming.

DMLDoc JavaDocs - A complete doclet of JavaDocs for all the packages and classes for DMLDoc. Essential reading for DMLDoc server-side programmers. Available for download or online at:

http://www.opmath.com/components/utilities/1.1/docs/api/

All documents in this catalogue are Copyright © 2003 The Open Math Company Limited, 7 Wyndham Street, Brighton, UK. All Rights reserved. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, inc.in the U.S. and other countries. Other trademarks are the property of their respective owners.

Readership & Scope

The DMLDoc Designer's Guide is aimed at people who intend to develop HTML pages for DMLDoc-based dynamic Web projects.

If you are planning on using DMLDoc, you are probably not working on your first Web site. Experienced Web site teams usually divide up according to specialised skills. Thus, if you are involved with the HTML code, then your expertise probably lies in content or presentation. DMLDoc is designed to support that kind of specialisation. In particular, DMLDoc gives you a way of building dynamic Web sites where:

- HTML documents *never* need to contain any program code.
- Program code *never* needs to contain any HTML.

DMLDoc allows you to continue using your favoured HTML tools, and provides you with a simple but powerful of accessing any sort of content from the server. The tools described in this Guide belong specifically to the content and presentation professional. The server-side programmers and database people see DMLDoc in a different way.

All the DMLDoc Standard Edition documentation talks about HTML, as if this is the only language that DMLDoc deals with. In fact, DMLDoc is capable of handling any Standard Generalised Markup Language (SGML), and, in particular, XML.

The user is free to experiment with these sorts of applications of DMLDoc. This manual should suffice in explaining the use of the required operations and commands for both HTML and XML applications.

If you want to understand in more detail how DMLDoc deals with Web pages, you should turn to the DMLDoc Programmer's Guide.

Contents

1	**Introduction**	5
1.1	... **The Basic Ideas Behind DMLDoc**	6
1.2	... **The Template**	7
1.3	... **DML - The DMLDoc Language**	8
1.4	... **Running DMLDoc**	9
1.5	... **The DMLDoc Runtime System**	10
2	**DML Instructions**	11
2.1	... **SUB**	12
2.2	... **LOOP** & **ENDLOOP**	14
2.3	... **IF** / **IFNOT** & **ENDIF**	16
2.4	... **IFDATAFOR** / **IFNODATAFOR** & **ENDIF**	18
3	**Making it all Work**	19
3.1	... **Template Style Guides**	20
3.2	... **The Complete DMLDoc Application**	24
3.3	... **Behind the Scenes**	25
4	**Case Studies**	26
4.1	... **The 'servlet report' resource template**	27
4.2	... **HTML Forms**	32
A	**The DML Specification**	35
A.1	... **Syntax**	35
A.2	... **Operators**	39
B	**Contact Details**	41

1 Introduction

DMLDoc provides one of the the simplest ways of making the transition from static Web content to dynamic Web content. Despite its simplicity, DMLDoc gives you all of the power available through any Java server-side programming environment or data source.

This is possible because of the way in which DMLDoc integrates with Java program code and, in particular, Java Servlets. The DMLDoc Web designer need not know anything at all about server-side programming. But the programmers involved in the Web project now have a straightforward way in which to separate presentation, content and control.

Using DMLDoc, programmers can build server-side components which make use of any Java-accessible data sources. They can make their components available to any client - including Microsoft®-specific clients via the Java COM Bridge.

Thus, DMLDoc can be used to build Web sites of every level of complexity. DMLDoc applications can run on every sort of Java-enabled computer, from a laptop to a mainframe.

By separating visualisation, dynamic content and control, DMLDoc allows you to build robust and scalable Web applications quickly.

1.1 Introduction: **The Basic Ideas Behind DMLDoc**

The basic goal for DMLDoc is to separate out the way in which content is generated or found, from the way in which it is presented (normally to a Web browser). A further separation is also made - between server-side content and server-side business logic. In this way, DMLDoc builds on a triad of application component types. This triad is common to all modern and efficient development environments:

Model - The structure of the data or content which is displayed by the Web site. The Model specifies what sort of data can be displayed, but does not deal with how the data is obtained, or where it is obtained from. Models are normally determined by the database schema or server-side component framework.

View - The appearance of data in the Web site. The View is normally constructed using an HTML file, along with graphics and perhaps some JavaScript (or ECMA script) code or even Java applets. One Model may have many Views, and Views can change over time as a Web site evolves or alters its graphic style. Alternate views may be used to accomodate different languages or different client devices.

Controller - The logic that drives the dynamic Web site. The Controller deals with two different issues: firstly, it reconciles the data sources or generated data available to it. Secondly, it provides some of the workflow control, by guiding the Web customer from one page to the next. DMLDoc uses Java as its server-side controller language. Controllers are normally implemented as Java Servlets.

The DMLDoc Web designer is primarily interested in the View aspects of the application. The Model is important to the DMLDoc designer too, and this model is created in co-operation with the server-side programmers.

1.2 Introduction: **The Template**

The DMLDoc template is the key element for the DMLDoc View. The template specifies the style of the Web page, together with its user interface functionality, without having to deal with how the data to be displayed gets there, or when this or other pages should be presented to the client.

What is the template made up of? It's an HTML file. Pure and simple. This is why DMLDoc is able to work alongside any set of HTML design tools - the only rule is that the tools must comply with at least the 1.0 standard for HTML. Thus, the Web Designer is free to create or reuse any HTML document, and make use of this document within a DMLDoc application.

Where does the data come from? Obviously there must be some way to introduce dynamically-generated data into the template. From the designer's perspective, this is a two-stage process:

Stage 1 - Declare that a particular token (for example, the string `MY_DATA`) should be associated with a dynamic data source. This is done within an HTML comment, normally placed near the start of the HTML document.

Stage 2 - Use that token freely within the HTML document. At runtime, DMLDoc associates the token with a dynamic data source. That source could provide a single value, or a sequence of values. DMLDoc automatically handles sequences of any sort of data objects.

Wherever that token is found in the template, it will be replaced with whatever dynamic values have been made available. The token can occur *anywhere* in the HTML template - it can be used to provide dynamically-generated content, or even dynamically-generated HTML tags.

