

Chapter 1: The Arrival of Fusebox

"...doctrine is not the point of arrival but is, on the contrary, the point of departure..."
—Joseph Brodsky

Long hidden in the world of academia, the World Wide Web began to attract the attention of businesses in the early 1990s. Initially unsure of how to best utilize the power of "infinite communication," companies were finally realizing the potential of the web. At the same time, emerging technologies were making it possible to create increasingly capable sites.

However, along with new desires and tools came new problems, and many well-intentioned development efforts were mired in problems inherent to working on the web. Programmers who were familiar with traditional desktop or client-server environments encountered a host of difficulties while trying to solve business problems in this new, stateless, request-and-receive environment. Chief among these problems was the lack of a computer language designed specifically to operate in a web environment.

Note

I used to work in a software development shop that did both desktop applications and web applications. The desktop team always had a good laugh at us on the web team because we could not "maintain state." I never quite figured out what was so funny about it, but I think those people are not laughing now, considering the growing popularity of the Internet.

Allaire's ColdFusion was the answer. ColdFusion was easy to use, and developers could quickly create web applications with it. A group of ColdFusion developers including (among others) Gabe Roffman, Joshua Cyr, Michael Dinowitz, Robi Sen, and Steve Nelson started discussing common solutions to the problem of reinventing the wheel with each new site development. The "House of Fusion CF-Talk" mailing list became home to the discussion. Threads on a better way to organize applications progressed, and ideas began to form within the group. Example frameworks were discussed, but a catalyst was needed to finalize the system.

That catalyst arrived when Steve and Gabe (both independent consultants at the time) started sharing an office in Charlottesville, Virginia in the spring of 1998. They began to swap notes and ideas about their own development practices and how they related to some of the CF-Talk ideas. Discussions of the state problem and design patterns led to ideas about organizing applications, controlling system actions, and separating functionality of pages.

Before long, Steve finalized his idea for a centralized controller page within each directory of a site. Gabe noticed the resemblance of this model to the breaker panel of a house. This "fuse box" ColdFusion application system was documented in a white paper as the first specification for Fusebox version 1.0. Joshua Cyr wrote an example calendar application based on that specification.

Impressed with the system, Robi Sen hired Gabe and Steve to create the first full-scale Fusebox implementation at eBags.com. This e-commerce site was a magnificent success and is still going strong today as the largest online retailer of bags. And the rest, as the saying goes, is history.

Since its inception, Fusebox has been based around an open exchange of ideas—a community. This community has been pivotal to Fusebox's success and growth. Make no mistake, we have come a long way since that first basic concept, but the fundamental ideas of Fusebox remain intact. These ideas are what make Fusebox the best web application framework for ColdFusion.

What Is Fusebox?

To explain what Fusebox is, let's look first at what it is not. Fusebox is not a software package—there is nothing to buy. Fusebox is also not a development environment, a compiler, or a code library, although there is some standard code involved with it.

However, Fusebox *is* a web development specification. People who use Fusebox (*Fuseboxers*) use a standardized development methodology and framework to help their projects succeed. Fusebox has the strongest and most dynamic community of ColdFusion developers on the Net today.

What is Fusebox all about in practice? There are two major aspects of Fusebox:

- Basic concepts of the purpose of Fusebox
- Fundamental Fusebox principles in practice

Each of these aspects is made up of other pieces, so let's take them one at a time.

Basic Fusebox Concepts

Fusebox borrows many of its essential concepts from other systems. First, Fusebox is a way to organize program code. Second, Fusebox is a way to manage growing applications by organizing the application directories in a hierarchy. Third, and perhaps most fundamentally, Fusebox is a way to think about applications and the art of building them. Comparisons have been drawn between the Fusebox controller file and computer networks. Now let's discuss each of these essential concepts in more detail.

A Way to Arrange Code

A great deal of thought has gone into the idea of organizing applications into some kind of logical framework. Code organization schemes are as old as computers. Back when computers used punch cards, programmers kept routines in tidy order. If you got a card out of sequence, your program did not function properly. And heaven forbid you drop a box of cards on the way to the computer! More discussions of code organization choices appear in Chapter 14, "Construction and Coding," such as the popular Model-View-Controller (MVC) framework.

The easiest concept to observe about Fusebox is the impact it has on your application's code—where you store files and how you arrange your directories. The Fusebox specification defines how to arrange your program's code into easy-to-manage chunks. What is the big deal with organized code?

Although many people quickly recognize the value of using a standard approach to arrange code, some of the benefits are often overlooked.

ColdFusion has been an incredibly successful language primarily due to its low entry threshold. That is, ColdFusion does not require a great deal of study and knowledge of programming to get started.

Because so many people who have had no formal background in programming or application development get involved in developing ColdFusion applications, a great deal of ColdFusion code is disorganized or poorly written. Sure, it does the job, but unfortunately, ColdFusion's reputation has unfairly suffered as a result. Instead of recognizing that ColdFusion accommodated an amazingly wide spectrum of experience from complete novice to application guru, rumormongers proclaimed that ColdFusion could not scale well.

Scaling means that a completed application will be able to successfully handle workloads larger than those for which it was originally designed. In the case of web applications, it usually translates to a large increase in the number of page requests or simultaneous users that the system handles.

Database issues aside, when an application is poorly designed and constructed, it is generally not able to scale very well. Program logic that works well for one or two users might overstress the server when several hundred users all try

to use it at once. Many ColdFusion applications are poorly designed and constructed, simply because their creators never considered the issues involved in effective application design.

