

DMLDoc: Programmer's Guide

P₁

dmldoc_std_01

Software Release: DMLDoc Standard Edition 01

Documentation Release: 01

S e t I n f o r m a t i o n


<http://www.set-i.com/products/>

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 - Describes the application of the DML language to the task of creating templates for dynamically-generated HTML pages. Essential reading for anyone creating Web pages for DMLDoc projects. Assumes some knowledge of HTML, and no knowledge of Java or other programming languages.

DMLDoc: Programmers Guide - this document.

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.set-i.com/products/dmldoc/1.1/docs/api/>.

DMLDoc: Technology Guide - Discusses how the DMLDoc packages can be extended in order to change or extend the DMLDoc data structures and behaviour. The guide describes the structures used by DMLDoc in detail, and gives pointers to the methods and fields detailed in the DMLDoc JavaDoc documentation. Assumes advanced Java programming knowledge.

DMLDoc White Papers & Roadmap - A number of documents covering the philosophy behind DMLDoc and future plans for products.

All documents in this catalogue are Copyright © 2000 Set Information 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 Programmer's Guide is aimed at people who intend to develop server-side code 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. 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 Java IDE, such as IBM VisualAge for Java, Symantec Visual Café, or Inprise JBuilder; you can even use DMLDoc with the 'raw' JDK.

For the server-side programmer, DMLDoc is presented as a collection of Java packages, complete with published APIs. *To get the best out of this API, it is well worth reading this manual* - DMLDoc acts as a framework for server-side Java programming, and the APIs' JavaDocs alone will not give you the whole story about this framework.

The APIs described in this Guide belong specifically to the server-side use of DMLDoc. Content providers see DMLDoc in a different way; they should refer to the Web Designer's Guide.

An Important Note ...

When the Guide talks about, for example, the `dmldoc` package, it is referring to `com.set_i.dmldoc`. Note that the package name uses `set_i` [underscore], while the Web site uses `set-i` [hyphen]. This is because the rules about URLs clash with Java syntax.

Contents

1	Introduction	6
1.1	... The Basic Ideas Behind DMLDoc	7
1.2	... Servlets & JSP	8
1.3	... Data Sources	9
1.4	... Developing with DMLDoc	10
1.5	... Running DMLDoc	11
1.6	... The DMLDoc Runtime System	12
2	The dmldoc API	13
2.1	... DMLDoc	14
2.1.1 Creating a DMLDoc	15
2.1.2 Interrogating a DMLDoc	16
2.1.3 Generating a Presentation	17
2.2	... DisplayTable	19
2.2.1 Inserting Data	20
2.2.2 Registering Data	21
2.2.3 Sorting	22
2.2.4 asDMLDoc	23
2.2.5 Constructing a Model	24
2.2.6 DisplayList Classes	25
2.3	... Template Resources	26
3	The http API	27
3.1	... HttpServlet	28
3.1.1 Indexing Parameters	29
3.1.2 doGet & doGetDispatch	31
3.1.3 Reporting Exceptions	33
3.1.4 setConfig	34
3.2	... HTML Form Elements	36
3.2.1 Select and Countries	37
3.2.2 Radio	40
3.3	... HTTP MultiPart Forms	42

4	DMLDoc & JDBC	45
4.1	... SQL Queries & ResultSet	47
4.2	... DisplayTable & ResultSet	48
5	DMLDoc & EJBs	49
5.1	... Accessing EJBs	50
5.2	... DisplayTable & EJBs	52
6	Making it all Work	53
6.1	... DMLDoc Runtime Behaviour	54
6.2	... Style Guides	57
7	Case Studies	60
7.1	... JDBC Contact Book	61
7.2	... EJB Contact Book	64
A	References	68
A.1	... Software	68
A.2	... Further Reading	69
B	DMLDoc JavaDocs	70
C	Contact Details	71

1 Introduction

DMLDoc is one of the most flexible and scalable ways in which you can create Internet front-end components for Java-based, server-side systems. DMLDoc is squarely aimed at 'serious' Web development projects. The software described here suits this role because of two crucial factors:

- *Scalability* - DMLDoc was designed to minimise the processing required at an important hot-spot: the point where the response to a client request is assembled. Added to this, the framework introduced by DMLDoc is a good match for the threadsafe requirements of Java servlets.
- *Maintainability* - DMLDoc makes a clean break between presentation, content, and control. Added to this, DMLDoc never requires that the presentation code such as HTML should be broken up into pieces - complete HTML documents can be used as templates for dynamic generation of pages, leading to much more manageable applications.

The maintainability issue is a vital one: all serious Web sites undergo continuous change. It is wrong to assume that all change can be supported through dynamic content - presentation styles and business logic can change too, as can the data sources themselves. DMLDoc encourages a maintainable server-side coding style, and banishes presentation issues from the code.

Banishing presentation from the server-side code itself grants a good deal of flexibility: precisely the same server-side code can be used to present data in a variety of styles or languages, and to a variety of targets, such as HTML, XML, WAP and plain text.

Surprisingly, then, DMLDoc is also one of the the simplest ways of making the transition from static Web content to dynamic Web content...

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. 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 effective development environments:

Model - The structure of the data or content which can be displayed by the system. The Model specifies what sort of data can be displayed, but does not deal with how the data is obtained, where it is obtained from, or what form the display will take.

View - The appearance of data in a Web site, or elsewhere. The View is often constructed using an HTML file, along with graphics and perhaps also 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.

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. DMLDoc uses Java as its server-side controller language.

We take the Model to be the foundation stone of the application design and development process, because it is the Model - for each form or report - that is most easily distilled from the application's requirements.

DMLDoc eagerly maintains a separation between the development of Views and the development of server-side logic and data sources. The Model is the one shared element across the development tasks. DMLDoc provides some facilities for helping with this sharing, and is rapidly acquiring more.

1.2 Introduction: **Servlets & JSP**

The dynamic Web began with the Common Gateway Interface (CGI). CGI allows the task of responding to an HTTP request to be handed off by the Web server to an external program or script. CGI also defines the way in which the script can discover the attributes of the HTTP request, together with its extended URL parameters.

Java servlets essentially reproduce the CGI arrangement, allowing Java code to service HTTP requests in much the same way. The arguments in favour of using Java in this role are very strong: Java is robust, secure, relatively easy to optimise, and offers the best opportunities for integration with the universe of back-end systems.

But there is a weakness: as defined and exhibited by Sun, Java servlets do not offer any ready means of separating presentation code - such as HTML - from business logic. Java Server Page (JSP) technology has been offered as a solution. In some respects superior to servlets, JSPs allow references to server-side Java components to be included in amongst the template.

But both Java approaches suffer from the same problem: client presentation logic is mingled with server-side business logic. The application development process cannot therefore proceed effectively, and application maintenance is almost impossible. A number of attempts have been made to define guidelines for the use of JSPs, which should keep logic and presentation apart. Unfortunately, these guidelines are slippery slopes, down which development teams usually slide back into chaos.

DMLDoc introduces an ultra-simple, data-driven language that presentation developers can use to define dynamically-sourced data. In so doing, it creates useful barriers between Model, View and Controller. DMLDoc works alongside Java servlets, because these represent the more efficient and flexible option. However, the server-side programmer need not - and should not - become involved with presentation details.

1.3 Introduction: **Data Sources**

DMLDoc provides a collection of extensible classes that support the reconciliation of server-side data with client presentation.

DMLDoc does not generate or specify data sources (such as database tables) directly. Instead, it gives the practical support to the server-side programmer, for a number of different kinds of server-side data. These kinds of data are:

- *Relational databases* - It is assumed that relational databases will be accessed via a JDBC connection. DMLDoc provides robust support for handling the `ResultSet` objects returned via JDBC.
- *Enterprise Java Beans (EJBs)* - Despite the computational cost of supporting EJBs, these components represent a very good fit with the overall Java servlet architecture. EJBs demand a completely different view of data in comparison with JDBC `ResultSet` objects; DMLDoc also provides support for this data model.
- *Synthesised data* - Some server-side systems - including scientific and statistical applications - generate new data points on the basis of specific request parameters. This arrangement is also supported by DMLDoc.

In all cases, it is crucial to maintain consistency, as data is extracted or generated, and then handed to the presentation system. Added to this, inconsistencies and exceptions must be caught and handled at the right points in the application. This manual shows how DMLDoc helps to manage these cases, and steps through example application code.

1.4 Introduction: **Developing with DMLDoc**

DMLDoc is presented as a collection of packages, each of which contains a fair number of classes. We took this route, because it is the most efficient and most flexible approach. If it looks daunting, that's an illusion.

In order to use DMLDoc, its packages have to be imported or included into your development tool set. How you do this depends on the kind of development environment you use:

- *Repository-based systems* - such as IBM VisualAge for Java, require you to import all the DMLDoc classes into your development environment.
- *File-based systems* - such as Inprise JBuilder, allow you to include the DMLDoc JAR file as a required library for your project.
- *JDK* - This is the simplest arrangement of all: simply include the DMLDoc JAR file with the `-cp` or `-classpath` switch.

These cases are covered in more detail in the [DMLDoc Installation Guide](#).

DMLDoc has a fully-documented API. The JavaDocs for the API are available for download from the Set Information Web site. They are also available online at a fixed URL, allowing you to link the DMLDoc JavaDocs into the JavaDocs for your own applications. Conceptually, the API divides into two:

- *Application API* - a simple API, used to build applications with DMLDoc.
- *Extension API* - a more complex set of methods, allowing the creation of specialisations of DMLDoc with alternative behaviours.

DMLDoc is also divided between its core - the `dmldoc` package - and its servlet framework - the `http` package. The `dmldoc` package can be used on its own, but `http` provides a useful framework alongside the core package.

1.5 Introduction: **Running DMLDoc**

DMLDoc makes writing server-side presentation components easier, but it is not a development tool. DMLDoc is essentially a pure runtime system, whose actions are controlled by the server-side application code.