1.3 Introduction: **DML - The DMLDoc Language**

DML is used to reconcile the dynamic data (the Model) with the HTML code in the template (the View). There are two really important aspects to this language:

Simplicity - This is a very simple language. In fact, it only needs to recognise three operations:

- *Substitution* - associating a token with a data source.
- *Repetition* - displaying a block of HTML repeatedly.
- *Conditionality* - displaying a block of HTML conditionally.

Standardisation - All DML keywords are used within HTML comments (of the form `<!-- comment -->`). Because of this, they can be introduced into the HTML template file using any Web design tool.

Because all DML keywords are contained within HTML comments, DMLDoc template files can be viewed by any Web browser, before the template is processed by the DMLDoc server application.

This is a significant advantage for the Web designer, because DMLDoc template development can proceed in the same way as conventional or static HTML development.

The designer is free to choose any text strings to act as substitution tokens, the only restriction being that these strings cannot contain some special characters, such as <, > or = (the full list of such characters is given in Appendix A).

When the HTML template is processed by the DMLDoc application, all the DML commands are temporarily removed from the HTML source. Thus, the HTML document delivered by the DMLDoc application becomes indistinguishable (from the client's point of view) from a conventional, static HTML file.

1.4 Introduction: **Running DMLDoc**

Completed DMLDoc HTML templates must be made available to the server in order to be used. For the Standard Edition of DMLDoc, the template files must be installed on the same file system as the application server itself. (Operating systems such as Solaris, Mac OS X and Windows XP support networked file systems, and these networked file systems can be used to host the templates.) DMLDoc works with any Java Servlet-compatible application server.

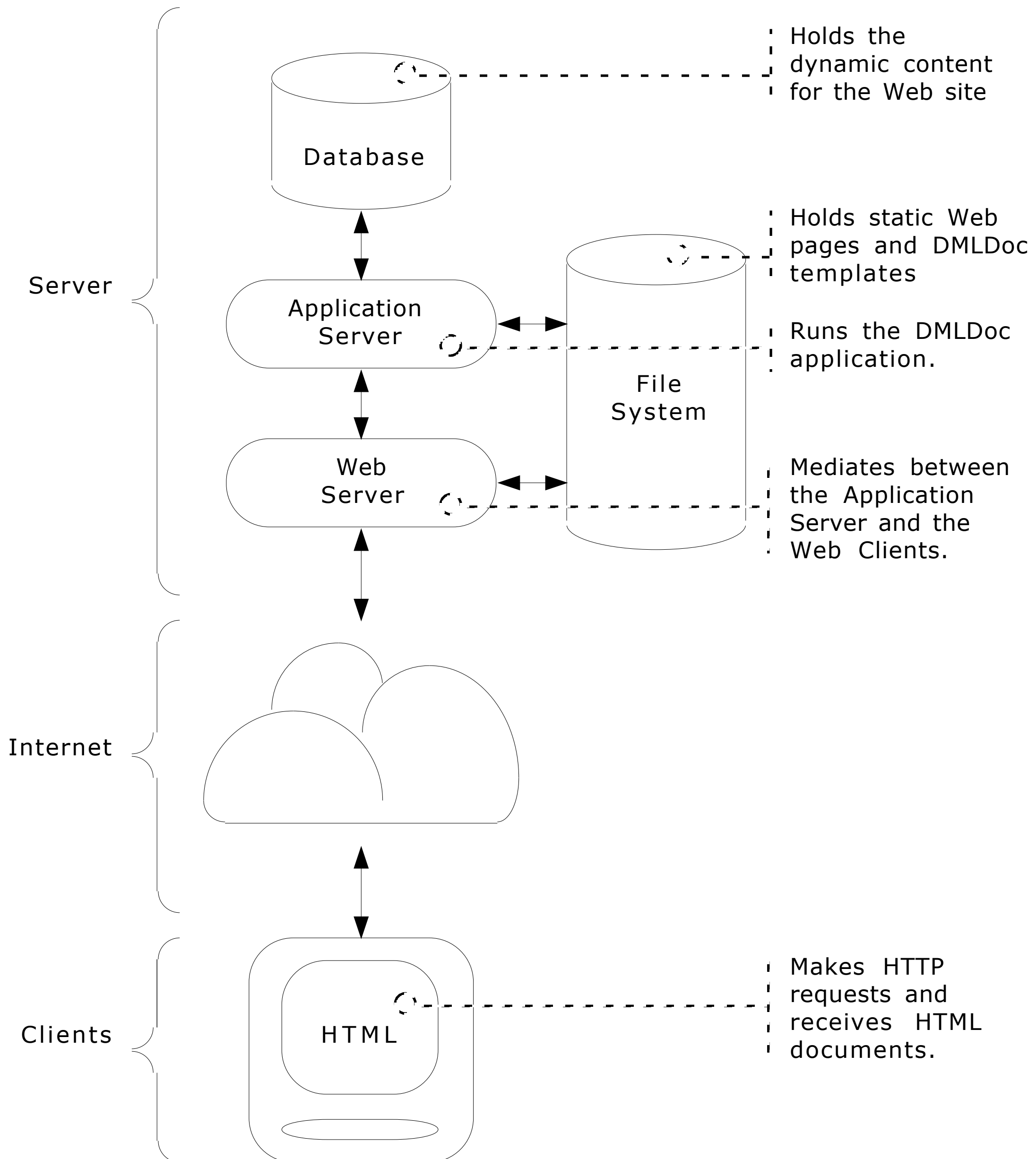
The programmer of the server-side logic must work to the same Model as the designer. If a first cut of the template is completed while the server-side logic is under development, it is possible for the programmer to examine the model that was used by the designer, and test whether the server-side logic matches this Model.

If no template is available while the server-side logic is being developed, then informal arrangements must be made to ensure that the Web designer's Model and the programmer's Model match one another. If differences occur between the two models, then the DMLDoc application will report these. At this stage in development, adjustments - either to the server-side logic or the template files - can easily be made.

At run-time, the DMLDoc application parses all the required template files when the application is first launched. Then, when Web requests are received, the appropriate server-side logic is executed, the correct template is populated with data, and returned to the Web client. The process of populating the template with data is a very efficient one.

