mxdu 2004

MACROMEDIA DEVCON ASIA PACIFIC › SYDNEY › AUSTRALIA

CFCs from the ground up

A fundamental introduction in ColdFusion Components

Kai Koenig, *Software Architect*
msg at.NET GmbH, Neuss (Germany)

.msg at.NET

Agenda

- Software components
- ColdFusion Components 101
- Advanced CFC development

About me

- **Working as software architect at msg at.NET GmbH, the only MM Premier Alliance partner in Europe**
  - Web applications
  - Component architecture models
  - Object-oriented development
  - Mobile devices
- **MM Advanced Certified CFMX Developer, MM Certified Flash Developer**
- **MM Certified Instructor for CF Curriculum**
- **Community work**
  - User Group Manager CFUG Nordwest
  - Blog in Black (www.bloginblack.de) – Germanys 1st MM-server related Weblog
- **Email: koenig@msg-at.net**

Agenda

- Software components
- ColdFusion Components 101
- Advanced CFC development

Software components

- **Formal:**
  - Using formal notations (relation-theory, graph grammars -› UML) to define software components
- **Semi-formal:**
  - „A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“
    (Szyperski, C., „*Component Software: Beyond Object-Oriented Programming*.“)
- **Easy:**
  - „Reusable building block of software“

Software components

- **Why ColdFusion Components?**
  - CFMX is a server-side scripting environment for Rich Internet Applications
  - CFCs are the architectural foundation of MX applications, they define and provide services for RIAs built with Flash, XML Web Services etc.
  - But: they‘re also useful for pure CFML development:
    - The pre-CFC CFML approach lacked in ways to separate between business logic, presentation logic and data layer, often all was mixed up in one template (see next slide)
    - Hard to build reusable CFML templates and to bundle frequently performed business logic

Software components


```
<HTML>
<HEAD>
<TITLE>User list</TITLE>
</HEAD>

<BODY>
<CFQUERY NAME="selectUserList" datasource="myDS">
SELECT Name,FirstName,PermissionSet
FROM Users
ORDER BY Name
</CFQUERY>

UserList:<BR>
<CFOUTPUT QUERY="selectUserList">
#Name#, #FirstName#: #PermissionSet#<BR>
</CFOUTPUT>
</BODY>
</HTML>
```

Data access

Presentation logic

Software components

- **CustomTags / UDFs?**
  - CustomTag:
    - Tags functionality is always public
    - Everybody could do everything with your application code!
    - Custom Tags were not designed to play the part of a software component
  - UDF:
    - Developer has to include UDF definition on every page using it, no easy way to group similar functions logically
    - Just local usage of the UDFs possible
    - No way to provide functionality to caller technologies besides ColdFusion templates
    - You‘re just allowed to use ‹CFSCRIPT› functionality in UDFs

Agenda

- **Software components – sort of a definition**
- **ColdFusion Components 101**
  - Creating a static CFC
  - ‹CFINVOKE›
  - Passing arguments
  - Self-documentation
  - Invocation vs. instantiation
  - Instance-based CFCs
  - Persistence
- **Advanced CFC development**

CFCs 101 - Creating a static CFC

mxdu 2004

- **Four steps to define a basic ColdFusion component:**
  - Define the component
    - ‹CFCOMPONENT›
  - Define the components functionality (methods)
    - ‹CFFUNCTION›
  - Define the information the methods require to work (arguments)
    - ‹CFARGUMENT›
  - Define the functions result when it executes
    - ‹CFRETURN›

CFCs 101 - Creating a static CFC

mxdu 2004

- **Step 1:**
  - Define the component using ‹CFCOMPONENT›
  - ‹CFCOMPONENT› is the top level wrapper tag to create and define a CFC object
  - Two optional attributes for producing enhanced self-documentation: HINT and DISPLAYNAME, covered later
  - Component name is defined by naming the .cfc-file

```
<CFCOMPONENT>
</CFCOMPONENT>
```