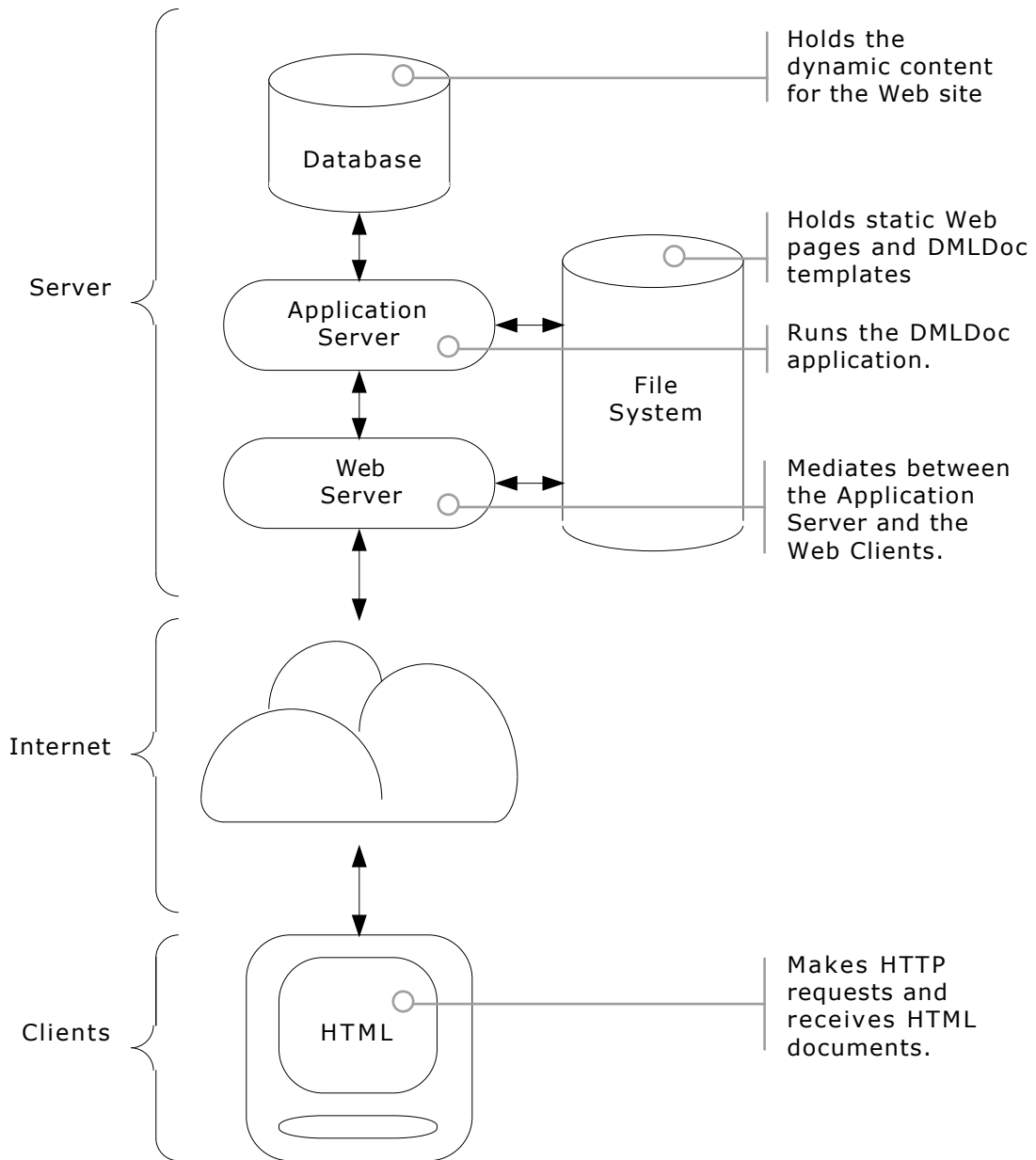
The classes in the core `dmldoc` package will run on any Java 2 (JDK 1.2 and above) virtual machine. Thus, DMLDoc can be used as a general-purpose report generation tool. However, DMLDoc was designed to be sympathetic to the servlet lifecycle, and this should be understood when using DMLDoc in other environments. The lifecycle runs as follows:

- *Initialisation* - This is a one-off operation, after which the servlet object may remain active for a very long time - perhaps weeks or months. Anything which can be done at this stage to optimise later processing is a major advantage.
- *Processing Requests* - This is likely to happen many thousands of times during the lifecycle of a servlet object. Computation at this stage should be minimised as far as possible, as should the creation of objects.
- *Shutdown* - The servlet can anticipate receiving some kind of event indicating that it can - and should - release any resources, and nullify references to objects.

While the `dmldoc` package is a universal textual reporting tool, the `http` package is specific to deployment in a servlet container - it makes use of both `dmldoc` and `javax.servlet.http` classes, and anticipates method invocations compatible with the Java servlet standard.

Whatever runtime system is used, the DMLDoc packages should be included in the classpath of the runtime system or container, *complete with the DMLDoc license JAR file* (the license file is not needed at the development stage). If a deployment tool - such as the J2EE deploytool - is used, it must also have access to the DMLDoc packages.

1.6 Introduction: **The DMLDoc Runtime System**



2 The dmldoc API

The classes contained in the `dmldoc` package are responsible for three kinds of tasks:

- *Parsing Template Views* - DMLDoc must parse each View template before generating a presentation. The parser is guided by the DML commands found in the template, and generates an object structure - the DMLDoc Document Object Model (DOM). The DOM is designed in such a way as to optimise the generation of presentations.
- *Managing Model Data* - When parsing the template's DML commands, the `dmldoc` classes construct the Model expressed by the template. The Model is implemented by these classes as a rich tabular data structure, with a wide range of access methods. Each time a DMLDoc application services a request, it must first populate the appropriate Model.
- *Instantiating Views* - Once the Model has been populated, the `dmldoc` classes can generate a presentation. The presentation is, in effect, an instantiation of a DOM, using an appropriate, populated Model.

The instantiation process will be successful, so long as the Model matches the DOM - this is the only restriction. Thus, a DMLDoc application can populate a Model with many different classes of content, and can manage many different DOMs which share the same Model specification. The latter is important in the case of multi-lingual Web sites. In these cases, Views in different languages can be used to display the same data, or similarly-structured data, with minimal extra coding effort.

The `dmldoc` package does not make a strong distinction between 'application' and 'extension' APIs. On the whole, developers are more likely to extend the structures and behaviour of Model management, and least likely to change the DML parser. Most developers will not need to extend either. In general, application API components are declared as *public*, whereas extension API components are declared as *protected*.

2.1 The dmldoc API: **DMLDoc**

DMLDoc is one of two key classes in the `dmldoc` package. It is used to create DOMs and to view DOMs using Models. The DMLDoc class has a complex internal structure, but it has a very simple API.

DMLDoc methods all fall into one of three loosely-defined categories: creating a new DOM from a template, interrogating an existing DOM, and generating a presentation. The following subsections give a summary of these functions. Subsequent sections show the DMLDoc API in actual use.

2.1.1 The dmlDoc API: DMLDoc: **Creating a DMLDoc**

The DMLDoc class has three constructors. They all generate the same kind of structure, and only differ in the way in which the template is presented. The template is normally a text file. This file can be specified directly, or handed as an `InputStream` or `BufferedReader`. The file specification approach is the most common one, and looks like:

```
DMLDoc myDoc = new DMLDoc(TPL_PATH, "tpl_view.html");
```

In this case `TPL_PATH` is an absolute file path, on a file system accessible to the server. Note that `TPL_PATH` *must* include the final directory separator.

Supplying an `InputStream` instead of a file location is useful where the template is held as a `Resource` within the application's package. However, in this case, the application developer is responsible for opening and closing the `Resource` as an `InputStream`.

When the DMLDoc instance is created, its `Model` is created alongside it. Using this `Model` is the subject of Section 2.2.

2.1.2 The dmlDoc API: DMLDoc: **Interrogating a DMLDoc**

In many cases, it is not necessary to interrogate the DMLDoc instance at all. If the DOM could not be created, an exception would have been thrown. If no exception is thrown, it is safe to assume that the DOM was constructed properly, that its Model can be populated, and that presentations can be generated.

The interrogation methods for a DMLDoc instance are actually defined on the DMLDoc superclass called DMLDocBlock. Currently, there are only two easily accessible methods, examples of which are:

```
System.out.println(myDoc.viewNames().toString());
```

The method `viewNames()` returns a structure that describes the specification of a Model that is relevant to this DMLDoc instance. The method is useful in verifying whether the server-side data can be reconciled with this View.

```
System.out.println(myDoc.toString());
```

The method `toString()` returns a String description of the DOM. The description includes a crude specification of the Model, together with the text that makes up the template. The method does its best to make all this readable, but, given that the template may be large and complex, `toString()` is likely to return a very big String.

2.1.3 The dmlDoc API: DMLDoc: **Generating a Presentation**

This is the easy part. Assuming that the structure of the Model is relevant to the DOM, the presentation will always succeed. Presentation is done using some flavour of the `view()` method, thus:

```
outWriter.println(myDoc.view());
```

prints the complete presentation described by the template used to build the DMLDoc instance, using the instance's Model in its current state.

An alternative Model may be used, and specified as an argument to `view(...)`. Thus, the following is synonymous with the above example:

```
outWriter.println(myDoc.view(myDoc.model));
```

In both cases, if something goes wrong in the process of instantiating the DOM (perhaps because the Model was malformed), an alternative strategy is followed: we assume that there is some client out there, waiting for the presentation response. We should therefore respond with some well-formed document. DMLDoc does this automatically, and reports the nature of the exception. The automatic response is itself - not surprisingly - a DMLDoc. Its template is held as a resource in the `dmlDoc` package. Details of how to alter this template are given in section 2.3.

In some cases, an exception generated during the process of generating a presentation should be intercepted at the application level. This can be done as follows:

```
outWriter.println(myDoc.viewWithException(myDoc.model));
```

In this case, no attempt will be made to find an alternative strategy, and the exception should be caught by the application code.

There is some subtlety in the way in which DMLDoc handles loops within the presentation template. Loops are of the form:

```
<!-- LOOP -->
<TR>
<TD NOWRAP>
  <A HREF="SUB_ACTION_URL?cmd=html&page=update_req&id=SUB_ID">update</A>
</TD>
<TD NOWRAP>SUB_FIRST_NAME SUB_LAST_NAME</TD>
<TD NOWRAP><A HREF="mailto:SUB_EMAIL">SUB_EMAIL</A></TD>
<TD NOWRAP>SUB_PHONE</TD>
</TR>
<!-- ENDLOOP -->
```

In this case, The tokens `SUB_ACTION_URL`, `SUB_ID`, `SUB_FIRST_NAME`, `SUB_LAST_NAME`, `SUB_EMAIL` and `SUB_PHONE` are all associated with different data sources. Those data sources are seen by the application developer as named columns in the `DisplayTable` Model for the template.

The loop will be executed as many times as there are data elements for each of the columns in the `DisplayTable`. If all of the columns have the same number of data elements, then rendering the loop is straightforward - the HTML code within the loop is rendered as many times as there are data elements, with new data used each time around the loop. Where different numbers of data elements are available, the following rules apply:

- The loop will be executed as many times as there are data elements in the longest column.
- If there is exactly one data element in a column, and there are more data elements in other columns, then that one data element will be repeated each time around the loop.
- In other cases, if the loop is repeating but has 'run off the end' of a column, the empty string "" is substituted.

2.2 The dmldoc API: **DisplayTable**

The `DisplayTable` class is used to represent and populate the Model. `DisplayTable` is essentially tabular in structure: it contains named columns for each of the data sources known to a DOM.

