Building a "Rich Client" or "Internet App" with Runtime Revolution

This is a description of my approach to building a "rich client" or "Internet app" (Internet application) with Revolution. Revolution is a Rapid Application Development (RAD) tool which uses an English-like scripting language and allows cross-platform development and deployment of interactive applications.

New in August 2003: The English Windows PC version of the Lab has been integrated with the Internet! Other versions will be converted in the future. Go to Home page and then to the English download link.

My early prototype can be downloaded at www.reactorlab.net/erc/. Related projects can be found at Fourth World Media's RevNet, Tactile Media's Tactile Media Player, and the Himalayan Academy's Cyber Study Hall. RevNet and the Tactile Media Player require the Revolution development environment.

The goal is a standalone application on a client computer that accesses stacks and media files which are posted on web servers. In this project, local copies of stacks are saved to the client's disk for use off line. Whenever the project is on line and new versions of any of the stacks or files are available on the server, they are automatically downloaded, written over the older copies on the client disk, and opened while the application remains open.

The standalone application file contains only a few lines of "bootstrap" script in addition to the Revolution runtime engine. Thus, every stack and file in the project can be updated to new versions while the application is open, except for the Revolution runtime engine.

If local copies for off-line use are not required, another option would be to install Revolution as a "helper application" in a standard web browser. Then, when a user clicks on a link in a web page that is linked to a Revolution stack, the stack is downloaded and opened. I found that installing helper applications in web browsers was not an easy task and doing so required a different procedure in different browsers. I suspect that many users would not be able to do this successfully. Thus, I would choose to use the method described here even if local copies for off-line user were not required.

For a discussion of the advantages of "Internet apps" relative to web browsers, see Fourth World Media's article "Beyond the Browser".

Engine

The "engine" stack will become the standalone application file. This stack has minimal script - essentially a bootstrap script - with all the rest of the script for the project contained in other stacks. This is done so that all of the scripts in the project can be updated to new versions while the application is open.

The engine stack can optionally have one or more substacks that enable the engine to contact the home server in the event that it becomes separated from its support stacks and files. This discussion assumes that the engine stack has these "default" stacks.

I have called this the "engine stack" because it becomes merged in the standalone application file with the Revolution engine that executes stacks and scripts, even though my engine stack and the Revolution engine are not the same thing.

On startup, the script in the engine stack sets the value of a global variable that contains the file path to the folder in which the engine stack resides (by getting "the long name of this stack" and then parsing it). All

other files in the project will be contained in subfolder "support" of this main folder. The engine standalone file and the support folder and its contents must remain together on the client disk in the same folder. The name of the folder containing the engine standalone file and the support folder can be changed whenever the project is not open. The name of the support folder and the names of its contents must not be changed by the user.

Of course, all names of folders, stacks, and developer-written handlers are arbitrary, as are the folder hierarchies shown.

Then the engine checks to see whether the main support stack "/support/scripts/comm_scripts.mc" is present. If it is, the engine stack sends message "bootstep01" to that stack. If the main support stack is not present, then the engine accesses its default substack copy of this stack.

These two steps - set a global with the path to main folder and send a message to the main support stack - are the only things that the engine's script does.

The handler "bootstep01" in the main support stack (or default copy) puts the stack in use and then checks to see whether the main url library stack "/support/scripts/liburl.mc" is present on disk. If it is, then the stack is put in use. If not, the default substack copy in the engine stack is put in use. Stack liburl.mc is part of the Revolution distribution.

Next the handler checks to see whether the main interface or "directory" stack is present on disk at "/support/scripts/directory.mc". If it is, then that "directory" stack is opened. If not, a simple "connect" stack is generated by script (or could be a substack of the engine stack). The "connect" stack just has a button that contacts the home server and causes the full set of support files on disk to be downloaded, saved to disk, and the new directory stack opened.

Directory

When the directory stack opens, it displays a list of any modules with local copies present on the client's disk.

When the user clicks to go on line, the directory first downloads an "updater" stack from the home server and opens it. The updater stack checks to see whether the versions of the support files on the server are more recent versions than on the client.

Currently, lists of stacks and version information are contained in plain text files. The server has a copy and the local client has a copy (/support/scripts/script_list.txt). Any files which are older, or not present, on the client are updated. An alternative to having this information in a text file would be to execute a script that gets the names of files in a folder and also gets their modification dates.

Updating of a file occurs by the updater stack first downloading and then over-writing the old copy of the file on the client's disk. Then the updater activates the new copy. For example, for stacks that are not open but "in use", this activation occurs by "stop using stack...", then by "delete stack...", then by "start using stack..."

Note that, for stacks in use, the old copy of the stack must be deleted from RAM, even though it may not be open. That is, for a stack that is in use but not open, stop using the stack, delete the stack (from RAM), and then start using the new copy. This is true for a stack in use even if it is open and its destroyStack property set to true and then closed. For an open stack that is also in use: stop using the stack, close the stack, delete the stack (from RAM), then finally open the new copy and put it in use.

The updater should also check if the old version of the stack is in the front or back scripts, and activate the new copy as necessary. An old stack in a front or back script must removed from the front or back scripts,

and then the stack must be deleted from RAM before the new copy of the stack is inserted into the front or back scripts.

