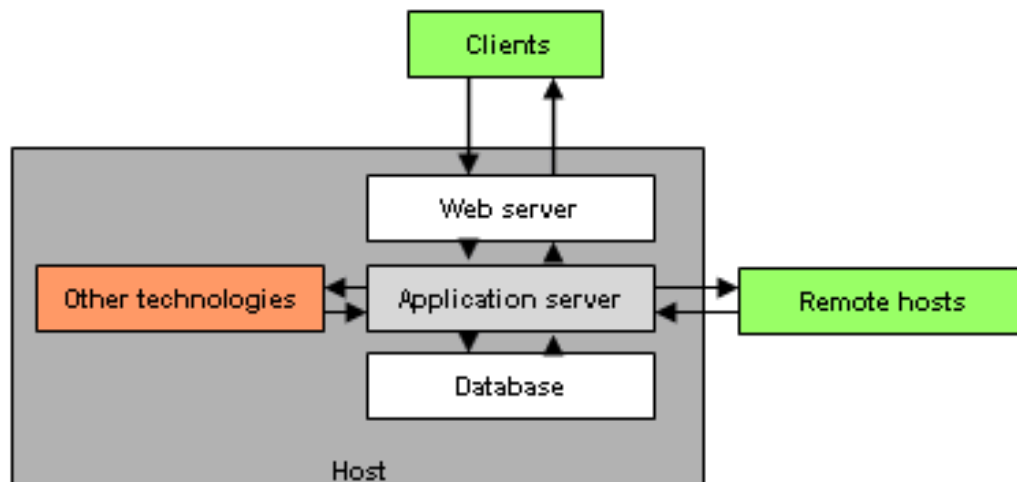


INTRODUCTION TO DEVELOPMENT OF DYNAMIC WEB APPLICATIONS

Svein Nordbotten



Svein Nordbotten & Associates

Bergen 2007

Table of Contents

Preface	7
Session 1: Introduction	8
Web-pages and web sites	8
Dynamic web sites	9
CGI and PERL	10
Applications Program Interfaces.....	10
ColdFusion Markup Language.....	11
Evaluating a web site	11
Visit counter example	12
Exercises.....	13
Session 2: ColdFusion MX	15
ColdFusion Markup Language.....	15
Data types	15
Variables.....	16
Expressions.....	16
Flow control statements	18
Other components	19
Guessing game example	20
Exercises.....	23
Session 3: Web market research	24
Introduction	24
Market research.....	24
Applications outline	24
Implementation	25

Exercises.....	31
Session 4: Web database for opinion polls.....	33
Files and databases	33
Databases.....	33
CFMX SQL.....	34
Basic SQL elements	34
Opinion polling.....	35
INSERT statements.....	37
SELECT statements.....	40
UPDATE statements.....	41
Statistical report.....	45
DELETE statements	47
Exercises.....	49
Session 5: Web perception application	50
Introduction	50
Outline of the design.....	51
Design of the database	52
Design of the dynamic web process	53
Opening page	53
Demographic form	54
Displaying the first chart	58
Perception report form.....	61
Next chart.....	64
Thanks to the participants	67
Exercises.....	68
Session 6: Web search engine.....	69
A search engine.....	69
Selection menu	70
File collection	71
Indexing the files.....	73
Searching the collection.....	75
Deleting a registered collection	77

Final remarks.....	79
Exercises.....	79
Session 7: e-learning	80
Web courses.....	80
Course architecture.....	80
Authorization and authentication.....	81
Registration and authorization	82
Authentication	86
List of content	87
Lectures.....	88
Session 8: Web shop	91
e-business	91
Business promotion	91
Buying products	94
Purchasing products.....	101
Exercises.....	103
Session 9: Web agents	104
Web agents	104
Agent 1	104
Advanced agent	107
Remarks on scheduling	113
Other Internet Agents.....	114
Exercises.....	114
Session 10: Data exchange - syndication	116
XML	116
Web Distributed Data Exchange	116
Data exchange between art galleries.....	117
Implementation of the exchange.....	118
DTD for WDDX.....	123
Exercises.....	123
Session 11: Regular expressions and CFScript	125
Regular expressions and string processing	125

Re-visiting the search engine	125
Implementation	130
CFScript language.....	131
Comparing CFML and CFScript.....	131
Conclusion.....	133
Exercises.....	133
Session 12: Re-using code.....	134
Re-using code.....	134
ColdFusion approaches.....	134
Custom Tags.....	134
User-Defined Functions	137
UDF libraries.....	138
CFX Tags	139
Exercises.....	139
Session 13: Distributed processing	140
Distributed processing	140
Client-side processing	140
JavaScript	140
Flash and ActionScript.....	142
Exercises.....	149
Session 14: Components.....	150
CFMX Component technology	150
Authorization example.....	150
Component for generating unique random numbers	150
Introspection.....	153
Exercises.....	154
Session 15: Web services	155
Web services	155
<i>Universal Description, Discovery, and Integration</i>	156
<i>Web Service Description Language</i>	156
<i>Simple Object Access Protocol</i>	156
<i>Web services creation and consumption in CFMX</i>	156

<i>A web service example</i>	157
<i>About the implementation of the example</i>	158
Exercises	158
A bibliography for further studies	159

Preface

This publication is an extract from an interactive online course given at the University of Bergen, the University of Hawaii and a few other organizations in the period from 2001 to 2007. The purpose was to provide the students with knowledge of and training in the use of methods available in ColdFusion suite of tools from Allaire-Macromedia-Adobe.

The course were provided with a number of interactive components as quizzes, search tool, virtual classroom, communication channels, control of participants' performance and tools for the teacher. Even without these interactive components the course was thought to be of interest and is offered readers in the present reduced form.

Bergen 2007.

Svein Nordbotten

Session 1: Introduction

Web-pages and web sites

The **Internet** was initiated in the 1970's as a further development of the **ARPANET**. The **World Wide Web**, **WWW**, was developed and introduced in 1989 by Tim Berners-Lee and Robert Cailliau at the European Particle Physics Laboratory (CERN) as an Internet tool for collaborative knowledge-sharing. It became in short time very popular. Today, the **WWW** comprises a large number of computers that make files available through the Internet according to the HyperText Transfer Protocol, **HTTP**. Today, it is estimated that more than **250 M people worldwide** are using the web.

The visible content of a web file is called a **web page**. If a web page is prepared according to the **HTTP** protocol, it can be transferred from a host computer using appropriate software to a requesting client through the Internet. Most pages are prepared by means of the tag-based language **HTML**, frequently supplemented with some additional tools. If the requesting client has the necessary **browser** software installed, the file received can be displayed and, if wanted, a new request can be generated.

A **web site** is usually a set of web files hosted by a computer running a **web server**. Design and implementation of a web site has several aspects:

- the **content** embedded in the site
- the **page** sent from the site
- the **functionality** of the site

The topics of a web site are varying and depend on the owner's interests and mission. In this course, we will **not** in this course discuss which appropriate pages for web publication are, and which are not. Examples of both interesting and less interesting pages can easily be found.

The layout of pages is a fascinating subject. All kinds of **backgrounds colors** and **patterns, fonts** of different kinds and sizes, etc., are among the layout factors from which the designer may choose. Some pages have **animation** or **voice** embedded, and using programs transferred to and acting in the client computer. The layout of a page is an important subject because it has a significant impact on how the receivers will perceive the page. So far, the layout has to a large extent been determined by the latest hypes and layout advises, and the heuristic design rules offered have been based on opinions and **limited empirical facts**. Large scale investigations of people's perception of alternative layouts are needed. However, layout is **not** the subject of this course either.

The subject of this course is the **functionality** required to change the web arena from basically **static** to **dynamic** applications. The required functionality is the web site's ability to react on a visitor's behavior over a shorter or longer time period expressed by a series of requests and responses. It is called dynamic because the web pages returned to the client depend on the visitor's previous interaction.

Dynamic functionality can be approached in many ways. In this course, we limit our discussion to the functionality based in the server and disregard other approaches.

Dynamic web sites

The basic model of the web interaction is:

1. A set of **pre-developed** static web pages are stored on a host server.
2. A user sends a **request** for a web page to the host.
3. The host sends a **copy** of the requested page to the client.
4. If desired, points 2 and 3 are **repeated** for new pages.

The service that manages the host as a node in the web is called a **web server**. In the static model, [Figure 1](#), the host has **no ability** to analyze the request and adjust the response accordingly. The **response** is a requested pre-designed web page. The request-response exchange is therefore called static. However, the exchange protocol used, **HTTP**, provides possibilities for some **additional** items of information sent with the **request** without any instructions from the

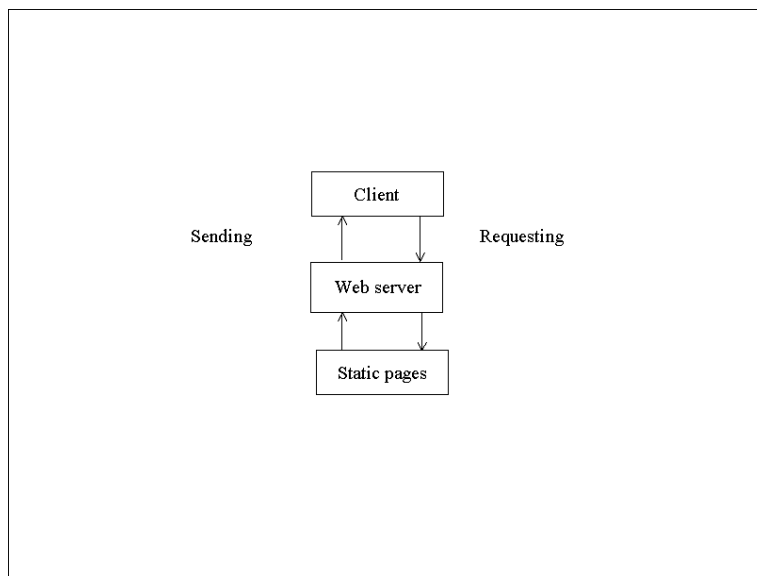


Figure 1: Basic web server model

requester. In the same way, the responding host can include additional information with the **response**, usually hidden for the receiver. The host also has capabilities for forwarding messages to other programs beyond the web server for additional processing. These possibilities for information processing **behind** the scenes make it possible to create the additional functionality.

We shall use the term **dynamic web site** to emphasize that we are not concerned with a simple set of web pages with **HTML** tags, but with applications in which the pages returned to the client can be dynamically adjusted to fit the individual requests of the client. This course can serve as a first illustration of a dynamic web site. You have already experienced in this course that when you submitted your **personal access code**, the system became accessible to you. If you had

submitted an invalid access code, the host would instead have sent you a message adjusted to an unacceptable access code. The site must know your identity prior to your request. You will soon also see that if you try to go on to the **next session** before it is officially opened, you will receive a message informing you that the session is not yet open. However, when the opening **date** is passed, and you have passed the **test** at the end of the previous session, the system will respond by giving you access to the session. The system must be able to compare your request with its **clock time** and with your recorded test performance. If a student has not yet completed the required test, the host will return a message saying that the test must be done before the student can proceed.

Important characteristics of the dynamic web site are the ability to **authenticate** you, i.e. to verify your identity, **record** your performance history, keep track of your **interactions** from when you start a session and until you leave, and sometimes even from session to session. In this course, you can for example request your personal progress report, and the system will generate the content of the report to you while another student will get a completely different report and perhaps for a different number of sessions.

CGI and PERL

The first step towards dynamic web pages is the possibility for a remote client to request the execution of a process at the host. Use of the **FORM** tags of **HTML** requires for example that the server can perform a processing of the data submitted on the form. A program must exist for this purpose at the host site, and the web server must be able to communicate with this program. We will refer to such a program as a **script**, and the addressable files in which the script is stored as **templates** to avoid any confusion with other types of programs, files, and pages.

The **Common Gateway Interface**, **CGI**, is a protocol specifying how certain scripts can communicate with web servers. One of the most frequently used tools for creating such scripts is the scripting language **PERL**. A **PERL** script stored in the host computer can be supplied with data from a request, for example sent by an **HTML FORM**. The script can be designed to perform a variety of tasks such as saving and retrieving data from a database, updating a log, keeping track of visitors, running a course, etc. It can also be designed to perform its task and then leave the result to the web server, which returns a web page generated by means of the script to the requesting client. Programming languages such as **C**, **C++**, **C#** and **JAVA** can also be used for creating scripts. One reason for the popularity of **PERL** is that scripts programmed in **PERL** can be ported from one **operating** system to another with little or no modification.

Applications Program Interfaces

A **PERL-CGI** application is time-consuming because **PERL** scripts must be loaded, executed, and unloaded **each time** they are used as interpretive programs, and do not offer the flexibility that may be required.

To improve this situation, **Web Application Interface Servers** were developed. An application server is a **service** operating behind the web server. It processes script code, which the web server does not understand, and returns the results to the web server for sending to the requesting

client. The applications server is a resource of permanently loaded executable programs. The resource of loaded programs for **WINDOWS** operating systems usually written as **Dynamic Load Libraries**, **DLL's**, is referred to as an **Applications Program Interface**, **API**. The benefit of using an **API** is **increased** speed and flexibility because no loading and interpretation is needed. The **disadvantage** is that the **API** programs must be implemented **specifically** for each type of operating system, and require more **memory** space.

ColdFusion Markup Language

The most well-known **API** tools include the **Active Server Pages**, **ASP**, and **ASP.NET** from Microsoft, the open source system **PHP: Hypertext Preprocessor**, **PHP**, **Inline Internet Systems**, **iHTML**, and **ColdFusion MX**, **CFMX**, from Macromedia. We are leaving the comparisons between the tools to evaluators and sales people, and concentrate on **CFMX** in this course because it is well developed, easy to learn, and reliable.

The language in which we design our scripts for **CFMX** is the **ColdFusion Markup Language**, **CFML**. The templates are recognized by their extensions, **.cfm** (or **.cfml**). You are referred to the section [Software](#) to get instructions for installing necessary software on your own **PC**.

In the previous paragraph, the advantage of using a web **API** instead of a **CGI** approach was emphasized. **CF** was introduced in 1995. It started out as a **scripting language based on CGI**. Later, the **API** was developed. The latest version of **CFMX** is implemented in **JAVA** resulting in a very efficient and portable **API** that can easily be extended by **JAVA**.

CFMX is widely used. Among the well-known companies that have taken advantage of **CF** in the development of their web sites are [Amazon](#), [Dell Computers](#), and [Federal Express](#). For an up-to-date list see [Ben Forta](#).

Evaluating a web site

Before starting to improve a web site, you should try to make an evaluation of its performance. **Evaluating** a web application requires empirical data. The most obvious source is the **log** of the activities of the web server. The first step towards collecting data on the use of pages was **counting** the number of visitors to the web site. The number of visitors tells the owner of the site if his/her site was visited at all, and how frequently. In most web servers, an **access log** system is embedded. The access log system continuously records all requests to the server as well as the server's retrieval of different files to compose the responses. A log, even for a completely static web site, gives data on **dynamic development** because it reflects visits during a time interval.

The server's **access log** usually records according to one of several formats:

- Old NCSA/CERN format
- Combined NCSA/CERN format
- Windows format

The **Windows** format can easily be imported into applications such as **EXCEL** and **ACCESS**. It contains the following fields:

- Date and time
- IP address of client
- Address of server
- HTTP method of request
- Requested URL
- Referring URL
- Browser type
- HTTP response code
- Number of bytes transferred
- Milliseconds between arrival of request and log recording

Typical studies of this kind can be inspected in [Nordbotten & Nordbotten 2001a](#) and [Nordbotten & Nordbotten 2001b](#).

There are several obvious **drawbacks** associated with the access log for web applications research. **First**, the access log is part of the server software guarded by the web server's administrator group, and the researcher may **not** always get access to this log. **Second**, the access log records **all** file retrievals necessary to assemble the requested page to the client including icons, images, etc. which are not needed for an application evaluation. **Third**, the log system only recognizes the client machine identified with an IP number. Many users are assigned **different** IP numbers by their net provider from one visit to the next. **Fourth**, there are often several users sharing a client computer.

Frequently, a more **customized** recording than that provided by the web server application log is needed for a satisfactory evaluation. Later in this course, we shall discuss how you can set up your own log to avoid these problems. At this stage, we start with a very simple example.

Visit counter example

Our first application example is a personalized course visit counter. It demonstrates a few properties of a dynamic web site. A direct link to the implemented example is located at the end of this session. The example application keeps track of the number of visits you have made to this course since you started your study.

The application script consists of 2 templates. The template used as for entering an example is by convention called ***index.cfm*** to avoid unwanted public listing of your templates. This template personalizes the response to your call for the example and remembers the number of visits.

1. `<!-- index.cfm -->`
- 2.
3. `<cfquery name="visits" datasource="db">SELECT firstname, visits FROM users WHERE
accesscode='#session.pin#'`
4. `</cfquery>`
5. `<cfoutput>`

6. `<div align="center">`
7. `<h1>#visits.firstname#'s site visit counter</h1>`
8. `<p>#visits.firstname: The number of visits you have made to this course site until
#TimeFormat(Now())#, #DateFormat(Now(),'mm/dd/yy')# is: <font
color="Blue">#visits.visits#</p>`
9. `</div>`
10. `</cfoutput>`

All **CFMX template** files are identified by the extension *.cfm*. Line 1 is a **comment** line that is used here to refer to the name of the template. The comment tag is similar to the **HTML** comment tag, but has 3 '---' while the HTML comment has 2. Except for this tag, all ColdFusion start and end tags begin with **CF** or **/CF**, respectively.

At this stage, we will postpone the detailed discussion of the **CFMX** tags, and limit the explanation to the more general issues. In the listing, you may recognize some **HTML** tags that can be **intermixed** with **CFMX** tags.

Lines 3 - 4 illustrate the **CFQUERY** tags ColdFusion uses to **exchange** data with a database. The statement between these tags is a regular **SQL SELECT** statement used to retrieve the user's **first name** and the number of recorded **visits** from a database referred to as **db**. In other words, this application assumes that the users are recorded in a database and that their numbers of visits are updated in the database each time they log in. The term *'#session.pin#'* refers to the string you submitted as your access code.

Lines 5 -9 output the query result from the database. If you are acquainted with **HTML**, you will recognize most of the content between the **CFOUTPUT** tags.

There is one more point that you should observe. That is the *#visits.firstname#* and *#visits.visits#*. The surrounding #'s indicate that we refer to the value of the included named variable, in this case *firstname* and *visits*, respectively, in the retrieved query object *visits*.

In the next session, the syntax of the **CFML** will be discussed in more detail.

Exercises

- a. Visit the web sites of [Amazon](#), [Dell Computers](#), and [Federal Express](#), and spend some time to study what these pages can provide. Make a short report for yourself containing ideas to use later in this course.
- b. Get acquainted with the web site of this course. The **Calendar** is important, and it is suggested that you print it out for easy consultation. In the **FAQ** section, you will find useful information about the course. Consider the course design and structure, and suggest changes and improvements to the author (svein@nordbotten.com).
- c. The text book used, **Programming ColdFusion** by Rob Brooke-Bilson, will be referred to as **RBB**. This book is not a novel you can read sequentially. You should use it as a **manual**, and

read about the task you are currently working on. Read **Chapter 1** of RBB before you start on the next session.

Session 2: ColdFusion MX

ColdFusion Markup Language

The **CFML** is the tool by which we express our ideas, conditions, and goals for the applications in this course. **CFML** is a **tag-based scripting language**. As in any other programming language, **CFML** has its own syntax, but can be intermixed with **HTML**. As a scripting language, it has borrowed a number of concepts from **PERL** and other programming languages.

A **CFMX script** is stored in one or more files with the extension **.cfm**. These files are referred to as **CFML templates**. The extension makes it possible for the web server to identify which requests it should pass on to the **CFMX** application server. No harm is done if a usual **HTML** page is named with the **.cfm** extension, but a **CFML** template with an **.htm** or **.html** extension will not work. In this course, we use the **.cfm** extension for all templates.

The first **difference** you will observe between **HTML** pages and **CFML** templates is the **CF** tags permitted in the latter. All **CFML** tags starts with the 2 letters **CF**. Most **CF** tags come in pairs with a starting and an ending tag with the form **<CF..>** and **</CF..>**, respectively.

When you try to **view the source** of a page generated by **CFML** templates by clicking the **View/Source** option in your browser, you will **only** see the **HTML** source sent for display. The **.cfm** template, which generated the **HTML** display, will not be available for the client.

CFML has the following components:

- Data types
- Variables
- Expressions
- Control statements
- Other components

In this session, we will **review** each component. This session is a brief summary of **CFML**. For more **details** and **precise** descriptions, you are referred to **RBB**. Special topics will be discussed in each session and illustrated by **implemented examples** that you can run, copy, modify, and try on your own **PC**.

Data types

As in any language, **CFML** supports a set of different **data types**:

- Boolean
- Strings
- Numbers
- Date/time
- Lists
- Arrays

- Structures
- Query objects
- Component objects

Any data type variable can be assigned a value directly or by special functions in a **CF** tag.

Variables

There are 3 aspects associated with a **variable** that you should keep in mind:

- Name
- Value
- Scope

The variable **name** identifies the variable, and must begin with a letter, contain no spaces or special characters, and should not be identical to reserved names in **CFML** or **SQL**. The **value** is either numeric, logical, or string. A string is enclosed by single or double **quotes**. The **CFML** variables are all **typeless** which means that you can assign a value of any type to any variable. However, the use of a variable can be restricted by the data type of its value.

A variable is most frequently defined by the **CFSET** tag:

```
<CFSET variable_name="xxx">
```

The value you substitute for **xxx** in the tag is the assigned value of the variable here symbolized by **variable-name**.

The **value** of a defined variable can be obtained by surrounding the variable name with #, i.e. **#variable_name#**. This can be used in another set tag, for example for counting:

```
<CFSET new_variable=#variable_name# + 1>
```

The **scope** of a variable is in general limited to the template in which it is set. As we shall see later, the variables are frequently given **wider** scopes.

Expressions

In **CF**, an **expression** is a construct in which data are acted on by different operators. By means of an expression, the result of operators acting on variables can be evaluated. There are 2 categories of operators:

- Basic operators
- Functions

There are 4 types of **basic** operators:

- Arithmetic

- Comparison
- String
- Boolean

Well known **arithmetic** operators are +, -, / and *, exemplified by:

```
<CFSET evaluation_variable=(#new_variable# +2)/5>
```

where **new_variable** has already been assigned some value by previous operations. The result of an arithmetic expression is a new value assigned to a numeric variable, in this example named **evaluation_variable** with value **#evaluation_variable#**.

The **comparison** operators require 2 values and give a **Boolean** result, 0 (false) or 1 (true). Some of the most frequently used comparison operators are **EQ**, **NEQ**, **GT**, **GTE**, **LT** and **LTE**. The following expression **illustrates** use of a comparison operator:

```
<CFSET x=#new_variable# EQ 3>
```

where **#x#** will be a Boolean variable assigned the value **false** (0) if **#new_variable# NOT EQ 3**, or **true** (1) if **#new_variable# EQ 3**.

A string is a sequence of symbols. The **concatenation** operator, **&**, is used for concatenating 2 strings. There are also 2 operators for **string comparison**, **CONTAIN** and **DOES NOT CONTAIN**, by which 2 strings can be compared. The result of the concatenation is a string, while the 2 comparison operators result in Boolean values.

In the group of **Boolean operators**, the usual logical operators **NOT**, **AND**, **OR**, **XOR**, **EQ** and **IMP**, are available. An expression with logical operators gives a Boolean result.

Functions are advanced operators with pre-defined actions on data values. There are a number of **different types** of functions built into ColdFusion:

- Array functions
- Date/time function
- Decision/evaluation functions
- Encoding/encryption functions
- File/directory functions
- Formatting functions
- International functions
- List functions
- Mathematical functions
- Miscellaneous functions
- Query functions
- Security functions
- String functions
- Structure functions
- Undocumented functions

You are referred to **RBB**, Appendix B, and the literature for **details** about the individual functions available in **CFMX**.

Flow control statements

Flow control and **decision making** require control statements. **CFMX** has 3 different types of **conditional control** statements:

- If/else statements
- Switch statements
- Loop statements

The syntax for the **two-way branching** if/else statements is:

```
<CFIF "logical_expression EQ True">
```

```
execute block 1
```

```
<CFELSE>
```

```
execute block 2
```

```
</CFIF>
```

The logical expression can for example be **#y# GT #x#**.

The **multi-way** branching switch statement complex has the following syntax:

```
<CFSWITCH EXPRESSION="expression">
```

```
<CFCASE VALUE="value1">
```

```
execute block1
```

```
</CFCASE>
```

```
<CFCASE VALUE="value2">
```

```
execute block 2
```

```
</CFCASE>
```

```
<CFCASE VALUE="value3">
```

```
execute block 3
```

```
</CFCASE>
```

....

</CFSWITCH>

The expression is assumed to be valid, possible to evaluate, and producing a value compatible with the values in the **CFCASE** tags.

There are several **types of loops**. We illustrate the loop statements with the simplest, often referred to as the **FOR** loop:

<CFLOOP INDEX="LoopCount" FROM="start_no" TO="end_no">

execute block

</CFLOOP>

#LoopCOUNT# will be incremented by 1 for each loop started. It starts with the value assigned to **start_no** and end with the value specified for **end_no**.

Other components

There are **other** language components in **CFML**, the most common of which are:

- Output
- Include
- Comments

The output tag permits displaying the results from **CFMX** operations, for example the results from a database query:

<CFOUTPUT QUERY="query_name" MAXROWS="max" GROUP="group" STARTROW="row">

Text

</CFOUTPUT>

where **query_name** is a name given in a previous query tag, max is the maximum number of rows wanted, and row specifies the row from which to start output. This tag generates a display of the content of a collected query object.

The **CFINCLUDE** tag permits reference to another **CF** template. The syntax is simple:

<CFINCLUDE TEMPLATE="template_name">

where **template_name** is the absolute **URL** or the relative address to the template. This tag functions similar to a call to a subroutine.

CFML also includes a tag for non-executable **remarks**. It is similar but not identical to the **HTML** remark tag:

```
<!-- text --->
```

The three dashes **and** a blank before, and a blank **and** three dashes following the text are required, in contrast to the **HTML** remarks with two dashes before and after the text.

Guessing game example

This example, as all examples in this course, is implemented for you to try out. For **running** the example, a link is provided at the end of the session. The example is a simple **guessing** game. You will be asked to guess the sum of all integers from 1 and up to a number that is generated randomly. When you submit your answer, you will either get a feedback confirming a **correct guess**, or a message that your **guess** was wrong together with the correct answer. I recommend that you start by **running** the example a couple of times (use the **Back** button to return to the start and refresh the display). When you feel acquainted with the game, and then proceed to the study of the templates below.

All applications must have an *Application.cfm* template. The purpose of this template is to specify properties, which are valid for all other templates of the application. The application name is a typical example. All *Application.cfm* templates must include the **CFAPPLICATION** tag, but can also include other tags. The *Application.cfm* of the guessing example is very simple:

```
1. <!-- Application.cfm --->
2. <CFAPPLICATION NAME="Guess"
SESSIONMANAGEMENT="YES"
SESSIONTIMEOUT=#CreateTimeSpan(0,0,30,0)#>
```

The first statement is giving the template name within a comment tag. Note that the comment is different from the comment you may know from **HTML**. The second statement is the **CFAPPLICATION** tag which in our present example has only 3 attributes, the **NAME** that specifies the name of the application, the **SESSIONMANAGEMENT** that says that the application must remember session variables, and finally, **SESSIONTIMEOUT**, that specifies the length of the time in days, hours, minutes and seconds the system must remember these variables if the user is inactive. This template can also include **other** tags as we will see in later sessions.

The game problem itself is a very **simple** dynamic application containing only 2 display templates of which the second **depends** on the information submitted from the first and makes the application dynamic. The **first** template is named *index.cfm*, which eliminates the possibility that uninvited visitors are browsing the content of our folder.

The task to be solved by this template is to **present** the game for the user and **collect** his/her name and guess by means of a form. An important parameter is the random upper **limit** for the integers to be summed. The task for the **second** template, *response.cfm*, is to calculate the correct answer

to the problem, decide whether the guess submitted is correct or not, and present the conclusion for the user.

Template *index.cfm* looks like this (the numbers to the left are not part of the script, but placed there to make it easier to refer to the different parts of the template):

1. `<!-- index.cfm -->`
2. `<CFSET temp=Randomize(second(Now()))>`
3. `<CFSET session.target=#RandRange(50,100)#>`
4. `<h2>Guess!</h2>`
5. `<form action="response.cfm">`
6. `<cfoutput>`
7. `<p>My name is <input type="text" name="name"></p>`
8. `<p>I guess the sum of all integers from 1 to #session.target# is <input type="text" name="guess"></p>`
9. `</cfoutput>`
10. `<p><input type="submit" value="Submit"></p>`
11. `</form>`

[Figure 1](#) shows the invitation to submit a guess for evaluation.

Guess!

My name is

I guess the sum of all integers from 1 to 52 is

Figure 1: My guess

Lines 2-3 illustrate the **CFSET** tag. The first command defines a variable called **temp** and give it a value computed by the **CF** function *Randomize(second(Now()))*. The **purpose** of this is to get a **random seed** for the next line. This is obtained by using the second at the moment of execution as an argument. If you look carefully at the expression, you will see that there are in fact **3 nested functions**. The innermost *Now()* gets the **time** (year, data, hour, minute, second) from the internal

clock of the computer, the intermediate *Second(Now())* extract the **seconds** from the time object, while the outer function uses the seconds to generate a **random seed**.

Line 3 defines a variable *session.target* and assigns to the variable a **random integer value** in the range from 50-100. This is done by the **CF** function *RandRange(50,100)*. Note that the expression is **enclosed** by # before and after the function. This simply means that the function value is assigned to the variable *session.target*. By qualifying a variable by *session*, the variable is made **persistent**, i.e. the variable *session.target* retains its value for use in other templates called by the same user within the same session. The detailed explanation is postponed to the next session.

Following a usual heading in Line 4, a **FORM** tag block is the **remaining** of the template. It is an ordinary **FORM** tag as described in the **HTML** texts with one **exception**: The text and input tags in Lines 7-8 are enclosed in **CFOUTPUT** tags. This is required to get the correct interpretation of the **CF** variable value *#session.target#*.

When the form generated by *index.cfm* is submitted by the user, the 2 variables, **name** and **guess**, followed with assigned values are sent with the request to the server. The **second** template *response.cfm*, controls the processing of the transferred variables and the returned response to the client.

1. <!-- response.cfm -->
2. <CFSET sum="0">
3. <CFLOOP INDEX="count" FROM="1" TO="#session.target#" >
4. <CFSET sum=#sum#+#count#>
5. </cfloop>
6. <CFIF #sum# EQ #guess#>
7. <cfoutput>
8. <h3>#name#,your guess was correct!</h3>
9. </cfoutput>
10. <CFELSE>
11. <cfoutput>
12. <h3>Sorry,#name#, the sum is #sum#.</h3>
13. </cfoutput>
14. </cfif>

Line 2 assigns value "0" to the variable **sum**. By means of **CFLOOP**, Lines 3-5 add the integers from 1 to **#session.target#** and save the results in the variable **sum**. In Line 6, a **CFIF** tag instructs the server to **test** if the guess is **correct**, and Lines 7 to 9 inform the user about a **correct** guess.

If the **sum** is **not equal** to the guess, the **CFELSE** is selected and the page sent to the user informing that the guess was not correct as well giving the correct answer.

[Figure 2](#) demonstrates the answer to an incorrect guess.

Sorry, Svein, the sum is 1378.

Figure 2: Evaluation of my guess

Exercises

- a. Read **Chapter 2** in **RBB** carefully, you will get repaid for the knowledge very soon.
- b. The guessing example consists of 3 templates, *Application.cfm*, *index.cfm* and *response.cfm*. **Establish** a **CFMX** script folder on your computer with sub folders for each example. The folder must be within the document root defined for your web server. **Copy** the templates from the browser display to your script folder. **Delete** the line numbers, save the files with extension *.cfm* and run the example.
- c. Consider how to **modify** your template *index.cfm* to generate also a random upper limit between 100 and 200 for the range. **Give** the new template the name *game_modified.cfm*. Try it out.

Session 3: Web market research

Introduction

Our next dynamic web application is design and implementation of a **hypothetical market research** on the web. You will find the visual web pages as illustrations. At the end of the session you will also find a link to the implemented session application example.

Market research

Does the **order of links** on a page have any influence on the selection by the user? This question has appeared in several connections. One way to approach an answer to this question is to design a web **experiment**.

