Steve Rubenstein's Recurring Billing
Architecture

Version 1.1
Last Updated October 15, 2013

While the architecture does make heavy use of ColdFusion components, they are not called directly, either via URLs, form actions or web services. This is to ensure security and validate that all required fields are present, which a component cannot do without returning an error directly to the user. Instead, all components are called via normal cfm templates, and a control template exists for each type of feature that then calls the appropriate component. While this method does not take full advantage of the benefits of components, it ensures that only authorized users perform each action and allows for better internal error handling.

Directory Structure:

importExportTemp/	Non-public directory with a subdirectory for each client (primary company). Used for storing temporary uploaded files and export files.
root/	Files for shopping cart system and later customer interface.
admin/	Admin screens. Also contains login and security.
c/	Customer directories. Used for storing custom headers, footers and uploaded product images.
images/	General images used on the site.
include/	Included files that are not called directly via the admin screens.
config/	Configuration files specific to that installation, including application variables and exception handling.
function/	General user defined functions used across the entire site.
merchant/	Code for processing credit cards via the various merchant account gateways.
security/	Authenticate users (login/logout), authorize their permissions, and exception & error handling.
template/	Templates used to display products, categories and invoices / receipts.
control/	Primary directory to store all business logic and processing code, including components, for system functionality.
c_dir/	Business logic and processing code for individual functions. Typically has matching directory name in v_dir.
p/	Partner directory. Contains subdirectory for each cobrand partner to store custom header, footer and other cobrand-specific code. Not currently used, but would be if system is primary system and has cobrand sites.
scheduler/	Scheduled scripts for processing subscriptions, payments, etc. Added to the ColdFusion Administrator.
view/	Primary directory to store all display code.
v_dir/	Display code for individual functions.
webservice/	Stores code for all web service components and webservice-specific

	code.
webserviceSecurity/	Security code to validate each target type. Stored in a separate directory since they are stored as application variables and accessed via all web services..
ws_dir/	Directory that mirrors the control directory structure to separate webservice-specific code.

In the admin screens, all requests are sent via the admin/index.cfm file. All URLs begin with:

index.cfm?method=control.action

"control" – the control subdirectory which contains the function. Loads the control_dir file.
"action" – the name of the function called via the control file.

For web services functions, requests are sent to each webservice component that mirrors the components from the control directory except that the component begins with WS.

Each directory contains an index.cfm and an Application.cfm if necessary. These files, respectively, are used to disable directory browsing and to prevent users from accessing template files directly that should not be called directly.

index.cfm
<CFSET blockDirectoryBrowsing = True>

Application.cfm
<CFLOCATION URL="#Application.billingUrl#" AddToken="No">

The "blockDirectoryBrowsing" variable does nothing, but is simply used so that the template is not blank. ColdFusion will search up the directory tree until it finds an Application.cfm file, which will then redirect the browser to the default homepage upon recognizing the invalid page request.

The Application.cfm exists only at the top level of each subdirectory, i.e., the "images" directory but not any sub-directory within it. It is only in those directories that are not meant to be called directly.

Included Filename prefixes:

act_	Performs processing, but no display.
comfirm_	Displays comfirmation messages after processing form.
control_	Main file in each directory that controls the processing for a given action. There is a primary control_#dir# name that controls access to all subsequent control files for individual functions.
dsp_	Displays html to browser. Typically used for displaying query results.
email_	Sends email.
error_	Display error messages either after form validation or after redirect.
header_	Displays section header.

fn_	User defined function.
footer_	Displays section footer.
form_	Displays html form.
formParam_	Determines form variables for display in form.
formMaxlength_	Stores maximum length for text form fields.
formValidate_	Validates form fields.
lang_	Variables that store text displayed to user, typically for form validation errors.
nav_	Displays navigation elements.
qry_	Sends a query to the database.
qryOrderBy_	Determines sort parameters of query.
qryParam_	Determines parameters passed to where clause in query.
security_	Ensures user has permission for requested object (user, product, etc.)
sched_	Scheduled script run automatically by ColdFusion Scheduler.
var_	Stores variables used for processing.
WS*.cfc	Name of ColdFusion components used for web services functions.
ws_	Web services file called from web services component.
wsact_	Web services file that is included by ws_ files.
wslang_	Stores error messages returned by web service functions.