However, ColdFusion's reputation as a small-scale application server is not deserved, and indeed has been greatly reduced in recent years. Projects such as RoomsToGo.com (the largest retailer of furniture online) and Autobytel.com (one of the largest car sales sites) have proven that ColdFusion can scale effectively when the application is properly conceived and executed. Both of these high-volume sites are not only ColdFusion efforts, but they also both use Fusebox as their architecture.

Note

Still quite Fusebox, you may notice that AutoByTel.com uses `?action=` rather than the familiar `?fuseaction=` in the URL.

A Way to Manage Growing Applications

The responsibilities of a manager can be defined by a constant struggle to determine when a project will be completed and how much it will cost. If, as a manager, you do not have a good idea of how the application's code is organized, then the uncertainty factor rises rapidly. Fusebox project managers can use this framework again and again, regardless of the size of the project, which aids in code reuse. For example, you might have written a report for accounting last week. This week, you can literally copy and paste it into the whole intranet without changing a thing.

Using Fusebox is good for project managers, but Fusebox also mimics the way managers work. If you have ever managed a project, you know that most of your time is spent coordinating the efforts of others. Managers do not tend to produce much, but they encourage and allow those underneath them to produce more, better, faster. They are skilled at assigning tasks, which helps manage a project's size. A comparison of Fusebox and corporate managers reveals that they share some of the same methods for getting work done. Fusebox encourages code to be highly specialized and focused. The structure helps to control all the smaller pieces. The controller files excel at delegating tasks.

If you are a reader of the *Dilbert* comic strip, you are quite familiar with the Pointy-Haired Boss (PHB), the epitome of a manager without a clue. *Dilbert* is funny because we can relate to the title character; most of us have worked for a PHB at one time or another. Consequently, it might be hard to imagine corporate managers as a model for solving a problem, but there is a reason that management hierarchy rules the structure of companies today.

People can only process so much information in any given day. Using a flat organization (where managers are nonexistent and everyone is an equal), every member of the group must communicate his ideas and information to every other member of the group on a regular basis. Obviously, this is inefficient and can easily grow to the point where everyone spends all his time involved in communication, with no time left to actually get work done.

ColdFusion applications that lack a real structure act similarly. Every page can link to every other page. In fact, every page must link to every other page to get something done. If you add a page, all the other pages must be updated. Every page must be aware of every other page in the system.

This is where the manager comes in. Sacrifice one person's ability to directly produce but give that person the role of facilitating communication, and everyone else gets more done. By filtering out all the other stuff that goes around other parts of the organization, team members can focus on the skills for which they were hired in the first place.

The manager can see the "big picture" of the organization—what projects are currently underway, what areas of uncertainty lie ahead, and what teams need resources to complete their job. A good manager is a master delegator. If a manager goes "into the trenches" to produce and directly contribute, communication suffers and projects slow down.

PHBs notwithstanding, by organizing and regulating the flow of information between teams, managers streamline business. Fusebox's controller file duplicates the purpose of a manager by passing requests off to focused portions of the application. If an employee needs to arrange a meeting with another department to complete a project, the manager knows whom to contact and facilitates the communication. Similarly, if a web site user wants to see a list of all products on sale, the Fusebox controller knows which files accomplish that task.

A Way to Think About Applications

When you sit down to work with an application, do you have a mental concept of it in your head? What does it look like in your mind's eye? Can you fit it all into your head at once? Do you think of the application's code as being organized in a particular way, or does it all just sort of run together in a big jumble?

Part of the problem of working with unorganized code is just trying to envision the application. Without a defined model, the picture that forms tends to be abstract. It is hard to imagine what happens when a particular template runs, how information is passed from one template to another, and what a change in one template does to another template. You might tend to think in terms of web servers and page requests or maybe streams of ones and zeros flying around if you understand that kind of thing. Although there is nothing wrong with this view, there is nothing tangible about it either. You cannot visualize the application. Or, as Beethoven said:

"[The work] rises, it grows, I hear and see the image in front of me from every angle... and only the labor of writing it down remains..."

Beethoven was a musical genius who was fully capable of composing a complete symphony in his head. Most folks are not as good at programming as Beethoven was at composing; we need a structure to our applications. Thankfully, we use things every day that can give us some valuable models for our applications. One example that closely resembles Fusebox is a computer network router.

Fusebox Mimics Networks

Over time, routers have helped organize network traffic immensely. Two of the most successful early network topologies were ring and bus architectures, shown in Figure 1.1.

These two architectures had one thing in common: They were based, as are all networks, on the idea of broadcasting packets of data from one machine to another.

As you can see in Figure 1.1, the bus architecture (at top) worked by machines sending packets addressed to other machines onto the bus, the central line in the figure. Each machine would check the address on every packet that went by. If the packet belonged to the machine, it would make a copy of the packet and read it. Otherwise, the packet would be ignored.

Acting somewhat like a bus network in a loop, IBM's Token Ring worked by each packet attaching a token that carried the address of the recipient. When the token passed the recipient machine, the machine stripped off the packet. If the token passed a non-recipient machine, nothing happened and it continued around the ring.

As you can imagine, both of these approaches meant that a lot of packets were flying around on the wires. Then came a new topology, and the problem got worse.