Using this process, a Web site can be constructed from a mixture of conventional, static HTML pages, and dynamic DMLDoc documents.

1.5 Introduction: **The DMLDoc Runtime System**



2 DML Instructions

DML supports only three operations. These operations are specified using four commands, and a total of only eight keywords. Together, these commands can be used to create any imaginable Web page.

Despite this simplicity, there is some subtlety in the way in which the DML commands are used together - the following specifications and case studies should therefore be useful. The structure of this guide puts specifications before examples, in order to build a solid foundation, before examining some working code.

In the following examples, DML keywords are always shown in UPPER CASE. In practice, the keywords are not case-sensitive, but we think it is good style to stick to UPPER CASE for clarity.

The HTML comments which contain the DML commands must be placed on new lines in the HTML template file - that is, *the comment containing a command should be the only HTML element on that line*. Again, we think that this aids clarity.

In the following subsections, littoral examples - code fragments which could be typed as is - are shown in **`Courier Bold`** font. Where an example contains some text which could have any value, the given value is shown in `plain Courier`.

2.1 DML Instructions: **SUB**

The **SUB** command - short for SUBSTITUTION - is used to associate a server-side data source (we call this the *substitution source*) with a token in the HTML template file. The token is known as the *substitution target*. In summary, the **SUB** command is the way in which the template describes its Model.

The command may take the form:

```
<!-- SUB contentSource CONTENT_TARGET -->
```

At runtime, the token CONTENT_TARGET will be replaced - wherever it is found in the template - by the server-side data value or values associated with the token contentSource.

As we have already stated, the substitution target can appear anywhere in the template file. For example, it could be used as follows:

```
<!-- SUB widthSpec SUB_WIDTH -->
...
<TABLE WIDTH="SUB_WIDTH">
...
```

In this case, the DMLDoc application dynamically sets the percentage width of an HTML table.

Note that each substitution target can only be declared in one **SUB** command - otherwise there would be an ambiguity over where the content should come from. In other words, each target can gain data from only one source.

However, the substitution source can be declared in any number of **SUB** commands - so long as each **SUB** associates this source with a different target. In other words, each source can supply data to many targets.

The **SUB** command may also take the form:

```
<!-- SUB content -->
```

This version of the command conflates the identifiers for the substitution target and substitution source. In this case, the DMLDoc application will look for a source called `content`, and will look for targets with the same name.

In this case, the same example would be written:

```
<!-- SUB widthSpec -->
...
<TABLE WIDTH="widthSpec">
...
```

The effect here is exactly the same as in the previous example.

Substitution targets can be used within **LOOP** blocks and **IF** blocks. A simple *rule of substitution* is applied in all cases:

- Where the target is found outside of a **LOOP**, then the target is substituted with the first data element associated with the source.

- Where the target is found inside a **LOOP** block, then the loop block is repeated for each of the sequence of values associated with the substitution source.

Note that if a **SUB** command is issued inside a **LOOP** block or an **IF** / **IFNOT** block, then it only applies to the text within that block.

In general, **SUB** commands do not have to appear before the substitution target which they introduce. However, it is good practice to issue the **SUB** command as early in the template as possible.

2.2 DML Instructions: **LOOP** & **ENDLOOP**

The **LOOP** command introduces a block of HTML which should be rendered some number of times. Typically, **LOOP** commands are used in connection with HTML tables. The form of the command is:

```
<TABLE WIDTH="100%">
<!-- LOOP -->
    <TR>
        <TD>CONTENT_TARGET</TD>
    </TR>
<!-- ENDLOOP -->
</TABLE>
```

In this case, the rendered table will have as many rows as there were values associated with the substitution source.

Every **LOOP** command must have an associated **ENDLOOP** command. If it does not, then the DMLDoc application will report an error during the Web application's initialisation process.

In many real-world cases, the **LOOP** contains multiple substitution targets. A simple example would be:

```
<TABLE WIDTH="100%">
<!-- LOOP -->
    <TR>
        <TD>CONTENT_TARGET_COL_A</TD>
        <TD>CONTENT_TARGET_COL_B</TD>
    </TR>
<!-- ENDLOOP -->
</TABLE>
```

In a case such as this, the substitution sources may have different numbers of values associated with them. If this happens, the empty string "" is substituted to make up for any missing values once the shorter sequence is exhausted.

If no values are found for any substitution sources, or the **LOOP** contains no substitution targets, then the whole **LOOP** block is ignored, and nothing within the block is rendered.

One special run-time case exists for loops. This applies when:

- The loop contains multiple substitution targets.
- At least one of the substitution targets is associated with more than one value.
- One of the substitution targets is associated with exactly one value.

In this case, *that one value is repeatedly used on each iteration of the loop*. In the following example, CONTENT_TARGET_COL_A may be associated with the value { 50 }, while CONTENT_TARGET_COL_B is associated with the values { "one", "two", "three" } :

```
<!-- LOOP -->
    <TR>
        <TD HEIGHT="CONTENT_TARGET_COL_A">
            CONTENT_TARGET_COL_B
        </TD>
    </TR>
<!-- ENDLOOP -->
```

In a case such as this, you might expect CONTENT_TARGET_COL_A to have the same value each time round the loop: 50, whereas CONTENT_TARGET_COL_B would have a different value each time: "one", "two", "three". The above rule guarantees just that.

2.3 DML Instructions: **IF** / **IFNOT** & **ENDIF**

The **IF** command introduces a block of HTML which should be rendered either once or not at all. The form of the command is:

```
<!-- IF contentSource -->
    <TD>CONTENT_TARGET</TD>
<!-- ENDIF -->
```

In this case, if the data for `contentSource` is equal to the string "true" then the HTML code:

```
<TD>CONTENT_TARGET</TD>
```

will be rendered. If `contentSource` does not evaluate to the string "true" then nothing will be rendered.

The converse command - **IFNOT** - also exists. The form of the command is:

```
<!-- IFNOT contentSource -->
    <TD>Sorry - there is no content!</TD>
<!-- ENDIF -->
```

In this case, if `contentSource` does not evaluate to "true" then the HTML code:

```
<TD>Sorry - there is no content!</TD>
```

will be rendered.

