3 Before You Begin Coding—Application Planning

THE IMPORTANCE OF APPLICATION PLANNING cannot be overstated. An application plan provides a roadmap for the development process with various waypoints and milestones signaling the completion of a phase of the plan. Developers often jump straight into the development of application code without a plan. It's great that they're eager to get started, but that's not the most important thing.

Imagine you wake up one morning and decide that you're going to compete in a local 5K road race. You show up at the advertised starting point and pay your fee to register for the race. You get your number and go over and stand with a swarming crowd of people. Someone walks over and without explaining the course or giving anyone a map, fires the starter's pistol.

What's the result? Mass confusion? A couple of things are working against you. Recall that we didn't mention whether you'd prepared for this race. We only said that you woke up and decided to run it. You showed up and made your investment, but you received no instructions and had no idea of the direction that you were to run. You had no clearly defined course, rules, or a finish line.

This same type of scenario happens all too often in application development. The fact is that there is plenty of work to be completed before a developer starts to crank out application code. The work includes planning and preparing the roadmap or framework on which the application will be structured. It also includes the planning of the physical structure of the environment on which the application will function and the development of a strict methodology for the application-development process. All are essential to the successful completion of the project.

In this chapter, we discuss how attention to architectural considerations, effective planning, and sticking with a standard development methodology for your project can help you to successfully complete your application-development project. The topics of discussion in this chapter help paint a clear picture of the application-development process and can help every developer gain a better understanding of how to effectively improve his or her contribution to the process.

Application Architecture

Application architecture is an important concept to understand. The architecture of an application describes the working parts of that application, how they are defined, and how they interact with each other (see Figure 3.1).

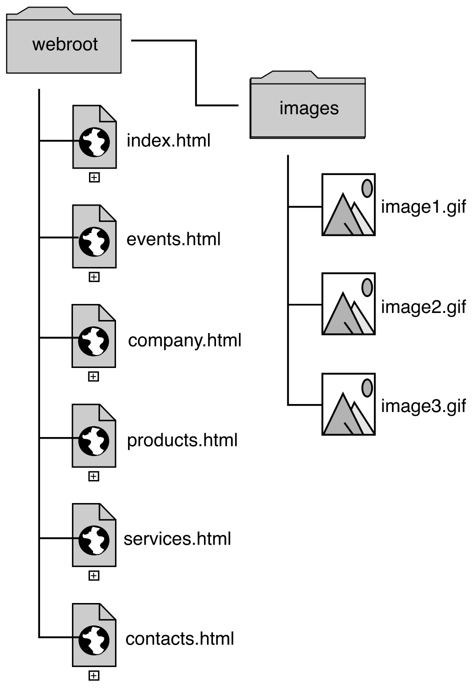


Figure 3.1 Typical web site architecture.

The application's architectural model gives developers a view of the big picture of the application rather than a close-up shot of every detail. It provides a framework that guides the interaction of various elements of the application.

The operating system (OS) or physical architecture of the server environment can influence an application's architecture. Your client's scalability requirements and their expectations in regards to the application's performance also have an effect on application architecture.

If you've ever bought something that was advertised to require "some assembly," you might be able to relate to how important it is to have a clear plan of the product and how each part interconnects and is intended to function. The instructions that are included serve much the same purpose as our application architecture. It gives you, the developer, a view of the product from several angles and from beginning to completion.

In software development, these instructions provide us with a clear vision of the organization of the application. It should detail the parts of the application and how those parts interact with other parts. It might also divide parts of the application into smaller subsystems based on their functions or behaviors. A good architecture should not stop there, however; it should also take into consideration the requirements for performance, extensibility, reuse, and presentation.

Understanding Tiered Architecture

Our discussion of application architecture now begins to take a more descriptive turn. In this next section, we take a closer look at two concepts of application architecture that define separate tiers within the application. These tiers function as a filter between logical sections of the application. The filters serve as a separator that enables the developer to group code by function and separate data from code.

Many of the concepts that we're going to talk about have a lot in common with object-oriented programming (OOP) and design principles. This is not necessarily the way that ColdFusion developers have traditionally looked at application development. As we move through the material in the next several sections, you'll see that these ideas begin to make more and more sense.

The goal of a tiered architecture is to enable the developer to separate application code into like chunks. Each tier of the application architecture contains code that has a similar purpose within the application. It is easy to take this separation of code to the extreme, but that's not what we're trying to do (nor is that very useful in most cases). Merely providing a logical division of code functionality is usually enough.

Two-Tier Architectures

We've all heard the term "client-server," right? Well, a two-tier application architecture is sometimes referred to as client-server architecture (see Figure 3.2). ColdFusion applications in their most basic form are two-tiered applications. There is code that creates an interface for the users to interact with, and there is data in the database that feeds the display. The presentation code is the first tier, or the presentation tier, and the database is the data tier.

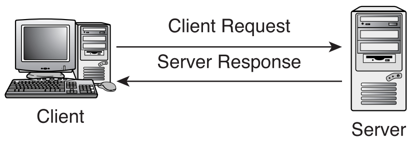


Figure 3.2 Client-server architecture.

It doesn't matter how many servers you've clustered that code across; we're not talking about physical environment architecture right now, just the functionality of the application code and how it is organized. Application architecture tiers are defined by their purpose. The following list breaks down the tiers in a two-tier architecture:

- Application tier
  - Presentation code
  - Business logic code
  - Business rule validation code
  - Component interaction code
  - Database interaction code
- Data tier
  - Storage of data

The typical two-tier application architecture is one where your application code is in one tier and your data is in another. This means that there is no separation between your presentation code, your data interface code, and your business logic code. This is typically the way that most ColdFusion developers start creating applications because typical static web sites have no need to separate code into functional groups. Check out the sample code in Listing 3.1.

Listing 3.1 **Two-Tier Application Sample Code**

```
<html>
<head>
     <title>2-Tier Application Example</title>
</head>
<body>
<cfquery name="getCustomerEmails" datasource="ICFMX">
      SELECT CustomerName, CustomerEmail
     FROM Customers
</cfquery>
<table>
     <tr>
           <td>Customer Name</td>
          <td>Customer Email</td>
     </tr>
     <cfoutput query="getCustomerEmails">
     <tr>
          <td>#getCustomerEmails.CustomerName#</td>
          <td>#getCustomerEmails.CustomerEmail#</td>
     </tr>
     </cfoutput>
</table>
</body>
</html>
```

Of course, syntactically, there is nothing wrong with this code. It returns the needed values from the database, creates a layout for the page, and outputs the results into that layout.

The problems that you can run into with a two-tier architecture are the real killer. One of the strengths of ColdFusion is that it enables developers to easily write reusable and portable code that can be accessed from anywhere in the application. A two-tier application architecture does not play to this strength; in fact, it's only a little better than static Hypertext Markup Language (HTML). I know that you've got dynamic data coming out of the database, but if you want to change the layout of items at the top of the page or change the navigational elements of a page, you've got to touch several code templates to get this done.

What if you need to use the same query on 50 pages throughout your application? You are forced to write that query 50 times and maintain all 50 instance of that query, which can be a pain if the requirements for the query change or if the structure of your database needs to change. We later talk about code reuse in much more detail, but you should understand now that creating a well-planned architectural model enables you to avoid reworking your application down the road.

N-Tier Architectures

Applications that consist of more than two tiers are often called *N*-tier applications. *N*-tier application architectures provide a model for developers to build highly scalable and reusable applications. The *N*-tier architecture focuses on breaking the application into logical segments. By doing this, the developers can support each segment individually and maintain the application in sections instead of recoding the entire system as a result of minor change requests. Do you remember our code example from the typical two-tier application? Contrast what we saw in Listing 3.1 with the code shown in Listing 3.2.

Listing 3.2 ***N*-Tier Application Code Sample**

```
<cfinclude template="common/header.cfm">
<cfinclude template="data/getcustomeremails.cfm">
 <table>
     <tr>
          <td>Customer Name</td>
          <td>Customer Email</td>
     </tr>
     <cfoutput query="getCustomerEmails">
     <tr>
          <td>#getCustomerEmails.CustomerName#</td>
          <td>#getCustomerEmails.CustomerEmail#</td>
     </tr>
     </cfoutput>
</table>
<cfinclude template="common/footer.cfm">
```

Of course, this is just an example of one way that you could accomplish a bit more scalable application, and it's a very narrow example, but you get the picture. You now have reusable header, footer, and ColdFusion query or stored procedure templates that easily can be maintained from one place. This type of organizational structure for your code can help to alleviate numerous problems that can occur within the application-development process. When you start to think about all possible requirements that can go into the development of an enterprise application, you see that the more logically you can organize your code, the better off you are.