The newer topology, called a star, used a central hub to send packets from one machine to another. A hub was a simple concept; it took whatever came in one port and sent copies of it out on all the other ports. This made great sense when you had a single hub at the center of several computers. Shown in Figure 1.2 is the classic star network.

Networks based on hubs eventually reached a practical limit on the number of ports, so network engineers expanded the capacity of the hub by linking hubs together. However, that technique was not scalable. If you linked too many hubs, the multiplied traffic would cause collision problems, with traffic moving slowly.

The answer to this problem was the router. A router is the hub's smarter cousin. Whereas a hub simply copies what is received in one port to all the other ports, a router sends a packet only to the destination. Using strategically placed routers, network planners were able to create huge networks. In fact, the Internet is a series of interconnected routers.

By organizing and regulating the flow of traffic between workgroups, network routers streamlined the networking industry. Routers act as a big switch. If one computer wants to talk to another computer, the router essentially connects those two machines to each other. Similarly, Fusebox uses a master controller file to directly pipe a page request to the correct set of files. The controller file handles every action that the system can perform.

Traditional ColdFusion applications have no hub. Page requests are all point-to-point. As the system grows in size, developers can't remember which pages link where. It becomes extremely difficult to keep a clear picture of the application in your head. Imagine trying to memorize exactly what a pile of spaghetti looks like and where each noodle winds. Tough, huh? Now imagine a map of the interstate freeway system. That is a lot simpler, isn't it? It is pretty simple to find your way from Atlanta, Georgia to Atlanta, Idaho even if you have never been to either of those places. Freeways have structure, like Fusebox. Spaghetti has no structure, like traditional ColdFusion applications.

Back to Thinking About Applications

We started this discussion with the idea that applications are difficult to think about in tangible terms, and that Fusebox provides an easier concept to grasp. Each of the real-world examples (network routers and managers) is tangible if you are familiar with it. When someone mentions one of these examples, you instantly have an idea of what it is about and how it works.

Fusebox does something similar with applications. As you gain experience with Fusebox, you will become more comfortable with the associated terminology, and your applications will be easier to visualize. Now we just have to tell you what Fusebox really is.

Technical Fusebox Principles

If you are like us, you are probably thinking, "Okay, enough with the metaphors. Just what makes a Fusebox application in the real world?" Well, Fusebox has many technical aspects. Some people use all the ideas that have ever been labeled "Fusebox," whereas others only use some core components. For an application to be considered "Fusebox," it needs to exhibit all of the following characteristics:

- Use the Fusebox method of file and directory organization.
- Use the Fusebox core files.
- Switch on a "fuseaction" to control the flow of the application.
- Employ exit fuseactions (XFAs).
- Have all fuses contain Fusedocs.

What happens if an application only uses some of the five characteristics? Usually we call it "my own version of Fusebox." It might work perfectly well for you, in your own development environment, for your own application, but what would happen if everyone wrote applications differently? You could not proclaim that you "knew Fusebox" because you write it differently from others. Shared applications could not be said to be "written in Fusebox." The power of a widely used framework and methodology is lost. Most industries standardize on a certain set of specifications. When that happens, the industry booms.

In Part 2, "Fusebox Coding," we will discuss these concepts in greater detail. Throughout the book, we will look at other ideas and practices that (although not all Fusebox developers use them) are considered Fusebox "best-practices." For the following sections, we will give you some background so that you understand how the whole picture fits

together. Welcome to technical Fusebox. Put on your thinking caps!

File and Directory Organization

Any time you have more than a few files, it makes sense to reorganize them into a set of folders. Imagine you start a new school semester with a new laptop computer and you stick all your schoolwork in My Documents. After the first week, you have 10 files in there—a few Word documents, maybe an Excel spreadsheet of your class schedule, a Mind Map you created in Biology class (more on Mind Mapping in Chapter 14, "Construction and Coding"), and a handful of text files of notes. By next week, the number of files has doubled to 20. Sensing impending disaster by file overload, you wisely create folders for each of your classes and segregate your files. This works wonderfully through the end of the school year, but the next fall rolls around and you have all those folders from old classes—too many of them to quickly access folders for your current classes. You create a new folder called Freshman Year and move all your class folders from last year into that new folder. "Hey, I'm getting organized," you think. And you are right. Using folders to logically and physically separate files and subfolders is mandatory after you get a good number of them together.

What do we mean by "logically and physically"? Determining that you had too many files and putting them into separate folders is the physical separation. However, you didn't just create folders called folder1, folder2, and folder3 and randomly stick files in each one until they were equally full. You carefully named your folders Biology 101, Intro to Java, and Women's Studies and placed the files that you created for each of those classes into the corresponding folder. That is what is meant by "logically." Maybe your Intro to Java folder has twice as many files as Biology 101, but it makes sense to organize them according to class, not according to count.

The way we organize files and folders in Fusebox is nothing radical; it is just an extension of logical organization. Take, for instance, this site map of a couple HTML pages pictured in Figure 1.3.

The site map in Figure 1.3 is an example of the directory structure shown in Figure 1.4.

If the web site shown in Figure 1.4 were a Fusebox application, then the subfolders `customerservice`, `newsreleases`, and `catalog` would be called *circuits*. On the most basic level, a Fusebox circuit is just a directory. In Windows terminology, there are folders and subfolders, but in Fusebox, there is only one name for any directory, regardless of how deeply nested it is. Frequently, Fusebox developers refer to a circuit as being a *child* circuit of another circuit. That just means that a circuit is nested inside another circuit.

