



Mountain Climbing Journal Baroque Knowledge Management System High Level Design

Introduction

Hybrid HLD

MCJ has shouldered many metaphors spanning over two decades, consumed thousands of hours of my time with development and tinkering, and has grown into a baroque knowledge management system that is very hard to contain and express. The purpose of this high level design (HLD) document is to make some order and sense out of the sprawl. Even more important, though, is to try and express what the reason is for the system and the key considerations so that somebody might use and improve MCJ. It might also keep me on task a bit.

Typically an HLD would be used to describe a system before anything was built. In this particular case the journal system exists in a stable and useful form that fulfills most of the characteristics of the design being expressed. I'm working backwards by taking the system I built and writing up an HLD for it. I am writing this HLD on the journal system, running on an operating system created by the journal system, and using it as a prototype for this HLD document. There are some advantages to this, in that it makes it a lot easier to include examples illustrating what the finished system might look like. I do have quite a few technical solutions I've created to solve various problems I've faced with the journal system, and I'm able to provide details as part of the HLD. In design document terms, then, this is a hybrid solution description and HLD. As much as possible, though, I will try to keep this at the level of an HLD, while still retaining the advantages of its hybrid nature. This HLD hybrid does make it easier to understand what this document describes when I can display screen shots:

```

People | Places | Things | Times | Types | Entries | Realms | Journal | Images | URLs | Moun
a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
-----
realm: mcjrcatdb1 -> realm: 12977 ]
for y in @people_with_arts
  @arts_matching_people = Art.find :all,
  :conditions=> [{"
  artnum= "+y.artnum.to_s+" AND realm="
for z in @arts_matching_people
  z.classification=x.person
  @people with person << z
-----
/opt/mcjr/lib/Commcont.rb
Associate File | Save | Launch Editor | | &ccceeee

```

Scope

The HLD document presents the structure of the journal system, including the database, user interface (UI), and various layers of the surrounding application and supporting systems.

Definitions

MCJ - The Mountain Climbing Journal (MCJ) is a prototype of a journal system that L1G3R Information Systems developed organically and without design over a five year period. Earlier versions of MCJ go back as far as 1994. This document uses MCJ as a prototype and example, but its goal is to provide the design details necessary for somebody else to improve on MCJ.

Overview

This HLD will present all of the design aspects and define them in detail. This will include details about the user interface, hardware and software interfaces, performance considerations, design features, and architecture. It will also describe non-functional aspects like:

Security

Reliability

Maintainability

Portability

Reusability

Application Compatibility

Resource Utilization

General Description

MCJ Perspective

The Mountain Climbing Journal (MCJ) is intended to be used as a personal journal. It is true that it can be used for many different kinds of information, but the perspective comes from the perspective of a personal journal. MCJ facilitates freedom, consistency, reflection, and flexibility. First and foremost, though, the MCJ system is about exploration.

The greatest explorers used a journal to describe and trace what they saw in an open way. Part of the metaphor behind MCJ is that mountain climbing is a state of mind free from judgement. Judgement filters and transforms data too early. An example of a good journal entry is this entry from James Cook's journal of his first voyage:

Tuesday, 30th. Winds at North-West, Gentle breezes, and fair weather. Early in the A.M. a boat was sent to one of the Islands to get Sellery to boil for the People's breakfasts. While our people were gathering it near some empty huts about 20 of the Natives landed there--Men, Women, and Children. They had no sooner got out of their Canoe than 5 or 6 Women set down together, and cut and sacrificed themselves--viz., their Legs, Shins, Arms, and Faces, some with Shells, and others with pieces of Jasper. So far as our people could understand them, this was done on account of their husbands being lately killed and devoured by their Enemies. While the women was performing this Ceremony, the Men went about repairing the Huts without showing the least Concern. The Carpenter went with part of his people into the Woods to cut and Square some Timber to saw into boards for the use of the Ship, and to prepare two Posts to be set up with inscriptions on them.
From Project Gutenberg

Imagine how Captain Cook might have reacted at that time to this entry about a horrifying story. His entry is not derailed by any criticism. This does not mean that meaning can't be gleaned through analysis, it just means that the data has to get into the journal in as unfiltered a way as possible and the UI should facilitate this.

Tools Used

The table below lists the tools used in this design, the purpose of the tool, and the specific version of the tool used in the working prototype. All of the tools listed in the prototype column are included with the free MCJ Reference Operating System, along with source and build instructions documented on this site.

Tool	Purpose	Prototype
Database	Storage of configuration, markup, and entries	SQLite
Database Browser	Database browser with graphical user interface	SQLite Database Browser
User Interface	Cross-platform operating system integration	Real Studio
Operating System	Free platform to run MCJ applications	GNU/Linux
Web Framework	Isolation of control and presentation layer	Ruby on Rails
Web Server	Standard view and interaction via hypertext	nginx , Apache
Web Browser	View web pages	Firefox
Image Manipulation	Create, transform and convert images	GIMP , ImageMagick
Diagramming	Create, transform and convert vector drawings	Xfig
Text Layout	Creation of structured text layouts	LyX

Constraints and Assumptions

This design is constrained by my faith in, and skill with Javascript. Early on in the big push for a working journal system that captured keywords, I started to explore Ajax as a way to provide a richer integration with the local operating system. My conclusion was that the features I wanted would be difficult, if not impossible to create with Javascript. I am assuming that a locally run executable and database is a better solution than a Javascript-based solution. A good example of this is the feature where I use a slider bar to navigate local, live configuration files easily by changing colors on the output pane. This design constraint causes isolation between the main application and the presentation layer. This isolation has some benefits, but it also makes presentation review awkward, and complicates wide adoption.

