Linking the Past with Technology:

Web Based Multimedia Annotation and Linking in the DM Project

A Thesis in Computer Science

by

Timothy Andres

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Bachelor in Arts

With Specialized Honors in Computer Science

May 2014

Abstract

With the advent of cloud based document services, the idea of document sharing and annotation has become well known among typical computer users; however, the types of documents which these sorts of services support have typically been limited to traditional office productivity documents. In order to provide this sort of functionality to users who work with physical documents, the DM Project (formerly Digital Mappaemundi) was created to provide a web-based interface for annotation, linking, and sharing of text and image representations of physical documents. To this end, a single-page web interface was created using HTML 5, CSS, and Javascript, with a RDF data store, canvas viewer and editor, and annotation enabled text editor as its notable technical components. A Django based back-end was created to facilitate the persistence and sharing of the data generated by the users, using the W3C draft Open Annotation and Shared Canvas data models. The actual and potential uses of this system and the software architecture of the system are described in detail in this document by the current lead software developer of the project. This project has been used by numerous scholars in beta test use cases, and is being released as an open source project in the Spring of 2014.

Acknowledgements

The DM Project is a complex piece of software, and its development would not have been possible without the contributions of numerous software developers, designers, and humanities scholars. I'd like to thank Dr. Shannon Bradshaw, the former technical director of the DM Project who first recruited me to the project as a first-year student, and Dr. Martin Foys, the current director of the DM Project. I'd also like to thank Lucy Moss, a Drew University student and currently the only other active developer on the project. I'd also like to thank Adam Ducker and John O'Meara, who have contributed to the project as freelance developers in the past, and Freda Moore, who has contributed to the design of DM's user interface. I'd also like to thank our long-suffering pool of beta testers who have contributed invaluable feedback to the software design process, with special thanks to Catherine Crossley of the British Library for putting the latest iteration of the software through its paces while using it for the Virtual Mappa Project.

The DM Project would also not be feasible without the work of many other scholars, computer scientists, and software developers who have laid the technical and theoretical foundations for the software to be designed and developed. Software development is a deeply collaborative enterprise, and while there are far too many such contributors to name, their work is not unnoticed.

I'd also like to thank my thesis advisor, Dr. Jon Kettenring, and my entire thesis committee for their invaluable advice and feedback throughout the process of writing this thesis.

Table of Contents

Introduction

With the explosive popularity of cloud based document storage, even novice computer users have become familiar with the enormous benefits of storing their work online. Services like Google Docs allow users not only to store their work online, but also to easily collaborate with others in real time. These sorts of services usually also allow users to make their work available to others by sharing it with specific individuals directly, or making it publicly accessible and searchable.

Services such as Google Docs, Microsoft's Office 365, and Apple's iWork for iCloud already do an admirable job of making these sorts of documents available for collaboration and sharing, but they were clearly designed with the modern office in mind. This means that they are inherently limited to word processing documents, presentations, spreadsheets and the like. While this may work nicely for creating new content, and perhaps importing a decade or two of digital productivity documents, it leaves out an unfathomable mass of documents which were created in the days before modern computing using pen and paper, quill and canvas, and even chisel and stone. These documents in their physical forms hold a wealth of information beyond merely the words to be transcribed into a text file. While many scholars may study the transcribable content of these documents, others may be more interested in examining the

handwriting of the scribes to learn about the people who wrote the documents.[1] Others still may be interested in using visible and hidden signs of editing, or even features as simple as notes scrawled in a margin, to infer the thought process of the original writer. In order to make these documents available in a digital format without losing this embedded information, clearly something very different from a traditional word processor is needed.

Thanks to the power of digital photography and ever decreasing cost of digital storage, many traditional repositories of these physical documents have been digitizing these resources as archival quality image files and making them available online.[2] They are sometimes aggregated with specialized images of the documents, such as photographs taken under specific wavelengths of light to reveal hidden writing within the original document,[3] and then assembled into a sequence of images which represent the original document. In the past, the data which connected these images in their proper order with associated metadata was arbitrarily structured by the organization responsible for archiving the documents. The images would then be displayed in some sort of bare bones image viewer, usually just a webpage with forward and backward buttons and

---

[1] Take for example, the Scribal Hands Project (Mooney et al.), actual users of DM

[2] Astle and Muir., "Digitization and preservation in public libraries and archives" references this trend as early as 2002

[3] Shiel et al., "The Ghost in the Manuscript: Hyperspectral Text Recovery and Segmentation" §2

perhaps a table of contents.[4] The lack of a standardized data model has acted as a stumbling block to sharing the documents with software used by other organizations to store and view those documents, which in turn made it more difficult to develop a single software tool to manipulate documents from different sources.

Perhaps even more critically, in the past, there has been no standardized model for representing the data generated by scholars studying these documents, hindering the free flow of academic knowledge. The reported common practice among medievalists, for example, has been to use basic image manipulation software to crop sections of interest out of larger images, perhaps draw markings to indicate features of note, and copy those images into a word processing document with an arbitrary notes structure.[5] Others have even come up with methods involving the use of Excel spreadsheets to track their work.[6] Not only does this sort of system fail to maintain easily followable bi-directional connections between regions of interest and scholarly annotations; it fails to allow other scholars to connect and share their own work on the same documents. Multiple groups could easily spend months working on the same documents independently, only to perhaps eventually stumble upon each other's work after it has been published.

---

[4] See Stanford's Parker on the Web as an example. Sanderson et al. note the ubiquity of this problem in "Shared Canvas: A Collaborative Model for Digital Facsimiles" §1

[5] Foys and Bradshaw, *DM – Annotation to Dissemination* §5B

[6] Kramer, "What Does Digital Humanities Bring to the Table?"

Clearly, this is all far from ideal. One would want to see the majority of archived documents represented in a standardized format so that any software which supported the standard could utilize archived resources from a variety of sources. One would also like to see a standardized model for representing the work of scholars on these resources – flexible enough to be used by a variety of disciplines, yet well defined enough that the data is still processable for purposes like searching.

While data standards are crucial to the creation and sharing of this information, certainly no one expects the majority of these scholars to write out XML files by hand in order to share their work (although such attempts are not unheard of[7]). An intuitive user interface is essential to facilitating the creation, sharing, and publishing of this data. The DM Project was created with this purpose in mind in 2008[8], building a web based user interface for scholars to view and annotate manuscripts, canvases, and transcribed text documents, along with a supporting back end server to centrally store and serve the generated data. Originally called the Digital Mappaemundi project from the latin for “maps of the world,” the project gradually expanded to include more generalized resources like manuscripts, and its name was shortened to “DM”. At

---

[7] Blackwell, from a lecture on the Homer Multitext Project, whose contributors actually hand write standardized XML transcriptions

[8] Foys and Bradshaw, “Developing Digital Mappaemundi”

this time, the project currently has over sixty humanities scholars as beta users,[9] all generating data which can be exported in standardized formats to be shared with the world. These scholars have already used DM to annotate maps, scrolls, manuscripts, and even images of archaeological digs, and they are currently sharing their work through DM with their colleagues, students, and readers.

While the DM Project has been designed with the use case of medieval scholarship in mind, there is nothing in the interface of DM nor the data models it relies upon which restricts its use to those disciplines. In fact, the annotation tools which DM provides could be of use to practically any discipline whose practice involves the study of some sort of text-based or image-based data. Just as easily as DM can be used as a tool for medieval scholars to annotate ancient manuscripts, it could be used by a biologist to annotate microscopic images of cell cultures, a geologist to mark up images of strata, or even a medical doctor to make a few notes on an X-ray image. Each of these disciplines would likely benefit from some specialization of the interface and data to handle their specific use cases, but both DM and the data models it uses have been built with extensibility in mind. Furthermore, DM is being released as open source software, meaning that any developer can build upon DM's extensive codebase to build software which fits a user group's needs, and any institution can set up its own private instance of the tool on its own servers.

---

[9] Based on the number of unique user accounts on the most recent and second most recent iterations of DM hosted at Drew University.

My personal contribution to the DM Project began in early 2011. I initially focused on building pieces of the user interface, and gradually expanded my work to include both client-side and server-side data storage and manipulation. I have written significant portions of the user interface, including the entirety of its Canvas Viewer, as well as a Javascript based data store and manipulation framework, and have worked in conjunction with other DM Project contributors to build the back end web service which facilitates synchronization and sharing of the user generated data. I currently serve as the lead developer on the project, maintaining a previous release used by nearly sixty users, and developing the newest iteration of the software currently being alpha tested at the British Library. In this document, I will note pieces of the software which I have implemented for the purpose of describing my personal contribution to the project and its merit for an honors thesis, but I must emphasize that the development of the DM Project would not have been possible without the significant contributions of many other developers, to whom I am extremely grateful.

As the development stage of the DM Project comes to a close, it is being prepared for an official open source release to the general public. This will allow any institution to set up its own instance of DM to manage as it chooses, hosting its own data, controlling user accounts, and if it so chooses, allowing public access to the work its users create. It will also allow other teams to utilize pieces of DM's codebase, such as its extensive Javascript RDF data storage

and manipulation libraries, in other software projects with similar goals. In this way, DM will not just be limited to select institutions or niche academic projects; it will remain accessible to anyone with the resources and drive to use it. Those interested in examining DM's codebase or installing a standalone instance of the software can find the latest version at http://github.com/timandres/DM/.

Software Architecture Overview

The DM Project consists of two main software components: a web based user interface (UI) built using HTML5, Javascript, and CSS technologies, and a server-side web service built using the Django framework which stores and serves up data from persistent databases (DBs) based on the web interface's queries. These two components work in unison, communicating over the standard HTTP protocol, to provide a sophisticated UI backed by sharable cloud data storage, making both the DM software and the projects its users create available from anywhere with an internet connection. The figure below provides a high level overview of the interaction between these components, which will be discussed in greater detail in subsequent chapters.

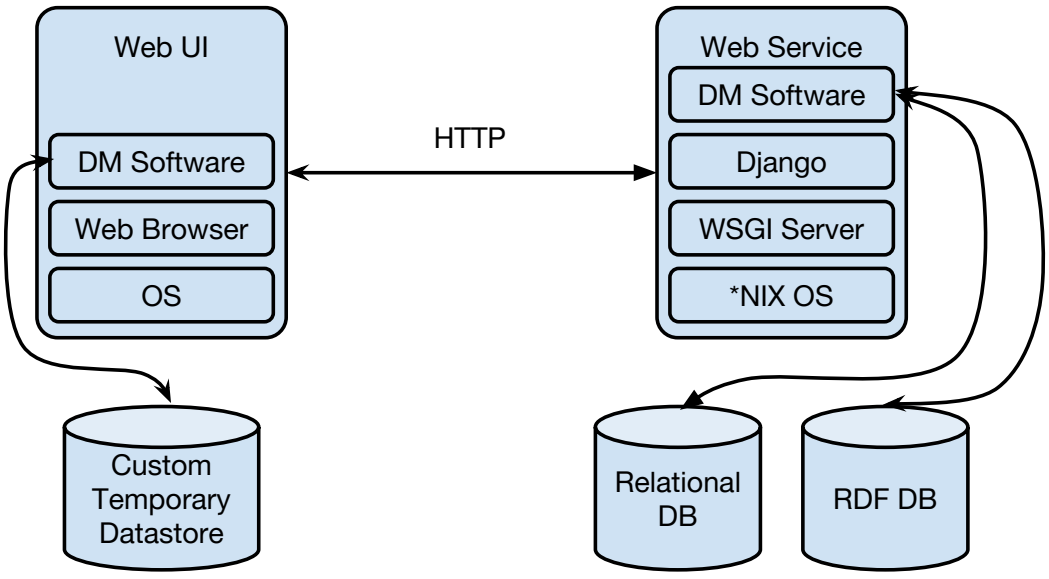


Figure 1: A high level overview of the DM software system

For the sake of those not familiar with the intricacies of web applications, consider this high level overview of a typical user interaction with DM. Users will always start by loading the UI in their browser by navigating to a DM server on the web, eg., http://dm.drew.edu. This causes the server to provide the browser with all the necessary files to render the UI. After the users log in with their usernames and passwords, the web service returns metadata on all the projects to which they have access respectively. The browser requests data on the last opened project from the web service in order to render an overview of that project's contents for the user. As the user opens individual resources within the project, the browser requests additional data from the web service at each step. As the user makes changes to the project, such as adding annotations, that data is temporarily stored within DM's browser side data store, then sent back to the web service for permanent storage on a regular basis (with the web service ensuring that only authorized users can make changes to the project). This communication strategy between the browser and web service, common to most rich web applications, ensures that data is efficiently transferred between the user's computer and the server.

The User Interface

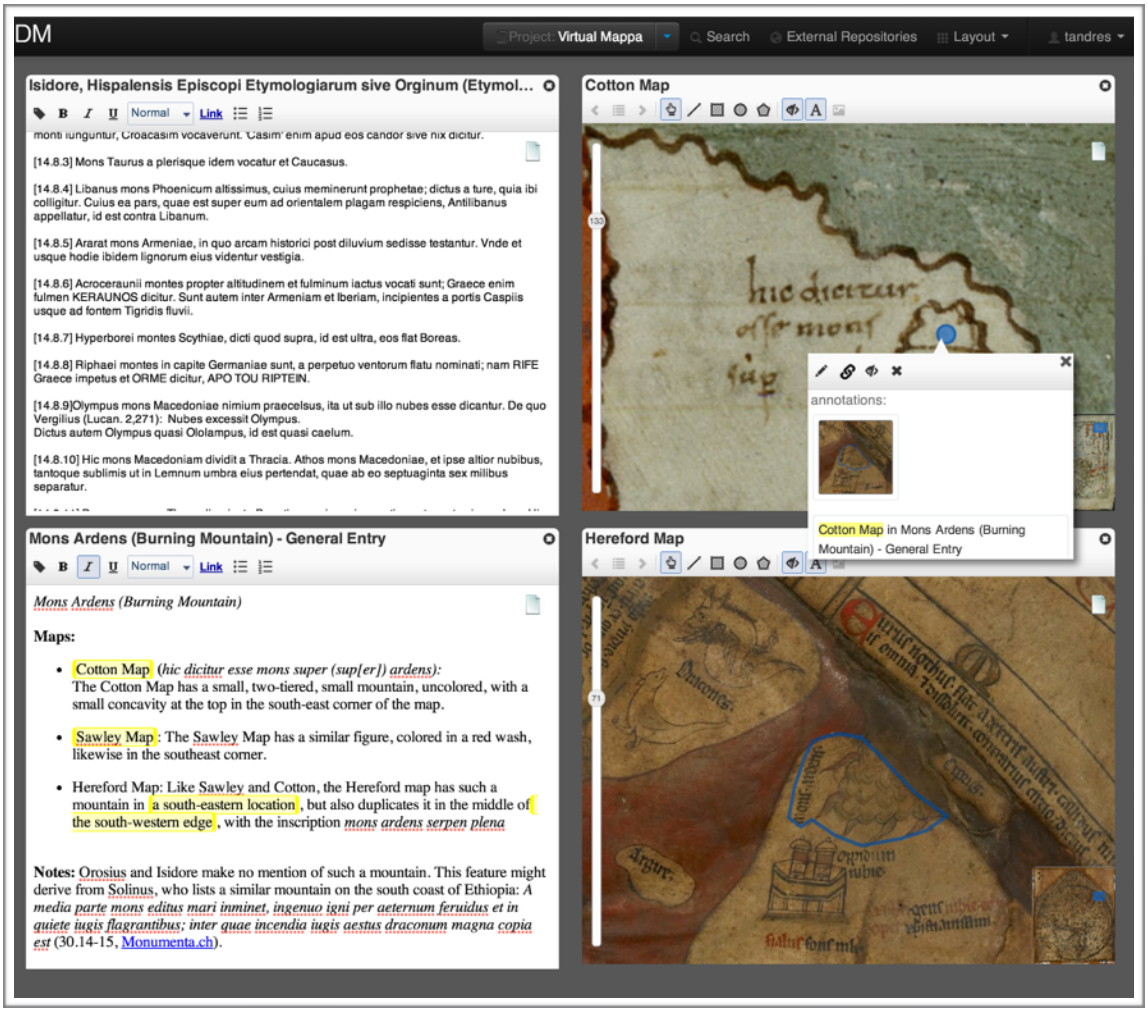


Figure 2: The DM User Interface in Action. Two text documents are shown on the left, the top one showing an excerpt of a transcription of Isidore, and the bottom one showing a user generated annotation on the canvases to the right, complete with dynamically linked highlight annotations in yellow. On the right, two canvases of medieval maps are shown zoomed in to areas related to "mons ardens" (which translates to burning mountains, or volcanoes). The white menu emanating from the top canvas was caused to appear by the user hovering their mouse over the annotated region of interest dot, showing a scrollable list of all annotations linked to that region of the canvas, including text documents and other canvas regions.