In both cases, the end of the **IF** or **IFNOT** block must be marked with the command **ENDIF**. If the block is not marked in this way, then the DMLDoc application will report an error during the application's initialisation process.

Note also that there is *no* **ENDIFNOT** command - both **IF** and **IFNOT** work with **ENDIF.**

For the above examples, the token `contentSource` is known as the *if-determiner* for the command. An if-determiner is obligatory for both **IF** and **IFNOT**.

The if-determiner behaves a lot like the substitution source in a **SUB** command. Thus:

- The same if-determiner can be used any number of times, in any number of **IF** and **IFNOT** commands, in the same template file.

- If the **IF** or **IFNOT** command occurs inside a **LOOP** block, then the data associated with it is accessed one value at a time, and the **IF** or **IFNOT** command is evaluated accordingly. Thus, the loop can actually contain different sorts of content, or different HTML code, each time around.

- An if-determiner can also serve as a substitution source. So, in the example:

```
<!-- SUB contentSource CONTENT_TARGET -->
<!-- IF contentSource -->
   CONTENT_TARGET
<!-- ENDIF -->
```

either `true` is rendered, or nothing is rendered.

2.4 DML Instructions: **IFDATAFOR** / **IFNODATAFOR** & **ENDIF**

Similar to the **IF** command, the **IFDATAFOR** command introduces a block of HTML which should be rendered either once or not at all. The form of the command is:

```
<!-- IFDATAFOR contentSource -->
    <TD>CONTENT_TARGET</TD>
<!-- ENDIF -->
```

In this case, if some data is held for contentSource then the HTML code:

```
<TD>CONTENT_TARGET</TD>
```

will be rendered.

The converse command - **IFNODATAFOR** - also exists. The form of the command is:

```
<!-- IFNODATAFOR contentSource -->
    <TD>Sorry - there is no content!</TD>
<!-- ENDIF -->
```

In this case, if no data is held for contentSource then the HTML code:

```
<TD>Sorry - there is no content!</TD>
```

will be rendered. In both cases, the end of the **IFDATAFOR** or **IFNODATAFOR** block must be marked with the command **ENDIF**. If the block is not marked in this way, then the DMLDoc application will report an error during the application's initialisation process.

The **IFDATAFOR** and **IFNODATAFOR** commands are unaffected by being placed within a **LOOP** block - they work the same way irrespective of the number of times they are evaluated as the loop is executed.

3 Making it all Work

This section deals with the practicalities of developing real-world DMLDoc applications. The points raised here are relevant only to the Web designer - the tasks for the database administrator and server-side programmer are covered in the other Guides.

The Web designer plays the pivotal role in a DMLDoc project - a key aspect of DMLDoc development is that the design and content of the Web pages should define the form of the rest of the application. To this end, we have found the following strategies useful:

Develop the templates first - As with all user-facing software applications, the best place to start the specification for the whole application is often with the user's view of data. Thus, with DMLDoc applications, you should aim to build the templates first, then develop the server-side logic.

Start with a concrete example - Even with the best HTML tools, creating a good page layout is usually a complex task. When developing for DMLDoc, begin by creating a static page, complete with some representative content. Then, when you are happy with the layout, begin the process of replacing that representative content with substitution targets, together with **`SUB`**, **`LOOP`** and **`IF`** commands.

Share a data dictionary - One of the most difficult tasks in a DMLDoc application is synchronising the use of substitution source names between templates and server-side logic. In common with most dynamic Web tools, the Standard Edition of DMLDoc does not have its own shared Data Dictionary; future versions of DMLDoc may do.

In the mean time, it's a good idea to use some *ad hoc* synchronisation method - such as copying and pasting **`SUB`** commands from the HTML templates to the Java servlet source code.

3.1 Making it all Work: **Template Style Guides**

None of the following guidelines are compulsory, but we have found all of them very useful in building real DMLDoc applications:

Place all SUB commands at the start of the template file - One of the nice features of the **SUB** command is that it can be used to indicate to the developers that a template file requires some particular data source. The best way to take advantage of this is to place all **SUB** commands at the start of the template, thus:

```
<HTML>
<!-- file: tpl_music -->
<!-- Used to provide a 'jukebox' style of music selection.  -->
<!-- Copyright (c) 1999, 2000, 2002 The Open Math Company Limited -->
<!-- author: Bruno Beloff <bruno.beloff@opmath.com> -->
<!-- created: 13 Dec 1999 last updated: 20 May 2002 -->

<!-- SUB base_url SUB_BASE_URL -->
<!-- SUB clsid SUB_UID -->
<!-- SUB music_url SUB_MUSIC_URL -->
<!-- SUB music SUB_MUSIC -->
<!-- SUB volume SUB_VOLUME -->

<HEAD>
 <TITLE>totallybrighton: music selector</TITLE>
 <BASE HREF="SUB_BASE_URL">
</HEAD>

<BODY BGCOLOR="#FFCE00" LINK="#OOADCE" VLINK="#000000">
...
</BODY>
</HTML>
```

Declare if-determiners in SUB commands - The if-determiner - the data source used to decide whether to render an **IF..** block - can optionally be declared as the substitution source in a **SUB** command. We think that if-determiners should *always* be declared in **SUB** commands, whether this is necessary or not.

Doing this allows the use of the if-determiner to be declared at the start of the file:

```
<HTML>
<!-- file: tpl_search_full.html -->
<!-- Used to enter the search criteria.  -->
<!-- Copyright (c) 2000, 2002 The Open Math Company Limited -->
<!-- author: Bruno Beloff <bruno.beloff@opmath.com> -->
<!-- created: 09 Feb 2000 last updated: 20 May 2002 -->

<!-- SUB base_url SUB_BASE_URL -->
<!-- SUB s_mode_visible -->
<!-- SUB s_mode_options SUB_S_MODE_OPTIONS -->

<HEAD>
<TITLE>totallybrighton: search criteria</TITLE>
<BASE HREF="SUB_BASE_URL">
</HEAD>

<BODY BGCOLOR="#FFCE00" LINK="#OOADCE" VLINK="#000000">
...
<!-- IF s_mode_visible -->
<TD NOWRAP ALIGN="RIGHT">Show...</TD>
<TD ALIGN="LEFT"><SELECT NAME="s_mode"> SUB_S_MODE_OPTIONS </SELECT></TD>
<!-- ENDIF -->
...
</BODY>
</HTML>
```

Watch out for <BASE HREF=".."> - When you are developing a static Web page, testing the page is pretty easy - just open your HTML file with your chosen Web browser. Developing for DMLDoc is a little bit more complicated, because the system used to host the DMLDoc application is likely to be different from the one on which you develop the template file.