Circuit Inheritance

Circuits are more than just a collection of files. When you (using your imagination) organized your freshman year class folders into one folder, what benefit did it bring? It reduced the number of folders in one location, but it also allowed you to apply security (if you had some important files in those folders) to one folder and have subfolders inherit the security settings from their parent folder. If you were to mark the folder Freshman Year as read-only, you would not be able to modify files in subfolders of Freshman Year. Imagine if this feature were not built into Windows; you would have to go through each file and modify its properties by hand, which would be quite time-consuming.

Fusebox has the same system of inheritance. In fact, it is even more powerful than Windows's version. Inside each circuit is a file called `fbx_settings.cfm`. Variables set in `fbx_settings.cfm` are available to the rest of the circuit, including any nested circuits. This means that you can apply security to check whether a user is logged into one circuit's `fbx_settings.cfm` file; then every file in that circuit can forego its own security validation. Because the files in the circuit that contain the secured `fbx_settings.cfm` inherit its settings, they cannot be accessed unless the user is logged in. We will further discuss the security of circuits and the `fbx_settings.cfm` file in later chapters.

Note

Inheritance is a term used in object-oriented programming, but in Fusebox, it basically means what it

sounds like it means—a child circuit can use the variables from its parent circuit.

One good way to decide how to partition your application into circuits is to determine what parts of your application might be considered miniature applications. If we were building an e-commerce site (which we will do throughout Part 3, "Fusebox Lifecycle Process (FLiP)"), we would make the catalog/product viewer a circuit, as well as product reviews, checkout, customer service, search, and user management. Let's not forget about the home circuit, too. Every application needs a home circuit, which is used to tie all the circuits together into one application. Just as we applied a setting to `fbx_settings.cfm` and had all files in that circuit inherit those settings, we can apply settings to the home application, and every circuit in the entire application will inherit those settings.

File Organization

To review the terminology acquired thus far, a folder or directory in Fusebox is called a *circuit*. Any subfolders or subdirectories are said to be *child circuits* of the main circuit, which might be called the *parent circuit*. We would be remiss if we did not correct ourselves by replacing the term *file* with *fuse*. Often, ColdFusion files are called *templates*. Although the terms *template*, *file*, *page*, and *fuse* are relatively interchangeable, a file that is used in a Fusebox application carries two unique distinctions that any old ColdFusion template or file does not necessarily have: structured naming conventions and rules of use.

Fuses Versus Plain Templates

The first distinction is that Fusebox fuses follow a carefully defined yet extensible file-naming system. The content that can and should be contained in a fuse depends on its file prefix.

Table 1.1 File Prefixes for Fuses

Fuse Prefix	Fuse Type	Description
dsp_	Display	Only fuse type that can contain display, be it HTML, WML, or SOAP packets.
qry_	Query	Only fuse type that can perform database interactions, whether it be via <code><cfquery></code> or <code><cfstoredproc></code> . Always returns a recordset.
act_	Action	Any ColdFusion that does not fall into the previous two fuse types. Used primarily for data manipulation, form validation, and external systems interaction such as <code><cfmail></code> , <code><cfpop></code> , and so on.
lay_	Layout	These files assemble the page request for presentation to the user by handling headers, footers, and embedded fuseactions.
fbx_	Fusebox framework reserved fuse	The seven reserved Fusebox fuses use the <code>fbx_</code> prefix. No user-defined files should use this prefix.

For the majority of files in your Fusebox application, exactly what code should get which of the preceding four fuse prefixes is a no-brainer. A fuse that runs a `<cfquery>` tag to get a recordset of all products in a database with prices over \$30 would use the `qry_` prefix (that is, `qry_GetProducts.cfm`). A fuse that loops over the recordset created by a `qry_` fuse and displays each row formatted into a `<table>` would use the `dsp_` prefix (that is,

`dsp_showProducts.cfm`). A fuse that performs some complex validation based on the results of a user-submitted form would use the `act_` fuse prefix (that is, `act_validateProduct.cfm`).

The `fbx_` prefix is used for the reserved Fusebox framework files—those files that get the Fusebox job done by creating the inheritance structure, creating the central controller file, handling the layout and nesting of displays, and other tasks. For now, you should understand that nearly every fuse with an `fbx_` prefix is not a file you create, although you do edit the contents of some of those files. We will talk more about `fbx_` fuses starting in Chapter 3, "The Fusebox Framework."

The second distinction between a plain ColdFusion template and a Fusebox fuse is that each fuse should perform a discrete task. If a user is placing an order at the end of a checkout process, an example system has a number of tasks to perform, including checking the inventory, verifying and processing the credit card, updating the inventory, inserting the order into the database, sending an email to the shipping and accounting departments, sending an email to the customer, and finally displaying a "thank you" page to the customer. Each one of these steps would be a fuse. A good Fusebox developer would not combine these steps into one fuse, as tempting as it might be. A good rule of thumb is that most query and action fuses should fit on one screen in ColdFusion Studio.

Fuse Rules

A complete discussion of fuses and fuse content comes in Chapter 5, "The Fuses," but in the meantime, some hard and fast rules are available to keep in mind about what kind of code can and cannot be in certain fuse types.