In most cases, the application developer need not worry about the internals of `DisplayTable`: this is because a `DisplayTable` is created automatically when a `DMLDoc` is created. Subsequently, the developer only needs to know how to insert data into the `DisplayTable`, or register data with it. The `DisplayTable` will be accessed automatically when the presentation is generated.

Functionally, the `DisplayTable` acts as an ordered container for column objects. The class supports polymorphism at two important points:

- *Columns* - Columns may be of any class that implements the `DMLDoc DisplayList` interface. Different classes of columns can have different behaviours - for example, they may or may not support the update or delete operations. The `dmldoc` package includes a number of implementations of `DisplayList`, which are used by default in certain settings.
- *Fields* - The fields in each column may be of any class that implements the standard Java 2 `Comparable` interface, since this is the minimum requirement for a column to be sortable. Field values are always held as object references. When a presentation is generated, the `toString()` method is applied to each field in turn.

`DisplayTable` methods all fall into one of four categories: inserting data, registering data, sorting, and rendering the `DisplayTable`. The following subsections give a summary of these functions. Subsequent sections show the `DisplayTable` API in actual use.

2.2.1 The dmlDoc API: DisplayTable: **Inserting Data**

When you create a DMLDoc object, a `DisplayTable` is created automatically for you. This `DisplayTable` has columns matching the Model described in the template, and initially contains no data. You can access this `DisplayTable` with:

```
myModel = myDoc.model;
```

A very simple template might only make reference to two data sources, for example `baseHref` (perhaps used to give a location context to the document) and `actionUrl` (used to locate a servlet). Values for these data sources can be given to the Model, and a presentation generated, thus:

```
myModel.clear();
myModel.appendTo("baseHref", "http://www.mydomain.com/");
myModel.appendTo("actionUrl", "http://www.mydomain.com/servlet");
outWriter.println(myModel.view());
```

The `clear()` method is invoked to remove data that may have been inserted into the `DisplayTable` on a previous response to a request. The `appendTo(Comparable, Comparable)` method will always succeed, unless the identifier of the data source (such as `"baseHref"`) is not known to the `DisplayTable`, or the column was read-only. Note that the identifier for the data source (or column) is itself a `Comparable`. Columns also support `get` and `update` methods, so it is possible to do something like:

```
curActionUrl = myModel.getAt("actionUrl", i);
myModel.updateAt("actionUrl", i, curActionUrl + "/Index");
```

Like `appendTo(Comparable, Comparable)`, the `getAt(Comparable, int)` and `updateAt(Comparable, int, Comparable)` methods throw an exception if the column identifier is not known to the `DisplayTable`. All of these methods are described in detail in the `dmlDoc` JavaDocs.

2.2.2 The dmlDoc API: DisplayTable: **Registering Data**

The data insertion methods described above make sense when you are adding small amounts of unstructured data to the `DisplayTable`. In many cases, the aim is to present some volume of structured data. That structured data may already be available in some tabular form. The `DisplayTable` class supports this scenario, by enabling you to register a tabular data structure with a `DisplayTable` instance. An example of this goes as follows:

```
model = viewDMLDoc.model;
model.register(db.findAll());
```

In this case, the `ResultSet` returned by `db.findAll()` is registered with the `DisplayTable` belonging to `viewDMLDoc`.

Registering data is different from inserting data, because no data copying actually takes place. This is important to understand, because if the `ResultSet` that was registered with the `DisplayTable` was subsequently altered, then the `DisplayTable` data would appear to change. In the case of `ResultSets`, that's not too much of a problem, because `ResultSets` are normally read-only structures.

In order to detect data inconsistencies, a strict rule applies to the `register(..)` methods: *all* of the columns in the given structure *must* already be known to the `DisplayTable` instance. An exception is thrown if this is not the case, and no data is registered.

In addition to `ResultSets`, it is also possible to register one `DisplayTable` against another. Examples of this are given in Section 3.2, on HTML forms.

It is also possible to register only certain rows of the data structure; this is useful where `ResultSets` are very large. All of the `register(..)` methods are described in detail in the `dmlDoc` JavaDocs.

2.2.3 The dmlDoc API: DisplayTable: **Sorting**

Once data has been inserted or registered, the `DisplayTable` may contain many fields of data for each column. The `DisplayTable` is not fussy about how many fields are available for each column, or whether columns have the same or different numbers of fields - this is all resolved automatically when a presentation is generated.

When presenting multiple rows of data, both the application developer and the client user may want to sort the data rows in different ways. If the data was obtained from a SQL query, the best approach may be to issue another query with the appropriate "ORDER BY ..." parameters. However, there may be reasons why this should not or cannot be done:

- *Efficiency* - Access to the original data may be computationally expensive (the data may have been obtained from a messaging system) or it may accentuate a bottleneck (in the case of a heavily-loaded database).
- *Part-Set Sorting* - It may be desirable to sort only the rows of the `ResultSet` that were registered against the `DisplayTable`, rather than the whole `ResultSet`.
- *Unavailability of a Sort Option* - Some data sources, such as EJBs, may not offer a sort option. In this case, sorting *must* happen in the `DisplayTable`.

Sorting is achieved as follows:

```
String[] sortOrder = {"company", "lastName", "firstName"};
model.register(db.findAll());
model.sort(sortOrder);
return viewDMLDoc.view();
```

The sort order can include any or all of the column identifiers known to the `DisplayTable`. In this version of DMLDoc, the only available sort direction is ASCENDING.

2.2.4 The dmlDoc API: DisplayTable: **asDMLDoc**

We have already seen that a `DisplayTable` can be generated automatically for any well-formed template `View` - in fact, it almost always is. Here we look at the converse arrangement, where a `View` can be generated automatically from a `DisplayTable Model`. There are two reasons why you might need this functionality:

- *System Monitoring* - The system administrator needs to inspect some data, but there is no need to present this data as a 'styled' HTML page. A generic `View` will suffice.
- *Application debugging* - You have developed an application template, but it is behaving strangely. You can debug the template by examining a definitive view of the data used to instantiate its `Model`.

Generating a `View` from a `DisplayTable` can be done as follows:

```
model = viewDMLDoc.model;
model.register(db.findAll());
return model.asDMLDoc().view(model);
```

The method `asDMLDoc()` generates a generic template for *any* `DisplayTable` instance. Thus, `model.asDMLDoc()` returns a DOM that can be viewed. However, we do not want to view this generated DOM with its own `Model` (which has no data) but with the `Model` we started with.

The template that is used by `asDMLDoc()` is a meta-template - a template used to create templates. Not surprisingly, the meta-template is also a `DMLDoc` template!

If you want to change the style of the meta-template, you can edit the resource file `com.set_i.dmlDoc.tpl_DisplayTable.html`. Information on how to do this is given in Section 2.3.

2.2.5 The dmlDoc API: DisplayTable: **Constructing a Model**

So far, we have only looked at situations where the `DisplayTable` was generated automatically for a `DMLDoc` instance. From the point of view of a business application developer, this is the most important scenario - we believe that, in most cases, the design of the Model should precede the implementation of server-side code.

However, here we examine how to construct a `DisplayTable` explicitly, and suggest some optimisations that can be gained from this approach. The following example should give some guidance:

```
String[] baseHrefVal = { BASE_HREF };
String[] actionUrlVal = { ACTION_URL };
model = new DisplayTable();
model.addColumn("baseHref", new DisplayROArray(baseHrefVal));
model.addColumn("actionUrl", new DisplayROArray(actionUrlVal));
model.addColumn("dynamicData");
```

In this trivial example, only the column identified by `dynamicData` can accept new data. The `baseHref` and `actionUrl` are held in read-only columns of fixed size. Such columns ignore the `clear()` method. Thus, when the application responds to a request, it only need do:

```
model.clear();
model.appendTo("dynamicData", aDataPoint);
myDoc.view(model);
```

Here, the `clear()` method only has an effect on the `dynamicData` column, so this is the only column that needs to be updated - or indeed can be updated - in response to a request. Any attempt to update the `baseHref` or `actionUrl` columns will result in a `UnsupportedOperationException`. The following subsection summarises all of the implementations of `DisplayList` provided with `DMLDoc` Standard Edition, version 1.

2.2.6 The dmlDoc API: DisplayTable: **DisplayList** Classes

DMLDoc manages the use of the alternate implementations of `DisplayList` automatically. A developer, therefore, does not need to know this aspect of `DisplayTable` operations in order to build sound applications. However, some optimisations can be gained by the use of the alternate or new `DisplayList` classes. The existing implementations are:

- `DisplayVector` - This is the default column class: when a `DisplayTable` is created for a new `DMLDoc`, all of its columns are of this class. Columns of class `DisplayVector` allow updating and appending of fields. `DisplayVector` responds to `clear()` by removing all of its elements.
- `DisplayResultSet` - This column class is used when a `ResultSet` is registered against a `DisplayTable`. `DisplayResultSet` mediates between the `DisplayList` interface and the `ResultSet` methods. A `DisplayResultSet` is read-only, and ignores `clear()`.
- `DisplayArray` - A column of a fixed size. Columns of class `DisplayArray` allow updating of fields, but not appending of fields. `DisplayArray` responds to `clear()` by setting all of its elements to `null`.
- `DisplayROArray` - A column of a fixed size. `DisplayROArray` is read-only - columns of class `DisplayROArray` do not allow updating or appending of fields. `DisplayROArray` ignores `clear()`.

The class of any column in a `DisplayTable` can be changed as follows:

```
model = viewDMLDoc.model;
model.updateColumn("baseHref", new DisplayROArray(baseHrefVal));
model.updateColumn("actionUrl", new DisplayROArray(actionUrlVal));
```

Full details of all these classes - and the `DisplayList` interface - are given in the `dmlDoc` JavaDocs.

2.3 The dmldoc API: **Template Resources**

Template resources are used by DMLDoc to provide standard HTML response pages in debugging or exception handling situations. The templates can be used as they are, or they can be altered by the application developer. DMLDoc Standard Edition, version 1 of uses two templates:

- `com.set_i.dmldoc.tpl_DMLDocWriteException.html` - This template is used to report exceptions that occurred during an attempt to generate a presentation. In a production environment, the template can be altered to display an apology for the response failure.
- `com.set_i.dmldoc.tpl_DisplayTable.html` - This is the meta-template, used by `asDMLDoc()` to generate standard Views of `DisplayTables`. The [DMLDoc Web Designer's Guide](#) includes a Case Study for this meta-template.

To change either template, simply unpack the `dmldoc` JAR file, and edit the template file. It would be a good idea to read the [DMLDoc Web Designer's Guide](#) first. Editing either file is easy, but the following warnings should be observed:

- *Do not alter or remove any SUB commands* - The `dmldoc` code assumes that the template describes a particular Model. Altering the DML **SUB** commands is likely to alter the Model. If you do not need a particular data source, simply do not use it in the body of the template.
- *Do not rename the files* - Template files must retain their original names.
- *Remember that the templates are used universally* - All servlets running on the same server will make use of the same template.