Special Design Aspects

Reflection and consistency are special considerations for MCJ that should be called out. I take this to an extreme stance. I am writing this entry using MCJ, running on the MCJ GNU/Linux Reference OS. The same application can compile and document the creation of the OS. In my mind this is a way to short-circuit quality assurance. I am one person and need to prove proper function and design of the prototype without a lot of testing cycles. As an example, marking up and presenting the

sequence of the OS build is much different than a sequence of journal entries. This also serves as a form of proof that the MCJ design is applicable to general sets of data, transforming that data into meaningful information via keywords, sequences, and classification. Finally, the information and relationships should be able to be presented in navigable form, regardless of the original data. This conforms to the philosophy of MCJ that the initial gathering of data should be open, without filtering, and without concern that the information cannot be analyzed and classified at a later time. It doesn't matter if the set of data comes from the captain's ship log, a dream journal, or the build steps for an operating system. Reflection and consistency validates the design.

Main Design Features

Freedom

It is very important that any knowledge management system be free, free in the sense that the tools used to create the information, the data store, and the presentation are able to be modified without encumbrance. The owner of the information, the person or organization that builds information by relating data, should own the continued means of presenting and maintaining that information. A related consideration is that the owner of the information should be able to maintain the system without relying on others. This might mean that the system should be able to operate whether a particular Software as a Service (SaaS) company stays secure, performs, or remains in business. It might also mean that a more manual system should be used due to the technical skills of the operator. Perhaps a paper journal might be better? There are different aspects of freedom. My solution, my offering in this regard, is that I'm providing a complete system for free that satisfies my requirements and design intent. I am aware, though, that the level of complexity of my technical solution may well be too complicated for many. This design document will give you the opportunity to design a system for yourself that provides you more freedom if you wish.

In my professional life as a systems engineer and architect, I maintain that the most direct way to gain freedom when implementing information technology (IT) is to express the configuration items (CIs) of the system precisely and own the process that expresses those CIs into infrastructure. Gaining control of the environment after the fact with fuzzy tools that guess at implementation is expensive and imprecise. Sure, there will be mistakes in specification, but if you converge on a more and more accurate understanding and definition of your CIs, then you will waste less time with discovery and auditing your IT landscape. My goal with L1G3R.COM is to specify my baroque knowledge management system and related CIs in a clear enough way that if you do happen to have the time and interest to embrace the prototype system, you will have freedom to configure and morph your own knowledge management tool.

Persistence

It is important that care be taken in choosing the data store and relationships so that this has a good chance of migration and usage over time. As an example, if I had my data on a CP/M 2.2 single sided, double density floppy disk in the form of WordStar documents with a DataStar database circa 1983, then I likely could migrate that information today, in 2010, assuming that the floppy disk was still readable. We have a much more likely means of persistence today, in that relationships can be captured in the same document as the presentation.

Minimal Data Filters

There should be little distraction, seeding, or filtering when using the main data input console on the user interface. This just means that there shouldn't be ads, and that there are no distracting requirements of the user. An example of this would be extra navigation for a frequent task like tagging keywords. As humans, it is impossible to avoid filtering data as we observe; however, it is important to keep this to a minimum so that as much unfiltered data as possible gets inside the journal before

analysis takes place. Often the most interesting and valuable information is not apparent at the time of entry.

Cumbersome operational tasks can distract input of data and keyword identification. As an example, going out to a command line to find and edit a particular file or publish the web site should be avoided. Maintenance and status pages should be easy to use, yet remain undistracting.

Flexibility

Whatever works best for the user of the journal should be possible. This includes flexibility in the presentation and in the types of categories available.

Keyword markup

Keyword markup is the single most important design feature of MCJ. To provide this, yet keep the data filters to a minimum, keywords are classified as a person, place, thing, or time, and they can be added without additional navigation. This is not flexible, because if it were, it would detract from the design decision of minimal data filters.

Configuration Management

The knowledge management system should be able to track the CIs of itself. As much as possible, this should be integrated with the presentation layer in hypertext form so that the configuration is browsable.

Categorization

Categorization of the entries should be very flexible. This includes the relationships as well as the presentation. It should be possible to categorize in multiple dimensions. As an example, you should be able to assign articles to both time periods, arbitrary sequences, and membership in multiple groups.

Partitioning

It should be possible to partition different domains of information and data and yet use the same instance of the application. As an example, you might want to keep your dream journal separate from your writings you publish about ecology, yet you don't want to have to maintain separate databases.

Distinct Entry Comments

The system should allow entries that are distinct from the original entry, but still directly attached. This allows commenting and use of the entry without disturbing the original words. I've had particularly confusing entries in my personal journal where I have commented repeatedly about a puzzling reference. In most cases I figure out what I meant. Even if I don't, the speculation over time is interesting. This applies to both the time of the article and the entry itself. I may write a memory entry that fits into a time sequence, but it is helpful to know when I actually wrote down the memory.