Creating a usable user interface which allows viewing, editing, annotation, and linking of images and text documents, all while allowing cross-platform online collaboration is no small task. Since sharing and collaboration are central goals of the DM Project, it was clear from a very early stage that a web based tool was the best option. This would allow data to be stored on a centralized server, allowing users to collaborate on the same project, and eventually publish it to the web. The choice to make DM web based also freed users from having to worry about storing applications and specialized files on their computers to work on projects. Instead, DM allows them to simply remember a web address and login information.

The DM development team knew from the start that implementing such a complex tool in a web environment would be challenging. Early prototypes of the project were built in Macromedia Flash (now Adobe Flash), but the team quickly migrated the project to open web standards like HTML5 as it became clear that modern browsers could support the necessary functionality to display and edit images and text documents. This provided both the benefit of being more conducive to open sourcing the codebase of the project (as the Javascript community is very open source oriented), and the unforeseen benefit of being more compatible with mobile devices, which often support the Flash plugin either poorly or not at all.

Since most of the target users for DM have a workflow that revolves around original documents which are represented by archival images, the first task was to create an image viewer that could display digital representations of these documents. Because the images representing these documents, or the surrogates for the original documents as scholars prefer to call them,[10] are often large and detailed, panning and zooming functionality is an absolute requirement for any sort of image viewer. Early prototypes of DM only allowed for points to be placed on these images, severely limiting the types of features which users could annotate. To annotate specific sections of these documents, users must be able to select specific portions of these images in order to annotate them, meaning that they must be able to non-destructively mark out regions of interest as abstract shapes on the images.

In order to make annotations with any sort of semantic content, it is of course also necessary to have some basic way of entering text into the interface. But at the same time, many users also work with completely transcribed versions of the resources they study,[11] meaning that a text editor with formatting and linking functionality is also needed. Because these text documents can also be extensive objects of study themselves, there is a need to allow users to select regions of interest within the texts for annotation and linking.

---

[10] Kropf, “Will That Surrogate Do?”

[11] Shannon Bradshaw, Private Conversation

Just as it would be necessary for scholars to have both the documents being studied and their own notes in front of them when working with physical media, it is also necessary for DM's interface to allow scholars to view multiple documents at the same time for the purposes of comparison and annotation. This means that DM must also implement an interface analogous to the window system of modern graphical operating systems, allowing users to open multiple resources at once and rearrange them into an optimal configuration.

From a technical perspective, nearly all of this user interface would have to be implemented in Javascript, with supporting CSS files to define styling. A limited number of HTML files were used to define the overall layout, but the vast majority of the user interface objects in DM's codebase define their layout entirely in Javascript by using the Javascript's HTML Document Object Model (DOM) API. The Google Closure library and compiler are used as a framework to maintain the multitude of Javascript classes and modules in DM's codebase, as well as for its vast array of user interface objects and utility functions. The ubiquitous jQuery library is also interspersed into the codebase where its use was deemed efficient.

The Canvas Viewer

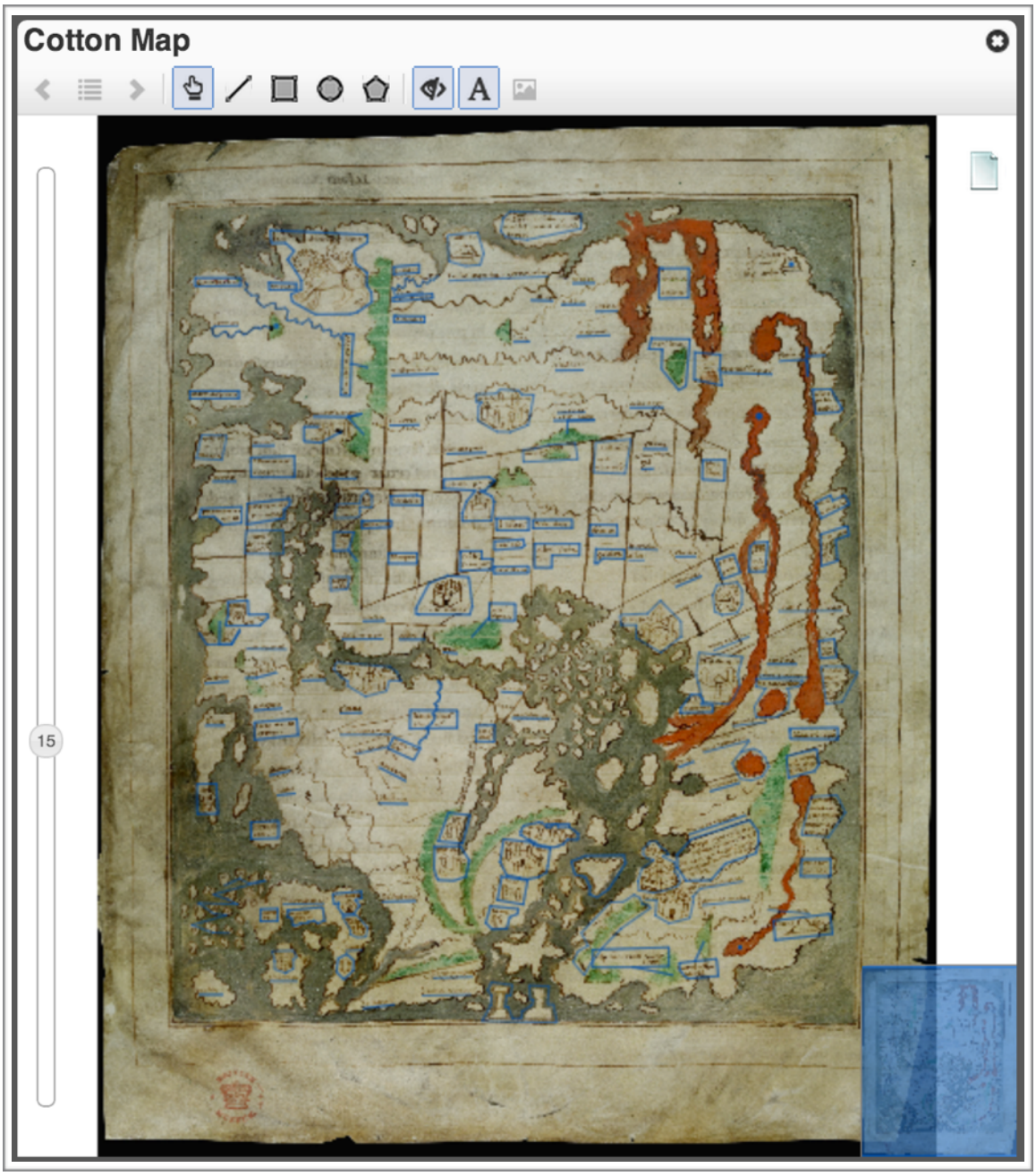


Figure 3: DM's Canvas Viewer showing a wide view of a medieval map, annotated with many regions of interest. A toolbar for navigation, selection, and drawing is shown at the top. A slider on the left controls the zoom level, with 15% zoom indicated by the "15" in the slider. A marquee view on the right with a smaller image of the canvas entirely covered in a blue selection box indicates that the entire canvas is being viewed at once.

The Canvas Viewer serves as DM's tool for image viewing and annotation. Because the images of the documents DM's users study are often large and contain intricate details, the ability to zoom in and out on those images within the browser is absolutely critical to a usable product. It must also allow users to select regions of interest on those documents for the purpose of annotation by drawing bounding shapes around them (without actually modifying the image being studied). Furthermore, because many images of a resource under study may exist,[12] it must also support the interchange of those images while preserving the annotation data overlaid onto them. These core requirements essentially amount to a lightweight non-destructive image editor, complete with the ability to draw free-form vector shapes and rescale images, all within the environment of a web browser.

In the first iteration of the Canvas Viewer's development, the OpenLayers mapping library was used as a base for development. The OpenLayers library was developed as an open source browser based map viewing tool, complete with support for tiling images for the purpose of panning and zooming across large documents, and built-in drawing tools. Having these features already built was obviously tantalizing to the DM development team with its limited resources. However, as the development of the Canvas Viewer progressed, it quickly became clear that the OpenLayers library was actually far too extensive for the needs of the project. While it comes with many tools needed to build the

---

[12] Sanderson and Albritton, *Shared Canvas Data Model §2*

Canvas Viewer already included, it brought with it a highly complex architecture designed to compensate for the geometric intricacies of precise mapping (not usually encountered in ancient maps and manuscripts). This meant that unused code from the OpenLayers codebase was being included in the project, and that development efforts were being slowed by the complexity of the OpenLayers architecture.

In an effort to make the codebase of the project more lightweight and to speed development, the decision was made to rewrite the Canvas Viewer using a less substantial Javascript library. Because the data standards adopted by DM borrow upon the Scalable Vector Graphics (SVG) specification for the description of regions of interest on canvases,[13] it seemed a natural choice to use a Javascript SVG library to implement the Canvas Viewer. The team then settled upon Raphael JS as the ideal candidate for such an implementation, based on its simple Javascript API for manipulating SVG documents. Although this Canvas Viewer implementation is not the one currently used in the DM codebase, the process of its development is still informative toward the design choices made in the current implementation.

Using the Raphael JS library, I implemented a new Canvas Viewer from the ground up. All canvases were treated as SVG documents, in which the background image was rendered as the base layer, and regions of interest

---

[13] Sanderson et al., *Open Annotation Data Model* §3.2.3.1

drawn by the users were represented as real SVG objects within the browser's Document Object Model (DOM). This provided a far simpler system for development purposes than the cumbersome Open Layers library, while still providing many of the features necessary for a viable Canvas Viewer.

The first task was to implement the panning and zooming functionality of the Canvas Viewer. Fortunately, the SVG standard includes the concept of a "view box," which acts as a sort of scalable and moveable window onto a larger document.[14] This meant that no explicit conversion between pixels on the screen relative to pixels on the larger canvas was necessary, as the browser's implementation of the view box abstracted that conversion away. This greatly simplified the implementation of panning and zooming, allowing me to simply implement event handling for mouse panning and zooming events to adjust the view box property of the SVG document displayed in the browser, rather than manually rescaling and repositioning the image and all of the regions of interest drawn onto it.

Secondly, I had to implement the region of interest drawing functionality, along with the ability to recognize which regions of interest the user is trying to select with the mouse when clicking on the canvas. Unlike the OpenLayers library, Raphael JS did not include prebuilt tools to allow users to draw shapes and freeform markings onto SVG documents, so I was tasked with creating tools

---

[14] *SVG 1.1* §7.7

to transform a user's clicks on the canvas into SVG descriptions of shapes to render. Fortunately, because the SVG shapes generated were live DOM objects, the task of recognizing which regions of interest were being selected was as simple as adding standard event listeners to those objects for mouse events.[15]

While the Raphael JS library proved overall very convenient from a development standpoint for implementing the core functionality of the Canvas Viewer, it quickly became clear as it was tested in the development phase against the hundreds of regions of interest that users had drawn on sample canvases that its performance decreased significantly with scale. Basic operations like panning and zooming bogged down to completely unusable speeds as these regions of interest were rendered on the images. Once again, the Canvas Viewer would have to be reimplemented using a new graphics library.

In search of better performance, I turned to the new HTML5 Canvas API. Unlike SVG, which is a verbose specification for vector shapes, HTML5's Canvas API works by allowing a client to draw images at a bitmap level, i.e., nearly pixel by pixel.[16] This allows for very good performance, but its standardized API is highly tedious, only implementing basic operations like moving to points and drawing geometric shapes. Even more tedious, the HTML5

---

[15] *SVG 1.1* §16.1

[16] "The Canvas Element." *HTML Living Standard*. §4.12.4.2

Canvas API does not provide anything like HTML5's DOM, leaving it up to the client to maintain data about where shapes are located for the purposes of handling mouse events and redrawing the canvas as objects on the canvas are moved. It is well known that the HTML5 Canvas API provides excellent performance thanks to hardware accelerated browser implementations,[17] but implementing such a complex tool as the Canvas Viewer using only the standard API would almost certainly not have been possible given the DM development team's limited resources.

On my recommendation, the DM development team turned to a Javascript library under active development called FabricJS. FabricJS is a powerful wrapper for the HTML5 Canvas standard, providing a Javascript object model for the shapes and images drawn on the canvas analogous to the DOM provided by SVG.[18] It also provides SVG parsing tools, which are particularly useful due to DM's region of interest data being described in SVG, and even some rudimentary drawing and editing tools. However, it has no analog to SVG's view box concept for panning and zooming, meaning that DM's Canvas Viewer has to implement the scaling and moving of the entire canvas, regions of interest and all, on its own.

---

[17] "Unleash the Power of Hardware-Accelerated HTML5 Canvas." Microsoft Developer Network.

[18] FabricJS `fabric.Object`

I undertook the task of reimplementing the Canvas Viewer using the FabricJS library. I had already architected a fairly robust system utilizing the Raphael JS library, and I found it required very few changes to adapt to the FabricJS API. The core of this architecture is the Canvas class (`sc.canvas.Canvas`), which maintains the representations of the images of a particular canvas and the regions of interest drawn on that canvas as FabricJS objects. (Note that throughout this document *Canvas object* will refer to Javascript objects defined by a class in DM's codebase, while *canvas* will refer to the abstract concept of a visual resource under study as defined by the Shared Canvas Data Model.) These FabricJS objects are indexed by their unique identifiers, and stored in an ordered list to define the z-ordering of the objects on the canvas[19]. By indexing references to these objects, and keeping record of their position on the screen, the Canvas object can provide enough data to determine with the help of the FabricJS library which object the user selects with either a mouse or a touchscreen, allowing annotation tools to use this information.

The Canvas object also maintains the ratio of the size (in pixels) at which it is displayed on screen to the size at which it is represented in the data. For example, if the canvas is being shown at a size of 500×500 pixels on screen,

---

[19] The z-ordering, i.e. the order in which objects are rendered from bottom to top on the z-axis, is not necessarily specified by the data (in accordance with the Shared Canvas Data Model), but a user may need to change the order in which the regions of interest appear to access a particular region.

while it is actually 5,000×5,000 pixels in size, then it would have a ratio of 1/10. This would then mean that all FabricJS objects on the canvas are being rendered at one tenth of their actual size and coordinates, which the Canvas object would compensate for when outputting data to be saved. By maintaining this ratio, and controlling all access to objects on the canvas, the Canvas object can allow canvases to be resized on the user's display while still maintaining the integrity of the underlying data. By also maintaining the positions of the resized canvas on the user's screen, and compensating geometrically for the difference, the Canvas object can also support being repositioned as it is scaled on the user's screen.

DM's Canvas objects are rendered within the context of a Canvas Viewport object (`sc.canvas.CanvasViewport`), which is responsible for managing the position and scaling of the Canvas object. The Canvas Viewport object owns the HTML5 canvas element in which the DM canvas is rendered, and is responsible for re-rendering all of the objects owned by its Canvas object on the HTML5 canvas as the canvas is modified, panned, and zoomed. By centralizing the final responsibility for rendering graphics, the Canvas Viewport object can provide extra efficiency by throttling requests to render new HTML5 canvas frames and allowing rendering to be paused and resumed when making many changes to the Canvas object in bulk.