It seems only logical that it would be easier to support and maintain presentation code in one area that is separate from the code and that supports the business rules and business logic. Likewise, if all your data access is through stored procedures, you can keep up with that code easier and with the templates that you use to call your stored procedures as well. The *N*-tier architecture enables you to separate support for application components, such as databases, mail, and file servers, into their own logical areas within your application structure and within other servers as well.

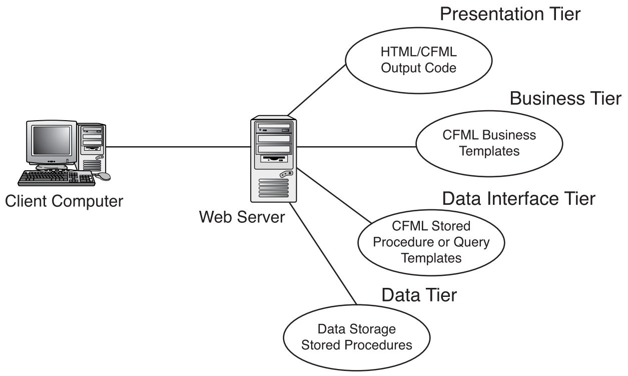


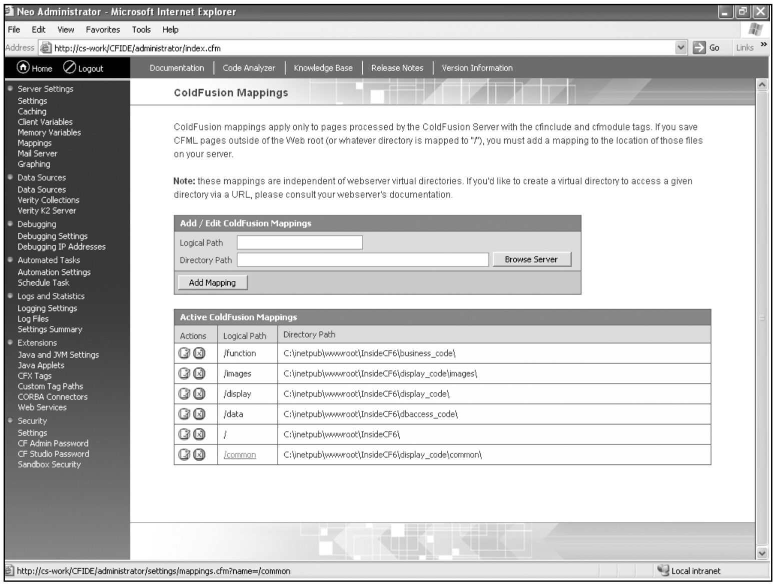
Figure 3.3 *N*-tier application architecture.

In a ColdFusion application, you often access databases that reside on the same server as the ColdFusion Markup Language (CFML) templates. With an *N*-tier architecture, you add an additional layer to your application and can separate the presentation-level code from the data in the database. This means that the code that supports what the user sees is separated from the code that makes up the pieces of business logic that run the application. The business-oriented code also is kept separate from the data that is at the heart of the application.

This type of application architecture makes much more sense and is easier to keep up with if you employ ColdFusion mappings. Take a look at Figure 3.4 to get a feel for ColdFusion mappings.

You might know that by using ColdFusion mappings, you can refer to templates using the `CFINCLUDE` tag by simply referring to that mapping. Note that the mapping is merely an alias for a physical directory to which your server has access. Don't worry if you're not familiar with how ColdFusion mappings are set up or even how they work. We discuss them thoroughly in Chapter 25, "Administering the ColdFusion Server."

For your application to perform specific functions or to conduct certain transactions, the various layers of the application interact with each other. One of the advantages of separating your application into a layered architecture is to minimize the interaction of the client or presentation layer with the backend data. The reason for this is that all your business logic code is wrapped into the business layer and only this layer interacts directly with the data. In OOP approaches, these different layers are sometimes referred to as the interface and the implementation.



ColdFusion Mappings

ColdFusion mappings apply only to pages processed by the ColdFusion Server with the cfinclude and cfmodule tags. If you save CFML pages outside of the Web root (or whatever directory is mapped to "/"), you must add a mapping to the location of those files on your server.

Note: these mappings are independent of webserver virtual directories. If you'd like to create a virtual directory to access a given directory via a URL, please consult your webserver's documentation.

Add / Edit ColdFusion Mappings

Logical Path	
Directory Path	

Active ColdFusion Mappings

Actions	Logical Path	Directory Path
	/function	C:\inetpub\wwwroot\InsideCF6\business_code\
	/images	C:\inetpub\wwwroot\InsideCF6\display_code\images\
	/display	C:\inetpub\wwwroot\InsideCF6\display_code\
	/data	C:\inetpub\wwwroot\InsideCF6\dbaccess_code\
	/	C:\inetpub\wwwroot\InsideCF6\
	/common	C:\inetpub\wwwroot\InsideCF6\display_code\common\

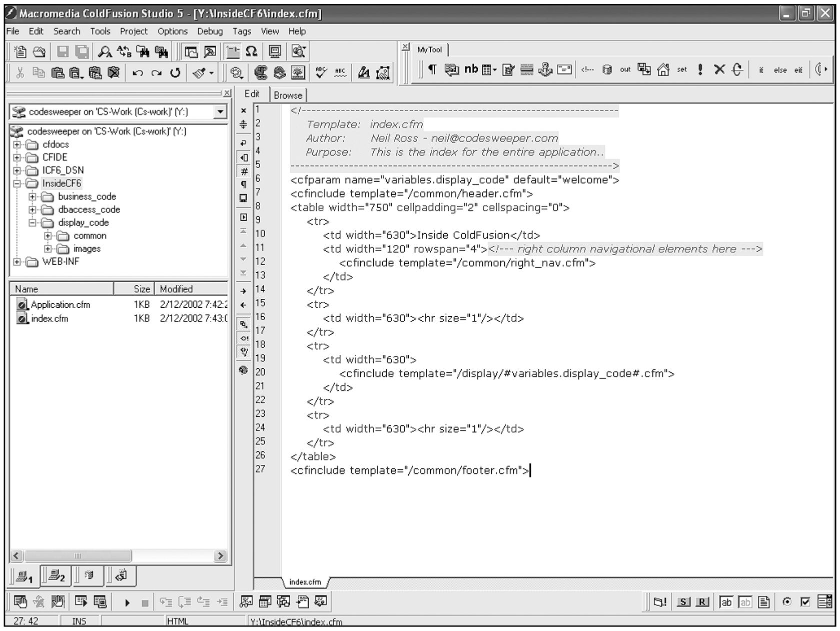
Figure 3.4 ColdFusion MX CFAS mappings.

Application Layers

Developers should take the time to understand the fundamentals of application layers to write well-organized applications that are easy to maintain. In this section, we take a look at how applications can logically be divided into layers, regardless of the physical architecture on which the application is loaded. At its most basic, the *N*-tier application architecture can be divided into the following tiers or layers:

- Presentation
- Business
- Data Interface
- Data

The layers of your application code can be separated by your physical file and directory structure. Figure 3.5 shows the ColdFusion Studio development interface. The directory structure is set up to mimic the division of application layers that we've been discussing.


Macromedia ColdFusion Studio 5 - [Y:\InsideCF6\index.cfm]
File Edit Search Tools Project Options Debug Tags View Help
codesweeper on 'CS-Work (Cs-work)' (Y:)
cfdocs
CFIDE
ICF6_DSN
InsideCF6
business_code
dbaccess_code
display_code
common
images
WEB-INF
Name Size Modified
Application.cfm 1KB 2/12/2002 7:42
index.cfm 1KB 2/12/2002 7:43
Edit Browse
index.cfm
27: 42 INS HTML Y:\InsideCF6\index.cfm

```
<!------------------------------------------------------------------
   Template:  index.cfm
   Author:    Neil Ross - neil@codesweeper.com
   Purpose:   This is the index for the entire application..
------------------------------------------------------------------->
<cfparam name="variables.display_code" default="welcome">
<cfinclude template="/common/header.cfm">
<table width="750" cellpadding="2" cellspacing="0">
   <tr>
      <td width="630">Inside ColdFusion</td>
      <td width="120" rowspan="4"><!--- right column navigational elements here --->
         <cfinclude template="/common/right_nav.cfm">
      </td>
   </tr>
   <tr>
      <td width="630"><hr size="1"/></td>
   </tr>
   <tr>
      <td width="630">
         <cfinclude template="/display/#variables.display_code#.cfm">
      </td>
   </tr>
   <tr>
      <td width="630"><hr size="1"/></td>
   </tr>
</table>
<cfinclude template="/common/footer.cfm">
```

Figure 3.5 Physical directory structure of our application as seen in ColdFusion Studio 5.