When editing is complete, the `dmldoc` JAR file can be remade. However, remaking the JAR is not necessary, as the classes can be accessed from the unpacked directory structure.

3 **The http API**

Whereas the `dmlDoc` package deals with universal text reporting operations, the `http` package is solely concerned with supporting application servlets. It is not necessary to use the `http` package - even if you are building servlets - but the two packages complement one another very well. It *is* necessary to include the `dmlDoc` package when using the `http` package. The `http` package is concerned with three separate types of servlet operations:

- *Servlet Response to HTTP GET and POST* - The `http` package contains its own specialisation of the `javax.servlet.http.HttpServlet` class. This abstraction captures generalisations about URL parameter handling, response object management, and exception handling, and other tasks.
- *HTML forms elements* - Forms elements are supported through the use of Java classes that represent different types of form elements. These classes provide a robust, expressive and efficient method of generating forms and receiving forms data.
- *HTTP MultiPart Forms* - The `javax.servlet.http` package contains no direct support for multi-part forms. The DMLDoc `http` package remedies this, with support for multi-part binary and text data sent by the browser to the server.

Wherever practical, the `http` classes provide a framework for servlet construction. In other words, the `http` structures and methods provide a coordinated set of fine-grained services that mediate between the `javax.servlet.http` layer and the application's business logic.

Because of this approach, the use of the DMLDoc `http` services is best understood by example. The following subsections give details of each of the important methods, while Section 6, Making it All Work, gives a broader view of effective servlet structure.

3.1 The http API: **HttpServlet**

`HttpServlet` is the key class in the `http` package. If you are building a DMLDoc application that uses servlets, then your servlets ought to subclass `com.set_i.http.HttpServlet`, rather than `javax.servlet.http.HttpServlet`.

The DMLDoc `HttpServlet` methods all fall into one of three loosely-defined categories: handling URL parameters, intercepting `doGet(..)` and `doPost(..)` methods, and establishing the configuration of a servlet.

3.1.1 The http API: HttpServlet: **Indexing Parameters**

Almost inevitably, a servlet will expect some parameters to be passed to it¹, in the form:

```
http://my.domain/servlet/MyServlet?param1=value1&param2=value2
```

These parameters are accessible to the servlet with a method on the request object:

```
req.getParameter("param1");
```

The method returns the value for the parameter. If the URL did not carry a value for the parameter, then the method returns the empty string "". If the parameter was not named in the URL, then the method returns `null`.

This is logical, but unhelpful when it comes to acting on given (or, worse, omitted) parameters.

Java has a `switch` statement intended for precisely this sort of decision point. Unfortunately, the Java `switch` statement can only act over `int` values. The first task, therefore, is to map the `String` parameter values onto `ints`. We use the following in the application servlet's class definition:

```
private static final String CMD = "cmd";
private static final String[] CMDS = { "html", "debug", "init" };
private static final int CMD_HTML = 0;
private static final int CMD_DEBUG = 1;
private static final int CMD_INIT = 2;
```

When the servlet is initialised, it should include the following method:

¹ We deal here with an HTTP GET method, the case for PUT is essentially the same.

```
public void init(ServletConfig conf) throws ServletException
{
    ...
    registerIndices(CMD, CMDS);
}
```

The `String[]` must contain only `String` identifiers which are unique, ignoring case. Now, on a GET or a PUT, the servlet can use the following form:

```
switch (getParamIndex(CMD, req))
{
    case CMD_HTML:
        out.println(normal(req));
        break;
    case CMD_INIT:
        init(conf);
        out.println(initialize());
        break;
    case CMD_DEBUG:
        out.println(debug(req));
        break;
    default:
        out.println(report(req,
            "unknown command:" + req.getParameter(CMD)));
}
```

The `getParamIndex(...)` method returns the position of the command in the original `String[]`, or `-1` if the parameter is missing, or its value is unknown; this switch construct is therefore always effective. Added to this, listing all legal parameter name / value sets at the start of the servlet's class definition is a big help in creating maintainable code.

The `HttpServlet` class contains a number of methods for registering parameter values against different parameter names, and for displaying parameter name / value sets.

3.1.2 The http API: HttpServlet: **doGet** & **doGetDispatch**

A minimum amount of housekeeping work must be done by all `doGet(..)` and `doPost(..)` methods. This includes:

- Setting the content type for the response.
- Gaining a `Writer` object from the response object.
- Write the response data to the `Writer` object.
- Catching and handling any exceptions that were thrown in the process of servicing the request.
- Closing the `Writer` object, irrespective of any exceptions.

The DMLDoc `HttpServlet` class provides `doGet(..)` and `doPost(..)` methods that do all of these these tasks. At the core of these methods are calls to methods called `doGetDispatch(..)` and `doPostDispatch(..)`, respectively. The servlet developer should implement the latter methods on his or her subclass of `com.set_i.http.HttpServlet`. For example:

```
protected String doGetDispatch(HttpServletRequestRequest) throws Exception
{
    switch (getParamIndex(CMD, req))
    {
        case CMD_NORMAL:
            return(normal(req));
        case CMD_INIT:
            init(conf);
            return(initialize());
        case CMD_DEBUG:
            return(debug(req));
        default:
            return(report(req,
                "unknown command:" + req.getParameter(CMD)));
    }
}
```

The cost of this arrangement is one extra method call. However, the return on this spend is an assurance that the `doGet(..)` method will be well behaved.

Note that the `doGetDispatch(..)` method is declared as throwing any exception, anticipating that *anything* could go wrong with any of the service methods. This is not normally a good Java programming style, but in this case it is acceptable - we know that the exception will be caught by the next method down the execution stack - the `doGet(..)` - and will be reported properly.

A special case exists where the application servlet must perform some action *before* the initial tasks of the default `doGet(..)` method. For example, the servlet may want to establish a `Session` object. In this case, the best alternative is to override the `doGet(..)` method, perform the necessary task, then reestablish the original control flow:

```
public void doGet(HttpServletRequest req, HttpServletResponse rsp)
{
    session = req.getSession();
    super.doGet(req, rsp);
}

protected String doGetDispatch(HttpServletRequest) throws Exception
{
    ...
}
```

The `doGetDispatch(..)` method will proceed as before.

The `com.set_i.http.HttpServlet` class does implement its own, functioning `doGetDispatch(..)` method. However, this method merely reports (by way of an HTTP response) that it should be overridden by some subclass.

3.1.3 The http API: HttpServlet: **Reporting Exceptions**

The examples in the preceding subsections made use of the method:

```
report(HttpServletRequest, String)
```

The `report(HttpServletRequest, String)` method generates a complete report on the servlet, together with details of the request that triggered the `doGet(..)` or `doPost(..)` method.

The `String` parameter is included in the report. It can be used to carry an exception's `getMessage()` if an exception is caught within application code, or any other descriptive message. It could even contain HTML tags (but think carefully before you do that).

The `report(..)` method is used by the `com.set_i.http.HttpServlet.doGet(..)` and `com.set_i.http.HttpServlet.doPost(..)` methods, to report on any exceptions raised by `doGetDispatch(..)` and `doPostDispatch(..)`. The singular advantage of this method is that it can be relied on to produce a valid response to a request, even when the application code has tripped up.

The format of the report is determined by a resource template, located at:

```
com.set_i.http.tpl_ServletReport.html
```

This template can be altered in the same way as those for the `dmlDoc` classes. The template really should be altered in a production environment, because it reveals more about the workings of your servlet than you might want to reveal to the general public.

Details of how to alter the template are given in Section 2.3.

3.1.4 The http API: HttpServlet: **setConfig**

In many cases, application servlets are developed on one platform, tested on a second platform, then deployed on a third.

This is a sound practice - the production or deployment platform must be treated as something of a sacred space, which is tightly controlled and open only to stable code. A test platform may also be made available, this should be a clone of the deployment platform. It should be used as a stepping stone for deployment candidates, and as a place to do performance measurements and other experiments.

Application servlets must be configured for each platform, and this configuration should *not* require recompilation of the servlet source code - that way lies the dark side. The Java servlet standard makes an allowance for this arrangement in the form of initialisation parameters.

The `setConfig()` method takes this a stage further, by allowing developers to create application resource files containing all the relevant initialisation parameter names and values. Thus if `com.mydomain.website.MyServlet` does:

```
public static String TPL_PATH;
public static String ACTION_URL;
public static String JDBC_URL;
...
public void init(ServletConfig conf) throws ServletException
{
    setConfig();
    ...
}
```

the framework will look for a resource text file at `com/mydomain/website/MyServlet.conf` and attempt to parse it. Resource files of this sort are included in the example DMLDoc applications, and have a very simple syntax.

The Syntax is:

```
parameterName <whitespace> parameterValue <newline>
```

The resource files may also include comments. Comments must begin on a new line, and are of the form:

```
# comment <newline>
```

When the `setConfig()` method parses the configuration file, it looks for static variables in the servlet's class whose names match those in the configuration file. Where matches are found, the static variable is assigned the given value. There are some subtleties to this process:

- *Reassignment* - The method is intended to be used where the servlet program code does not itself assign any value to the static variable. If it does, `setConfig()` issues a warning to `System.err`. This also holds where the configuration file carries more than one assignment for the same static variable.
- *Class String* - Is the only class for a static variable under these circumstances. Attempting to assign a value to a static variable of another class will cause a casting exception.
- *Access restrictions* - Any attempt to assign a value to a static variable which is not declared as `public` (or not declared at all) will cause a `ServletException` to be thrown.
- *Initialisation* - A side-effect of `setConfig()` is to reinitialise the servlet. Thus, `setConfig()` should only be called from the servlet's `init(..)` method.

A related method `setConfig(String)`, allows the developer to specify the name of the resource file. In this way it is possible to share configurations between servlets in the same package.

3.2 The http API: **HTML Form Elements**

The `http` package contains a number of classes that provide support for HTML forms elements. The original goal for these classes was to ensure that HTML tags would never have to be handled by DMLDoc server-side code.

Although the Standard Edition, Version 1 of DMLDoc does not make provision for every type of forms element, the original goal still applies, and has largely if not wholly been achieved.

The use of these forms elements imposes simple but specific requirements on the Web designer who is developing forms templates for DMLDoc applications. The [DMLDoc Web Designer's Guide](#) shows how this is done.

The `http` forms elements support both the creation of HTML forms, and the handling of data returned from the browser on a form submit. The following subsections show how this is done.

3.2.1 The http API: HTML Forms: **Select & Countries**

The HTML SELECT tag introduces a list of items that the user may choose. There are two very different subtypes of SELECT, depending on whether the MULTIPLE modifier is used - the absence of MULTIPLE restricts the user's choice to only one item, whereas the presence of MULTIPLE enables the selection of any number of options.

Browsers are free to implement both forms of SELECT in any way they choose, and give any meaningful visual representation of the SELECT structure - Web designers should at all times be aware of this.