The Canvas Viewport object also effectively performs the job of the SVG view box, allowing the canvas to be scaled and panned on the user's screen. It provides numerous methods for rescaling and repositioning the canvas to respond to user actions like panning and zooming with a mouse, as well as utility methods for zooming into specific regions of interest when following a trail of annotation links. Like SVG's view box, the Canvas Viewport object also is responsible for translating coordinates as displayed on the user's screen into coordinates relative to the canvas data by geometric calculations. It also builds upon FabricJS's library to translate mouse events over given points on the HTML5 canvas into events on specific regions of interest on the canvas being displayed. All of this information is made easily accessible to other pieces of code using the Canvas Viewport through an event listening system much like that provided natively by the browser for SVG documents. For example, a client of the Canvas Viewport object can register an event handler for any mouseover events on the viewport, and that handler will then be provided with the canvas coordinates and URI identifiers of any resources which the user mouses over from that point onward.

While the Canvas Viewport object maintains responsibility for repositioning and scaling canvases on the user's screen, responsibility for interpreting user input from mouse and touch devices is delegated to objects which implement the Canvas Viewport Control interface (`sc.canvas.Control`). Such Control objects are instantiated with a reference to a Canvas Viewport

object, and they can interact with the Canvas Viewport and the Canvas object it owns through any of their public methods. This provides a clean separation of the display logic of the Canvas Viewport and Canvas objects from the user interaction logic of interpreting mouse, touchscreen, and keyboard events. This architecture was used to implement the user interfaces for panning and zooming functionality, as well as region of interest selection.

The core panning and zooming user interface responsibilities are delegated to the Pan Zoom Control (`sc.canvas.PanZoomControl`) and Zoom Slider Control (`sc.canvas.ZoomSliderControl`) objects, which inherit from the base Canvas Viewport Control class. The Pan Zoom Control object allows for an interface that will be familiar to anyone who has ever used an online mapping service like Google Maps. The Pan Zoom Control monitors clicking and dragging events on the Canvas Viewport, panning the canvas inside the Canvas Viewport as users drag the mouse using the public API of the Canvas Viewport class. It also monitors for double click and mouse wheel (scroll wheel or trackpad scroll) events, instructing the Canvas Viewport to zoom in on the indicated area accordingly. The Zoom Slider control shown in Figure 4 provides a visible slider element which the user may drag with the mouse to zoom in and out on the image. The position of the slider is kept up to date by subscribing to “bounds changed” events on the Canvas Viewport being managed. Although no multitouch specific controls were designed for the Canvas Viewer due to time constraints, the Canvas Viewport Control architecture would easily allow such

controls to be implemented and substituted at runtime for the standard mouse controls on a multitouch device.

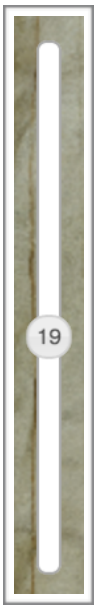


Figure 4: The Zoom Slider. The "19" within the center circle indicates that the canvas is zoomed out to 19% of its actual size. Moving the slider up zooms in, and moving the slider down zooms out. The slider position and labeling stays synchronized as the user zooms in via other means.

The more complicated issue of selecting regions of interest is also solved using the Canvas Viewport Control interface. A base class called Feature Control (`sc.canvas.FeatureControl`) handles the saving of regions of interest, also referred to as features when they are drawn onto a canvas, to the Databroker object. The Draw Feature Control class subclasses the Feature Control class to provide utility methods for drawing the features onto the canvas. Individual controls for drawing regions of interest using geometric and freeform shapes subclass the Draw Feature Control class (`sc.canvas.DrawFeatureControl`). The implementation of these controls uses the mouse events fired by the

Canvas Viewport object to create and update FabricJS objects on the underlying Canvas Object in realtime.

Instances of all of these classes are unified under a single Canvas Viewer object, which owns the Canvas Viewport, the Canvas Viewport Controls, and the underlying Canvas object. The Canvas Viewer class is responsible for defining the layout of the user interface, including a toolbar with buttons for activating drawing and selection tools, the position of the zoom slider, and a miniature "marquee" overview which indicates the section of the canvas on which the user has zoomed in relative to the entire canvas. In order to allow the Canvas Viewer to be used outside of the DM environment, for example as an embedded feature in a webpage, there are actually two classes called Canvas Viewer in the DM codebase. The first, called `sc.canvas.CanvasViewer`, implements all of the functionality described above, while the second, called `atb.viewer.CanvasViewer`, is a thin wrapper around the afore mentioned Canvas Viewer which integrates it with DM's global event handling system and Viewer Grid system (both of which will be discussed in more detail subsequently). Unless otherwise specified, it should be assumed that "Canvas Viewer" refers to the `sc.canvas.CanvasViewer` class or an instance of the class.

Preventing unwanted interaction between the Canvas Viewport Controls is unfortunately not as straightforward as simply disabling all other controls

while one is in use. For example, controls like Pan Zoom Control must be deactivated to allow a user to click and drag to draw a region of interest on the canvas, while the Zoom Slider Control must still remain active and visible. Because of this complexity, the logic of enabling and disabling these controls is offloaded into the Canvas Toolbar class (`sc.canvas.CanvasToolbar`). The Canvas Toolbar class is also responsible for rendering the graphical toolbar element shown in Figure 5, which contains buttons for selecting and drawing regions of interest, paging through the folia in a manuscript, showing and hiding regions of interest, and selecting different image options for viewing a canvas.

Figure 5: The Canvas Toolbar. On the left are three buttons for navigating between canvases in a manuscript. In the middle are buttons for switching between pan/zoom functionality (the pointing hand button), and various shape drawing tools. On the right are buttons for showing and hiding regions of interest, transcriptions, and alternative image representations of the canvas.

The Canvas Viewer's marquee in the right hand corner (shown in Figure 6) acts as a simple yet powerful tool for users working with very large and detailed documents. Much like the marquee seen on online mapping services, it shows a wider view of the entire document, while indicating the section visible in the main Canvas Viewport as a box within the marquee. The user can click within the marquee to move the main Canvas Viewport to that section, or drag the box indicating the bounds of the main Canvas Viewport to the region they wish to see. The bounding box representation updates live as the portion of the canvas

visible in the Canvas Viewport changes. This prevents users from losing the context of detailed views of the canvases they study, and beta testing users have expressed that they find it to be a useful tool when working on large canvases.[20]

Figure 6: The Marquee Viewport in the bottom right corner indicates via a blue box that the Canvas Viewer has been zoomed in to a portion of the top right segment of the canvas (specifically, the portion covered by the box).

This marquee functionality is implemented by having the Canvas Viewer maintain two Canvas Viewport objects. The first, referred to in the codebase as the main viewport, is the large Canvas Viewport on which the user pans, zooms, and works with regions of interest. The second, referred to as the marquee viewport, always maintains the same view of the canvas, but the Canvas Viewer

[20] Dot Porter and Marie Turner (University of Pennsylvania), Private Conversation

draws a bounding box matching the main Canvas Viewport's field of view within it. By subscribing to the viewport bounds changed events of the main Canvas Viewport, the Canvas Viewer is able to maintain the position of the bounding box in the marquee Canvas Viewport. By also subscribing to click and drag events on the marquee Canvas Viewport, the Canvas Viewer is able to update the main Canvas Viewport's bounds using the class's public API. This sort of extensive code reuse is just one example of why the numerous layers of abstraction used to build the Canvas Viewer interface the user sees are justified as a software architecture choice.

The Text Editor

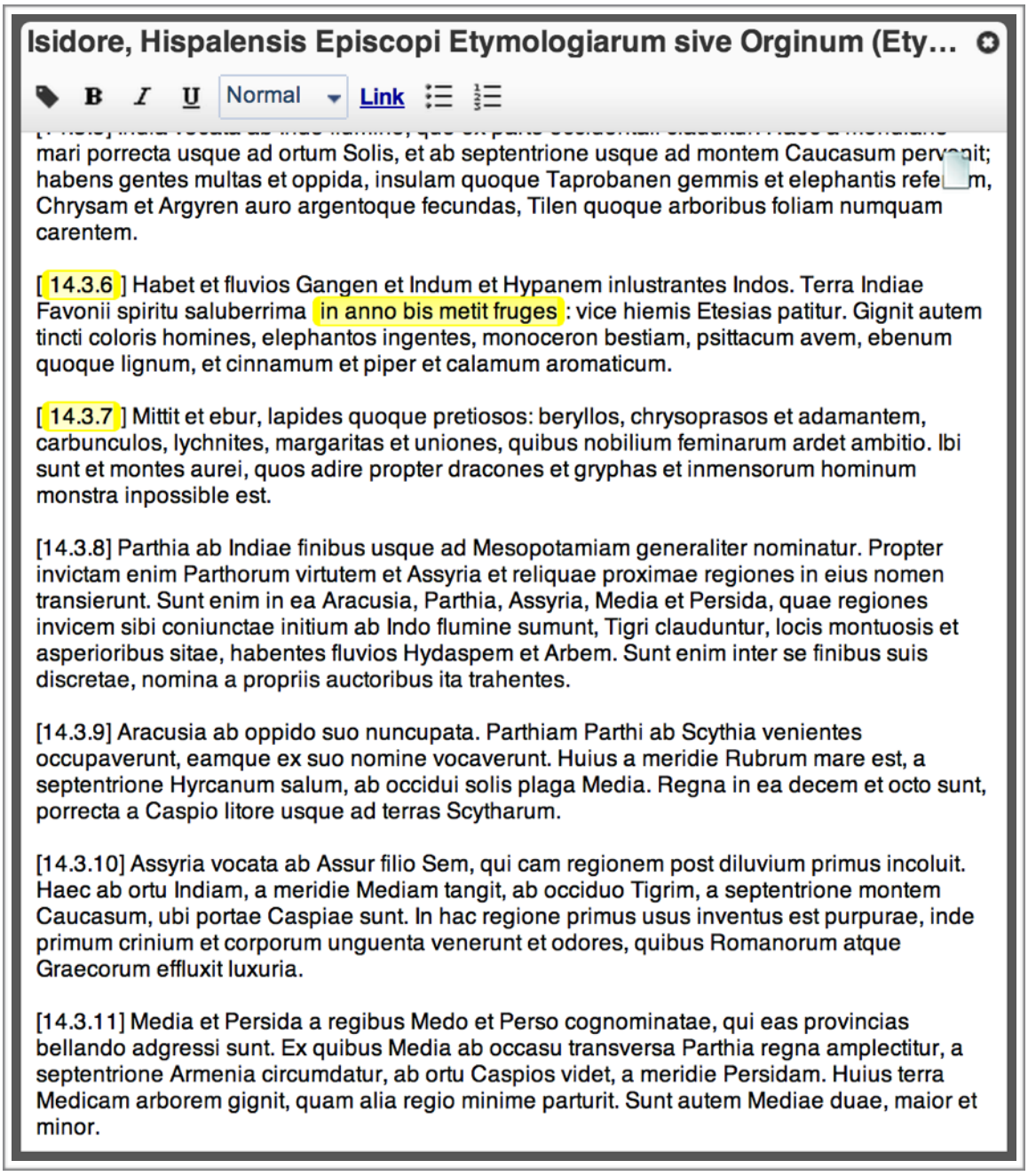


Figure 7: The DM Text Editor, showing a transcription of Isidore. Formatting tools are available from the toolbar shown at the top, and dynamic regions of interest within the text are shown as yellow highlighted sections.

DM requires a text editor for two purposes. First, in order for users to generate meaningful annotations with semantic content, users must be able to type in simple textual descriptions of the resources they wish to annotate. Secondly, users may also want to work with transcribed or translated versions of the resources which they study, resources which naturally exist as digital text documents. The DM user interface uses one text editor to serve both of these purposes, allowing users to generate anything from simple plain text to HTML formatted documents.

Very early in the development of DM, the development team chose to use the rich text editor in Google's Closure Library as the foundation for the DM Text Editor on the basis of its robust plugin architecture. While browsers have provided rich text editing features with varying degrees of support for several years, there are inconsistencies in the implementations that make the use of a Javascript library like Closure very desirable. At the time, the Google Closure Library text editor was in use in Gmail and various other Google services,[21] making it a clear leader as an open source rich text editor for web browsers. It's use of the HTML DOM for the rendering and editing of text content also makes it trivial to export formatted text documents as standard HTML data, and it allows the DM codebase to attach event listeners to elements within the text to allow for rich interaction with the document.

---

[21] "Introducing the Closure Library Editor" *Closure Tools Blog*.

One of my first tasks on the DM Project was to explore a way of augmenting the Closure Library text editor to allow users to create interactive highlights within the text documents, analogous to the freeform selection of regions of interest on a canvas. The Closure Library editor already provides basic highlighting functionality that simulates the physical act of highlighting text in a book using specific colors, but DM requires highlights to be interactive elements which can respond to mouseover and click events for the purposes of annotation. On my recommendation, the development team chose to implement region of interest selection, often colloquially called highlights in the development process, using native HTML span elements.

All regions of interest selected by the user are given a standardized CSS class name,[22] which styles the region with a yellow highlight background and start and end delimiters. The uniform resource identifier (URI) given to each individual highlight is stored as a property of the element, easily accessible from Javascript code when responding to user interactions with the elements on screen. The use of an inline span tag also simplifies the logic of tracking the position of highlights in the document, as the Closure Library editor automatically keeps those tags in place as the user edits other sections of the document.

---

[22] The CSS class name used is `atb-editor-textannotation`

However, the use of HTML span tags to delineate selections does have the distinct disadvantage of not allowing overlapping selections of regions of interest in the text, as overlapping tags are not valid in the HTML5 standard.[23] For example, the simultaneous existence of a selection of the first two thirds of a sentence and the last two thirds of a sentence would not be supported, as the start and end points of those selections overlap. While a workaround for this issue is theoretically feasible, the current implementation of DM does not yet support these overlapping selections. In order for the interface to support these overlapping selections, a more complicated representation of the highlights within the HTML markup of the text document would be required, and the user interface for selection those highlights would become notably more complicated. Because of this, the DM development team has not yet tackled this issue.

To the user, the functionality of DM's Text Editor should appear quite familiar. It does not have a concept of pagination like frequently used word processing tools (as pagination is a physical construct unnecessary in an online environment), but it otherwise supports many of the formatting features like bolding and bulleted lists which users expect from WYSIWG[24] editors. It also crucially provides hyperlinking functionality, allowing users to reference any sort of resource available on the web, even if it is not in a format that DM can directly

---

[23] *HTML: The Markup Language* §4.3.1 "Misnested tags"

[24] Common abbreviation for What You See Is What You Get

open. Users can define regions of interest within the text editor simply by highlighting a section of text using either the mouse or the shift and arrow keys (as in any other text editing software), and clicking the highlight button in the Text Editor's toolbar (shown on the far left in Figure 8). These formatting features are accessible to the user regardless of whether they are editing a simple textual comment annotation on another resource, or transcribing an entire manuscript.

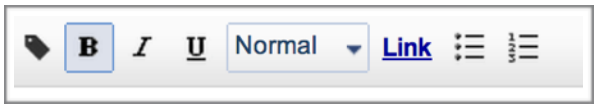


Figure 8: The DM Text Editor toolbar, with the highlight button at the left, and standard formatting tools to the right.

The Text Editor is implemented using two classes in DM's codebase. The Text Editor class (`atb.viewer.TextEditor`) owns an instance of the Closure Library's text editor class and the toolbar which the user can interact with for formatting. This class is responsible for interpreting the highlights the user creates within the text document and automatically saving them (and the contents of the document) to the front end data storage system for synchronization with the server. The development team chose to implement the highlighting functionality as a plugin to the Closure Library text editor (`atb.viewer.TextEditorAnnotate`), which adds the necessary span tags to the user's selections to create the highlights the user sees. Thanks to the full-featured and robust nature of the Closure Library text editor, the majority of the responsibility for the core formatting features of the editor are implemented by simply using the provided plugins from the Closure Library.

However, known issues still remain with the current implementation of the Text Editor. Most pressing among these issues, a user could copy a portion of a text document containing a highlight into another text document, meaning that two instances of the same highlight (with the same “unique” identifier) could exist within a user’s project, or even within the same text document. Ideally, highlights would be automatically removed from any content pasted into the Text Editor; however, it is not possible to intercept the content to be pasted within the browser using Javascript.[25] This means that some sort of comparison system would have to be set up, somehow identifying which was the original content and which is the newly pasted content. Due to limited development resources, the DM development team has been unable to successfully address this issue. Other minor issues with the formatting of content pasted from word processors like Microsoft Word remain, and have been treated as lower priorities.