In practice this means that - because the runtime location of your template is different from the development location - relative links will not work. The simplest solution to this problem is to include a <BASE HREF=".."> tag in the head of you template. For example, you could have:

```
<HEAD>
<TITLE>totallybrighton: search criteria</TITLE>
<BASE HREF="http://my.development.server/templates/">
</HEAD>

<BODY BGCOLOR="#FFCE00" LINK="#OOADCE" VLINK="#000000">
<IMG SRC="images/logo.gif" BORDER="0">
...
```

and then change the BASE HREF before deployment.

The examples on pages 20 and 21 show a more sophisticated technique: the value of the BASE HREF is set by a **SUB** command. This is the ideal solution where more than one runtime system is being used, but may cause problems where the template is to be viewed directly by the developer.

A simple edit operation between development and deployment may still be necessary - we found that the best technique is to comment out the BASE HREF line until the template is ready for deployment.

Remember where your code is verified - If you are familiar with JavaScript, or any scripting system, then you will be familiar with a fundamental problem of interpreted languages: errors in your source code do not normally show up until the code is executed. Thus, simple syntax errors may not turn up until some way into your testing procedure.

DMLDoc is different in this respect: all the DMLDoc templates for an application are accessed when the application first starts up. At this stage, DMLDoc reads and verifies all the text of the templates. One benefit of this approach is that all syntax errors are detected and reported before the DMLDoc application attempts to run. An example error message is:

```
com.set_i.demo.jswdk.UploadServlet.init(ServletConfig): failed:
[DMLDocBuildException: com.opmath.util.dmldoc.DMLDocBlock.parseLine():
badly-formed SUB on line 16: <!-- SUB -->] at:[07 June 2001 12:18:57
GMT+01:00]
```

For the Standard Edition of DMLDoc, syntax errors are reported to the error channel of the application server that is hosting the DMLDoc application.

Future versions of DMLDoc could carry a separate verifier, which can determine whether a DMLDoc template is well-formed, and can be run at any time by the Web designer.

In the mean time, there is no substitute for careful composition of DMLDoc commands. Remember that, currently, the syntax-checking procedure cannot verify whether the substitution sources described in the template actually match those used in the server-side Java code.

3.2 Making it all Work: **The Complete DMLDoc Application**

Most of this Guide deals with the specifics of composing DMLDoc templates. This section takes a break from that, to review the complete DMLDoc development process:

Develop the application templates - The subject of this manual. The application templates include those specifically for your application, together with customised versions of the DMLDoc *resource templates*. Details of the resource templates are given in Section 4 of this Guide.

Install the DMLDoc package - The DMLDoc package must be made available both to the application server, and to the Java development tool or the javac compiler - this process is explained in the Installation Guide.

Develop the application's business logic - Developing a DMLDoc application is very like developing a standard Java Servlet application, only *much* easier - The Programmer's Guide steps through some example applications.

Develop the data access code - DMLDoc applications can access data from any JDBC data source, from persistent Enterprise Java Beans (EJBs), or any other Java-accessible system, for example a Java Messaging System - again, the Programmer's Guide illustrates how this is done.

Install and run the application - DMLDoc applications require a combination of Web Server and Java Servlet container, but it is not fussy about which combination of products is used. The Installation Guide shows how this installation can take place for several combinations of operating system and application server. Details of how to run the chosen application server will be covered by that products documentation.

3.3 Making it all Work: **Behind the Scenes**

As with a great deal of technology, it is not necessary to understand the inner workings of DMLDoc, but it is maybe easier to learn if you do. So, if it helps, this is how the run-time system operates:

Initialisation - This happens when the DMLDoc application is loaded by the Servlet container. At this stage, the template files are read. For each template, two highly-structured objects are built: a Document Object Model (the DOM) representing the template's View, and a data table (a ComparableTable), representing the template's Model.

Reporting initialisation failures - Exceptions are raised at this stage if the templates may be missing or misnamed, or the templates contain badly-formed DMLDoc commands. Such errors are reported to the application server error channel. The DMLDoc application should halt if any error occurs at this stage.

Web request - When the application receives an HTTP request , the application business logic accesses the appropriate data, registers this data with the appropriate ComparableTable, then requests that DMLDoc generate the response.

Instantiating the DOM - DMLDoc generates the correct Web page response by merging the ComparableTable data with its DOM for that page. The data merging is governed by the **`SUB`**, **`IF..`**, and **`LOOP`** commands.

Reporting request failures - Exceptions are raised during the process of responding to a Web request if the Web client has called for an unknown function, or the data generated by the business logic is incomplete. In either case, DMLDoc can generate a customised Web page in response, and continue running.

4 Case Studies

The following case studies examine two templates in detail.

The first case study examines a resource belonging to the DMLDoc package itself. The resource file serves as templates for generic exception handling. It has been composed in such a way as to provide minimum graphic value and maximum information. DMLDoc developers can modifying the template to conform to their Web site's graphic style.

The template is held within the `utilities-v011.jar` file, which contains all of the DMLDoc runtime system. Within the JAR file, the location of the template is:

```
com/opmath/util/dmldoc/tpl_ServletReport.html
```

To access these files, you should unpack the JAR file. Your Java tools will most likely provide options to do this, alternatively you can issue the command:

```
jar -xvf utilities-v011.jar
```

This will create the appropriate directories or folders, which will contain these templates. The DMLDoc system can be run using either the JAR file, or these unpacked directories.