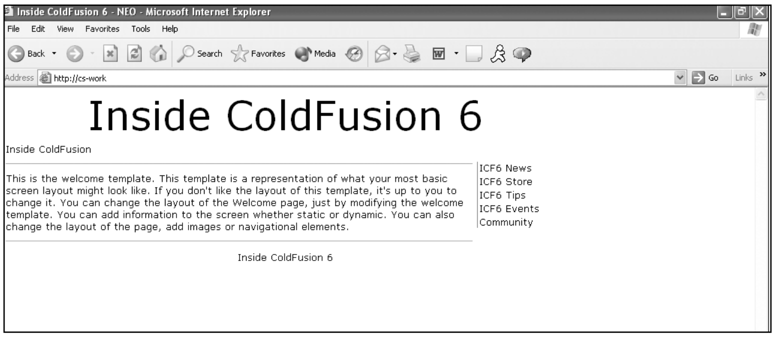
The Presentation Layer

The presentation layer is made up of all elements of the application with which the user sees and interacts. This list of elements can include colors, font faces and styles, images, form controls, navigational elements, and more. These elements together are often referred to as the graphical user interface (GUI) or the user interface (UI). These elements define the look and feel of the application and their consistency and flow can determine the disposition of the user experience.

Users generally interact with ColdFusion applications through the use of a web browser. I say "generally" because ColdFusion applications are client-independent. This means that the ColdFusion Server is not concerned with the presentation-layer method or with the application logic and processing. This enables ColdFusion to output data to different types of clients, including web browsers (HTML), Personal Data Assistants (PDAs) (cHTML, XHTML), cell-phones and other wireless client browsers (WML), and even voice browsers (VoiceXML).

The UI

It seems that a new technology pops up every day to enable designers and developers to produce more feature-rich UIs. In an *N*-tier application, the presentation code for the UI should never access the database directly. Likewise, the UI code should never handle business rules or process logic. It should merely provide for screen layout. Figure 3.6 shows a simple UI.


Inside ColdFusion 6 - NEO - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://cs-work
Go Links

Inside ColdFusion 6

Inside ColdFusion

This is the welcome template. This template is a representation of what your most basic screen layout might look like. If you don't like the layout of this template, it's up to you to change it. You can change the layout of the Welcome page, just by modifying the welcome template. You can add information to the screen whether static or dynamic. You can also change the layout of the page, add images or navigational elements.

ICF6 News
ICF6 Store
ICF6 Tips
ICF6 Events
Community

Inside ColdFusion 6

Figure 3.6 A simple UI for our application.

When discussing UI considerations, again the application architect and designer have to rely on the completeness of the requirements that have been gathered. These requirements should bring into better focus issues such as layout, colors, fonts, screen resolution, plug-ins, scripting, and browser compatibility that are necessary to deliver an application that meets the customer's needs.

Layout

I'm no artist or graphic designer and you might not be either, but paying close attention to layout of the screens throughout your application not only makes your application more aesthetically appealing to the user, but also it enhances the usability of the application itself. Consistency is one of the key elements that you need to remember about layout. You don't want your user hopping from page to page in your application and being forced to search for common elements such as navigation and form controls.

Correctly treating the unused spaces in the browser screen and having a balanced layout that utilizes all areas of the screen are important. Remember that you can say as much with a clean and professional layout as you can with lots of flashing banners, blinking text, and dancing cartoon characters.

Advertising and publishing companies employ experts in layout who provide ads with balanced and proportional layouts that mix typography, images, and text to convey a message. Each element is important to layout.

Online, layout is not quite as cut and dry. We have to worry about users' screen resolution settings and color-quality settings. Many web designers and developers use framed layouts for their sites. Some employ dynamic navigational elements. One thing is clear, however; consistency in your chosen layout method is key to making and keeping users comfortable with your application.

Colors

You should consider colors when planning the look and feel of your application. Colors can calm or can stir up emotions in users. Be careful with colors, however. Many times, designers or developers use too many colors or the wrong colors. This has the undesired effect of pulling the user's attention away from the content, and it distracts their focus.

Color on the web is quite different from color in print media. After the printed item is completed, the color cannot be changed and everyone sees the same color. It does not work this way online, however. There are many limitations to presenting color on the web. Some of the things that you should keep in mind include the following:

- Monitors that display your web site or application have a wide variety of depth settings.
- Color settings can be calibrated differently or use different gamma settings.
- OSs affect the way that color is displayed.
- Web browsers affect the way that color is displayed.

Color should be a big concern to anyone who is developing a UI. You should know that there are certain colors that are considered browser-safe. By working within the limits of those browser-safe colors, you can be more confident that your users are seeing the colors that you intended them to see.

Note

You can download the color-safe browser palette and find more tips regarding colors from Lynda Weinman's web site at `www.lynda.com`.

Screen Resolution

When planning your application, keep in mind the screen resolution settings of your users' video adapters. Most computers are shipped with video settings defaulted to 800×600 pixels. However, newer flat-panel monitors often have a lowest resolution setting of 1024×768.

Many developers forget that the actual usable screen real estate is less than what the resolution setting would lead you to believe. Check out the popular resolution settings and corresponding browser window sizes shown in Table 3.1 to help you keep your layout on target.

Table 3.1 **Screen Resolutions and Browser Window Sizes**

Screen Resolution Setting	Browser Window Size
640×480	600×300
800×600	760×420
1024×768	955×600

It's always a good idea to design your application so that the lowest resolution monitors can view and interact with your application without any formatting problems or excessive scrolling. Don't guess at what your users' resolution settings are. If at all possible, this information should be provided as part of your requirements-gathering process.

Fonts

We communicate user instructions with text on our screens, so, of course, we need to make sure that we pick fonts that are going to be easy for the user to read. The most commonly used fonts are Arial and Times New Roman. However, most web presentations look better with a font from the sans-serif font family. These fonts include Arial, Helvetica, or Verdana. The reason for using these standard fonts is that the user must have the specified font on his or her system, or you must push that font to the user along with your code.

By varying font sizes, styles, and colors, you can convey not only information, but also the importance of that information. Keep in mind that it's not a good idea to use more than one or two fonts in your application. In addition, remaining consistent with font sizes and styles when calling out headings or identifying notes throughout your application is an important factor in maintaining continuity.

Images

As we move through the development of our application, we are adding more and more content and are developing web graphics to support that content. As they say, a picture is worth a thousand words, and if you have the choice of reading all those words or checking out a cool graphic, I'm sure you'd choose the graphic.

There are, however, drawbacks to stuffing your application with graphics and photos. One of the largest drawbacks is the effect that overloading your web pages with references to images can cause. The reason for this is that the user must wait for all images to download before the requested page is fully viewable. Thus, if you're downloading lots of images on every page, you can slow down your page response times.

Another thing to keep in mind when you're planning your graphical presentation and considering photos, graphics, and other graphical elements (such as Flash or Shockwave files) is that the bandwidth of the end user likely is not as fast as your access to your development server (often local).

Client-Side Scripting

Many web applications use client-side scripting languages such as JavaScript and VBScript to enhance the user experience. Developers often utilize client-side scripting for form validation, user alerts and confirmation, and new windows.

VBScript is a scaled-down version of Microsoft's Visual Basic (VB) language. Support for VBScript is built in to Microsoft's Internet Explorer (IE). However, if your user is accessing your application using a Netscape or other web browser, he or she might need to download special plug-ins to support the code.

JavaScript was originally a creation of Netscape. It was called LiveScript, but the name was changed when Netscape and Sun Microsystems struck up a partnership and developed the language jointly. JavaScript shares some of the same structures and syntax with Java, but the languages are totally separate. JavaScript is most often the client-side scripting language of choice for most developers because of its cross-browser compatibility.

Browser Compatibility

Speaking of browser compatibility, developers need to be wary of problems with the compatibility of their code with various browsers. Such compatibility often affects the placement of elements on the screen.

The key to avoiding problems with browser compatibility is to find out what browser and browser version your users are using. Of course, unless your application will be viewed within a closed environment such as an intranet, this task is next to impossible. What you must do is build a profile of your typical user and build the application to support that profile.

Even after determining your typical user profile and building your application, you still have to test for compatibility. In your user's preferred browser and its nearest competitor, test your application for proper placement and alignment of screen elements. In other words, if the typical user's browser is IE 5.5, test with it, but also test with a comparable version of Netscape Navigator.

You also need to be aware that the user might have disabled support for client-side scripting languages such as JavaScript. If the user's browser does not support your code, you should plan ahead for how to handle interacting with that user.

The Business Layer

As we continue to break out the layers or tiers of *N*-tier application architectures, let's pause for a moment to get a general understanding of the business logic layer and the role it plays.

Just as every business is run on operating procedures or business rules, every application operates under a similar set of criteria. These criteria are what we call the business rules of the application. These business rules are strung together in business process logic. These business processes enable developers to create complete transactions that users will be guided through during their interaction with the application.

The business layer should provide us with an area in which we can write and store the code that interprets our business rules. It should also provide for a way to follow the logical flow of the business, or the business process logic.