Assume that a market research company is trying to **measure** the public's preference of 2 competing products, **A** and **B**. The company designs a **web form** that has two radio buttons, the upper for product **A** and the lower for product **B**, by which the visitor can express his/her preference for one of the products.

From experience, the market researcher knows that the ordering of A and B on the form may have an effect and that the preference may **not** be persistent. Two identical forms are therefore designed. One is called **Form 1** and has product **A** listed first. The other is called **Form 2** and has product **B** listed first. A script is designed for **random selection** of the form to be displayed for each visitor. The visitor is asked to request a second form in a week. The second form is another random selection of one of the forms. To attract customers to the web site, a **lottery** is set up for persons completing the 2 forms.

No names or addresses are required for completing the forms. To participate in the lottery drawing, the participants must, however, send in a separate form, called **Form 3**, with their name and address. A condition for participation in the lottery is that the two forms with preferences were returned.

Applications outline

[Figure 1](#) is an **outline** of the system we want to implement. The numbers indicate the **flow** in the system for each form requested, completed, and returned. The **challenge** will be to find a way to connect the submitted preference forms to the anonymous participants. Further, if a participant is willing to unveil his/her name to participate in the lottery; another solution must be developed to check that the person has answered the two required forms.

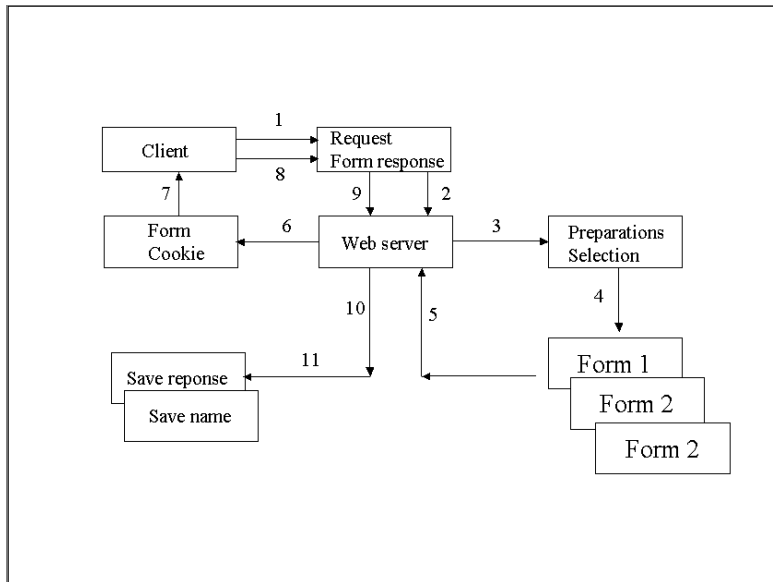


Figure 1: Outline of the market research system

Implementation

The first step is to establish a subfolder within which we will keep the application. The *Application.cfm* (the capital **A** is important only if you are working on a web server installed on platforms using Linux or UNIX) template must be saved in this folder. Recall that the content of this template is valid for the whole application, i.e. for all templates in the same sub folder. The application template for the market research application looks like this:

1. `<!-- Application.cfm -->`
2. `<CFAPPLICATION NAME="market_research"`
3. `SESSIONMANAGEMENT="yes"`
4. `SESSIONTIMEOUT=#CreateTimeSpan(0,0,30,0)#`
5. `SETCLIENTCOOKIES="yes">`
6. `<CFSET session.path="c:\myapplications\market_research">`

The application.cfm has 2 tags in addition to the name comment tag. The **CFAPPLICATION** tag specifies 3 attributes. **SESSIONMANAGEMENT** permits the use of session variables that have a scope comprising all templates in the applications, but are limited to a session time span. By means of the function **CreateTimeSpan(days, hours, minutes, seconds)**, the time span for each market research session variables is limited to 30 inactive minutes, i.e. if you leave your computer for 29 minutes it will still remember your session variables.

However, because we ask the visitors to come back in a week, we need a technique for **recognizing** a client when he submits the second and third form. We have several options. In this application, we use **cookies**. A cookie is identification **invisible** for the user, which the server sends a client when responding to a request, and which the client, if willing, saves in a special file. Next time the client makes a request to the server that issued the cookie, the cookie identifier will be attached to the request, and the server will know from which client the request is sent.

Use of cookies requires that the client has a browser able to receive cookies, and that his browser is set to receive cookies. This technique is useful if the client must be identified over a time period **longer** than a session. Setting cookies requires the third attribute, **SETCLIENTCOOKIES**, in the **CFAPPLICATION** tag. Note that **SETCLIENTCOOKIES** also influence the persistency of **session variables**! Read carefully what **RBB** writes about Session Variables in Chapter 7.

Finally, Line 6 specifies in a **CFSET** tag a session variable containing the path to a folder in which we want to store data. If for example you want to save your data in a subfolder **market_research** within the folder **c:\myapplications**, the value of **session.path** in the **Application.cfm** should be **"c:\myapplications\market_research"**. In all your templates, you can then use **#session.path#** instead of the longer **c:\myapplications\market_research**. This is also a very effective technique if you develop your applications on one computer with a directory structure that differs from that of the web server on which the application finally will be published.

We recall the rule that the entrance template to applications should be named **index.cfm** to prevent unwanted browsing of the folder if the browsing option has not been turned off. In the current applications, **index.cfm** is an information and menu template ([Figure 2](#)):

1. <!-- index.cfm -->
2. <h2><fontcolor="Red">Market research</h2>
3. <p>This is a market research to investigate the public's preferences for Product A and Product B. If you respond and complete the requirements stated below, you will be eligible to participate in a lottery.</p>
4. <p>The requirements are:</p>
5.
6. Request, complete and submit questionnaire 1 today
7. Request, complete and submit questionnaire 2 in a week
8. Request, complete and submit questionnaire 3 after you have submitted questionnaire 2
9.

10. <p>The 2 first questionnaires require only a single selection and click before you submit the response. The third questionnaire asks you for an e-mail address for notification in case you become a lucky winner in the lottery.</p>

11. <p>The market research sets a cookie in your browser. It is time-limited and will be automatically deleted after 15 days.</p>

Market research

This is a market research to investigate the public's preferences for Product A and Product B. If you respond and complete the requirements stated below, you will be eligible to participate in a lottery.

The requirements are:

1. Request, complete and submit [questionnaire 1](#) today
2. Request, complete and submit [questionnaire 2](#) in a week
3. Request, complete and submit [questionnaire 3](#) after you have submitted questionnaire 2

The 2 first questionnaire requires you make a single click only before you submit your response. The third questionnaire asks for you e-mail address for notification in case you become a winner in the lottery.

The market research sets a time-limited cookie in your browser.

Figure 2: Introduction to the market research

As the listing shows, this template could have passed as a **HTML** file since it contains no CFMX tags. To be consequent, we have given it the extension *.cfm*. [Figure 2](#) shows the menu.

A **random** selection between Form 1 and Form 2 is necessary. The template *prepare.cfm* contains the necessary script:

1. <!-- prepare.cfm --->
2. <CFSET temp=Randomize(Second(now())) >
3. <CFSET selected_number=RandRange(1,2)>
4. <CFIF #selected_number# EQ 1>
5. <CFLOCATION url="form1.cfm">
6. <CFELSE>
7. <CFLOCATION url="form2.cfm">
8. </CFIF>

Each time a pseudo random algorithm is started, it needs a **seed**. If the seed is the same, the sequence of random numbers will also be the same for all applications. **Randomize()** in Line 2 is a mathematical CFMX function that instructs the server to plant an initial seed based on the internal server **clock**. The next function, **RandRange(lowest, highest)** in Line 3, generates a

pseudo-random integer in the range between the lowest and the highest parameter values, in our case either 1 or 2.

Lines 4 to 8 represent an **if-else** block for selecting the form to present for the visitor. If the logical expression **#selected_number# EQ 1** is true, **Form 1** is selected by the tag **<CFLOCATION url="form1.cfm">**. The **CFLOCATION** tag is very useful because it **redirects** the control to another template. If the condition in Line 4 is not true, i.e. **#selected_number# EQ 2**, **Form 2** is selected for sending to the client.

The *form1.cfm* and *form2.cfm* generates the page shown in [Figure 3](#). The 2 forms are identical

Preference for products

Thank you for visiting this page and expressing your opinion. Complete and submit this form. If this is the first form you submit, please request another form: http://nordbotten.com/courses/cf/applications/market_research/, and submit it in a week. If you have submitted 2 forms and wish to participate in the lottery, request and submit the form: http://nordbotten.com/courses/cf/applications/market_research/form3.cfm.

Please mark your preference by clicking a button. Comparing the 2 products A and B, I prefer:

☐ Product A

☐ Product B

Figure 3: Form 1

except for the ordering of the 2 products and the value of the attribute *form_type* which is "1" in *form1.cfm* and "2" in *form2.cfm*, and we need consider only *form1.cfm*.

1. `<!-- Form 1 -->`
2. `<CFIF IsDefined("cookie.user_id") EQ 0>`
3. `<CFCOOKIE NAME="User_Id" VALUE="#Now()#" EXPIRES="15">`
4. `</CFIF>`
5. `<h2>Preference for products</h2>`
6. `<p>Thank you for visiting this page and expressing your opinion. Complete and submit this form. If this is the first form you submit, please request another form: http://nordbotten.com/courses/cf/information/files/market_research/, and submit it in a week. If you have submitted 2 forms and wish to participate in the lottery, request and submit the form: http://nordbotten.com/courses/cf/information/files/market_research/form3.cfm. </p>`
7. `<p>Please mark your preference by clicking a button. Comparing the 2 products A and B, I prefer:</p>`
8. `<CFFORM ACTION="save.cfm">`
9. `<INPUT TYPE="hidden" NAME="form_type" VALUE="1">`

10. <p><INPUT TYPE="Radio" NAME="Preference" VALUE="A"> Product A</p>

11.<p> <INPUT TYPE="Radio" NAME="Preference" VALUE="B"> Product B</p>

12.<p> <INPUT TYPE="submit" name="SUBMIT" VALUE="Submit"></p>

13. </CFFORM>

The form templates start with a **CFIF** tag in Line 2 that tests if a variable called **cookie.user_id** has been defined. The function *IsDefined("cookie.user_id")* returns value 1 if true and 0 if false. The **first** time a client requests one of the 2 form templates, we know the function must return a 0, while in **later** requests from the same client, the function will return 1 because the request now also includes the hidden cookie. If the function returns value 0, the variable **User_Id** is defined in Line 3 by the **CFCOOKIE** tag. The variable is assigned a unique value obtained by using the clock function **Now()**. We also require that the cookie will **expire** in 15 days. Using cookies, we are able to connect 2 or more responses from a client without requesting any further identification. **CFORM** has some additional features compared with the **HTML FORM**.

After some informative text in Lines 5, 6 and 7, the **CFFORM** tag follows, specifying the subsequent action by template *save.cfm*. Two radio buttons are included in the form for the visitor to flag his/her product preference. The form ends with a submit button. One of the 2 radio buttons must be pressed. Note that we do not ask the visitor for his/her name for anonymity reasons.

When the completed form is submitted, template *save.cfm* is executed. Line 2 tests if the file to which the data should be written has been established with a heading, and if not, it is established by a **CFFILE** tag with attribute **ACTION="write"** in Line 3.

It saves the returned response identified with the cookie **user_id**, the **form_type** and the preference in a text file named *response.txt*. In Line 5, the template makes use of a **CFFILE** tag that can have several actions and attributes. In this application we use the action "**append**", which require 2 attributes, the **FILE** in which the data from the form should be saved, and the **OUTPUT**, the variable values to be saved. Note that the complete path for the file is required, and we make use of the value of the session variable set in *Application.cfm*.

1. <!-- save.cfm --->

2. <cfif IsDefined("reponse.txt")EQ 0>

3. <cffile action="WRITE" file="response.txt" output="Response text file">

4. </cfif>

5. <CFFILE ACTION="append" FILE="#session.path#response.txt" OUTPUT="User id: #cookie.user_id#, Form_type: #form.form_type#, Preference:#form.preference#">

6. </CFFILE>

7. <CFLOCATION URL="index.cfm">

The last 2 building bricks of the market research application are a form requesting the email address if participation in the lottery is desired ([Figure 4](#)), and a template for saving the address.

Your e-mail address

If you are eligible for participating in the lottery, i.e. you have requested, completed and submitted the two questionnaires, we need your e-mail address to notify you in case you become a winner in the lottery.

Your name:

e-mail address:

Figure 4: Form 3

Since we already know the user identification of the participant, the form can be quite simple:

1. <!-- Form 3 -->
2. <p>Thank you for visiting this page and expressing your opinion. If you have complete and submitted 2 forms with your preferences for Product A and Product B, you are eligible to participate in the lottery. </p>
3. <CFFORM ACTION="save2.cfm">
4. <p>Your name:<cfinput type="Text" name="name" required="yes"></p>
5. <p>e-mail address:<cfinput type="Text" name="email" required="yes"></p>
6. <p><INPUT TYPE="submit" name="SUBMIT" VALUE="Submit"></p>
7. </CFFORM>

The cookie identification will also be attached to this form when returned, and we can check that the visitor is eligible as a participant in the lottery.

The *save2.cfm* template takes care of saving the cookie user identification of the visitor, his/her name, and address in *address.txt*. As for the first text file, the template tests for the existence of the file, and establishes the file if necessary:

1. <!-- save2.cfm -->
2. <cfif IsDefined("address.txt")EQ 0>
3. <cffile action="WRITE" file="address.txt" output="Address text file">

4. </cfif>

5. <CFFILE ACTION="append" FILE="#session.path#address.txt" OUTPUT="User id: #cookie.user_id#, Name: #form.name#, Email address:#form.email#">

6. </CFFILE>

7. <CFLOCATION URL="index.cfm">

By **sorting and merging** saved responses in response.txt and address.txt, we can establish a list of names and addresses for all visitors that have returned 2 forms with preferences, and the form with their e-mail address. The lottery can easily be carried out.

For market research analysis, the file *response.txt* will give **2 main classes** of data: First, all responses can be used to analyze the overall preferences for Products A and B. Second, all responses sorted by Form 1 and 2 can be used to analyze the effect of the ordering, and third, the responses ordered by user identification and time received can be used for investigating the preference persistence over time.

The template *report.cfm* gives an unedited display of the response.txt file. It can easily be copied and processed by EXCEL.

1. <!-- report.cfm -->

2. <cffile action="READ" variable="report" file="c:#session.path#response.txt"></cffile>

3.<cfoutput>#report#</cfoutput>

Inspect the values of the cookies, which indicate that some visitors have been visiting several times. You can also see the content of *address.txt* from the menu page by means of a similar template, *report2.cfm*.

Exercises

a. Read **Chapter 3** in **RBB** about ways to pass data between templates. In the first part of Chapter 7, you can read about application templates and cookies, and jumping to **Chapter 12**, you will be able to read more in detail about the **CFFILE** tag.

b. Study the application **Market research** carefully. In the first form, it is possible to request **questionnaire 1**, **questionnaire 2**, or **questionnaire 3**. It would be more professional if only **questionnaire 1** could be called if this was the first visit, only **questionnaire 2** if the first had already been submitted and only **questionnaire 3** if both the previous questionnaires had been completed. Try if you can see a way do this improvement.

c. Copy and install the templates on your own computer and try to run the application. You may meet a few problems, but don't give up.

d. Extend the assortment to 3 competing products. How would you re-arrange the experiment to obtain an unbiased set of preference responses?

Session 4: Web database for opinion polls

Files and databases

In Session 3 we used the tag **CFFILE**, which permits storing and retrieval of files on disk when needed. However, the files created by **CFFILE** can only be **written**, **appended**, **read** or **deleted**. If any operations on individual records, such as conditional **modifications** or **retrieval** of records, are needed, the file has to be read into the memory before it can be processed and then re-written. Modern **databases** can store complex collections of data, and flexible methods for inserting and retrieving data exist.

Databases

This is **not** a course in databases, but databases are frequently **needed** in connection with web applications. We shall limit our discussion in this session to how an established database can be used to serve the needs of web applications by means of the **SQL language** included in **CFML**. An established database includes one or more named **tables**. Each table has one or more named **columns**, and zero or more identifiable **rows** with column data.

The connection between a web applications and the database we shall use, is through **Java Database Connectivity**, **JDBC**, drivers which permit interrogations and updating of a database by means of **SQL** statements. In some cases, the **JDBC** driver will also have to cooperate behind the scene with **ODBC** drivers.

Databases, which can be used as a **back-end** to ColdFusion applications, are limited to those for which adequate drivers **exist** for the operation system/hardware platform used. For **Windows XP/2000**, on which this course is based, drivers for **MS Access**, **Excel**, **MySQL**, **SQL Server**, and a few other are included. Drivers developed by others exist, however, for a number of other databases/platforms. More efficient native drivers for **DB2**, **Informix**, **Oracle** and **Sybase** are available from **Macromedia** to be used in connection with the **CFMX Enterprise** version.

In this course, we use a **MS Access database**. The database used has no significance for the functionality discussed. The **MySQL** database can be downloaded free and is an excellent alternative.

An installed and specified database must be named as a **datasource**, and registered with the **CFMX Administrator** as an accessible database. The name is usually some shortcut, and in this course, the datasource name used is **db**. If the database corresponding to a datasource name exists, the database may be extended with the tables needed. Registering the datasource with the **CFMX Administrator** means that it has to be listed as a datasource known to the **CFMX** system. The registering of a datasource should be done with your **CFMX Administrator**. If you installed **CFMX** with the **standalone** web server, the URL of the Administrator is **http://localhost:8500/CFIDE/ADMINISTRATOR**. If you use another web server, it is the same without the port address "**:8500**".

CFMX SQL

ColdFusion opens for interaction with a data base by means of a special pair of tags, **CFQUERY** and **/CFQUERY**, between which the **SQL** statements are inserted. The tag syntax is:

```
<CFQUERY NAME="myquery" DATASOURCE="#session.datasource#" >
```

SQL statements

```
</CFQUERY>
```

The **NAME** is a useful **reference** to the output from a database transaction if needed in other places of the current or other templates. We shall see examples later on. The only requirement is that a unique name is chosen. The **DATASOURCE** parameter specifies which database is relevant, and in our course the name is **db**, which implies that a **<CFSET session.datasource="db">** must be set, usually as part of the *Application.cfm* template.

Basic SQL elements

We consider 4 **SQL** statements:

- SELECT
- INSERT
- UPDATE
- DELETE

We shall discuss applications of each type in detail below.

Within each statement, a number of **clauses** can be specified of which some are restricted to use with only one or two of the **SQL** commands. The list of the most commonly **SQL** clauses in alphabetic order, is:

- FROM
- GROUPED BY
- HAVING
- INTO
- JOIN
- ORDERED BY
- SET
- VALUES
- WHERE
- UNION

For construction of **compound statements**, **SQL** provides also a large set of operators:

- = : equal to
- <> : not equal to

- < : less than
- > : greater than
- <= : less than or equal to
- >= : greater than or equal to
- + : plus
- - : minus
- / : divided by
- * : multiplied by
- AND : both conditions must be true
- OR : one or both conditions must be true
- NOT : ignores a condition
- IS [NOT] NULL : value is [not] null
- IN : value is within a list
- BETWEEN : value is in the range between two values
- LIKE : value is like a % or _ value
- EXISTS : tests for a non-empty set

In addition to the datasource name, you must also know the **name** of the **table**, the **columns** and **rows** from which you want to retrieve data or add data. Table, column and row specifications will be discussed in the next section.

Opinion polling

A **polling institute** is the scenario for the service we shall use to explain the SQL statements.

The institute investigates **changing opinions** among 5 competing brands of a certain commodity. The organization uses a **rotating** panel of voter who are interviewed about their preferences each week. The panel must include a certain number of voters to provide **significant** results. For each voter, name, telephone, age and home area is recorded for the establishment of the panel. Each **Monday**, the panel voters are interviewed about their opinions about the preferred alternative among the 5 competing products. For each interviewed panel voter, name, telephone number, and age may also change and are checked. The updating takes place on **Wednesday**. Each **Thursday**, statistics are generated from the records and made available, and each **Friday**, one fraction of the panel is substituted with new voters. [Figure 1](#) shows the processes in the system.

In this session, we assume that the **db** datasource has a specified table named **Voters**. The box at the left hand in [Figure 1](#) shows the columns of the table.

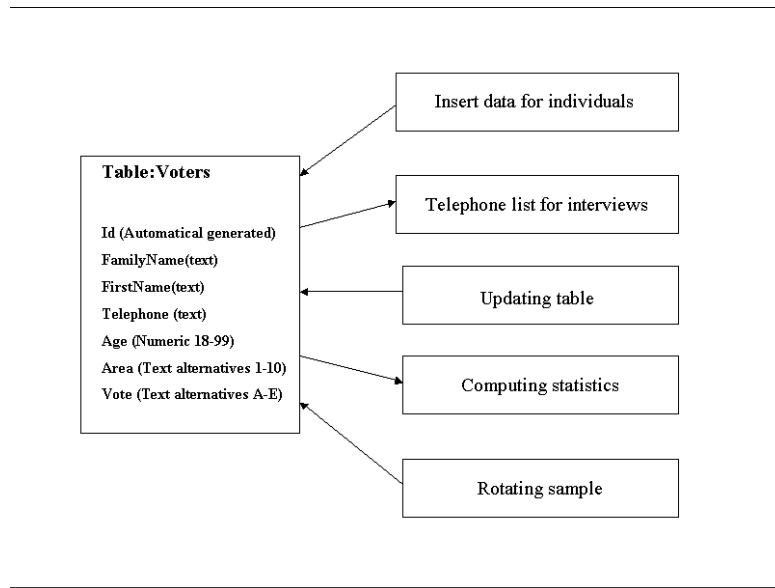


Figure 1: Opinion poll application

A link to a complete demonstration of the application is available at the end of the session. As usual, the application starts with an *Application.cfm*.

1. <!-- Application.cfm -->
2. <CFAPPLICATION NAME="database"
3. SESSIONMANAGEMENT="yes"
4. SESSIONTIMEOUT=#CreateTimeSpan(0,0,30,0)#>
5. <CFSET session.datasource="db">

In this *Application.cfm* the cookies used in the example of the last session, are not needed, but a **CFSET** tag for setting the *session.datasource* variable has been added.

A menu, ([Figure 2](#)), implemented by the *index.cfm* template introduces the application. This is an ordinary HTML type of form template with simple hyperlinks for the different options. We could have named the file *index.htm* as well.

Opinion polls

The Opinion polls system is **initialized** with a sample of panel members. Each Monday a **List** of panel members to be interviewed is generated. After the interviews, a table with data for each of the panel members is **updated**. The table is the basis for computing **statistics** for the week. At the end of the week, the first panel member on the list is **deleted**, and a new member **added** at the end of the list.

1. **Initialize** table of panel members
2. **List** panel members for interviews
3. **Update** table of panel members after interviews
4. **Compute** statistics for the week
5. **Delete** first and add new panel member

Figure 2: Application menu

1. <!-- index.cfm -->

2. <h2>Opinion polls</h2> <p>The Opinion polls system is initialized with a sample of panel voters. Each Monday a List of panel voters to be interviewed is generated. After the interviews, a table with data for each of the panel voters is updated. The table is the basis for computing statistics for the week. At the end of the week, the first panel voter on the list is deleted, and a new voter added at the end of the list.</p>

3.

4. Initialize table of panel voters

5. List panel voters for interviews

6. Update table of panel voters after interviews

7. Compute statistics for the week

8. Delete first and add new panel voter

9.

and needs no further comments.

INSERT statements

It is not our concern to identify and recruit a representative panel of persons willing to serve on the panel. We simply assume that the required number of persons has been recruited and information has been collected about their names, telephone number, age and area in which they live. Votes will be collected later by interview.

Form to be used for adding new members to the interview panel

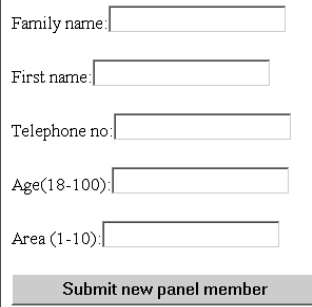


Figure 3: Form for establishing the list of panel members

To establish the list of panel voters, a form template will be needed to record and for transcribing collected information to a table in the database. The form can look like [Figure 3](#) and the template like the following:

1. `<!-- form.cfm --->`
2. `<h2>Form to be used for establishing the interview panel</h2>`
3. `<form action="add.cfm">`
4. `<p>Family name:<input type="text" name="FamilyName"></p>`
5. `<p>First name:<input type="text" name="FirstName"></p>`
6. `<p>Telephone no:<input type="text" name="Telephone"></p>`
7. `<p>Age(18-100):<input type="text" name="Age"></p>`
8. `<p>Area (1-10):<input type="text" name="Area"></p>`
9. `<p><input type="submit" value="Submit new panel voter"></p>`
10. `</form>`

As you see from Line 3 in the template, the form requests a template named *add.cfm* when it is submitted for further processing. The **form.cfm** template also uses ordinary **HTML** tags, and could as well have been named with a *.htm* extension. However, we use consequently the *.cfm* extension.

The purpose of the *add.cfm* template is to **append a new row** with values to a database table called **Voters**. The panel voters need to be assigned a unique *Id* number, and we start by introducing the mechanism needed for this in Lines 2-5. The core in this is the variable *application.id* which keeps track of the last identification number assigned. The qualifier

application makes the variable persistent within the application, in other words, the service will keep the number in mind from session to session.

The tag in Line 2 tests if the application has been active before and the **application.id** variable has been defined. If not, the variable is defined in the next line and assigned value "0". If the application has already been used the control is transferred directly from Line 2 to Line 5, and the value of the identification variable is increased by "1", i.e. the first time the identifier variable is assigned the value "1", next time the value "2", and so on.

```
1. <--- add.cfm --->
2. < CFIF NOT IsDefined("application.id")>
3. < CFSET application.id="0">
4. < /CFIF>
5. < CFSET application.id=#application.id#+1>
6. <CFQUERY NAME="add" DATASOURCE="#session.datasource#">
7. INSERT INTO Voters(Id,FamilyName, FirstName,Telephone,Age,Area,Vote)
VALUES('#application.id#','#FamilyName#','#FirstName#','#Telephone#','#Age#','#Area#','#-')
8. </CFQUERY>>
9. <CFLOCATION url="form.cfm">
```

This template uses in Line 6 the **QUERY** tag with an **INSERT INTO** statement which specifies the name of the table, **Voters**, and all predefined column names of the table following in a parenthesis. It is extremely important that the list contains column names correctly spelled. The next attribute is **VALUES** with a parenthesis containing all the values which should be saved in a new row of the table.

Note 2 important details. for each specified variable, a value represented by the variable name enclosed by # before and after the name, for example **#FamilyName#**, is needed. If the variable is a **text** type, this should be indicated by single quotes before and after the name and the #'s, for example **'#FamilyName#'**. As you can see, we specify all values as text, because all variables were specified as text in when we defined the table in the database. The last value is **'-'**. The surrounding single quotes tells us that this is another **text** variable and the value of the variable **Vote** is set preliminary to **'-'** because the person has **not** yet been interviewed.

After all panel members have been appropriately recorded by means of this form, the polling can start.

SELECT statements

Monday morning the telephone interviewers need a list for calling the panel voters. This list can be retrieved by calling the template "*list.cfm*" which illustrates the use of the statement **SELECT**. The CFQUERY tag includes a **NAME="list"** which will be used below. The **SQL** statement **SELECT** is followed by a list of those variable which are required. The attribute **FROM** specifies from which table, **Voters** in our application, the variables should be retrieved. As for all **SQL** statements, the syntax is very strict. Only variable names specified in the named table are accepted.

1. <--- list.cfm ---
2. <CFQUERY NAME="list" DATASOURCE="#session.datasource#">
3. SELECT Id,FamilyName,FirstName,Telephone,Age,Area FROM Voters
4. </CFQUERY>
5. <h2>List of panel voters</h2>
6. <TABLE >
7. <TR>
8. <TD>ID</TD>
9. <TD>Family Name</TD>
- 10.<TD>First Name</TD>
- 11.<TD>Telephone</TD>
12. <TD>Age</TD>
13. <TD>Area</TD>
14. </tr>
15. <CFOUTPUT QUERY="list">
16. <TR>
17. <TD>#id#</TD>
18. <TD>#FamilyName#</TD>
19. <TD>#FirstName#</TD>
20. <TD>#Telephone#</TD>

21. <TD>#Age#</TD>

22. <TD>#Area#</TD>

23. </TR>

24. </CFOUTPUT>

25. </TABLE>

The content of the column **Vote** is excluded from the listing to avoid that the interviewer reminds the panel voters about their answers last week.

The second part of the template concerns the **tabulation** of the list. The **TABLE** tags used are well known from **HTML** which permits more elaborate tables with background, borders, etc. In this example, we keep it as simple as possible. Note that the results from the **CFQUERY** can be referred to in the **CFOUTPUT** tag by its name **QUERY="list"**. All the retrieved rows are considered as a **Query Object** with the **NAME="list"**. By referring to the attribute **QUERY="list"** in the **CFOUTPUT** tag, each retrieved row from the table **Voters** will be listed according to the table specifications. No loops are needed.

[Figure 4](#) illustrates the output from this template.

List of panel members

ID	Family Name	First Name	Telephone	Age	Area
11	Peter	Smith	618 347 6578	23	1
12	Camilla	Reagan	481 333 6234	58	7
13	Neil	Butterfly	785 384 2365	33	2

Figure 4: List for the interviewers

UPDATE statements

The third step is a process which permits **updating** of the panel table. For this we use the **SQL UPDATE** statement. Updating requires that all fields of the update form are populated. Since the fields **FamilyName**, **FirstName**, **Telephone**, **Age** and **Area** are usually unchanged; a direct use of the insert form used to create the initial table would require a large amount of unnecessary typing of redundant information. The solution is first to **retrieve** a list of **Id's**, and **then** updated with the changes needed.

The first template *retrieve.cfm* retrieves the unique **ID** of the current panel voters and produces a list with **links** to each individual row of the table:

```
1. <!-- retrieve.cfm -->
2. <cfquery name="retrieve" datasource="#session.datasource#">
3. SELECT Id FROM Voters
4. </cfquery>
5. <p><h2><font color="Blue">List of ID for updating</font></h2></p>
6. <CFOUTPUT QUERY="retrieve">
7. <TABLE>
8. <tr>
9. <td><a href="retrieve2.cfm?Id=#Id#">#Id#</a></td>
10. </tr>
11. </table>
12. </cfoutput>
```

This template, called from the menu, is extremely simple. The **SQL** has a **SELECT** statement, which retrieves only the **ID** from each row, and tabulates a list of **ID**'s ([Figure 5](#)). The table has

List of ID for updating

[11](#)
[12](#)
[13](#)

Figure 5: List for updating member data and votes

one small **finesse**, the **ID**'s are linked in order to retrieve wanted rows of **Voters** by a click if wanted. The reference used is "**retrieve2.cfm?Id=#Id#**" which refer to a second retrieval template, *retrieve2.cfm*, and the attribute **Id=#Id#**. Remember that **#Id#** is the value of the **ID** variable for the particular row considered. By attaching the name-value after the template name

by question mark ?, the value **#Id#** is made available to template *retrieve2.cfm* when this template is processed. The output is illustrated in Figure 5.

We now want to design a template, which **returns** an individual form with the **old values** for the selected panel voter to the client for updating changes in data of the voter. The task is solved, see Lines 2-4, by a **CFQUERY** to the **#session.datasource#** combined with a **SELECT** of all variables values from **Voters** for the panel voter with **Id=#Id#**. A pre-filled updating form is created by means of the query object named *retrieve2* in the **CFQUERY** tag:

```
1. <!-- retrieve2 --->

2. <cfquery name="retrieve2" datasource="#session.datasource#">

3. SELECT Id,FamilyName,FirstName,Telephone,Age,Area, Vote FROM Voters WHERE Id='#Id#'

4. </cfquery>

5. <cfform action="update.cfm" >

6. <cfoutput query="retrieve2">

7. <h2><font color="Blue">Update form for Id:#Id#</font></h2>

8. <table>

9. <tr><td>Family name:</td><td><input type="text" name="FamilyName"
value="#retrieve2.FamilyName#"> </td> </tr>

10. <tr><td>First name:</td><td><input type="text" name="FirstName"
value="#retrieve2.FirstName#"></td></tr>

11. <tr><td>Telephone:</td><td><input type="text" name="Telephone"
value="#retrieve2.Telephone#"></td></tr>

12. <tr><td>Age:</td><td><input type="text" name="Age" value="#retrieve2.Age#"></td></tr>

13. <tr><td>Area:</td><td><input type="text" name="Area" value="#retrieve2.Area#"></td></tr>

14. <tr><td>Vote:</td><td><input type="text" name="Vote" value="#retrieve2.Vote#"></td></tr>

15. </table>

16. <input type="hidden" name="ID" value="#Id#">

17. </cfoutput>

18. <p><input type="submit" value="Update"></p>

19. </cfform>
```

Since the **Id** is permanently assigned to each panel voter, it cannot be updated and should not be included in the form, but it has to be attached and the returned values to get the correct line updated. This is solved by sending it to the client as a **hidden** variable as can be seen in Line 16. The form will be **displayed** with the old values on the client screen, and all values (except the Id value) can be **changed** according to the information received at the interview and with the vote added ([Figure 6](#)).