CFCs 101 - Creating a static CFC

mxdu 2004

- **Step 2:**
  - Define methods using ‹CFFUNCTION›
  - Each function that a CFC is able to perform is defined in a named block of code wrapped by this tag
  - Attributes: NAME (function name, required), OUTPUT (allowed to send output to the browser) and RETURNTYPE (type of the data returned by that function)

```
<CFFUNCTION name="calculate2plus2" output="no"
            returntype="numeric">
</CFFUNCTION>
```

CFCs 101 - Creating a static CFC

mxdu 2004

- **Step 3:**
  - Assume our method doesn't need any external information passed in... discussed later
- **Step 4:**
  - Define the methods result when executing using <CFRETURN value>
  - value is allowed to be a single expression, returning multiple expressions is possible by using structures

```
<CFFUNCTION name="calculate2plus2" output="no"
            returntype="numeric">
  <CFRETURN 2+2>
</CFFUNCTION>
```

CFCs 101 - ‹CFINVOKE›

- Using ‹CFINVOKE› to invoke a CFC method:

```
<CFINVOKE component="cMath"
          method="calculate2plus2"
          returnvariable="aSum">

<CFOUTPUT>
#variables.aSum#
</CFOUTPUT>
```

- ColdFusion instantiates cMath.cfc and invokes calculate2plus2() on that instance
- The methods result is stored in the variable defined in RETURNVARIABLE attribute

CFCs 101 - Passing arguments

- **Revisiting step 3 now...:**
  - Most often, methods need external information to do their work, passing information to a method is possible using ‹CFARGUMENT›
  - ‹CFARGUMENT› tags are placed inside the function before any other code
  - NAME attribute is required, REQUIRED/DEFAULT are used to define attributes obligating and if not required to define a default value for them, TYPE defines the arguments type

```
<CFFUNCTION name="calculateSum" output="no"
            returntype="numeric">
  <CFARGUMENT name="nAddent1" required="Yes"
              type="numeric">
  <CFARGUMENT name="nAddent2" required="No"
              type="numeric" default="0">
  <CFRETURN arguments.nAddent1 + arguments.nAddent2>
</CFFUNCTION>
```

CFCs 101 - Creating a static CFC

mxdu 2004

- **The completed basic static CFC:**

```
<CFCOMPONENT>
  <CFFUNCTION name="calculateSum" output="No"
              returntype="numeric">

    <CFARGUMENT name="nAddent1" required="Yes"
                type="numeric">
    <CFARGUMENT name="nAddent2" required="No"
                type="numeric" default="0">

    <CFRETURN arguments.nAddent1 +
                arguments.nAddent2>
  </CFFUNCTION>
</CFCOMPONENT>
```

CFCs 101 - Passing arguments inline

- Using ‹CFINVOKE› to invoke a CFC method:

```
<CFINVOKE component="cMath" method="calculateSum"
          returnvariable="aSum"
          nAddent1="4"
          nAddent2="7">
<CFOUTPUT>
#variables.aSum#
</CFOUTPUT>
```

- Arguments are passed inline of the ‹CFINVOKE› tag

CFCs 101 - ‹CFINVOKEARGUMENT›

- Another way to pass arguments to a method is using ‹CFINVOKEARGUMENT›

```
<CFINVOKE component="cMath" method="calculateSum"
          returnvariable="aSum">
  <CFINVOKEARGUMENT name="nAddent1" value="4">
  <CFINVOKEARGUMENT name="nAddent2" value="7">
</CFINVOKE>
<CFOUTPUT>
#variables.aSum#
</CFOUTPUT>
```