Business Rules

As stated earlier, every business runs on rules, and business rules in a ColdFusion application define how the application enables users to interact with your data. For example, if a business rule states that the user must provide a valid email address to log in to the application, the application must enforce this rule. One way to enforce such a rule is to enable a user to register by providing his or her email address. We then can generate a password and email it to the email address that the user provided.

Because business rules are defined to help maintain the integrity of your data, the business rules are not defined by the application architecture. Just the opposite is true: The application architecture is defined to help support the business rules, the business processes, and the process logic.

The example that we just mentioned might use one or more specialized pieces of code to make the application adhere to the business rule that the user must provide a valid email address. One piece of code might check whether the email address provided is in the proper format. Because it is up to you how and where you enforce these rules, you might choose to enforce them using client-side scripting (using, for example, JavaScript or VBScript). This can also be accomplished using a user-defined functions (UDF) or regular expression.

Business Process Logic

Compliance with business rules usually means that your application transactions have to follow some sort of defined business-process logic. A process is an action that the user might take that requires multiple steps. Business processes enable users to move through complicated procedures in a way that breaks the process into logical steps. The user continues interacting with the application along the way until the application has gathered all the information that it needs to complete the process.

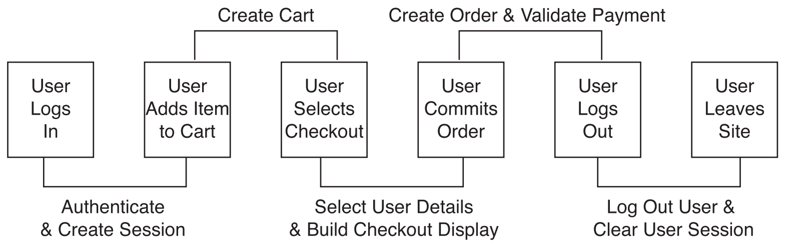

Create Cart
Create Order & Validate Payment
User Logs In
User Adds Item to Cart
User Selects Checkout
User Commits Order
User Logs Out
User Leaves Site
Authenticate & Create Session
Select User Details & Build Checkout Display
Log Out User & Clear User Session

Figure 3.7 Step-by-step process logic.

Each step of a transaction should lead the user step-by-step through the process and should enforce the defined business rules along the way.

For example, many shopping cart applications require new users to register, providing personal information and ordering information along the way. They often require users to not only provide their mailing address, but also to provide billing and shipping addresses as well. A user would be overwhelmed if he or she were presented with all these requirements on the same screen.

A business process would break these requirements into chunks that are easy for the user to deal with and easy for the developer to manage. Not only do you do a service to your users, but also you do a service to your developers.

Data Interface Layer

The data interface layer provides access to the data in your backend database. It does not include any business logic or constraints on how we interact with the database. The data interface layer merely enables us to do things like `SELECT`, `INSERT`, `UPDATE`, and `DELETE` data from our database. The data interface tier should not contain code that defines any business rules or business processes, nor should it contain any code that comprises any presentation-level elements.

In your ColdFusion application, the data interface layer could be made up of templates that contain the `CFSTOREDPROC` calls for the stored procedures that you've written in the data layer. It could also contain `CFQUERY` tag calls that you've separated into functional groups or individual templates.

If you think about it, it makes sense to keep the data requests separate from the code that controls the presentation. The structure of the data itself seldom changes. The presentation-layer code, however, changes more often than anything else in the application. If you keep your `CFQUERY` tag calls in the same template with your presentation code, you run the risk of inadvertently messing up that code. It is better to separate the database calls into separate templates that are accessed much less frequently and that are less prone to programmers' fat fingers.

Data Layer

The data layer is responsible for the storage of your data. This is your database. The data layer is where you can tune and optimize data storage and data access and provide direct access to the data in your backend database. The data layer exists whether you employ a two-tier or *N*-tier architecture with the business logic layer as a buffer to ensure the integrity of your data through the business rules that are enforced.

The data layer is where you create stored procedures to perform queries against the data in the database. These might be very basic functions or might consist of advanced filtering and joining techniques (to be discussed later). Your stored procedures should not care about the business rules of the application, only the input and output parameters that they'll be handling. The business rules serve as a buffer to make sure that the only values that get to your stored procedures are the ones that are valid.

It is within the data layer that you optimize the performance of your database. Performance enhancements such as table indexes enable you to access data faster within the database, which speeds the query process. In turn, this improves the performance to the end user through the presentation layer.

Summary

The architectural model for any application is critical to the application's success. Whether the application is scalable, easy-to-maintain, and easy-to-extend depends on the quality of its architecture. An application's architecture is intended to give a better picture of how the entire system works. It shows how the parts work, and it enables us to see this picture from several different perspectives.

Resource Planning

A major part of the "hard" work of making your application-development efforts successful is ensuring that you've properly planned the environment in which the application is going to live. The importance of resource planning really can't be understated. To use a simple, but familiar, example, imagine that you went to work and that there was no desk for you to use, no lights for you to turn on, and no computer for you to bang out code. You'd find it pretty hard to work, wouldn't you?

Similarly, for your application to function properly, it has to have a properly designed and thought-out working environment.

In many cases, achieving success with a project implementation depends as much (if not more) on resource planning as it does on coding. How involved you are in the initial stages of project planning likely determines the amount of input that you personally have on the type and amount of hardware used. Nonetheless, being as familiar as you can with the "best-case scenarios" makes you an invaluable advisor should you find yourself in that role.

The following section attempts to familiarize you with various processes you can use to properly plan and utilize your resources while tailoring them to suit your specific applications and make the most of your fiscal resources.

Environment Considerations

Obviously, there are many times when you are tasked with application building in situations in which the architecture and environment have already been decided for you. In those cases, the best you can do as a developer is to make sure you write solid code, adhere to best practice standards, and make yourself aware of the subtleties of each specific OS/database combination where that code might be run.

Still, if you're lucky, there are times when your advice as the expert is sought in making the decisions about the platforms with which the application will live. Developers often approach these situations with a "go with what you know attitude." Although there's nothing wrong with that attitude on the surface, you'll find yourself of much more value to the client or customer if you can identify pros and cons of each platform based on the client's specific situation.

As you read on, you should gain a better understanding of how you can approach this "needs analysis" and determine the best choices for any given situation.

OSs

We're hesitant to even begin writing this section because we can hear the screams of each devoted faction ringing in our ears. Again, in many cases, you might play no role in this decision. Still, if you were asked to make a recommendation (and I'm sure you hold your own strong opinions), it would help if you were able to make that recommendation using logical analysis of the situation instead of just your personal devotion to one choice or the other.

When dealing with applications that make use of ColdFusion Server, you have three major choices in the world of OSs. Currently, versions of ColdFusion Server are available for Windows, Solaris, and Linux. Table 3.2 briefly outlines each choice and its benefits and drawbacks.

Table 3.2 **Benefits and Drawbacks of OS/ColdFusion Combinations**

Version	Common Benefits	Common Drawbacks
ColdFusion Windows	Ease of administration; tight integration with Internet Information Server (IIS); hardware components are generally cheaper	Slightly less overall stability; not accepted in some enterprise environments
ColdFusion Solaris	Greater overall stability of the server; often the only choice in environments that have standardized on Solaris	Increased complexity of administration; employees to manage this type of equipment are more expensive than their Windows counterparts; hardware components are generally more expensive

Version	Common Benefits	Common Drawbacks
ColdFusion Linux	The cheapest of all three choices from an OS standpoint, as many versions are free; can be run on the same, less-expensive hardware as Windows versions; can prove to be slightly more stable under certain conditions than running ColdFusion on Windows	Short product history (the first version of ColdFusion for Linux wasn't available until release 4.5); increased complexity of administration

Of course, Table 3.1 makes some broad generalizations regarding the different OS choices, and it's very brief in its discussion of the benefits and drawbacks of each choice. The intent, however, is to get you to think along those same lines as you attempt to identify the best choice for your customer (even if the customer is you!).

If you're a Linux guru already, you've eliminated one of the items I've listed in the drawbacks column. Similarly, if you already have Solaris machines that cannot be retooled to run Windows, you're on your way to a choice there as well. The key is to identify all the benefits and drawbacks of each choice, evaluate where you are and where you need to go, and then make the most informed decision possible.

The differences in stability and performance between the three versions are all pretty negligible, and the choice usually comes down to the hardware and technical resources that you or the customers have in-house.

Choosing a Database

Just as you must decide which OS you want to use for your ColdFusion Servers, you also must decide where your data is going to be stored. Just as with the OS discussion, there are many cases (especially when dealing with legacy applications) where the data is already in a database, thus eliminating the need for you to make this choice.

There are still situations, however, where the application you are building is gathering or creating the data. In those cases, it becomes necessary to decide which database you're going to use and how you want to implement it.