Update form for Id:11

Family name:	<input type="text" value="Peter"/>
First name:	<input type="text" value="Smith"/>
Telephone:	<input type="text" value="618 347 6578"/>
Age:	<input type="text" value="23"/>
Area:	<input type="text" value="1"/>
Vote:	<input type="text" value="A"/>

Figure 6: Form for updating individual data

When the form has been corrected and the update submitted, it calls a small template *update.cfm* template:

1. `<!-- update -->`
2. `<cfquery name="update" datasource="#session.datasource#" >`
3. `UPDATE voters SET`
4. `FamilyName='#FamilyName#',`
5. `FirstName='#FirstName#',`
6. `Telephone='#Telephone#',`
7. `Age='#Age#',`
8. `Area='#Area#',`
9. `Vote='#Vote#'`
10. `WHERE ID='#Id#'`

11. </cfquery>

12. <CFLOCATION url="retrieve.cfm">

This template demonstrates the **CFQUERY** tags with the statement **UPDATE**. The **UPDATE** statement specifies the table, **Voters**, which should be updated and use the attribute **SET** with a list of name-value pairs of the variables to be updated. The update refers to the row **WHERE ID=#ID#**. Also in the **UPDATE** statement, the different components must have exact names and acceptable values with no comma delimiter before the **WHERE**. Finally, by means of the **CFLOCATION**, the control is transferred to the *retrieve.cfm* template to produce the list and for selection of another voter for updating.

Statistical report

When **all** received updates have been performed, the database table is ready for providing data for statistics of the week. In this example, we aim at very **simple statistical report** about the number of panel voters having preference for each of the 5 alternatives. The computation can easily be **extended** to computations of the preference frequencies by age and by area. The presentation of statistics requires 4 steps to be performed:

1. Defining frequency variables
2. Retrieving data
3. Computing statistics
4. Presenting results

The template *compute.cfm* takes care of these tasks:

1. <!-- compute.cfm -->

2. <cfset A="0">

3. <cfset B="0">

4. <cfset C="0">

5. <cfset D="0">

6. <cfset E="0">

7. <cfquery name="statistics" datasource="#session.datasource#">

8. SELECT vote FROM voters

9. </cfquery>

10. <cfloop query="statistics">

11. <cfswitch expression="#vote#">

12. <cfcase value="a"><cfset A=#A#+1></cfcase>
13. <cfcase value="b"><cfset B=#B#+1></cfcase>
14. <cfcase value="c"><cfset C=#C#+1></cfcase>
15. <cfcase value="d"><cfset D=#D#+1></cfcase>
16. <cfcase value="e"><cfset E=#E#+1></cfcase>
17. </cfswitch>
18. </cfloop>
19. <cfoutput>
20. <p><h2>The votes this week:</h2></p>
21. <table>
22. <tr><td>
23. Alternative:</td>
24. <td>Frequency:</td>
25. </tr>
26. <tr>
27. <td>A</td><td>#A#</td>
28. </tr>
29. <tr>
30. <td>B</td><td>#B#</td>
31. </tr>
32. <tr>
33. <td>C</td><td>#C#</td>
34. </tr>
35. <tr>
36. <td>D</td><td>#D#</td>

37. </tr>

38. <tr>

39. <td>E</td><td>#E#</td>

40. </tr>

41. </table>

</cfoutput>

After using **CFSET** to define a set of 5 **aggregate** variable all with values "0", a **CFQUERY** with **SELECT** of all variables in all rows of Voters retrieves the data from the database. It is followed by a **CFLOOP** block with a **CFSWITCH** and connected **CFCASE** statements to loop through all rows of the retrieved data from Voters, and to count the number of cases with preference for A, B, C, D and E, respectively.

When the **looping** has finished, **tabulation** of the five frequency variables and their values is done by means of a simple set of **TABLE** tags. [Figure 7](#) shows the distribution of 3 panel voters we have used for this example.

CFMX has also included some powerful Graphing and Charting possibilities, which could be used for creating favorable alternatives to the statistical reporting used above.

The votes this week:

Alternative: Frequency:

A	1
B	1
C	1
D	0
E	0

Figure 7: Votes of current week

DELETE statements

The final part of the system is to make the system **rotating**, i.e. create a templates which **add a**

Figure 8: Form for rotating the members

new at the end of the list and **delete the first voter** of the list. The template called for this purpose from the menu is *form2.cfm*. It creates a form for filling in data for a new voter ([Figure 8](#)):

1. <!-- form2.cfm --->
2. <h2>Form to be used for deleting and adding voters to the panel</h2>
3. <h3>New voter data:</h3>
4. <form action="delete.cfm">
5. <p>Family name:<input type="text" name="FamilyName"></p>
6. <p>First name:<input type="text" name="FirstName"></p>
7. <p>Telephone no:<input type="text" name="Telephone"></p>
8. <p>Age(18-100):<input type="text" name="Age"></p>
9. <p>Area (1-10):<input type="text" name="Area"></p>
10. <p><input type="submit" name="NewMember" value="Submit new panel voter"></p>
11. </form>

When the form is completed and submitted, it calls upon template *delete.cfm*. It starts by retrieving all **ID** values from **Voters** by means of a simple **SELECT** statement. The **Id** of the first row can be found in different ways. We use a **CFLOOP** statement with the attribute setting **STARTROW="1"** and **ENDROW="12"** which only extracts the **first Id** which is defined as **voter_out=#ID#**. This variable is used in a **CFQUERY** with the SQL statement **DELETE** from **Voters** and for the row which has **Id=#voter_out#** which completes deleting the first voter of the panel.

1. <--- delete.cfm --->
2. <cfquery name="retrieve" datasource="#session.datasource#">
3. SELECT Id FROM Voters
4. </cfquery>
5. <cfloop query="retrieve" startrow="1" endrow="1" >
6. <cfset voter_out=#retrieve.id#>
7. </cfloop>

8. <cfquery name="delete" datasource="#session.datasource#">

9. DELETE FROM Voters WHERE id='#voter_out#'

10. </cfquery>

11. <CFSET application.id=#application.id#+1>

12. <cfquery name="add" datasource="#session.datasource#">

13. INSERT INTO Voters(Id,FamilyName, FirstName,Telephone,Area,Vote)

14. VALUES('#application.id#','#FamilyName#','#FirstName#','#Telephone#','#Age#','#Area#','-')

15. </cfquery>

16. <CFLOCATION url="form2.cfm">

The **second** part of this template adds the submitted data for the new panel voter at the end of the table by means of a **CFQUERY** and an **INSERT** statement. Note that the new value of **ID** is computed by the tag in Line 11.

In this section we have discussed some of the **simplest and most basic SQL** statements available in **CF**. In later sections we shall demonstrate some further features of the language available within **ColdFusion MX**.

Exercises

- a. In this session, we have interacted with a database. You can develop complex applications interrogating local as well as remote databases, which frequently are the main cores of information systems. Read the **Chapters 4 and 5** of **RBB** carefully and compare the example of this session with the text. See if you can point at possible improvements.
- b. Try to copy this session's templates, implement them on your own server and run them.
- c. In Session 3 you learned about the **CFFILE** tag. It is possible to by-pass the use of a database in the application discussed in this session using the **CFFILE** tag instead. Consider how you could re-write the templates in the opinion polls application using the **CFFILE** tag, and consider if it would be efficient if you had a panel of 10 000 voters.
- d. Look through the **Chapters 1-10** and see if they can provide you with new ideas for preparing the project proposal you must submit before you can advance to the next session.
- e. Read Chapter 17 in **RBB**, and try to design a weekly graphical report on the votes.

Session 5: Web perception application

Introduction

How to **present a visual message** to the public in such a way that it will be correctly perceived and remembered? This is a task of great **importance** for politicians, information agencies, product promoters, news media, educators and many more. It is an essential task for an information agency with the responsibility to disseminate facts to the public about local and global events as a **background** for democratic decisions expressed by an election or a referendum.

A news promoter may have **several options** for a fact presentation, for example, it may be:

- discussed in a text form
- displayed in a table
- expressed in a graphical chart
- read from an audio recording
- played as a video

The problem is that we have **little knowledge** about the relative effects on the public of the different multimedia options.

One approach to learn more about the impacts of the different alternative fact representations on the public opinion can be to design and carry out an **experiment** using the web. A web experiment must satisfy the general **principles for experimental** designs and will be a highly dynamic web application. In this session, we shall study the design and implementation of such an experiment limited to the 3 first presentation forms listed above.

The first step is to create a system which **simulates** the alternative presentations to the public. As already pointed out, such a system can be designed and implemented as a dynamic web application. The next step is to **recruited** participants to participate in the experiment. Ideally, the participants should be a representative sample of the population, which we want to inform. This can, however, be both expensive and difficult, and frequently we recruit volunteers, a strategy, which may decrease the representativity of the sample. The perceptions of each participant from the exposed presentations are measured and recorded, and will finally be **tested** for significant differences.

As an example in this session, we illustrate the experimentation on the net by a set of 25 **socioeconomic facts** which we present in alternative ways and observe how the participants perceive the different presentations. The recruited participants are recorded by their individual attributes, which permit to classify their expressed perceptions according to participants' categories.

A **keyword** distributed to each recruited participant gives access to the experiment and serves to keep input from participants separated. Each participant in the experiment will be presented with a **random sequence** of the facts. The **representation** of each fact is also randomly determined. The

presentation is displayed for each participant for a **period** of random length. Following each presentation, the participant will receive a form for reporting his **perception** of selected characteristics connected to the fact. The purpose of the randomization of presentations and time is to avoid any systematic influence on the results.

The experiment has been carried out with several groups. A published paper in *.pdf* format with some results is available in [Nordbotten & Nordbotten](#).

Outline of the design

[Figure 1](#) gives an outline of the experiment. We implement a CFMX application system with a **database** as a back-end component for keeping references to a **gallery** of presentations and for storing the **feedback** from participants.

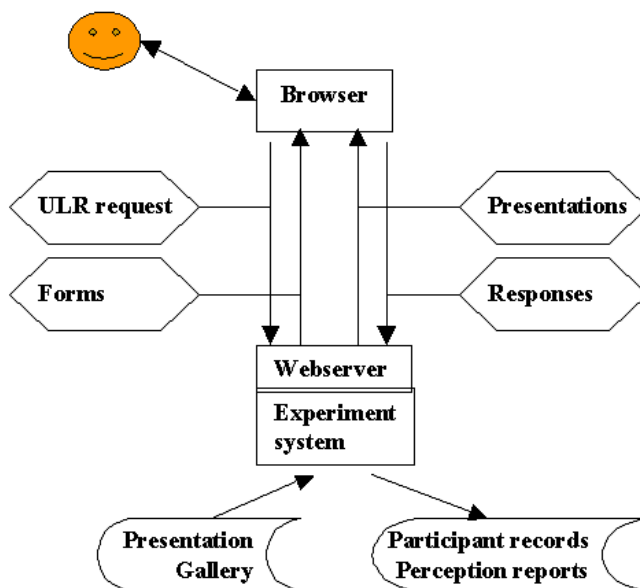


Figure 1: Overview of experiment structure

We focus our discussion to the **web application** dimension of the experiments and assume that the **participants** are recruited by other professionals. The participants will work with their browsers on client computers connected to our server by the WWW. Between each client and the server, there are 4 streams of messages, two in each direction. From the client, **requests** for participation and **responses** to forms will be moving to the server. From the server, **presentations** and related **forms** for responses will be sent to the clients.

The experiment is based on a set of 25 international facts from the socio-demographic domain. Each fact is represented in 3 alternative presentations, **textual**, **tabular** and **graphical**, in the gallery of images. Other representations, such as **voice** and **animation**, can be added.

Each participant who completes the experiment will have seen all 25 facts each presented in **one and only one** of the 3 alternative presentations. Following each presentation, the participant is asked to complete a standard form about perceived characteristics of the fact presentation.

With a satisfactory number of participants, the recorded data will permit **analysis** of perception characteristics of the 3 **representational forms** subject to background factors expressed by individual demographic characteristics, fact and representation types, position in sequence and display time.

Design of the database

The starting point for the design of the experiment is a **database**. Several database systems can be used including Microsoft Access and MySQL. The requirement to the database is that there exists a JDBC driver for the server platform used. In this session, **MS Access** is used as a simple solution.

The database is a collection of **tables** with **columns** corresponding to variables. Each table row is a recording of an event, and each column contains the event values for the variable represented in the particular column and for the particular row.

Four tables are used for the experiment:

1. calls(participant,fact)
2. charts(chart,fact,type,url)
3. form1(participant,day,hour_minute,host,gender,age,subject,resident, keyword)
4. form2(participant,chart,wait,fact,type,unit,topic,measure,tid,category,startform2,endform2)

The names within the **parentheses** are the column variables of the respective tables. **Table 1, 3** and **4** have all the variable **participant** which means they can be combined to give a wide empirical description of the participant and his/her demographic attributes as well as his/her interaction with the experimental system.

Table 1 is used to keep track of the facts presented for each participant to **avoid** that he/she is exposed to the same fact more than once. Each participant is identified by an internal identifier, participant, which is kept during the session in order to make the presentations and responses connectable during the analysis.

The **second table** is a listing of all fact presentations in the gallery. Since there are 25 facts, all represented by 3 different presentations, the table has 75 rows, each with a distinct chart number, reference to the fact represented, the type of presentation, and a pointer to the chart images in the gallery. This table is the basis for **random drawing** representations and a link to the gallery with the image files to be presented. Note that the gallery is not part of the database, but a collection of image files in a separate directory.

Form 1 is used for recording when a participant joined the experiment (date, hour and minute) as well as his/her demographic data values for the attributes gender, age group, professional background (in work/studies), country of residence and keyword.

The **last** table is the main table in which the feedback from the participants for each presentation is saved. The table variables **reflect** the participant, the chart reported on, the time it was displayed for the participant, the fact presented in the chart, the chart topic perceived by the participant, perceived measure used in the chart, if the presentation refereed to a single/multiple time(s), category, start times and end times are variables used to measure the times needed for completing the forms.

A table containing the 'correct' values for each fact also exists for use during the subsequent analysis of the experiment. Since we are not discussing the analysis in this course, this table is not included in the listing above.

Design of the dynamic web process

The design of the web process to be implemented has 6 main templates of which some have attached templates for background processing:

1. Opening page explaining the application and a login form
2. Demographic form
3. First chart display
4. Perception report form
5. Next chart display
6. Thanks for participating

In addition to these templates, an *Application.cfm* template setting the session management must be created for specification of persistent session variables. The variable session.datasource is also set in this template.

Opening page

The first template, *index.cfm*, **opens** the experiment for the participant. It explains the application and asks the participant for his/her **keyword**. As indicated above, the keyword is not a password, but simply a means for recording to which recruitment **group** the participant belongs.

The opening template contains a small form which is used to record the participants **key** (in the application example, you can use the key="CF"). When the form is submitted, the control is transferred to template *form1.cfm* (Line 3).

1. <--- index.cfm --->

2. <p>(There is a relative long introduction of the application which has little relevance for the implementation. You can read the full text in the application example.)</P>

3. <FORM ACTION="form1.cfm" METHOD="post">

4. <p>If you wish to participate, please type your keyword: <INPUT TYPE="TEXT" NAME="KEYWORD" VALUE=" " "></p>

5. <p>and click the button,</p>

6. <p> <INPUT TYPE="SUBMIT" NAME="response" VALUE="Participate"></P>

<p>else go to your preferred URL.</p>

After reading about the experiment as indicated in Line 2, the visitor is offered the option to **cancel** his/her participation as indicated in [Figure 2](#).

The applications is carried out according to the following rules:

1. When you decide to participate, click the Participate button at the bottom of this page. A *Demographic form* will be displayed, and you will be asked to give certain demographic data about yourself. These data will be used to classify your contributions in socio-demographic categories.
2. After submitting the completed form, a presentation of a fact chosen at random will be generated and displayed. The length of time the presentation is exposed, varies randomly within a fixed time interval as part of the applicationsal design, and sometimes the presentation is not displayed at all.
3. When the presentation has disappeared, a second form, the *Report form*, will follow with questions about the presentation you just watch. The form has been designed to be technically easy to complete, and the time for responding to this form is not time limited. There are probably questions which you have to leave Unanswered because of the way the fact was presented. To learn from your contribution how facts should not be presented, is just the aim of this applications. *Do not be tempted to use your Back button to return to the presentation. The Back button will destroy your report and your evaluations will be lost.* At the end of each report form, you will be invited to continue.
4. Steps 2. and 3. above will be repeated, but in such a way that you never will be presented with the same fact more than once.

The applications will go on as long as there are facts left and you agree to participate .

If you wish to participate, please type your keyword: and click the button

or else go to your preferred URL.

Figure 2: Introductory page

Demographic form

Before starting to record the demographic data, it is important to give the participant an **identification** permitting to recognize him/her each time he/she returns a feedback. One option is use of **cookies** as we did in the Market research application. Since some people do **not permit** their browsers to accept cookies, we use another technique in this experiment.

As soon as a visitor has decided to become a participant, control goes to template *form1.cfm*. According to Line 2, the participant is assigned an **internal identifier**, **#session.participant#**, computed from the **time** the form was processed by the server. A simpler time identifier can be used, but in this session we compute the identifier from scratch to demonstrate the principle in detail. There is a microscopic risk that 2 or more participants can be assigned identifiers within the same second. A few other variables are also set to last through the session in Line 3-6, including the day, time, **IP** number of the client, and the keyword which all is required saved in the database.

Then follows the demographic form in Line 11-64 displayed for collecting the background data about the participant. Even though advantage is taken of the **HTML** form possibilities including menus to check the relevant option, the template is long, but does not introduce any new **CFMX** features.

1. <--- form1.cfm --->

2. <CFSET session.participant=Second(#Now())+60*Minute(#Now())+60*60*Hour(#Now())+60*60*24*Day(#Now())+60*60*24*30*Month(#Now())+60*60*24*30*12*(Year(#Now())-1999)>

3. <CFSET session.day=#DateFormat(Now())#>

4. <CFSET session.time=#TimeFormat(Now())#>

5. <CFSET session.host=cgi.remote_addr>

6. <CFSET session.keyword=#form.keyword#>

7. <H2>Demographic Form</H2><P>

8.

9. <p>The information you give on this form is needed for classifying your answers about the presentations in demographic and social categories such as male-female, age group, etc. Your identity will not be revealed by your answers. </p>

10. <i>Please click at the arrow symbol at the right hand side of each reply box to get the reply options displayed and make your selection.</i> <p>

11. <FORM ACTION="first_chart.cfm" METHOD="POST">

12. <TABLE>

13. <TR>

14. <TD>

15. <I>Is your gender:</I><P></TD>

16. <TD><SELECT NAME="gender">

17. <OPTION value="0">Unanswered

18. <OPTION Value="1">Male

19. <OPTION Value="2">Female

20. </SELECT></TD>

21. </TR>

22. <TR>

23. <TD><I>Is your age:</I><P></TD>

24. <TD><SELECT NAME="age">

25. <OPTION value="0">Unanswered

26. <OPTION value="1">19 years or less

27. <OPTION value="2">20-24 years

28. <OPTION value="3">25-29 years

29. <OPTION value="4">30-49 years

30. <OPTION value="5">50 or more

31. </SELECT></TD>

32. </TR>

33. <TR>

34. <TD><I>Are you studying/working in:</I></TD>

35. <TD><SELECT NAME="subject">

36. <OPTION value="0"> Unanswered

37. <OPTION value="1"> Humanities (incl. religious and humanitarian services)

38. <OPTION value="2"> Social sciences (incl. law and business)

39. <OPTION value="3"> Life sciences (incl. medicine, ontology, psychology)

40. <OPTION value="4"> Natural sciences (incl. engineering, construction)

41. <OPTION value="5"> Agriculture, fisheries, industry

42. <OPTION value="6"> Business

43. <OPTION value="7"> Services (incl. work in the home)

44. <OPTION value="8"> Central and local government

45. <OPTION value="9"> Other

46. </SELECT></TD>

47. </TR>

48. <TR>

49. <TD><I>Are you living in:</I><P></TD>

50. <TD><SELECT NAME="resident">

51. <OPTION value="0"> Unanswered

52. <OPTION value="1"> Denmark

53. <OPTION value="2"> Norway

54. <OPTION value="3"> Sweden

55. <OPTION value="4"> Other European countries

56. <OPTION value="5"> United States

<OPTION value="9"> Other countries

57. </SELECT></TD>

58. </TR>

59. </TABLE>

60. If you want to continue, click the button

61. <INPUT TYPE="SUBMIT" NAME="submit" VALUE="CONTINUE">,

62. please be patient while the display is compiled,

63. <p>else leave for the URL you prefer.</p>

64. </FORM>

Demographic Form

The information you give on this form is needed for classifying your answers about the presentations in demographic and social categories such as male-female, age group, etc. Your identity will not be revealed by your answers.

Please click at the arrow symbol at the right hand side of each reply box to get the reply options displayed and make your selection.

Is your *gender*:

Is your *age*:

Are you *studying/working* in:

Are you *living* in:

If you want to continue, click the button , please be patient while the display is compiled, else leave for the URL you prefer.

Figure 3: Collecting background data

[Figure 3](#) shows how this form appears to the user. When submitted, this template calls upon the template *first_chart.cfm*.

Displaying the first chart

The template *first_chart.cfm* has 2 tasks:

- take care of and **save** the data submitted by the participant's completed demographic form.
- **prepare** for sending the first chart presentation to the participant.

The **first task** requires the **CFQUERY** tag in Line 3 followed by an **INSERT INTO** statement. This statement specifies all the **variables** we want to insert into the table **form1** of database *#session.datasource#* and their **values**. The first 4 are values of **session variables** defined in the previous template. The next 4 are values submitted by the participant on the demographic form. Note that we refer to these by preceding their names with the qualifier *form* because they were submitted by the last read form. The value of the keyword variable was also defined as a session variable in the previous template. All variables are introduced to the database as texts, i.e. they are surrounded by single quotes.

1. <!-- first_chart.cfm --->

2. <!-- Store answers from Participant's form -->

3. <CFQUERY NAME="ADDFORM1" DATASOURCE="#session.datasource#">

4. INSERT INTO form1 (participant,day,hour_minute,host,gender,age,subject, resident, keyword)
VALUES('#session.participant#', '#session.day#', '#session.time#', '#session.host#', '#form.gender#',
'#form.age#', '#form.subject#', '#form.resident#', '#session.keyword#')

```

5. </CFQUERY>

6. <!-- Randomize and draw a random first chart number. The parameters must be set 1 and 75 where 75 is
the number of facts-->

7. <CFSET random=RANDOMIZE(#session.participant#)>

8. <CFSET session.chart=#RANDRANGE(1,75)# >

9. <!-- Retrieve the first fact number, type and url corresponding for chart number -->

10. <CFQUERY NAME="SELECT1" DATASOURCE="#session.datasource#">

11. SELECT chart,fact,type,url FROM charts WHERE chart=#session.chart#

12. </CFQUERY>

13. <!-- Set session variable values -->

14. <CFSET session.fact= #select1.fact#>

15. <CFSET session.type=#select1.type#>

16. <CFSET session.url= #select1.url#>

17. <!-- Update calls with the participant identification and the fact number displayed -->

18. <CFQUERY NAME="INSERTCALL" DATASOURCE="#session.datasource#">

19. INSERT INTO calls(participant, fact) VALUES('#session.participant#','#session.fact#')

20. </CFQUERY>

21. <!-- Set the initial value of session.counter --->

22. <CFSET session.counter =1>

23. <!-- The image should be displayed for session.wait sec. --->

24. <CFSET session.wait=#RANDRANGE(10,20)#>

25. <CFOUTPUT>

26. <META HTTP-EQUIV="Refresh" CONTENT="#session.wait#; URL=back.cfm">

27. </CFOUTPUT>

28. <--- first_chart --->

29. <!-- Display first image, and refresh with form2.cfm -->

```

30. <H2>This is the first presentation of 25:</H2><P><P>

31. <!-- Present the display selected: -->

32. <CFOUTPUT>

33.

34. </CFOUTPUT>

35.

36. The presentation will be removed after a randomly determined time.

The **second task** of the template is **preparing** for the display of the first chart. This is the core of the application, and rather complex. Be patient, it is **not** that difficult as it may look at the first glance. There are also some comments included which should help to understand the algorithm.

The first step is to use the **RANDOMIZE(#session.participant#)** function to provide the system with a seed for the pseudo-random algorithm which will be used. Since the value of the **session.participant** variable is unique, it is an ideal for use as a seed. Note that this function does not randomize the **session.participant** variable, it prepares for the use of the following **RANDRANGE** function. The first chart to be presented is determined by the function **session.chart=RANDRANGE(1,75)** which provide a random integer in the range 1 to 75 (Recall that the number of different presentations charts in the gallery collection is $3 \times 25 = 75$). To find the location of the chart, the fact it represents and its type, requires a **search** (Lines 10-12) using SQL statement **SELECT** in the database **#session.datasources#** **FROM** table **charts** for the row **WHERE** chart value is **#session.chart#**. The location of the chart is contained in the variable **url**. We need to refer to this query and name it **select1**.

The 3 variables **session.fact**, **session.type** and **session.url** are set and assigned the values retrieved from the **query** named **select1** in Line 14-16. For **future** use, the table **calls** must be updated with the values of **#session.participant#** and **#session.fact#** to **avoid** presenting the participant with the same fact more than once. This we do with a simple **CFQUERY** with an **INSERT INTO** statement in Line 19.

session.counter is a variable used to keep track of the **number of presentations** sent to the participant. At this point it is initialized with value **1**. In Line 24 of the template, the length of the **time** in seconds the current presentation should be **exposed** is determined as a random integer between **10** and **20** seconds.

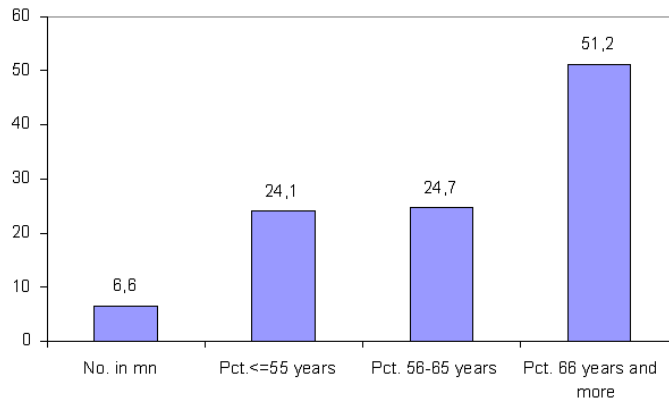
Line 26 contains a particularly **interesting** **HTML** statement. The **META** statement will keep the presentation of the current presentation (not yet presented!) on the client's screen for **#session.wait#** seconds, before it passes the control to the **form2.cfm** through a small template, **back.cfm**, (not reproduced here) with a warning against the temptation to use the back-button. To simplify, you can substitute **back.cfm** with **form2.cfm** in Line 26.

We are now approaching the end of the preparations for sending the selected presentation to the client. Lines 30 to 36 take care of this final purpose of this template. As you see from Line 33, the presentation image is stored at the location to which the URL `#session.url#` points. [Figure 4](#) show a random presentation.

This is the first presentation of 25:

Severely handicapped persons in Germany 1997.

(Source: Federal Statistical Office, Germany)



The presentation will be removed after a randomly determined time.

Figure 4: Presentation example

Perception report form

By means of the previous template the presentation is displayed for the participants for the randomly determined time period and the control redirected to the template `form2.cfm` for display of **form 2**. The main purpose of `form2.cfm` is to **collect the perception report** from the participant. A few other facts are also recorded, for example `session.startform2` for recording the start time for working on **form2** in Line 2.

1. `<!-- Form2 returned to the visitor: --->`

2. `<CFSET session.startform2=#LSTimeFormat(Now(),"HH:MM:SS")#>`

3. `<H2>Report Form</H2>`

4. `<P><P>`

5. `Please do not get tempted to use your Back button, it may ruin your participation.<P>`

6. `<p>This is your report on the presentation which was displayed for you. Some presentations are intentionally displayed for only a short period in order to test for how long time such a presentation should be`

displayed, f.ex. on TV. It is not expected that you always can give answers to all questions. If the presentation was not displayed, leave the form blank.</P>

7. <i>Click at the arrow symbol at the right hand side of each reply box to get the reply options displayed and make your selection.</i> Please take the time you need to complete the report.<P><P>

8. <!-- This is the call for the template to record the answers in the data base and to generate the next display: -->

9. <FORM ACTION="next_chart.cfm" METHOD="POST">

10. <TABLE>

11. <TR>

12. <TD><I>Was the topic of the presentation within the domain of:
</I></TD>

13. <TD><SELECT NAME="topic">

14. <OPTION Value="0">Unanswered

15. <OPTION value="1">Demographic, social and health conditions

16. <OPTION value="2">Education, culture and art, religion

17. <OPTION value="3">Work, employment, professions

18. <OPTION value="4">Production, services, economy

19. <OPTION value="5">Research and development

20. <OPTION value="6">Entertainment, tourism, traveling

21. <OPTION value="7">Environment, climate, natural resources

22. <OPTION value="8">Government, elections, politics

23. <OPTION value="9">Several domains, other

24. </SELECT></TD>

25. </TR>

26. <TR>

27. <TD><I>In which measure the presented fact expressed
by:</I></TD>

28. <TD> <SELECT NAME="measure">

29. <OPTION Value="0">Unanswered

30. <OPTION value="1">Numbers, values (e.g. 3000 persons, 400 dollars)

31. <OPTION value="2">Percentages, fractions, points (e.g. 30%, 1/4 of population, 110.9 (1990:100)

32. <OPTION value="9">Other

33. </SELECT> </TD>

34. </TR>

35. <TR>

36. <TD><I>Were the facts related to a single or multiple point(s)/period(s) of time:</I></TD>

37. <TD><SELECT NAME="tid">

38. <OPTION Value="0">Unanswered

39. <OPTION value="1">Single point/period(e.g. Dec.31 1999, Year 2000)

40. <OPTION value="2">Multiple points/periods(e.g. Dec. 31 1999 and 2000, Year 1900 and 2000)

41. </SELECT></TD>

42. </TR>

43. <TR>

44. <TD><I>Were the facts presented in a single or multiple categories:</I></TD>

45. <TD><SELECT NAME="category">

46. <OPTION Value="0">Unanswered

47. <OPTION Value="1">A single category(e.g.. one country)

48. <OPTION value="2">Several categories (e.g. male/female, age classes, geographical areas)

49. </SELECT>

50. </TD>

51. </TR>

52. </TABLE>

53. <p>Thank you so far!</P>

54. If you want to continue, click the button

55. <INPUT TYPE="SUBMIT" NAME="submit" VALUE="CONTINUE">,

56. (please be patient while the display is compiled),

57. else go to the URL you prefer.

58. </INPUT>

59. </FORM>

In general, you find *form2.cfm* is **long** but rather trivial. After an introductory text, the starting form tag is on Line 9 specifying *next_chart.cfm* as the template to be called on when the form is submitted. The form has an ordinary structure with a table tag and ends at Line 59. The form as presented for the participant can be seen in [Figure 5](#).

Report Form

Please do not get tempted to use your Back button, it may ruin your participation.

This is your report on the presentation which was displayed for you. Some presentations are intentionally displayed for only a short period in order to test for how long time such a presentation should be displayed, f.ex. on TV. It is not expected that you always can give answers to all questions. If the presentation was not displayed, leave the form blank.

Click at the arrow symbol at the right hand side of each reply box to get the reply options displayed and make your selection. Please take the time you need to complete the report.

