# Archiving Content and Projects

## Why You Need to Archive Your Content

The traditional publishing workflow can be seen as a linear process in which content moves from one station to another; the content changes hands in a “hand off,” which might be the only time that the two roles talk with each other about the project. (If the hands that are doing the handoffs feel particularly disconnected, they might talk about it as “throwing the manuscript over the wall.”) In that kind of workflow, the final form of the content before being printed is usually the typeset pages, and the archive of the typeset interior is the de facto content archive for the project.

Most ebook production has been based on converting typeset pages to distributable ebooks. This is usually a one-time process that many publishers outsource to the lowest bidder. If there are changes in the content, often these changes don’t get into all editions unless the publisher invests in a manual reprint correction process that covers all bases. So there are often several versions of a successful product, and no one is sure which one is the “right” one. Worse, opportunities to repurpose the product and distribute it in a variety of forms are missed, because the product content is “locked up” in print formats.

We can do a lot better than that; in fact, we need to do better than that, if we are going to publish our content to all of the formats and platforms that are available and still maintain a sense of control over what is going on with each. For anyone who has worked in a busy publishing operation with hundreds of products and many separate editions of the same content sets, it is clear that there has to be a better way.

What if a publisher poured all of their content into a central archive that was indexed and linked, so that all of the content was always available for any purpose that the publisher could imagine, with the assurance that the archived version is the corrected, up-to-date edition of the content? What if conversion to ebook and other new formats was a simple, automatic or mostly automatic process from this archive? What if content changes and corrections could be propagated automatically to all of the places where the content has been published? What if a publisher could make all of their published content available on a new platform simply by defining a single transformation process?

In this chapter I will be describing a set of practices that will enable a publisher to curate all of its content in a central archive, from which every published form of it can be dynamically drawn.

## Server-Based Versioned Archiving

The central content archive is the hub of an XML publishing workflow. All content flows into this hub for permanent curation, and all content flows out of it for publication.

Choosing a Version-Control System

|  |  |  |
| --- | --- | --- |
|  | Subversion | GIT |
| paradigm | centralized | distributed |
| implied workflow | centralized | non-linear, non-centralized |
| commit | one-step, remote | two-step: local commit, remote “push” |
| working copy | any path within repository | whole repository |
| implied repository organization | monolithic: one repository | atomistic: many repositories per server |
| client cmd | svn | git |
| programmable hooks | yes | possible, with difficulty |
| extension languages | several bindings | several bindings |
| WebDAV | yes | possible, with difficulty |

## Subversion: The Recommended Content Archive Platform

Using [Subversion](http://subversion.apache.org) to house content archives on web servers has several advantages:

The “centralized” content and authoring model of Subversion fits the production workflow better than the “distributed” model of git. Git’s model was designed for distributed open-source software projects and is a good fit for individual projects (see “Git: A Good Option for Individual Projects,” below).

* All content changes are automatically versioned, and each version is accompanied by the username of the person who made the change and a descriptive log message.[[1]](#footnote-1) This makes it easy to compare any two versions of the content at any later time, and to see what happened during the course of product development. It is sometimes very valuable or even crucial to be able to do this easily, but it cannot be known in advance when it will be needed. When versioning and log messaging are just part of the daily process, it becomes trivially easy to find such information later as needed.
* The archive, or a small part of it, can be “checked out” to a local working copy, which makes a very convenient and easy-to-learn way of working with the contents of the archive. On our teams, the editors have all gotten proficient at working with Subversion in this way.
* The archive is available as a WebDAV shared filesystem. This means that non-technical users can mount the remote archive to their computer filesystem and edit files as though they were local. In some cases, this can be the most effective way for users to interact with the content. (I once managed the editorial development of a large, multi-year, multi-million dollar project in which the CEO of the publishing company wanted to read and edit the entire manuscript late in the editing process. Rather than saddling myself or someone else with the chore of emailing files around, I set him up with a network share to the manuscript. At any time, he could open the latest committed version in the archive and edited it “in place,” saving his changes as versioned commits but having the exact same user experience as opening a file from the local network.)
* All XML content can be validated when it is “committed” (saved) to the archive, ensuring that the content matches the publisher’s requirements for that kind of content. See [below](#ServerSideXMLContentValidation) for more about this important part of the system.
* All content can be automatically indexed, and published to the web or other places, every time a change is committed to the archive.