In general, there are two types of database systems: desktop database tools and server databases. Table 3.3 illustrates the differences between the two.

Table 3.3 **Desktop Versus Server Databases**

Characteristics	**Desktop Database**	**Server Database**
Processing of changes to the data	Occurs on the client machine, regardless of where the source file is located.	Occurs at the server, where all the hard work is done.
Availability of data	Originally designed to be used by one user at a time.	Designed with multiple simultaneous requests for data in mind.
Reliability of data	With multiple users accessing what was originally intended for one user, data integrity failures can occur.	Data stored in a database built to utilize the client-server model is extremely reliable even while being accessed and manipulated by multiple users simultaneously.
Network traffic	As most processing of the data is done client-side, the entire package must then be returned to the source machine, causing a tremendous amount of unnecessary network traffic.	Typically, the only things sent over the network in the client-server model are instructions to the database server and result sets, resulting in much less overall network traffic.

If you're serious about application performance, and you want your site to be highly available under all types of traffic load, you should ensure that you're not using a file-based database. Desktop databases were never really meant to be web-connected, and it shows.

Aside from being less reliable than their server counterparts, desktop databases are much, much slower and less secure. Because a large part of any ColdFusion application's processing time is spent waiting for result sets to return from the database, this can have a huge performance impact on your application as a whole.

You need to take a close look at the resources that are available to manage the data after it's there; in addition, make sure that you're not trying to leverage an Oracle database administrator (DBA) for Sybase work, or vice-versa. Believe it or not, the subtleties between different database management systems are often more complex than those between OSs. This makes it all the more critical to know who is available to manage the data after it's in the database.

Contrary to popular opinions, designing table layouts in SQL or Oracle doesn't make you a DBA. As you're application begins to become more heavily trafficked, you inevitably discover that you have to do some tweaking at the database level to keep things running smoothly. At this point, the DBMS you've chosen and the folks you have around you to help you support it become extremely important to you—so choose carefully.

Assessing the Needs of Your Application

What a wonderful world it would be if we were able to peer into our crystal balls and determine, well in advance, just what type of resource configuration our applications would require to function at their very best. As it is, however, we are forced to try to deliver our best estimates.

During the discovery phase of any section of a project, the first thing you usually want to do is a needs analysis. Determining an appropriate application-environment configuration is no different. Based on the requirements you've already received, as well as how you believe that the application should work in its finished state, you should be able to determine roughly how you need to configure your environment to suit your application.

Of course, to determine the actual hardware requirements of the application servers themselves, you'll want to conduct systematic load testing of the finished code. This is an absolutely necessary step and one that is covered in great detail in Chapter 26, "Performance Optimization and Scalability Planning." For now, we deal with planning the environment as a whole to suit your application.

First and foremost, you need to determine the overall architecture that you plan to implement. Depending on the needs of your application, this is where you should begin to investigate DMZs, firewalls, Network Address Translation (NAT) and Secure Socket Layer (SSL) hardware/software combinations, and so on.

Ideally, you want to expose as little of your infrastructure as possible to the outside world. The most traditional method of accomplishing this is by placing your public servers (such as the web server) on a protected subnetwork that has no access to your internal network. In this way, you can keep the public portion of your network accessible, without introducing undue security risks.

Further, you might want to make use of a firewall of some sort to restrict traffic between all computers on your network in accordance with the security policies that you or your network administrators have defined. The firewall can become an integral part of your overall design; it acts as the traffic cop, pushing traffic meant for your public servers to your predefined subnetwork and keeping unwanted intrusion from your internal systems. Figure 3.8 demonstrates this design.

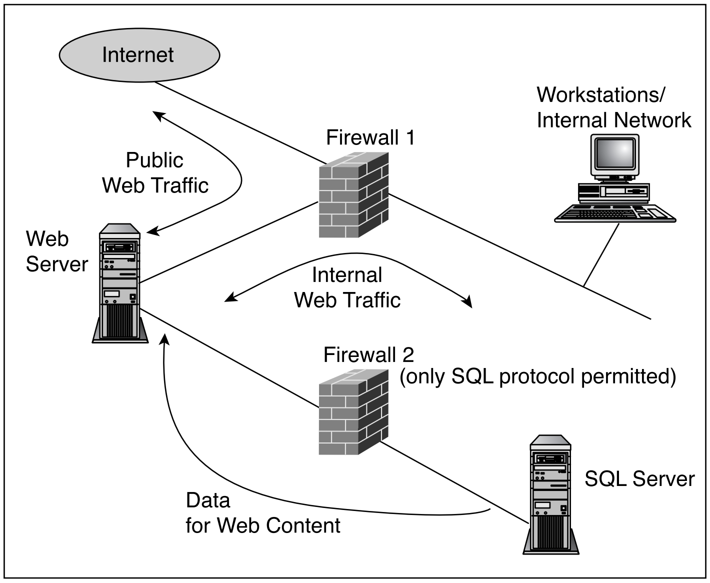


Figure 3.8 Example architecture.

After you've determined the overall design of the environment in which your application will live, you should address (as much as is within your power as the developer) the security of the public servers themselves.

Depending on the combination of OSs and web server software you have chosen, the actual methods for securing these servers differ. Still, the key things that you want to check are as follows:

- Access to public content on the servers is read-only for web users.
- The user designated as a pathway user for web clients (such as the IUSR_LocalMachine account in Windows) has only the bare minimum permissions required to gain access to material you want to make public.
- Configure your servers to limit vulnerability from denial-of-service attacks by using appropriate monitoring and filtering software, along with configuring appropriate timeouts.
- Restrict the web server itself to only server files designated within your defined web content tree.
- Disable the capability to obtain directory listings. In this way, you can help ensure that only the content you explicitly designate as public material is seen.

Of course, this is by no means a complete list, but it should help you begin to look in the right direction in thinking about securing your servers. If security is a major concern for you and your organization, there are many consultancies now offering full security audits of your architecture.

Another part of your initial security strategy should be checking vendor sites to make sure that you've installed and configured all necessary security patches. Most vendors maintain sites explicitly designed to keep their customers updated with recent security releases. After you've decided which software products you're going to use, it's a good idea to visit the vendor sites for these products and sign up for the various security alert mailing lists that are offered. These mailings typically update you as new security issues are discovered, and they tell you the actions that you need to take to protect your configuration.

Depending on the type of application that you are designing, there might be other pieces of your configuration that you need to snap in; but in almost every case, you have to start by defining a configuration and security model. After you've done the work up front, you can move on to the actual design and implementation of your application with a good understanding of the policies and limitations behind the scenes.

Planning Ahead for Scalability

When you're designing your architecture, leave yourself a path to upgrade should your application suddenly get bombarded by traffic that is much higher than you expected. When you are initially designing the architecture, you are forced to make a best-guess estimate of the amount of traffic your site will actually receive. Although this is fine as a starting point, you might find that as you gain a larger user base, you need to reevaluate how your environment is designed and expand your architecture to better suit the needs of your growing user base.

One of the easiest ways to do this is through clustering and load balancing. Clustering and load balancing enable you to have multiple servers in your environment that serve exactly the same purpose as their twins, thus splitting the work between two or more partners.

In a typical architecture, there are three places that you can cluster: at the web server, at the application server, and at the database server.

Unless you've implemented a three-tiered architecture with your application server physically separated from your web server, you likely will be performing clustering at the web/application server or at the database server.

To cluster, you need to identify where your traffic bottlenecks exist. By performing analysis on your traffic patterns and monitoring the resource usage on your application and database machines, you should be able to determine where you get the most bang for your buck.

Obviously, if your analysis shows you that your ColdFusion Servers are spending the vast majority of their time in an idle state and your database server is nearly never idle, you should consider clustering your database to spread the load.

Conversely, if the database is idle nearly 100 percent of the time and your ColdFusion Server's resources are strained, you should cluster at the ColdFusion layer.

Depending on the database that you've chosen, you should be able to find a wealth of information on your clustering options at the vendor site.

If you've determined that you need to cluster at the application-server layer, the first step in analyzing your traffic pattern is to define how many servers are necessary to service all your concurrent requests during peak load times.

Clustering ColdFusion Servers in this sense refer to having a group of web/application servers work together to service the entire site. When clustering in this way, each member of the cluster typically hosts a complete copy of the entire site so that any incoming request can be answered by any node on the cluster.

Alternatively, you might choose to centralize all web content in a single location, giving all cluster members access to this content. The content is made available on a separate physical machine on the network, typically referred to as a content or file server. The main problem with this type of configuration is that although your application servers are still clustered, the content server represents a single point of failure in that if it becomes unavailable, none of the clustered application/web servers can service any incoming requests.