Was the **topic** of the presentation within the domain of:

In which **measure** the presented fact expressed by:

Were the facts related to a **single** or **multiple** point (s)/period(s) of time:

Were the facts presented in a single or multiple **categories**:

Thank you so far!

If you want to continue, click the button , (please be patient while the display is compiled), else go to the URL you prefer.

Figure 5: Report form

Next chart

The *next_chart.cfm* is similar to the *first_chart.cfm* template. The main difference is that the *first_chart.cfm* has to take care of saving the submitted data from the **demographic form** while the *next_chart.cfm* must save the data submitted from the **perception report form**.

The *next_chart.cfm* template **starts** by logging the *#session.endform2#* which is the **time** at which the form2 is completed and submitted. The **CFQUERY** block in Lines 4 to 6 saves the data in the **db** by a **sql** statement **INSERT INTO**


```

1.<!-- next_chart.cfm. -->

2.<CFSET session.endform2=#LSTimeFormat(Now(), "HH:MM:SS")#>

3.<!-- Store input from form2: -->

4. <CFQUERY NAME="INSERT3" DATASOURCE="#session.datasource#">

5. INSERT INTO form2(participant, chart, wait, fact,type,topic, measure,tid, category, startform2,
endform2)

VALUES ('#session.participant#', #session.chart#, #session.wait#, #session.fact#, #session.type#,
'#form.topic#', '#form.measure#', '#form.tid#', '#form.category#', '#session.startform2#',
'#session.endform2#')

6. </CFQUERY>

7.<!-- Test if the previous display was the last (no.25) --->

8. <CFIF #session.counter# EQ "25">

9. <--- If the previous display was no. 25 --->

10. <CFLOCATION URL="thanks.cfm">

11. <!-- If not the last fact, continue from here--->

12. <CFELSE>

13. <!-- Select a random chart number and test if fact already displayed: --->

14. <CFSET test="0">

15. <CFLOOP CONDITION =" #test# EQUAL 0 ">

16. <CFSET session.chart=#RANDRANGE(1,75)#>

17. <!-- Get the fact,type and url corresponding to chart number --->

18. <CFQUERY NAME="SELECT2" DATASOURCE="#session.datasource#">

19. SELECT fact,type, url FROM charts WHERE chart= #session.chart#

20. </CFQUERY>

21. <CFSET session.fact = #select2.fact#>

22. <CFSET session.type = #select2.type#>

23. <CFSET session.url = #select2.url#>

```

24. <!-- Check that fact no is not in calls from previous displays --->

25. <CFQUERY DATASOURCE="#session.datasource#" NAME="call">

26. SELECT fact FROM calls WHERE participant = '#session.participant#' AND fact= #session.fact#

27. </CFQUERY>

28. <CFIF #call.fact# NEQ #session.fact#>

29. <CFSET #test#="1">

30. </CFIF>

31. </CFLOOP>

32. <!-- A new chart number for an unused fact has been found -->

33. <!-- Record in calls that this fact has been used -->

34. <CFQUERY NAME="INSERTCALL" DATASOURCE="#session.datasource#">

35. INSERT INTO calls(participant, fact) VALUES ('#session.participant#','#session.fact#')

36. </CFQUERY>

37. <CFSET session.counter=#session.counter#+1>

41. <!-- The image should be displayed for session.wait sec.--->

42. <CFSET session.wait=#RANDRANGE(10,20)#>

38. <HTML>

39. <CFOUTPUT>

40. <META HTTP-EQUIV="Refresh" CONTENT="#session.wait#; URL=back.cfm">

41. </CFOUTPUT>

42. <CFOUTPUT>

43. <H2>This is presentation no.: #session.counter# of 25:</H2><P><P>

44.

45.

46. The presentation will be removed after a randomly determined time.

47. </CFOUTPUT>

48. </CFIF>

49. </BODY>

50. </HTML>

The **CFIF** tag in Line 8 tests if the just saved form was the last, i.e. number 25. If so, the control is directed to *thanks.cfm*.

The selection of a new presentation is more **complicated** than the first because we now have to check that the display drawn does not represent a fact already presented. Line 15 to 31 is a sequence of CF tags in which a display is randomly selected and tested, if the implied fact has already been used the sequence is repeated. When a presentation of an unused fact is found, the looping is finished and the processing continues for the new display as for the first display

Thanks to the participants

When a participant has responded to **25 presentations**, he/she is sent a page generated by the final template *tanks.cfm*. Again, this is a page without any **CF** tags and it qualifies as a **HTML** page including the reference to a mail address in Line 9.

1. <--- thanks.cfm --->

2. <HTML>

3.<BODY>

4. <H1>

5. SIS Statistical Information Systems</H1>

6. <P>You have completed the applications. Thank you very much for your patience and interest.</P>

7. <P>Please exit to the URL of your choice.</P>

8. If you have any proposals or comments, please send them to:

9. Svein Nordbotten

10. <!-- The session is terminated -->

11.</BODY>

12. </HTML>

[Figure 6](#) gives the last display for the participant who completes all 25 presentation reports.

SIS Statistical Information Systems

You have completed the applications. Thank you very much for your patience and interest.

Please exit to the URL of your choice.

If you have any proposals or comments, please send them to:

[Svein Nordbotten](#)

Figure 6: Thank you

The experiment discussed in this session has been carried out several times in different environments, and the experience for this type of experiments is encouraging.

Exercises

- a. Read **Chapter 10** in **RBB** about a **CFML** tag, **CFFORM**, which extends the functionality of the ordinary **HTML** tag **FORM** with a variety of control features.
- b. In the application discussed, the focus was on perception of socio-demographic facts. The framework used is general and can for instance be used for investigating the impact on web users from different layouts (background color, font sizes, animation, etc.) of web pages. Choose an aspect in which you have interest, and **re-write** the templates for an investigation of this aspect.
- c. Modify the templates to satisfy another experiment on the net.
- d. Consider the **changes** you must do in the templates if you do not have access to a database and have to rely on the **CFFILE** tag.
- e. There is more advanced solutions to avoid backtracking. Try to see if you can find one. If you succeed, inform the other students about the trick.

Session 6: Web search engine

A search engine

Among the most used tools on the web are the **search engines**. Through a systematic and continuous search, **huge indexes** with references to the addresses of hundreds of millions of web files are created and maintained. Also smaller organization may need search engines for their internal files. In this session, the development of the simple search engine is discussed and exemplified with a search engine for this course.

In contrast to the databases in which the data are nicely organized in tables, the type of data considered here are text files with varying content and length. Among the properties of text files are, in addition to the size, language, etc., the words included in the documents. The tool we discuss in this session is a **full text** search engine, which can be implemented by means of features in **ColdFusion**. The **basic** approach of a full text search engine for a (large) **set of text files** is an **index file** with one record for the different (important) words occurring in the set of texts. Each word record has attached **links** to the files in which the word occurs one or more times. By asking for files with one or several words contained, a search through the index file will give links to the **requested** files. Based on the numbers of occurrences in the requested files, **scores** can be computed.

Included in **ColdFusion** are the components for such a search engine. They originate from a system module called **Verity** included in **CFMX**.

To create search engine templates for your web system using **CFML**, you must take several steps:

1. **Prepare a menu**
2. **Register the collection of files which should be covered by the system**
3. **Index the files**
4. **Construct a query interface to the system**
5. **Prepare searching by means of the engine**
6. **Prepare an option for deleting a collection**

[Figure 1](#) indicates the system components. The right hand part of the figure represents how the search results first appear as references while the left side of the figure indicates that these references can be activated to retrieve the files searched. When the user receives the retrieved links, he should be able to select all or some of the links and retrieve the electronic documents in which he/she is interested.

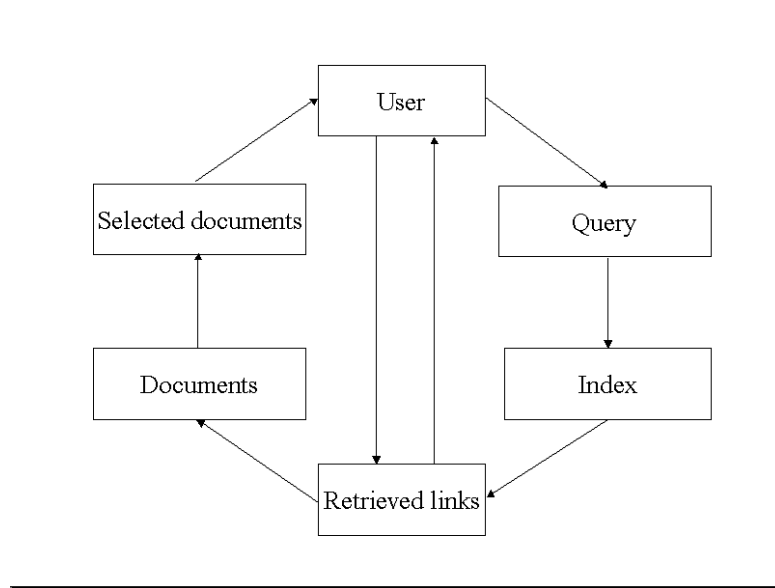


Figure 1: Outline of search system

In the following sections, the templates needed for implementing a search engine in **CFMX Verity** module are discussed.

Selection menu

The implementation of the full text retrieval system requires a usual *Application.cfm* template similar to those discussed in earlier applications (no datasource is needed). The first template, *index.cfm*, is a **menu** for selecting the required processes:

1. <!-- index.cfm -->
2. <h2>Search engine</h2>
3. <p>Select the process you want:</p>
- 4.<p>1. Define a collection of files </p>
5. <p>2. Indexing the files of a collection</p>
6. <p>3. Searching in a collection index</p>
7. <p>4. Deleting a file collection</p>

Search engine

Select the process you want:

1. [Register](#) a file collection
2. [Indexing](#) a file collection
3. [Searching](#) a file collection
3. [Deleting](#) a file collection

Figure 2: Search system menu

The menu, [Figure 2](#), generated by the template, displays 4 options,

- registering
- indexing
- searching
- deleting

Observe that a set of files representing text documents is assumed already **stored** on your disk for registration as a **collection**. Be careful with interpreting this term. For example, when the literature discussing creation of a collection, it does not mean creating the documents, but **defining** the location of the collection indexes, and when the documentation refers to **deleting** a collection, it means that the indexes and the definition of the collection are deleted, not the physical text files.

File collection

The collection, which we are going to define for this example, comprises the session text files of this course. Usually, the number of text files comprised by a collection is of course much higher.

The files can be of different types, distinguished by their extensions, **.htm**, **.html**, **.cfm**, **.cfml**, **.jpg**, **.gif**, **.pdf**, **.txt**, **.xls**, etc. Note that **graphical** files can also be included, but only indexed if they contain some text. Not all types of files are always relevant for the application. The first task is to distinguish between relevant and irrelevant files by extension during the indexing. In the example, we use only **.cfm** files.

The first step is naming the collection remembering that a collection is **not** the set of files themselves. The following template makes the required preparation for a collection:

Recording a collection

Name of collection:

Folder for collection:

Figure 3: Recording form

The *form_recording.cfm* displays a form, [Figure 3](#), requesting the information needed for definition of a collection:

1. `<!-- form_recording.cfm -->`
2. `<h2>Defining a collection</h2>`
3. `<form action="recording.cfm" >`
4. `<p>Name of collection:<input type="text" name="collection_name">`
5. `<p>Folder for collection:<input type="text" name="collection_path"> </p>`
6. `<p><input type="submit" value="Record"></p>`
7. `</form>`

The form requests 2 attribute values:

- What **name** should be attached to the collection,
- In which **folder** should the collection, i.e. the index files, be established

Recall that the **collection** consists of special index files **referring** to the files in the set in which we are interested, not the data files themselves. The collection path is therefore referring to the collection index.

The **FORM** tag leaves the process control to the template *recording.cfm* when the form is submitted. The *recording.cfm* template looks like this:

1. `<!-- recording.cfm -->`

2. <CFLOCK TIMEOUT="30" NAME="cfcollection_lock" TYPE="EXCLUSIVE">

3. <CFCOLLECTION ACTION="CREATE" COLLECTION="#collection_name#" PATH="#collection_path#" LANGUAGE="English">

4., </CFLOCK>

5. <p><h3>The collection is registered.</h3></p>

Keep in mind that the templates now discussed are tools for the **developer** to establish the search engine, **not** for the end users. Still, there might be several people working with the files, and to avoid any problems, this template uses the **CFLOCK/CFLOCK** tags. The **lock** makes certain that the developer can make the recording enclosed undisturbed by other developers and/or users. The **CFLOCK** in Line 2 has many possible attributes of which we **need** only 2 in the present application. **TIMEOUT** specifies the maximum time in seconds **CFMX** should wait to obtain a lock, not the duration of the lock. **TYPE** of lock and is set to "**EXCLUSIVE**", the alternative is "**READONLY**". An **exclusive** lock reserves the files completely for the developer's script while it is working within the locked area. **CFLOCK** tags should be used around all **read-write operations** for which there is a risk that 2 or more requests happen at the same time. Experience indicates that in applications with few users, this situation happens rather infrequently.

CFCOLLECTION tag has several options. The first is **ACTION**, which we give values "**create**" and "**delete**" in this example. The name of the **COLLECTION** is needed and received from the previous form, and so is also the attribute **PATH** which is the path of the folder in which the collection is to be established. **LANGUAGE** has "**English**" as default. If the text in the files are from other languages, an **International Language** add-on is available.

Indexing the files

The defined collection must be **populated/indexed** with data about the files in which we are interested. This is done by the process **indexing**. A second form, [Figure 4](#), is implemented for this purpose by the template *form_indexing.cfm*:

Indexing a collection

Name of collection:

URL path to source folder:

Full path to source folder:

File extensions:

Figure 4: Indexing form

1. `<!-- form_indexing.cfm -->`
2. `<h2>Indexing documents for a collection</h2>`
3. `<form action="indexing.cfm" >`
4. `<p>Name of collection:<input type="text" name="collection_name"></p>`
5. `<p>URL path to source folder:<input type="text" name="URLPATH"></p>`
6. `<p>Full path to source folder:<input type="text" name="KEY"></p>`
7. `<p>File extensions:<input type="text" name="extensions"></p>`
8. `<input type="submit" value="Index"></p>`
9. `</form>`

In addition to the name of the collection, the **URL path** and the **Full path** to the folder (not the connection folder!) containing the documents to be index are required. **Remember** the URL path must start with ***http://***. The full path of the same folder on the server should also be provided. This path starts with ***c:\.***, or another relevant disk reference. Finally, the extension(s) of the files to be included must be specified. The specification must be on the form ***.cfm, .pdf, .jpg***, etc. If more than one is needed, **comma** should be used as delimiter.

Multiple collections and **document folders** can also be specified with **comma** delimiters between names and paths.

From this form, the process control is left to the template *indexing.cfm*. This short template contains the very **powerful** tag **CFINDEX** tag in Line 3:

```
1. <!-- indexing.cfm --->

2. <CFLOCK TIMEOUT="30" NAME="cfindex_lock" TYPE="EXCLUSIVE">

3. <CFINDEX ACTION="UPDATE"
  COLLECTION="#collection_name#"
  TYPE="PATH"
  EXTENSIONS="#extensions#"
  RECURSE="YES"
  URLPATH="#urlpath#"
  KEY="#key#"
  LANGUAGE="English">

4. </cflock>

5. <p><h3><font color="Red">The collection is indexed.</font></h3></p>
```

For the reason given above for specification of collections, **CFLOCK**-/**CFLOCK** tags are used to enclose the indexing process because it can take some time if the number of files to be indexed is large. **Indexing** of the document files (populating the collection) is **completely** taken care of by means of the **CFINDEX** tag. The **ACTION** is specified to **"UPDATE"** which can be used both for **initialization** as well as **maintenance** of an index. **RECURSE** is set to **"YES"** indicating that the process should traverse all **sub folders** of text files if any. The remaining attribute values are obtained from the *form_indexing.cfm*.

Searching the collection

The previous processes have **established** the search engine. We are now ready to **prepare the use** of the search engine. A simple search form, [Figure 5](#), is the first step. It is implemented by the template *form_searching.cfm*:

```
1. <!-- form_searching.cfm --->

2. <h2><font color="Blue">Searching a collection</font></h2>

3. <form action="searching.cfm" >

4. <p>Name of collection:<input type="text" name="collection_name"></p>

5. <p>Search words:<input type="text" name="criteria">

6. <input type="submit" value="Search"></p>

7. </form>
```

Searching a collection

Name of collection:

Search words:

Figure 5: Search form

A search requires the **name** of the collection(s) and **search criteria**. A **simple search criteria** can be a **single word**, a **phrase**, **several words** delimited by comma between them as well as expressions based on the logical operators **OR**, **AND** and **NOT**. More complex search criteria can be created by using a special **Search Language** available in **CFMX VERITY** module.

The search data are sent for processing by means of the template *searching.cfm*. The core of this template is the **CFSEARCH** tag. This tag has a number of attributes. **Execution** of the tag also provides a number of variables about the search. Attributes in addition to those transferred from the search form, are **TYPE**, **STARTROW** and **MAXROW**. **TYPE** is given the value **"SIMPLE"** which is the most compact form, the alternative is **"EXPLICIT"** which is more flexible, but requires the criteria spelled out explicitly with all operators. **STARTROW** and **MAXROW** refer to the references retrieved. In our example we use the values **"1"** and **"10"**, respectively, for the two attributes.

The *searching.cfm* template is listed below. Following the **CFSEARCH** tag, the results of the search are sent for display by Lines 3-6. Line 5 makes use of 2 of the variables provided by the search process, i.e. **#collection_search.recordssearched#** and **#collection_search.recordcount#**. The value of the first variable informs about the total number of files referred to in the index, the second the number of files relevant according to the criteria specified in the previous form.

1. <!-- searching.cfm -->

```
2. <CFSEARCH COLLECTION="#collection_name#"
NAME="collection_search"
TYPE="SIMPLE"
CRITERIA="#criteria#"
STARTROW="1"
MAXROWS="10"
LANGUAGE="English">
```

3. <h3>The search gave the following results:</h3>

4. <CFOUTPUT>

5. <p>The collection contained #collection_search.recordssearched# files, and #collection_search.recordcount# files satisfying the search criteria "#criteria#".</p>

6. </CFOUTPUT>

7. <CFIF #collection_search.recordcount# LTE 0>

8. <CFOUTPUT> Sorry , no files were found for the search criteria "#criteria#".</CFOUTPUT>

9. <cfelse>

10. <p>These files found were:</p>

11. <table>

12. <tr><td>Score:</td><td>Link:</td></tr>

13. <CFOUTPUT QUERY="collection_search">

14. <tr>

15. <td>#collection_search.score#</td>

16. <td>#collection_search.url# </td>

17. </tr>

18. </CFOUTPUT>

19. </table>

20. </CFIF>

The remaining Line 7-20, controls a two way **branching** by a set of **CFIF-CFELSE-/CFIF** tags. The **decision** criteria in Line 7 select Line 8 if **no relevant** files were found. If **relevant** files were identified, Line 10-19 is executed. The results are presented in a **TABLE** construct with 2 columns: **Score** and **Link**. The score is a number in the **range** 0 to 1 where 1 is indicating the highest possible relevance. Link is a **sensitive** link to the actual file which can be **retrieved** by a click.

Deleting a registered collection

After some time a search engine can become obsolete. To make the system complete, a possibility to **delete** a collection with the contained indexes must also be included. The *form_deleting.cfm* template generates the necessary form:

1. <!-- form_deleting.cfm --->
2. <h2>Deleting a collection</h2>
3. <FORM ACTION="deleting.cfm" >
4. <p>Name of collection:<INPUT TYPE="text" NAME="collection_name"></p>
5. <p><INPUT TYPE="submit" VALUE="Delete"></p>
6. </form>

The only required information is the name of the collection. When the form, [Figure 6](#), is

Deleting a collection

Name of collection:

Folder of collection:

Figure 6: Deleting form

submitted, the template *deleting.cfm* takes care of the deletion:

1. <!-- deleting.cfm --->
2. <CFLOCK TIMEOUT="30" NAME="cfcollection_lock" TYPE="EXCLUSIVE">
3. <CFCOLLECTION ACTION="delete" COLLECTION="#collection_name#">
4. </CFLOCK>
5. <h3>The collection is deleted.</h3>

The *deleting.cfm* uses the same **CFCOLLECTION** tag as the *recording.cfm* did.

Final remarks

It is important to note that the hidden dynamics in this system, i.e. using the results of a search for output, cannot be done with any static tool.

A search engine of the type we have discussed in this session is an interesting tool generating a number of **research** questions. Problems, which can be raised, are how different groups express their search criteria? Do they learn over time using a search engine more efficient? How many users are able to express advanced search criteria? Are advanced search criteria more effective than simple? Should large text files be partitioned into smaller sub files to obtain more efficient searches?

Exercises

- a. Read **Chapter 16** in **RBB** about the **Advanced Techniques** and try to see possible improvements of the templates presented in this session.
- b. Copy all search engine templates to your **PC**. Prepare 3-4 small text files in English, or use some files you already have. Try to create a collection and to index the text files. Set up the search engine.
- c. Do explorative searches in your collection using words and logical expressions as search criteria. Study the Scores, and compare with the frequencies of the word appearances in you text file collection.
- d. Report on the Message board about your experience
- e. Developers interested in crawlers, should investigate **Web Spider**, a command-line utility, which can be found in the COLDFUSIONMX directory in your computer.

Session 7: e-learning

Web courses

In this session, implementation of web **courses** using **ColdFusion** is discussed. As a student of this course, you should be particularly well armed with good ideas from your personal experience. It is impossible to go through a complete course in detail. The course you are attending contains for example more than 1000 files of different types organized in a structure with about 180 folders. In this session, we concentrate on discussing a few essential problems common for most Web courses.

As an application example, a hypothetical web course on **Information Retrieval** is used. We assume that the following list can be used as a guide for our discussion:

- Course architecture
- Authorization and authentication
- Texts
- Illustrations
- Literature
- Evaluation

You find a link to the implementation of the example at the end of the session. You can either register yourself getting your own **PIN** code, or you can behave as already registered with e-mail "*dummy@dummy*" and PIN code "0".

Course architecture

Development of a web course, like any IT system, is an **art**. There are no absolute, proven rules for what is the right or the best approach. The more complex the objectives are, the more **elaborated** course structure will be required. In this example application, a folder with a flat organization of all needed files will be considered acceptable. All the files for the example is in a single folder (with the exception of a database located outside the directly accessible area and referred to as (**#session.datasource#**)).

Security considerations are important only in connection with course design. We use the course application as a case for discussing **authorization** and **authentication** of users which is a common task in many applications. Along the road, we shall also make comments to other forms of security. In [Figure 1](#), the overall organization for the example course is depicted. The figure indicates that topics we are particularly interested in discussing are authorization, authentication

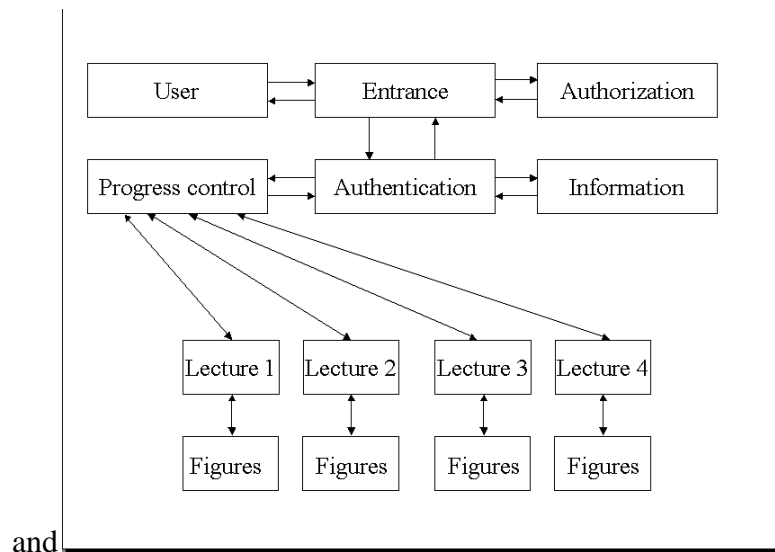


Figure 1:Course organization progress control.

Authorization and authentication

In mostly web courses, participation requires **authorization**, i.e. each participating student has been admitted by a sponsoring organization. By authorization, the participant receives some kind of **identification** to prove his/her right to enter the course. The requirements governing the authorization can vary from course to course. The technique of assigning the identification for proving the right to participate is a highly relevant subject to discuss in more detail.

The **first** template we shall discuss is the *index.cfm* which opens our example course scenario. It starts by a **welcome text** to **both** admitted and new, applying students. Line 3 differentiates actions for the two groups. If the caller is **new**, he/she is asked to go on for registration, while already **registered** students can proceed to the **login** as specified in Line 5-9.

Consider the login alternative first. The login process, [Figure 2](#), requires that the student types his/her **e-mail** address and personal **PIN** code which she/he received when registered. The

e-learning Information Retrieval

Thank you for your interest in this course on Information Retrieval. If you already have registered for the course, go on to the login. If you are new and want to join this course, we need some information from you, and you will need a personal identity number (PIN). Please click on [registration form](#).

If you already are registered, please continue

Access the course with your

PIN:

Click the button

Figure 2: Course entrance form

process of checking that a student is entitled to access the course as an admitted student is referred to as the **authentication**.

The *index.cfm* template is quite ordinary and looks like this:

1. `<!-- index.cfm -->`
2. `<h1> <i>e-learning</i> Information Retrieval</h1>`
3. `<p>Thank you for your interest in this course on Information Retrieval. If you already have registered for the course, go on to the login. If you are new and want to join this course, we need some information from you, and you will need a personal identity number (PIN). Please click on registration form. </p>`
4. `<p>If you already are registered, please continue </p>`
5. `<p>Access the course with your`
6. `<FORM METHOD="POST" ACTION="authentication.cfm">`
7. `<p>PIN: <INPUT TYPE="Password" name="submitted_pin" SIZE="17"></P>`
8. `<p>Click the button <INPUT TYPE="SUBMIT" NAME="response" VALUE="Submit"></p>`
9. `</FORM>`

Lines 6-7 of the template indicate that for **authentication**, the **e-mail** address and the **PIN** code are required. More identifiers, means a higher security, i.e. less risk for intruders.

Registration and authorization

If the student replies that he wants to **register**, the *registration.cfm* template is called ([Figure 3](#)):

Registration form

First name:

Family name:

E-mail:

All boxes must be completed for successful processing.

your registration.

Figure 3: Course registration form

1. `<!-- registration.cfm -->`
2. `<h2>Registration form</h2>`
3. `<form action="authorization.cfm" METHOD="post">`
4. `<pre>`
5. First name: `<INPUT TYPE="text" NAME="FirstName" SIZE="30">`
6. Family name: `<INPUT TYPE="text" NAME="LastName" SIZE="30">`
7. E-mail: `<INPUT TYPE="text" NAME="Email" SIZE="30">`
8. `</pre>`
9. `<p>All boxes must be compl hardware platform, and the limitations of the web server.`
10. `<p><INPUT TYPE="submit" VALUE="Submit"> your registration.</p>`
11. `</form>`

The **PIN** can either be self-composed, i.e. the person who request registration provides his/her own password, or it is assigned by the system. Self-composed **PINs** have the advantages that they may be easier for the owners to remember, and they can by special techniques (hashing) be kept secret also for the system staff. Compared with the system assigned passwords, the disadvantages of self-composed **PINs** are they may be easy to guess, and they cannot easily serve as internal identifiers. The above template has no field for providing self-composed **PINs** indicating that we have chosen to use system assigned identifiers.

The *registration.cfm* template leaves the control to the *authorization.cfm* template which can be **modified** in several ways. For example, the course **capacity**, which is set in Line 2, will depend on a number of factors as the nature of the course, the capacity of instructor. The **multiplier** set in Line 3 is another parameter, which can be changed. Increasing the value of the multiplier affects the security of authentication by reducing the risk that a valid password can be found by a potential intruder. Note that the value 5 of multiplier with random **PIN** generation used in this example, means that a potential intruder in average must make 5 trials to hit a valid **PIN** value because the size of the number space from which the codes are drawn, is 5 times the capacity. Since a valid **PIN** number has to be combined with the associated **email** address, the risk is considered low enough.

This application uses the datasource **#session.datasource#** with a table, **used_pin**, which has 2 columns, **email** and **pin**. For security reasons, the database is located outside the area available from the web and in our example specified in the *Application.cfm*.

The template has a **CFQUERY** tag named "list" with a **SELECT FROM** statement in Line 4. The **SQL** statement retrieves all values in column **pin**. The number of records in the table is available as the value of the variable **List.Recordcount**. In Line 7 is a **CFIF** tag test if the number of used **PIN**'s, i.e. the value **#List.Recordcount#**, already has reached the capacity limit. If so, a message about no vacant position is sent to the student. If there is still capacity, Lines 10-31 specify the further processing.

1. <!-- authorization.cfm -->
2. <cfset capacity="10">
- 3.<cfset multiplier="5">
4. <CFQUERY NAME="list" datasource="#db#">
5. SELECT pin FROM used_pin
6. </cfquery>
7. <CFIF #list.Recordcount# EQ #capacity#>
8. <h2>Sorry, the course has no vacant position.</h2>
9. <CFELSE>
10. <CFSET test="0">
11. <cfloop condition="#test# EQ 0">
12. <CFSET temp=randomize(second(Now()))>
13. <CFSET generated_pin=#RandRange(1, #multiplier*##capacity#)#>
14. <cfset test="1">

```

15. <CFLOOP QUERY="list">

16. <cfif #list.pin# EQ #generated_pin#>

17. <cfset test="0">

18. <cfbreak>

19. </cfif>

20. </cfloop>

21. <cfif #test# EQ 1>

22. <CFBREAK>

23. </cfif>

24. </cfloop>

25. <cfquery name="add_pin" datasource="db">

26. INSERT INTO used_pin(pin,email) VALUES('#generated_pin#','#email#')

27. </cfquery>

28. <cfoutput><h2><font color="Blue">Your PIN for the course</font></h2>

29. <p>You have been admitted to the course. Use the following PIN each time you log in to the course:
   <b>#generated_pin#</b></p>

20. </cfoutput>

31. </cfif>

```

The second part of the template, concerns the **generation** of a **PIN** code not previously used. A variable called **test** is set equal to **0**, followed by a **CFLOOP** block from Line 11-20. This loop runs as long as the condition "**#test# EQ 0**" is true. In Line 12-13, a **PIN** code is generated by a random generator in the range **1** to **#multiplier#*#capacity#**, and the variable **test** is set to "**1**". Then an **inner CFLOOP** block is inserted in Line 15-20 to check if the generated **PIN** is **free** and not already assigned to another student. .

The operation of this second **CFLOOP** block is **interesting** because referring to the name, "**list**", of the query, it automatically loops through **the query object**, i.e. all the retrieved PINs, and **compare** in Line 16 each used **PINs** with the new generated pin. If the new **PIN** code is found among the used pin codes, the variable **test** is again set to **0**, this inner **CFLOOP** is broken and the remainder of the outer **CFLOOP** in Line 21-24 is passed without any actions. The control is returned to Line 11. This **continues** until the inner loop is passed without being broken, which implies that the generated PIN is **unused**. In Lines 25-27, the new student is inserted into

database with **e-mail** address and **PIN** code, and in Lines 28-30, a **message** is generated for return to the student.

In a **real** life course, the registration form will usually be intercepted for an **off-line evaluation** against other criteria as previous training and grades. Following a positive external evaluation, the **PIN** code will then be generated and a message sent the applicant.

Authentication

If the student logging in has submitted his **e-mail** address and **PIN** code on the form generated by the *index.cfm* template discussed above, the process control is transferred to the template *authentication.cfm*. The purpose of this template is limited to check that the submitted **PIN** code exists.

This template starts by setting a variable **test** to **0** and queries the datasource for the list of all registered **pairs** of email and pin values. The query is named "**authentication**".

1. <!-- authentication --->
2. <cfset test="0">
3. <cfset submitted_pin="#form.submitted_pin#">
4. <cfquery datasource="#session.datasource#" name="authentication">
5. SELECT pin FROM used_pin
6. </cfquery>
7. <cfloop query="authentication">
8. <cfif #pin# EQ #submitted_pin# >
9. <cfset test="1">
10. <cfbreak>
11. </cfif>
12. </cfloop>
13. <cfif #test# EQ "1">
14. <cfset session.pin=#submitted_pin#>
15. <h3>Please, continue</h3>

16. <cfelse>

17. <h3>Your PIN code was not accepted.</h3>

18. </cfif>

As in the authorization template, this template also have a **CFLOOP** block in Lines 7-12 in which all retrieved **PIN** codes are compared with the submitted **PIN**. If one retrieved code matches the submitted, the variable **test** is set to "1" and the loop is broken. Line 13 contains the variable test. If **#test# EQ "1"** the authentication is positive. If the test condition is not true, Line 17 produces a **message** to the student that the **PIN** code was rejected.