The Viewer Grid

The Canvas Viewer and Text Editor act as the core viewing and editing tools of the DM interface, but for any sort of comparative or annotative work to be done, the user must be able to work with multiple instances of these tools at once. In a traditional desktop computer application, this problem has been solved for decades by window management systems which allow users to

---

[25] “Element.onpaste.” *Web API Interfaces*. Mozilla Developer Network.

arrange many tasks on the same screen. Clearly, DM's interface also requires a somewhat similar system for allowing multiple tasks, in this case Canvas Viewers, Text Editors, and any other tools developed in the future, to be open on the same screen. However, within the context of a web application, this problem is not so easily solved.

While in a traditional desktop computer application, a single parent process not visible to the user can manage multiple visible windows on the user's screen, allowing the user to open and close windows without losing any of the application's data, web applications do not lend themselves to this system. In order for a Javascript application to run in a web browser, at least one window must own the Javascript application. If that owner window is closed, then any child windows opened by this process become orphaned, and data can no longer be shared between them without the help of a remote web service. Further complicating the issue is the difficulty of keeping the DOMs of the windows separate in the Javascript code in order to prevent buggy behavior in a feature of web browsers rarely leveraged by web application developers.[26] For these reasons, the problem could not be easily solved by leveraging the window management system provided by the operating system on which the user's web browser runs.

---

[26] The `document` and `window` objects for each window would have to be very carefully referenced in Javascript code, because HTML elements created using the `document.createElement` method of one window behave unpredictably when rendered into the DOM of another window. I encountered this issue during the development process, but I have been unable to find well documented reports of similar issues, likely due to the rarity of such implementations.

Initially, the DM development team approached this problem with a simple “two up” view, in which either a Text Editor or Canvas Viewer could be open in designated spaces on either the left or right side of a single browser window (shown in Figure 9). This provided for basic comparison functionality by allowing the user to open two canvases side by side, and it enabled annotation functionality by allowing the user to have a text document or canvas open on one side with an annotative text document on the other side. While this worked for simple use cases, as users worked with more connected documents, it quickly became unclear which side of the browser window new resources should be opened on. And of course, no more than two resources could be open at once. This led to convoluted systems of backward and forward history buttons for the spaces on either side of the window (shown in the top right of Figure 9), and even an attempt at supporting popup windows to allow additional resources to be opened using real browser windows (which failed due to the technical limitations described earlier).

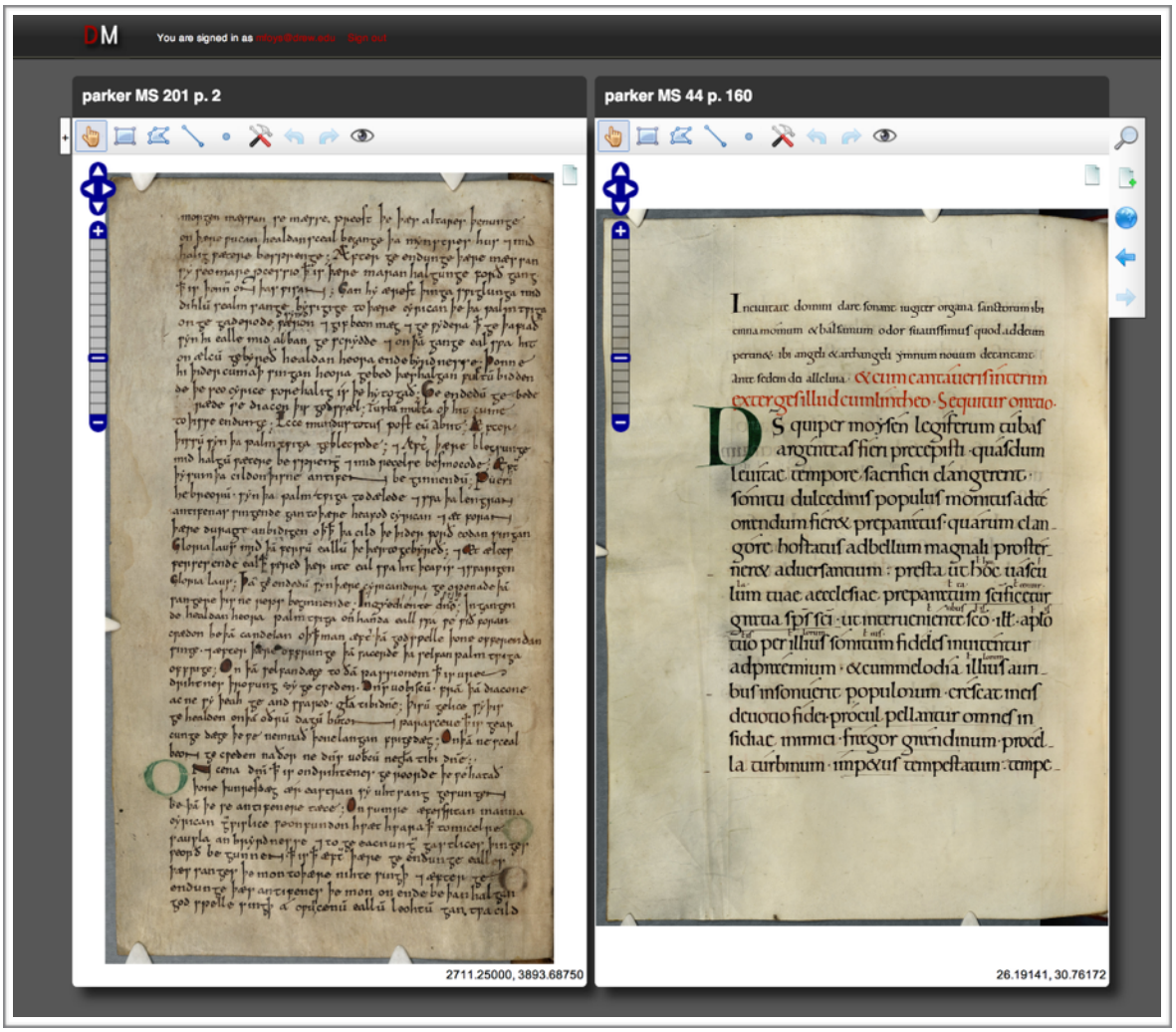


Figure 9: The "two up" interface of an earlier version of DM. The backward and forward buttons can be seen in the top right section.

Rather than making a comically awkward attempt at simulating a full featured window management system within one web browser window, the DM development team took the simpler approach modeling a variable size scrolling grid system in which those tools, referred to as Viewer objects (`atb.viewer.Viewer`) in the DM Codebase, could be laid out. This grid can be

of varying dimensions, ranging from a 1×1 view, in which every Viewer takes up the entire browser window, to an M×N grid, in which M viewers are evenly spaced out horizontally on the screen and N viewers are displayed vertically. By allowing users to resize this grid and rearrange Viewers, a simple Viewer management system which performs the vital functions of a user's desktop can be achieved.

The core module of the Viewer Grid model is the Viewer Container class (`atb.viewer.ViewerContainer`), which defines an abstract resizable rectangle to be rendered as an element in the Viewer Grid. This class implements the basic layout of an optionally editable title section at the top of the rectangle, a close button in the top right corner to remove the Viewer Container instance from the Viewer Grid, and a resizable space in which to render an instance of the Viewer class such as a Canvas Viewer or a Text Editor. It also handles resizing of itself and the optional Viewer instance it owns, a necessary feature for Viewer subclasses like the Canvas Viewer which must explicitly re-render their layout when resized rather than relying on the browser to dynamically reflow their layout.

These Viewer Containers can own instances of the Viewer class, which centralizes some of the common display logic of the Canvas Viewer and Text Editor. This class handles the display of graphical menus which display when users hover their mouses over resources (which will be explained in further detail

in the Linking section of this chapter), as well as displaying loading indicators and other common functionality shared by the Canvas Viewer and Text Editor. The Viewer class can also be subclassed to create other types of resource viewers, such as the simple audio player included in DM's codebase for demo purposes.

The layout of these Viewer Container instances and the Viewers they own is managed by the Viewer Grid class (`atb.viewer.ViewerGrid`), of which there is only one instance in DM's environment.[27] This class uses the Twitter Bootstrap Library's grid system to layout the Viewer Containers in an evenly spaced grid which resizes to fit the browser window. It provides public methods for resizing the grid into M×N configurations up to a limit of M=N=12,[28] which resize every Viewer Container instance it owns to re-render the layout, and subscribes to browser window resize events to maintain the layout of the grid even if the user resizes the browser window. It also supports adding Viewer Containers into arbitrary positions on the grid, enabling Viewers to open other Viewer instances immediately adjacent to themselves. At the time of writing, the Viewer Grid does not yet support rearrangement of the ordering of Viewer Containers; however, this feature is slated to be added into the final release of DM.

---

[27] The Viewer Grid class is not implemented so as to require that it be a singleton, but it is unclear of what use multiple viewer grids would be to a user.

[28] This limit is imposed by the use of Bootstrap classes for the column layout. Bootstrap splits the layout of a page into 12 equal parts.

The DM environment allows users to resize the layout of the grid by choosing from preset options in a drop down menu located in the top toolbar (shown in Figure 10). Users are allowed to pick from several layout options, including a “full screen” 1×1 layout, represented by boxes in the general design of the specified layout. Figure 2 shown at the beginning of this chapter shows the viewer grid in a 2×2 layout with Text Editors and Canvas Viewers within Viewer Containers. Although the Viewer Grid can support up to 12×12 layouts, the choices are limited to 3×3 (or 4×4 for larger monitors) since such dense layouts are impractical for users. When the user picks a different layout, the Viewer Containers within the grid remain in order by a left-to-right flow and resize to fit the new layout.

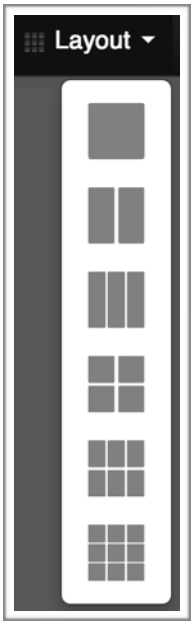


Figure 10: The Viewer Grid layout drop down menu. The top icon indicates showing one viewer at a time, while the second indicates showing two at a time, and so on.

Users of DM's latest release which includes the Viewer Grid have reported that it proves to be an effective tool for annotating and comparing multiple resources within the same scope of scholarly research.[29]

Linking

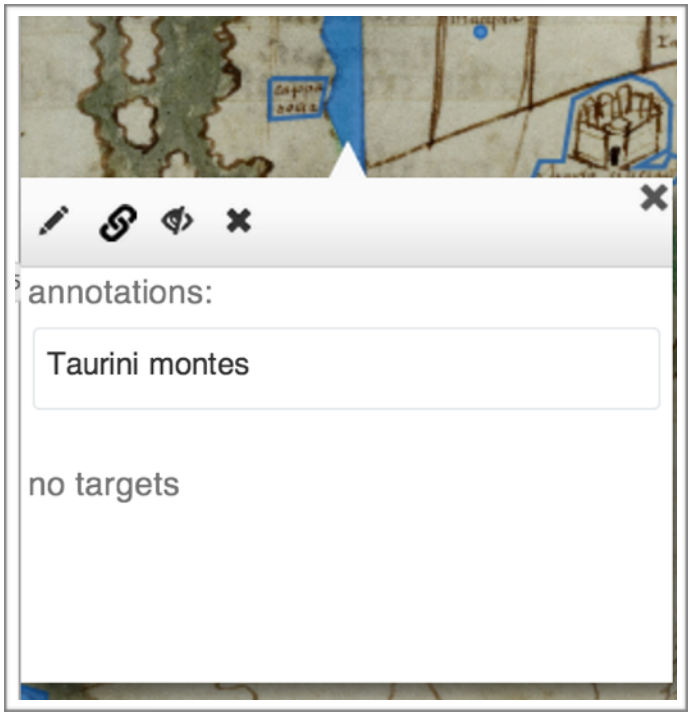


Figure 11: A Hover Menu is shown as a user hovers the mouse over a region of interest on a canvas. A "callout" triangle at the top points to the region of interest in question, and an "x" icon in the top right corner allows the user to hide the hover menu. A pencil icon triggers the text annotation functionality, and a link icon triggers the annotation linking functionality. The eye and "x" icons on the left are specific to canvas regions of interest, and allow the user to hide and delete the region respectively. The title of an annotative text is shown under "annotations:", and clicking on that title will open that text. The "no targets" text indicates that the region of interest itself does not annotate any other resources.

[29] Martin Foys, Catherine Crossley, Dot Porter, and Marie Turner, Private Conversations

Canvas Viewers and Text Editors allow users to view and edit most of the materials they study in an online setting, and they also allow users to select regions of interest within those resources. The Viewer Grid allows users to have multiple resources open at once for the purposes of comparative study. But in order to allow users to create annotations connecting those resources and regions of interest together, a systemwide annotation system is necessary.

In the base Viewer class (which the Canvas Viewer and Text Editor inherit from), a “hover menu” system is provided to allow a menu to pop up as the user hovers the mouse over resources in the DM interface. Each viewer is allowed to define custom buttons which appear in this menu, such as visibility toggling buttons in the Canvas Viewer, but they all share a button for creating a new text annotation and a new annotation link. The Viewer class also populates the hover menu with a list of the annotations which have the resource being hovered as a body or a target[30]. Clicking on any of these resources opens them in a spot in the Viewer Grid adjacent to that Viewer, and clicking on the remove button which appears when the user hovers over one of these resources in the list effectively deletes that particular annotation (without deleting the resource itself if there are other annotations which reference it). All linking in the DM environment is done through this hover menu user interface.

---

[30] This functionality is implemented in the `atb.ui.AnnoTitlesList` class, which uses several supporting classes to render representations of the resources linked by the annotations

The question of what elements on screen should trigger these hover menus does not have as clear of an answer as one might expect. For regions of interest like shapes drawn on a canvas or highlights in a text document, it is rather clear what element on the screen should be used to trigger these hover menus on a mouseover event. These resources have well defined boundaries visible on the screen to the user, making it intuitively clear that they should hover the mouse over that feature to show the hover menu. However, for resources like entire canvases and text documents, a metaphor is needed to show a hover menu which allows the user to interact with annotations on the entire resource. In the DM interface, these resources are represented by a document icon, which appears in the upper right hand corner of the viewer. Hovering over one of these document icons shows the same sort of annotation tools and linked resources using hover menus. This metaphor sometimes requires explicit explanation to the users, as it is less natural than hovering over small features on the screen, but it has proven to be an effective tool in practice.

While most of DM's current users use their desktops to interact with the DM workspace, attention has also been paid to the possibility of using DM on touchscreen devices like tablets and even smartphones. In this case, a user would have to tap on a resource to summon a hover menu. At the time of writing, little development effort has been put into ensuring that this works smoothly on touchscreen devices due to time constraints on the development

team. In the future, this will likely be addressed in conjunction with multitouch usability enhancements on tools like the Canvas Viewer and Text Editor.

The most common annotation feature utilized by users of DM is the creation of textual annotations, indicated by a pen and paper icon within the hover menus. This is effectively a shortcut for creating a new text document, and making it the body of an annotation which targets the resource the user originally hovered. This is particularly useful for one of the most common annotation tasks – identifying an item of interest, whether it be a canvas, a text document, or a portion of one of these resources, and writing a description or comments concerning that item. For example, a user might choose to create a text annotation on a region of interest they have just drawn on a canvas in order to describe that region. It also turns out to be the simplest type of annotation to implement from a software engineering standpoint.

Allowing arbitrary linking of all kinds of resources through annotations is a more difficult problem to solve. While it is clear what resource is intended to be the body of the annotation when a user clicks the chain link icon in a hover menu to create an annotation to another resource, it is more difficult to determine what resource should become the target of the annotation. I made the design decision to have the next resource the user clicks on become the target of the annotation, as this seems to be the most intuitive behavior. When the user first clicks the chain link icon in the hover menu, the color of the resource which

they selected as the body of the annotation changes to visually indicate that it will be the body of the annotation being created. A “Cancel link creation” button also appears at the bottom of the screen, should the user want to abort the creation of the annotation. Should the user not choose to cancel the creation of the annotation, the next resource they click on, whether it be the document icon of a canvas or text, or a highlight or region of interest on the canvas, the annotation will be created with that resource as the target. To visually indicate that the annotation has been created, the two newly linked resources flash, and a large chain link icon briefly appears on the center of the browser window. For a brief period of time, an “Undo link creation” button then appears in the same position as the “Cancel link creation” button, allowing the user to quickly correct a mistake.