Application Architecture

Presentation Layer

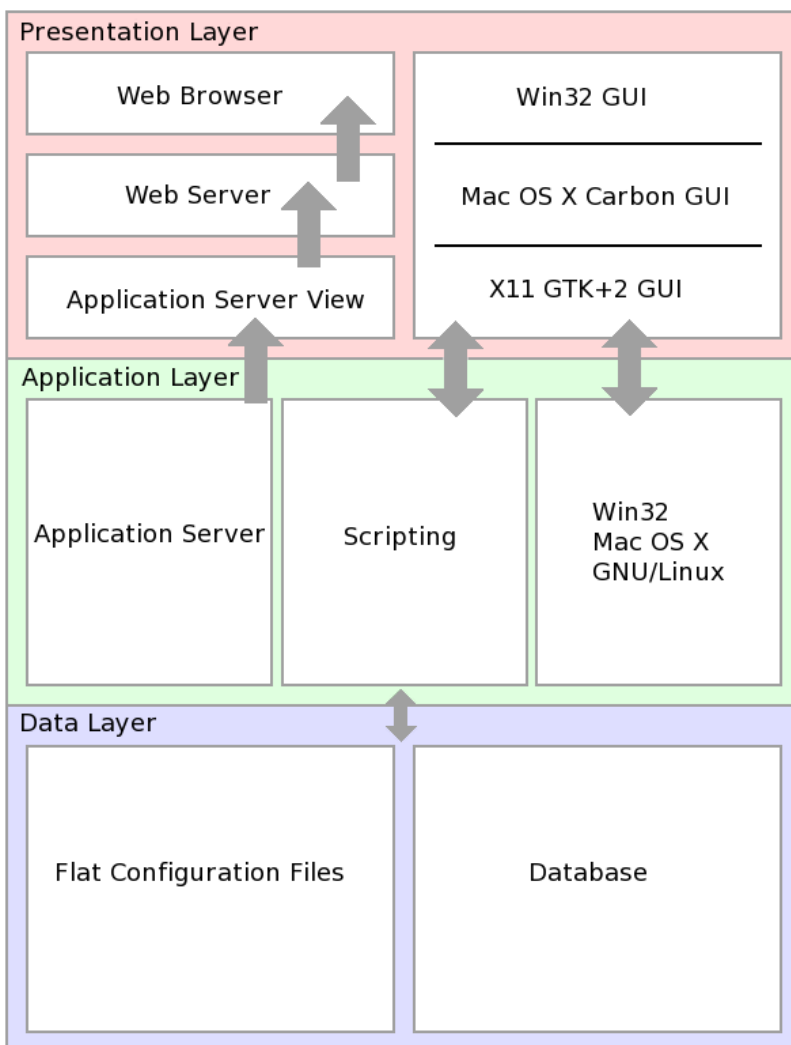
There are two ways that the user can view and modify entries. The journal can be viewed and edited with the GUI of a compiled multi-platform binary client. The journal can also be viewed in hypertext form via a web browser that communicates with a web server and application view.

Application Layer

There are three main components in the application layer. The application server provides processing that is passed to the application views and web server for presentation. The operating system and related tasks are controlled via scripts that the binary client configures and runs. Finally, the binary client is used to manage keywords, categories, and manages the entries.

Database Layer

All journal data and configuration is stored in either the database or in flat configuration files.



Technology Architecture

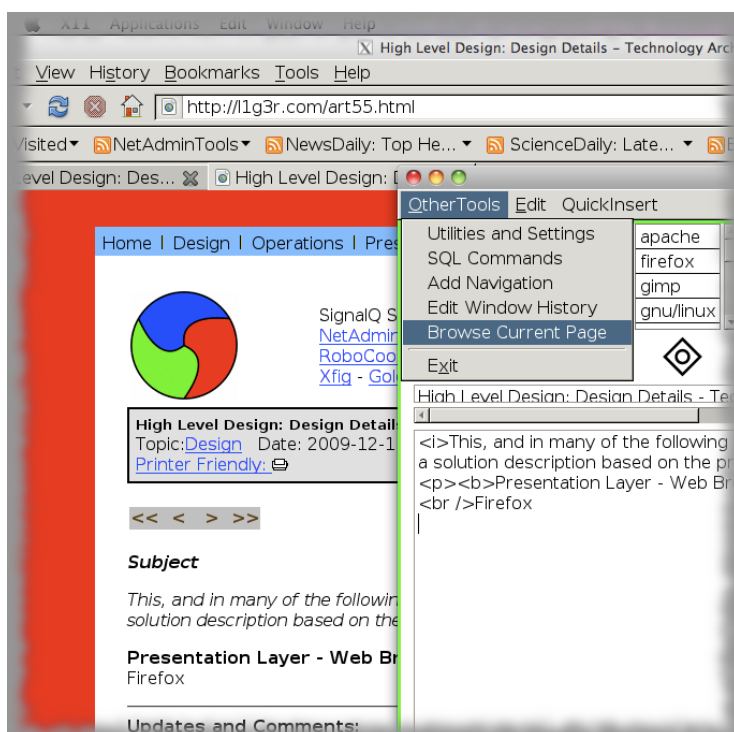
This section and many of the following sections will cross the line from a high level design over to a solution description based on the prototype.