The above templates demonstrate the principle. If we should implement the previously stated security policy, both the **registered email** address and the used **PIN** codes should be retrieved for each student, and compared with the **submitted email** address and **PIN** code.

List of content

After a positive authentication, template *content.cfm* displays a list of the course content for the student from which he/she can select a lecture ([Figure 4](#)). Note that Lectures 2-4 have no links

Requesting a news service

Our agent offers an email news service based on news from Washington Post and CNN every second hour. If you are interested in a particular topic, you can choose to be served with news from one or both sources. The agent is scanning the first page of the news sources every hour, and if your particular topic appears, a copy of the page is sent to you by email.

Warning: You will get the response as an html-page embedded in then email response. This may cause problems for some email-programs, and do not request too many days if you choose a frequently appearing topic, your mailbox can be filled up.

Your name:	<input type="text"/>
Your email address:	<input type="text"/>
The topic in which you are interested:	<input type="text"/>
No. of days you want the service:	<input type="text"/>
Source for the service:	
Washington Post:	<input type="radio"/>
CNN:	<input type="radio"/>
Washington Post and CNN:	<input type="radio"/>
	<input type="button" value="Submit"/>

Figure 4: News subscription form

since they are non-existent in the example.

1. <!-- content.cfm --->
2. <h2>Information Retrieval Course</h2>
3. <h1>Content:</h1>
4.
5. Lecture: Introduction
6. Lecture: Description and query language
7. Lecture: Document indexing
8. Lecture: File organization
9. Lecture: Search operation
10. Lecture: Evaluation
11. <p></p>
12. References
13. Figures
14.

The template is rather trivial and requires no further comments.

Associated with each lecture, a number of **special features** can be established. Access may, for example, be delayed to a certain **opening date** to avoid that the students rush through the course. The lecture can be finished by a **test** in which the student can check if he/she has read the lecture thoroughly and a pre-described test result must be achieved as a **condition to continue** with the next lecture. As a student of this **CFMX** course, you have personal knowledge about the functioning of the tests and their features. It is also easy to introduce closing dates for the lectures if wanted.

Lectures

From the list of contents, there are links to the different parts of the course. As illustration, only a few components are implemented in this example and listed below. On top of each template a **CFIF** tag has been included with a special condition, **IsDefined('session.pin')**. This tag tests if the client calling the page has been authenticated and a **session.pin** variable defined in Line 13 of

authentication.cfm. If this variable has **not** been defined, the control goes to **CFELSE** and to **CFABORT** at the end of the templates. This feature reduces the risk for arbitrary visits to individual pages of the course system.

Lecture 1 could look like this:

1. <!-- text.cfm -->

2. <cfif IsDefined('session.pin')>

3. <h1>A COURSE IN INFORMATION RETRIEVAL</h1>

4. <h2>Lecture 1:Introduction</h2>

5. <p>The topic information retrieval concerns the structure, analysis, implementation, search and dissemination of documents representing information. The purpose of an information retrieval system is to satisfy needs for information in a best possible way. </p>

6. <p>A typical modern information retrieval system is implemented in a host computer which can be accessed on internet from client computers. It is implemented with 2 sets of software, the client software and the server software.</p>

7. <p>The required client server is the basic software for working with the internet, while the server requires the general software to provide services on internet as well as specialized software for the information retrieval application. </p>

8. <p>The information retrieval application is build with a collection of documents as in an ordinary library or files as with a provider of electronic document representations as the core. To help the user to identify the documents in which he/she is interested, a set of files with meta data for the documents are developed and frequently organized in a database. In some applications, but far from all, even the electronic documents themselves can be included in the database.</p>

9. <p>To interact with the system, the user must use a query language which has been adjusted to the type of meta data in the database. The user must be able to describe the general properties of the unknown documents he/she wants to identify. On the other side, the retrieval system must be able to interpret the requests, communicate with the user for more details if necessary, and search in the system for the documents wanted. Figure 1 gives an overview of a retrieval system.</p>

10. <p>Depending on the users knowledge about the system, the components of the query language, the meta data for the documents included in the collection, and the composition of documents, the retrieval process may be more or less successful. To be able to compare one retrieval system application with a second, measures of performance are needed. For information retrieval, 2 measures, recall and precision, have been widely used.</p>

11. <p>If A is the subset of the documents which are relevant for a certain task expressed the query by Q, and B is the retrieved documents, the ratio $(A \text{ AND } B)/A$ is called the recall of the retrieval system for the query Q. The precision of the expressed Q for the same task is the ratio $(A \text{ AND } B)/B$. Since the evaluation of the recall in principle assumes that the set of relevant documents in the collection is known (if it is known, no retrieval problem exists), the set A has to be estimated. Precision, on the other hand, requires no knowledge outside the retrieved set B.</p>

12. <h3>Literature</h3>

13. `<p>Return to the Content.</p>`

14. `<cfelse>`

15. `<CFABORT>`

16. `</cfif>`

Note that the links to other texts, literature, figures, etc. are included as in a usual **HTML** page. [Figure 5](#) shows a part of the lecture.

Session 8: Web shop

e-business

One of the most talked about web applications is **e-shop**, **e-business** or **e-commerce**. Complete commercial systems are available from the shelf, new web shops have emerged and many have disappeared. Great expectations obviously exist for the future of web shops. These applications also demonstrate a number of web application aspects.

In this session we discuss and demonstrate some of the basic principles for a **web shop**. The example is a web shop, which are selling the **web scripts** we have introduced in this course. As all the other examples, our web shop application is not complete, and can be improved in many ways.

The **essential** templates of the application are discussed below. Some trivial templates as *conditions.cfm*, *shipping.cfm*, *support.cfm* and *about.cfm* are demonstrated in the running example available at the end of this session, but are not discussed below. It is **recommended** that you make yourself acquainted with the example before you start studying the templates in detail.

Business promotion

Operating a web shop requires product **promotion**, i.e. dissemination of information about the products offered, prices, sales conditions, shipping, information about the company and its addresses. In addition to distribution of information by **huge lists** of e-mail addresses and advertisements, a web shop must have a **home page** with links to required information and provide the possibility to order/buy products online. In our example, **Software Shop** has a home page generated by the template *index.cfm*. This homepage, [Figure 1](#), will serve as an introduction to this application.

Welcome to the Software Shop

The Software Shop has an exclusive suite of software for small companies. We have well satisfied customers and would be glad to see you among them. Please study our list of Product and if you find any item of interest, click for more details and price. You can buy the product safely on the net and the merchandise will be shipped to you according to the alternative you prefer.

- [Products](#)
- [Sales conditions](#)
- [Shipping](#)
- [Support](#)
- [About Software Shop](#)

Figure 1: Welcome page

In a fancy commercial application, the home page template should probably contain **icons**, **flash** or **applet driven animation**, etc. The main focus in our example is, however, the **dynamics** aspects. On the introductory page, a menu with links provides a list of the services offered to the customers. The example concentrates on **products, orders and sales**. On the Introductory page, the links to *products.cfm* and information templates are on Lines 5 - 9:

1. `<!-- index.cfm -->`
2. `<h1>Welcome to the Software Shop</h1>`
3. `<p>The Software Shop has an exclusive suite of software for small companies. We have well satisfied customers and would be glad to see you among them. Please study out list of Product and if you fin any item of interest, click for more details and price. You can buy the product safely on the net and the merchandise will be shipped to you according to the alternative you prefer.</p>`
4. ``
5. `Products`
6. `Sales conditions`
7. `Shipping`
8. `Support`
9. `About Software Shop`
10. ``

Except for the links, this page is of minimal interest in the context of dynamic web applications.

The *product.cfm* template generates a table with a row for each product. The template is a straight forward demonstration of the **HTML** table tag features.

1. `<!-- products.cfm -->`
2. `<h1>Products</h1>`
3. `<p>Our products cover web ColdFusion scripts. They all require a web server and a`
4. `ColdFusion application interface installed:</p>`
5. `<h3>Product list</h3>`
6. `<table border="2" cellpadding="10" cellspacing="10">`
7. `<th>`

```

8. <tr><td><b>Product name</b></td><td><b>Demonstration</b></td></tr>

9. <td><b>Price</b></td></tr></th>

10. <tr><td>Market research</td><td><a href=" ../omarket_research/">Demo</a></td><td>$
150,00</td></tr>

11. <tr><td>Opinion polls</td><td><a href=" ../opinion_polls/">Demo</a></td><td>$ 150,00</td></tr>

12. <tr><td>Perception investigation</td><td><a href=" ../perception/">Demo</a></td><td>$
300,00</td></tr>

13. <tr><td>Search engine</td><td><a href=" ../search_engine/">Demo</a></td><td>$ 200,00</td></tr>

14. <tr><td>Net course </td><td><a href=" ../course/">Demo</a></td><td>$ 350,00</td></tr>

15. <tr><td>Shop</td><td><a href=" ../shop/">Demo</a></td><td>$ 250,00</td></tr>

16. </table>

17. <p>Do you want to order? <a href="form.cfm">Yes</a></a href="index.cfm">No</a></p>

```

Figure 2 shows the table generated. Each product is listed with the possibility to get an online demo of the product. The table also informs the customers about the prices and has a link to ordering.

Products

Our products cover dynamic web applications. They all require a web server and a ColdFusion application server installed on your host computer:

Product list

Product name:	Demonstration:	Price:
Market research	Demo	\$ 150,00
Opinion polls	Demo	\$ 150,00
Perception investigation	Demo	\$ 300,00
Search engine	Demo	\$ 200,00
Net course	Demo	\$ 350,00
Shop	Demo	\$ 250,00
Web agent	Demo	\$ 150,00

Do you want to order? [Yes/No](#)

Figure 2: Product promotion list

Buying products

Buying products is taken care of by the *form.cfm* template which generates the form ([Figure 3](#)) by which necessary data about the customer are collected:

Customer form

If you are a new customer, please give the information necessary to send the products you buy to you:

First name:	<input type="text"/>
Last name:	<input type="text"/>
Street:	<input type="text"/>
City:	<input type="text"/>
State:	<input type="text"/>
Country:	<input type="text"/>
Zip no.:	<input type="text"/>
e-mail:	<input type="text"/>

Name and address

If you have previously bought products from us, please type your

e-mail address:	<input type="text"/>	e-mail address
-----------------	----------------------	----------------

Figure 3: Customer data

1. `<!-- form.cfm -->`
2. `<h1>customer form</h1>`
3. `<p>If you are a new customer, please give the information necessary to send the products you buy to you:</p>`
4. `<table cellpadding="10">`
5. `<form action="customers.cfm" >`
6. `<tr><td>First name: </td><td><input type="text" name="first_name"></td></tr>`
7. `<tr><td>Last name: </td><td><input type="text" name="last_name"></td></tr>`

```

8. <tr><td>Street: </td><td><input type="text" name="Street"></td></tr>
9. <tr><td>City: </td><td><input type="text" name="City"></td></tr>
10. <tr><td>State: </td><td><input type="text" name="State"></td></tr>
11. <tr><td>Country: </td><td><input type="text" name="Country"></td></tr>
12. <tr><td>Zip no.: </td><td><input type="text" name="zip"></td></tr>
13. <tr><td>e-mail:</td><td><input type="text" name="submitted_email"></td></tr>
14. <input type="hidden" name="sswitch" value="0">
15. <tr><td><td><input type="submit" value="Name and adress"></td></tr>
16. </table>
17. </form>
18. <p></p>
19. <p>If you have previously bought products from us, please type your</p>
20. <table cellpadding="10"></table>
21. <form action="customers.cfm" >
22. <tr><td>e-mail address: </td><td><input type="text" name="submitted_email"></td></tr>
23. <input type="hidden" name="sswitch" value="1">
24. <tr><td></td><td><input type="submit" value="e-mail address"></td></tr>
25. </form>

```

The **HTML FORM-/FORM** tags for new customers extend on Lines 5-17 and collect **name** and **address**. Another form block, on Lines 21 – 25, for 'old' customers collects only the **e-mail** address. The application must be able to 'remember' the data provided, and to **check** the e-mail address and retrieve the information when a customer responds as an 'old' customer. To distinguish which of the two blocks is submitted, a 'hidden' variable **SSWITCH** (the double ss is used to avoid confusion with the reserved word **SWITCH**) is used with value "0" set in Line 14 if the customer is a new, and "1" set in Line 23 if he/she is an 'old'. Both blocks call on the template *customer.cfm* when submitted.

Before we proceed to the *customer.cfm* template, we describe a **database table**, **customers**, which have to be defined in the **#session.datasource#**. . The table must have the columns:

- First_Name

- Last_name
- Street
- City
- State
- Country
- Zip
- Email

All variables are defined as text variables.

The *customer.cfm* template has 3 tasks.

- update the database table customers with the data about the customer if he is 'new' (Lines 3. - 5.),
- retrieve customer data from the database for display if he is an 'old' customer (Lines 7. -9.)
- display recorded customer data for the client.

1. <!-- customer.cfm -->

2. <cfif #sswitch# EQ "0">

3. <cfquery datasource="#session.datasource#" name="add">

4. INSERT INTO customers(first_name,last_name,street,city,state,country, zip, email) VALUES ('#first_name#','#last_name#','#street#','#city#','#state#','#country#','#zip#','#submitted_email#')

5. </cfquery>

6. </cfif>

7. <cfquery datasource="#session.datasource#" name="retrieve">

8. SELECT customer_id,first_name,last_name,street,city,state,country,zip,email FROM customers WHERE email='#submitted_email#'

9. </cfquery>

10. <p>You are recorded with the following data:</p>

11. <cfoutput query="retrieve">

12. <table>

13. <tr><td>First name:</td><td>#first_name#</td></tr>

14. <tr><td>Last name: </td><td>#last_name#</td></tr>

```

15. <tr><td>Street: </td><td>#street#</td></tr>

16. <tr><td>City:</td><td>#city#</td></tr>

17. <tr><td>State:</td><td>#state#</td></tr>

18. <tr><td>Country:</td><td>#country#</td></tr>

19. <tr><td>Zip:</td><td>#zip#</td></tr>

20. </table>

21 </cfoutput>

22. <cfset session.customer_id=#retrieve.email#>

23. <p>Are the above data correct? <a href="order.cfm">Yes</a> / <a href="form.cfm">No</a></p>

```

In Line 2 a **CFIF** tag checks if the data received concern a **new** customer. If so, **CFQUERY** tag for the datasource **#session.datasource#** followed by an **SQL INSERT INTO table customers**, inserts the submitted data.

Both new and old customers forms are then processed according to Lines 7-23. A **CFQUERY** tag block in Lines 7-9 named **"retrieve"** with a **SELECT** from table **customers** the row **WHERE email='#submitted_email#'**. Usually only one row is retrieved, but if the customer has submitted data as 'new' customer twice or more, there can be more than one record.

By means of a **CFOUTPUT** block referring to the **QUERY="retrieve"** in Lines 11-21, one or more sets of data for the e-mail address submitted is displayed ([Figure 4](#)). Finally, if the customer **accepts** the displayed data, the control is transferred to template **order.cfm**. Note that **#email#** is used as customer identification.

You are recorded with the following data:

```

First name: a
Last name: b
Street:    c
City:      d
State:
Country:   f
Zip:       g

```

Are the above data correct? [Yes/No](#)

Figure 4: Check customer data

This template presents the **form** for buying items with boxes to check for the different products. The template contains a set of multiple inputs ([Figure 5](#)). The values of the checked items are

Order

Please, mark the items you want to buy:

☐ Market research - \$150.00

☐ Opinion poll - \$150.00

☐ Perception investigation - \$300.00

☐ Search engine - \$200.00

☐ Net course - \$350.00

☐ Shop - \$250

☐ Web agent - \$150.00

Order

Figure 5: Order form

saved in a shopping list named **order**:

1. `<!-- order.cfm -->`
2. `<cfset order="">`
3. `<h1>Order</h1>`
4. `<p>Please, mark the items you want to buy:</p>`
5. `<form action="sum.cfm" method="post" >`
6. `<p><input type="checkbox" name="order" value="150">Market research - $150.00</p>`
7. `<p><input type="checkbox" name="order" value="150">Opinion poll - $150.00</p>`
8. `<p><input type="checkbox" name="order" value="300">Perception investigation - $300.00</p>`
9. `<p><input type="checkbox" name="order" value="200">Search engine - $200.00</p>`
10. `<p><input type="checkbox" name="order" value="350">Net course - $350.00</p>`
11. `<p><input type="checkbox" name="order" value="250">Shop - $250</p>`

12. <p><input type="checkbox" name="order" value="150">Web agent - \$150.00</p>

13. </select>

14. <p><input type="submit" value="Order"></p>

15. </form>

When the above form is submitted, the control is left to the *sum.cfm* template below. Here are several new **CFML** features introduced. In Line 3, an **array** of **items** is defined. In **ColdFusion** arrays can be defined with up to 3 dimensions. One great advantage with **CFMX** arrays is that the size (number of rows and columns) is not required specified, but is determined dynamically. In Line 4, the shopping list from the form is transferred to the array by means of the function **ListToArray(order)**. The reason is that an array can be manipulated more conveniently than a list.

In a **CFOUTPUT** block in Lines 5-9, the value of the variable **total** is computed in a **CFLOOP**.

1. <!-- sum.cfm -->

2. <cfset total="0">

3. <cfset items=ArrayNew(1)>

4. <cfset items=ListToArray(order)>

5. <cfoutput>

6. <cfloop from="1" index="count" to="#ListLen(order)#">

7. <p><cfset total=#items[count]##total#></p>

8. </cfloop>

9. </cfoutput>

10. <cfoutput>You have ordered products for \$#total#.</cfoutput>

11. <cfset session.total=#total#>

12. <p>Please return your creditcard information:</p>

13. <cfform action="sale.cfm">

14. <p>Card type:</p>

15. <p><input type="radio" name="card_type" value="VISA">VISA</p>

16. <p><input type="radio" name="card_type" value="Mastercard">MASTERCARD</p>

17. `<p><input type="radio" name="card_type" value="Ammerican Express">AMERICAN EXPRESS</p>`

18. `<p></p>`

19. `<p>Card number:<cfinput type="text" validate="creditcard" name="card_no"></p><p>`

20. `</p>Expire date: <input type="text" name="expire_date">`

21. `<p><input type="submit" value="Submit order"></p>`

22. `</cfform>`

The remaining of the template specify a form for the customer to specify credit card type, number and expire date ([Figure 6](#)). In Line 13, the tag **CFFORM** is used to be able to check the

You have ordered products for \$150.

Please return your creditcard information:

Card type:

☐ VISA

☐ MASTERCARD

☐ AMERICAN EXPRESS

Card number:

Expire date:

Figure 6:Payment form

credit card number. The attribute **VALIDATE** with value "**creditcard**" **checks** that the credit card number is a formally correct number (but unfortunately not for validity!).

Purchasing products

We shall need a **second database table**, **Sales**, which must be defined with the following 3 columns:

- customer_id
- total
- ddate

We assume that all are specified as text type.

Submitting the shopping list and the credit card information establish a **purchase**. The task of the template *sales.cfm* is to process the purchase. The **date** of the purchase is determined in Line 2 by using the value of the function **DateFormat(Now())**, and the transaction including customer identification, total value of the transaction and the date is booked in the datasource **#session.datasource#** by a **CFQUERY** tag with the **sql INSERT INTO**.

1. <!-- sale.cfm -->
2. <cfset ddate=#DateFormat(Now())#>
3. <cfquery name="sales" datasource="#session.datasource#">
4. INSERT INTO sales(customer_id,total,ddate) VALUES(#session.customer_id#, '#session.total#', '#ddate#')
5. </cfquery>
6. <cfoutput>
7. <p>Sales for customer no. #session.customer_id# has been processed.</p>
8. </cfoutput>
9. <p>Print out a receipt.</p>

The template sends a message that the transaction has been processed, and offers a printout of a receipt:

1. <!-- receipt.cfm -->
2. <cfquery datasource="#session.datasource#" name="receipt">
3. SELECT first_name,last_name, street, city,state,country, zip FROM customers WHERE customer_id = '#session.customer_id#'
4. </cfquery>
5. <h1>Software Shop</h1>
6. <cfoutput query="receipt">
7. <p>#first_name# #last_name#</p>
8. <p>#Street#</p>
9. <p>#City#</p>
10. <p>#State#</p>

11. `<p>#Country#</p>`
12. `<p>zip</p>`
13. `<p></p>`
14. `<p>We have charged your account with $$session.total# for products sent to you.</p>`
15. `<p> Thank you for ordering our products.</p>`
17. `</cfoutput>`
16. `<p>Yes</a href="index.cfm">Return to the product page. </p>`

The output **receipt** is illustrated in [Figure 7](#).

Software Shop

svein nordbotten

birkelundsbakken 11a

paradis

norway

zip

We have charged your account with **\$150** for products sent to you.

Thank you for ordering our products.

[Return](#) to product page.

Figure 7: Purchase receipt

To be a real functional application, an agreement has to be signed with a credit card company and a connection established with company.

Exercises

- a. In this session we have used the datatypes **list** and **array**. Read **Chapter 6** in **RBB** about these and other complex datatypes included in **CFML promotion**.
- b. Copy the templates and rewrite them for your favorite business idea. Complete and expand the 'surrounding' pages.

Session 9: Web agents

Web agents

An **agent** is a piece of software performing a repeating service according to a timetable without needing to be requested each time. On the web, an agent is located in a server from which it serves its clients. **CFMX** has features which make the language well suited for implementing such types of services. Even though many types of agents exist, we shall consider only two applications of agents leaving others for you to design.

The first application scenario is simple: Assume the existence of a **news agency** connected to the net. It updates frequently a news message page for a large range of different events. An organization, running an intranet, wants to offer the local intranet users a **mirror** of the news agency page to avoid unnecessary visits to the internet by the users of the intranet. One solution is to implement a **web agent** which periodically scans the news agency page, creates and maintains a **local mirror** which the local users can access.

In principle, the agent outlined is composed of:

1. the **retriever** controlled by the **scheduler**, and
2. the **dispatcher**

.

The retriever **collects and updates** the news data according to a frequency controlled by the **scheduler**. The time scheduler sets start and end **times** as well as **frequency** of the news collection. The dispatcher **returns** the news on **demand** from the clients.

In applications discussed so far in this course, activities were carried out when requested. A web agent application can work independently of requests, but it requires **adequate server resources** to avoid that it will be too engaged in performing agent tasks, and neglecting calls from clients for on demand service.

Agent 1

In the first application, **Agent 1**, a very basic service is discussed. A more advanced client, **Agent 2** is discussed in the last part of this session.

The first task considered is to retrieve the **News** from the **CNN** net service once every second hour, maintaining an updated local copy which can be requested by clients on **demand**.

The application includes templates for the following tasks:

- the news **collecting** and **maintenance** part
- the setting of the **scheduling** part
- the news **service** part

[Figure 1](#) outlines the scenario.

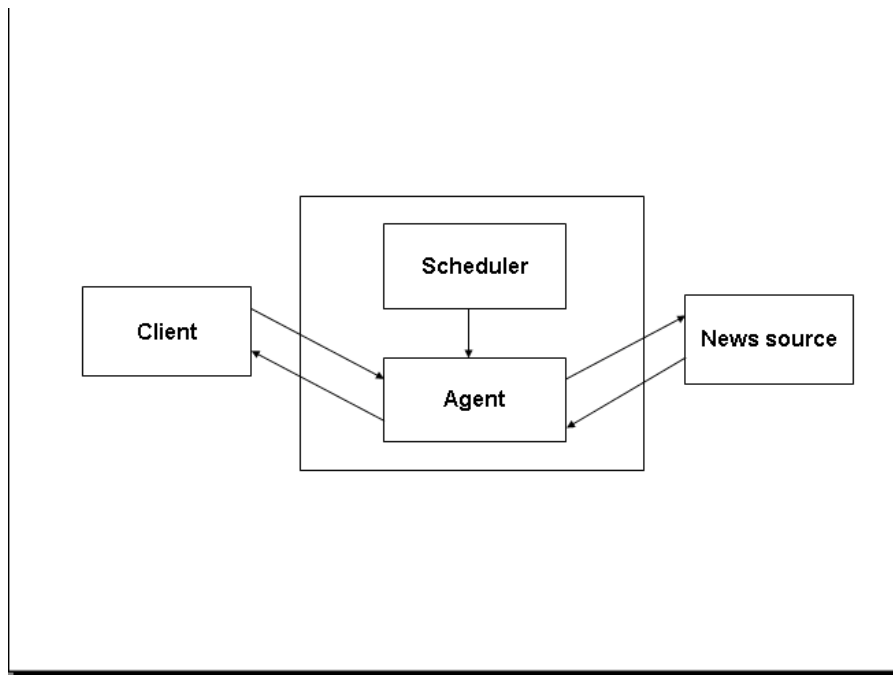


Figure 1: Agent 1 application.

The core of the first template is the powerful **CFHTTP** tag which can be compared with a **CFQUERY** tag, but with the important difference that the **CFHTTP** tag queries other web servers, and not a database.

1. <!-- agent.cfm -->
2. <cfhttp method="GET" url="http://www.cnn.com/" resolveurl="Yes" >
3. </cfhttp>
4. <cffile action="write" file="c:#application.path#\retrieved.html" output="#CFHTTP.FileContent#"> >

The **CFHTTP** tag in Line 2 indicates that we want to retrieve the **CNN** entry page in order to establish a local mirror of the **CNN** page. The attribute **RESOLVEURL**, set to "yes", resolves **URL**'s within the mirror page so they will also **function** for the client environment. The page is returned in a variable called **CFHTTP.FileContent**.

The content is saved in Line 4 to maintain a file named **retrieved.html**. The particular disk address at which the content is stored, can be set in the **Application.cfm** as an application wide variable **application.path**.

The agent **administrator** will need a form to specify the **start** and **end** times and the frequency of the news collection. This form is used only **once** for starting up the agent's activity, and eventually for terminating the service before specified. The template taking care of this task is:

```

1. <!-- schedule_form.cfm -->

2. <h2><font color="Blue">Scheduling or deleting the news collection</font></h2>

3. <p>a. Scheduling data collection:</p>

4. <form action="schedule.cfm" method="post">

5. <p>Startdate (mm/dd/yy):<input type="text" name="STARTDATE"></p>

6. <p>Starttime (hh:mm AM/PM):<input type="text" name="STARTTIME"></p>

7. <p>Update interval(sec):<input type="text" name="INTERVAL"></p>

8. <p>End date (mm/dd/yy):<input type="text" name="ENDDATE"></p>

9. <p>End time (HH:mm AM/PM):<input type="text" name="ENDTIME"></p>

10. <p>Timeout for request:<input type="text" name="TIMEOUT"></p>

11. <input type="submit" value="Schedule">

12. </form>

13. <p><a href="delete.cfm">Stop </a> the news collection. </p>

```

Note that the template refers to 2 actions, the template to **schedule** the agent in Lines 3-12, and the template to **delete** an existing time schedule in Line 13.

The *schedule.cfm* template for processing the scheduling data is short:

```

1. <!-- schedule.cfm -->

2. <cfschedule action="UPDATE"
task="Agent"
operation="HTTPRequest"
startdate="#startdate#"
starttime="#starttime#"
interval="#interval#"
enddate="#ENDDATE#"
endtime="#ENDTIME#"
url="http://application.url#/agent.cfm"
resolveurl="Yes" requesttimeout="120">

```

The template is simple, but has a number of attributes of which only *ENDDATE* is optional. It contains a variable, *application.url* which must be set in the *Application.cfm*.

If you have access to the **CFMX Administrator**, the schedule can alternatively be set by means of that utility. However, if you are renting time from an **ISP**, you will usually not have access to the **Administrator** feature.

If the stop option is selected in the *schedule_form.cfm*, the following template is run:

1. <!-- delete.cfm -->
2. <cfschedule action="delete" task="Agent" >
3. <cfoutput>
4. <H3>Agent task deleted.</H3>
5. </cfoutput>

The final component of the news service is a form for the clients to **request** the news:

1. <!-- index.cfm -->
- 2.<p><h2>Information collection by agent</h2></p>
3. <p>Do you want a news report? Yes/ No.</p>

where the link in Line 3 is all that is needed to respond to the request by serving the page *retrieved.html*..

In a real situation, a more elaborate form would be designed, but for demonstrating the principles of web agents, this simple form will do. As usual, you will find a link to an implemented example of the **agent** at the end of the session.

Advanced agent

Let us consider a more **advanced** agent. In addition to collecting news information regularly from one or **more** sources, **Agent 2** accepts **subscriptions** for news pages containing topics specified by the user in the form of keywords. Each time a news page is retrieved from the news source, the agent reads through the page to see if it contains any of the topics requested by the subscribers, and if so, it e-mails a copy of the news to the lucky subscribers.

On request, this agent sends a **subscription** form to a user asking about his/her name, email address, topic and source(s) of interest, as well as time interval for the wanted service. By submitting the form, a subscription is recorded by the agent, and the service will start. In this example, only a single topic per request is possible, but servicing more complex requests is quite possible.

In the application example demonstrated, two news sources, **CNN** and **Washington Post**, are copied every third hour. Each time, the copies are scanned for keywords provided by the

subscribers. Each time a hit is detected, the agent sends the subscriber(s) who provided the keyword, an **email** with a copy of the relevant news page attached.

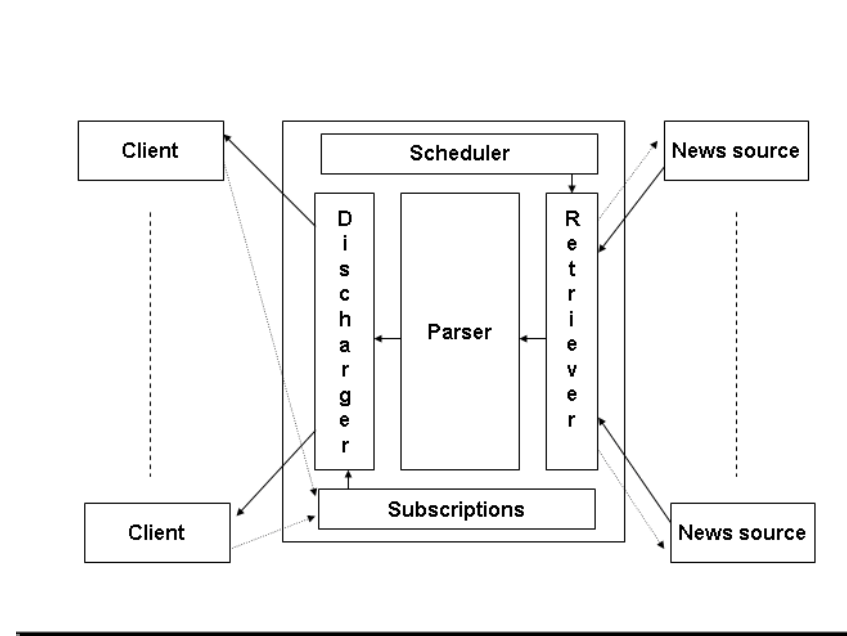


Figure 2: Components of Agent 2

[Figure 2](#) gives an overview of the application. This agent system consists of 5 logical parts

- the scheduler
- the news retriever
- the subscription service
- the parser
- the discharger

The parts will be discussed in a slightly different order in the following paragraphs.

Before the agent can be activated, the **news sources** and **retrieving frequency** must be specified. We focus on the scheduling and assume here that the 2 news sources, **CNN** and **Washington Post**, are already selected. The scheduling is similar to the one we described for the simpler **Agent 1** above, and implemented by the following form template:

1. <!-- schedule_form.cfm -->

2. <h2>Scheduling or deleting the news collection</h2>

3.<cfloop index="i" from="1" to="2">

```

4. <p>a. Scheduling data collection from source #i#:</p>
5. <form action="schedule.cfm" method="post">
6. <p>Startdate (mm/dd/yy):<input type="text" name="STARTDATE#i#"></p>
7. <p>Starttime (hh:mm AM/PM):<input type="text" name="STARTTIME#i#"></p>
8. <p>Update interval(sec):<input type="text" name="INTERVAL#i#"></p>
9. <p>End date (mm/dd/yy):<input type="text" name="ENDDATE#i#"></p>
10. <p>End time (HH:mm AM/PM):<input type="text" name="ENDTIME#i#"></p>
11. <p>Timeout for request:<input type="text" name="TIMEOUT#i#"></p>
12. <input type="submit" value="Schedule">
13. </form>
14.</cfloop>
15. <p><a href="delete2.cfm">Stop </a> the news collection from both sources. </p>

```

The *schedule_form.cfm* passes the control on to the *schedule.cfm*, which is identical with the template discussed in connection with the simpler agent. Also this agent can be scheduled by means of the **CFMX Administrator**.