After all of the support files, including the directory, have been updated if necessary to the latest version, the updater stack closes itself. The updater stack is not saved to the client disk.

In this manner, every disk file in the project on the client can be updated with the project open, except for the engine standalone application file.

If the engine standalone application file needs to be updated, the updater stack can download the new copy into a separate folder, and then instruct the user to quit the application, replace the old copy of the standalone file with the new copy, and then restart the application. Since the engine stack contains a very minimal script, it would only need to be updated when a new Revolution engine version is made available that contains new features that you want to use in the project.

Modules

The design of the directory stack interface is completely arbitrary and can be changed whenever desired by simply posting the new version on the home server and updating the stack's entry in /support/scripts/script_list.txt.

In the prototype project, the directory has two main elements: an "option" menu button that allows the user to select a server, and a field that lists stacks, which I call "lab modules", that are available on the selected server.

When the user is off line and selects a server, the directory checks to see if there is a local copy of a text file with a list of modules present on the server, "/support/labs/serverName/module_list.txt". If so, the list is displayed. If not, a notice is displayed in the field telling the user they must go on line to see the list.

When the user is on line and selects a server, the directory downloads from the server a text file with a list of modules and their version numbers. If there is a local copy of this text file on the client disk, the directory adds any new modules to the local text file. If there is not a local copy, the text file from the server is written to the client.

Modules that have local copies on the client are denoted as having a local copy both in the module list text file and in the field in the directory.

When a user clicks on a module that is displayed in the directory, the directory checks to see if there is a local copy on the client. If there is a local copy and the user is off line, or if the user is on line and the local copy is the most recent version, the directory opens the local copy. If there is not a local copy and the user is off line, the user is informed that they must go on line to access the module. If there is a local copy and the user is on line but the local copy is not the most recent version, then directory downloads the most recent copy of the module, saves it to disk, updates the module list text file on the client, and then opens the module.

When the user is on line and a module stack is downloaded and opened, the module stack may then download other files from the server. The module stack and its support files are saved to a separate folder inside the server's folder: /support/labs/serverName/module_folder/.

Scripting details

Downloading stacks

The Revolution commands I use are "load url" to download a stack into RAM, and then "go url" to open the stack. The load command is "non blocking" in the sense that the rest of the lines in a script will execute after the load command has been issued and while the load process is continuing. For example, other script lines can update a progress indicator during the download.

For downloading stacks and other files from the server, I modified scripts from the "download stack", a substack in the Revolution distribution's mctools.mc stack. In the Revolution development environment issue the command: go stack "download stack". Then open that stack's script to see the handlers. Alternatively, you can just use a copy of this stack in your project. In addition to "load" and "go url", the script of the download stack also checks the status of the non blocking load process and updates a progress indicator.

You can also inspect Runtime Revolution scripts for downloading stacks. See the script of the button revCommon of stack revLibrary in the Revolution development stacks.

Downloading other files

Currently, I simply use the Revolution command "get url" to download other files from the server. The "get" command is "blocking" in the sense that script execution stops until the get process is completed.

If you want to download a large file that is not to be used immediately, or if you want to update a progress indicator during the download process, you can use the non blocking load command instead.

Files that have been downloaded into a variable can be written to disk with the "put" command. The put command will create a new file or overwrite an existing file.

An example of getting a file from the server and writing it to the client's disk:

```
get url "http://myserver.com/myfiles/myphoto.jpg"
put it into url "binfile:C:/WINDOWS/Desktop/myphoto.jpg"
```

Note the binary file designation "binfile:" that is used with files that do not contain plain text. The plain text file designation is "file:".

Posting information to a CGI script on the server

The "post" command can be used to send information to a Common Gateway Interface (CGI) script on the server for processing. For example:

```
set the httpHeaders to "Content-type: text/plain" & cr
put cr & the time && "hello from stack" && the short name of this stack into tMsg
post tMsg to url "http://myserver.com/cgi-bin/mycgi.mt"
```

The file "mycgi.mt" is a Revolution script file in the cgi-bin directory of server that is executed by a copy of the Revolution engine in the cgi-bin directory. Other CGI languages such as Perl can also be used - but why bother when you know Revolution.

The "post" process requires that the CGI script return a response to the client. The response can include such information as results computed from input information in the message received, data retrieved from a database on the client, or a message saying that the message has been received and recorded to a log file on the server.

Security

Whenever executable scripts are downloaded from the web, the potential exists for a rogue script to cause damage to the client's disk files or to upload private information. This risk can be reduced by restricting access of the project to "trusted" servers, as is done in this project.

One option is to set the secureMode property to true. This prevents access to the disk file system or other resources of the client. Once the secureMode is set to true, it stays true for the rest of the time the Revolution application remains open. This protects the client's disk files but it also prevents saving copies of files to the client disk for use off line.

Another possibility is to add a "certificate of trust" to a stack that certifies that it can be trusted. Such a certificate might consist of a result of the md5Digest() function stored as a custom property of the stack. Set the lockMessages property to true, open the stack, check for a valid certificate and, if the certificate is valid, then send "open messages" or close and reopen stack.

Home | Tools