Some of these advantages can be achieved by using any server-based version control system. But Subversion is the only platform that allows us to combine all of these advantages at once.

The process of setting up a Subversion server has been [well-documented](http://subversion.apache.org/docs/) [elsewhere](http://svnbook.red-bean.com/); it requires a reasonably-capable server instance running [Apache](http://httpd.apache.org) on any of the major server platforms (Windows, Linux/UNIX, OSX).[[2]](#footnote-2)

One thing worth pointing out with this kind of arrangement is that there is no need to spend any time at all developing a custom content authoring and editing environment. A lot of developers are spending a lot of time and energy doing that, but it’s not necessary. Instead, we can work with our content using normal files on the filesystem, as we have always done, and make use of the full range of software tools that have been developed to work with files in this way. The difference is that these files are backed up to a server and are available to everyone on the team, just as they would be if we had invested in a web-based environment. In other words, we get the benefits of both web-based tools and filesystem-based tools, without the limitations of either. The downside? A thin layer of skill that can be learned by most professionals in a few hours of training.

## Git: A Good Option for Individual Projects

Git is a distributed version control system, which means that every working copy is a full copy of the archive. Distributed version control has become very popular among software developers, because it enables each developer to have a full copy of the repository, for the team to work in a less centralized way. Git’s distributed nature is not ideal for building a publisher’s centralized archive platform — it lacks many of the advantages listed above for using Subversion for this purpose. However, it is very well-suited to individual projects.

If you have a writing project that is in fact distributed among several contributors, or if it is a project that stands alone, you can always begin the project using git, and then bring it into Subversion for production archiving later. For example, I have begun this book project by creating a public git repository on GitHub.[[3]](#footnote-3) This enables other people to contribute sections and then submit them via git’s built-in workflow. It is an appropriate model for an open-source book project like this one. Later, I will probably publish it through an automated workflow connected to a Subversion-based book archive. When that happens, I will bring the book into the Subversion archive.

The nice thing about this approach is that both git and Subversion can be used for a project at the same time. For example, the content development can continue being done via git, and these changes can periodically be pulled into the Subversion archive and incorporated into the production workflow. (Explaining exactly how to do this is beyond the scope of this chapter. A later chapter?) {TODO}

## Archive Layout

We recommend that every publisher set up a single Subversion repository that contains all publisher content. Depending on the size of your publishing operation, this single repository might grow to hundreds of gigabytes over the years; there are no size limitations, and Subversion is remarkably efficient in the storage of content (the server-side archive is often smaller than a checked-out working copy of the same archive, even if the archive contains many separate versions of the same content, because Subversion is so efficient in compressing files and content revisions, even of binary files).

Here are the folders that we recommend as a starting place:

https://my.archiveserver.com

/products/ — all of the content for each product is inside this folder. It usually makes sense to put each product in its own subfolder within the products folder.

Product-One/ — all the files for the first product.

manuscript/ — the product manuscript.

interior/ — the typeset print interior.

cover/ — files for the print and digital cover.

outputs/ — outputs for distribution: interior PDF, digital PDF, ebook formats, app data packages, files for the website, etc.

stylesheets/ — stylesheets specific to this product.

Product-Two/

…

/stylesheets — global cascading stylesheets for the entire product line

/authors/ — information about each author can be put in this folder.

/editors/ — each of the editors can have their own subfolder, in which they can store their manuscripts-in-process and archive them when finished.