Presentation Layer - Web Browser

Firefox version 3.6.19 is included in the reference OS:



This version successfully builds on the latest version of MCJ OS (3.8), and is reasonably stable and feature rich. The journal should be completely self-contained. It shouldn't need an Internet connection or another computer to run, so including a web browser in the reference OS is a requirement. Launching a web browser from within MCJ via a remote X session on my Mac looks like this:



Presentation Layer - Web and Application Server

Nginx (engine x, is how you say it) is used for the web server. Nginx has tight support for Rails. Originally I used Mongrel for the application server, which had good nginx support; however, when I moved to Rails 3, I moved to Passenger.

Presentation Layer - GUI

The GUI needs to have tight integration with the OS for a variety of reasons. Web applications, because of security and technical constraints, do not have the integration needed to do things like recompile the operating system that the application is running on, or provide slider bars that scroll through configuration files and launch local editors to change them. I imagine that all of these things are possible, but my experience is slight in Javascript/Ajax, and, so, although I doubt that these things are quite there, or should be, technically, I'm not really an expert. MCJ uses either a Win32, Mac OS X Carbon, or X11 GTK+2 GUI. The reference OS includes GTK+2.16.1.

Application Layer - Server

The application server is Ruby on Rails 3. Most of the processing is done by controllers, with minor exceptions in some of the views.

Application Layer - Scripting

Perl, BASH, and Ruby are all used to maintain the system and publish web sites.

Application Layer - Binary Application

A fully compiled, mostly monolithic binary is used to recompile the OS, maintain keywords, and generally configure the system.

Data Layer - Database and Flat Files

The database is Sqlite. There are also flat, text configuration files for a variety of components, but in particular, the Ruby on Rails application is configured via flat text files.

Standards

Presentation

Communication between the web browser and the web server should be via standard HTTP, and the web pages should return validated HTML.

The prototype hypertext presentation uses HTTP 1.1, and the pages are HTML 4.01 Transitional.

Application

A common web development framework and platform should be used, and isolation between the data, application, and presentation layers should be part of the framework. The prototype uses Ruby on Rails 3, which conforms to a model, view, controller (MVC) architecture.

Database

The database should be compatible with most common application frameworks and should allow standard SQL queries. For the prototype, most configuration data and entries are stored in a SQLite database. SQLite is likely the most widely deployed database in the world. SQLite mostly conforms to the SQL-92 SQL database query language standard.

Database Design

This is the schema for MCJ:

ARTS	REALMS	CLASSIFICATIONS	CATS	TIMS
id INTEGER (P)	id INTEGER (P)	id INTEGER (P)	id INTEGER (P)	id INTEGER (P)
title TEXT	name TEXT	name TEXT	cat TEXT	hours FLOAT(2)
classification TEXT	head TEXT	longtitle TEXT	u1 TEXT	u1 TEXT
date DATE	mid TEXT	shorttitle TEXT	u2 TEXT	u2 TEXT
atme TIME	foot TEXT	pagetitle TEXT	u3 TEXT	u3 TEXT
entry TEXT	longtitle TEXT	realm TEXT	u4 TEXT	u4 TEXT
artnum INTEGER	shorttitle TEXT	client TEXT	u5 TEXT	u5 TEXT
realm TEXT	pagetitle TEXT	matter TEXT	u6 TEXT	u6 TEXT
	rectcode TEXT	description TEXT	artnum INTEGER	artnum INTEGER
	prop TEXT	project TEXT		
	style TEXT			