The `com.set_i.http.Select` class works for both subtypes of SELECT. The way in which a `Select` structure is created is not dependent on whether the Web designer has specified MULTIPLE or not. However, on handling the returned data, the server-side programmer must be aware of how the `Select` structure was being used.

To create a `Select` statement, you can do the following:

```
String[] myNames = { "Natasha", "Bruno", "Mathew" };  
Select mySelection = new Select("people", myNames);
```

This creates a `Select` structure whose names (the elements that will be visible to the user) and values (the elements that will be returned by the form submit) are the same - they are both specified by `myNames`. It is also possible to create `Select` structures with different names and values - check the JavaDocs. In all cases, the names must be unique.

It is possible to preselect one or (if the structure is to be used in a MULTIPLE SELECT) more entries. This is done with:

```
mySelection.setSelectedName("Bruno");
```

If this is done, then the named entry will appear as `SELECTED` in the HTML presentation. It is also possible to select an entry using `setSelectedValue(String)`, but be warned: values do not have to be unique, and `setSelectedValue(String)` will only select the first instance of an entry with the given value.

The `Select` object is actually implemented as a `DisplayTable`, and all of the `DisplayTable` methods apply. The identifiers given to each column are a concatenation of the id given when the `Select` was created together with a standard name for the column - "Names", "Values", and "Select". Thus, for the above example, the "Values" column is called "peopleValues". The developer can view the state of the `Select` structure with:

```
mySelection.asDMLDoc.view(mySelection);
```

Depending on the application, the developer may want to create a name / value index set for the values that can be returned by the `Select` structure. An example of this approach is given in the following subsection on `Radio`.

Once a select structure has been constructed, it can be registered against the `DisplayTable` for the appropriate `Model`. Because the columns in the select structure are well-behaved with respect to `model.clear()` operations, the select structure needs only to be registered once against the `Model`, in the initialisation phase. The following example makes use of the special `Countries` select structure:

```
requestReqDMLDoc = new DMLDoc(TPL_PATH, "tpl_request_req.html");
countries = Countries.select("country", com.set_i.http.Countries.GB);
requestReqDMLDoc.model.register(countries);
```

When a `model.clear()` operation is carried out, the "Names" and "Values" columns remain unchanged, whereas the "Select" column entries are reset to null - in other words, no entry is preselected to `SELECTED`.

`Countries` provides `Select` structures that are dedicated to displaying HTML `SELECT` lists of the countries of the world - these are frequently used in address forms.

`Countries` provides two really useful benefits: firstly, it saves the effort of assembling a list of countries. Secondly, it provides an international standard list of countries that can - and should - be used by all relevant applications. Standardising on one list of countries makes database operations over country fields *much* more reliable.

The `Countries` class simply provides a way of creating `Select` objects that are already populated with country names. Thus, `Countries` is an abstract class, with a class method that allows you to create a countries-populated `Select` object:

```
Select country = Countries.select("country",  
                                com.set_i.http.Countries.GB);
```

This 'pseudo-constructor' takes as its second argument the language for the countries. The language does not affect the values for the countries selector - it only effects the names. Not that the DMLDoc Standard Edition, Version 1 only supports English. Future versions will support other languages. In the mean time, the developer can subclass `Countries` to incorporate other languages.

A `Countries` `SELECT` structure returns the international standard two-letter country code. If you want the country name instead, it is possible to do, for example:

```
countryName = Countries.getName(com.set_i.http.Countries.GB,  
                                req.getParameter("country"));
```

The `http` package JavaDocs give more details of the use of `Countries`.

3.2.2 The http API: HTML Forms: **Radio**

Although visually very different, the HTML RADIO element has a similar structure and function to the SELECT element (especially so when the SELECT element does not specify MULTIPLE). The `com.set_i.http.Radio` class reflects this similarity.

Like a Select object, the Radio object is a tabular structure, with separate, named columns. The columns provide the following information for each element of the RADIO group:

- The HTML NAME of the radio button - in other words, the radio value returned when the form is submitted.
- The text accompanying the radio button - like the Names column in the Select structure, this text has only a visual use.
- Whether a radio button is checked by default.

The JavaDocs for the `com.set_i.http.Radio` class give instructions for the construction of Radio objects, and the necessary HTML and DML code.

Given that a RADIO form element is going to cause some radio name to be returned on the form submit, it is a good idea to declare all the valid radio options at the start of the servlet class:

```
private static final String RQMT = "rqmt";
private static final String[] RQMT_TEXTS =
    { "new project.", "extra site.", "deploying developed software." };
private static final String[] RQMT_VALUES =
    { "rqmtFirst", "rqmtExtra", "rqmtDeply" };
private static final int RQMT_FIRST = 0;
private static final int RQMT_EXTRA = 1;
private static final int RQMT_DEPLY = 2;
```

Now the DMLDoc and the `Radio` element can be built in the servlet's `init()` method:

```
requestReqDMLDoc = new DMLDoc(TPL_PATH, "tpl_request_req.html");
...
rqmt = new Radio(RQMT, RQMT_TEXTS, RQMT_VALUES);
requestReqDMLDoc.model.register(rqmt);
```

Like the `select` object, the `Radio` object will respond to `model.clear()` by keeping the text and names in place, but removing and default selection on the radio buttons. Thus, it doesn't need to be re-registered each time a GET or POST is serviced.

If a form needs to be returned to the user (perhaps because it failed a server-side validation) then the user's original radio button selection can be set on the new form with:

```
model.clear();
model.appendTo("baseHref", BASE_HREF);
model.appendTo("actionUrl", ACTION_URL);
rqmt.checkValue(lr.getRequirement());
```

Where the object `lr` carries all the original form settings.

`Radio` objects are inherently simpler than `Select` objects, because there is no ambiguity here about multiple selections. The `checkValue(String)` method strictly allows only one button to be checked - it clears all others. Likewise, all `Radio` names must be unique - the constructors for `Radio` throw a `DisplayTableException` if a given names element is not unique. New elements may not be added to the radio group once it has been constructed.

3.3 The http API: HTML Forms: **HTTP MultiPart Forms**

Multi-part form data - strictly, data from a HTTP "multipart/form-data" POST operation - is used when the browser must send binary data, or large volumes of any data back to the server. This normally happens as the result of a form submit.

The most common use of this technique is in supporting a file upload operation from a form. The file upload operation is not often used - quite possibly because it has been so difficult to implement! However, "file" is a standard INPUT form element TYPE, and is widely supported by Web browser software.

The form looks like:

```
<FORM METHOD="POST" ACTION=".." ENCTYPE="multipart/form-data">
  <INPUT TYPE="file" NAME="userfile" SIZE="60" MAXLENGTH="128">
  <INPUT TYPE="input" NAME="text" SIZE="60" MAXLENGTH="128">
</FORM>
```

(The value of ACTION is omitted for simplicity.)

On the server side, we create a parameter index to cope with all the INPUT fields in the form:

```
private static final String POST = "post";
private static final String[] posts = { "userfile", "text" };
private static final int POST_FILE = 0;
private static final int POST_TEXT = 1;
```

Now we need to implement a method that will process the POST - in other words, we should override the default `doPostDispatch(..)` method. The method makes use of the `com.set_i.http.Multiform` class, as follows:

```
public synchronized void doPostDispatch(HttpServletRequest req)
    throws Exception
{
    DisplayTable model = upCmtDMLDoc.model;
    MultiForm pMF;
    FileOutputStream outStrm;
    int i;

    model.clear();
    pMF = new MultiForm(req);

    for (i = 0; i < getValueCount(POST); i++) {
        switch (getIndex(POST, pMF.getNextParam()))
        {
            case POST_FILE:
                outStrm = new FileOutputStream(IMG_PATH +
                    pMF.getParamFilename());
                pMF.writeParamValue(outStrm);
                outStrm.close();
                model.appendTo("baseHref", BASE_HREF);
                model.appendTo("imageFile", IMG_REF +
                    pMF.getParamFilename());
                break;

            case POST_TEXT:
                model.appendTo("text", pMF.getParamValue());
                break;

            default:
                throw new Exception("unknown parameter:[" +
                    pMF.getParamName() + "]");
        }
    }
    return(upCmtDMLDoc.view());
}
```

The switch statement must exist inside a loop, because the multi-part POST is supplied as sequential data - given the volume of data that might be returned, it is not practical to pick randomly amongst the available parameter values. In the above case, the loop block should only be executed twice, because there only are two valid parameters.

The method is synchronized, because it makes use of a unique resource - in this case, a `FileOutputStream` object for a named file - that might be required in order to respond to another POST. In general, the aim is to implement `doGetDispatch()` and `doPostDispatch()` methods in such a way that they do not need to be synchronized - even if individual service methods must be. More on this in Section 6 'Making it All Work'.

When using the `MultiForm` methods, care must be taken to ensure that form elements are accessed in the correct order. `MultiForm` won't break if you get it wrong, but the results could be confusing. Use the code on the previous page as a template for your own servlet.

A better implementation of a multi-part form would provide for an enumeration of form parts. This is likely to be provided in future releases.

4

DMLDoc & JDBC

To explain the relationship between JDBC data access and DMLDoc, this section steps through the process of defining parts of a JDBC Web application. Examples in this section are all taken from the JDBC Contact Book demonstration. The source code for the demonstration is available from The Set Information Web site. Specifics of the demonstration application are given in Section 7, Case Studies.

To begin, we define a Model associated with one part of the application. The Model is established using the DML **SUB** command, at the start of the HTML template:

```
<!-- SUB action_url SUB_ACTION_URL -->
<!-- SUB contactId SUB_ID -->
<!-- SUB firstName SUB_FIRST_NAME -->
<!-- SUB lastName SUB_LAST_NAME -->
<!-- SUB emailAddress SUB_EMAIL -->
<!-- SUB phoneNumber SUB_PHONE -->
```

The first parameter of each **SUB** command names the data source, as it is known to the server-side application. The second parameter is the target, as it is used in the template. The template can therefore contain:

```
<!-- LOOP -->
<TR>
<TD NOWRAP>
  <A HREF="SUB_ACTION_URL?cmd=html&page=update_req&id=SUB_ID">update</A>
</TD>
<TD NOWRAP>SUB_FIRST_NAME SUB_LAST_NAME</TD>
<TD NOWRAP><A HREF="mailto:SUB_EMAIL">SUB_EMAIL</A></TD>
<TD NOWRAP>SUB_PHONE</TD>
</TR>
<!-- ENDLLOOP -->
```

The first task of the server-side application is to establish a connection to the database. This should be done in the `init()` method of the Servlet, and looks something like:

```
Class.forName(JDBC_DRIVER).newInstance();
DriverManager.setLoginTimeout(10);
conn = DriverManager.getConnection(jdbcURL + CloudscapeDB;create=true");
```

In this case the driver class was `COM.cloudscape.core.JDBCdriver`. The connection object can now be used to create SQL statements and execute queries against the database. If the connection cannot be made successfully, then a `UnavailableException` must be raised.