For action fuses that use the `act_` prefix, there can be no display. Whether the requesting client is a browser, a WAP phone, a web service, or a `<cfmodule>` tag, `act_` fuses can only contain CFML logic and service calls. Use of `<cfoutput>` should be strictly limited to looping over query resultsets to perform manipulations of the data. In rare instances, an action fuse can perform database queries if, for example, you are trying to emulate the functionality of a stored procedure by running a query, looping through it, and performing subsequent queries based on the results of the first query. However, it would be wisest to `<cfinclude>` that nested query as a `qry_` fuse. Here is an example of an action fuse that creates a name list:

```
<!--- act_nameBuild.cfm --->
<cfset nameList="">
<cfloop collection="#attributes.stName#" item="aName">
  <cfset nameList=ListAppend(nameList,aName,)>
</cfloop>
```

This code takes a structure of names (`attributes.stName`) and creates a list of the key values (`nameList`).

Like action fuses, query fuses that use the `qry_` prefix cannot present display to the client. Query fuses should be as modular as possible to allow maximum reuse of the query. This means that the only code that should be contained in a `qry_` fuse is the `<cfquery>` (or `<cfstoredproc>`) tag, a few `<cfparam>` tags to create defaults for the query, and *maybe*, just maybe, some looping afterward to rearrange the data output if it could not be accomplished in the query. Query fuses must always generate a recordset. It is also a good idea to name the recordset (via the `name` attribute in the `<cfquery>` tag) the same name as the fuse, minus the file extension and fuse prefix. For example, the fuse `qry_customerDetails.cfm` has a recordset called `customerDetails`:

```
<!--- qry_customerDetails.cfm --->
<cfparam name="attributes.customerID" default="0">
<cfquery name="customerDetails" datasource="#request.dsn#">
  SELECT * FROM customers
  WHERE customerID=#attributes.customerID#
</cfquery>
```

Unlike the previous two fuses, display fuses are the only type that can present output to the client. As a consequence, display fuses should contain a minimal amount of ColdFusion logic. Common ColdFusion tags that are used in display fuses include `<cfoutput>`, `<cfloop>`, `<cfset>`, `<cfparam>`, and `<cfif>`. Of course, you are free to use whatever tags

are necessary to get the job done for a fuse, but if you find yourself writing blocks of ColdFusion code, it should probably be moved into an action fuse. Here is an example of a display fuse:

```
<!-- dsp_customerDetails.cfm -->
<table border="1">
<tr>
  <td>Name</td><td>Managers</td>
</tr>
<cfoutput>
<tr>
  <td>#customerDetails.name#</td><td>#nameList#</td>
</tr>
</cfoutput>
</table>
```

Because Fusebox encourages different kind of code to be created in different fuse types, the result is a complete separation of display from logic. Due to this separation, debugging is easier, development is faster, and the entire application has increased cohesion, bringing many benefits, discussed further in Chapter 2, "Is Fusebox Right for You?"

Note

Cohesion relates to how well a fuse, fuseaction, or circuit performs exactly one function or achieves a single goal.

The Fusebox Core Files

Applications that were developed in early versions of Fusebox consisted of a "federation" of circuits; in a metaphysical sense, each circuit lived in its own world, only vaguely aware of ties to other circuits. Circuits did not share the benefits of inheritance. To simulate inheritance, it was commonplace to tie circuits together, which reduced modularity. Extended Fusebox (XFB), created by Hal Helms, was developed to solve these problems. XFB gave circuits complete independence from each other but also created a true structure among the circuits, uniting them into one "wrapper" application. In the latest release (version 3), Fusebox builds on XFB by using a formalized set of core files that sets up a structure for the circuits, creates the system of inheritance and settings, provides for the switch-case that controls the application flow, and controls layouts and nested displays. Each core file uses the `fbx_` prefix.

Table 1.2 The Core Files

Processing Order	Filename	Description
1	<code>fbx_fusebox30_Cfxx.cfm</code>	Commonly referred to as "the core file," this file sets up the entire Fusebox framework that calls the other core files and fuses. The <code>xx</code> denotes ColdFusion version-specific files.
2	<code>fbx_circuits.cfm</code>	Establishes the relationship of circuits to each other by "registering" circuits with the application.
3	<code>fbx_settings.cfm</code>	One per circuit, this file allows circuit-wide (and child circuit) settings and inheritance.

4	<code>fbx_switch.cfm</code>	Little more than a <code><cfswitch>/<cfcase></code> statement, this file controls the application flow, based on the fuseaction.
5	<code>fbx_layouts.cfm</code>	One per circuit, this file controls which layout file to use for the request.
N/A	<code>fbx_savecontent.cfm</code>	Fusebox relies on <code><cfsavecontent></code> , introduced in CF5. This file emulates the native tag's functionality for people using versions of ColdFusion earlier than version 5.

All these files might sound confusing at this point, but fear not; they will be discussed fully in Part 2.

Controlling the Flow

Unfortunately, we both find ourselves flying more than we would like. Last year, we flew cross-country at least a half-dozen times and over the course of one two-month period, we flew nine times combined. But out of all those frequent-flyer miles accumulated, few were on delayed flights. We never got "bumped," and for the most part, the tickets were inexpensive and we were able to get on the flights we wanted, even on short notice.