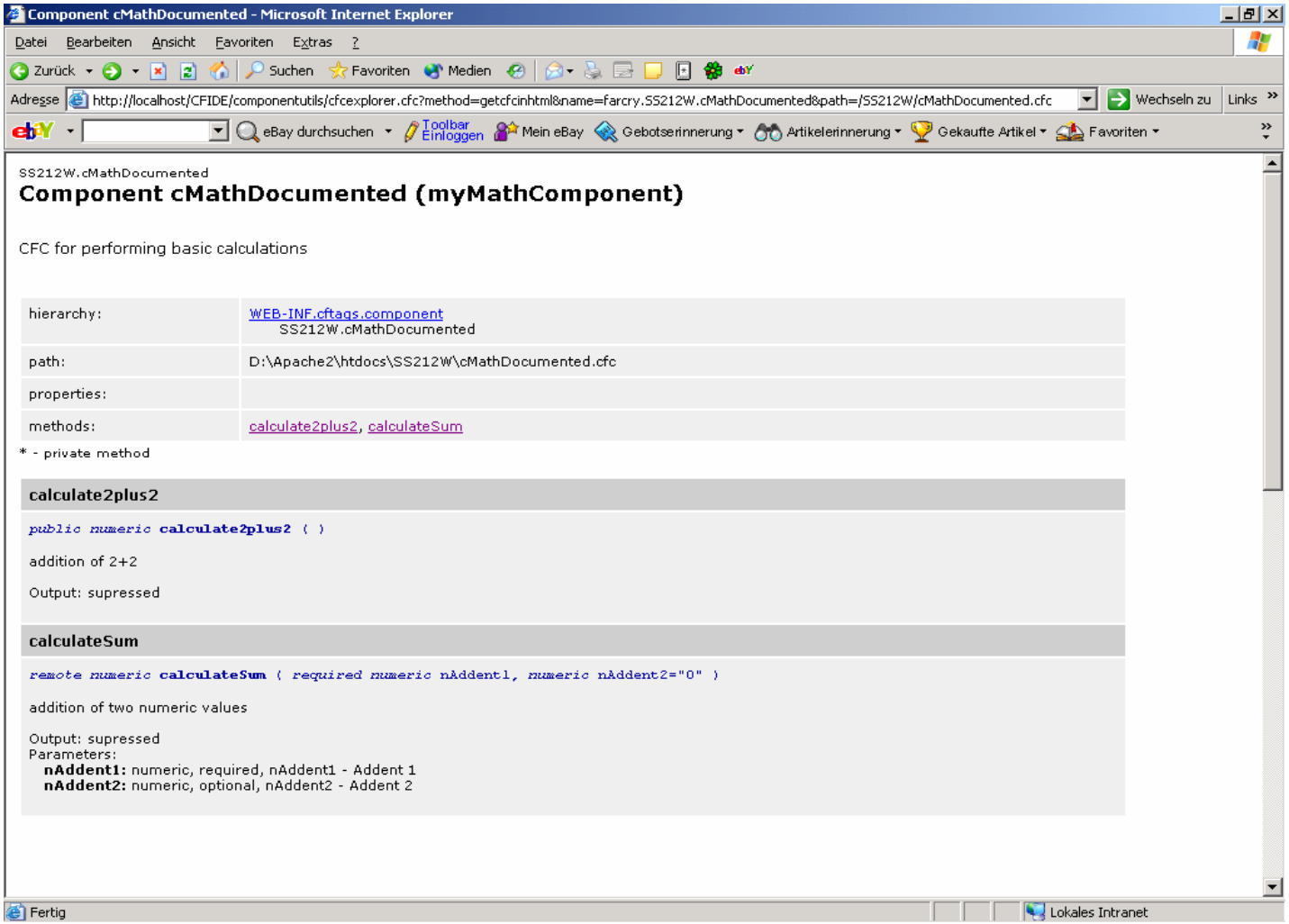
- Might be an interesting solution if there is a need to pass arguments depending on external conditions