The subscription can be taken care of by means of a very simple form illustrated in [Figure 3](#), and

Scheduling the service

Specify the scheduling data (End date and time may be left blank):

Startdate (mm/dd/yy):

Starttime (HH:mm:ss):

Update interval(sec):

End date (mm/dd/yy):

End time (HH:mm:ss):

Timeout for request:

Figure 3: Agent scheduling form

be implemented by the *service.cfm* template:

1. <!-- service.cfm --->
2. <FORM action="register.cfm" method="post">
3. <TABLE>
4. <TR><TD>Your name:</TD><TD><INPUT name="nname" type="text"></TD></TR>
5. <TR><TD>Your email address:</TD><TD><INPUT name="email" type="text"></TD></TR>
6. <TR><TD>The topic in which you are interested:</TD><TD><INPUT name="topic" type="text"></TD></TR>
7. <TR><TD>Source for the service:</TD><TD></TD></TR>
</TD></TR>
8. <TR><TD>Washington Post:</TD><TD><INPUT name="source" type="radio" value="1">
</TD></TR>
10. <TR><TD>CNN:</TD><TD><INPUT name="source" type="radio" value="2">
</TD></TR>
11. <TR><TD>Washington Post and CNN:</TD><TD><INPUT name="source" type="radio" value="3">
</TD></TR>
12. <TR>
13. <TD>No. of days you want the service:</TD><TD><INPUT name="days" type="text">
</TD></TR>
14. <TR><TD></TD><TD><INPUT type="submit" value="Submit"></TD>
15. </TR>
16. </TABLE>
17. </FORM>

This form indicates the similarity with a **search** system in which only one keyword is permitted in the search. An improvement of this form would be to permit **multiple** topic keywords, and topics which were structured into a **complex request** as is for example possible in the search engine discussed in a previous session.

The content of the subscription form is registered in a table called **agent2** at a datasource *#session.datasource#* set in the *Application.cfm* .:

1. <!-- register.cfm --->
2. <cfset MyDate=now()>

3. <cfset untill= DateFormat(DateAdd('d',#form.days#,MyDate),'mmmm dd yyyy')>

4. <CFQUERY name="register" datasource="#session.datasource#">

5. INSERT INTO agent2(nname, email,topic, untill) VALUES('#form.nname#',
'#form.email#','#form.topic#','#untill#')

6. </CFQUERY>

7. <cfoutput>

8. <H3>#form.nname#

9. Your request has been recorded.</H3>

10. </cfoutput>

Lines 1-6 take care of the registration in the database. If the registration is successful, a message is returned to the subscriber by email.

The main template of Agent 2 covers the **retrieval**, **parsing** and **discharging** of news to the subscribers:

1. <!-- agent2.cfm -->

2. <cfhttp method="GET" url="http://www.cnn.com/" resolveurl="Yes" timeout="300">

3. </cfhttp>

4.<cffile action="write" file="#application.path#\temp.html" output="#CFHTTP.FileContent#">

5.<cffile action="READ" file="#application.path#\temp.html" variable="retrieved">

6.<cffile action="write" file="#application.path#\retrieved.html" output="#retrieved#">

7. <cfhttp method="GET" url="http://www.washingtonpost.com/" resolveurl="Yes" timeout="300">

8. </cfhttp>

9. <cffile action="write" file="#application.path#\temp.html" output="#CFHTTP.FileContent#">

10. <cffile action="READ" file="#application.path#\temp.html" variable="retrieved2">

11. <cffile action="write" file="#application.path#\retrieved2.html" output="#retrieved2#" >

12. <cfquery name="subscribers" datasource="#application.datasource#">

SELECT nname, email, topic, untill, source FROM agent2

```

13. </cfquery>

14. <CFSET PresentDate=#DateFormat(now(),'mmm dd yyyy')#>

15.<CFLOOP query='subscribers'>

16. <CFIF #PresentDate# LT #subscribers.until# >

17.<cfif (#subscribers.source# EQ "1" OR "3") AND (#retrieved# NEQ "")>

18. <cfset source_name="CNN">

<19. < cfset text=Trim(#subscribers.topic#)>

20. <cfset Position=REFindNoCase(#text#,#retrieved#,1)>

21. <cfif #Position# GT 0>>

22. <cfmail
to="#subscribers.email#"
from="svein@nordbotten.com"
type="html"
server="alf.uib.no"
subject="News: On #text# from #source_name#">

#retrieved#

</cfmail>

23. </cfif> 24. </cfif>

25. <cfif (#subscribers.source# EQ "2" or "3") AND (#retrieved2# NEQ "")>

26. <cfset source_name="Washington Post">

27. <cfset text=Trim(#subscribers.topic#)>

> 28. <cfset Position=REFindNoCase(#text#,#retrieved2#,1)>

29. <cfif #Position# GT 0>

30. <cfmail
to="#subscribers.email#"
from="svein@nordbotten.com"
type="html"
server="alf.uib.no"
subject="News: On #text# from #source_name#">

#retrieved2#
31. </cfmail>

32. </cfif>

```


33. </cfif>

34. </cfif>

35. </cfloop>

This template describes the collection of news from 2 sources. Lines 2-11 take care of **retrieving** and storing the news from the 2 sources. Note the **difference** in saving retrieved pages from *agent.cfm*. In some cases, implementation by a temporary file, *temp.html*, may make it easier to get an acceptable execution of the agent. The template can be made more elegant by looping 2 times through most of the lines.

The query named **subscribers** in Line 12 selects the subscriber data and stores them in a query object. The **CFLOOP** query block follows with the remaining lines. This loop is run for each subscriber. However, the first line in the loop is a **CFIF** tag testing if the stop date for the subscription has passed. In that case, the subscriber is skipped. The regular expression in Line 20 **tests** the retrieved news page from CNN for the topic. It continues in a similar way with the news from Washington Post.

Finally, we shall need a template to delete the whole service. Template *delete2.cfm* deletes both sources at the same time. It would be easy to introduce an arrangement which permitted to delete a selected source.

1. <!-- delete2.cfm -->

2. <cfschedule action="delete" task="Retrieve1" >

3. <cfschedule action="delete" task="Retrieve2" >

4. <cfoutput>

5. <H3>Agent task deleted.</H3>

6. </cfoutput>

The regular expressions in Lines 20 and 28 will be discussed in further detail in session 11.

Remarks on scheduling

The scheduling of agents can be **difficult** to implement. There are, however, several alternatives for scheduling processes. The ColdFusion Administrator has been referred to several times, but it requires that the developer has access to this utility.

Another possibility is to use the **meta tag**, e.g. <META HTTP-EQUIV="REFRESH" content="18000">, at the top of the *agent.cfm* template and drop the scheduling template. The template must be kept **running** as long as the service is offered. Using the example, the agent

template will be **executed** every 18000 second, i.e. every 5 hour. This alternative is simple, but requires that the administrator of the agent keeps the template running as long as he offers the service.

A third alternative for Windows is to select **Settings -> Control board -> Schedule tasks** which offers a **wizard** for setting the execution of tasks as running the agents at specified times.

Other Internet Agents

The **CFHTTP** tags open for the development a number of different Internet agents. Information can be **downloaded** from remote hosts as in our agent examples, but information flow can also be **uploaded** from a local host to a number of remote hosts. For example, a news agency has its own corps of field reporters uploading their news from their laptops to the agency's host computer as soon as they finish their stories. Newspapers around the country can subscribe to the news according to specified time and topic profiles from the agency. The agency host downloads automatically the new stories to pre-set folders in the newspaper hosts according to specified time schedules and topic profiles. The individual newspaper can process the accepted stories in its folder for stories from the news agency according to their individual editorial policies.

A famous type of agents is the stock **exchange monitoring agents**. They monitor the stock exchange values on a continuous basis and signals automatically crucial changes to subscribers of their services.

Spider agents crawling around in the Internet is another well-known application. In the spider applications, the agent visits a set of already recorded web sites, parses the pages for relevant content, saves their content, follows identified links to new pages, and repeats the parsing. Some spiders re-visit already recorded pages and make comparisons for updating.

Exercises

- a. The **CFHTTP** tag used in this session is discussed in **Chapter 14** in **RBB**. Study the text; it will give you further ideas about the many possible applications of this tag.
- b. The examples in this session are simple agent applications. Using the powerful **CFHTTP** tags with its many optional attributes, **complex agents** can be implemented. Agents surveying the **exchange** market are popular. Using action "POST" makes it also possible to work as a distributor. The agent is then scheduled to **release and distribute** messages from another process to a list of remote servers as well as clients.
- c. **CFSCHEDULE** is described in **Chapter 22** of **RBB**. Even though this tag is mainly related to the application server management, we have demonstrated that it has clearly also interest for the application programmer.

d. If you have some ideas for developing agents, write them down and post them on the message board.

Session 10: Data exchange - syndication

XML

XML (Extensible Markup Language) is considered a language which in the future may **replace HTML** for coding web pages. It is, as **HTML**, a derived subset of **SGML** (Standard Generalized Markup Language) and permits a programmer to define new tags and their meaning for different applications.

The tags can be defined in a **DTD** (Document Type Definition), which may either be **embedded** in the **XML** file in which it is used, or, if the tags are used in several files, **DTD** may be in a **separate** file downloaded with the **XML** files and read by the browser. The presentation style of the **XML** file content can be separated from the structure of the file in a **style sheet** prepared by means of **XSL** (Extensible Style Language). **XLL** (Extensible Link Language), a language for defining hypertext **links** in **XML** web pages belongs also to the family.

The **XML document object** is a special **data type** in **CFMX**. An **XML** document can easily be converted to this data type by means of the function **XmlParse()**. A great advantage of the **XML document object** is that it can be processed by the **ordinary CFMX** structure functions.

An **XML** document must be strictly **well-formed**. It is well-formed if it is free of syntax errors. If syntax errors exist in an **XML** document, it cannot be processed.

XML is an extensive topic which deserves its own course. In this course, we have space only to refer the interested student to the **short introduction** of **XML** in **RBB**, and to the many **books** issued in the recent years on the topic. In the remaining of this session, we shall discuss a technology for **exchanging** data between different types of computer platforms based on **XML**. This technology, **WDDX**, does not require any extensive knowledge about the **XML** language.

Web Distributed Data Exchange

WDDX (Web Distributed Data Exchange) is an **open** technology for **exchanging** all kinds of data types by means of an **XML** representation. It is intended to be used for exchanging data among applications implemented by different software languages, and run on different hardware platforms. **WDDX** is constructed on the simple assumption that data to be exchanged is '**serialized**' from the local language of the sender to a representation readable by all systems and '**deserialized**' to the foreign language of the receiver. The exchange requires that serialization and deserialization functions exist for each environment participating in the exchange. Required **WDDX** functions have already been prepared for many programming languages and can be downloaded **freely** from the net.

In **CFMX**, the serialization and deserialization are controlled by a special tag, **<CFWDDX ACTION=".." INPUT=".." OUTPUT="..">**. The tag can be used without any knowledge of **XML**. The **ACTION** attribute can take 2 values in which we are interested: **cfml2wddx** is used for serialization, and **wddx2cfml** for decentralization (2 more will be discussed in the last session). Other languages/platforms have corresponding instructions, which permit each partner

to use the instructions of her/his technology to serialize data to be sent and deserialize data received.

Data exchange between art galleries

The **WDDX** technology is demonstrated in the following scenario. Consider 2 cooperating **Art Galleries**, **A** and **B**, which have agreed to inform each other in text and images about their stocks of paintings at the end of each month. This kind of **loose** cooperation is often referred to as content **syndication**.

The following data about each painting should be included in the exchange:

- **Identification**
- **Name** of picture
- Name of artist
- **Time** at which the painting was completed
- Estimated **value**
- **Picture** of the painting

Each gallery has its own database not available on the web. A problem for the data exchange is that the partners have different hardware **platforms**, run different **operating** systems and **database** systems. Fortunately, **WDDX** modules exist for both system environments.

[Figure 1](#) shows the data exchange scenario between the 2 galleries.

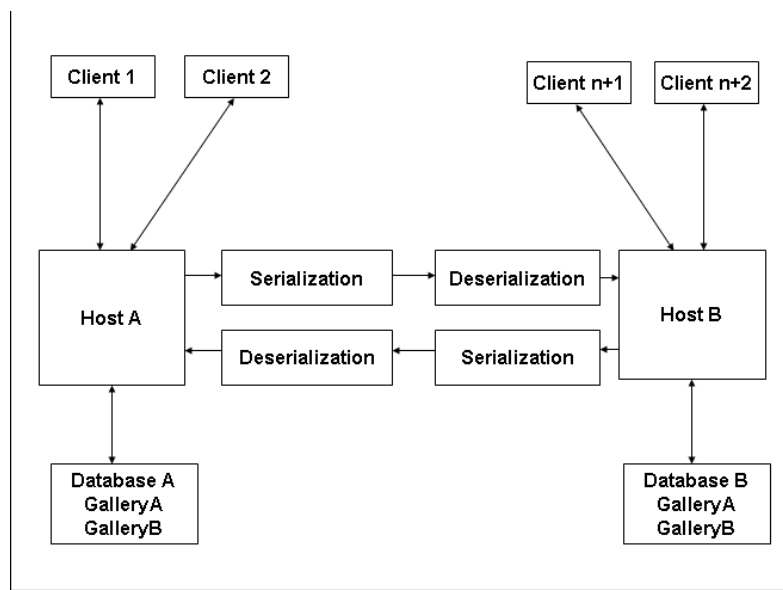


Figure 1: Data exchange between Gallery A and Gallery B

Implementation of the exchange

To demonstrate the **WDDX** data exchange, we consider the following templates:

1. *index.cfm* generating menu for the application
2. *Form_store.cfm* and *store.cfm* for packing and storing new pictures data
3. *Transmit.cfm* for retrieving, and sending stored pictures data
4. *Packet.cfm* for inspecting a received **WDDX** packet
5. *Table.cfm* for unpacking and view received image data.

The menu is quite ordinary and should not need any explanation:

1. `<!-- index.cfm -->`
2. `<div align="center">`
3. `<h2>Menu for WDDX example</h2>`
4. `<table>`
5. `<tr><td>Store picture data in a database</td></tr>`
6. `<tr><td>Transmit data</td></tr>`
7. `<tr><td>See table </td></tr>`
8. `<tr><td>See packet</td></tr>`
9. `</table>`
10. `</div>`

The *form_store.cfm* initiates the form for uploading a new picture from your client computer of to the server of the first gallery assuming you are in charge of this gallery and have acquired the pictured painting:

1. `<!-- form_store.cfm -->`
2. `<center>`
3. `<h2>Storing picture data in a database</h2>`
4. `<p>Select an image stored on your PC. It doesn't matter what you call the painting(image) or the artist's name.</p>`
5. `<cfform action="store.cfm" method="post">`
6. `<table>`
7. `<tr><td>Picture name:</td><td><cfinput name="name" type="text"></td></tr>`
8. `<tr><td>Picture artist:</td><td><cfinput name="artist" type="text"></td></tr>`
9. `<tr><td>Picture created:</td><td><cfinput name="created" type="text"></td></tr>`
10. `<tr><td>Picture value:</td><td><cfinput name="est_value" type="text"></td></tr>`
11. `<tr><td>Picture file:</td><td><input name="picture" type="file" ></td></tr>`
12. `<tr><td></td><td><input type="submit" value="Submit"></td></tr>`
13. `</table>`
14. `</cfform>`

[Figure 2](#) illustrates the use of the form which is trivial.

Storing picture data in a database

Select an image stored on your PC. It doesn't matter what you call the painting(image) or the artist's name.

Picture name:	<input type="text" value="Outcry!"/>
Picture artist:	<input type="text" value="Edvard Munch"/>
Picture created:	<input type="text" value="1893"/>
Picture value:	<input type="text" value="1.000.000"/>
Picture file:	<input type="text" value="C:\Documents and Settings\Bla gjennom..."/>
<input type="button" value="Submit"/>	

Figure 2: Example menu.

The *store.cfm* is more interesting. Line 2, uploads the selected picture from a user's computer to the server of the first gallery and stores it in a temporary binary image file. The next statement reads this file into a variable called *picture*. Then a new structure, *picture_data* is defined in Line 4, and the structure is populated by Lines 5-9. **Structure** is a very convenient data type because it can be processed as a single object or variable. Our variable is *picture_data*.

```
1. <--- store.cfm --->
2. <cffile action="UPLOAD" filefield="form.picture" destination="#session.path#\temp.jpg"
   nameconflict="OVERWRITE">
3. <cffile action="READBINARY" file="#session.path#\temp.jpg" variable="picture">
4. <cfset picture_data=StructNew()>
5. <cfset picture_data.name=#form.name#>
6. <cfset picture_data.artist=#form.artist#>
7. <cfset picture_data.created=#form.created#>
8. <cfset picture_data.est_value=#form.est_value#>
9. <cfset picture_data.picture=#picture#>
10. <cfwddx action="cfml2wddx" input="#picture_data#" output="picture_packet">
11. <cfif not IsWddx(#picture_packet#)>
12. <p>Picture_packet is not a well-formed XML. </p>
13. <cfabort>
14. </cfif>
15. <cfquery name="delete" datasource="#session.datasource#">
16. DELETE FROM wddx
17. </cfquery>
18. <cfquery name="packet_in" datasource="#session.datasource#">
19. INSERT INTO wddx(packet) Values('#picture_packet#')
20. </cfquery>
21. <center>
22. <cfoutput>
23. <p>Picture data structure has been successfully stored in the database.</p>
24. </cfoutput>
25. <p><a href="index.cfm">Return</a> to menu.</p>
26. </center>
```

Line 10 uses the **WDDX** function to convert the object *picture_data* to a single text string, the **WDDX** packet *picture_packet*. This is an **XML** page and is checked in Lines 11-13 for being well-formed. A not well-formed **XML** page cannot be processed, and if so the process is aborted. As usual, **session.path** must be set in the *Application.cfm* pointing to the directory in which you store your **WDDX** example.

Lines 15-17 are included for this example to delete old paintings in the database. In a real application, these lines would not be included. Lines 18-20 store all the painting data represented as a text string in a variable named *packet* in the table **WDDX**. This demonstrates a compact way of storing data in a database. Alternatively, the data could of course have been stored in a table with one variable for each painting attribute, which would have permitted searching in the database by attribute values.

To send a copy of the table to a collaborating gallery, we need to **retrieve** the packet from the database. Since it is already **serialized** before it was saved in the database, we may only want to **test** the packet again for being well-form before it is sent to the receiving gallery. The packet can be sent to the other gallery by means of the **CFHTTP** tag we already know from a previous session. To avoid working with 2 servers, we 'substitute' the transfer .by a remark tag and continue as if we are on the receiving server of the other gallery

The template *transmit.cfm* for the **sending** data from one gallery to another based on **CFMX** looks like this:

```
1. <!-- transmit.cfm -->
2. <cfquery name="packet_out" datasource="#session.datasource#">
3. SELECT packet FROM wddx
4. </cfquery>
5. <cfif Not IsWDDX(#packet_out.packet#)>
6. <p>Packet_out.packet is not a well-formed XML</p>
7. <cfabort>
8. </cfif>
9. <!-- Code for packet transfer to the other server -->
10. <cffile action="write" file="#session.path#\packet.txt" output="#packet_out.packet#">
11. <center>
12. <h2><font color="#0000FF">Dump of transmitted packet</font></h2>
13. <cfoutput>
14. <cfdump var="#packet_out#" >
15. </cfoutput>
16. </center>
```

Lines 2-4 **retrieve** the packet(s) naming the query object *packet_out*. Since a **WDDX** packet is an **XML** file, it is again **tested** for being well-formed. Line 9 is the substitute for the real transmission. In a real application, Line 9 must be replaced by **CFHTTP** and **CFHTTPPARAM** tags to initiate the transfer to the server of the second gallery where Line 10 takes care of storing the received packet in a text file, *packet.txt*, at the other server.

Lines 11-16 apply the **CFDUMP** tag which displays the complex **WDDX** packet object by components. [Figure 3](#) illustrates the result generated by this tag. Note that this dump is **not** the the packet string representation, but an interpretation of the packet content.

Dump of transmitted packet

query	
PACKET	
1	
wddx encoded	
struct	
ARTIST	Edvard Munch
CREATED	1893
EST_VALUE	1.000.000
NAME	Outcry!
PICTURE	image

Figure 3: Packet component.

The packet text string itself can be displayed by the template *packet.cfm*:

1. `<!-- packet.cfm -->`
2. `<center> <h2>Display of WDDX packet</h2>`
3. `<cffile action="read" file="#session.path#\packet.txt" variable="packet2">`
4. `<cfoutput > <textarea rows="15" cols="70" wrap="virtual"> #packet2#</textarea>`
5. `</cfoutput>`
6. `</center>`

The **XML** file for representation of the image used in the example is illustrated in [Figure 4](#). You can explore the tags and representations. You can for example see that the whole image used in the example is represented by a string of 22065 characters.

Display of WDDX packet

```
<wddxPacket version='1.0'><header/><data><struct><var
name='EST_VALUE'><string>1.000.000</string></var><var
name='PICTURE'><binary
length='22065'>/9j/4AAQSkZJRgABAQAAQABAAQ/2wCEAAAsAB4AIQAnACEAHAAAsACc
AJAAnADIALwAsADUAQgBvAEgAQgA9ADUAQgCIAgEAZgBQAG8AoQCNAKkAggCeAIOAmwCYA
LEAxwD/ANgAsQCBAPEAvwCYAJsA3gEuAOAA8QEHAQ8BHQEgARoArADVAtkBTwE2ARUBTAD
/ARgBHQES/9sAgxEALwAyADIAQgA6AEIAggBIAEgAggESALcAmwC3ARIBegESARIBegESA
RIBegESARIBegESARIBegESARIBegESARIBegESARIBegESARIBegESARIBegESARIBegES
SARIBegESARIBegESARIBegESARIBegESARIBegESARIBegESARIBegESARIBegESARIBeg
QH/xAAZAAADAQEBAIAAAAAAAAAAAABAgMEAAx/xAA2EAeAAgEDAwMDAwMEAgIDAQEBAhE
AAyExERFRBGfE4GRIjKhFLHBIQJSOTPhQ/BTYnIO8f/EABgBAAMBAQAAAAAAAAAAAAAAAA
AABAgME/8QAjhEBAQACgIDAQEAAwADAAAAAAAAECERIhMUDE1E1YTJcCQSh8P/aAAwDAQ
CEQMRAD8AOKRhIn7/tg1pvFb5Sc46Ra3KQNVKJa6iRjTfK5yTGSNOtmE62FfbbF1GQb
leb3wfU1xiq9jO6tVKYQXMKPswXmR5xjRAfkU4Y6kbLE/nKnO5f7h375VxyV9qXORLJc
e2D6VPn2zSR1bC2xpxvkO/Tq/pfYgr2Kc4J09K/GaCEeGzgkic6sT7YcbH2ROERKIEzns
```

Figure 4: The WDDX XML file.

We shall return to the **DTD** for the **WDDX** packets in a moment. Notice that the variables represented in the packet are **ordered by the values** of the variables, not by their names. A whole painting structure is represented in a text string with a surprisingly low number of bytes.

The last template, *table.cfm*, let you see the picture data after it has been deserialized at the simulated receiving end. When received, the packet was saved in *packet.txt* as an **XML** file. This file is read and transformed to a variable *picture_data2* by Line 2 and deserialized by Line 3 to a structure object, *deserialized_packet*, with five variables. In a real application, the **CFWDDX** function in Line 3 must be replaced by the **WDDX** function corresponding to the receiving gallery's platform.

We have now the structure object. To display the picture, we must first save the picture component in Line 4 so it can be binary read in Line 13.

In the remaining lines the content of the structure is displayed for the user.

```
1. <!-- table.cfm --->
2. <cffile action="read" file="#session.path#\packet.txt" variable="picture_data2">
3. <cfwddx action="wddx2cfml" input="#picture_data2#" output="deserialized_packet">
4. <cffile action="WRITE" file="#session.path#\picture_received.jpg"
   output="#deserialized_packet.picture#" nameconflict="overwrite">
5. <center>
6. <cfoutput>
7. <h2><font color="Blue">Received picture data</font></h2>
8. <table>
9. <tr><td>Name: </td><td>#deserialized_packet.name#</td></tr>
10. <tr><td>Artist:</td><td> #deserialized_packet.artist#</td></tr>
11. <tr><td>Created:</td> <td>#deserialized_packet.created#</td></tr>
12. <tr> Value: #deserialized_packet.est_value#</td></tr>
13. <p></p>
14. </table>
15. </cfoutput>
16. </center>
```

To be able to exchange the pictures, both galleries must have the templates discussed above supplemented by the **CFHTTP** tag to send the **WDDX** packets to each other. The final result from our example for this exchange process is shown in [Figure 5](#). Remember that you must set the **session.url** also in the *Application.cfm* of the example.

Received picture data



Name:	Outcry!
Artist:	Edvard Munch
Created:	1893
Value:	1.000.000

Figure 5: The final display of the received data.

DTD for WDDX

Referring to the introduction of this session, you may wonder how the **document type definition** for the **WDDX** looks. Here is a [link](#) to the **DTD** file defining the language of **WDDX**. If you activate this link, the **WDDX** document type definition is displayed for you by means of an **XML** document object. Recall that the use of the **WDDX** technology does not require that you know anything about neither this **DTD**, **XML** nor **XML document objects**.

Exercises

- Copy the displayed templates, complete the example with the statements required for sending the picture data to a remote host (for example another student), and test out your application.
- Download the **WDDX** software needed for receiving packets in a **JAVA** environment, and see if you succeed to deserialize the packets sent from the **ColdFusion** host in a **JAVA** environment.

c. For simple mass data exchange, a less general technology based on **XML** can be used. Consider the requirements for a scenario you know, and use **RBB** as a guide to planning your own data exchange design.

Session 11: Regular expressions and CFScript

Regular expressions and string processing

Regular expressions are a way of specifying **text processing** conditions. It is used in a number of programming languages, but it is unfortunately not standardized. It is a very central part of the scripting language **PERL**, and the regular expression feature in **ColdFusion** is almost **compatible** with **PERL 5**. In **ColdFusion**, regular expressions are used in 2 *basic* ways:

1. **Searching** for symbol patterns in a string of symbols.
2. **Replacing** symbol patterns in a string with new symbol patterns.

There are 4 **RE functions** in **CFMX**:

1. **REFind(REGex, String [, Start] [,ReturnSubExpressions])**
2. **REFindNoCase(REGex, String [, Start] [,ReturnSubExpressions])**
3. **REReplace(String, REGex, SubString [,Scope])**
4. **REPlaceNoCase(String, REGex, SubString [,Scope])**

The 2 first functions used for searching are identical with the exception that the first is **case sensitive** and the second is **case insensitive**. These functions can take up to 4 arguments of which only the 2 first are required. The first required argument is the **regular expression** containing the condition for identifying a **symbol pattern** in the text **string** given as the second required argument.

Similarly for the third and the fourth functions used for replacing. These functions have, however, 3 required arguments: A **string** which is to be processed, a **regular expression** identifying substrings, and a **new** substring to replace identified substring(s).

We have already used the second regular expression function, **REFindNoCase(REGex, String [, Start] [,ReturnSubExpressions])**, for parsing text in the *agent2.cfm* template of the **Agent2** example in Session 9 in order to identify wanted keywords in a text of news.

The syntax for forming regular expressions is based on sets of **operators**, **character classes** and/or **Portable Operating System Interface (POSIX) classes**. For a more detailed description of the syntax for regular expressions, consult **RBB**.

Re-visiting the search engine

In Session 6, we studied how to build a search engine by means of the **Verity** module included in **ColdFusion**. One of the **VERITY** functions was indexing. [Figure 1](#) explains the content of this function. Given the documents of a registered collection, the indexing function processes each document by **parsing**, i.e., identifying and marking each word of the document, for building a frequency list containing each different word appearing in the document. When parsing of all documents is completed, a **word frequency list** for the whole collection has been generated. For each recorded word, links have been established to all documents in which the word occurs.

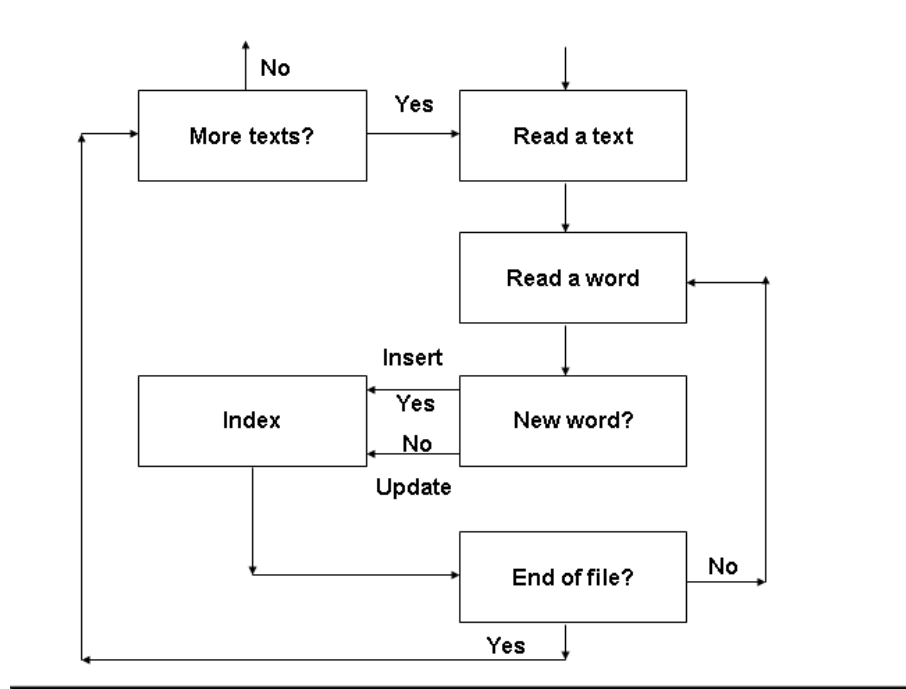


Figure 1: Logical diagram for the indexing process.

The result will be an **inverted index** of words each with frequency of occurrence in the document collection.

Documents containing requested words can then easily be localized by means of the index. Based on each word's total frequency in the collection and its local frequency in a document, different kinds of **scores of relevance** for the individual document can be computed.

In the following example, we demonstrate how to construct a template, which **reads** a text file, **parses** the text by means of a regular expression and **builds** an inverted word index for the document. To make the demonstration complete, the application has an introductory form by means of which you can **upload** your own text file. The second template, *parser.cfm*, **uploads**, **parses** and **prints results** from the specified file.

The introductory *index.cfm* file is simple:

1. <!-- index.cfm -->
2. <h3>Text file indexing</h3>
3. <cform action="parser.cfm" method="POST">

4. Name of file:<input type="file" name="file_name">

5. <input type="submit" value="Submit">

6. </cform>

The purpose of this template is to permit the user to **specify** a text file at the client computer for uploading to the server for processing.

The next step is to **search** sequentially through the text string in the file to **identify** each separate word, **record** the identified word in a list if it not exists already or **increment** a frequency counter if it exists, and **repeat** for next word. For each existing word, its frequency counter is incremented by 1. When the document is exhausted, the words are sorted in descending order with the **most frequent** on top of the list.

In the following template *parser.cfm*, Lines 2-3, the file you specify in *index.cfm* is uploaded, read into variable **file_up** as a string value, and its characters counted. The core of the process is surrounded by 2 pairs of **CFLOOP** tags, one nested in the other and both function as 'while' loops. The first, starting at Line 8 and closing at Line 26, is traversed as many times as the process identifies words. The second enclosing Lines 13-20 is traversed for each character in the words.

The core of the parsing process starts by finding each word. This is done by Line 9 in which the position of the **end delimiter** of the current word is identified. Since the beginning is the character following the end delimiter of the previous word, the word can be extracted by the tag of Line 10.