One way to solve this problem is by partnering the content server and the database server, and then making each one the other's failover. In other words, although the database server's primary function in this setup is to handle incoming database requests, it would also contain copies of the content so that if the content server were to fail, the cluster nodes could look to the database server for the content. Subsequently, should the database server go offline, the content server will have a complete copy of the database that can come online to answer database requests in the event of a failure. This type of setup eliminates the need for redundant hardware, but continues to ensure high availability of all components in the cluster. Figure 3.9 demonstrates this model.

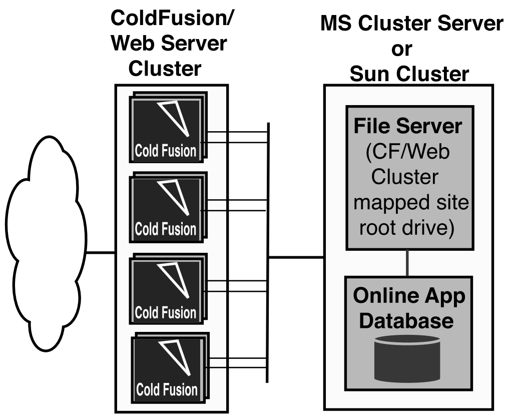


Figure 3.9 High-availability model.

Load balancing is the next major key in scalability planning. After you've decided to cluster your application servers, you need to come up with a plan for the distribution of incoming traffic between each member of the cluster.

There are many ways that you can do this, but by far the most popular choice today is through the introduction of a hardware layer. This hardware layer, typically called a load balancer, sits on the network in front of all your web-application servers. When any requests come in for your site, the load balancer answers these requests by deciding to which server on the cluster it wants to direct this traffic. The way in which it makes that decision can be controlled by the way in which you configure the load balancer itself.

Typically, if your site is located at `www.somesite.com`, `www.somesite.com` resolves to the Internet Protocol (IP) address of your load balancer. After traffic hits the load balancer, the load balancer can decide to which member of the server cluster it should translate that request. This can also give you an additional layer of security because, in this model, the load balancer itself really is the only piece of hardware that needs to have a public Internet address. After traffic has been sent there, it can usually translate the request to a private network address of a machine on the cluster for response.

After you've decided that you are going to need to cluster and implement some sort of load-balancing strategy, the next step is defining how many cluster members you actually need to handle the amount of traffic you are experiencing during peak times.

There are many different strategies and methodologies available for determining this number. We will present one that we've used successfully in the past.

First, it helps to have a general understanding of how cluster size is determined. Generally, there are a few standard factors that you need to consider:

- Expected peak requests per hour
- Expected peak simultaneous requests
- Maximum average response times that you feel are acceptable for your user base
- Specific characteristics of your application (database versus content-intensive, state management needing to maintained or not, and so on)
- Environmental characteristics (locations of firewalls, private networks, routers, and so on)

After all items have been considered, testing is required to determine the exact number of cluster members necessary to meet the expectations that you have defined.

As with any other methodology, there are many testing methods available. The one we've used successfully in the past is outlined here:

1. Determine what you think is the maximum number of requests per hour experienced by the site. You can use past data obtained from web access logs or other monitoring utilities to give you some idea of the number with which you might want to start. In addition, you need to define what you have decided is the maximum tolerable response time for your site.

   This is an important factor because it is used to determine exactly how many cluster members you need to achieve this response time.

   In addition, when you're determining your maximum tolerable response time, be sure to keep in mind that you're defining this as the "maximum allowable response time during peak periods of load."

   Although it might be desirable for you to say, "I don't want any page to take more than three seconds to return to the user," this would be a very unrealistic response time to expect under peak periods of load. The lower your maximum allowable response time, the more cluster members you likely need to keep that time down.
2. Set up a controlled test (using an enterprise-load testing tool) that stresses the site with the amount of load that you've defined in Step 1. Controlled load testing is a science all its own, and if you've never been exposed to it before, you might want to refer to Chapter 26, for more information on how to go about load testing a site in this manner.

3. Calculate the average response time of your site while you are placing it under load from the load-testing tool.
4. If the response time you saw in Step 3 is greater than the maximum tolerable response time that you defined in Step 1, you need to add another member to the cluster and start the process again at Step 2. Repeat this process until the response times you see under peak load during your testing fall within the allowable range that you defined in Step 1.

Again, you might find that you have a better way to determine your cluster needs. There is nothing wrong with developing your own methodology for determining these needs—just make sure that whatever method you use, you have some quantifiable way to measure the improvement in performance as you add members to the cluster.

Summary

In this section, we examined how to define the environment in which our application runs, as well as how to determine the specific needs of our application from an architectural standpoint. We also talked about how to expand our environment for scalability after we determine the need.

To successfully build a web application with any technology and not just ColdFusion, you have to make sure that you are properly planning and using your resources.

Making proper use of the resources you have means that you're keeping track of what they're doing and how they are doing it. This keeps everything running smoothly. The job of monitoring all components of your setup is an ongoing task throughout the life of your application, and it's a necessary step in making sure that you can identify and deal with any potential problems early on.

In addition to the number of servers you have available to you and the way that your network is architected, you also need to examine the type of hardware that you have to make sure that it's up-to-date and appropriate for the task.

You really can't over-plan your resources. It's better to have too much hardware available to you than to come up short after it's too late.

There's an old adage that the prepared companies determine the maximum amount of hardware they need for peak load and then double it just to be safe. Although that's unrealistic to expect, the core message is true. It's better to be prepared for the worst than it is to get left wishing you had.

Depending on how involved you are in the predevelopment stages of the project, you might find that most or all architecture and environment planning work has been done for you. Nevertheless, as a developer, it's important that you keep yourself well-versed in the different ways in which an application environment is planned and deployed so that you can offer expert advice based on your experiences when you are inevitably called upon to do so.

After you've completed the planning of your architecture and hardware, you then can begin thinking about how you're actually going to build the core application. Just as the architecture and hardware layer requires proper planning to ensure success, so too does the application-building process.

In the next section, we dive right into one of the first steps of this planning by beginning a discussion of the various development methodologies that are available for you to use when designing your application.

Development Methodologies

Development methodology in relation to ColdFusion application design refers to the existence of a defined set of conventions or procedures to guide you through a development project. A development methodology might define your application's physical file structure and might dictate what type of template goes into what folder. It might define naming conventions for your code templates and how those templates interact with each other.

There are many popular development methodologies floating around the ColdFusion development community. Any one of them might meet your needs and help you to design more scalable and easier-to-maintain applications. Understanding the approach of these development methodologies might even help you to define your own practice.

It is not our intention here to explain all intricacies of any development methodology. We want to help you understand the use and importance of utilizing a development methodology in your projects.

Fusebox

One of the most popular, widely known, and widely accepted development methodologies is Fusebox. Fusebox has successfully been adapted to serve as an application-development methodology for Active Server Pages (ASP), PHP, and Java Server Pages (JSP) applications. This discussion focuses on Fusebox 3, which is the latest specification. Although some of the fundamental concepts are the same, Fusebox 3 bears little resemblance to Fusebox 2. Fusebox 3 is built around the following key features:

- A nested model that supports communication between circuits. Circuits now can have a parent/child relationship, leveraging the power of inheritance. This is a departure from Fusebox 2, where circuits were independent of each other.
- A nested layout model
- Fusedocs, which provide a Program Definition Language and documentation in an eXtensible Markup Language (XML) format
- A defined set of key or core files
- Exit fuseactions (XFAs)
- A public application program interface (API)

Core Files

Fusebox 3 introduces different core files, each prefixed with FBX_:

- **`FBX_Fusebox_CFxx.cfm`.** The *xx* is dependent on the version of ColdFusion supported. Your Fusebox application has a separate file for each version of ColdFusion supported. The `FBX_Fusebox_CF`*`xx`*`.cfm` file replaces many of the custom tags from Fusebox 2 and should be called by the default file of your application's home circuit.
- **`FBX_Settings.cfm`.** This is an option file that is used to set variables. The job of the FBX_Settings.cfm file is to set up the environment in which the application runs. This replaces app_Globals.cfm, and variables specific to a circuit application are set in the app_Locals.cfm files used in Fusebox 2.
- **`FBX_Circuits.cfm`.** This is required in the home circuit and provides mappings of circuit aliases to physical directory paths. The circuit aliases are not required to be the same as the directory names.
- **`FBX_Switch.cfm`.** This is a `CFSWITCH` statement that contains a `CFCASE` for every fuseaction that the circuit handles.
- **`FBX_Layouts.cfm`.** This is an optional file, used to set the variable Fusebox.layoutDir, which points to the directory where layout files are kept. It also sets Fusebox.layoutFiles, which points to the layout file to be used.

Fusedocs

Unlike Fusebox 2, which used a proprietary format developed by Hal Helms, Fusebox 3 uses XML to document what a fuse does and the required input/output. The fusedoc has three elements. Responsibilities is the first and it is required.