The reason we recommend this kind of folder arrangement as a starting point is that it has proven to be a good way to organize everything:

People who are involved in product development for a particular product can access that product folder without needing to check out a copy of the entire archive; this proves important as the archive grows in size and can no longer fit in the working file space of some users’ small hard drives. For example, if the publisher has 1,000 backlist products, but each team is working on 3 products at a time, the members of a team can check out working copies of the three products they are working on, and collaborate on those files in almost-real-time, without taking up hard drive space with the rest of the archive (which might be hundreds of gigabytes by that point).

* You can use access control to limit access by product and by output build or build type to particular users or groups. (For example, a particular app developer might get their own output builds that they have access to, but they shouldn’t access other developers’ builds or the product sources.)
* Shared stylesheets have a folder location (/products/stylesheets) that is named in a way that is analogous to the product-specific stylesheets (/products/Product-Title/stylesheets). This enables the creation of a system that collates a cascade of stylesheets from the global and product-specific folders.
* The same approach can be used for other resources, such as fonts, shared and product-specific media, and so on.

## Interfacing with the Archive

Once you have a Subversion content archive set up on a server, there are several ways to view and edit the content. You can use a Subversion client and check out working copies, or you can mount an archive directory to your filesystem and use it as though it were a local filesystem. Either way, your content files are being backed up on a server every time they are committed, a copy of every version is being saved, and you are using familiar desktop software to work with the files.

For casual users (such as authors or reviewers) who have a much lower level of investment in the “system,” it might be worthwhile to provide a web-based way for them to interact with the content.

## Check out a local working copy of (part of) the archive

The way that we most often work with the content archive is that individual contributors will check out a working copy of one or more product folders to their own computer system. This requires having a Subversion “client”—a program that interfaces with the server—installed on your working machine. There are several good choices here (see [below](#SubversionClients)). Once you have a Subversion client installed, the typical workflow is as follows:

1. Check out a working copy of some part of the archive.You’ll put your working copy into its own subfolder. So, for instance, you might check out https://my.archiveserver.com/products/MyBook to a local folder called MyBook, and inside this folder the files will be a copy of what is on the server.
2. Create new files and “add” them; edit or delete existing files. You can create and edit files as normal. To add a new file to the repository, you have to execute the Subversion “add” command on that file. Similarly, to delete a file from the repository, you have to execute the Subversion “delete” command on that file.
3. “Update” your working copy with changes that others have made. If more than one person is working on a set of files, then you will need to update your local working copy to make sure that you have others’ changes before doing your own work.
4. “Lock” files or otherwise communicate about who is making changes to which files. Subversion includes a locking mechanism, by which users can communicate, “I have this file and am working on it.” This is useful to avoid conflicts (incompatible changes by different users) when a number of people are working on the same set of files, and so many changes are being made that it’s not convenient to communicate about it all the time. In practice, it is often easier simply to communicate with each other about who has which files. (When I was studying to become a private pilot, the instructor said it was always important to know who was in control of the airplane at any time, so we used a hand-off sequence: one would say “You’ve got the controls,” then the other would confirm with “I’ve got the controls.” A similar sequence might be overkill in discussing files; but email makes it very easy to communicate with team members about who has control of a particular set of files at any given time.)
5. “Commit” changes to the repository. Until this point, all changes are “local” on the user’s own system. To get these changes to the archive server, they have to be “committed,” at which point the changes are transmitted and a new revision is stored that contains all of the changes in that “commit.”

## Subversion Clients

### All Platforms: svn

If you are comfortable working with the command line, the svn command is standard across all platforms. It is pretty easy to use (as command-line apps go), and it’s the same everywhere.

Most users, however, haven’t seen a command line since, say, 1991 and will want to use a desktop client for their day-to-day work. Non-technical publishing professionals (editors, designers) are much happier when they can work in a familiar interface.