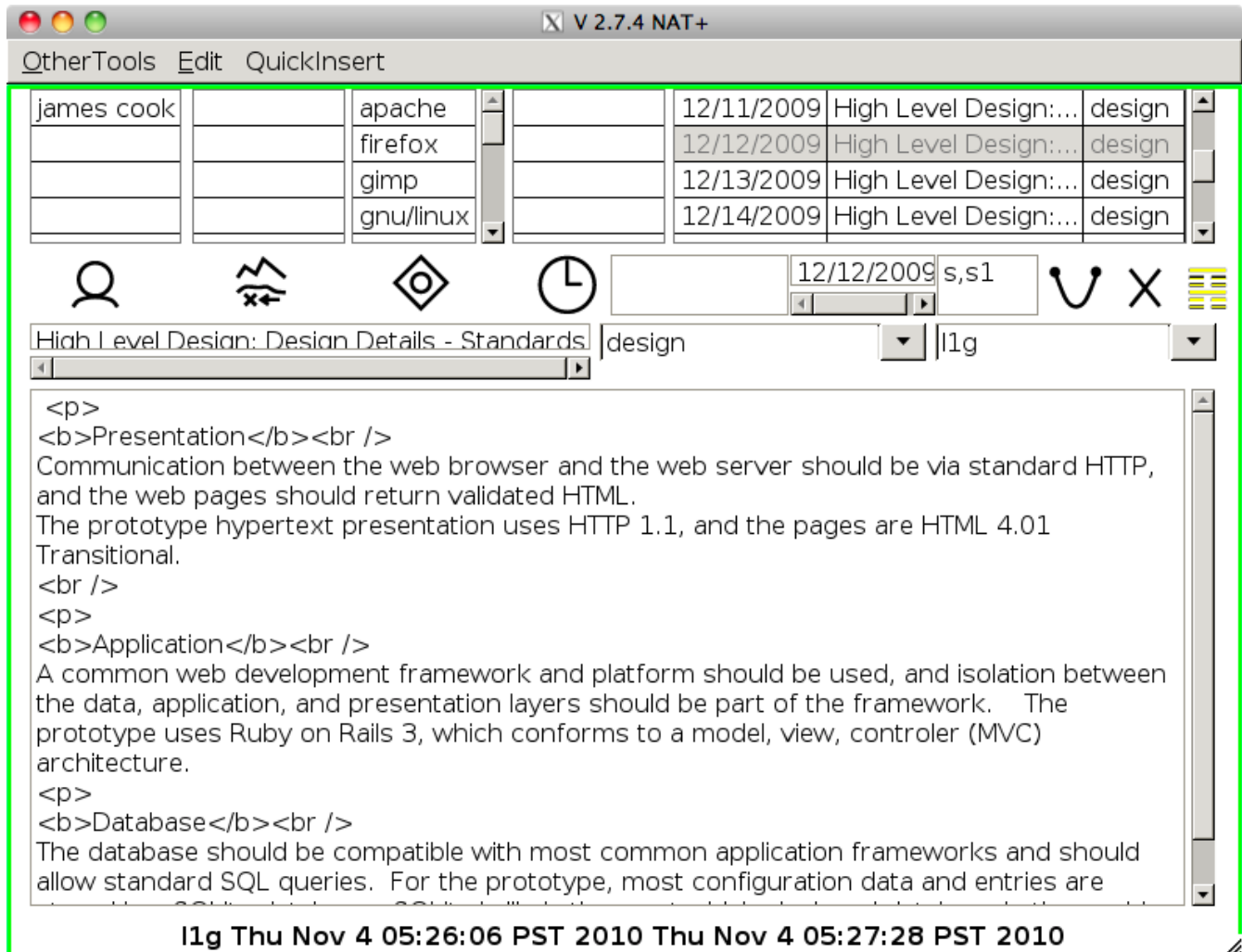
PEOPLES	PLACES	THINGS	TMES	SETTINGS
id INTEGER (P)	id INTEGER (P)	id INTEGER (P)	id INTEGER (P)	id INTEGER (P)
realm TEXT	realm TEXT	realm TEXT	realm TEXT	parameter TEXT
artnum INTEGER	artnum INTEGER	artnum INTEGER	artnum INTEGER	value TEXT
person TEXT	place TEXT	thing TEXT	tme TEXT	

URESOURCES	IMGS	PRESFILES	ACOMMANDS
id INTEGER (P)	id INTEGER (P)	id INTEGER (P)	id INTEGER (P)
realm TEXT	realm TEXT	presfile TEXT	acommand TEXT
artnum INTEGER	artnum INTEGER	prespath TEXT	acommandval TEXT
tag TEXT	tag TEXT	preslongname TEXT	acommandlongname TEXT
file BLOB	file BLOB	presshortname TEXT	acommandshortname TEXT
desc TEXT	alt TEXT	prescolt TEXT	
label TEXT	label TEXT	prescolb TEXT	
classification TEXT	classification TEXT	prescol TEXT	
		presicon TEXT	