The other two elements are properties and io. Responsibilities provide an explanation of what the fuse will do. Properties contain a number of subelements that provide information, such as history, related to the fuse. The last element, io, defines the input and output of the fuse. A more detailed explanation of these elements and their attributes can be found in the Fusebox 3 documentation.

Fusebox Basics

Fusebox helps developers build robust and scalable web applications easily, surely, and quickly—and it's surprisingly simple. A Fusebox application works by responding to requests to do something—a fuseaction, in Fusebox parlance. This request might come about as a result of a user action (a user submitting a form, for example, or clicking a link), or it might occur as a result of a system request.

When you submit the form, `index.cfm` is called. In fact, everything an application can do is done by sending a fuseaction request to `index.cfm`. This file, so central to the methodology, is called the fusebox. When a fusebox is called, a variable called fuseaction is also sent.

The fusebox's main job is to route a fuseaction request to one or more code files called fuses. These files are typically small and have well-defined roles.

The routing begins with a `CFSWITCH` statement, located in the `FBX_Switch.cfm`, that examines the value of the fuseaction. After it finds a matching value in one of the `CFCASE` statements, the code between those particular `CFCASE` tags is executed. In the Fusebox methodology, the `CFCASE` code is used to set up an environment in which one or more fuses can be called to perform whatever actions are needed to do the work requested by the variable fuseaction.

A fuse can be used to display a form, check whether a user's password and username match those found in a database, and show a menu of user options. In short, anything a web application can do can be done through the use of fuses.

Well-written fuses are very short and restrict themselves to doing only one or two things. They are easier to write (and maintain), are less buggy, and are easier to debug. They also facilitate code reuse. You seldom need a catch-all type of fuse, but you often will need fuses that handle a specific task.

Fuses have one or more *exit points*, which are areas where the action returns to the fusebox. Every link on a user menu is an exit point, just as each drill-down action to get more information is an exit point. Some exit points are visible and require user interaction, such as submitting a form or clicking a link. Other exit points are not visible. An example of an invisible exit point is a call to a `CFLOCATION` tag within a user-authentication template. Whether generated by user interaction or by the application itself, exit points in fuses always return to the fusebox with a fuseaction.

This is a great place to discuss the feature of XFAs. XFAs are the exit points of a fuse. In Fusebox 2, fuseactions were hard-coded into exit points:

```
<form action="index.cfm?fuseaction=verifyUser">
```

However, this impairs code reusability. If you want to reuse a fuse—whether in another application or in a different point in the same one—you now must deal with the fact that your exit fuseactions might not be the same. They might vary according to the context in which they are used. This means that you must begin introducing conditional statements in your code. Now, each time you want to reuse the fuse, you must open up the file, alter the existing code, and then save it. This introduces the very real possibility that you will introduce a bug into the code. Further, it reduces readability (and maintainability) of the code, making it just plain ugly.

Fusebox 3 enables you to use XFAs to replace the hard-coded references. This lets you create fuses that can be used anywhere, without worrying about where a form should be submitted or a link should be pointed.

```
<form action="index.cfm?fuseaction=#XFA.submitForm">
```

The value of the fuseaction is set in the `FXB_Switch.cfm` file:

```
<cfswitch expression="#Fusebox.fuseaction#">
        <cfcase value="verifyUser">
                <cfset XFA.submitForm="verifyUser">
                <cfinclude template="qry_verifyLogin">
        </cfcase>
        <cfcase ...
</cfswitch>
```

Fusebox Conventions

It's helpful to categorize fuses into different types. For example, some fuses display information, forms, and so on to users; others work behind the scenes to do things such as process credit cards; and still others are responsible for querying databases. Many Fusebox developers find it helpful to use a prefix when naming a fuse. Such a prefix conveys the fuse type. Examples of these are shown in Table 3.4.

Table 3.4 **Fusebox 3 Prefixes**

Fuse Name with Prefix	Explanation
`dsp_ShowProductInfo.cfm`	A display type fuse used to show or request information from a user
`act_SaveUserInfo.cfm`	An action type fuse used to perform an action without displaying information to a user
`qry_GetUserInfo.cfm`	A query type fuse used to interact with data-sources without displaying information to a user
`url_ProcessOrder.cfm`	An action type fuse used to redirect an action

These naming conventions are best thought of as suggestions. You can use the naming conventions outlined here—or not, depending on what suits you best.

If you do use them, will you use an underscore to separate the prefix from the fusename, or will you use the mixed-case spelling that many developers prefer? Personally, I like the mixed-case usage, but you might prefer something else. Whatever you choose, don't get bogged down in disputes over naming conventions; Fusebox is primarily about developing successful applications, not about naming schemes.

Encapsulation

You have seen how fuseactions are returned to the fusebox, and you have had a glimpse of the mechanism the fusebox uses to call helper fuses (a `CFSWITCH` statement). Fuses really do all the work in a Fusebox application. The fusebox itself acts like a manager, delegating work to one or more of these fuses.

One of the principles of modern programming is encapsulation, which states that, as much as is possible and reasonable, applications should be divided into areas of related functionality. We do this constantly in many aspects of our lives.

One of the great things about Fusebox is the flexibility it gives you as the developer. This not only lets you decide on things such as naming conventions, but also it enables Fusebox developers to experiment with new ideas without fear of running afoul of any Fusebox police. This, in turn, lets Fusebox evolve, whether to stay abreast of technology developments or simply to incorporate good ideas that weren't originally envisioned. Different developers have different goals and bring with them different techniques. Such creative diversity can only be good for the methodology.

One idea that has proven very successful in the object-oriented world is that of inheritance, a mechanism for reusing code (and surely the Holy Grail of many programmers). Having written a perfectly good circuit application, such as a user module, for one application, we surely want to use it in others and to do so without having to make wholesale changes to the code.

The original Fusebox specification doesn't make this particularly easy, so you had to muck about, changing `app_Locals.cfm` and `app_Globals.cfm`. The idea of inheritance made this process both easier and safer. Many developers welcomed the idea of nesting in Fusebox 3, which was a departure from the Fusebox 2 concept of treating each fuse or circuit as independent. Although Fusebox 3 supports a concept of inheritance that enables child circuits to be inherited from their parent circuit, circuits should have the same properties or functionalities associated with inheritance in object-oriented languages.

Summarizing Fusebox

Although we could spend all day delving deeper into the Fusebox methodology, let's wrap this discussion up by recapping what we've just discussed.

Fusebox is an application-development methodology that enables developers to create highly reusable and scalable applications by providing a framework for the application. This framework employs the use of a template that is central to the application that serves as a fusebox. It routes requests, calls includes, and sets variables.

Fusebox enables applications to be clustered or nested off the main fusebox to ensure that your application can be easily extended and enhanced. Fusebox enables developers to write code that is reusable and easy to maintain.

To learn more about Fusebox 3, visit `www.fusebox.org/`. This site contains the latest Fusebox specification and a number of excellent articles, presentations, and tutorials on Fusebox.

cfObjects

cfObjects is another popular development methodology available to ColdFusion developers. cfObjects purports to be a simple and efficient framework for building applications that can take advantage of object-oriented programming principles, including inheritance, polymorphism, and encapsulation.

Object-oriented programming enables us to better model the problem domains within our systems and designs. With an OOP methodology and framework, we can tackle very complex applications using an approach that is natural because it is similar to the way a human solves daily problems.

By combining the power and flexibility of ColdFusion with an object-oriented methodology, ColdFusion developers gain the capability to develop, share, and extend reusable class hierarchies that solve the common problems that occur in web-application design.

By making use of ColdFusion features such as exception handling, verity collections, and the request scope, you can implement OOP constructs and class libraries.

cfObjects is a collection of highly specialized custom tags, Visual Tag Markup Language (VTML) files, and Studio Wizards. At the core of the framework are different custom tags:

- `CREATEOBJECT`
- `INVOKEMETHOD`
- `DECLAREMETHOD`
- `COLLECTGARBAGE`
- `CACHECLASS`
- `DUMPALL`

These custom tags enable developers to create object-oriented class libraries or object collections. The value or the framework is in the depth of the class libraries, not the framework components. cfObjects was designed to enable any company, organization, or individual to create specialized class libraries that plug into the framework. That is, cfObjects is not a class library—it is a framework for implementing and using class libraries.

Classes

Classes are implemented as a collection of ColdFusion templates residing under a class subdirectory (one per class). The class subdirectory must exist under a directory named by a ColdFusion mapping. Within each class subdirectory, there must exist a special file named `class.cfm`. This file defines the class and superclass name. `Class.cfm` is loaded by `CREATEOBJECT`.

Methods

Methods are implemented as specialized custom tags that exist within the class subdirectory. Each method should contain a call to `DeclareMethod` as the first line within the CFML file.

When a method is invoked, the cfObjects framework makes all passed attributes available to the method, including the special attribute named "self," which is a reference to the object instance. This is useful for accessing the instance variables of the object.