### Windows: TortoiseSVN

We have a lot of users who are using Windows; the best client by far on Windows is [TortoiseSVN](http://tortoisesvn.net). Tortoise is a Windows Explorer shell extension that adds a bunch of entries to the right-click Explorer context menu, and the choices that are shown are those that are appropriate to the context. Because Tortoise is an Explorer extension, archive actions can be done from within any Explorer window, including a File SaveAs or Open dialog inside an application. This flexibility makes it very easy to integrate archive actions into a Windows workflow.

Unlike TortoiseSVN, every other Subversion client is a standalone application that you work within instead of using the system’s own Explorer/Finder program.

### Mac OS X: Cornerstone

We have been very happy with [Cornerstone](http://www.zennaware.com/cornerstone/) on Mac OSX. It is very stable, has good keyboard shortcuts, and easy to learn.

### Linux GUI: SmartSVN

[SmartSVN](http://www.smartsvn.com/) [seems to be the best current GUI tool](http://svn-ref.assembla.com/linux-svn-client-reviews.html) on the Linux desktop.

### Every Platform: Your Web Browser

A web browser isn’t a client per se, because you can’t edit a repository this way, but at least you can browse the content and see what is there.

## Mount the Archive as a Web Folder

Another option that is especially attractive to the most non-technical users is to mount the Subversion archive as a “web folder.” This is possible because Subversion, when hosted by Apache, uses the WebDAV protocol. On every major desktop platform there are tools available to enable Subversion archives to be mounted to the filesystem.[[4]](#footnote-4)

The advantage of working this way is that, once the archive folder or subfolder is mounted to the system, the user doesn’t need to know anything about working with Subversion; they just work with the files normally as though they were on the local filesystem. For example, we had an executive editor, the CEO of a large publishing company, review and edit all the content for a large complex project through this method. As far as he was concerned, he was opening and editing files in Microsoft Word, and the entire version control infrastructure was transparent to him.

That said, there are a couple of disadvantages to working this way:

Depending on how fast the user’s internet connection is, they might experience a real lag in file operations that are actually taking place over the network.

* There is generally no way for the user to create a custom log message to describe what the commit contains — when they hit “Save”, a new commit is created.

### Windows and macOS: WebDrive

Although macOS has, and Windows used to have, WebDAV support built in, it is much easier and more reliable to install a third-party client called [WebDrive](https://southrivertech.com/products/webdrive/) (<https://southrivertech.com/products/webdrive/>). Once it is installed, you can “mount” the archive address on your system and use it as though it were a local hard drive.

### Linux and UNIX: davfs2

Install davfs2 (it should be available using your package manager, or you can get it from <http://savannah.nongnu.org/projects/davfs2>). Then use the command  
$ mount -t davfs "https://my.server.com/myarchive/path" "/local/path"  
to mount it. (You can also add an entry to the /etc/fstab file in order to automate this process; read the project documentation for details.)

## Web-Based Archive Interface

It is a fairly straightforward exercise for a good web programmer to create a web interface that will browse, display, and even edit the contents of a content archive. Such an interface could be constructed using the following components:

* Mount the archive to the web server filesystem using one of the tools available for that purpose (WebDrive, davfs, etc.).
* Create a web application than responds to requests for paths by examining that filesystem and serving the content that is there.
* For every XML content schema that is supported, the publisher provides an XSLT stylesheet that the web application uses to convert XML content to XHTML on the fly and serve it to the viewer.
* CSS stylesheets, both global and product-specific, are also linked so as to provide a well-designed view of the content.
* A “comment” button on each page can make it possible for reviewers to post comments about the content in each file.
* Optionally, an “edit” link can provide the raw XML in a simple <textarea>, allowing copyeditors to make small changes to the content without having to get a working copy of the product.
* More ambitiously, a “WYSIWYG” editing widget, such as [TinyMCE](http://www.tinymce.com/), can be used, with the caveat that such tools can only work with content that follows the tool’s own interpretation of HTML. Any content that is more technical and precise is liable to be messed up by such tools — let the buyer beware!
* If editing capabilities are going to be provided, then the web developer will want to use the Subversion lock functionality to lock files that are being edited.