In most circumstances, the servlet need only make one database connection. That database connection will be held throughout the lifetime of the servlet, and be used for all queries, by all of the service methods and all instances of the servlet. Some care must be exercised to make sure that clashes in database access do not take place.

Databases vary in the way in which the connection should be closed. Whatever the technique, the connection must be closed when the servlet is destroyed, or when the servlet encounters an exception that would render it unable to process further requests. For the Cloudscape database, closing the connection is done as follows:

```
try {
    DriverManager.getConnection(jdbcURL + ";shutdown=true");
    System.err.println("database shut down abnormally");
} catch (SQLException sqlE) {
    System.err.println("database shut down normally");
}
```

More information about the Cloudscape database is available at:

<http://www.cloudscape.com/>

4.1 DMLDoc & JDBC: **SQL Queries & ResultSet**

SQL queries are executed using a statement object obtained from the database connection. A typical query runs as follows:

```
Statement st = conn.createStatement(  
    java.sql.ResultSet.TYPE_SCROLL_INSENSITIVE,  
    java.sql.ResultSet.CONCUR_READ_ONLY);  
  
ResultSet rs = st.executeQuery(  
    "SELECT * FROM " + CONTACT_TABLE +  
    " ORDER BY lastName, firstName");
```

The `ResultSet` object (strictly a `java.sql.resultSet`) is described in detail in the Java 2 JavaDocs. Note that there are some important differences between this and previous versions of `java.sql.resultSet` - the DMLDoc demonstration applications and the `dmldoc` classes all make use of features that were first introduced in the Java 2 version.

Note that this query returns all of the columns in `CONTACT_TABLE`. That makes sense in this case, because the Model introduced by the template also includes all of these columns. If that were not the case, The `SELECT` statement would have to list precisely the columns that were to be displayed.

The `ResultSet` object returned by the query contains zero or more rows of actual data (the `ResultSet` also contains some metadata). It is acceptable to register a `ResultSet` containing no data against a `DisplayTable`, and to generate a presentation with that `DisplayTable`. However, it is more usual to catch the 'no data' case, and communicate this to the template using an extra data source - often called `nullData` - that can be set to "true" or "false".

4.2 DMLDoc & JDBC: **DisplayTable & ResultSet**

A big advantage of DMLDoc is that the application need never manipulate the `ResultSet` object - the `ResultSet` can simply be registered against the `DisplayTable`. The 'view all entries' service method in the JDBC Contact Book example runs as follows:

```
private synchronized String viewRsp() throws Exception
{
    DisplayTable model = viewDMLDoc.model;

    model.clear();
    model.appendTo("action_url", ACTION_URL);
    model.register(db.findAll());
    return viewDMLDoc.view();
}
```

In this case, the `findAll()` method encapsulates the code that obtains the statement object and issues the SQL query, as given on the previous page.

The `model.register(..)` method causes a reference to the `ResultSet` object to be held for each relevant column in the `DisplayTable`, along with the `ResultSet` column name - this is done using `DisplayResultSet` objects, attached to each relevant column in the `DisplayTable`.

When `viewDMLDoc.view()` is invoked, the `DisplayResultSet` object translates between `DisplayTable` accesses and `ResultSet` accesses. It is therefore essential that the `ResultSet` is not altered between the time when it is registered and the time when the `view()` operation is completed.

Once the `view()` operation is completed, the `ResultSet` can be discarded. In the above example, it never is discarded. However, subsequent `register(..)` operations will remove all references to the old `ResultSet`.

5 DMLDoc & EJBs

This section looks at the process of establishing and maintaining a connection to an Enterprise JavaBean (EJB) server, together with the process of finding EJBs and inserting EJBs into a `DisplayTable`. The examples here are taken from the EJB Contact Book demonstration. Specifics of this application are given in Section 7, Case Studies.

The Model's requirements in the case of an EJB application are in every way identical to those for a JDBC application. This is because the Model does not contain any information about the source of the data.

The structure of a DMLDoc EJB application is largely the same as any other DMLDoc application:

- *Initialisation* - DMLDoc objects are created from the template files, parameter indices are registered, and a connection is established to one or more data sources.
- *Servicing Requests* - URL parameters are used to determine the details of the request. The appropriate data is then found or inserted, and a presentation is generated.
- *Shutdown* - The servlet closes connections and nullifies references to external objects.

The chief difference between JDBC and EJB applications is the way in which data is accessed and given to the `DisplayTable`. But there are performance issues too - EJB accesses are computationally expensive, so the more local caching that can be done, the better.

To this end, the EJB Contact Book demonstration sorts existing `DisplayTables`, rather than attempting to construct new `DisplayTable` data, and sorting that. *The application does not attempt to maintain these `DisplayTables` on a per-user basis - a real-world application must!*

5.1 DMLDoc & EJBs: **Accessing EJBs**

A DMLDoc application that accesses EJBs should make contact with the EJB server in the `init()` method, much in the same way as a JDBC application establishes contact with a database Driver. This EJB access looks like:

```
InitialContext ctx = new InitialContext();
Object objRef = ctx.lookup(CONTACT_SERVICE);
ContactHome contacts = (ContactHome) PortableRemoteObject.narrow(
    objRef, ContactHome.class);
```

From the point of view of the DMLDoc developer, the `ctx` and `objRef` objects are simply stepping stones on the goal of accessing the EJBs. The `contacts` object is the crucial one - it is an instance of the `ContactHome` class, which implements the Home interface of the EJB. The Home interface allows individual persistent objects to be discovered.

Again, as with the JDBC case, if the `ContactHome` instance cannot be obtained, then there is nothing more that can be done, and the servlet must raise an `UnavailableException`.

Unlike the JDBC database driver, the EJB container can be treated like an out-of-process component - it has its own lifecycle, which is unaffected by the lifecycle of any client servlets. Thus, when the DMLDoc application servlet is shut down, it need only do:

```
contacts = null;
```

(In our example, the `ctx` and `objRef` objects were local to the `init()` method, and have long since been erased from the execution stack.)

Finding EJB data is a very different process to making an SQL query. An example of a find - functionally equivalent to the `db.findAll(..)` of the

previous section - is:

```
private synchronized void viewRsp()
    throws Exception
{
    DisplayTable model = viewDMLDoc.model;
    Iterator itr = contacts.findAll().iterator();
    Contact thisContact;

    model.clear();
    model.appendTo("action_url", ACTION_URL);

    while (itr.hasNext()) {
        thisContact = (Contact) itr.next();
        model.appendObject(thisContact);
    }
}
```

The purpose of this method is to generate a presentation that shows all the 'Contact Book' entries. In pursuit of this, the `contacts.findAll()` method returns an iterator over `Contact` objects known to the server.

The `Contact` class is in fact the Remote interface of the contact EJB. Like any other Java interface, it determines what can be done, but does not define how. A DMLDoc EJB application needs to know exactly this - in other words, it needs to know what can be done with the EJB, but does not need to concern itself with how the EJB implements its interface.

The implementation of these actions is determined by a combination of the core `ContactEJB` class (the DMLDoc application never needs to see this) and the deployment settings of the EJB. These deployment settings include transaction boundaries for EJB functions, making EJB access threadsafe for servlets in a number of important respects. Note that reentrancy is only an issue for Entity EJBs, and not Session EJBs - Session EJBs can never service more than one servlet thread.

5.2 DMLDoc & EJBs: **DisplayTable & EJBs**

The example given in the preceding subsection included the lines:

```
while (itr.hasNext()) {
    thisContact = (Contact) itr.next();
    model.appendObject(thisContact);
}
```

Thus, the method iterates over every available `Contact` object, and inserts these one by one into the `Model`. From the point of view of `DMLDoc`, this is the most important difference between JDBC access and EJB access - a JDBC query `ResultSet` demands a column-oriented view of data, whereas a EJB `Iterator` demands a row-oriented view of data. `DMLDoc` supports both - in the same `DisplayTable`.

As it turns out, `appendObject(Object)` is the only method that `DisplayTable` needs to implement in order to insert EJB data. (Note that `DMLDoc Standard Edition, Version 1` does not supply an equivalent `registerObject(Object)` method.)

The operation of `appendObject(Object)` includes a relatively standard use of the Java Reflection API: the interface of `Object` is examined for public access methods of the form `getField()`, where `Field` is assumed to identify an instance variable. Where `Field` matches the name of a column in the `DisplayTable`, that column has the value of `getField()` added to it. (However, if the column class does not support `add(Object)` then `appendObject(Object)` will throw a `DisplayTableException`.)

Note that the `appendObject(Object)` method is much less strict than `register(ResultSet)`. There is a good reason for this: typically, the developer has much more control over SQL queries than EJB structures. Thus, the `DisplayTable` simply makes a 'best effort' to reconcile this data.

6 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 server-side programmer - the tasks for the Web Designer are covered in the [DMLDOC Web Designer's Guide](#).

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 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.

Share a data dictionary - One of the most difficult tasks in a DMLDoc application is synchronising the use of data 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 will. In the mean time, it's a good idea to use some *ad hoc* method, such as a personal Web server page giving details of each substitution source to all the team members.

Understand servlet runtime behaviour - Servlets can look deceptively simple (especially when they are well-structured). The horrible reality is that servlet runtime behaviour is quite subtle, and not always well-expressed by the source code. The following subsections should help to make the most important points clear.

6.1 Making it all Work: **DMLDoc Runtime Behaviour**

This section reviews the runtime operations associated with a DMLDoc servlet application in the general case. The points raised in this section should be taken as guidelines for the design of any sort of DMLDoc application:

Initialisation: the license - The first time a `DMLDoc` or `DMLDocBlock` object is created, the constructor checks for the presence of a license file. If the license file is found and is valid, then the license details are written to `System.err`, and the object is created. If the license file is not found, or is invalid, the `DMLDoc` or `DMLDocBlock` object cannot be created.

The validity of the license is determined by the hash value making up the last component of the license details. For DMLDoc Standard Edition, licenses have no time limit, and are free of charge.

The license file is provided in a JAR file, whose internal structure reflects that of the `dmldoc` JAR file - the license is located within the JAR file, at `com/set_i/dmldoc/license.txt`. The license is accessible, irrespective of whether the JAR is unpacked. The license must be installed in the classpath of the servlet container, but is not required by the Java compiler.

Initialisation: the conf file - If you have chosen to create a configuration file to accompany the servlet, you should make sure that it is read at the start of the servlet's `init(..)` method, for example with:

```
public void init(ServletConfig conf)
    throws ServletException, UnavailableException
{
    super.init(conf);
    this.conf = conf;
    setConfig();
    ...
}
```

In this example, if the servlet class is `MyServlet`, then the `setConfig()` method will look for a file called `MyServlet.conf`.