CFCs 101 - Self-documentation

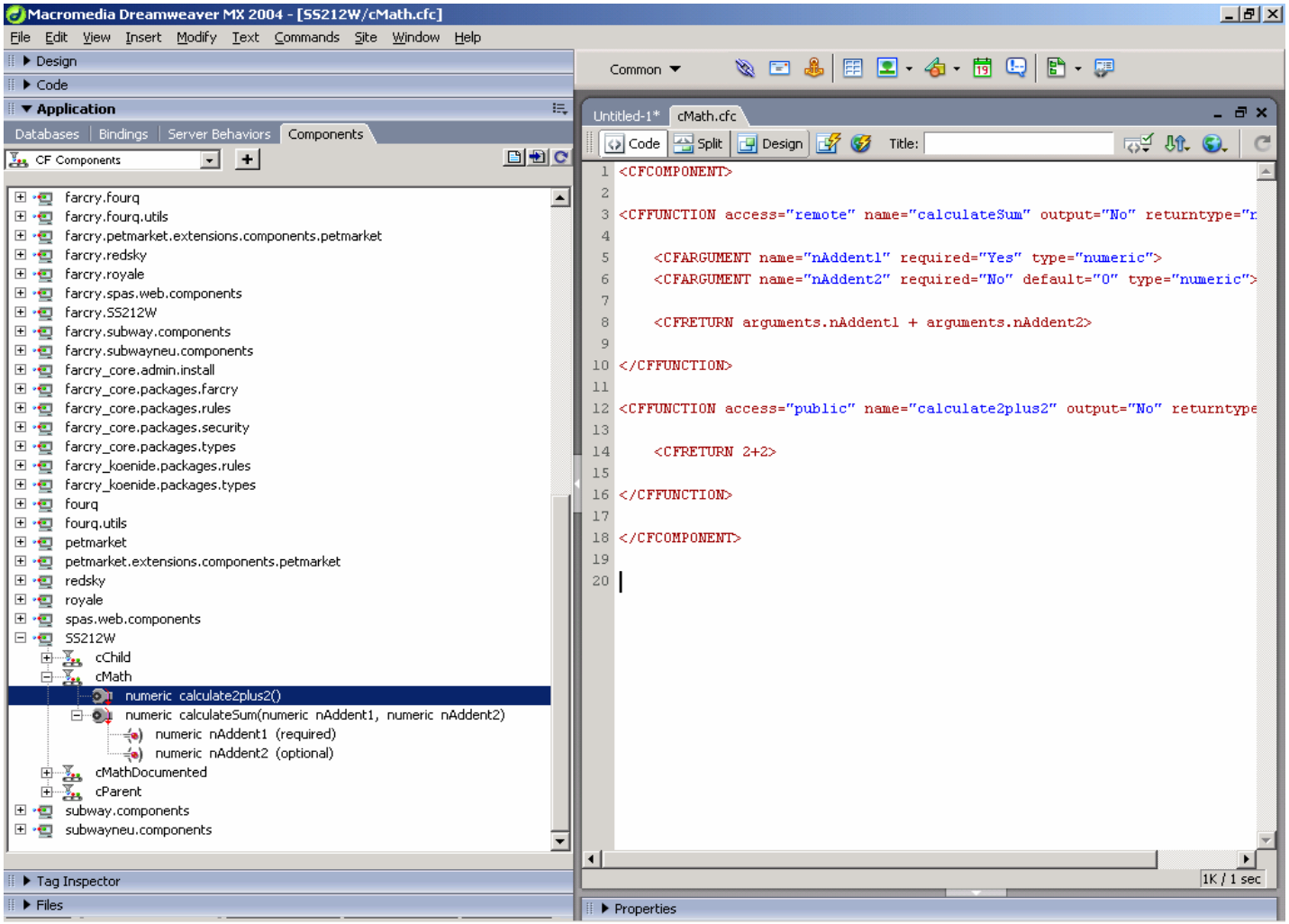
mxdu 2004

- **‹CFCOMPONENT›, ‹CFFUNCTION› and ‹CFARGUMENT› allow to specify DISPLAYNAME and HINT attributes for enhanced self-documentation**
- **Demonstration:**
  - CFC Browser
  - Self-documentation features in DWMX 2004

CFCs 101 - Self-documentation



CFCs 101 - Self-documentation



CFCs 101 - Instantiation + invocation

mxdu 2004

- **Important to understand the differences between instantiation and invocation:**
  - Instantiation means that an instance of a CFC is created
  - Invocation means calling (invoking) a method of a CFC
- **ColdFusion offers a lot of ways to support both concepts for dealing with CFC methods**
- **We already learned using ‹CFINVOKE› for invocation**