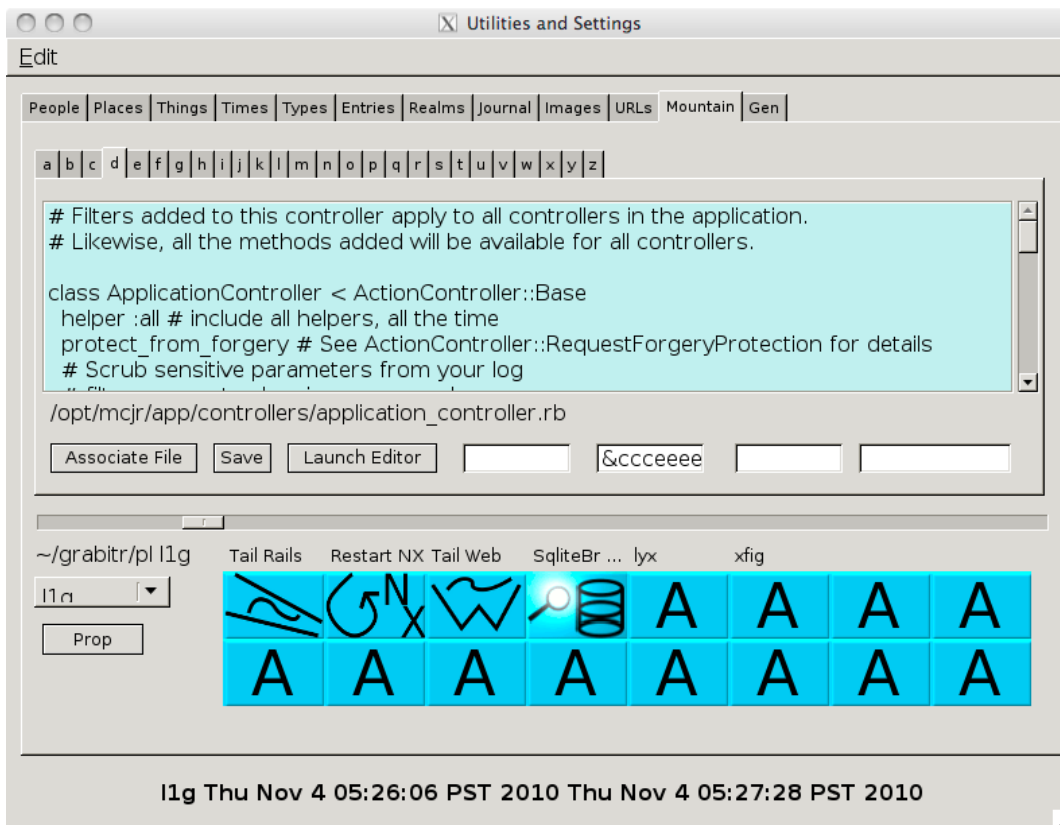
BUILD
out TEXT
artnum INTEGER

User Interface

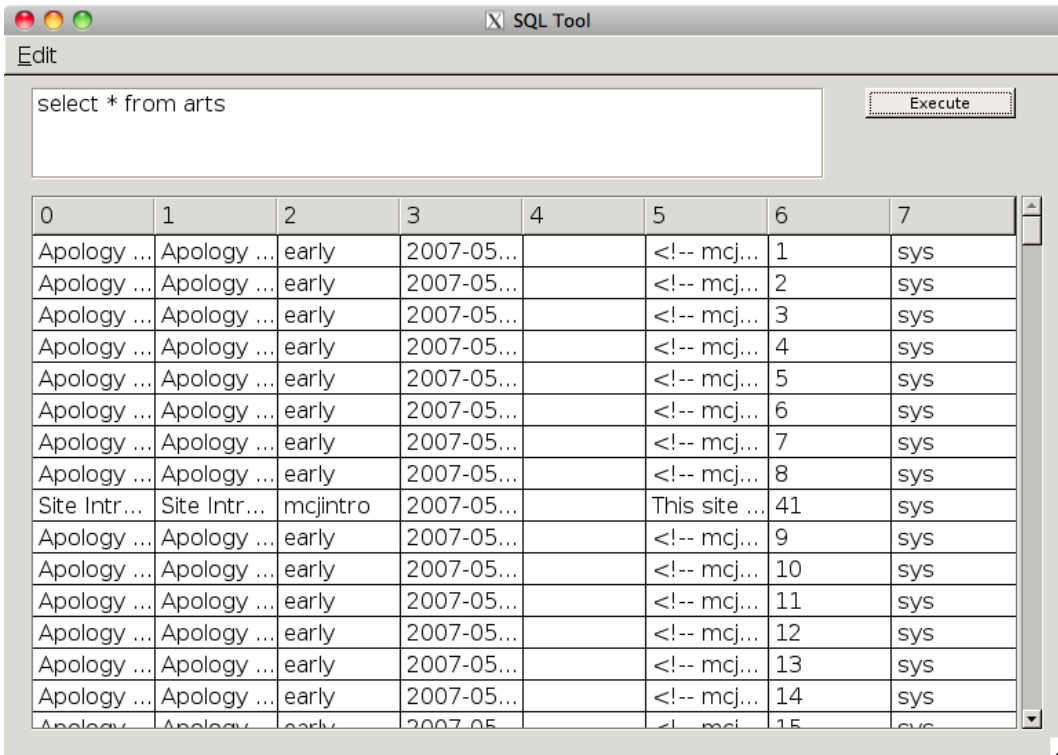
Journal entries are created in the large lower window. Associations with keywords is done with minimal distraction by providing icons on the top canvas. There are list boxes for person, place, thing, and time, directly above the corresponding icons. All other items related to most journal entries are also available in the main window, including date, classification, entry type, realm, title, and sequence information. Here is a screen shot of the main window:



The very top has four list boxes for people, places, things, and times keywords. Clicking on the keywords brings up a list of associated articles in the list box at the the far right. Below that, the icons for the keywords are embedded in the canvas. Just to the right of time, there is a box that lists the current search, the date assigned for the entry, a box for entry type and sequence: (J)ournal, (M)emory, (S)ubject, (D)ream, (C)onversation, and S with the number of the sequence following. There are icons for new, delete, and mountain on the far right. Mountain lists all articles in the list box and saves the current article. The row of boxes just above the main entry box, from left to right, include the title, category of entry, and realm. A realm is a related set of domains that share keywords. The intent is that everything that needs to be set for a typical journal entry is visible on the first layer, in this window, without further navigation. The other main window is Utilities and Settings.