The order of events shown above is quite important - `setConfig()` must be invoked before any other tasks in the `init(..)` method. This is in part because `setConfig()` causes the servlet to be initialised again (no, `setConfig()` does not function the next time around!) and in part for the very obvious reason that many other tasks in the `init(..)` method need the values obtained from `setConfig()`.

Initialisation: templates - Preceding sections discussed the fact that constructing `DMLDoc` objects from templates is a computationally expensive operation, whereas instantiating a `DisplayTable` and generating a presentation is relatively cheap.

This is a good reason for building all `DMLDoc` objects in the servlet's initialisation phase, but there is another reason: the process of constructing a `DMLDoc` object from a template may fail - this is usually because the DML commands in the template are malformed, or the template cannot be found. Managing the servlet is a great deal more straightforward if all such failures can all be caught at the same time.

Remember that the `init(..)` method is not invoked by the servlet container until the first GET or POST request is encountered. Added to this, the `init(..)` method cannot report any failures to a Web client, because it is not given the response object. Therefore, the system administrator should kick start the servlet with any valid request, and then check the server's error logs to make sure that the servlet was initialised successfully.

Servicing requests - The developer should always remember that the servlet may service requests in a multi-threaded environment, with different threads making reference to the same servlet objects. For `DMLDoc` objects, this is not really an issue because, once built, the DOM is effectively a read-only structure.

However, this is a live issue for the DMLDoc's `DisplayTable`: any method involved in servicing a request can update the `DisplayTable`. If you use the default DMLDoc arrangements, then there is only one `DisplayTable` per DMLDoc object. Therefore, there is scope for corruption of the Model, when two or more threads perform an interleaved update on the same `DisplayTable` object.

The solution is to synchronise service methods - such methods must perform all of the update operations on the `DisplayTable` object, and perform the `view()` method on the DMLDoc object. Once `view()` has been performed, the object can be released. An example is:

```
private synchronized String searchRsp(String first, String last)
    throws Exception
{
    DisplayTable model = searchRspDMLDoc.model;
    String nullReport;

    model.clear();
    model.appendTo("action_url", ACTION_URL);
    model.appendTo("prevFirstName", first);
    model.appendTo("prevLastName", last);
    model.register(db.query(first, last));

    nullReport = new Boolean(model.columnSize("contactId") == 0)
        .toString();
    model.appendTo("nullSet", nullReport);

    return searchRspDMLDoc.view();
}
```

So long as this is the only service method that uses `searchRspDMLDoc.model`, all should be well.

6.2 Making it all Work: **Style Guides**

The following are recommendations for the structure of DMLDoc applications and the objects that might be created in the process of servicing requests. DMLDoc in no way demands that these techniques are applied, but they should serve as a starting point:

The 'big servlet' class - There are numerous small advantages in containing all service methods for an application within the same class.

In particular, where shared resources such as database connections are concerned, opening the resource and then maintaining the resource across multiple servlet classes can be awkward and complex. Sharing such resources across multiple servlets tends to create both thread safety problems and 'memory leakage', which arises when resources are not relinquished at the appropriate time.

This should help to answer the question: *'How many different servlet classes should I define for my application?'* The answer is: *'As many as possible, taking into account that each shared resource should be contained within one class.'* So, in practice, the answer is often *'One.'*

The dispatch / service method structure - All of the examples supplied for DMLDoc use the same structure:

- `doGet(..)` or `doPost(..)` methods are implemented only when the behaviour of the `com.set_i.http.HttpServlet` methods must be overridden.
- `doGetDispatch(..)` or `doPostDispatch(..)` is implemented. This method should only contain a switch statement that invokes the appropriate service method. The `doGetDispatch(..)` or `doPostDispatch(..)` should not be synchronized, because this would provide too coarse a blocking on servlet activity. Because these methods should not be synchronized, they should not update any shared resources.

Protecting resources - Except in special circumstances - for example, where a service method only performs a task for an `init(..)` or `destroy()` method - the service method should be synchronized.

For resources such as the `DisplayTable` objects belonging to `DMLDoc` objects, this blocking technique is adequate, because there one-to-one mapping between `DisplayTable` objects and servlet instances - only methods belonging to the servlet should be able to access its `DMLDoc DisplayTable` objects, and every instance of the servlet has its own `DMLDoc DisplayTable` instances.

Some servlet containers are capable of executing only one copy of each servlet. Where deployment is targeted exclusively at such containers, synchronisation might be the end of the story as far as protecting resources is concerned. However, other servlet containers are capable of handling several clones of the servlet. The container pools these clones in order to respond quickly to requests.

In this situation, synchronisation will not protect resources that are external to the servlet instance (the `DMLDoc` objects are part of the servlet instance so they don't count here). For example, a database connection could be used by all of the clones, and 'transactional' activities could overlap. Severe failures will inevitably result.

The solution is to declare the servlet as implementing the `SingleThreadModel`. No other changes are necessary to the servlet in order to implement this interface. A servlet declared in this way will not be subject to multi-threading. This is an essential property of a servlet such as the JDBC example, where the Cloudscape database requires that all queries - including explicitly read-only queries do not overlap. Conversely, it is not essential for a servlet such as the EJB example where transactionality is guaranteed at only the essential points.

Where a servlet is declared as implementing the `SingleThreadModel`, synchronisation of service methods is irrelevant.

Form objects - A typical form may cause 10 or 20 fields to be returned to the servlet as URL parameters. Handling this data can create spaghetti code, so we suggest a generally useful technique: create a new class of object that represents the form on the server. This form object should have the following properties:

- *Fields* - Instance variables should be the same as the HTML form fields, with the same names. All of these instance variables should be of type `String`.
- *Access Methods* - There should be access methods for each of its instance variables, with each method named `getField()`, where `Field` is the name of the instance variable. There is no need to supply `setField(..)` methods.
- *Constructor* - The class needs only one constructor. The constructor takes the `HttpRequest` object as its only argument, and extracts the form data from this.

Once the class is made available, handling form data is very easy. In the following example, the service method is responding to a licence request form submission, using a `LicenseRequest` form object:

```
private synchronized String licenseCheck(HttpServletRequest req)
    throws Exception
{
    LicenseRequest lr = new LicenseRequest(req);

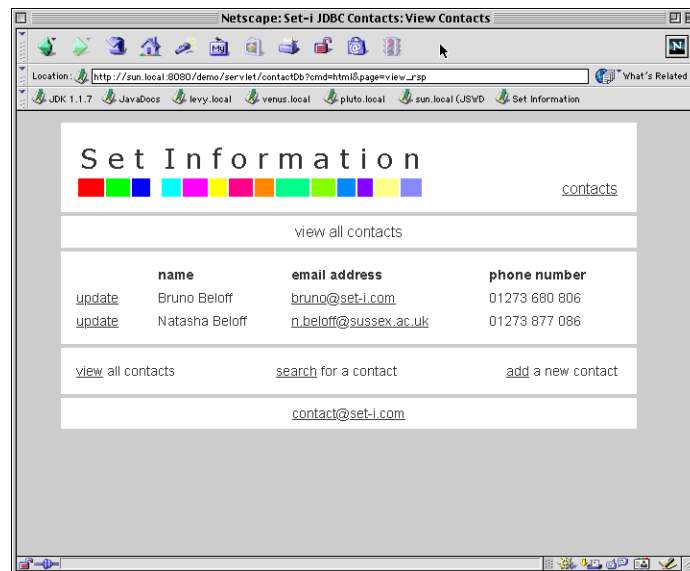
    switch (getParamIndex(HIST, req))
    {
        case HIST_NONE:
            return licenseCheckNew(lr);
        case HIST_SUBS:
            ...
    }
}
```

7

Case Studies

The following case studies look at two DMLDoc demonstration applications. These applications provide almost the same functionality - a 'Contact Book' with names, email addresses and phone numbers - but achieve this functionality with different technology. The first application uses a JDBC database connection, whereas the second uses an EJB server.

A sample of the servlet in action looks like:



This screen shot was taken from the JDBC version.

Both demonstration applications can be downloaded from the DMLDoc Web site. Instructions for installing and running the demonstrations are included with each download.

7.1 Case Studies: **JDBC Contact Book**

The JDBC Contact Book application contains two classes:

- `ContactServlet` - Provides servlet functionality. In essence, `ContactServlet` acts as an interface between the client browser and the database methods contained in the `ContactConnection` class.
- `ContactConnection` - Includes methods to establish a database connection, create tables, search for data, update records, and add new records to a Contact Book database.

In addition to the two classes, the application includes nine different HTML templates. For this application, Views tend to go in pairs, thus:

- `tpl_add_req.html` and `tpl_add_cmt.html` - Used to request that a new record should be added to the Contact Book, and report that the record was added successfully.
- `tpl_search_req.html` and `tpl_search_rsp.html` - Used to request a search with certain criteria, and report the results of the search.
- `tpl_update_req.html` and `tpl_update_cmt.html` - Used to request that an existing record should be updated, and report that the record was updated successfully.

Other Views are supplied, these cover an opening page (`tpl_index.html`), a view all records page (`tpl_view_rsp.html`), and a page reporting that a search resulted in no records being found (`tpl_null.html`).

At runtime, the application must be able to locate the templates, the location in the local file system is specified by the servlet's class variable `TPL_PATH`. This variable is assigned a value by `setConfig()`.

The servlet requires its own configuration file; this will be read by `setConfig()`. The contents of the file, as supplied are:

```
TPL_PATH          /export/home/bruno/jbproject/demo_jdbc_011/html/
ACTION_URL        http://sun.local:8080/demo/servlet/contactDb
JDBC_URL          jdbc:cloudscape:
```

All three values will most likely have to be changed if the application is to run successfully in the new environment: `TPL_PATH` gives an absolute file path - including the final separator - for the HTML templates. `ACTION_URL` gives the URL for hyperlinks in each View to find the servlet. The class variable `JDBC_URL` identifies the database driver. If a Cloudscape database is used 'in process' then this value does not need to be changed.

In order for the configuration file to be found by `setConfig()`, the file must be made available to the servlet container. The easiest way to do this is to place the configuration file in the same directory as the `ContactServlet.class` and `ContactConnection.class` files. Alternatively, you can use a directory structure such as `conf/com/set_i/demo/jdbc/ContactServlet.conf`, and include the `conf/` directory on the runtime classpath.

The split between the two classes - `ContactServlet` and `ContactConnection` - is there in an attempt to have only business logic in the servlet class. DMLDoc enables the servlet to dispense with presentation code, while `ContactConnection` enables the servlet to dispense with database access code. We could make the servlet's service methods even simpler, by supplying a class that represented a Contact Book 'form object' - Section 6.2 explains how this can be done.

In order to run the JDBC Contact Book example, you must add the database driver, the dml doc package, and the dml doc license to the servlet container's class path. You also need to configure the servlet container to map requests to `contactDB` to the class `com.set_i.demo.jdbc`, and give the container access to the compiled application classes. The [DMLDoc Installation Guide](#) gives details on how this should be done.