Other Files:

filename.cfm – All ColdFusion files meant to be called directly via the browser, not included.
Filename.cfc – All ColdFusion component names begin with a capital letter and no prefix.

Security & Permissions

Each user is limited to their permissions that are assigned either directly to that user or via the group(s) of which that user is a member. Permissions are separated into categories, where each permission category has a maximum of 32 permission listings. For each permission listing, there is an corresponding list of one or more actions that pertain to that permission. Each action is an actual function within the system. Permissions are enforced using the bitwise method. More information on this method is in the ProposedSecurityModel.pdf file.

Primary Company

All users in the system must be part of a company. If a customer is a consumer, simply leave the company information blank. Each company may have zero or more users. A primary company is the company which is using the billing system to bill their customer companies. Primary companies are basically users of the billing system itself.

Affiliates, cobrands and vendors are companies first, and then an affiliate, cobrand or vendor listing is added for that company. They must be created by starting with the company, but can be viewed and updated separately.

Primary Targets – primaryTargetID & targetID

In many tables, the record may refer to one of many possible objects, such as a user, company, group, affiliate, cobrand, etc. For instance, the Note table stores notes for all objects in the system. You can record a note about a particular user, company, invoice, etc. With each note, we record which object the note refers to. However, since the Note may refer to more than a dozen objects, we need a method to determine the type of object it refers to (i.e., a user or company) and the primary ID of that object (companyID = 5). The alternative is to create a separate Note table for each object type, which could mean over a dozen Note tables.

primaryTargetID – An integer field that corresponds to the object type (e.g,. "user").
targetID – An integer field that corresponds to the primary ID of that object (e.g., userID = 5)

The primaryTargetID value for each object (a.k.a., a primary key from a table) is linked via the avPrimaryTarget table and is also stored in a function in the code for quick code-level access. The function file is include/config/fn_GetPrimaryTargetID.cfm . When a new table is created, the table information is added as a row in the PrimaryTarget table via an admin screen. The code-level function is updated with the new object information as well. Other than the system configuration information, this is the only file that may vary across customer installations, such as if they add their own tables or add new tables in a different order. While this is annoying, as long as the new table/object is added via the admin screen, all is well.

companyID vs. companyID_author

Definitions:
"client" – The company using the billing software to bill their own customers
"customer" – A customer being billed via the billing software

In some tables, it is necessary to store the ID of multiple companies and/or users. For instance, the Invoice table contains both companyID field and companyID_author fields where companyID is the customer that the invoice is for, and companyID_author is the client who is invoicing the customer. Similarly, the Invoice table contains a userID field which is the contact person at the customer for that invoice and a userID_author field which is the client employee that created or last updated the invoice.

In "setup" tables which store client preferences, such as a product created by the client, the database stores the "companyID" of the company the product belongs to and the "userID" of the user who created or last updated the product. Because there is no companyID or userID necessary to refer to the customer, the companyID and userID fields refer to the client.

In essence, in some tables companyID refers to the client and in other tables it refers to the customer; and userID refers to the customer or the client employee that created that record. This is admittedly confusing, but the generally concept is that if there is only a single companyID or userID necessary in the table, then it is silly to append "_author" to it.

Data Archiving

Archival Requirements:

1. Provide audit history of records that have changed
2. Maintain referential integrity for historical records
3. Enable database join on a single field from foreign key to primary key

1. Provide audit history of records that have changed

Wherever possible and practical, the system stores all versions of customer data. For instance, if the customer's name or address changes, it is necessary to track all instances of the data for future reference.