While this interface for creating annotations linking arbitrary resources may sound straightforward when described from a user interface perspective (as any reasonably good user interface process should), implementing the software to enable this system is a more complex problem. In order for the software to recognize that the user has clicked on a second resource to be used as the target of an annotation, I implemented a global event dispatching system which notifies the piece of software creating the annotation that the user has clicked on a resource somewhere in the workspace. This event dispatcher, implemented

using the Closure Library's event utilities,[31] is owned by the Client App (`atb.ClientApp`), a utility singleton class to which virtually every module of the DM codebase maintains a reference.

This global event dispatcher serves as a centralized communication point for DM's Viewers. Viewers are responsible for notifying the event dispatcher anytime a user clicks on a representation of a resource, such as a highlight in a text document, a region of interest on a canvas, or a document icon. By subscribing to these resource click events, the utility methods for creating these annotations[32] can be notified of the resource on which a user clicks (and most importantly, its unique identifier). When such an annotation link is created, a link creation event is also fired on the event dispatcher, allowing Viewers to implement the visual flashing behavior of the newly linked resources.

Project Management

Collections of resources relevant to a scholar's work are referred to as Projects in the DM environment. These projects aggregate all of the canvases which a user works with, along with any standalone (i.e., not solely for annotative purposes) text documents they have created, and they can be shared with other users of DM with varying levels of editing permissions. Users can also

---

[31] The ClientApp maintains an instance of the Closure Library's `goog.events.EventDispatcher` class, on which instances of `goog.events.Event` with custom event type strings are fired. These classes follow the HTML Event Dispatcher interface.

[32] These utility methods are members of the singleton Client App

download their data in a standardized RDF format should they ever want to migrate it to another system or simply maintain their own local backup of their projects.

Figure 12: The project button and drop down menu. The title, in this example "Virtual Mappa", is always shown by the button. Clicking the blue down chevron shows the drop down menu, which has a button to create a new project, and an arbitrary number of alternative projects which can be switched to by clicking (in this case, "Testing Project" is the only other project available).

Users can manage multiple independent projects in DM's interface. Switching between these projects is enabled by a project chooser button group in the toolbar at the top of the DM window (shown in Figure 12). Clicking on the down arrow shows a list of projects to choose from, along with an option to create a new project. The interface automatically returns to the last project that the user opened, and when a user switches to a different project, they are warned that all of their open resources will be closed and saved before switching. Should the user want to work with two projects at once, they must open another window and navigate to the DM Workspace, in which case they would not be able to make links between the two separate projects.

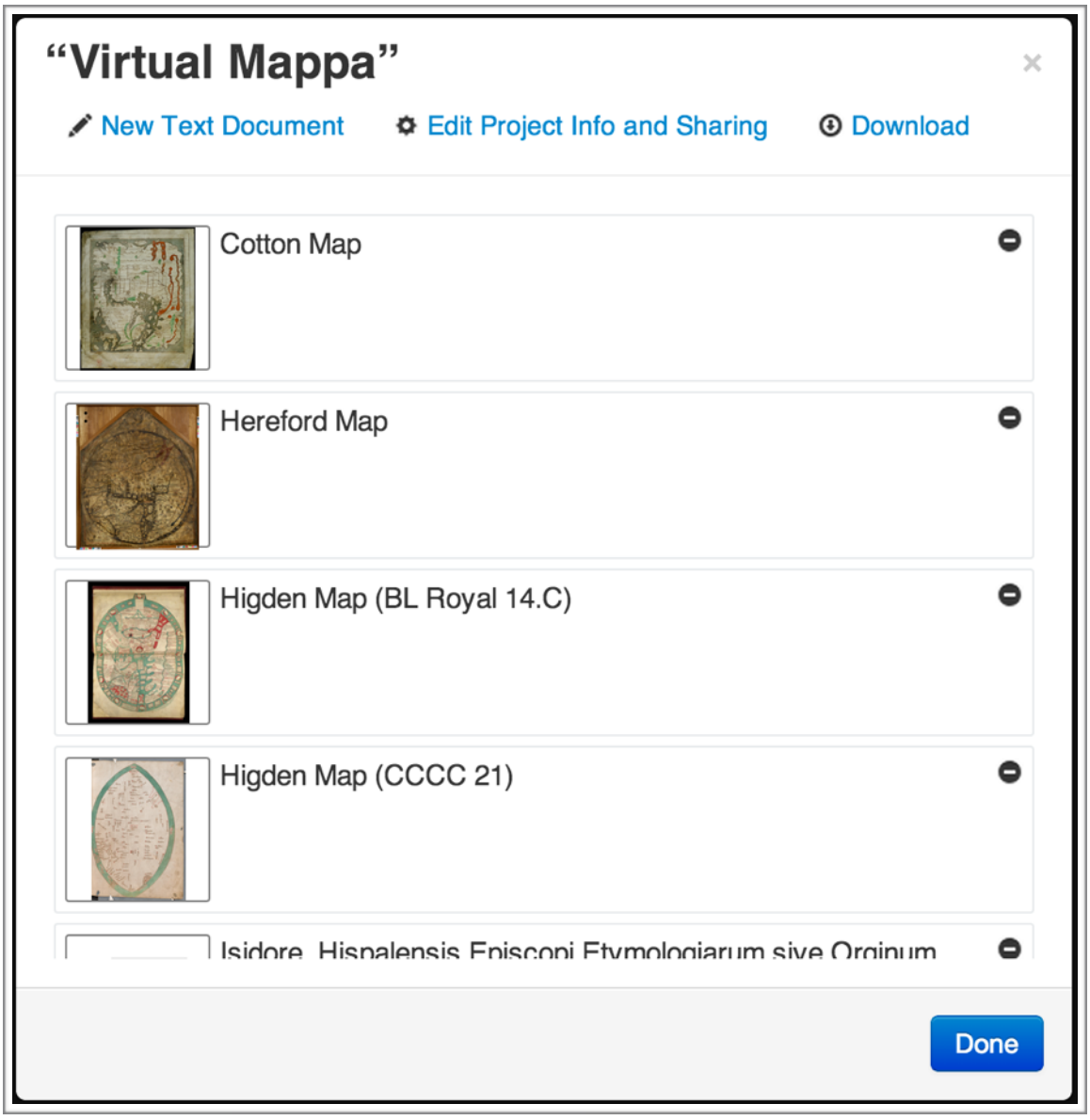


Figure 13: The Project Viewer showing the contents of the “Virtual Mappa” project. The showing of this tool was triggered by clicking on the project button shown in Figure 12. Top level project resources are shown with titles and thumbnail icons, while buttons for creating new texts, editing the project info and sharing settings, and downloading the project are shown directly below the title.

In order to view or edit the contents of a project, the user can click on the project button showing the title of the current project. This brings up a dialog with the title of the project as a header and the contents of the project in a (shown in Figure 13) scrolling view below. The user can click on any of the resources in the scrolling view to open them in the viewer grid, and they can click on the corresponding remove icon in the top right corner of each of the resources to remove that resource from the project. The resources shown as members of the project are currently sorted alphabetically, although the underlying code does support defined orderings for the resources. Functionality to manually reorder the resources within this project viewer is certainly feasible, but it has not been made a development priority.

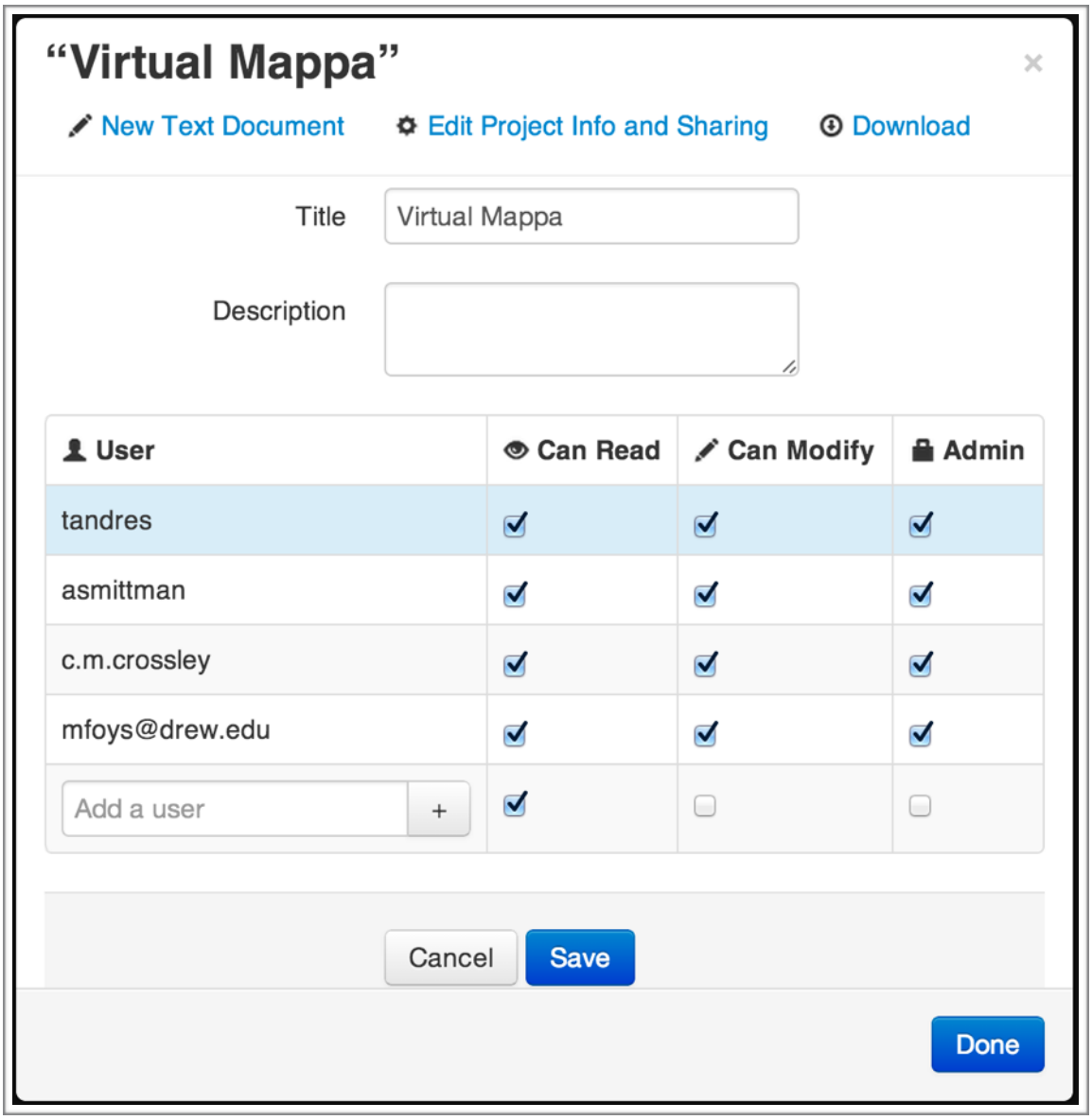


Figure 14: The Project Editing View. Shown by clicking on the "Edit Project Info and Sharing" button, this view allows the user to edit the title and description of the project, and to edit the sharing permissions. The current user is shown at the top and highlighted in blue, while other users are listed by their usernames below. An user with admin permissions may change other users' permissions by checking the associated permissions checkboxes, and can add another user by typing their username. Unchecking all permissions boxes removes a user from the project.

The project viewer also provides buttons for creating new text documents, editing the metadata and sharing permissions of the project, and

downloading the project's raw data. The "New Text Document" button allows the user to create a text document with the Text Editor which exists as a top level resource in the document. These top level text documents are listed within the project viewer, and thus do not need to be linked to any other resource in the project to be accessible. The use of this functionality is particularly appropriate for including transcriptions or translations of documents being studied within the project, as opposed to texts which only have the purpose of annotating another resource. Clicking on the "Edit Project Info and Sharing" button reveals an interface (shown in Figure 14) that allows the user to edit the title and description of the project, as well as manage the users allowed to view and contribute to the project. Administrators of the project can add users by their usernames and choose to give them read, modify, and administrative privileges by checking off the corresponding boxes in the interface. Finally, a "Download" button allows the user to download all of the data from their projects as a RDF files in the turtle format, should they ever want to export or manually back up the data they have created.

One feature conspicuously absent from the current project management interface is the ability to upload documents into the DM workspace. Text documents can rather easily be added to the workspace by simply copying and pasting their contents into a new text document in DM's text editor; however, it is not so simple for images of canvases. Building an image upload feature which allows users to create canvases from images stored on their computers is a high

development priority for the DM team; however, it is still in the process of being implemented at the time of writing.

Search

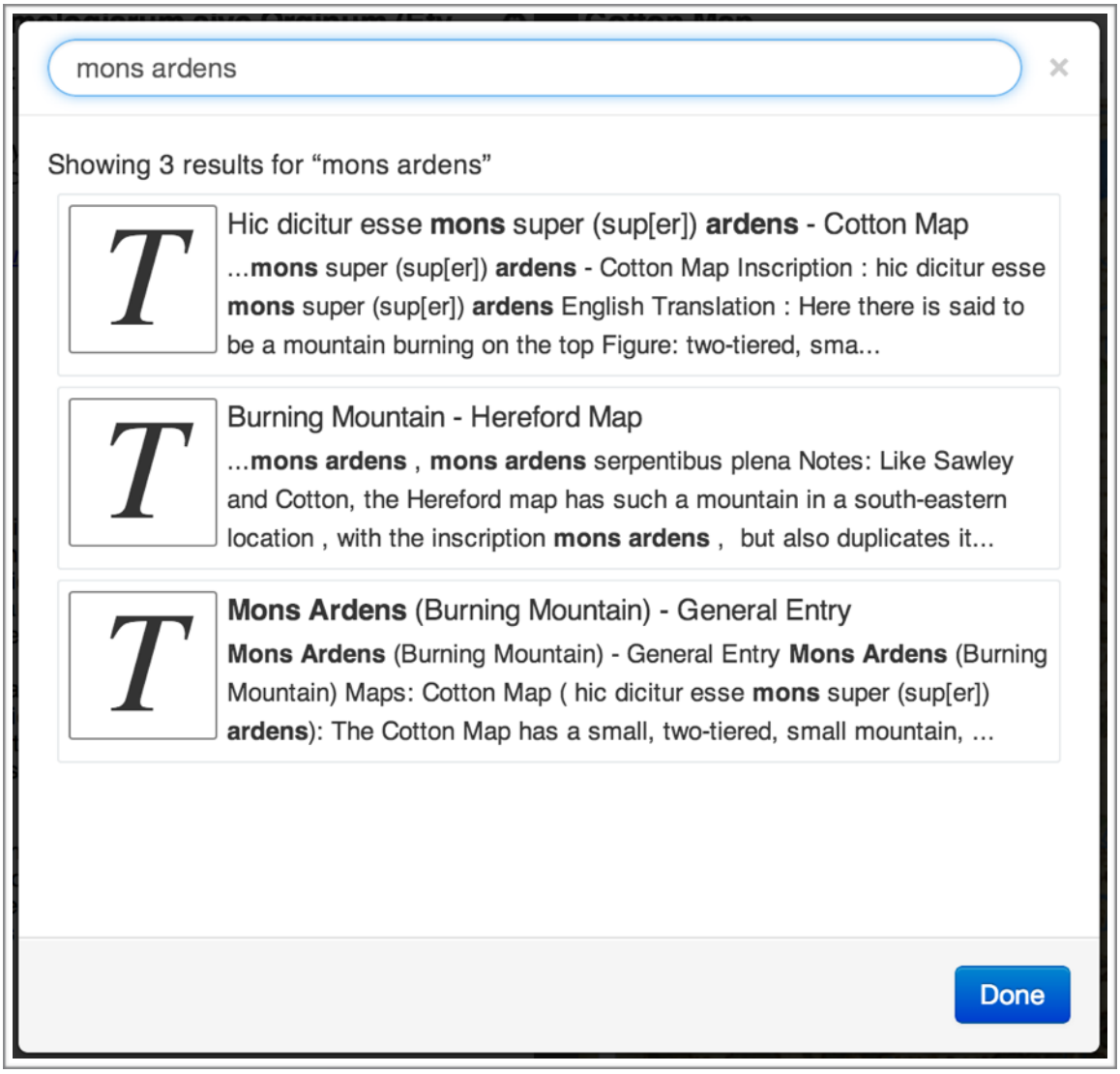


Figure 15: DM’s search engine user interface displays three text documents matching a search for the terms “mons ardens.”