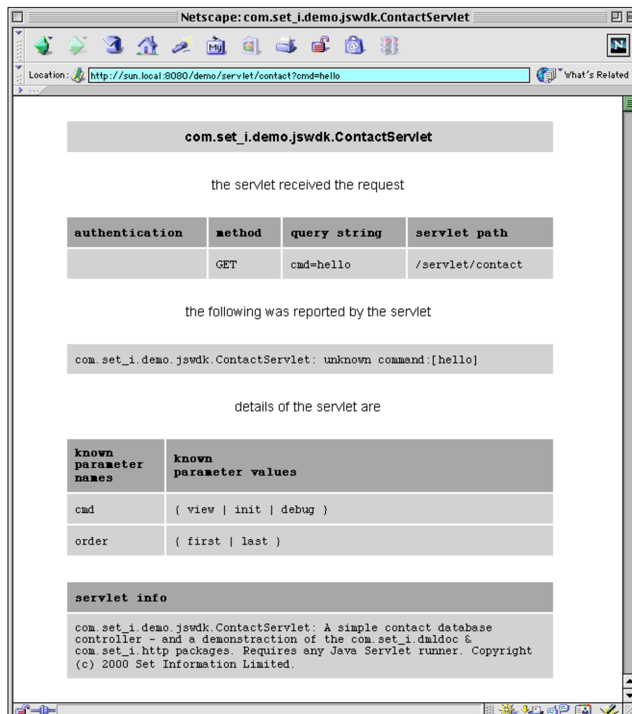
The second example concerns the use of HTML forms that have been generated using DMLDoc.

4.1 Case Studies: **The 'servlet report' resource template**

This template is used by DMLDoc when an exceptional condition is raised during the processing of a Web request. Such a condition occurs when, for example:

- The application received an HTTP parameter (included with the URL that triggered the servlet) that it did not recognise.
- The business logic that was triggered by the request did something that failed unexpectedly.
- The data given to DMLDoc in order to generate a response did not match the Model described by the template.

A typical response, using the standard template, looks like this:



The standard HTML template contains the following HTML code (the full version is available in the `tpl_ServletReport.html` file).

Firstly, the Model defined by the template is expressed by the **SUB** commands:

```
<!-- SUB class SUB_CLASS -->

<!-- SUB auth SUB_AUTH -->
<!-- SUB method SUB_METHOD -->
<!-- SUB query SUB_QUERY -->
<!-- SUB servlet SUB_SERVLET -->

<!-- SUB paramNames SUB_PARAM_NAMES -->
<!-- SUB paramValues SUB_PARAM_VALUES -->

<!-- SUB message SUB_MESSAGE -->
<!-- SUB servletInfo SUB_SERVLET_INFO -->
```

The Model includes most of the environmental information available to a Servlet. In addition, there is the class of the servlet, a message specific to the situation, and the standard servlet info report.

The HEAD of the HTML document does not carry a BASE tag, since the file does not require any locatable resources. It does, however, obtain the TITLE of the document dynamically:

```
<HEAD>
<TITLE>SUB_CLASS</TITLE>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
</HEAD>
```

The class information is repeated at the top of the document:

```
<TR ALIGN="CENTER" BGCOLOR="#CCCCCC">
    <TD>
        <FONT FACE="Arial, Helvetica, sans-serif">
            <B>SUB_CLASS</B>
        </FONT>
    </TD>
</TR>
```

Next, a nested table displays the Servlet environment information. The first row contains only static content, whereas the second row contains dynamic content:

```
<TR ALIGN="LEFT">
    <TD BGCOLOR="#CCCCCC">
        <CODE>SUB_AUTH</CODE> 
    </TD>
    <TD BGCOLOR="#CCCCCC">
        <CODE>SUB_METHOD</CODE> 
    </TD>
    <TD BGCOLOR="#CCCCCC">
        <CODE>SUB_QUERY</CODE> 
    </TD>
    <TD BGCOLOR="#CCCCCC">
        <CODE>SUB_SERVLET</CODE> 
    </TD>
</TR>
```

A small trick is used here to overcome a limitation of some browsers: if any of these substitution targets carry no data, then the cell might not be rendered at all. Here, the ` ` character makes sure that some text is always present in the cell, and therefore the correct `BGCOLOR` of the cell will be rendered.

The following segment shows the whole table used to display the parameter names and values known to the Servlet.

HTTP GET and PUT parameters are managed by a number of DMLDoc classes, and can be accessed easily. This management system keeps a list of known parameters, and the possible values each named parameter can legally have.

Here, a **LOOP** command is used to repeatedly generate rows for the table, with one row for each name / value pair. The template does not have to define how many times the loop should be repeated; this is determined automatically by DMLDoc:

```
<TABLE WIDTH="100%" BORDER="0" CELLSPACING="2" CELLPADDING="8">
    <TR ALIGN="LEFT">
        <TD BGCOLOR="#999999" WIDTH="20%">
            <CODE><B>known<BR>parameter names</B></CODE>
        </TD>
        <TD BGCOLOR="#999999" WIDTH="80%">
            <CODE><B>known<BR>parameter values</B></CODE>
        </TD>
    </TR>

    <!-- LOOP -->
    <TR ALIGN="LEFT">
        <TD BGCOLOR="#CCCCCC" WIDTH="20%">
            <CODE>SUB_PARAM_NAMES</CODE>
        </TD>
        <TD BGCOLOR="#CCCCCC" WIDTH="80%">
            <CODE>SUB_PARAM_VALUES</CODE>
        </TD>
    </TR>
    <!-- ENDLOOP -->

</TABLE>
```

Take care when editing this template file: remember that, at runtime, the `DisplayTable` for this template is populated automatically. Thus, if you remove any existing **SUB** commands, this automatic process is likely to fail.

But the fact that the template may contain **SUB** commands that you do not need should be of no concern: just because a **SUB** is present in the HTML file does not force any action onto the designer - in other words, there is no requirement to make use of substitution target, just because you have declared it.

Conversely, if a DMLDoc application component attempts to register some data against a substitution source, then that source *must* have been declared in the template.

To summarise these rules:

- If some intended substitution target is used in the template, but not declared in a **SUB** command, then the substitution target string will be rendered *as is*. Assuming that no DMLDoc application component attempts to register data against this target, no substitution will take place, and no error will be reported.

- If a substitution target is declared in a **SUB** command, but not used in the template, then no substitution will take place, and no error will be reported - irrespective of whether the DMLDoc application component registers data against its substitution source.