2. Maintain referential integrity for historical records

If a field value is updated, it is preferred to know which version of the data was in effect for a related instance. For example, if an invoice references a particular address and the address is later updated, when viewing that invoice, the system should be intelligent enough to automatically determine the invoice used the old address and not the new one without having to manually determine which address was in effect on the date of the invoice.

3. Enable join on a single field from foreign key to primary key

Where possible, a single primary key field is used in each table. All single primary keys are "identity" or "autonumber" fields automatically generated by the database – integer fields incremented with each new record. Using a single field primary key enables a direct relationship from a separate table via the primary key / foreign key method. This is more efficient when querying and joining tables than joining on multiple fields, such as an ID field and a version field.

Archival Types:

A. Archive individual updated fields, not the entire row
B. Archive entire row by making original row inactive and inserting new row
    i. Update necessary foreign key values with updated primary key
    ii. Copy existing data to replicate settings of original row

A. *Archive individual updated fields, not the entire row*

When user and company fields such as first name, email, company name, etc. are updated, only those fields that were updated are archived. Because it is likely that all of the user or company fields were not updated all at once, it is un-necessary to archive all user or company fields if only a few fields were changed. Moreover, just because a user's last name changed – perhaps the person got married – it does not mean their first name changed. While it is possible that a person changed both their first and last names at the same time, it is not nearly as common.

B. *Archive entire row by making original row inactive and inserting new row*

Unlike user and company fields, an address change typically means more than one address field has changed, e.g, it is unlikely that only the street address changed. Often times, the address, city, state and zip code may change. Archiving only those fields within the address that changed make it difficult to view the full old address vs. the full new address. Piecing together the old address when each address field is stored in a separate row is not intuitive or even correct since a single address field does not adequately describe an address.

For these types of instances, such as updating an address, the entire row is archived and a new row is inserted. This makes it easy to view the full old address or the full new address. However, inserting a new row also creates a new record with a new primary key. The records are linked so that it is possible to determine that row #3 is a copy of row #2, which was itself a copy of row #1. This link is stored using 2 fields: ID_parent and ID_trend.

addressID – Primary key field for Address table

addressID_parent – addressID of address that was updated to create this new address. In other words, we "updated" the address record of addressID. But rather than updating the addressID record itself, we created a new addressID where the addressID_parent is the old addressID. So if addressID = 5 and we update that record, we insert a new record and thus create a new addressID = 6 where 6 was created from 5. The record represented by addressID 6 has a addressID_record = 5.

addressID_trend – ID of original address that was created. For instances where an address has been updated more than once, we want to know not only which addressID was the immediate parent, but also the original address. In our example above, if we updated addressID = 6 (which was itself created from addressID = 5) we create a new addressID = 7 that has addressID_parent = 6 and addressID_trend = 5, since that is the original addressID for this series of addresses. When a record is the first instance, the addressID_trend = addressID. So if addressID = 5 is the first address listed for a user, then addressID_trend = 5.

i. *Update necessary foreign key values with updated primary key*
If an address is updated, there are records that should maintain a link to the old address and other records that should reference the updated address. For instance, an invoice is associated with a particular billing address and that address is also the billing address for a credit card.

If the billing address is updated, the billing address for the invoice should not change since it was the address at the time of the invoice. In the payment records where the credit card was used to pay, the billing address used also should not change. But going forward, the credit card should reference the new address or the credit card charge may be rejected because of incorrect billing address. So the addressID field in the CreditCard table is updated with the new addressID, i.e., addressID is set to 6 where addressID = 5.

ii. *Copy existing data to replicate settings of original row*
In instances where a record is more than a row in a single table, all records must be replicated when the record is updated and a new row is inserted. For instance, a custom price not only has a row in the Price table, but rows in other price-related tables such as volume discounts and which companies the price applies to. When a price is updated, a new price is created and all rows in the other price-related tables are copied with the new priceID.