DM's linking tools allow users to create a massive web of annotation data which can inherently be somewhat cumbersome to navigate. While a user may want to start exploring a resource by viewing all the annotations on a particular document and its regions of interest, they might also have a specific word or phrase in mind which they'd like to explore. This is where a search engine becomes extremely useful. I implemented the user interface for DM's search engine, which queries the back end web service's Apache Solr search engine server.

All of the text documents generated within DM are indexed by a search engine, usually but not necessarily hosted on the same server as DM's web service. As a user types in a search query, suggestions based on text documents found within the current project are presented to the user, updating in realtime as the user types. When a user executes a search query in DM's search interface, DM's web service returns data from the texts which the search engine finds matching the user's query within the currently open project. The interface then displays a list of these text documents, which the user can open by clicking. Once the user opens the text document, they can explore the annotations on that document. For example, if the user searched for "mons ardens", they might choose to open a text document which was written describing the "Mons Ardens" feature on a canvas. If the user opened that text document, they would be able to see using the linking interface that the

document targets a section of the Cotton Map canvas inscribed with “Mons Ardens”.

Because text documents are often used for transcribing features on a canvas, it would be ideal if the search interface would treat these cases specially, showing a thumbnail image of the feature being annotated and opening both the text and the feature on the canvas at the same time. Such a feature has been discussed by the DM development team, but due to time constraints, it has not yet been implemented at the time of writing.

The Data Model

One of the main objectives of the DM Project has been to allow users to share their work with others, but this sharing is not limited to users of DM. It is quite conceivable that other pieces of software will have overlapping functionality with DM or extend its functionality. Therefore, it is imperative that the data generated by users of the DM workspace be generated using a well defined data model in order to allow others to use that data, as well as to allow DM to import data from other systems. Using a standardized and widely adhered to data format ensures that the data users generate with DM will be portable – that is, other systems will be able to use it. Just like the standardization of HTML has allowed a menagerie of web browsers to be created for viewing HTML based web pages, the standardization of annotation data models should allow DM's data to be consumed and edited by many different pieces of interoperable software in the future.

Such standards are being actively developed by the digital humanities community, and two in particular are clearly suited for the DM Project's goals. The DM Project has adopted the Shared Canvas Data Model, a beta standard which builds upon the Open Annotation Data Model, a World Wide Web Consortium (W3C) draft standard. Both of these data models are built upon a single core data model – the Resource Description Framework.

RDF – Resource Description Framework

While crafting open data standards to describe the inherent properties of documents and scholarly annotations made on those documents, a portion of the digital humanities community has decided to adopt the aptly named Resource Description Framework (RDF). Developed for the purpose of encoding machine readable semantic metadata into web documents, RDF provides a simple yet powerful model for describing facts about the physical world and concepts in terms of easily consumable data.

The most basic unit of the RDF data universe is a Universal Resource Identifier (URI), a superset of the better known Universal Resource Locator (URL). A URI need not point to a location on the internet where the resource exists (although it often does), but it must be a string which uniquely identifies a resource.[33] For example, the person “Tim Berners-Lee" is described in RDF by the URI `http://www.w3.org/People/Berners-Lee`. The RDF specification also allows for literal values like strings and integers, but these are used to indicate properties of a resource like a name, not to uniquely identify it.[34] The example URI given for Berners-Lee provides a human reader with a good clue as to the person it is describing, but to a computer it is of course nothing more than a string of ones and zeros. So long as that string is unique, it can be used as an identifier for information about the resource.

---

[33] Berners-Lee et al., *RFC 3986 - Uniform Resource Identifier (URI)* §1.1, §1.1.3

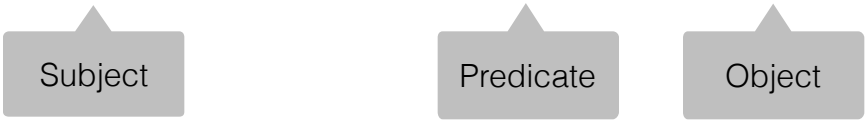
[34] Manola et al., *RDF Primer* §2.4

In order to describe the resource “Tim Berners-Lee” given by the example URI, we must define some sort of relationship for the data. Traditional relational database design might suggest creating a People relation implemented as a table in a database, with specific and finite attributes which a person such as Berners-Lee can have. We would then have to develop some sort of serialization to allow that data to be communicated between computers, and in an object-oriented programming language, a class to represent the data within every piece of software that reads and manipulates it. We would then have to do the same thing for every other kind of data which a person might be connected to. RDF provides a more flexible alternative – triples with subjects, predicates, and objects.

Suppose we’d like to tell the computer that Berners-Lee has the first name “Tim” and the last name “Berners-Lee”. First name and last name are clearly attributes that will be commonly used to describe people, and they have well defined and unique meanings, so the relationships “first name of” and “last name of” lend themselves to being identified by URIs. A standard ontology called Friend of a Friend (FOAF) has been established listing these and more relationships between people, referred to as agents. The URI for first name in the FOAF ontology is `http://xmlns.com/foaf/0.1/firstName`, which can be abbreviated using RDF prefix syntax as `foaf:firstName`, where “`foaf:`” is expanded to the full URI `http://xmlns.com/foaf/0.1/` by simple substitution. Likewise, last name is abbreviated as `foaf:lastName`. Using the terse RDF

triple language (Turtle), we could now describe the data as these two triples:

```
<http://www.w3.org/People/Berners-Lee> foaf:firstName “Tim” .
<http://www.w3.org/People/Berners-Lee> foaf:lastName  “Berners-Lee” .
```



or more succinctly in Turtle syntax:

```
<http://www.w3.org/People/Berners-Lee>
      foaf:firstName “Tim” ,
      foaf:lastName  “Berners-Lee” .
```

We can add some additional data to this example. For instance, Berners-Lee is a person, so we can say that he is of type foaf:Person using the standard RDF “type” expression abbreviated by either “`rdf:type`” or simply “a”. In Turtle format, this now gives us:

```
<http://www.w3.org/People/Berners-Lee>
      a foaf:Person ,
      foaf:firstName “Tim” ,
      foaf:lastName  “Berners-Lee” .
```

We could continue adding data about Berners-Lee by simply adding more triples about him, like an email address and official title (also defined in the FOAF ontology). We could also define our own ontology with a “creator of” relationship, represented by another URI we’ve chosen, and list among the many possible objects of that statement, the world wide web, the semantic web, and the W3C itself. Assuming that each of these creations were also

represented by URIs, we could also add arbitrary data about them, all without breaking any piece of software that simply wants to know Berners-Lee's first and last names.

Now suppose that we have a much larger dataset of people, and for some of them, we have triples which relate people together using the `foaf:knows` relationship. Using just this simple model of triples, we can ask the computer for information about particular people and about the relationships between those people. In fact, an entire querying language called SPARQL (analogous to SQL in the relational database world) has been developed to find and manipulate these relationships and data.

Based on this simple model of triples, we can create a complex graph of information, complete with embedded metadata like names and types. And because RDF is a data model rather than a data format, it can be conceptualized and manipulated in the same ways whether it exists in the memory of an application, in a serialized XML or Turtle file, or in the tables of a database. This powerful flexibility has made RDF a natural choice for encoding semantic data throughout the internet – in fact, the embedding of semantically meaningful RDF data throughout the web is one of the reasons that search engines can now return explicit answers to simple queries rather than just relevant pages.[35] As the reigning standard of the semantic web, RDF is well

---

[35] Herman et al., *RDFa 1.1 Primer* §1

suited for the needs of the digital humanities community seeking to publish its data on the web as well.

The Open Annotation Data Model

The highly extensible nature of the Resource Description Framework of modeling and storing data makes it extremely useful for storing networks of linked data, even if the data is composed of resources of various unrelated formats. This makes it an ideal system for storing user and machine generated annotations which link these resources together with semantically meaningful connections. The W3C community draft Open Annotation Data Model defines a core RDF resource called an Annotation, which links resources of any kind together by defining a target and body relationship, in which a body resource annotates a target resource. The power of this simple relationship lies in the fact that it is not limited to a specific set of RDF resources – any resources which can be described in RDF can be linked using this simple paradigm. In addition to this simple annotation relationship, the model also defines RDF resources called Specific Resources and Selectors, which form the foundation of region of interest selection in DM.

In the Open Annotation Model, all annotations are RDF resources with the type `oa:Annotation`. The body resource (or resources) of the annotation is defined by the `oa:hasBody` property, which semantically indicates that the body resource somehow describes what is being annotated. The target resource (or

resources) of the annotation is defined by the `oa:hasTarget` property, and semantically indicates what the body resource is describing. It is possible for an annotation to only have a target property and not a body property, in which case, the annotation is treated like a virtual bookmark pointing to a resource.[36] These annotations can also have an `oa:motivatedBy` property, with possible values like `oa:commenting`, `oa:questioning`, or `oa:replying` to indicate the purpose of the annotation,[37] and they can have metadata identifying their creators and time of creation associated with them. Essentially all connections between resources in the data generated by DM follow this `oa:Annotation` RDF resource specification.

Because scholars may want to annotate specific portions of the resources they study, the Open Annotation Data Model also specifies RDF resources of the type `oa:SpecificResource` and `oa:Selector`. Instances of the `oa:SpecificResource` class have two key properties: `oa:hasSource`, which identifies the resource in which the specific resource is located, and `oa:hasSelector`, which points to an `oa:Selector` which identifies the boundaries of the specific resource within the source document.[38]

The Open Annotation Data Model specifies several types of selectors for various types of resources, but DM uses only two: the `oa:SVGSelector` and the

---

[36] Sanderson et al., *Open Annotation Data Model* §2.1

[37] Sanderson et al., *Open Annotation Data Model* §2.3

[38] Sanderson et al., *Open Annotation Data Model* §3.1

`oa:TextQuoteSelector`. The SVGSelector identifies a specific resource on a canvas using an SVG shape as its boundary, stored as the string representation of the SVG element as a `cnt:chars` property of the selector.[39] The TextQuoteSelector identifies a segment of text within a text document by quoting it using the `oa:exact` property. It can also include the preceding and following text to help identify the precise position of the specific resource within its source.[40] The Open Annotation Data Model does specify an alternative `oa:TextPositionSelector`, which identifies the location of a specific resource in a text document relative to the total number of characters in the document;[41] however, DM does not use this option for two reasons. First, DM's Text Editor uses span tags within the content of its text documents to identify specific resources, meaning that one can easily query for the span tag with a given URI to find the position of the specific resource in the document (this is far easier than constantly maintaining position information in non-whitespace sensitive HTML). Second, the TextPositionSelector does not include the actual text content of the specific resource in its data, meaning that one would have to search through the entire source text document to simply find that content.

---

[39] Sanderson et al., *Open Annotation Data Model* §3.2.3.1

[40] Sanderson et al., *Open Annotation Data Model* §3.2.2.2

[41] Sanderson et al., *Open Annotation Data Model* §3.2.2.1

These core features of the Open Annotation Data Model serve as the core for DM's annotation data, and the foundation for the Shared Canvas Data Model.

The Shared Canvas Data Model

The Shared Canvas Data Model builds upon the Open Annotation Data Model to represent the physical resources that scholars study in the most flexible way possible. While it may be tempting for a software developer to simply think of the resources that scholars study purely as images, sometimes a single image cannot capture the full content of a given resource. There may be archival images of varying quality which have been used by different scholars to describe the same resource, and there may also be different content visible under different kinds of photography, such as infrared or ultraviolet photography. Because of these possibilities, scholars are often careful to call the images of the resources they study "surrogates" for the real resource.[42] In the case of visual resources, the Shared Canvas Data Model refers to the real resources as "canvases".

The Shared Canvas Data Model defines an RDF resource of type `sc:Canvas` to represent the abstract concept of the resource being studied. These canvases are simply defined by width and height properties (using the exif namespace), and metadata like a title and provenance information. The size

---

[42] Sanderson et al., "Shared Canvas: A Collaborative Model for Digital Facsimiles" §1

of the canvas is defined in pixels, but the size need not necessarily correspond to the resolution of any particular image taken of the physical resource; it is simply a reference point for defining the specific resource regions of interest on the canvas.[43] By separating the positioning of these regions of interest from the variety of images that could be used to provide a view of their source canvases, the Shared Canvas Data Model attempts to future-proof the data as different image representations of physical resources become available.

In order to define one or more image representations of a canvas, the Shared Canvas Data Model uses Open Annotation `oa:Annotation` resources linking canvases to images. In order to indicate that an image is a representation of a canvas, the RDF data must first indicate that the URL of the image has the type `dctypes:Image`, and should also include appropriate `exif:width` and `exif:height` properties. An annotation must then be created, with the image as its body, and the canvas as its target. In order to indicate to a client rendering the canvas that the image is intended to be rendered on the canvas, the Shared Canvas Data Model defines a value of the `oa:motivatedBy` property called `sc:painting`, which should be set on the annotation establishing the link between the image and canvas.[44] Because more than one such image annotation may exist in the RDF data, multiple image options can be

---

[43] Sanderson and Albritton, *Shared Canvas Data Model* §2

[44] Sanderson and Albritton, *Shared Canvas Data Model* §3

represented for a single canvas, giving the user the opportunity to choose which representation they would like to see when examining the canvas.

While sometimes scholars do study resources that exist as only a single page such as large maps, they often work with multi-page resources such as manuscripts. Because these manuscripts represent an ordered collection of canvases, the Shared Canvas Data Model expresses manuscripts as RDF resources with the types `ore:Aggregation` and `rdf:List`. Each manuscript declares that it contains a canvas by referencing it with the `ore:aggregates` property, and it defines a default ordering by using the standard RDF list pattern. The manuscript should also have metadata such as a title associated with it. Because it is not always clear when reassembling an ancient manuscript how the document was originally ordered, the Shared Canvas Data Model also allows alternative sequences of canvases in a manuscript to be defined. It also allows Ranges of the manuscript to be defined and labeled, allowing scholars to annotate sections of entire manuscripts along with sections of individual canvases.[45]

Text Documents

Storage of text documents in RDF data has not been as strictly defined by data models in the digital humanities community, largely because it is relatively straightforward. While adapting the DM system to communicate its

---

[45] Sanderson and Albritton, *Shared Canvas Data Model* §4.2

data entirely in RDF formats, I chose to use the `dctypes:Text` and `cnt:ContentAsText` RDF types to define the text resources created for use with DM's text editor. Although according to the Dublin Core Metadata Terms standard, the URI of the text resource can actually be a URL pointing to some sort of a text document[46] (an HTML document for example), for the sake of simplicity, I chose to use the `cnt:chars` property[47] to include the HTML content of text documents as a string literal within the RDF data. This keeps all of the data necessary to render a text document in one format, making the implementation of data synchronization much easier.

Regions of interest within text documents are described within RDF data and within the HTML of the text document using the Open Annotation Data Model. Regions of interest within text documents – the highlights which users see within the text editor – are represented using the `oa:SpecificResource` and `oa:TextQuoteSelector` resources described earlier. The Specific Resources and Selectors are declared in the RDF data as specified in the Open Annotation Data Model Section, and within the HTML of the text document, the RDFa standard for embedding RDF data within an HTML document is used. The HTML span elements surrounding the specific resources are given `about` attributes which indicate the URI of the selector, and they are also given `typeof`

---

[46] Dublin Core Metadata Usage Board, *DCMI Terms* "Text"

[47] Koch et al., *Representing Content in RDF* §3.3

attributes[48] which match the `oa:TextQuoteSelector` RDF type. By also setting the `property` attribute[49] of the HTML element to `oa:exact`, all of the data about the RDF selector resource, including the textual content of the highlight, is semantically declared within the HTML of the text document in a standards compliant manner.