- If a DMLDoc application component attempts to register data against a substitution source, and the source was not defined in a **SUB** command, a runtime exception will be raised.

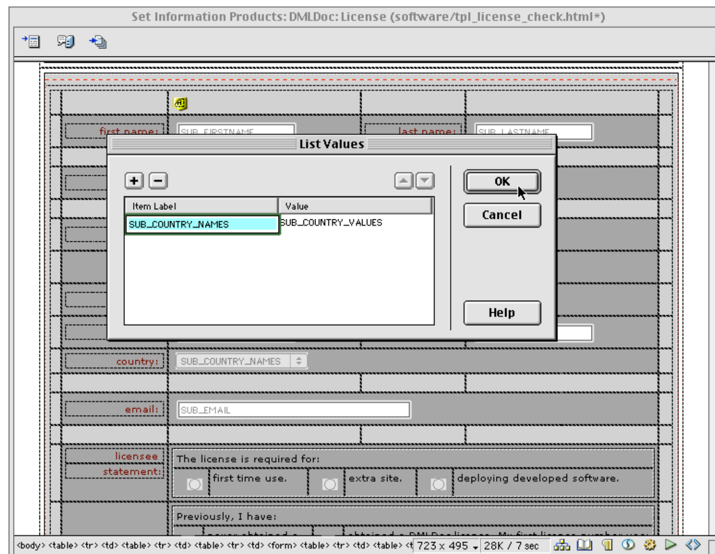
4.2 Case Studies: **HTML Forms**

DML commands are sufficiently flexible that no special operators are required to create HTML forms. However, certain techniques are worth studying in order to construct form elements effectively. Here we look at two specific problems: SELECT structures and Radio buttons.

SELECT structure look like:

```
<SELECT NAME="country">
        <OPTION VALUE="-1" SELECTED> s e l e c t </OPTION>
        <OPTION VALUE="ALBANIA">Albania </OPTION>
        <OPTION VALUE="ALGERIA">Algeria </OPTION>
        ...
</SELECT>
```

Using DML, it is possible to have all of the OPTION elements generated dynamically. Using Macromedia Dreamweaver, it is possible to describe a SELECT structure as part of a form, whose only OPTION element is specified by substitution targets:



This will generate the following HTML:

```
<SELECT NAME="country">
    <OPTION VALUE="SUB_COUNTRY_VALUES">
        SUB_COUNTRY_NAMES
    </OPTION>
</SELECT>
```

Now add the appropriate SUB and LOOP commands, and the data source used to determine which entry is selected:

```
<!-- SUB countryNames SUB_COUNTRY_NAMES -->
<!-- SUB countryValues SUB_COUNTRY_VALUES -->
<!-- SUB countrySelect SUB_COUNTRY_SELECTED -->
...
<SELECT NAME="country">
    <!-- LOOP -->
    <OPTION VALUE="SUB_COUNTRY_VALUES" SUB_COUNTRY_SELECTED>
        SUB_COUNTRY_NAMES
    </OPTION>
    <!-- ENDLOOP -->
</SELECT>
```

When the template is instantiated by the DMLDoc server-side application, a complete SELECT structure will be built, without the server-side application having to explicitly introduce any HTML tags.

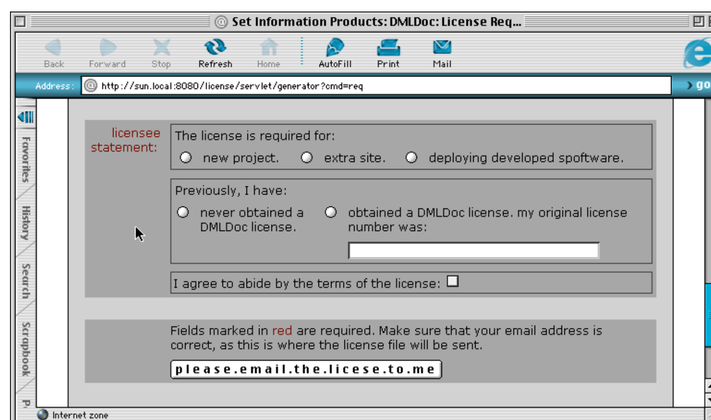
The server-side part of such an application is described in the DMLDoc Programmer's Guide.

Radio buttons work in much the same way, the only problem here is that the text accompanying the buttons is not formally part of the button structure.

Thus, the Web designer should take care to ensure that the HTML structure encompassing the button group can be looped over successfully. An example is:

```
</TR>
    <!-- LOOP -->
    <TD ALIGN="RIGHT" VALIGN="TOP">
        <INPUT TYPE="radio" NAME="rqmt"
               VALUE="SUB_RQMT_VALUES" SUB_RQMT_CHECKED>
    </TD>
    <TD ALIGN="LEFT" VALIGN="TOP">
        SUB_RQMT_TEXTS
    </TD>
    <!-- ENDLOOP -->
</TR>
```

Here, `SUB_RQMT_VALUES` names the button within the `rqmt` radio button group. The substitution target `SUB_RQMT_CHECKED` determines whether the button is checked by default, and `SUB_RQMT_TEXTS` is the text accompanying each button. The result looks like:



A The DML Language Specification

The conventions used here attempt to match those used in The C Programming Language, Second Edition [B.W. Kernighan & D.M. Ritchie, 1988]. Thus, literal text is in `Courier`, syntactic elements are *italicised*, optionals are marked *opt*. Expansions of syntactic elements are in the form:

syntactic-element:
one possible expansion
an alternative expansion

A.1 The DML Specification: **Syntax**

A.1.1 Tokens

There are two classes of tokens: identifiers and keywords. Tokens are separated by white space (the space character, tab, newline, carriage return, form feed) and by the characters '#', '/', '|', '%', ',', '.', ';', ':', ''', '?', '=', '&', '"', '<' and '>'. Tokens may contain any characters except these separators. At least one separator must be present between tokens.

A.1.2 Identifiers

An identifier can be any token. In other words, it can be composed of any characters except separators. Both internal identifiers - for example, substitution targets - and external identifiers - for example, substitution sources - can be of any length. All characters are significant, up to any restrictions placed on instances of `java.lang.String`.