But if this were 1950, it's unlikely we would have had such success. Back then, airlines were frequently late, ticket prices were higher, and direct flights were the only way you could fly. If you wanted to go from Los Angeles to New Orleans, you had to wait for a long time for a direct flight, or take any number of flights, each one quite expensive, and there was little guarantee that you would not miss a connecting flight. Airlines were operating on a point-to-point system but recognized that something needed to change to allow an increase in cities served, passenger volume, and profits realized. In the early 1960s, United Airlines created the world's first hub-and-spoke system of airports, linking western U.S. routes with eastern U.S. routes via major connecting hubs. At the time, the other major airlines nay-sayed this model, claiming it would destroy the industry. However, after a few short years, all the airlines had their own hub-and-spoke systems.

The system used by the airline industry these days is similar to the system used by Fusebox. In addition, the system used by the airline industry up until the early 1960s is similar to most non-Fusebox ColdFusion applications. Non-Fusebox ColdFusion applications tend to rely on page-to-page links. A form that is collecting information submits to a form-processing page. The form-processing page has a "thank you" section at the bottom. In Fusebox applications, every page links and submits to one controller file, which then processes the user's request. If you are on a page with a form, it submits to the Fusebox, which then passes control to the form-processing page. Because all links, form actions, JavaScript redirects, `<cfmodule>`, and `<cflocation>` tags go through the Fusebox, the hub-and-spoke system is duplicated in web applications.

Figure 1.5 shows a comparison of point-to-point versus hub-and-spoke systems in airlines and web applications. Point-to-point models can work well in small sizes, but they must be upgraded to hub-and-spoke systems to handle increases in traffic and complexity. This hub-and-spoke is most obvious in the "controller file," called `fbx_switch.cfm`.

A Look at `fbx_switch.cfm`

So what makes a Fusebox application actually run? In ColdFusion, you point a browser to a `.cfm` template, and the web server picks up the request and passes it off to ColdFusion Server. From there, ColdFusion processes the `Application.cfm` file—including any files referenced—and then processes the page, again including any files needed. When ColdFusion is done, the page goes back to the web server and you get your page. But in Fusebox, instead of pointing to individual fuses, you always point to one file: the default web document, which is usually `index.cfm`.

This `index.cfm` file includes the core file (`fbx_fusebox30_CFxx.cfm`), which runs code to set up the Fusebox framework, including `<cfinclude>`ing the `fbx_settings.cfm` fuses from the circuits. Next, it includes the `fbx_switch.cfm` for the target circuit, and then finishes off by handling any layouts to be used. But again, where do you stick your code—the stuff that actually does something?

All the fuses—your code—gets `<cfinclude>`d from the `fbx_switch.cfm` file. Here is an example `fbx_switch.cfm` file:

```
<cfswitch expression = "#fusebox.fuseaction#">
  <cfcase value="main">
    <cfinclude template="act_ReadFiles.cfm">
    <cfinclude template="act_MakeFileIntoStruct.cfm">
    <cfinclude template="dsp_map.cfm">
  </cfcase>
  <cfcase value="admin">
    <cfinclude template="act_getAllDirs.cfm">
    <cfinclude template="dsp_admin.cfm">
  </cfcase>
  <!-- ... other <cfcase> tags here... --->
</cfswitch>
```

That is all there is to an `fbx_switch.cfm` file. We will do a little more explaining later, but for now, just think of the Fusebox as a big switchboard, running different code depending on a user-supplied variable.

Fuseactions Control Flow

If the user wants your application to run the "main" fuseaction, the request that follows will do it. The `<cfswitch>` statement switches on `fusebox.fuseaction`, which is passed on the URL string like this:

```
<a href="index.cfm?fuseaction=home.main">
```

or in a hidden form field like this:

```
<input type="Hidden" name="fuseaction" value="home.main">
```

or in a `<cflocation>` tag like this:

```
<cflocation url="index.cfm?fuseaction=home.main">
```

Being an astute reader, you are probably wondering where `fusebox.fuseaction` comes from if we are only passing around `url.fuseaction` and `form.fuseaction`. Remember how we mentioned that the core file sets up the Fusebox framework? One job that the core file has is to take any incoming Fuseaction variable, no matter what the scope is, and copy the second part of it (after the dot) into a variable called `fusebox.fuseaction`. Referencing `fusebox.fuseaction` is similar to referencing `ListLast(url.fuseaction, ".")`. Instead of needing two `<cfswitch>` blocks, one switching on `url.fuseaction` and the other switching on `form.fuseaction`, we can have one that does the job no matter what scope it is. It even works if you pass in the variable `fuseaction` in a `<cfmodule>` tag:

```
<cfmodule template="index.cfm" fuseaction="home.main">
```

Now we have four different ways to call a fuseaction, and only one set of code to handle all those requests.

Every time `index.cfm` (or whatever your default web document is) is run, it expects the variable `fuseaction` to be passed in. However, sometimes that variable is not passed in. This case occurs when a user types in something like this:

```
http://www.thirdwheelbikes.com/
```

There is no `?fuseaction=main.welcome` in that URL, so Fusebox needs a default fuseaction. Default fuseactions are key to understanding how a Fusebox application responds. You set a default fuseaction in the root `fbx_settings.cfm` file for your application like this:

```
<cfparam name="attributes.fuseaction" default="main.welcome">
```

Now for every request of the Fusebox application, we can be assured that the variable `attributes.fuseaction` is available. Now the system can respond appropriately. Now that a fuseaction is known, how does the framework know which circuit to go to?

Which Circuit Runs?