Projects

In order for users to organize their work into logically separable collections, DM defines the concept of a project which contains canvases, manuscripts, texts, and annotations. I chose to implement a data model for DM's projects based on the `ore:Aggregation` specification, while making use of the Oxford University permission ontology for describing the levels of access users are granted to the project. Within this self defined data model, projects are defined to have both the types `dm:Project` and `foaf:Project`, which define that the resource is specifically a project created with DM but which also conforms to the more general RDF community idea of a project, and `ore:Aggregation`, which indicates that it aggregates resources into a distinct collection. Because projects also contain a large mass of data which may be specific to that project, they are also treated as named RDF graphs within this data model, meaning that the RDF data within the project will not unintentionally pollute other projects.

---

[48] Adida et al., *RDFa Core 1.1 - Second Edition §8.1.1.3*

[49] Adida et al., *RDFa Core 1.1 - Second Edition §8.3.1*

All of the “top level” resources which a user sees when they open the project viewing tool of the DM interface are required to be aggregated by the project, i.e., triples must exist stating that the project `ore:aggregates` that resource.[50] Because the `ore:aggregates` relationship is inherently unordered, projects may also be instances of `rdf:Lists` to define an order in which the resources should be displayed.[51] In order for a project to contain resources which are not seen in the “top level” view by the users, such as annotative texts, the RDF triples which describe these resources must be members of the project’s named RDF graph, whose identifier is derived from the URI of the project. This model allows users to have multiple projects which share references to unique resources, such as canvases from archival libraries, while still generating annotation data that can be separated between projects.

Project sharing is enabled through the use of the Oxford University RDF Permissions Ontology, which defines fine grained levels of access to a resource. DM uses only a subset of these permission levels: `perm:mayRead`, `perm:mayUpdate`, and `perm:mayAdminister`. The read permission level is self explanatory, and the update permission level allows a user to create and modify resources and annotations within the project. The administration level permission allows users to perform tasks like changing the title of a project or deleting it entirely. These permission levels are hierarchical, i.e., a user with

---

[50] Lagoze et al., *Open Archives Initiative Object Reuse and Exchange* §3.3

[51] Manola et al., *RDF Primer* §4.2

administrator privileges must also have update and read permissions, and a user with update permissions must also have read permissions. Following the Permissions Ontology specification, these RDF permission properties are defined upon RDF user resources, which have the type `foaf:Agent`. These permissions need not be declared specifically within the context of the project's named graph, as they are considered to be universally true.

Browser-side Data Management

As the DM team started the development of the latest iteration of the software, it made the use of real RDF data in every part of the software a high priority. Previous iterations of the software had used non-standard representations of the data in JSON and Javascript object formats, which were convenient for rapid prototyping, but lacked the connected graph structure innate to RDF data. This made it difficult to query for connections between resources within the browser, requiring constant queries to the back end web service over the network. The newest iteration of the DM software replaces this approach with a synchronized RDF data store accessible from the browser.

During the initial rewrite of DM's browser side software to use RDF data everywhere, very few good options for open source Javascript RDF toolkits existed. I initially tried to implement the software using RDFQuery, a tool mainly designed for gleaning embedded RDFa data from webpages, which seemed to be the default solution for working with RDF data in web browsers at the time. However, RDFQuery could not provide sufficient performance even when dealing with one manuscript at a time due to its apparent lack of an indexed data storage system. It became clear that DM would need its own custom RDF data storage solution, much of which I subsequently implemented.

The Quad Store and RDF Named Graphs

Although the core relationship of RDF is usually called a triple, these triples often exist within named graphs to separate data from different sources[52] (such as projects in the case of DM). The simplest way of storing this broader relationship of triples and named graphs is to treat triples as quads, composed of the subject, predicate, and object of the triple, along with the URI of its graph as a context. To allow for extremely fast data lookups, I implemented an indexed Quad Store class (`sc.data.QuadStore`) in DM's codebase.

In order to allow for extremely fast lookups, DM's Quad Store leverages hash maps to provide constant time lookups of quads matching a pattern. When a quad is inserted into the Quad Store, the Quad Store generates all possible combinations of the values of the quad and wildcard placeholders for those values as keys. References to the quad object are then stored in a single hash map for every one of these keys. This means that if the quad "`s, p, o, c`" is added to the Quad Store, a reference to that quad will be stored for keys like "`s, *, o, c`", "`s, p, *, c`", "`s, *, *, c`", and so on, where "`*`" indicates a wildcard. While this method does add a bit of overhead to the loading of quads into the store as the keys are constructed and hashed,[53] it allows clients of the Quad Store to execute queries using combinations of specific values and

---

[52] Klyne et al., *RDF 1.1 Concepts and Abstract Syntax* §1.6

[53] Hashing is implicitly performed by the browser's Javascript engine using standard Javascript object dictionaries. My testing has shown that implementing a hashing algorithm in Javascript slows down the key creation and lookup processes.

wildcard placeholders and to receive results nearly instantaneously via a constant time hash map lookup. This small tradeoff is very acceptable, as users may expect a slight delay when loading data over a network (which will immediately be followed by indexing and storing the data), but any lag in querying local data is justifiably seen as sluggish performance.

The Quad Store provides a public interface for querying for quads matching exact values and wildcard placeholders. The simplest of these methods, `query`, returns a set of all quads matching a query constructed with string values or null wildcards for subject, predicate, object, and context values. For example, this method could be used to query for all quads which have the predicate `rdf:type` and object `oa:Annotation`, effectively returning quads whose subjects are URIs of Annotation data. The Quad Store also provides convenience methods for just getting the subject, predicate, and so on values of quads matching a given query, as well as checking for the existence of at least one quad matching the query. It also provides methods for deleting quads matching a query. These public methods allow clients of the Quad Store to do all of the basic kinds of data storage, querying, and manipulation necessary for software like DM.

In some cases, it becomes necessary to run the same queries on multiple Quad Store instances. For example, when telling the web service which quads have been deleted since the last synchronization, it may be necessary to query

both the deleted data and current data in order to ascertain the type of resource being dealt with. In order to support this, I also implemented a Conjunctive Quad Store class (`sc.data.ConjunctiveQuadStore`). Instances of this class can accept multiple Quad Store instances against which it can query. It implements the same public querying method signatures as the Quad Store class by aggregating the query results from multiple quad stores and returning one combined result. It also implements the quad addition and deletion method signatures of the Quad Store class by applying those actions to all of the Quad Stores which it references.

In order to prevent clients of the Quad Store from always having to specify the context of quads in situations where the context is usually the same across multiple queries, I also implemented a Graph class (`sc.data.Graph`), which represents a named RDF Graph. Instances of the Graph class are instantiated with a single context, and they maintain a reference to an instance of the Quad Store (or a Conjunctive Quad Store) which is solely responsible for storing the data. This Graph class provides similar public querying, addition, and deletion methods to the the Quad Store class, which simply wrap calls to the underlying Quad Store instance with the context always bound to the queries. This design pattern was inspired by the Python RDFLib abstraction of RDF Graphs from RDF stores, which proved to be very effective in the development of the Python based DM web service. Because projects in DM are represented

with named RDF graphs, it becomes convenient in most circumstances to simply use an instance of a Graph rather than an instance of a Quad Store.

The Databroker

The Quad Store class provides the core service needed to store RDF data within the web browser, but in order to allow data to be synchronized with the back end web service, additional data must be maintained. A singleton Databroker class (`sc.data.Databroker`) is responsible for managing this data storage by facilitating the importing of new data obtained from the web, tracking the creation and deletion of RDF data in the DM workspace by the user, and triggering the synchronization of this data with the back end web service.

In order to synchronize the data stored in the user's browser with the data stored on the server, the Databroker must keep track not only of the current corpus of data, but also what data has been newly generated and what data has been deleted since the last synchronization with the server. To maintain this information, the Databroker actually maintains three Quad Stores – one main Quad Store which contains all of the current RDF data, and a new Quad Store and deleted Quad Store to keep track of freshly created and deleted data respectively. To keep these stores up to date, the Databroker must broker access to the creation and deletion of data, rather than simply providing blanket public access to the main Quad Store instance. To this end, it implements public

methods for adding and deleting data, and it initiates the parsing of data from sources like the back end web service and other web sources.

Because manipulating RDF data at the level of triples and quads can be somewhat tedious, I implemented a Resource class (`sc.data.Resource`) which allows its clients to query and manipulate data as properties of RDF resources rather than raw RDF triples. For example, rather than querying a Quad Store instance for all triples of the form "`<resource uri>, dc:title, *`", a Resource instance can be instantiated with the URI in question and a reference to an RDF named graph, and the client can call its `getOneProperty` method with `dc:title` as a parameter. This Resource class is then responsible for performing the underlying queries against the named graph and returning properly formatted values. It also provides methods for creating new resources, and adding properties to existing ones. These methods all call the appropriate methods of the Databroker instance so that new and deleted data remains properly synchronized. This system significantly simplifies the code needed to query and update resources in DM.

The browser side data store cannot be expected to have all of the data needed to render projects and resources already loaded and up to date; data will usually need to be loaded from the web. I implemented methods within the Databroker which allow for RDF data from any URL to be downloaded via AJAX and parsed into the Quad Store. In order to tie these in with the Resource class

framework, I also implemented a Deferred Resource class (`sc.data.DeferredResource`), which follows the jQuery Deferred Promise object pattern[54] to abstract away the process of requesting RDF data over the network for individual resources. When a client requests a Deferred Resource instance for a given RDF resource, the Databroker checks to see if an `ore:isDescribedBy` property exists for that resource giving a URL where more data about the resource can be obtained.[55] So long as such a property exists, the Databroker fetches that data and parses it into the main Quad Store, and the client which requested the Deferred Resource can add a callback handler function which will be executed once the data has been fully loaded. This allows a task like loading all of the data necessary to render a canvas to be performed as easily as by calling `.defer()` on a Resource instance, and providing code to be executed after the data is loaded to the `.done()` method of the Deferred Resource returned. This greatly simplifies the data loading logic used by tools like the Canvas Viewer and Text Editor.

The Databroker class allows parsers which implement its Parser interface (`sc.data.Parser`) to be registered for various RDF formats in order to interpret RDF data in the many formats in which it is available online. The Parser interface I designed defines an asynchronous `parse` method which accepts an RDF serialization as a string and calls a handler function with a list of parsed triple

---

[54] jQuery Software Documentation, http://api.jquery.com/category/deferred-object/

[55] Lagoze et al., *ORE Specification – Vocabulary* §2.2.4

objects. In order to allow the Databroker to parse XML, Turtle, and N3 serializations of RDF data (some of the most common formats), I adapted preexisting open source Javascript parsers to conform to the Parser interface I defined. The RDFQuery library is used to implement XML formatted parsing[56], and I slightly modified the open source parser code of the Node.js based N3.js library to work within a browser setting for Turtle and N3 parsing. Because the Parser interface I designed is asynchronous, I was able to advantage of HTML5 Web Worker technology to allow the Turtle and N3 parser to run in what is similar to a separate process, utilizing multiple processor cores when available for faster performance. Because the Databroker chooses a parser automatically based on the `Content-Type` header of the files it loads, this implementation allows clients of the Databroker to request RDF files in various formats without worrying about the details of how it must be parsed.

In order to get the RDF data generated by the user out of the Databroker's Quad Store instances and back to the server, it is also necessary to be able to serialize RDF data in various formats. I implemented a Serializer interface which allows RDF triples and quads to be asynchronously serialized into a string in a given RDF format. Like the Parser interface, the Serializer interface defines a set of formats which it can handle, allowing it to be registered to the Databroker which selects the appropriate Serializer for a requested

---

[56] The RDFQuery parsers and serializers have proven to be unreliable when dealing with escape characters, leading me to have the DM web service use Turtle or N3 formatted data whenever possible.

format. It also defines its serialization interface asynchronously to facilitate the possibility of Web Worker implementations. I implemented a Turtle format serializer which conforms to the W3C team submission for the Turtle language.[57] Although DM's back end web service currently always communicates through Turtle formatted data by default, I also adapted the existing RDFQuery XML serializer to conform to DM's Serializer interface. Between these two serializers and their matching parsers, DM can read and write RDF data in some of the most common formats used on the web. Thanks to the standardization of the Serializer and Parser interfaces, open source contributors can easily write plugins to extend this functionality should new formats become more common.

In order to implement synchronization with a web service in a way which was not strictly dependent on the implementation of the DM web service, i.e., in a way which allows the DM browser interface to be used with another web service if one was later developed, the DM development team designed a simple Sync Service interface which the Databroker uses to send information to the server. The Sync Service used by the Databroker is provided as a parameter to the Databroker's options at construction, and it has access to all of the Databroker's Quad Store instances. The Databroker then calls the Sync Service instance's `requestSync` method at regular intervals, at which point the Sync Service is responsible for checking for newly added or deleted data and sending those updates to its respective web service via AJAX. DM's default

---

[57] Beckett and Berners-Lee, *Turtle - Terse RDF Triple Language*.

implementation of the Sync Service works by grouping RDF resources by their resource type to determine which URL at the web service the data should be sent. The Sync Service is also responsible for updating the Databroker's new and deleted Quad Store instances after data synchronization has been confirmed. This system ensures that all of the data a user creates within their browser gets sent back to the server for storage and sharing.

Server-side Web Service

In order to permit the data which the user creates in the web interface of DM to be saved, the DM development team had to create a custom dynamic web service to store and serve up all of the necessary data. The DM development team chose very early on to implement this web service using the Django framework, a Python based dynamic web server framework used by some of the most visited sites on the web. The prebuilt modules of this framework combined with its vibrant plugin development community made it significantly easier to implement the user authentication, data storage, and even search functionality of DM. By using the RDFLib Python library, it was also possible to cleanly integrate the querying and storage of pure RDF data using the 4store RDF database.

One of the core functionalities of the DM web service is to serve up the HTML, CSS, and Javascript files needed to render the DM interface. Since the vast majority of these files do not need to be customized for each user, they can be served up by any static web server, such as Nginx or Apache. However, the login pages and the main workspace HTML page must be served dynamically with custom content for each user. After checking user authentication using Django's built in modules, these pages are served up using Django's template system. This functionality was once relegated to an entirely different Django project from the data storage and serving, having two separate Django server

processes running on two separate ports. However, because of the added difficulty of dealing with cross-domain requests, the DM development team chose to implement the latest version of DM as a single Django project, with the dynamic interface and data serving being handled by the same server.

The DM web service leverages both Django's object relational model (which facilitates data storage through a traditional relational database) and RDFLib's data store connectors to store user data. This allowed the development team to take a best of both worlds approach to data storage using relational database and RDF storage and querying mechanisms.

User login information and other metadata is stored using Django's built in model, allowing DM to leverage the built in authentication services Django provides.[58] To simplify queries for sharing permissions, these are also stored using Django's object relational model, and are verified anytime data is requested from the server. This data is translated into an RDF representation when the web interface requests any kind of data about users and the projects they can manage following the standards described in the Data Model section of this document.

Projects are stored as named RDF graphs using RDFLib's data store connectors. These graphs contain all of the data associated with a project, like annotations, canvases, text highlights, and regions of interest, with the

---

[58] "User Authentication in Django" https://docs.djangoproject.com/en/1.5/topics/auth/

exception of text documents (for reasons which will be explained subsequently). Most queries for data about specific resources from the web interface trigger queries against these project graphs, using simple queries similar to the Javascript methods I implemented for the browser side code as well as more complex and powerful SPARQL language queries.

In order to allow for faster load times for requests from the DM interface, some caching is implemented using additional named graphs which are maintained with every update sent to the web service. Due to slow loading times, the list of “top level” resources in each project are maintained in this way so that users do not have to wait for a slow query to execute before getting the most basic information about their projects. Instead of performing these expensive queries each time the data is requested, they are instead run on the much smaller corpuses of data sent by the web interface in update requests. Work is currently underway to implement this sort of caching system for canvases and texts, although the introduction of the enterprise grade 4store RDF database into the system has significantly decreased these problematic query execution times.[59]

Although the DM interface does not yet include an interface for reviewing the history of text documents, the DM web service does maintain such a history.