You can download the framework for cfObjects along with all the custom tags. Remember, however, that cfObjects is not a product. It provides a framework and a development methodology for ColdFusion developers to follow. You can learn more about cfObjects at `www.cfobjects.com`.

SmartObjects

SmartObjects is a freely available, open-source framework that enables developers to convert a directory of CFML templates into a customizable, reusable, object-oriented component, or class. This is done by placing a class definition file, called `public.cfm`, in the directory. This template uses the `CF_CLASS` custom tag to define the directory as a class, and it makes it available to be used by other applications.

After it is defined, SmartObjects classes are not accessed directly through a web browser, but are called by—and embedded in—other templates. This is called creating an instance of, or instantiating, the class using the `CF_OBJECT` custom tag. Each instance of the class is called an object, and it is represented by a structure variable type.

Every CFML file in the class directory is called a method, and it represents any executable function such as add, edit, delete, or find. The calling application uses the `CF_CALL` custom tag to execute any of the methods that the object supports.

Classes can inherit methods from other classes, which means that they do not have to define all their own functions by themselves. Base classes provide functions that subclasses can override or extend. Using inheritance, you can simplify your applications by maximizing the amount of reusable code that you can access. You can add new methods to a class, or you can replace existing methods with new functionality. You can learn more about SmartObjects at `www.smart-objects.com`.

Other Established Methodologies

There's no one methodology that we can recommend that you use. We do think, however, that understanding several development methodologies helps you to better understand the application-development process. I'm sure no one has thought of every possible complication that can arise in application development. Let's take a look at a few more development methodologies.

Switch_box

Switch_box is a methodology that has definitely been influenced by other popular development methodologies. The use of the word "box" in "Switch_box" is really more coincidental to "box" in "BlackBox" or "Fusebox" than a variation on these techniques. In fact, some of the conventions of the Fusebox methodology are used in Switch_box.

The idea of classification and organization of information is as old as the human mind. It is how we understand ideas and hold on to them. It is intrinsic to our human nature. Thus, the idea for boxes and nesting of boxes is a really great way for our minds to understand the problem paradigm.

Switch_box introduced the notion of a message vector as means for directing program execution from a uniform resource locator (URL). It builds on the traditional dotted-object syntax with two important distinctions. First, it is designed to be a Hypertext Transfer Protocol (HTTP) attribute. Second, it is intended to show the clear separation of a method name from the object name.

The word "vector" means direction, and directions are calculated from coordinates. In the case of Switch_box, the axes for coordinates are the object list and the method list.

In traditional object-oriented syntax, sometimes discerning the method from the object can be a guessing game and requires beforehand-knowledge of objects and their methods. In Switch_box, the objective was to make the separation clear and unambiguous. This is the reason for putting the colon between the object tree and method tree. In Switch_box, the rule for a message vector is that any message without a colon is a method for the current box; otherwise, all message vectors must be properly formed with the object tree and method tree. In addition, in the method axis, Switch_box enables the use of a compound method tree. This feature is necessary for handling predicate noun actions and makes the process of writing program code more efficient.

Along with a message vector, the notion of a "switch operator" has also been introduced. The switch operator is merely a means to calculate the message vector path. The standard object-oriented example Books.Catalog.Adults.Inventory.Display would be in the message vector format of Books.Catalog.Adults.Inventory:Display would read as "Books switch Catalog switch Adults switch Inventory switch Display." The `CF_SWITCHBOX` custom tag handles the message-vector switch operators. To find out more about this methodology, check out Switch_box online at `www.switch-box.org`.

BlackBox

The BlackBox style of developing ColdFusion applications is based on the premise that developers like to have control over their pages. Its strengths enable developers to access the brains of an application without worrying about the predefined presentation tier.

BlackBox enables the same functionality to be used in several sections of the application, but in very different ways. It is designed so that employing this functionality creates the illusion of functions. It enables developers to easily nest applications and to create attractive URLs. BlackBox also enables developers to easily integrate multiple applications within the same web site.

BlackBox is a very simple methodology. It employs a few custom tags and functions to access individual templates and to access integrated applications. You can find out more about the BlackBox methodology by visiting `www.black-box.org`.

Developing a Personal Methodology

In the last several pages, we've looked at several development methodologies. Some of them are popular, well-publicized, and quite successful; others continue to struggle to gain acceptance in the ColdFusion development community.

I'm not going to stand up and tell you to use one or the other. I'm not going to try to point out the strongest attributes of one or the cracks in others. What I will tell you is that having a development methodology is important and that creating an established set of development guidelines helps you to maintain a minimum level of standardization in your application code.

Here are a few things to consider when you start to develop your own ideas about development methodology:

- **Coding standards.** Every application needs to have some coding standards in place. Coding standards include everything from tag case to the proper scoping of variables. Presentation-tier code calls for the combination of HTML and CFML. The presentation of your HTML and CFML code should be standardized so that it is easy for one developer to read another developer's code or to extend the application for any given point.
- **Code commenting.** I know that this could be wrapped up in a bit of information regarding coding standards. However, I'm isolating this topic because of its importance. Too many times I've seen developers write code that makes total sense to them but to anyone else, it's like reading Klingon.

Commenting should be part of your standard template. A usage comment should be included at the top of the template to provide information on the purpose of the template, its creation date, and its author. At the bottom of the template, I like to include a revision log so that developers understand the iterations that the template and the embedded presentation or functionality have gone through and the reasons that the code was added or removed.

- **Naming conventions.** The names that you give to files, links, and processes are as important as the accuracy and organization of the code that you write. It is important to employ a naming convention in your application that is easy to understand. The naming convention should help to avoid confusion in relating the process call, the template name, or the link to their functions or purposes within the application.
- **Application framework.** Of course, ColdFusion developers can take advantage of the application framework provided by the `Application.cfm` and `OnRequestEnd.cfm` templates. Remember to use these templates properly and not to call query code or employ presentation elements within these templates. They are best used for the evaluation and creation of global variables and application-level variable.
- **Business logic calls.** Business logic calls can usually be divided into functions and actions. Think of functions as specific tasks that the application processes perform. You can use a business logic template to handle the variable values that are created in a multistep process in a particular application transaction.
- **Data interface transactions.** The handling of data interface transaction should have some type of organized approach. By data interface transactions, we are referring to the templates that invoke the calls to stored procedures or that make query calls. By defining each template by its purpose and function within the application, developers can easily find and update code. This area also gives a perfect location to employ commit and rollback strategies and query caching strategies.
- **Variable standardization.** Web-based applications handle variables by passing them from template to template along the URL string or by passing them from form template to action template. The standardization of variable scopes is one of the issues that arises in any application and that gets more and more difficult to manage as your application gets more complex.

There are several custom tags out there that can help with this issue. They simplify variable testing and evaluation by converting all variables to the same scope. You can convert them to a local scope, but I suggest the request scope to avoid any conflicts with variables of the same name that are created within the template that calls the standardization implementation.

- **Exception and error handling.** ColdFusion give us the power to use its built-in, error-handling features. We have the capability to use tags, such as `CFERROR`, `CFTRY`, `CFCATCH`, `CFTHROW`, and `CFRETHROW`. These are error-handling methods that we can employ within our code. We can also take advantage of ColdFusion's site-wide error handling. These types of errors are things that happen while the template is being processed by ColdFusion. They are pretty predictable and enable us to define custom error-handling templates for a more user-friendly, error-message display.

  Exceptions might occur within our application at runtime. Validation and client-side scripting should take care of most of these, if not all. On the off chance that you've left out some validation or some validation does not work properly, you should have a catch-all in place to handle situations where the user supplies invalid values to the application.

Well, hopefully this gives you a good starting point and a heads-up on the things you need to consider when planning your application-development methodology. As we mentioned earlier, it helps to study the existing and established methodologies so that you can glean their strengths from them and adapt them to your needs. You also should make sure to learn from the mistakes that other developers have made in the past. Many of the existing methodologies have been refined as a result of those mistakes and are much more thorough today.

Summary

Understanding your application requirements is crucial to the planning of the infrastructure on which your application runs and to the development of an architectural plan for your application. The definition of a development methodology for your project is also important. We've seen how a methodology can simplify the interaction between logical chunks of code and can make the reuse of code easier to implement . We've also discussed a few established methodologies and some that are on the brink. Whether you choose one of these methodologies or come up with one of your own making, remember these tips:

- A methodology should serve as a framework for the development of your application.

- Your methodology should be well-defined.
- Your methodology should be well-documented.
- Your methodology should be adhered to throughout your application.
- Your methodology should be understood and employed by all developers on your project.

Summary

With a better understanding of application planning, application architecture, and development methodologies, you're ready to get into the code. The next several chapters walk you through some basic and not-so-basic aspects of CFML. Have fun!