The fuseaction variable always contains two values in what we refer to as a *compound fuseaction*. We know that the second part of the fuseaction is the actual fuseaction; it determines which set of fuses is run for a given user request. Imagine that a user requests the `order.basket` fuseaction like this:

```
<form action="index.cfm?fuseaction=order.basket" method="post">
```

We know that the `basket` fuseaction will be run, but we also know that the `fbx_switch.cfm` file in the `order` circuit should handle the request for that fuseaction. Using this compound fuseaction, we can specify any fuseaction in any circuit to be run. The first part is the circuit, and the second part is the fuseaction.

Imagine that a customer is buying a product on your e-commerce site. Halfway through the checkout process, she decides to review your company's privacy policy. All of the fuses in the checkout process are in a circuit called `checkout`, which is off the root of the site. The page that she wants to see, `dsp_privacy.cfm`, is in the `policy` circuit, also off the root. How would you write the link from the checkout page to the privacy policy fuseaction? How about this:

```
<a href="index.cfm?fuseaction=policy.privacy">
```

The first file that runs is `index.cfm`, which is the file that the link requested. The Fusebox framework file (`fbx_fusebox30_Cfxx.cfm`) is included from the `index.cfm` page. The core file establishes the framework and calls the `fbx_switch.cfm` file from the `policy` circuit, which looks like this:

```
<cfswitch expression = "#fusebox.fuseaction#">
  <cfcase value="privacy">
    <cfinclude template="dsp_privacy.cfm">
  </cfcase>
  <cfcase value="security">
    <cfinclude template="dsp_security.cfm">
  </cfcase>
</cfswitch>
```

The `dsp_privacy.cfm` file is processed because the second part of the fuse-action (the part after the dot) matches the `<cfcase value="privacy">` tag. After the Fusebox core file finishes processing, your display file is presented to the customer, who is satisfied with the privacy of your site and happily completes her purchase.

Exit Fuseactions

By now you might be imagining the `fbx_switch.cfm` as something of a roadmap for a circuit. In one easy-to-read page, it shows every possible action that can occur, every possible page that can be displayed, along with each fuse used to accomplish the job. Each fuse is specialized and knows its job well ("I insert a new customer into the database." "I display the privacy policy," and so on). But because each fuse contains links, form actions, and `<cflocation>`s, each fuse must know where those links go—what the circuits are called and what fuseactions are in which circuits. That is a tall task for a measly little fuse. Fuses should be designed so that they contain as few dependencies on the rest of the application as possible. It is with this in mind that we use XFAs rather than hard-coded

fuseactions. Here is a sample XFA:

```
<form action="index.cfm?fuseaction=#xfa.submit#" method="post">
```

It's not too complicated, huh? XFAs are nothing more than fuseaction values as variables. They are set in the `fbx_switch.cfm` just before `<cfinclude>`ing the fuse:

```
<cfcase value="privacy">
<cfset xfa.continue="policy.payments">
<cfset xfa.back="policy.security">
<cfinclude template="dsp_privacy.cfm">
</cfcase>
```

This allows the `fbx_switch.cfm` file to control where each fuse it calls is allowed to go next. In the preceding case, the file `dsp_privacy.cfm` has two links: most likely a forward and backward button to guide an inquisitive customer through the array of site policies. Because each fuse should not be required to be aware of its surroundings, its own exit locations must be dynamic. This becomes especially important if a fuse links to another circuit. If the target circuit is moved or no longer exists, the fuse does not need to be updated—only the `fbx_switch.cfm` file does. XFAs also promote reusability of code. By using XFAs, we can write the fuse once but use it in different locations, and the links can go to different places. XFAs are completely discussed in Chapter 6, "Exit Fuseactions."

Fusedocs

Because an individual fuse is supposed to be designed and coded as if it were unaware of the larger application using it, a formalized method needs to exist to describe everything that the fuse can assume and everything that it cannot. By carefully documenting every variable that a fuse is allowed to reference (`#variable#`) and every variable that a fuse is supposed to create (`<cfset foo="bar">`), we can be assured minimum coupling by our most discrete units.

A Fusedoc appears at the top of every fuse. Using XML, a Fusedoc describes all the variables that the fuse needs to get its job done as well as all the variables that it needs to create. It identifies the scope, structure, mask, and potential values of those variables. Following is an example Fusedoc. It is for an action fuse that deletes a task. Do not worry if all the code is foreign to you. A full explanation follows the listing.

```
<!---
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE fusedoc SYSTEM "http://fusebox.org/fd4.dtd">
<fusedoc fuse="act_deleteTask.cfm" language="ColdFusion"
specification="2.0">
  <responsibilities>I delete a task and any user_tasks associated with
that task.</responsibilities>
  <properties>
    <history author="Nat Papovich" date="08/10/2001" email="nat@fusium.com"
role="Architect" type="Create"/>
  </properties>
  <io>
    <in>
      <list name="taskID" scope="attributes" optional="False"/>
    </in>
    <out>
      <boolean name="success" scope="variables"/>
    </out>
  </io>
</fusedoc>
--->
```

If you have not picked up XML yet, do not fear. Fusebox.org has tag chooser and tag insight files available to plug into ColdFusion Studio, which greatly simplifies creating Fusedocs:

```
<fusedoc fuse="act_deleteTask.cfm" language="ColdFusion"
```

```
specification="2.0">
```