---

[59] In testing against heavily annotated canvases, load times were reduced from upwards of three minutes to approximately fifteen seconds simply by switching from RDFLib’s SQLAlchemy store (which builds on a relational database) to 4store.

I chose to implement text document storage using Django's object relational model using an append only pattern. This means that instead of overwriting the content of a text document with each update, a new version of that document is created with each update and marked as the currently valid text document. By maintaining timestamps and information on which user made the update, versioning history can be maintained.[60] Using the Django object relational model also has the added advantage of making text documents easier to integrate with the Django-Haystack project for the implementation of the search engine. Much like user and project permission data, text data is serialized on demand into RDF according to the data model specified earlier in this document. This makes the non-RDF implementation of text document storage invisible to clients of the DM web service.

All of these querying and update features of the DM web service are exposed through a RESTful API of semantically meaningful URLs.[61] For example, a HTTP GET request to `/projects/<project_uri>/canvases/<canvas uri>/specific_resources/<specific_resource_uri>/` will return data about the specified `oa:SpecificResource` RDF resource on the given canvas in the given project, including relevant annotation data. Likewise, a PUT or POST request to that URL would update that data, after verifying that the

---

[60] However, much tighter integration with the front end interface would be required to produce the kind of live shared editing environment seen in services like Google Docs – something that is far beyond the current resources of DM's development team.

[61] Designed with the principles of the W3C "Cool URIs for the Semantic Web" interest note (by Sauermann, and Cyganiak) in mind

authenticated user has modification privileges on that project. The web service even uses HTTP content negotiation to return data in requested RDF formats. However, because the HTTP specification is not conducive to deleting portions of resources, as the HTTP DELETE method is specified to delete the entire resource[62], I was forced to implement a workaround to the RESTful pattern. In order to delete specific triples about a given resource, `/remove_triples` is added to the URL for the resource, and the triples to be removed are sent using the PUT method to that URL. This adheres to the spirit of the REST protocol as closely as possible, and seemed to be a more feasible method of implementing RDF synchronization than communicating all updates and requests through SPARQL syntax update statements. By following the well known REST standard as closely as possible, the DM development team has attempted to make it as easy as possible for developers to write other applications which may query the web service.

One exception to the idealistic pure RDF implementation of HTTP communication with DM's web service is search results. Search queries are still performed using RESTful protocols, but data is returned using the JSON format. This is because search results are inherently ordered by their relevance ranking, and RDF lists are somewhat cumbersome. Furthermore, it makes little sense to store search results within the web interface's Quad Stores, as that data is usually used only once, and the web interface would need to know the URI of

---

[62] See Fielding et al., *HTTP/1.1*, §9.7. HTTP DELETE requests do not usually have bodies.

the search results RDF list in order to then query it from its main Quad Store. Even further complicating the issue, search results often return short segments of documents with highlighted search terms, meaning that a separate RDF representation of these snippets would need to be modeled and implemented. In light of these factors, I chose to return search results in a simple JSON list format, with the option of including an RDF serialization of all of the text documents returned as a string property within the JSON response. This greatly simplified the implementation of text searching within the DM interface.

These search responses are powered by the Django-Haystack API, a project which connects Django's object relational model to various open source search engine servers, such as Apache Solr (which is currently being used by the latest version of DM available to users). DM's web service is configured to notify the search engine server of each update to text documents using the Django-Haystack API. When a user runs a search query for a term, the DM web service uses the Django-Haystack API to run a search for that term while filtering the results to the given project and only currently valid text documents. In the future, the DM development team plans to add a feature which also returns the regions of interest which some text documents are used to annotate for display in the web interface.

Perhaps surprisingly, despite the central role images play in DM's main use cases, the DM web service takes very little responsibility for storing those

images. Much of the digital humanities community has already adopted the International Image Interoperability Framework (IIIF) standard for image serving, which defines an API for rescaling, rotating, and selecting regions of archival images. Furthermore, several standalone image servers including Loris and djatoka have already been implemented to support the IIIF API. DM supports images stored on any server so long as they are accessible by a URL, and nearly all of the data currently being used in DM's test cases references images stored on physically independent servers, including basic static file servers and IIIF compliant servers. The only case in which DM's web service takes responsibility for serving images is for the still in development user uploaded images feature, which currently relies only on a static file server serving files from an uploads directory. Theoretically, even this functionality could be offloaded at some point to a standalone image server.

While the data storage and serving functionality of DM's web service may be less glamorous than the interactivity of DM's web interface, it is absolutely critical to the success of the entire DM system. DM's web service serves as the foundation of data storage in the system, ensuring that a user's data is accessible to them through the internet on any computer in the world, and enabling them to collaborate on shared projects with other users.

Conclusion

As it stands at the time of writing, DM is a unique tool with the potential to greatly aid scholars in their day to day workflow. With its HTML5 interface, there is the potential for DM to be accessed from anywhere an internet connection is available, allowing scholars around the world to collaborate on shared projects and to share interactive representations of their work with the world. Over sixty scholars have used the DM software in its various iterations on projects including the study of manuscripts, maps, scrolls, tapestries, and even handheld photos of active archaeological dig sites in the field, all while providing feedback to help refine the software. The overall reaction from the community has been largely positive, with institutions like the University of Pennsylvania already working to set up their own instances of DM. While the software has not yet reached a mainstream usage stage, it has great potential to be used in conjunction with other tools as humanities scholarship migrates into the digital age.

That being said, DM is still a work in progress. As the development team prepares for the eventual release of the project as open source software, critical features such as the image upload mechanism described earlier must still be implemented. Other features, such as displaying markers from canvases with relevant text documents in the search feature, allowing drag-and-drop reordering of Viewers within the Viewer Grid, and allowing resources from DM to

be embedded into other webpages much like Youtube videos or Google Maps widgets have been suggested, but it is not clear how many of these will be feasible within the available development resources. Performance tuning to improve the sometimes complex queries performed by the web service, including caching strategies, is also necessary before the DM software is deployed for broader use. Perhaps most importantly, DM still needs to be tested with a heavier user load to better understand how the software performance scales under real world use cases.

As DM approaches an official open source release, the future of the software remains somewhat uncertain. Both members of the current development team (myself included) will have other obligations at the end of this academic year as we depart Drew University, and the departure of faculty from Drew University who have worked on the project means that it will be difficult to bring on and coordinate new developers to work on the project. At the time of writing, the University of Pennsylvania's Schoenberg Institute of Manuscript studies has expressed interest in hosting the project, and I have already assisted them in setting up their own preliminary instance of DM on their servers. However, no group has yet come forward willing to take on the further development of the software. While publishing the codebase on an active open source community like Github will make contributions from other developers possible, we simply have no idea how much interest there will be from the community in devoting development resources to DM.

That being said, I still personally believe that the DM Project has a bright future. There are many humanities scholars keen on modernizing their workflows with digital tools, eagerly awaiting a tool that can translate the sorts of commenting and tagging functionality that software like Google Docs and even Facebook have made ubiquitous on the web into something suitable for extensive academic work which includes non-textual resources. Recognizing that such tools have the potential to quickly become a staple of the scholarly workflow, Eric Poehler, a Classicist who has used DM in an excavation of Pompeii, has quipped about the buzzword "'Digital Humanities', or what in ten years we will likely call again 'Humanities'".[63] It is worth noting that this quote comes from a description of a course he is currently teaching at the University of Massachusetts Amherst called "Doing the Digital Humanities," in which the students are exploring resources using DM. I have been lucky enough to see the reactions of scholars during their first times using the DM software, and they have been overwhelmingly positive. While further software development may prove challenging, I hope that at the very least DM can serve scholars well for several years – certainly longer if development resources are devoted to it – and perhaps inspire the creation of even more functionality to better facilitate scholarly study.

---

[63] Poehler, *Honors 391*

Appendix

RDF Namespaces Used in this Document

Prefix	Namespace
cnt	http://www.w3.org/2011/content#
dc	http://purl.org/dc/elements/1.1/
dcterms	http://purl.org/dc/terms/
dctypes	http://purl.org/dc/dcmitype/
dm	http://dm.drew.edu/ns/
exif	http://www.w3.org/2003/12/exif/ns#
foaf	http://xmlns.com/foaf/0.1/
oa	http://www.w3.org/ns/oa#
ore	http://www.openarchives.org/ore/terms/
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
sc	http://www.shared-canvas.org/ns/

References

Adida, Ben, Mark Birbeck, Shane McCarron, and Ivan Herman, eds. *RDFa Core 1.1 - Second Edition*. W3C. 22 Aug. 2013. Web. <http://www.w3.org/TR/2013/REC-rdfa-core-20130822/>.

Andres, Timothy, Shannon Bradshaw, Adam Ducker, Lucy Moss, and John O'Meara. *DM*. Computer software. Web. <http://github.com/timandres/DM>.

Astle, Peter J., and Adrienne Muir. "Digitization and preservation in public libraries and archives." *Journal of Librarianship and Information Science* 34.2 (2002): 67-79.

Baranovskiy, Dimitry. *Raphael - Javascript Library*. Computer software. Web. <http://www.raphaeljs.com/>

Beckett, David, and Tim Berners-Lee. *Turtle - Terse RDF Triple Language*. W3C. Mar. 2011. Web. <http://www.w3.org/TeamSubmission/turtle/>.

Berners-Lee, Tim, R. Fielding, and L. Manister, eds. *RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax*. Jan. 2005. Web. <http://tools.ietf.org/html/rfc3986>.

Blackwell, Christopher. "Scholarship Outside the Codex: Citation-based Digital Workflows for Integrating Objects, Images and Text without Making a Mess." Schoenberg Symposium. University of Pennsylvania, Philadelphia. 23 Nov. 2013. Lecture.

Bootstrap. Computer Software. Twitter. Web. <https://www.getbootstrap.com/>.

“The Canvas Element.” *HTML Living Standard*. Web Hypertext Application Technology Working Group. Accessed 27 Mar. 2014. Web. <http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html>.

Closure Library. Computer software. Google. Web. <https://code.google.com/p/closure-library/>.

Django. Computer software. Django Software Foundation. Web. <http://www.djangoproject.com/>.

Dublin Core Metadata Usage Board. *DCMI Metadata Terms*. Dublin Core Metadata Initiative. 14 Jun. 2012. Web. <http://dublincore.org/documents/2012/06/14/dcmi-terms/>.

Dutton, Alexander. *Ontology for modeling permissions*. Oxford University. Web. <http://vocab.ox.ac.uk/perm/index.rdf>.

“Element.onpaste.” *Web API Interfaces*. Mozilla Developer Network, 28 Sep. 2013. Web. <https://developer.mozilla.org/en-US/docs/Web/API/Element.onpaste>.

FabricJS. Computer software. Web. <http://www.fabricjs.com>.

Fielding, Roy Thomas. “Representational State Transfer (REST)”. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000. Web. <http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm>.

Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. W3C. Sep. 2004. Web. <http://www.w3.org/Protocols/rfc2616/rfc2616.html>.

Foys, Martin, and Shannon Bradshaw. "Developing Digital Mappaemundi: An Agile Mode for Annotating Medieval Maps." *Digital Medievalist* (2011) Web. <http://www.digitalmedievalist.org/journal/7/foys/>.

Foys, Martin, and Shannon Bradshaw. *DM – Annotation to Dissemination*. Grant Proposal. National Endowment for the Humanities, 26 Jul. 2012. Web. <http://www.neh.gov/files/grants/drew_university_dm_from_annotation_to_dissemination_enabling_users_to_assemble_collections_of_images.pdf>.

Herman, Ivan, Ben Adida, Manu Sporny, and Mark Birbeck, eds. *RDFa 1.1 Primer - Second Edition*. W3C. 22 Aug. 2013. Web. <http://www.w3.org/TR/xhtml-rdfa-primer/>.

HTML: The Markup Language (an HTML language reference). W3C, 28 May 2013. Web. <http://www.w3.org/TR/html-markup/Overview.html>.

International Image Interoperability Framework. Web. <http://iiif.io>.

"Introducing the Closure Library Editor." Web log post. *Closure Tools Blog*. Google, 14 July 2010. Web. <http://closuretools.blogspot.com/2010/07/introducing-closure-library-editor.html>.

jQuery. Computer software. Web. <http://www.jquery.com>.

Klyne, Graham, Jeremy J. Carrol, and Brian McBride, eds. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. 10 Feb. 2004. Web. <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.

Koch, Johannes, Carlos A. Velasco, and Phillip Ackerman, eds. *Representing Content in RDF 1.0*. W3C. Web. <http://www.w3.org/TR/Content-in-RDF10/>.

Kropf, Evyn. "Will That Surrogate Do? Reflections on Digitally Mediated Collaborative Description for Islamic Manuscripts at the University of Michigan." Schoenberg Symposium. University of Pennsylvania, Philadelphia. 23 Nov. 2013. Lecture.

Kramer, Michael J. "What Does Digital Humanities Bring to the Table?" Web log post. *Issues in Digital History*. Northwestern University, 25 Sept. 2012. Web. 26 Mar. 2014. <http://www.michaeljkramer.net/issuesindigitalhistory/blog/?p=862>.

Lagoze, Carl, Herbert Van De Sompel, Pete Johnston, Michael Nelson, Robert Sanderson, and Simeon Warner, eds. *Open Archives Initiative Object Reuse and Exchange*. 17 Oct. 2008. Web. <http://www.openarchives.org/ore/1.0/rdfxml.html>.

Manola, Frank, Eric Miller, and Brian McBride, eds. *RDF Primer*. 10 Feb. 2004. Web. <http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>.

Mooney, Linne, Simon Horobin, and Estelle Stubbs. "About the Project." *Late Medieval English Scribes*. University of York, n.d. Web. <http://www.medievalscribes.com/index.php?page=about&nav=off>.

OpenLayers. Computer software. Web. <http://www.openlayers.org/>.

Parker on the Web. Stanford University. Computer Software. Web. <http://parkerweb.stanford.edu>

Poehler, Eric. *Honors 391 Spring 2014: Doing the Digital Humanities*. University of Massachusetts Amherst. Web. <http://honors391spr2014.wordpress.com>

RDFLib. Computer Software. Web. <https://github.com/RDFLib/rdflib>.

RDFQuery. Computer software. Web. <https://code.google.com/p/rdfquery/>.

Sanderson, Robert, and Benjamin Albritton, eds. *Shared Canvas Data Model*. 14 Feb. 2013. Web. <http://www.shared-canvas.org/datamodel/spec/>.

Sanderson, Robert, Benjamin Albritton, Rafael Schwemmer, and Herbert Van De Sompel. "Shared Canvas: A Collaborative Model for Digital Facsimiles." *International Journal on Digital Libraries* 13.1 (2012): 3-16. Web.

Sanderson, Robert, Paolo Ciccarese, and Herbert Van De Sompel, eds. *Open Annotation Data Model*. 8 Feb. 2013. Web. <http://www.openannotation.org/spec/core/>.

Sauermann, Leo, and Richard Cyganiak, eds. *Cool URIs for the Semantic Web*. W3C. 03 Dec. 2008. Web. <http://www.w3.org/TR/cooluris/>.

Scalable Vector Graphics (SVG) 1.1 (Second Edition). W3C. 16 Aug. 2011. Web. <http://www.w3.org/TR/SVG11/Overview.html>.

Shiel, Patrick, Malte Rehbein, and John Keating. "The Ghost in the Manuscript: Hyperspectral Text Recovery and Segmentation." *Codicology and Palaeography in the Digital Age*. Norderstedt: Instituts Für Dokumentologie Und Editorik, BoD, 2009. 159-174. Print.

Solr. Computer Software. The Apache Software Foundation. Web. <https://lucene.apache.org/solr/>.

“Unleash the Power of Hardware-Accelerated HTML5 Canvas.” Microsoft Developer Network. Accessed 27 Mar. 2014. Web. <http://msdn.microsoft.com/en-us/hh562071.aspx>.

Verborgh, Reuben. *N3.js*. Computer software. Web. <https://github.com/RubenVerborgh/N3.js>.

4Store. Computer software. Garlik. Web. <http://4store.org/>.