A.1.3 Keywords

Keywords are *not* case-sensitive. The following are keywords:

```
SUB LOOP ENDLOOP IF IFNOT IFDATAFOR IFNODATAFOR ENDIF
```

A.1.4 Meaning of Identifiers

Identifiers are only used for three purposes in the DMLDoc language: substitution sources, substitution targets and if-determiners.

Together, the substitution sources and if-determiners contained in a DMLDoc template are said to make up the template's Model.

A.1.4.1 Substitution Sources

These are external identifiers, in that they refer to data that is managed by an external system - in this case, the DMLDoc server-side application.

At any one time, a substitution source may be associated by the DMLDoc application with zero or more data elements. These data elements must always implement the `java.lang.Comparable` interface.

When a DMLDoc template is instantiated, the DMLDoc system attempts to render all of these data elements according to the DMLDoc commands in the template.

A.1.4.2 Substitution Targets

These are internal identifiers, in that they are associated with a single substitution source within the template.

Substitution targets can occur anywhere within a template, so long as the separator rules are upheld. There is only one exception to this rule: substitution targets do not operate within DMLDoc commands (of course,

they can be declared within the **SUB** command).

When a DMLDoc template is instantiated, the DMLDoc system replaces substitution targets with the appropriate data associated with the relevant substitution source.

Because the final output of the association process is of class `java.lang.String`, the Java method `toString()` is first applied to each value.

A.1.4.3 If-Determiners

These are external identifiers, in that they refer to data that is managed by an external system - again, this is the DMLDoc server-side application.

As with a substitution source, at any one time an if-determiner may be associated by the DMLDoc application with zero or more data elements. These data elements must always implement the `java.lang.Comparable` interface.

When a DMLDoc template is instantiated, the if-determiner is used to determine whether or not an `IF` or `IFNOT` block should be rendered.

If-determiners are essentially boolean in nature. Unfortunately - for very good reason - Java objects of class `Boolean` do not implement the `java.lang.Comparable` interface. Because of this, the following convention is observed: if-determiners can be associated with an instance of any Class that implements `java.lang.Comparable`. The boolean evaluation is performed by comparing the result of performing the Java method `toString()` on the data element with the `java.lang.String "true"`.

A.1.5 Commands

Commands are the only way in which DMLDoc operations can be introduced into a template. Commands take the form:

command:

<!-- *statement* -->

The characters surrounding the statement were chosen because they match the syntax of an SGML comment. However, unlike SGML comments, DMLDoc commands must be the first element on the line of a template file.

A.1.6 Statements

Statements are used to modify the behaviour of DMLDoc with respect to the plain text blocks and substitution targets in the template. Statements take the form:

statement:

zero-arity_operator
unary_operator identifier
binary_operator identifier $identifier_{opt}$

Each DMLDoc keyword falls into only one category of *operator*.

A.1.7 Blocks

Blocks mark out a section of a template whose text should be treated as one unit by one DMLDoc command. Blocks can be nested to any depth. A block takes the form:

block:

block-command text block-closing-command
block-command $text_{opt}$ *block* $text_{opt}$ *block-closing-command*

A *block-command* or *block-closing-command* is simply a DMLDoc command that relates to a block. Every *block-command* has two functions: it marks the start of the block text and tells DMLDoc how to treat the block's *text*. The *block-closing-command* only acts as a syntactic marker for the end of the block.

A.1.8 **Text**

Text includes any arrangement of characters. The only restriction on the arrangement is that the block delimiters - *block-command* and *block-closing-command* - must be separated by separator characters, if they are present. It is assumed that the text is parsed by an external system (such as a Web browser).

A.2 The DML Specification: **Operators**

The following summarises the DMLDoc operators. The use of each operator is covered in more detail in Section 2 - DMLDoc Instructions.

A.2.1 **SUB** *binary_operator*

Declaration. Associates a substitution source identifier (the first argument) with a substitution target identifier (the second argument). If only one argument is present, it serves as both the identifiers.

A.2.2 **IF** *unary_operator* *block-command*

Introduces conditional block. The block introduced by IF is rendered if and only if the the if-determiner (the only argument) has the value "true".

A.2.3 **IFNOT** *unary_operator* *block-command*

Introduces conditional block. The block introduced by IFNOT is rendered if and only if the the if-determiner does not have the value "true".

A.2.4 **IFDATAFOR** *unary_operator* *block-command*

Introduces conditional block. The block introduced by IFDATAFOR is rendered if and only if the the if-determiner has data associated with it.

A.2.5 **IFNODATAFOR** *unary_operator* *block-command*

Introduces conditional block. The block introduced by IFNODATAFOR is rendered if and only if the the if-determiner does not have data associated with it.

A.2.6 **LOOP** *zero-arity_operator* *block-command*

Introduces conditional block. The block introduced by LOOP is rendered as many times as there are values for any substitution targets contained within the LOOP block. If there are no substitution targets, or the substitution targets have no values associated with them, then the LOOP block will not be rendered.

A.2.7 **ENDIF** *zero-arity_operator* *block-closing-command*

Marks the end of a block introduced by IF, IFNOT, IFDATAFOR or IFNODATAFOR.

A.2.8 **ENDLOOP** *zero-arity_operator* *block-closing-command*

Marks the end of a block introduced by LOOP.

B Contact Details

DMLDoc is developed and distributed by:

The Open Math Company Limited

7 Wyndham Street
Brighton
East Sussex BN2 1AF
United Kingdom

Tel: +44 1273 680 806

Email: contact@opmath.com
Web: http://www.opmath.com/

The DMLDoc classes resources and examples are distributed strictly under the terms of the GNU Lesser General Public License (Lesser GPL). The license is included with the source and binary distributions, and is also available at:

http://www.gnu.org/licenses/lgpl.html#SEC1

Both source and binary distributions are available for free download, with no formal support arrangements. Alternative levels of support are available from The Open Math Company.

For all users, *The Open Math Company is very eager to hear from users of DMLDoc* - we welcome bug reports and suggestions for new features. We will do our best to help with installation or programming problems. A FAQ for DMLDoc designers and programmers will be available shortly.