Starting at the top (below the mumbo about XML versions and doctypes), the tag `<fusedoc>` wraps the whole Fusedoc. Anything that falls within the opening and closing `<fusedoc>` tag is considered part of the Fusedoc. The `<fusedoc>` tag contains a number of attributes, including the name of the fuse:

```
<responsibilities>I delete a task and any user_tasks associated with
that task.</responsibilities>
```

Next, the `<responsibilities>` tag describes what this fuse accomplishes, in plain English:

```
<history author="Nat Papovich" date="08/10/2001"
email="nat@fuseboxtraining.com" role="Architect" type="Create"/>
```

The `<history>` tag is like a stamp of who did what when to this fuse. This example shows the creator of the fuse:

```
<io>
<in>
  <list name="taskID" scope="attributes" optional="False"/>
</in>
<out>
  <boolean name="success" scope="variables"/>
</out>
</io>
```

The final section describes the input and output (`<io>`) of the fuse: what variables are coming in and what variables need to go out. This example shows that a variable called `taskID` in the attributes scope is available and that a variable called `success` in the local variables scope should be created.

We now have an understanding of the benefits that Fusedocs offer to our applications. If you are uncertain about the XML format in which to write Fusedocs, don't worry; <http://www.fusebox.org> has a free Fusedoc toolkit that plugs into ColdFusion Studio and makes writing Fusedocs a snap.

That about covers the basics of Fusebox except for one thing—a common scope for variables that users can modify.

A Common Scope

One final characteristic that is fundamental to Fusebox is a shared scope for incoming variables. We have already mentioned how the Fusebox core file copies the second part of the incoming form, url, or attributes-scoped variable `fuseaction` into a variable called `fusebox.fuseaction`. That is how the `fbx_switch.cfm` knows which `<cfcase>` tag to run for the request. However, the core file also copies all url and form-scoped variables to the attributes scope. This feature is commonly referred to as "form URL to attributes" because it was originally accomplished using a custom tag created by Steve Nelson called `formurl2attributes.cfm`.

Having a single scope to refer to user-defined variables means that portions of Fusebox applications suddenly become incredibly reusable. Imagine you have a `<cfquery>` tag like this:

```
<cfquery name="customerDetail" datasource="#request.DSN#">
  SELECT * FROM CUSTOMERS
  WHERE customerID=#form.customerID#
</cfquery>
```

This bit of code is saved into a file called `qry_customerDetail.cfm`, but it can only be used if the previous page contains a form with a field named `customerID`. If you also want to be able to pass the `customerID` variable on a URL string, you would have to rewrite that query file to be something like this:

```
<cfquery name="customerDetail" datasource="#request.DSN#">
```

```

SELECT * FROM CUSTOMERS
<cfif IsDefined("form.customerID")>
WHERE customerID=#form.customerID#
<cfelseif IsDefined("url.customerID")>
WHERE customerID=#url.customerID#
</cfif>
</cfquery>

```

The code is generally the same as the earlier code, but it now takes into consideration `customerID` coming from two different scopes. This is a kludge. What happens now if you want to be able to use this query file from `<cfmodule>`? You would have to add yet another `<cfelseif>` to the growing code to accommodate. In addition, you would have to add this chain of `<cfif>`s to every query you wanted to reuse.

Because URL, form, and attributes scope are all available for users to write to, Fusebox uses the attributes scope as a catch-all for those scopes. Early in the processing of the Fusebox core file, all variables in the form and URL scopes are copied to the attributes scope. Now you can write the same query like this:

```

<cfquery name="customerDetail" datasource="#request.DSN#">
SELECT * FROM CUSTOMERS
WHERE customerID=#attributes.customerID#
</cfquery>

```

By using one scope to reference all incoming variables, our applications gain a huge boost in reusability. We are no longer concerned with exactly how a request is being made. We do not need to differentiate between a `<cfmodule>` request and a form post. Reusability rules.

That Was Fusebox

A long time ago, a really smart Neanderthal knocked the hard edges off a big stone and made a wheel. That kind soul shared the invention with others, which enabled us to evolve into a highly intelligent bunch of people. Although no Fuseboxer claims to be as smart as that one Neanderthal tens of thousands of years ago, we do take pride in creating a solution once and sharing the results. When a developer shares a technique that is particularly helpful, it can eventually be combined into Fusebox. Developers also make and share tools to speed up coding time. We can benefit by sharing our common knowledge. That is how Fusebox started—a couple guys found something great and shared it with the world.

Fusebox solves many problems, both by encouraging a way of thinking about applications and by providing cold, hard, technical solutions. The Fusebox framework files have been created for you, and you can focus on making applications quickly. The system of file and directory naming and organization helps you manage growing applications. In Fusebox, ColdFusion developers have found a methodology and framework that enables their applications to scale easily while they get to work less.

Should a solution be used just because it works? If you are in the middle of making a cake and realize you need some more butter, should you use the grocery store's online shopping and have it delivered to your door for an extra ten bucks? What about your cake? It will be waiting for the delivery to arrive. Maybe driving a couple of blocks to the store is a better idea. The latest technology should not be used blindly just because it is cool.

Expounding on technical specifications might convince some of you to use Fusebox, but for most of you, you are probably wondering what good Fusebox is. How will it save you time, money, and stress? Transitioning from a traditional point-to-point model of building ColdFusion applications to the more rigid Fusebox framework might seem like too much work. Just wait for us to cover the benefits of using Fusebox in the next chapter.