The position of the delimiter following the end of the current word is found by means of the function **REFindNoCase(..)** in which the criterion for finding the next word delimiter is the regular expression appearing as the first argument. In this template, two **POSIX classes**, **[[:punct:]]** and **[[:space:]]**, are used. Note that in the function, the square bracket must be **repeated** to be correctly interpreted! The first class matches most punctuation characters not appearing as part of English words. The second class matches spaces. As indicated above, the same results could also be obtained by a regular expression based on **character classes**.

With the position at the end of a word and the start position of the word, the 3 **arguments** needed for the string function **MID()** in Line 10 extract the word are available. Lines 11-20 determine if the current word already is in the frequency list. If the word has already been recorded, the frequency number is incremented in Line 16. If not, Line 25 inserts the word, and the next word is extracted and tested against the list of words.

1. <!-- parser.cfm -->

2. < CFFILE action="upload" filefield="file_up" destination="#path#\file_up.txt" nameconflict="overwrite">

3. < CFFILE action="read" file="#path#\file_up.txt" variable="file_up">

4. <CFSET document_size="#len(file_up)#">
5. <CFSET start="1">
6. <CFSET frequency_structure=StructNew()>
7. <CFSET characters_processed="0">
8. <CFLOOP condition="#characters_processed# LT #document_size#">
9. <CFSET position=REFindNoCase("[[:punct:]][:space:]",#file_up#, #start#>
10. <CFSET word=Mid(#file_up#,#start#,#position#-#start#)>
11. <CFSET hit="0">
12. <CFSET counter="0">
13. <CFLOOP condition="#counter# LT #StructCount(frequency_structure)# AND #hit# EQ 0">
14. <CFIF StructKeyExists(frequency_structure,#word#)>
15. < CFSET frequency=StructFind(frequency_structure,#word#)>
16. < CFSET StructInsert(frequency_structure,#word#,#frequency#+1,"true")>
17. < CFSET hit="1">
18. </CFIF>
19. <CFSET counter=#counter#+1>
20. </CFLOOP>
21. <CFIF #hit# EQ 0>
22. <CFSET StructInsert(frequency_structure,#word#,1)>
23. </CFIF>
24. <CFSET start=#position#+1>
25. <CFSET characters_processed=#characters_processed#+#Len(word)+1>
26. </CFLOOP>
27. <div align="center"><h2> Word frequency list </h2></div>
28. <table align="center" border="1">


```

29. <tr>
30. <th>Word:</th><th>Frequency:</th>
31. <CFSET mylist="#ArrayToList(StructSort(frequency_structure,"numeric","desc"))#>
32. <CFLOOP index="word" list="#mylist#">
33. <CFOUTPUT>
34. <tr>
35. <td> #word# </td><td> #StructFind(frequency_struct,word)# </td>
36. </tr>
37. </CFOUTPUT>
38. </CFLOOP>
39. </table>

```

The last part of this template, Line 27-39 is an ordinary **tabulation** of the list. Line 31 orders the word frequency list in frequency descending order, while Line 35 prepares the word and its frequency for display in a table row.

[Figure 2](#) shows a short text,

Text: Introducing Regular Expressions ColdFusion Mx includes support for regular expressions. If you have worked at all with Perl, you probably know all about Regular Expressions because they are such a central part of Perl's string handling and manipulation capabilities, and generally walk and in hand with the Perl language itself. As a rule, Regular expressions aren't nearly as important to ColdFusion coders as they are to perl coders, but that doesn't mean that they aren't incredibly useful. This chapter will introduce you to Regular Expressions and explain how they can be used in Cold fusion applications.

Document size: 616 characters.

Figure 2: A test text.

and [Figure 3](#) demonstrates the indexing result.

Word frequency list

Word:	Frequency:
Expressions	5
Regular	5
Perl	4
they	4
and	4
As	3
you	3
to	3
t	3
with	2
all	2
aren	2
in	2
ColdFusion	2
a	2
are	2
that	2
coders	2
rule	1
This	1

Figure 3: The most frequent words in the extracted list.

Implementation

The application has been implemented as an example and is available by the link at the end of this session. Observe that your file used in this example should be a *.txt* file and that the **absolute address** to the file on your own computer must be specified. You can try files with other extensions, but the results may be affected by the formatting code of the particular file type and create problems. If you for example parse a *.htm* file containing a number of **<P>** tags, the P's are surrounded by the symbols **<** and **>** which are considered word delimiters. Consequently, the "word" **P** appears with high frequency. The best way to try out the parser is to prepare/use a document in **NotePad** (or another **ASCCI** text processor.)

In a real application, the frequent words such as 'and', 'or', 'but', 'the' and words such as pronouns 'I', 'you', 'she', etc., are specified in a list called a **stop-word** list used to exclude these words from the word frequency list since they have little significance when the frequency list is used for searching for keywords. In a real application, the **formatting** specifications should also be eliminated from the text before indexing.

CFScript language

In the introductory session of this course, it was pointed out that **ColdFusion** is based on the **tag-oriented** language **CFML**. However, **CFMX** also includes a **scripting** language which can be activated by the tag **CFSCRIPT**. The **CFScript** language has syntax similar to ordinary programming languages such as **C**. It has syntactically also many similarities with **JavaScript** (we shall return to **JavaScript** in the next session). While **JavaScript** is a tool for extending **HTML** on the **client-side**, the scripting tool **CFScript** is an extension of **CFMX** on the **server-side**.

The syntax of **CFScript** is **simple**, and is discussed by **RBB** in Chapter 19 of his book. You should take note of the following basic rules:

- **CFScripts** can be **included** in **CFMX** templates, but must always be **enclosed** by the tag pair **CFSCRIPT** and **/CFSCRIPT**.
- **CF tags cannot** be used within a **CFScript**.
- **CF functions can** be used in **CFScripts**.
- **Variables** defined in a **CF** template are **available** in a **CFScript** and vice versa.

CFScripts usually give more **compact** code than **CFML**, and many programmers trained in conventional programming feel more comfortable with **CFScript** than with the tag based statements of **CFML**.

Comparing CFML and CFScript

To demonstrate the **CFScript**, we are returning to the *parser.cfm* template of the **Regular Expression** example discussed in the first part of this session. The code for parsing is now developed by means of **CFScript** instead of the **CFML** tags. The **CFFILE** tags in the beginning of the parser template of the previous session, which are needed for uploading and handling the file to be parsed, and the final **TABLE** tags for displaying the results, are **kept** unchanged in the example of this session's *parser_script.cfm* while the tags for processing and preparing the word frequency list are **substituted** with **CFScript** code.

The *parser_script.cfm* template starts with the **CFFILE** tags needed to upload to location *#path#file.txt* and to read the text file. Then follows the **CFSCRIPT** block, and the template ends with the **CFOUTPUT** tag block. The value *#path#* must be set compatible with the web page organization of your server.

Note that like many other language syntaxes, the syntax of **CFScript** requires that each statement is terminated with a semicolon. As mentioned, **CFScript** permits the use of all **CFMX** functions and variables.

Almost a **one-to-one** correspondence between the tags of the previous example and the scripting statements can be obtained. Because of the strong similarity between the two templates, *parser_script.cfm* can easily be interpreted and understood without any further explanation. Note, however, the use of the special **CFScript** function *WriteOutput(string)* appearing in Lines

9 and 10. It writes text to the output stream. The visible **gain** of scripting compared with the tags in this example is **less** text.

```
1. <!-- parser_script.cfm -->
2. <CFFILE action="upload" filefield="file_up" nameconflict="OVERWRITE"
   destination="#path#\file2.txt">
3. <CFFILE action="READ" file="#path#\file2.txt" variable="file_up">
4. <CFSET document_size="#len(file_up)#">
5. <CFSCRIPT>
6. start=1;
7. frequency_structure=StructNew();
8. characters_processed=0;
9. WriteOutput("Text: #file_up#<br>");
10. WriteOutput("Document size: #document_size# characters.<br>");
11. while (#characters_processed lt #document_size#){
12. position=REFindNoCase("[[:punct:][:space:]]",#file_up#,#start#);
13. word=Mid(#file_up#,#start#,#position#-#start#);
14. hit=0;
15. counter=0;
16. while(#counter# LT #StructCount(frequency_structure)# AND #hit# EQ 0){
17. if (StructKeyExists(frequency_structure,#word#)){
18. frequency=StructFind(frequency_structure,#word#);
19. StructInsert(frequency_structure,#word#,#frequency#+1,"true");
20. hit=1; }
21. counter=#counter#+1; }
22. if (#hit# eq 0)
23. StructInsert(frequency_structure,#word#,1);
24. start=#position#+1;
25. characters_processed=#characters_processed#+#Len(word)+1; }
26. </CFSCRIPT>
27. <CFOUTPUT>
28. tid2: #TimeFormat(now(),'hh:mm:ss:ll')#
29. </CFOUTPUT>
30. <div align="center"><h2><font color="Blue"> Word frequency list </font></h2></div>
31. <table align="center" border="1">
32. <tr>
33. <th>Word:</th><th>Frequency:</th>
34. </tr>
35. <CFSET mylist=#ArrayToList(StructSort(frequency_structure,"numeric","desc"))#>
36. <CFLOOP index="word" list="#mylist#">
37. <CFOUTPUT>
38. <tr><td> #word# </td><td> #StructFind(frequency_structure,word)#</td></tr>
39. </CFOUTPUT>
40. </CFLOOP>
41. </table>
```

For the example of this session, the code was implemented with the same *index.cfm* template as in the previous example containing the form required for identifying the text file to be uploaded. The example can be tested by using the link at the end of this session. If used at the same text file as you used for the example of the previous session, the results should be identical.

The advantage of **CFScript** will become significant in more complex processing tasks. It seems to be a trend that scripting is used for developing **User Defined Functions** and **CF Components** designed for intensive **re-use**. These topics will be discussed in the next sessions.

Conclusion

One of the **objectives** of ColdFusion is to be a **RAD (Rapid Application Development)** tool. The advantage of **CFScript** is more compact, elegant and efficient code. The price to be **paid** for using **CFScript** may, however, may be less rapid development. **Optimum** development strategy will depend on the nature of the application **task** and the developer's **preferences, knowledge and experience** with the **CFML** tags and scripting.

Exercises

- a. In the example, we have not paid attention to how the frequency list should be **stored** for efficient use. What about links to the indexed text documents? How should a **stop** word list be taken into account? List the storage alternatives you can think of, and prepare a **pro & contra** discussion for the alternatives imagining you need to convince a client for the solution you think is most suitable.
- b. Implement your storage solution, and consider which **search strategy** will be optimal if you are searching for documents containing one, two or more words. It is usual to indicate document **relevance** with a **score indicator**. How would you assign scores to the different documents? Can you extend your design to work with **logical** expressions?
- c. Do not forget to read **Chapter 18** by **RBB**. The potentials of **RE** are much wider than what has been discussed in this session, and **knowledge** about these potential are useful for designing a number of different systems.
- d. Select one of the processing templates developed in your project and **rewrite** it by means of **CFScript**.
- e. Insert **time-stamps** combined with **CFOUTPUT** at the beginning and end in both the tag and the script versions. Run both of them, compare the timing outputs and see if you can detect any **speed differences**. Discuss what you in fact are **measuring**?
- f. I recommend studying **Chapter 19** on Scripting in **RBB**. My guess is that CFScripting will become more **important** in the future in connection with **User defined Functions, Components** and **Web Services** which are 3 topics to be discussed in the following sessions.

Session 12: Re-using code

The topic for this session is **re-using code** the aim of which is to make development less work extensive and applications to perform more efficiently. This aim is obtained by spending more resources and time on new tasks considered to be repetitive, and to organize the results in such a way that they can easily be used when needed for development of new applications. There are others important issues associated with this objective, which we cannot discuss in this course, among which are scaling and load-balancing.

Re-using code

ColdFusion approaches

When the programmed computers emerged in the 1940's, it was soon recognized that a large proportion of development time was in fact used for **re-inventing** and **re-coding** programs already prepared, but unfortunately not always taken well care of and made available or distributed. Program and routine libraries therefore became established, systematized, published and distributed. About 20 years later, the **object-oriented programming** took off, and **class libraries** were developed and assembled with extended possibilities for re-use.

ColdFusion MX includes a battery of approaches for facilitating systematic re-use of codes including:

- CFINCLUDE Tags
- Customs Tags
- User-Defined Functions
- CFX Tags
- Components

You have most likely already used the **CFINCLUDE** tag in your work and probably saved yourself for some repetitive work. By means of this tag you can include a template with a sequence of tags frequently used in other templates. One **drawback** is that you have to keep track of the address at which the tag to be included is located.

In this session, we discuss **Custom Tags** and **User_Defined Functions**. You are referred to **RBB** for information about **CFX** tags. Visit also Macromedia web site for additional information.

Components and **Web services** will be discussed in later sessions.

Custom Tags

ColdFusion includes almost 100 tags. **ColdFusion** also includes a feature, **Custom Tags**, **CF_** tags, which permits users to define their own tags for operations or procedures frequently used. **CF_** tags are **easy to use**, **save** the developer for re-writing the code for the specific operation,

permit definition of attributes, can be **stored** in standard directories, and **decrease** the possibility for creating bugs in the tedious re-writing of code.

This session's application examples include a tailor-made logging of application use. Evaluating web applications requires observations and data. One type of data is log data reflecting the users' **movements** from one template to another during a visit to the web site and possibly for a sequence of visits. Log data can **identify templates frequently visited** by users, users' **paths** through the application, and the **time** spent by users at each template or sequence of templates. The purpose is usually to provide knowledge to developers for **improving** the applications.

Let us consider the following scenario: Imagine that **milestones** (important points in an application) in can be identified in an application. At each milestone, a log recording can be embedded. Recorded **passing** of each milestone will be data of the type required.

What should be **recorded** in connection with a milestone passing? There are at least 4 useful facts for any log record:

- **ID** for the user (*ID*)
- **Timestamp** for passing the milestone (*TimeStamp*)
- Name of **current milestone** passed(*CurrentMilestone*)
- Name of **previous milestone**(*PreviousMilestone*)

A **CF_** tag, which takes care of the recording of these 4 facts, can be defined by the following template, *logmilestone.cfm*:

```
1. <!-- LogMilestone.cfm --->
2. <cfset timestamp=Now()>
3. <cfset id="#session.pin#">
4. <cfset CurrentMileStone=#cgi.SCRIPT_NAME#>
5. <cfif Not IsDefined('session.PreviousMilestone')>
6. <cfset session.PreviousMilestone="">
7. </cfif>
8. <cfset milestonerecord="#id#, #timestamp#, #CurrentMilestone#, #session.PreviousMilestone#">
9. <cfset session.PreviousMileStone=#CurrentMilestone#>
10. <CFIF NOT FileExists('logrecords.htm')>
11. <CFFILE ACTION="write" FILE="#session.path#\logrecords.htm" OUTPUT="<h3><font
    color=""Blue"">Milestone records</font></h3>">
12. </CFIF>
13. <CFFILE ACTION="append" FILE="#session.path#\logrecords.htm" OUTPUT="#milestonerecord# <p>"
    addnewline="yes">
```

Note that the above listing shows that this template is **not different** from a regular template. Lines 2-4 take care of generating 3 of the 4 variables wanted for the log recording. The value, *#session.pin#*, is assumed set when the users entered the application. The 4th variable, **PreviousMilestone**, does **not exist** if the recording is the first milestone of the current session. Then Lines 5-7 define the *session.PreviousMilestone* to be blank ("").

Line 8 composes a list called *milestonerecord*. Lines 10-12 test the existence of the *file logrecords.htm*. If the file does **not exist**, i.e. this is the first log recording, done, the template establishes the file. Finally, Line 13 appends the list *milestonerecord* as a record to the file *logrecords.htm*

The **Custom Tag** template can be saved in different places. A recommended rule is to save it in a pre-installed directory called **CustomTags** in a special directory, **Customtags**, within the general **CFMX** directory. However, frequently an **ISP** host does **not permit** its customers access to this directory. The easiest way is then to copy the tag template to the directory in which it is used. The tag template can be **called** from any template by the tag **CF_LogMilestone**.

The **CustomTags** have a number of properties, possible uses as well as restrictions. In this session, only the most elementary aspects are discussed. You are recommended to study Chapter 27 of **RBB**.

We use the **guess** example from Session 2 to illustrate application of the **CF_LogMilestone** tag. We assume that the display of the problem and the response from the system are the 2 interesting milestones in this application. In the present example, *index.cfm* template of session 2 is renamed as *index_.cfm* after the **CF_LogMilestone** tag is included at the top of the template:

```
1. <!-- index_.cfm --->
2. <cfset session.pin=#xxx#>
3. <CF_LogMilestone>
4. <CFSET temp=randomize(second(Now()))>
5. <CFSET session.target=#RandRange(50,100)#>
6. <h2><font color="Red">Guess!</font></h2>
7. <form action="response_.cfm">
8. <cfoutput>
9. <p>My name is <input type="text" name="name"></p>
10. <p>I guess the sum of all integers from <b>1</b> to <b>#session.target#</b> is <input type="text"
    name="guess"></p>
11. </cfoutput>
12. <p><input type="submit" value="Submit"></p>
13. </form>
```

In Line 2 we have to set some value for the **session.pin**. Similarly, the *response.cfm* template of session 2 is renamed as *response_.cfm* after the **CF_LogMilestone** tag is included:

```
1. <!-- response_.cfm --->
2. <CF_LogMilestone>
3. <CFSET sum="0">
4. <CFLOOP INDEX="count" FROM="1" TO="#session.target#" >
5. <CFSET sum=#sum#+#count#>
6. </cfloop>
7. <CFIF #sum# EQ #guess#>
8. <cfoutput>
9. <h3><font color="Blue">#name#, your guess was correct!</font></h3>
10. </cfoutput>
11. <CFELSE>
```


12. <cfoutput>
13. <h3>Sorry, #name#, the sum is #sum#.</h3>
14. </cfoutput>
15. </cfif>

To make the example complete, a menu is need, and *index.cfm* is included:

1. <!-- index.cfm --->
2. <center>
3. <h2>Menu</h2>
4. <p>Do you want to:</p>
5. <table>
6.
7. Run the application
8. See the log
9.
10. </table>
11. </center>

The options of Lines 7-8 make it possible to run the application **several** times and then and inspect the listing of the log file in which the milestone passing are recorded. The list displays **all activities** of the example applications and you will see recordings from **other** users before you entered the example. The list can easily be copied and processed for analytical purposes by for example **MS Excel**. With real id numbers, you will then be able to study how many times the individual user tests the example application before he/she gets bored, how long time is used in average to figure out an answer, etc.

User-Defined Functions

User-Defined Functions, **UDF**, is another feature which makes re-using code easier. Compared with **Custom Tags**, **UDF** can be an advantage if you have some **data manipulation** (calculations, string parsing, etc.) to be done in different applications. A **UDF** can either be coded by general **CFML** tags, or as a **CFScript** block.

In the **Guess** example, *response_.cfm* includes the calculation of all integers from 1 up to a certain number. If we develop applications in which this calculation frequently is required, we may decide to make it a **UDF**. We decide to use the **CFScript** keeping in mind that this approach cannot use **CFML** tags. The **CFScript** is also compatible with earlier versions of **CF**.

We create a **UDF** called *sum_integers(number)* with one argument, **number**. The argument is the **upper limit** of the integer series.

1. <!-- UDF_lib.cfm --->
2. <!-- You can store as many UDF-functions as you like within the CFSCRIPT tags --->

3. <CFSCRIPT>

```

4. function sum_integers(number)
5. {
6.   sum=0;
7.   for(index=1; index LTE number; index=index+1)
8.   {
9.     sum=sum+index;
10.  }
11. return sum;
12. }
13. </CFSCRIPT>

```

Note that the **CFSCRIPT** block comprises all statements in the template, in this case all statements in the single function included.

UDF libraries

A function can be **saved** in the same template it is used. Obviously, when you need to use the function more than once, a better solution is required. An effective approach is to collect all your functions in a **library** template, which you name *UDF_lib.cfm*. If you add an include tag, `<CFINCLUDE template="UDF_lib.cfm">`, in your *Application.cfm*, all functions in your library will be available for all templates in your application.

Above, we have stored the *function sum_integers(number)* including the **CFSCRIPT** tags in the file called *UDF_lib.cfm*, and illustrate by the template *index.cfm* how we can make use of the library and its functions:

```

1. <!--index.cfm -->
2. < cfinclude template="udf_lib.cfm">
3. < center>
4. < Cfif IsDefined('number')>
5. < cfoutput>
6. < font color="blue"><h2>The sum is <b>#sum_integers(number)#</b></h2></font>
  < /cfoutput>
7. <cfelse>
8. < h2><font color="blue">Sum function</font></h2>
9. < cform action="index.cfm" method="post" enablecab="yes">
10. < p>I want the sum of all integers from 1 to
11. < cfinput name="number" required="yes" type="text">
12. < input type="submit" value="Submit">
  < /cform>
13. < /Cfif>
14. </center>

```

Note that we have included the **CFINCLUDE** tag in this file, since we have not shown the *Application.cfm* for this application.

CFX Tags

CFX tags are tags prepared in a foreign language and compiled to a *.dll* or *.class* file. This opens for preparing very effective tailor made tags. In this course, however, we shall only take note of this possibility without exploring it further.

Exercises

- a. You are encouraged to **copy and try** out the examples for yourself. An challenging task will be to work out how a component can be called by means of the **CFOBJECT** tag.
- b. Review your project templates and **identify** code sequences which you have used repetitively. **Transform** these sequences to **Custom Tags**, and **evaluate** what you gained in your project, and if you can **benefit** from your Custom Tags in future applications.
- c. The experts recommend that you code your **UDF** based on **tags**. Re-write the **UDF** script-based function *sum_integers(number)* as a **tag-based UDF** and test it.
- d. **RBB** discusses **Function Libraries** in Chapter 20. Read the chapter carefully. He refers to an **open source** repository named the **Common Function Library Project**. Import the [library](#) and see if you find any interesting and useful functions for your project in the library.
- e. There are many ways of constructing and using components. **RBB** discusses some of them in Chapter 22. Components build on **UDF** is discussed in detail in **RBB** Chapter 20. You are recommended to **read** this chapter, and try out some of the suggestions.
- f. Consider how you would **reorganize** your own project if you should have implemented it by means of components.
- e. You are welcome to **study** the **wsdl** of as well as and **invoke** the web service:
http://nordbotten.ifl.uib.no/webservices/access_webservice.cfc?wsdl called from an application on your own computer.

Session 13: Distributed processing

Distributed processing

Processing tasks can be **distributed** among several computers connected by a network. In this session, we shall consider 2 technologies for distributing a task between **a host and connected clients**.

Client-side processing

In web applications, many possibilities exist to let the clients do part of the processing. Transmitting **Java Applets** were among the first **client-side** technologies available. Later, alternative specialized tools for client-side processing have been developed. **JavaScript** is a tool for client-side processing which today can be used in connection with most popular browser software.

Macromedia's Flash has become very popular and powerful as another tool for client-side processing. It can also be combined with **ColdFusion** by means of **Flash Remoting**, an interface between the **Flash** processing (See [Colin Moock](#)) on the **client-side** and **ColdFusion** on the **server-side** (See [Tom Muck](#)).

JavaScript

In this session, we shall review **JavaScript** (See [David Flanagan](#)) combined with **CFMX** for client-side processing. **JavaScript** scripts can be embedded in **CFML** templates and sent as part of the **HTML** response to a requesting client. Most major browsers are able to interpret and execute **JavaScript** code.

JavaScript is a scripting language similar to **CFScript** already discussed. In contrast to **CFScript**, which is interpreted on the **server side**, **JavaScript** is executed on the **client-side**. **JavaScript** code can be included in **CFMX** templates surrounded by the tags `<SCRIPT LANGUAGE="JavaScript" TYPE="javascript">` and `</SCRIPT>`.

The application domain of **JavaScript** is wide. Most popular is probably use of the technology for **validity checking** of the answers to form requests before the form is submitted. Client-side checking saves the transmitting, server-side checking a possible a new request to the client because of invalid answers.

WE shall consider another application. The **session tests** of this course have all been developed as **server-side** applications. However, they could as well have been developed as **distributed** applications where the checking of the answer is done at the **client-side**, and the server is only engaged when you pass the test and want the points recorded at your progress record, or when you request a new test.

The example application we shall illustrate how the test form can be designed to be checked by the client computer by embedding a **JavaScript** in the **.cfm** file sent to the client.

As an example shortcut, we call the **form template** for *index.cfm* in the example. This template would look like this:

```
1. <!-- form.cfm --->
2. < cfoutput>
3. < SCRIPT language="JavaScript" type="text/javascript">
4. function evaluate(){
5. if(document.select.reply[1].checked) document.write("<center><font color=\"Blue\"><h3>Your answer
   was correct.<br><br>Do you want to continue?</font><br><br><a href=\"form.cfm\">Yes</a><a
   href=\"index.cfm\">No</h3></font></center>");
6. if(document.select.reply[0].checked) document.write("<center><h3><font color=\"Red\">Your answer
   was wrong.</font></h3>Correct answer is: JavaScript is a scripting language.<br><br><font
   color=\"blue\"><h3>Do you want to continue?</font><br><br><a href=\"form.cfm\">Yes</a><a
   href=\"index.cfm\">No</h3></font></center>");
7. if(document.select.reply[2].checked) document.write("<center><h3><font color=\"Red\">Your answer
   was wrong.</font></h3>Correct answer is: JavaScript is a scripting language.<br><br><font
   color=\"blue\"><h3>Do you want to continue?</font><br><br><a href=\"form.cfm\">Yes</a><a
   href=\"index.cfm\">No</h3></font></center>");
8. if(document.select.reply[3].checked) document.write("<center><h3><font color=\"Red\">Your answer
   was wrong.</font></h3>Correct answer is: JavaScript is a scripting language.<br><br><font
   color=\"blue\"><h3>Do you want to continue?</font><br><br><a href=\"form.cfm\">Yes</a><a
   href=\"index.cfm\">No</h3></font></center>"); }
9. < /SCRIPT>
10. < /cfoutput>
11. ><FONT color="blue"><H2>Question 7 from session 13 </H2></FONT>
12. <l><B>Question:</B></l><br> < P>JavaScript is a:</P>
13. <l><B>Answers:</B></l><br>
14. <form name="select" onsubmit="return false">
15. < P><INPUT name="reply" type="radio" onClick="evaluate()" value="1"> JAVA program.</P>
16. < P><INPUT name="reply" type="radio" onClick="evaluate()" value="2"> Scripting language.</P>
17. < P><INPUT name="reply" type="radio" onClick="evaluate()" value="3"> JAVA utility.</P>
18. < P><INPUT name="reply" type="radio" onClick="evaluate()" value="4"> JAVA interpreter.</P>
19. < /form>
```

The form with the question and answers is contained in the Lines 11. - 19. It **differs** in 2 respects from the server-side template form used in the tests. First, this form has no **INPUT** tag with **TYPE="submit"** because it should **not** be submitted to the server. Instead each input tag has an **ONCLICK** event attribute, which transfers the control to the **JavaScript** function *evaluate()* in Lines 4-8 with the **VALUE** attribute as a **parameter**. This **JavaScript** function is executed **locally** at the client's computer by means of an interpreter included in the browser.

The **JavaScript** function *evaluate()* is surrounded by the **SCRIPT** tags, which again is nested in the **CFOUTPUT** tags in order to display **CFML** based strings. The *evaluate()* function includes 4 **if clauses**, each of which tests if one of the 4 alternative answers was checked. The **if**-conditions make use of a **JavaScript document** object method, **document.select.reply[].checked**, which **select** the option checked (it is impossible to check more than one answer). Note that while the answers are numbered from 1 to 4, the **JavaScript** runs through a list of the options in which they are numbered from 0 to 3. Another **document** method, **document.write**, is used in the script to write if the checked alternative was correct or not.

The above distribution **saves** some interaction between client and host and let the client share part of the processing load with the host. With modern high memory capacity on the client side, an even more efficient and realistic distribution would be to transfer **all** the questions with answers prepared by the host for a client requesting a test, and let a **JavaScript** manage the **complete test** and report the final results only back to the server.

Flash and ActionScript

Flash has become known as a technology for developing **animations** on the net. However, the last versions of **Flash MX**, the scripting language **ActionScript**, the **Flash Remoting** and **ColdFusion MX** represent together a very general environment for developing advanced applications on the net.

Developing such applications **requires Flash MX**, installation of **Flash Remoting** and **ColdFusion MX** on the developer's computer. Running the developed and tested application, does not require more than any other **ColdFusion** applications. To accept and run a **Flash** application, the **client's** computer must have the **Flash reader** (free) installed.

The scripting language **ActionScript**, as **JavaScript** and **CFScript**, is developed on basis of the **ECMA-262** standard. The 3 languages deviate all from the standard, and each other, but there are obvious **similarities**.

The final **FLASH** result is called a **movie** (saved as a **.swf** file) which is generated from a **FLASH document** (saved as a **.fla** file). Usually a document is **created** by means of the **FLASH** graphic authoring tool ([Figure 1](#)) and **ActionScript** as a sequence of graphical **frames** connected to **scripts**. As an elementary demonstration of **Flash** application in combination with **ColdFusion**, a small example is included.

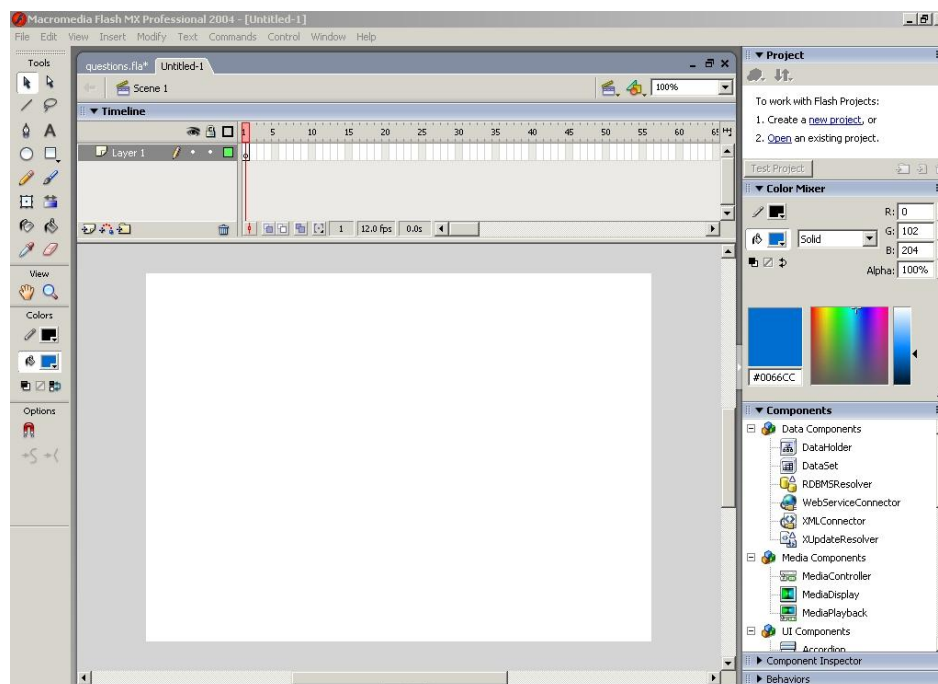


Figure 1: FLASH authoring tool.

The scenario is again the session tests of this course. An *index.cfm* page is used to call the **Flash** application which is loaded down to your computer. During the loading process, which starts by a *.html* page, the existence of a **Flash reader** is checked, and, if negative, also loaded before the movie:

1. `<!-- index.cfm -->`
2. `<center>`
3. `<h3> FLASH demonstration </h3>`
4. `<p>Do you want to take the session test?</p>`
5. `<p>YesNo`

This **CFML** template returns the *questions.html* and an attached **Flash** movie file *questions.swf*, containing the **Flash** application, to the requesting client.

The **Flash movie** contains a sequence of frames which are played subject to the control of the scripts also included in the *.swf* file. In this example, there are 5 frames:

- init,
- continue,
- correct,
- incorrect,
- end.