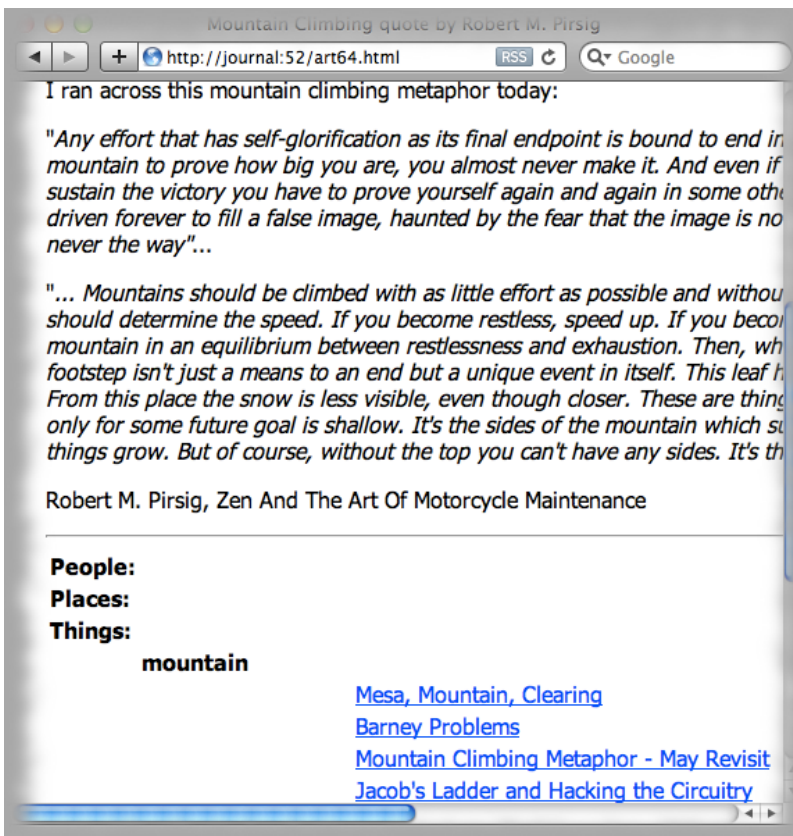


This window has tabs for major areas. Some areas, like the Mountain tab, have another tabbed interface. People, places, things, times, types (categories), and entries allow you to edit the raw entry, including the article number. The realms tab sets various things related to a website, including header information and page titles. The journal tab includes configuration items related to the operation of the journal, including external editor settings, shell commands, spell checker, and other similar items. Images and URLs allow these data types to be stored directly in the MCJ database individually. The design goal, which is not implemented yet in the prototype, is for images and URLs to be marked up individually so they don't clutter the entry with HTML markup. The last tab is used to generate the operating system that runs MCJ. The mountain tab (above) includes a slider bar attached to a tabbed interface to manage configuration files for the presentation layer. This includes web server and Ruby on Rails configuration files. The files can be edited with the configured external editor, and the entry colors can be changed to make groups of files stick out. The shell commands and their associated icons that are set up on the journal tab can be run from this page. In the above screen shot, the Sqlitebrowser icon is being pushed. At the bottom of both the utilities and main window is the status condition of the last web site propagation.

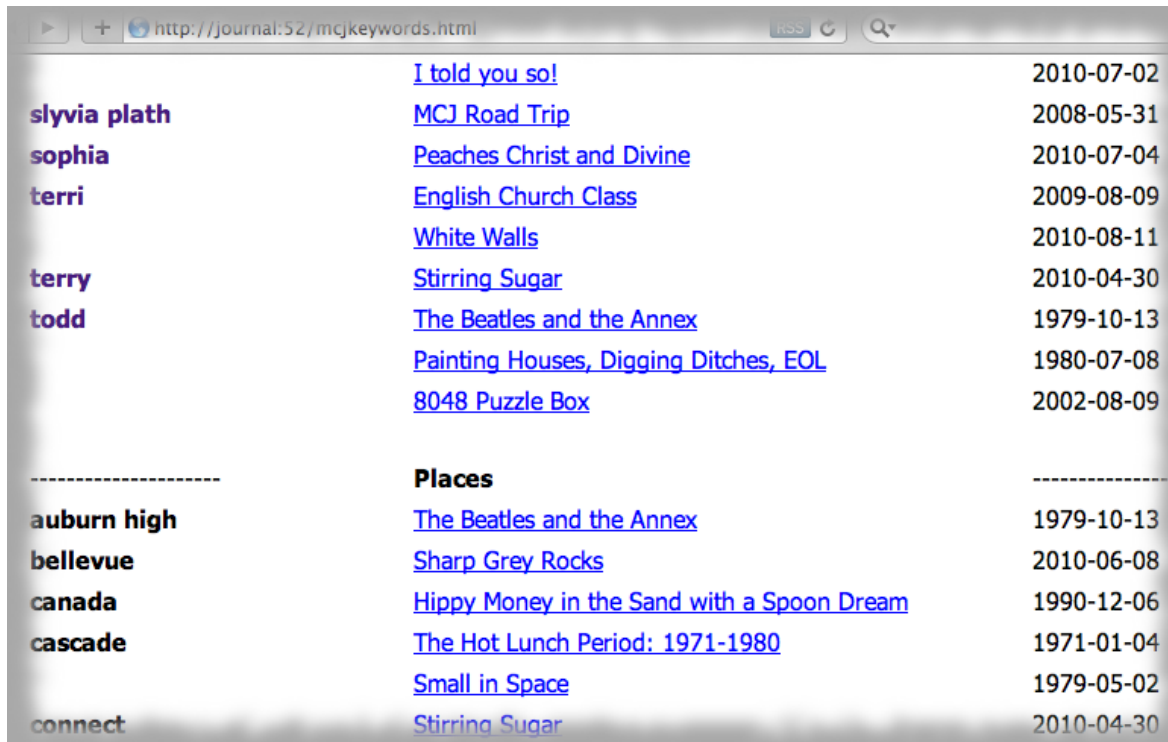


The above SQL Tool window can be used to run SQL commands against the MCJ database.