CFCs 101 - Using ‹CFOBJECT›

- **‹CFOBJECT› allows to instantiate a CFC before invoking a method:**

```
<CFOBJECT component="cMath" name="aMathInstance">

<CFINVOKE component="#aMathInstance#"
          method="calculateSum"
          returnvariable="aSum"
          nAddent1="4" nAddent2="7">

<CFINVOKE component="#aMathInstance#"
          method="calculateSum"
          returnvariable="anotherSum" nAddent1="9"
          nAddent2="15">

<CFOUTPUT>
#variables.aSum#<BR>
#variables.anotherSum#
</CFOUTPUT>
```

CFCs 101 - Using ‹CFOBJECT›

- **First step: instantiating the cMath.cfc with ‹CFOBJECT› ColdFusion creates a reference on the CFC object**
- **Several ‹CFINVOKE› tags are called on the specific instance of the CFC**
  - Creating the instance once, there is no need to automatically create and throw away the instances with each method invocation
  - Instantiating a CFC once is much faster when there are a lot of method calls to the same CFC on the same page compared to just using ‹CFINVOKE›

CFCs 101 - Instance-based CFCs

mxdu 2004

- **Instance-based CFCs allow to associate properties and methods**
  - Improved object-oriented development style
  - CFC methods are able to share instance data instead of passing the same information with each method call
- **Several variable scopes:**
  - variables: inside a component VARIABLES scope refers to private instance data
  - this: inside a component THIS scope refers to public instance data
  - var: inside a method the VAR keyword refers to local method data

CFCs 101 - Instance-based CFCs

mxdu 2004

```
<CFCOMPONENT>
  <!--- constructor area --->
  <cfset this.Name="Kai">
  <cfset this.Age=29>

  <CFFUNCTION name="func1"...>
    ...
  </CFFUNCTION>

  <!--- constructor area --->

  <CFFUNCTION name="func2"...>
    ...
  </CFFUNCTION>
  <!--- constructor area --->
</CFCOMPONENT>
```

CFCs 101 - CFC constructors

- **All code placed outside of ‹CFFUNCTION› is executed only when an instance of the CFC is created:**
  - Comparable to Javas constructor in classes (implicit)
  - Use that area to define variables and preset instance data
- **Best practice:**
  - Using an explicit init() method as a constructor returning the CFC instance to the caller

```
<CFFUNCTION name="init" access="public"
            returntype="cMath">
  ...
  <CFRETURN this>
</CFFUNCTION>
```

CFCs 101 - Persistence

- **Remember locking when using shared scopes**
  - Scope lock in SESSION scope
  - Use named locks in APPLICATION/SERVER scope
  - Read operations usually do not need to be locked
  - Write operations should be locked to avoid race conditions
- **It is important to watch memory usage and scope timeouts when dealing with a lot of scope persistent CFC instances**
- **CFC methods are cached, when making changes to a method, the persistent scope has to be cleared first**

CFCs 101 - Persistence

- Instance-based CFCs lifetime is equivalent to the pages lifetime
- It might make sense to place CFC instances in memory persistent scopes
  - Userdata
  - Shopping carts etc.
- Use SESSION scope for session related instances
- Use APPLICATION/SERVER scope for wider usage range - with just one app on the server, SERVER scope is preferable (marginally faster)

Agenda

- Software components – sort of a definition
- ColdFusion Components 101
- Advanced CFC development
  - Security
  - Remote invocation

Advanced CFCs - Security

- **The CFC architecture provides two ways of setting up security**
- **Role-based security:**
  - Using ColdFusion security (‹CFLOGIN› etc.) to restrict CFC method access
- **Visibility of methods:**
  - Using Java-like visibility attributes to restrict access to methods depending on the client security context

Advanced CFCs - Role-based security

- If the application is using the ColdFusion Authentification framework (‹CFLOGIN› and ‹CFLOGINUSER›) method access could be restricted by providing ROLES attribute in method definition:

```
<CFFUNCTION name="calculateSum" output="No"
            returntype="numeric"
            roles="teacher,student">
  ...
</CFFUNCTION>
```

- Just users authentificated with one of the given roles are able to invoke the method
- Omitting the attribute allows access by all users