That’s the basic shape of what such a system would look like. In our work, we have been satisfied with a system that allows users to view and comment on content files, but not to edit them in the browser — at this point, we haven’t felt that it was worth the investment of time and effort to provide a browser-based editing environment. If we did have a situation like that, then it would make sense to provide browser-based editing.

For example, if I were again starting a large writing and editing project that involved dozens of authors, editors, and reviewers, then I would most definitely take the time to set up browser-based editing of manuscripts, because it would save me from the endless chore of emailing manuscripts to everyone all the time. But I probably would make sure that the content requirements stayed well within the confines of “normal” HTML during this stage of the project, and only apply technical coding to the files after the authors and content reviewers had done their work; and once the technical coding had been applied to the files, I would “turn off” the WYSIWYG editing component for that content, because I would want to avoid any opportunity for the markup to be ruined by a tool that is too clever by half.

There are many specialized “content management systems” available. Most of them require working within a custom interface, usually in a web browser. By comparison, the approach we are outlining here is superior in almost every way for working professionals who are interacting with content every day.

## Archive Automation and Content Auditing

Subversion allows other processes to be “hooked into” the process of committing files to the archive. These “hooks” can occur at several stages of the commit process, but we will focus on the two most common: pre-commit auditing, and post-commit automation.

### Pre-Commit: Content Auditing

A pre-commit hook can be used to verify the format and integrity of the files that are being committed to the archive. This process is called “validation” or “auditing.” (“Validation” is a technical term for verifying the format of XML files; “auditing” is a more general term.) If the content that is being committed does not meet the standards that have been established, the commit can be blocked.

This is a very powerful mechanism: You can ensure that all content that is committed to the archive does in fact meet the standards that you have established for the format and integrity of the content. You only have to program the pre-commit hook to do the auditing/validation that you want, and to return with a non-zero exit status if the committed files do not pass the audit. It is also beneficial to provide a meaningful error message to the user who attempted the commit, so that they can correct the error and commit successfully next time.

For example: Suppose that editors are working in Microsoft Word, but they should only use certain features of Word (paragraph and character styles, tables) and not others (text boxes, bold and italic font formatting). The archive server can be programmed to audit each Word document that is committed, and only allow the commit to succeed if the documents use the allowed features.

### Post-Commit: Process Automation and Notifications

The archive server can also have a post-commit hook, which is a process that runs after the commit completes. A wide variety of tasks can be automated at this point. Three of the most common are automatic file conversion, indexing, and notifications.

Automatic File Conversion. Going back to our previous example, every time an editor commits a Word document to the archive, once the commit has succeeded, the server can automatically convert the document to HTML for use on the web and in ebooks, or to ICML for use in typesetting with InDesign.

Automatic Content Indexing. Similarly, the content in the committed Word document can automatically be added to a full-text search index, so that users can find content related to a particular search term in real time as the content is being developed. Or, this step can be saved until the content is finalized and publish-ready.

Automatic Notifications. One of the most common uses for post-commit hooks is to notify various people when particular files are committed, depending on the rules that you set up. For example, an editor or project manager can be notified whenever files are committed in a project that they are overseeing. Or suppose you license content to app developers, and you keep each developer’s outputs in their own subfolder. The server could automatically notify the app developer whenever files in their outputs are updated, so that they can update their app.

1. Users don’t necessarily see the benefit of these log messages right away, so it requires persistent training to get everyone to write descriptive log messages all the time. [↑](#footnote-ref-1)
2. I recommend setting up the server to use https and “basic” authentication; this is a simple setup that nevertheless provides good security. [↑](#footnote-ref-2)
3. <https://github.com/seanharrison/professional-publishing> [↑](#footnote-ref-3)
4. For this to work, the Apache configuration for the Subversion repository in question has to contain the following line:   
   SVNAutoversioning On [↑](#footnote-ref-4)