Besides the binary client executable windows above, another important part of the user interface is the web presentation. Below is an example of a journal entry with the people, places, things and times listed with hyperlinks to other related articles:



Another example, below, is the keywords page, which lists all keywords and associated articles:



	I told you so!	2010-07-02
slyvia plath	MCJ Road Trip	2008-05-31
sophia	Peaches Christ and Divine	2010-07-04
terri	English Church Class	2009-08-09
	White Walls	2010-08-11
terry	Stirring Sugar	2010-04-30
todd	The Beatles and the Annex	1979-10-13
	Painting Houses, Digging Ditches, EOL	1980-07-08
	8048 Puzzle Box	2002-08-09
-----	Places	-----
auburn high	The Beatles and the Annex	1979-10-13
bellevue	Sharp Grey Rocks	2010-06-08
canada	Hippy Money in the Sand with a Spoon Dream	1990-12-06
cascade	The Hot Lunch Period: 1971-1980	1971-01-04
	Small in Space	1979-05-02
connect	Stirring Sugar	2010-04-30

Files

The sqlite database file is named mcj.rsd. This file should stay where you run the application from. There are also key configuration files for the nginx web server and the rails application. All of the key configurations files can be edited from the mountain tab. There is also an option on the entries tab to export the entries to individual text files.

Reports

The main reporting on the entries is via the presentation layer. There are lists of entries over any number of two digit days from 10-99. There are also reports on all keywords used and entries by category. Finally, the included SQL tool can query anything in the database.

Error Handling

The prototype doesn't handle errors very well. One thing that MCJ does do, is that it closes all transactions every time it writes. This cuts down on performance, but that is not a huge consideration when there is just one user. Generally, if there is an error in the prototype, there is a message that pops up, you click OK, and the program terminates. Likely these are failed or nil results, and these should be trapped. The user should be able to choose to store the modified data in the entry window in either the database or a temporary store at the time of failure. This is a difficult design decision. On the one hand, it might not be best to store the data if there is a program error, because this might corrupt the database. If the choice is up to the user, the user might do this.

Interfaces

There is no API available for MCJ; however, the Sqlite database serves this purpose. The presentation layer accesses the information via the Sqlite database (currently read-only). The binary

client application sets up, maintains, and creates this database.

Help

Every option should have help available easily. This is currently not designed or implemented in the prototype. The only help available within the application is on the first entry in the journal, which outlines the operation of MCJ. One solution that comes to mind is if a combination of a hover and key would create context-sensitive help. In honor of WordStar, I plan to implement this in the prototype using F1.

Performance

The graphics and interface should be fast enough to run over a remote X session. The application is CPU and Memory intensive, especially for large journals with more than 500 entries.

Security

One design feature of MCJ is that it can be run completely standalone with now Internet connection whatsoever. It is quite possible to download the tarball for the root file system from Sourceforge, along with the other source packages, and be able to study, build write, and modify the system for years. Although it is possible to run the web presentation as a Ruby on Rails application with Passenger and Nginx serving up web content, it is quite easy to pull those down as static pages. This is how the web content you are reading is presented. Interactive applications are very difficult to secure well over time, and require constant diligence. Static HTML is easier.

Reliability

Almost all data is stored in a Sqlite database file. This has proven to be fairly robust. I've used a single store of data for over five years through development and production, testing code that had bugs and crashed, and never, once, have I had a corrupted database. Sqlite is quite reliable. The store is monolithic, which just means that I can back up the file with a single file copy. Further, I can retrieve it on any platform. The prototype does not qualify as reliable. It does what I need, but I have lost some entries I was working on. The intent is that this is a single-user system, so this level of reliability is acceptable; however, I will improve the prototype over time.

Maintainability

The Ruby on Rails code is quite maintainable, simply because it uses a framework that favors "convention". This makes Ruby on Rails quite maintainable vs. Perl or PHP, because the code pretty much looks the same no matter who is writing it. The REAL Studio code I use for the binary client is not very maintainable. Anybody using this design for their own purposes would likely want a more open development platform. The source code is available, and should be readable; however, most of this code is the first code that worked. I haven't revisited it after I got it working. It is generally not commented, which makes the code difficult to maintain.

Portability

The data and relationships are stored in a very standard database that could easily be moved between applications. Ruby on Rails (RoR) has abstraction between the data (model), application (controller), and view (presentation) baked in, and this helps keep the software portable. The software could be ported at any level. For instance, if it was decided in the future that the entire interface should be web-based, a client could be build that maintained the keywords. RoR is available on most platforms, so moving between platforms is easy as well. Further, the binary client, although not as flexible as RoR, is compiled and available for Mac OS, Microsoft Windows, and GNU/Linux.

Reusability

The Ruby on Rails code used in the presentation and application layers can be modified and adapted easily. The interface to the journal facilitates this with the mountain tab. The source is open and free (GPL), and this helps ensure that the system components are reusable. The biggest drawback with the prototype system is that a closed source compiler is used for the binary application.

Application Compatibility

The application is not compatible with any standard besides the presentation layer. The schema is proprietary. Sqlitebrowser can open up the Sqlite database and review the entries and categories, and this program is included in the reference OS.

Resource Utilization

MCJ takes quite a bit of memory and CPU to publish large sets of journal entries. A minimum requirement would be 1.7 GHz with 1 GB of RAM. It takes several minutes to propagate an entire website with hundreds of entries and keywords using a 3 GHz dual-core processor and 4 GB of RAM. For simple personal journal use, any Intel Mac would be ideal. The default X server is fairly generic, yet will provide decent screen resolution on most systems, so it should be possible to create a new system using commodity hardware that would run just fine for less than \$400 (US) at today's prices. The design goal is that MCJ should run on anything purchased new within the last few years.

Future Considerations

A full open source development environment for the binary client would be preferable. It is unlikely that I will port the code, but perhaps in the future I'll do this.