Advanced CFCs - Method visibility

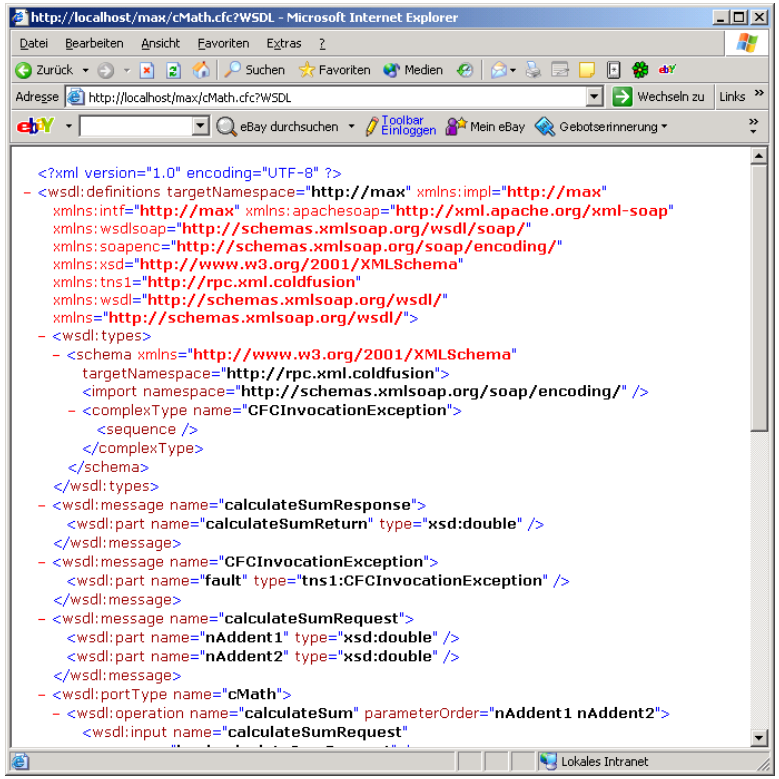
- **The CFCs ACCESS attribute provides a mechanism to restrict usage of methods to a client security context:**
  - Public: available to a local page or another local CFC
  - Package: available to the declaring CFC, CFCs of the same package or CFCs extending that component
  - Private: available to the declaring CFC and CFCs extending that component
  - Remote: available to local or remote page or another local or remote CFC. Also accessible via URL, Flash Remoting clients or as a XML web service

Advanced CFCs - Integration

- **Methods which are allowed to be invoked remotely are available for non-CFML clients:**
  - Flash Remoting
    - Allows to invoke CFC methods out of Flash MX (2004) applications
    - CFCs are one of the possible server-side application models for Macromedias Rich Internet Application vision
  - XML WebServices
    - Apache Axis 1.1 included in CFMX 6.1 provides an easy way to publish a CFC method as a web service method
    - ColdFusion web services are invoked through calling the .cfc file appending ?WSDL as a query string

Advanced CFCs - Remote invocation

- **Calling** `http://<server>:<port>/<folder>/<componentname>.cfc?WSDL` **delivers an on the fly created WSDL description**



Advanced CFCs - Encapsulation

mxdu 2004

- **Using CFCs for encapsulation purposes of third party software**
- **Leveraging non-CF APIs for usage in CFMX applications**
  - ImageJ – image manipulation
  - HTMLDOC – creating .pdf files on the server

Summary

- ColdFusion components are the architectural foundation for the service layer of RIAs and multi-tiered CFML applications
- Quite easy to build a basic and static CFC
- Invoking and passing arguments to a CFC is supported in several ways
- Instance-based CFC are comparable to objects
- CFCs support security and remote invocation

References

- CFMX 6.1 Documentation “Developing CFMX Applications”, chapter 11
- Rob Brooks-Bilson: “TopTen Tips for ColdFusion CFC Developers”
  http://www.oreillynet.com/pub/a/javascript/2003/09/24/coldfusion_tips.html
- Sean Corfield: “ColdFusion MX Coding Guidelines”
  http://www.corfield.org/coldfusion/coding_standards/

References

- Ben Forta: “Avanced ColdFusion MX Application Development”, chapters 16, 17, 22
- Brendan O’Hara: “Design Patterns in CF”, series of articles in CFDJ #3-#7 2003