The JDBC Contact Book application accepts two parameters. The use of these parameters can be taken as a general guide for a servlet interface. The parameters are:

- `cmd` - The mode in which the application should respond, together with some housekeeping options:
 - `html` - The application should respond using the HTML templates supplied with the application.
 - `debug` - The application should respond using the `asDMLDoc()` method. This will generate a presentation using the data associated with the preceding presentation of the same page. If a presentation has not yet been generated for the given page, the data set will be empty.
 - `init` - Causes the servlet's `init(..)` method to be reinvoked. The application's home page is returned as the response. An option such as this is *essential* in order to allow new templates to be reloaded, without stopping and restarting the server.
 - `create_db` - Creates the database tables. This option causes an exception to be thrown if the tables are already in place. The command should only be executed once, the first time the application is used.
 - `halt_db` - Causes the database driver to be shut down cleanly. The database can be restarted by issuing a `cmd=init`. A real-world application should make this command password-protected.
- `page` - The function that the servlet should perform. There is normally a one-to-one mapping between these servlet functions and service methods in the servlet class.

7.2 Case Studies: **EJB Contact Book**

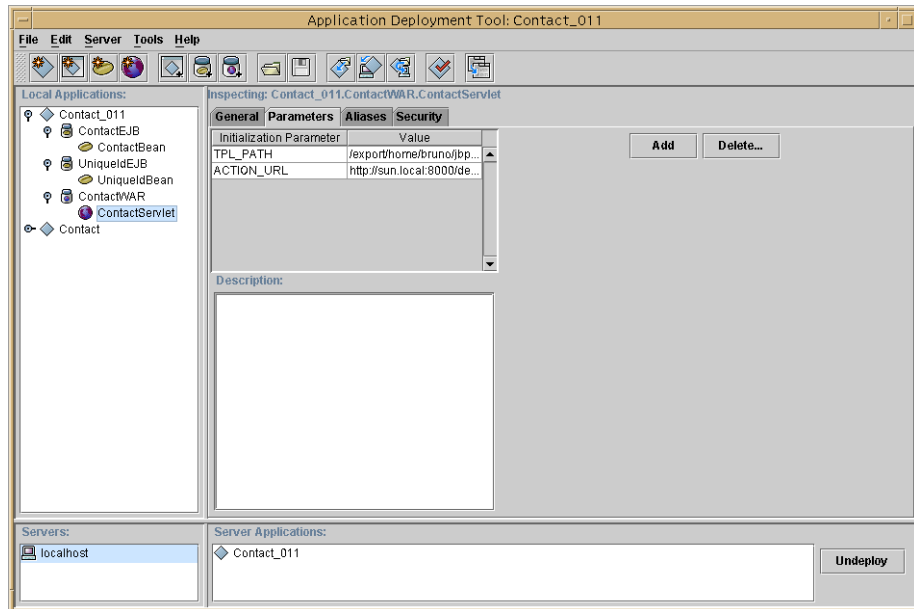
The EJB Contact Book demonstration is a three-tier application. All three tiers can be deployed on the same host, or they can be split up over three host machines.

Aside from its Web browser client, the application contains three components:

- *Web Component* - This contains a servlet - perhaps confusingly, it is also called `ContactServlet` - that mediates between client requests and EJB accesses. The HTML templates used by the servlet are more-or-less identical to those used by the JDBC demonstration.
- *Contact Book EJB* - In this demonstration, Contact Book entries are each implemented as EJBs. This approach greatly simplifies the data access logic of the application, and solves some important transaction and synchronisation problems.
- *Unique Id EJB* - Although many relational databases are capable of generating unique keys for records, we decided not to rely on that feature being available. The Unique Id EJB is capable of generating unique keys for any number of different middle-tier or back-end application components. In this application, it is called upon to generate unique Ids for Contact Book entries.

Unlike the JDBC demonstration, the EJB example does not make use of a configuration file. There are two reasons for this: firstly, there is no standard way of introducing a configuration file to the Web Component deployment package. Secondly, an EJB server's deployment tool should provide a simple and reliable way of setting initialisation parameters that can be saved along with the rest of the deployment specification.

For the Sun J2EE deploytool, the initialisation parameters are set as follows:



Here, the parameter `TPL_PATH` is set to `/export/home/bruno/jbproject/...` and the parameter `ACTION_URL` is set to `http://sun.local:8000/demo/...`. The parameters are accepted in the servlet's `init()` method with:

```
TPL_PATH = getInitParameter("TPL_PATH");  
ACTION_URL = getInitParameter("ACTION_URL");
```

Appropriate settings for your host and file system must be given, otherwise either the templates will not be found, or the servlet will not be found.

The architecture of the application is such that the servlet need not be concerned about data persistence or transaction issues. However, the deployment process must be. Specifically, the deployment tool must be used to specify:

- Each of the three application components, in terms of the classes and interfaces used to represent them.

- The SQL code used to find the persistent objects. The deployment tool can go some way in generating this code, but the code must be completed by the developer.
- The relationship between the application components. In other words, which components refer to which other components, and by what names.
- The aliases given to the application and its Web component, in order that URLs can be mapped to servlet accesses.

All of these aspects of the EJB Contact Book application deployment properties are covered in more detail in the [DMLDoc Installation Guide](#).

In order to run the EJB Contact Book application, two processes must be started. The first process is the database driver. Unlike the JDBC example application, here the database driver is expected to run out of process from the servlet container. The second process includes the servlet container and EJB container. It is likely to also contain the Web server.

A third process - providing the deployment tool - must also be run in order to deploy the application. The deployment tool can normally be shut down once the the application has been successfully deployed.

The EJB Contact Book application accepts three parameters. The options here could be expanded to include, for example, different languages or different technologies - supported by different sets of templates - such as XML and WAP. The current options are:

- `cmd` - The mode in which the application should respond, together with some housekeeping options.
 - `html` - The application should respond using the HTML templates supplied with the application.

- `debug` - The application should respond using the `asDMLDoc()` method. This will generate a presentation using the data associated with the preceding presentation of the same page. If a presentation has not yet been generated for the given page, the data set will be empty.
- `init` - Causes the servlet's `init()` method to be reinvoked. The application's home page is returned as the response. An option such as this is *essential* in order to allow new templates to be reloaded, without stopping and restarting the server.
- `page` - The function that the servlet should perform. There is normally a one-to-one mapping between these servlet functions and service methods in the servlet class.
- `order` - The sort order for any relevant View. This application sorts the most recent Model for the appropriate page, rather than going back to the EJB server.

Two important notes should be given concerning the structure and function of this application:

The `order` parameter function is implemented assuming a single-user environment - this is a dangerous thing to do! We did this to keep the code as simple as possible, and to show the difference in performance between sorting a `DisplayTable` and returning to the EJB server to gain a new set of data. In a real-world application - if it was considered important to sort data cached by the servlet - that data should be attached to the user's session.

The `init()` method makes a new connection to the EJB server every time it is invoked, irrespective of whether it already has a connection. The servlet itself makes no concessions as to whether multiple threads make use of a servlet instance. In this case, this is fine, because the EJBs do not carry any session information. Session Beans must be handled differently.

A **References**

A.1 **References: Software**

J2SDK - DMLDoc has been tested with a number of different versions of the J2SDK 1.2. The most recent version is 1.3.

DMLDoc should work properly with the 1.3 version, but it has not been tested with 1.3 yet. A suitable Java Development Kit (JDK, now known as the Java 2 Platform, Standard Edition or J2SDK) can be downloaded from:

<http://java.sun.com/j2se/>

JSWDK - The Java Servlet Web Development Kit (JSWDK) provides a suitable servlet container. (DMLDoc applications should work with any standard servlet container.) The JSWDK 1.0.1 is available from:

<http://java.sun.com/products/jsp/download.html>

J2EE - The Java 2 Enterprise Edition (J2EE) provides a Web server, servlet container and EJB container, and database. The J2EE was used to develop the DMLDoc EJB features. This software can be downloaded from:

<http://java.sun.com/j2ee/>

Cloudscape - The J2EE database is the Cloudscape 3.0 database, owned by Informix. The Cloudscape 3.0 database was used to develop the JDBC-related features of DMLDoc. The latest version (Cloudscape 3.5) can be downloaded from:

<http://www.cloudscape.com/>

A.2 References: **Further Reading**

We found the following books useful in explaining both the overall pattern of servlets, and some of the gory details:

Arnold K. and Gosling J.

The Java Programming Language, Second Edition

Addison Wesley, 1998

ISBN: 0-201-31006-6

Web: <http://java.sun.com/docs/books/javaprogram/secondedition/>

Berg C. J.

Advanced Java Development for Enterprise Applications, 2nd Edition

Sun Microsystems Press, 2000

ISBN: 0-13-084875-1

Web: <http://www.ajdea.com/>

Campione M., Walrath K., Huml A. *et al*

The Java Tutorial Continued

Addison Wesley, 1999

ISBN: 0-201-48558-3

Web: <http://www.java.sun.com/docs/books/tutorial/index.html>

Williamson A. R.

Java Servlets by Example

Manning, 1999

ISBN: 1-884777-66-X

Web: <http://www.browsebooks.com/Williamson/>

B **DMLDoc JavaDocs**

The complete set of JavaDocs for all versions of DMLDoc are available from the Set Information Web site, at:

<http://www.set-i.com/products/dmldoc/javadocs/>

This set of JavaDocs includes the following packages: `com.set_i.dmldoc`, `com.set_i.http`, and `com.set_i.license`. Note that the package names use `set_i` (with an underscore) while the Web site uses `set-i` (with a hyphen).

The JavaDocs for DMLDoc Standard Edition, Version 1 are also available online, at:

<http://www.set-i.com/products/dmldoc/1.1/doc/api/>

If you are writing JavaDocs for an application that makes use of DMLDoc, you may want to link your JavaDocs to this online version. The above location contains a package-list file, so the javadoc compiler will be able to link to DMLDoc classes and methods. Do this with:

```
javadoc ... -link http://www.set-i.com/products/dmldoc/1.1/doc/api/ ...
```

When new versions of DMLDoc are released, the old versions' JavaDocs will be retained on the Set Information site, at their original URLs.

C **Contact Details**

DMLDoc is developed and distributed by:

Set Information Limited

7 Wyndham Street
Brighton
East Sussex BN2 1AF
United Kingdom

Tel: +44 1273 680 806

Fax: +44 1273 699 224

Email: contact@set-i.com

Web: <http://www.set-i.com/products/>

Set Information is registered in England, No. 3851469.

No formal support arrangements are available for the Standard Edition of DMLDoc. The software, like the documentation, is supplied *as is*, with no warranty whatsoever as to the suitability of the product for any task. Set Information Limited cannot accept any liability or consequential liability for the operation of DMLDoc, or its possible non-operation.

Legalese aside, *Set Information 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.

If you are using DMLDoc for a specific project, please let us know - your experiences will most likely be useful to future developers.