Frame **init** ([Figure 2](#)) displays a question and the multi-choice answers. When a button is clicked,

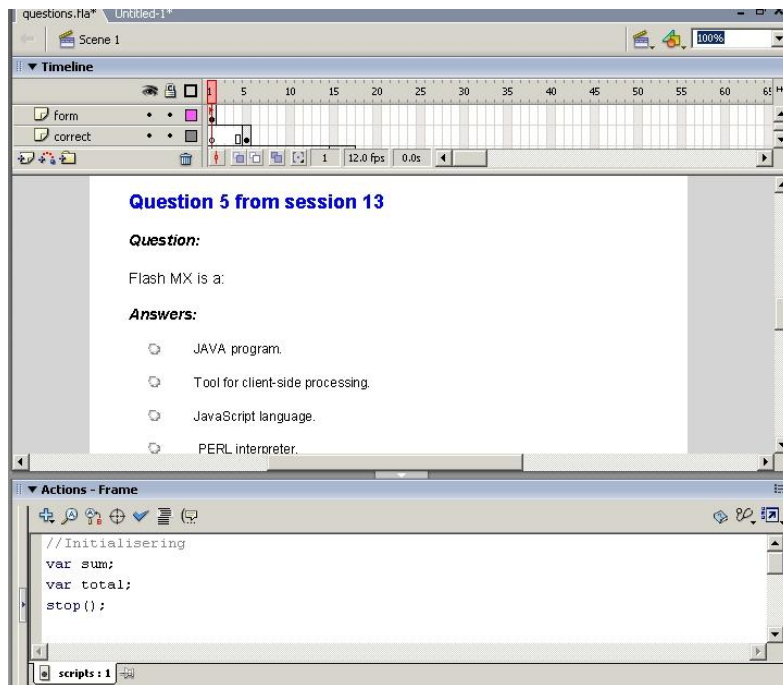


Figure 2: FLASH frame *init*

an attached **script** directs the play through **continue** ([Figure 3](#))

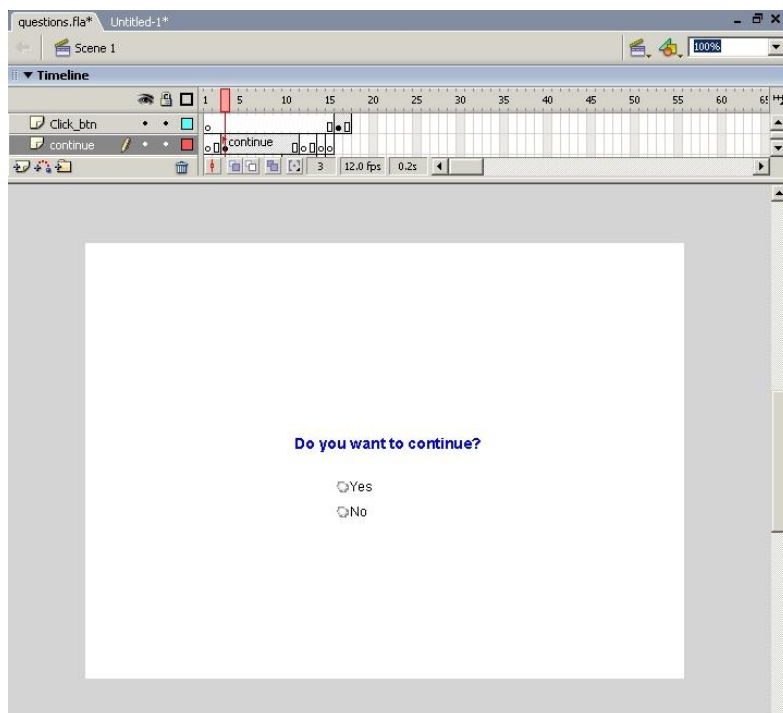


Figure 3: FLASH frame *continue*

to either frame **correct** ([Figure 4](#))

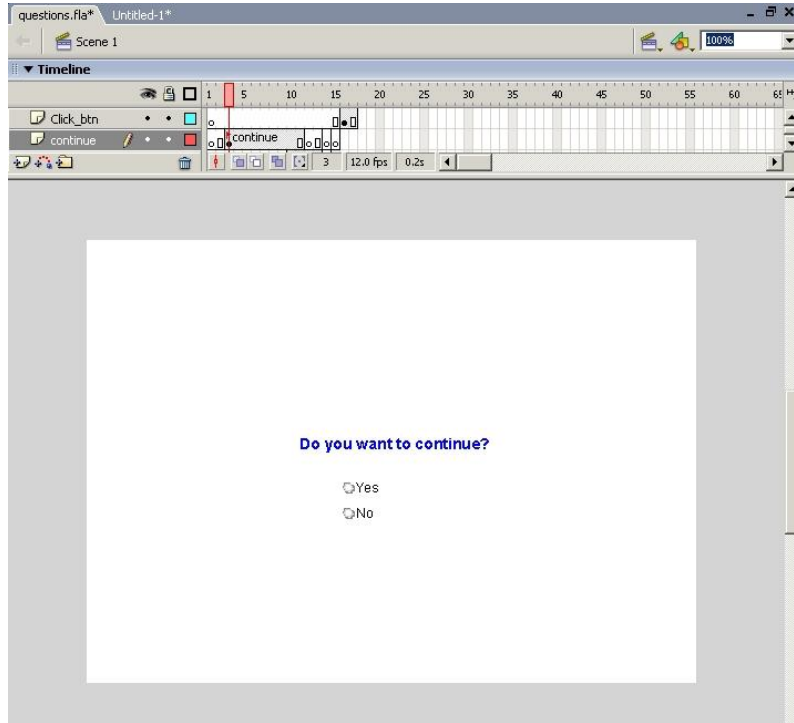


Figure 3: FLASH frame *continue*

or **incorrect** ([Figure 5](#))

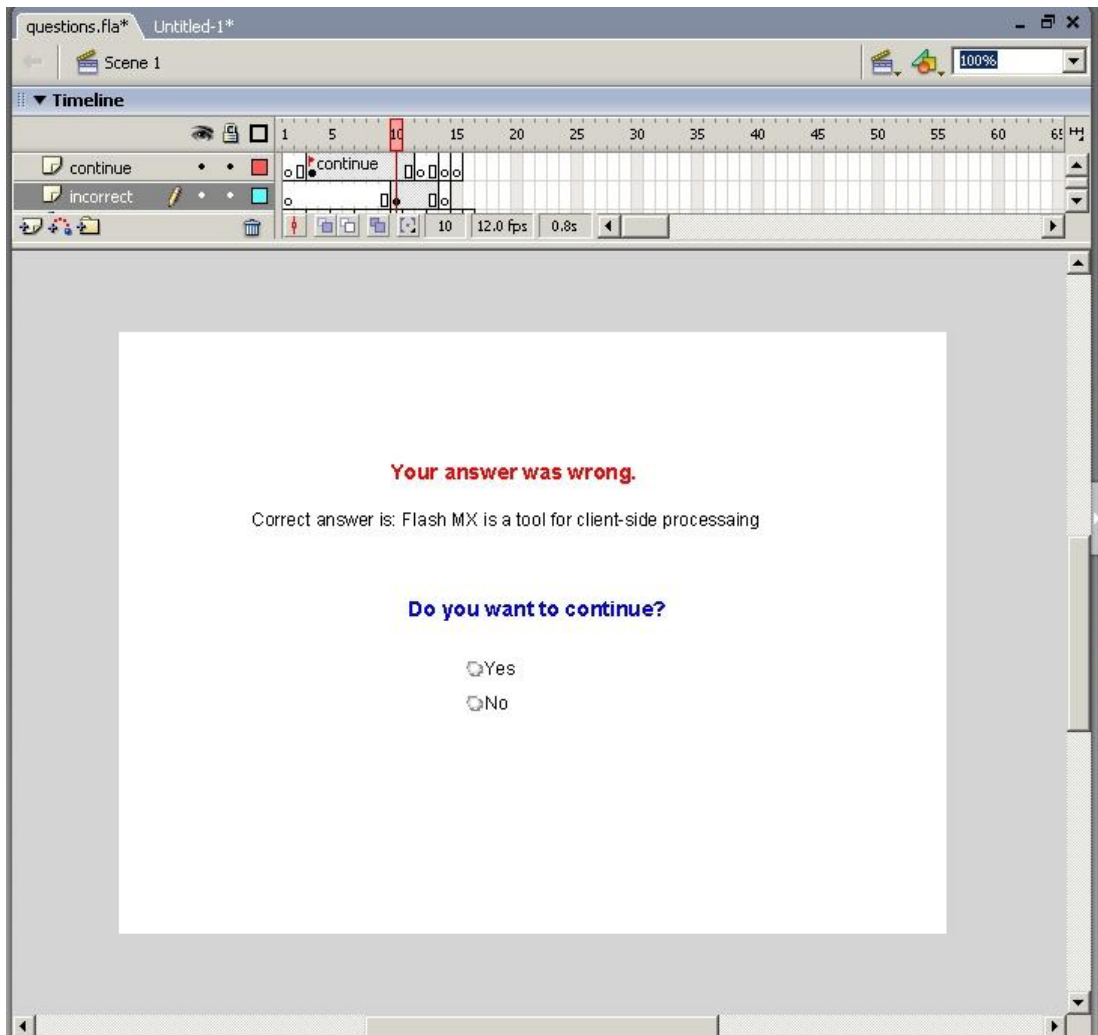


Figure 5: FLASH frame *incorrect*.

depending on the button clicked. Passing **continue**, to either of these 2 frames, the content of **continue** is activated as an overlay. The content is 2 buttons, continue **Yes** or **No**. A second **script** connected to both **correct** and **incorrect**, determines if the play head should be moved back to **init** or forward to **end** ([Figure 6](#)) depending on whether the **Yes** or **No** button of the overlay is clicked.

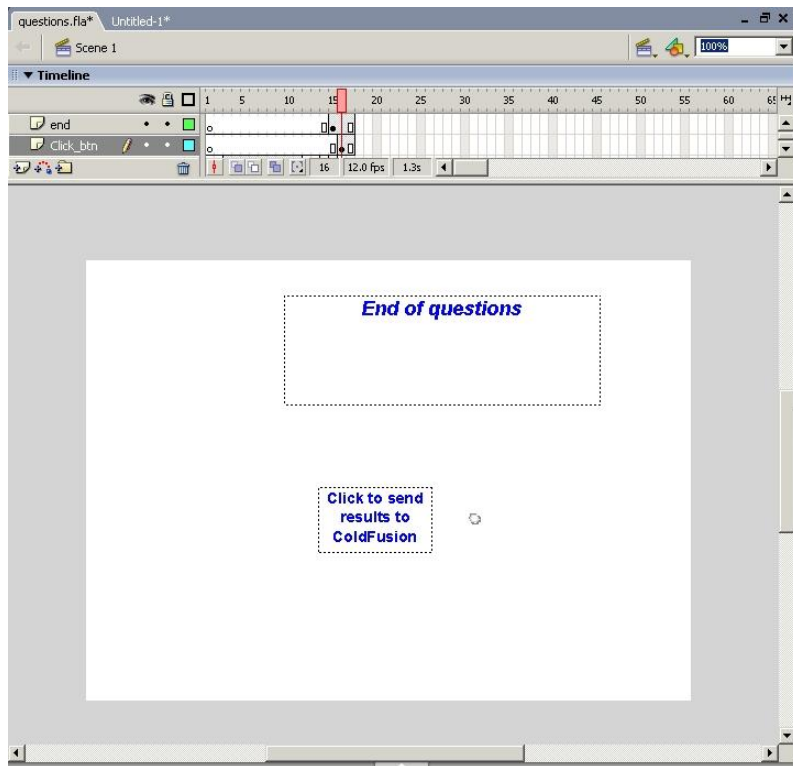


Figure 6: FLASH frame *end*.

The **ActionScript** code connected to the **init** frame looks like::

```

1. //Initialization
2. var sum;
3. stop();
4. answer1.onRelease = function(){
5.   gotoAndStop("incorrect");
6. };
7. answer2.onRelease = function()
8. {
9.   gotoAndStop("correct");
10. };
11. answer3.onRelease = function(){
12.   gotoAndStop("incorrect");
13. };
14. answer4.onRelease = function()
15. {
16.   gotoAndStop("incorrect");
17. };

```

Line 1 is an ActionScript comment while in Line a variable called **sum** is defined. Then the actions are stopped until an **event** occurs. Four events with associated functions are defined in Lines 4-17. Each corresponds to the **release** of a mouse button over a radio button in frame **init**. All functions are similar and result in **moving** the play head to one of two named frames, **correct** and **incorrect**.

The **continue** overlay frame is passed and activated when the play head is moved to either of the 2 frames mentioned above. The script associated with the overlay frame is:

```
1. //Continue
2. stop();
3. yes_btn.onRelease = function() {
4. gotoAndStop("init");
5. };
6. no_btn.onRelease = function() {
7. gotoAndStop("end");
8. };
```

The **continue** frame, which is overlaid on both the **correct** and the **incorrect** frame, has a **yes** and a **no** button. When the mouse button is released over the **yes** frame button, the play head is returned to the **init** frame, while when the mouse button is released over the **no** frame button, the play head is moved to the **end** frame. Neither the **correct** nor the **incorrect** frame has separate scripts associated to them.

The script connected to the **end** frame looks like this:

```
1. //End
2. if(sum<1) {sum=0};
3. this.createTextField("Result_txt",1,200,200,200,30);
4. Result_txt.text="Your number of correct answers is:" + sum;
```

The last **script** contains code which sends the results back to a **ColdFusion** server-side program for storing in a database or for further processing .

```
1. //Transmit
2. dataSender = new LoadVars();
3. button_submit.onrelease = function ()
4. dataSender.send("
5. http://nordbotten.net/courses/HSH/sessions/session13/examples/flash/feedback.cfm", "_self",
  "GET"); } stop();
```

The **transmit** script creates an **object** called **dataSender** with a **method** **send**. In our simple example, no data is sent. The method simply requests a **CFMX** page, **feedback.cfm**, from the server ([Figure 7](#)). A more efficient technology, **FLASH Remoting**, exists for client-server collaboration. As **FLASH** itself, **FLASH Remoting** is outside the scope of this course.

This is sent from the CFMX server to confirm that the results have been received from the FLASH test example.

Figure 7: CFMX response to FLASH request..

Exercises

- a. **Free** manuals and guides are available on the net. You are encouraged to retrieve information necessary for taking **advantage** of the fact that practically all browsers can interpret JavaScript scripts if you can design them.
- b. A challenging task will be to work out how a more **complete quiz** can be developed by means of JavaScript.

Session 14: Components

CFMX Component technology

ColdFusion **Component** is another approach to systemizing the code for convenient future re-use by the developer or for distribution to other developers. The **Component** feature has some of the **object-oriented** characteristics, but is not considered a true object-oriented programming approach. In **CFMX**, components serve also as an **interface** to other technologies such as **Flash** and **Web services**. A component usually contains **a set of functions** also called methods, and is similar to a **UDF** library file.

For the application developer, the components are **hiding the details** of the code and focusing on the **functionality**. The component technology permits the use of all **CFMX** tags and functions, can access databases, protocols, foreign systems and code, which makes the technology extremely **flexible**.

Authorization example

In Session 7, we discussed **authorization** and **authentication**. It was pointed out that authorization and authentication were processes used in many applications for security reasons. If authorization, i.e. providing users with **credentials** for entering the application, can be designed in a way which permits the code to be **re-used**, we can save ourselves for future re-design of the same task for each application.

To illustrate the principles of **CFMX Components**, we shall design a simple application for generating unique accesscodes as a component, which can easily be embedded into all applications of an organization.

The basic **requirements** to the accesscodes are:

1. The accesscode should have a **large number** of possible values compared with those used,
2. The accesscode should be assigned **randomly** to the users,
3. The accesscode should be unique, i.e. not assigned to more than one user.

An accesscode alone is **not a very safe** precaution against intruders, but as long as the system does not contain **sensitive** personal or **economic** information of importance, a simple accesscode can be satisfactory. If you want higher security level, consider to combine the accesscode with for example the user's **email address**, his birth place, etc.

Component for generating unique random numbers

We envisage that the component to be developed should be called from an application template to which the accesscode is supplied and which saves it in a database for future authentications. In our session example, the application is substituted by a single template which **calls** on a component, and **displays** an unused accesscode as requested. [Figure 1](#) illustrates the structure of a component with 4 functions (methods) included.

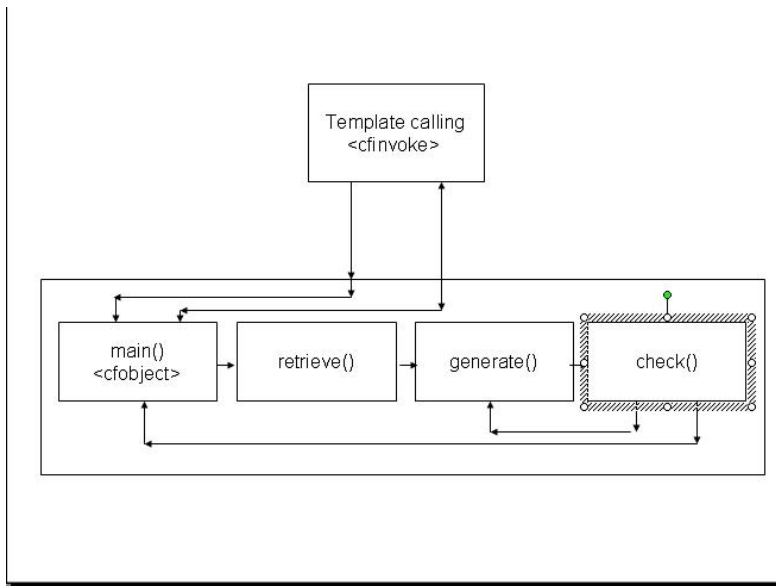


Figure 1: Model of component component *access_no*

A main function controls the process. It can be compared with the main procedure of a traditional computer program. In addition, 3 more functions/methods are included in the component. The 4 functions of the component are:

1. **main()**, which controls the processing within the component and returns the result to the calling template. It also displays the list of all used access numbers.
2. **retrieve()**, which retrieves the list of previously used numbers,
3. **generate()**, which generates a random number within the specified area 10000-99999,
4. **check()**, which checks if the generated number is unused or already used.

The component is **invoked** from template, *access_no.cfm*. This template plays the **role** of an application needing the processing of the component. For the purpose of demonstration, this template displays a list of the most **recently** generated access numbers, and the requested **new** number:

1. `<!-- access_no.cfm -->`
2. `<cfinvoke component="access_component" method="Main" returnvariable="access_no"></cfinvoke>`
3. `<center>`
4. `<h2>A vacant access number is:</h2>`
5. `< cfoutput>`
6. `#access_no#`
7. `</cfoutput>`
8. `</center>`

The **CFINVOKE** tag in Line 2 calls the function/method **main()** within the component. The value of the variable **access_no** is expected to be returned by the component to the calling

template. In the example, Lines 4-7 display the values of the requested new number (See [Figure 2](#)). A component can also be called by means of the **CFOBJECT** tag, the *createobject()* function in **CFScript** and via a *http* call. Consult **RBB** for detailed information on how to call a component.

A vacant access number is:

34756

Figure 2: The results from component *access_no*

The component itself is displayed below. Note the **file extension** *.cfc* used for identifying component files. The function *main()* calls on the 3 other functions, and depending on whether the value of the variable *accept_no* is 1 or 0. The *main()* also returns the value of *access_no* to the calling template if *accept_no* is 1, or repeats execution of functions *generate()* and *check()* until *accept_no()* becomes 1.

```

1. <!-- access_component.cfc --->
2. < cfcomponent hint="This component returns an unused accessnumber in the interval 10000 - 99999">
3. <!-- main() --->
4. <cffunction hint="Controls the process" name="main">
5. < cfset used_no="#retrieve()#">
6. < cfset accept_no="0">
7. < cfloop condition="#accept_no# EQ 0">
8. <cfset access_no="#generate()#">
9. < cfset accept_no="#check(access_no,used_no)#">
10. </cfloop>
11. <cfreturn access_no>
12. < /cffunction>
13. <!-- retrieve() --->
14. <cffunction hint="Retrieves list of used numbers" name="retrieve">
15. <cfif Not FileExists('#session.path#\sessions\session12\examples\component\Used_no.txt')>
16. < cfset Used_no="10000,99999">
17. < cffile action="write" file="#session.path#\sessions\session12\examples\component\Used_no.txt"
    output="#Used_no#">
18. < /cfif>
19. <cffile action="read" file="#session.path#\sessions\session12\examples\component\Used_no.txt"
    variable="Used_no">
20. <cfreturn Used_no>
21. </cffunction>
22. <!-- generate() --->

```



```

23. <cffunction hint="Generates a random number in the interval 10000 - 99999" name="generate">
24. < CFSET temp=randomize(second(Now()))>
25. < CFSET access_no=#RandRange(10000,99999)#>
26. < cfreturn access_no>
27. < /cffunction>
28. < cffunction hint="Checks whether the generated number has not been used, saves and returns it, or
    has been used and return to Generator" name="check">
29. < cfargument name="access_no" type="numeric" required="true">
30. < cfargument name="used_no" type="any" required="true">
31. <cfset accept_no="1">
32. <cfloop index="i" list="Used_no" delimiters=", ">
33. < cfif #i# EQ #access_no#>
34. < cfset accept_no="0">
35. < cfbreak>
36. < /cfif>
37. < /cfloop>
38. <cfif #accept_no# EQ "1">
39. < cfset Used_no=ListAppend(Used_no,#access_no#)>
40. < cffile action="write" file="#session.path#\sessions\session12\examples\component\Used_no.txt"
    output="#Used_no#">
41. < /cfif>
42. < cfreturn accept_no>
43. < /cffunction>
44. < /cfcomponent>

```

The function *retrieve()* tests if a file, *used_no.txt*, exists and creates the file if not. The function then retrieves the file with all previously used numbers and stores it in a **list** named *Used_no* which is returned to the *main()*. The function *generate()* is quite simple. It generates a random number, *access_no*, between 10000 and 99999, and returns the number to the *main()*.

Function *check(Used_no,access_no)* is called and checks the random number value *#access_no#* against the values in the list of used number. If the number is unused, *check()* saves the updated list of used numbers and returns *accept_no* with value 1, and main will be able to return *#access_no#* to the template which called the component, else the Lines 7-10 are repeated.

Note that the path to the component in the **FILE** tags is assumed specified in the *Application.cfm*. In the example, a small *index.cfm* template is included to start the generation.

Introspection

A component can be stored in the folder of the application using it, or in any other folder below the web root. Its properties can be inspected by so-called **Introspection**. Call the **URL** of the component by means of your browser. This call activates the **CFC Explorer** which requires your username and password. It displays a page with the meta-information about the component.

[Figure 3](#) demonstrates part of the information for the component *access_no.cfc*. Note that the addresses given are not real addresses.

Component access_no	
This component returns an unused accessnumber in the interval 10000 - 99999	
hierarchy:	WEB-INF.cftags.component courses.cfm.sessions.session13.examples.Component.access_no
path:	C:\net-copy\Courses\cfmx\sessions\session13\examples\Component\access_no.cfc
properties:	
methods:	check , generate , main , retrieve
* - private method	
check	
check (<i>required numeric</i> access_no, <i>required any</i> used_no)	
Checks whether the generated number has not been used, saves and returns it, or has been used and return to Generator	
Output: enabled	
Parameters:	
access_no : numeric, required, access_no	
used_no : any, required, used_no	
generate	
generate ()	
Generates a random number in the interval 10000 - 99999	
Output: enabled	
main	
main ()	
Controls the process	

Figure 3: Introspection of component *access_no*

Exercises

- You are encouraged to **copy and try** out the examples for yourself. A challenging task will be to work out how a component can be called by means of the **CFOBJECT** tag.
- There are many ways of constructing and using components. **RBB** discusses some of them in Chapter 22. Components build on **UDF** is discussed in detail in **RBB** Chapter 20. You are recommended to **read** this chapter, and try out some of the suggestions.
- Consider how you would **reorganize** your own project if you should have implemented it by means of components.

Session 15: Web services

Web services

Web services are the last option for re-using code to be discussed in this course. In addition to re-using code, web services also contribute to making web processing more efficient by **distributed** processing because they permit a number of applications on different locations to share as well software as computing facilities.

The technology can best be explained by [Figure 1](#). Assume a number, **k**, of end users (clients)

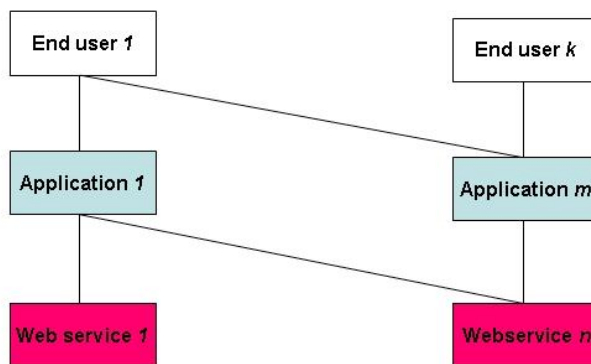


Figure 1: Interaction among end users, applications and web services.

requesting applications from a set of **m** servers. Some of the applications require, however, **special components** residing in a second set of **n** servers which are providing the result of their components as **Web Services**. The services are, like the applications of the first set of servers, offered on the net and can be **integrated** in the different applications just as if they were local components. To the end users, the applications they request will **appear** as if the applications are executed by the application servers. In the figure, it is, as an example, assumed that end user 1 requests both **application 1** and **m**, while end user **k** only request **application m**. To deliver, **application 1** needs both web services 1 and **n**, while **application m** only needs **web service n** to deliver. Of course, most applications do **not** at all require the services of another server.

Web service is a concept which has entered the web arena in the last few years. Even though the technology is supported by major vendors and seems to have a potential for contributing to **remote, distributed** processing on the net it is still **controversial**.

It uses a set of standards/technologies including:

- TCP/IP
- HTTP,
- XML,
- Universal Description, Discovery and Integration (UDDI).

- Web Service Description Language (WSDL)
- Simple Object Access Protocol (SOAP),

The **Web service** is an open source technology supported by many **important actors** in the web domain. A **Web service** is said to be **created** by its developers, and **consumed** by their users. **ColdFusion MX** supports creation as well as consumption of **Web services** by the **CFMX Component technology** discussed in a previous session. [Journals](#) for web services have been established for developers and consumers to maintain their knowledge about the state of the art.

In this session we review **UDDI**, **WSDL** and **SOAP**, and discuss how you create and consume a web service.

Universal Description, Discovery, and Integration

The first condition for being able to consume a web service is to know of its existence. An **UDDI** registry is a facility established for locating sources of web services. It can be a local enterprise restricted or an open, public registry. Market actors as [IBM](#) and [Microsoft](#) are running global **UDDIs**. A popular **UDDI** is [XMethods](#) to which you can get your own web service registered. You can also publish your web services in your own **UDDI**. **CFMX** does not contain any special features for establishing an **UDDI**, but the design and development of an **UDDI** can be done by means of **CFMX**..

Web Service Description Language

WSDL is an **XML format** for describing net services. Description of web services are therefore standardized in an **XML** based document prepared by the **Web Service Description Language**, **WSDL**. The **WSDL** describes the requirements for **consuming** a web service. The **CFMX** developer does not need to be concerned about writing the **WSDL** for the component he/she develops, it is done **automatically** by **CFMX**. As we shall see later in this session, likewise to view the **WSDL** of an existing component does not require more than typing the **URL** of the component with a short extension. A visit to [W3C](#) gives useful and official information on web services.

Simple Object Access Protocol

SOAP provides an **XML** messaging framework used by web services. For a **CFMX** developer, it is not necessary to learn the **SOAP** technology, or be at all concerned about it, since it is handled **automatically** by the **CFMX web service technology**.

Web services creation and consumption in CFMX

Consider the following scenario. A number of developers are working on different web applications within a worldwide organization. Applications need all accesscode authorization, and the codes must be unique for each user independent of application. A **Web service** providing unique accesscodes can be the solution.

A web service example

We use the component developed above to serve in the scenario outlined above. A few changes have to be done. In CFMX, **Web services** are created using **components**. Each **CFFUNCTION** tag in a **Web service** component needs an attribute **ACCESS="remote"** and an attribute **RETURNTYPE="string|number|boolean| etc"**. Except for these two conditions, the **Web service component** looks and works as an ordinary **component** and may be considered as an "external" component from a CFMX developer's point of view. The following example is therefore quite similar to the example discussed above in connection with components.

The **use** of a **Web service** is referred to as service **consumption**. An application needing support from a **Web service** must have a way to call on the service. The application server can, just as if it should apply one of its own components, use the **CFINVOKE** tag. However, in the case of calling a **Web service component**, the tag must contain an attribute **WEBSERVICE="<name>"**. The value **<name>** is either the **URL** of the **WSDL** of the service, or an alias name registered by the **CFMX Administrator** of the **consuming** server. The call for the service is embedded in a template of the consuming application.

The calling is illustrated by *access_no.cfm* template, which requests a unique accesscode, and displays the received code for the client who wants to see the example. The template is very similar to that used in the component example. The only difference is other attributes in the **CFINVOKE** tag. Note that **WEBSERVICE** and **RETURNVARIABLE** are required attributes.

The web service is supposed to be registered by the **CFMX Administrator** of the consuming server, and named (alias) *access_no*.

1. <!-- access_no.cfm -->
2. <cfinvoke webservice="access_no" method="Main" returnvariable="access_no"></cfinvoke>
3. <center>
4. < h2>A vacant access number is:</h2>
5. < cfoutput>
6. #access_no#
7. </cfoutput>
8. </center>

In the same way as local components, **web services** can be called in alternative ways, e.g. by means of the tag **CFOBJECT**, the function *createobject()*, etc.

The **Web service** known to the application server as **access_no** differs from the local component, *access_no.cfc*, introduced above only by the function tag **remotely** called. The remaining functions are called within the component and need **no** changes.

1. <!-- Webservice: access_no ---->
2. <cfcomponent hint="This component returns an unused accessnumber in the interval 10000 - 99999">
3. <!-- main() -->

4. `<cffunction hint="Controls the process" name="main" access="remote" returntype="any" >`
5. ...
6. ...

About the implementation of the example

To demonstrate a web service, we need a **client** and **2 servers**, an application server consuming the web service of the second.

Exercises

- a. You are encouraged to **copy and try** out the examples for yourself. An challenging task will be to work out how a component can be called by means of the **CFOBJECT** tag.
- b. There are many ways of constructing and using components. **RBB** discusses some of them in Chapter 22. Components build on **UDF** is discussed in detail in **RBB** Chapter 20. You are recommended to **read** this chapter, and try out some of the suggestions.
- c. Consider how you would **reorganize** your own project if you should have implemented it by means of components.
- d. You are welcome to **study** the **wsdl** of as well as and **invoke** the web service:
http://nordbotten.ifi.uib.no/webservices/access_webservice.cfc?wsdl called from an application on your own computer.

A bibliography for further studies

- Bandyopadhyay, N (2001): *e-Commerce: Context, Concepts and Consequences* . McGrawHill. NY.
- Brook-Bilson, R. (2003): *Programming ColdFusion*. Second Edition. O'Reilly. Ca.
- Castro, E. (1999): *PERL and CGI*. Peachpit Press. CA.
- Flanagan, D. (2001): *JavaScript: The Definitive Guide*, 4th Edition. O'Riley, Ca.
- Forta, B. (2003): *Certified Macromedia ColdFusion MX Designer Study Guide*. Macromedia Press.
- Forta, B. (2003): *The ColdFusion MX Web applications Construction Kit*. Macromedia Press.
- Forta, B. a.o.(2003): *Advanced ColdFusion Applications Development*. Third Edition. Macromedia Press.
- Hatfield, B. (1999): *Active Server Pages for Dummies*. IDG. CA.
- McFarland, D. S. (2004): *Dreamweaver MX 2004: The Missing Manual*. O'Riley, Ca.
- Mohammed, R., Fisher, R., Javorski, B., and Cahill, A.(2001): *Internet Marketing: Building Advantage in the networked Economy*. McGrawHill. NY.
- Moock, C. (2002): *ActionScript for Flash MXC. The Definitive Guide*. 2nd Edition. O'Reilly.Ca.
- Nordbotten, J.C. and Nordbotten, S. (1999c): *Search Patterns in Hypertext Exhibits*. 32th Hawaii International Conference on System Sciences. Hawaii. 1999.
- Nordbotten, J.C and Nordbotten, S. (2001a): *Perception of Statistical Presentations Investigated by Means of Internet Experiments*. 34th Hawaii International Conference on System Sciences. Hawaii. 2001
- Nordbotten, J.C. and Nordbotten, S. (2001b): *Student Search Patterns in a Statistical Web-DB*. Human-Computer Interface Conference
- Reding, E.(2001): *Building an E-Business: From the Ground Up*. McGrawHill. NY.
- Sklar, D. (2004): *Learning PHP 5*. O'Reilly,Ca.
- Trachtenberg, A. (2004): *Upgrading to PHP 5*. O'Reilly. Ca.
- Wall, L., Christiansen, T. and Schwartz, R. (1996): *Programming Perl*. O'Reilly. Ca.