Spectator: An Open Source Document Viewer

Jean-Philippe Gauthier, Adam Roegiest
Kira Systems
Toronto, Canada
{jp.gauthier,adam.roegiest}@kirasystems.com

ABSTRACT

Many information retrieval tasks require viewing documents in some manner, whether this is to view information in context or to provide annotations for some downstream task (e.g., evaluation or system training). Building a high-quality document viewer often exceeds the resources of many researchers and so, in this paper, we describe the design and architecture of our new open-source document viewer, Spectator. In particular, we provide a look into the algorithmic details of how Spectator accomplishes tasks like mapping annotations back to the canonical document. Moreover, we provide a sampling of the use cases that we envision for Spectator, potential future additions depending on community need and support, and highlight situations where Spectator may not be a good fit. Furthermore, we provide a brief description of the sample application that we bundle with Spectator to demonstrate how one might use it within the context of a larger system.

CCS CONCEPTS

• **Human-centered computing → Interactive systems and tools**.

KEYWORDS

document viewer, open source, ocr, annotation

ACM Reference Format:
Jean-Philippe Gauthier, Adam Roegiest. 2020. Spectator: An Open Source Document Viewer. In *2020 Conference on Human Information Interaction and Retrieval (CHIIR '20), March 14–18, 2020, Vancouver, BC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3343413.3377986

1 INTRODUCTION

Suppose we had an annotation task that required a user (or users) to proceed through a document collection and mark up documents with respect to correctness, relevance, or other comments. Naively, one might be inclined to pay for an existing document viewer (e.g., a PDF viewer, Nvivo[1]) with requisite annotation functionality or create their own bespoke viewer (e.g., as a web app) or attempt to retrofit existing viewers. The former may require custom plugins for applications to harmonize annotation across different document formats (e.g., to collect and collate the annotations) [9, 15]. While the latter options may limit reproducibility due to the ever changing nature of how applications are rendered and created (e.g., new versions of a JavaScript engine may subtly change how users interact with the system due to changes in layout).

While there are many different document viewers available to the community today, several [2, 3] are not under active development, one [5] is desktop only which limits ease of deployment, two [1, 4] are primarily only for viewing and do not have good integration to select text in reference to the source, and the one [4] closest to our use case only works with PDFs. Accordingly, we believe that there still remains a need for a battle-tested document viewer that is capable of providing a high-quality experience for both developers and users in the community. We believe that Spectator, which is derived from our in-production document viewer, helps to fill that need in the community with a goal towards building an extensible behind the scenes framework to allow others to build on our work.

In this paper, we detail the design of our open source document viewer, Spectator[2] (Figure 1), which aims to address several of these issues. Our document viewer works with rendered images of documents (e.g., pages from a PDF, or rendered HTML images) allowing users to annotate sections of those documents and translates them back to identified words (identified through OCR). By placing this requirement of images rather than raw documents, we are able to use Spectator in a document format agnostic way, limited only by our ability to produce said images. In doing so, we have a canonical reference for what the document looked like at a fixed-point in time meaning that any subsequent examination refers to the same rendered document.

We then present a series of tasks for which we imagine someone might utilize Spectator. In particular, these tasks relate to generating training data for ML algorithms, test collection creation, and document annotation for learning and training. While we have designed Spectator to be meant for "real-world" use, we are also distributing it with the intent of facilitating consistent and replicable interactive IR user studies and experiments. We also hope that the community will help in building additional components or machinery around such a platform for the benefit of all.

Following with a brief overview of Spectator's limitation, we provide a brief overview of the demo application that allows users to annotate PDF documents and collects light behavioural metrics. We are distributing this application with Spectator as a means to show what is possible when one combines our document viewer with several other open source libraries. In doing so, we hope to kick start some of the interest in the framework so that others do not have to start from scratch and have existing structure to use as inspiration.

[1] www.qsrinternational.com/nvivo

[2] Available at https://github.com/kirasystems/spectator.

CHIIR '20, March 14–18, 2020, Vancouver, BC, Canada

ACM ISBN 978-1-4503-6892-6/20/03…$15.00
https://doi.org/10.1145/3343413.3377986

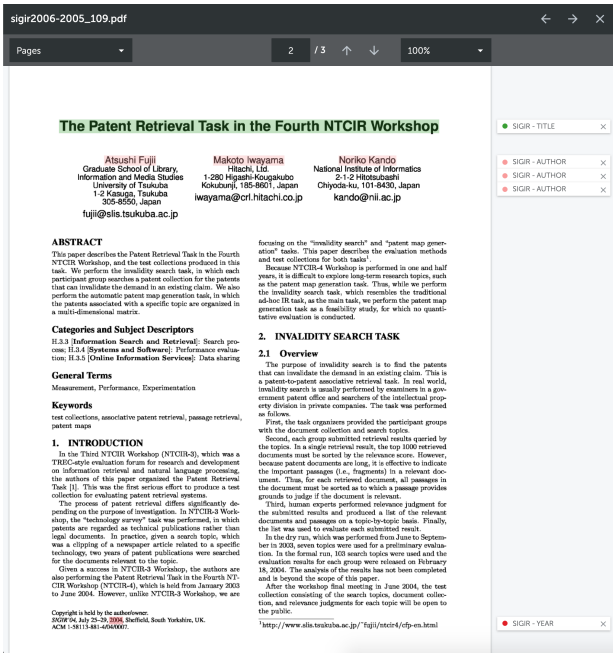

Figure 1: A screenshot of Spectator in the form of our demo application.

2 DESIGN

The major architectural choice made in our design viewer was to provide an image of the original source document to the user. Due to the nature of our task, it is necessary to run OCR over the original document and that is a lossy process. By providing a visible copy of the original document, we allow users to annotate and highlight text based upon what they *see* rather than what the OCR process (and any subsequent rendering) would dictate they should see. The one major pitfall to this approach is that we are required to pre-render to images all documents and then OCR those documents, even those digitally born documents, to ensure that we have the requisite image(s) and positional information. We note that while this approach mitigates some of the challenges associated with maintaining annotations on changing documents [9], it does create new efforts in that updated documents must be rendered and annotated. Figure 2 provides an example of how documents would be processed in a Spectator-based application. We note that is is possible to naively support various born-digital formats but this is not on our near term roadmap for Spectator.

To facilitate the annotation and "selection" of text, we require that the OCR process (or other post-processing) map the identified characters/runes to their (x, y) positions in the source document. We also require that this process returns an indexed sequence of characters. When highlighting, we begin recording the initial position of the cursor on the `mousedown` event and the final position of the cursor on the related `mouseup` event. Using these two positions, an R-tree [3] filled with the character positions and the indexed sequence of characters, it is possible to identify the starting and

[3] A specialized data structure for looking up spatially related data.

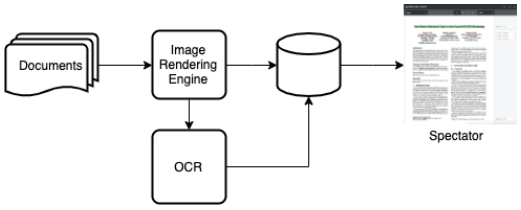

Figure 2: An example of the processing flow of documents for use with Spectator. Documents are first fed into an image rendering engine, the images are stored in a database, and then sent for OCR, the OCR'd text and additional layout information is stored in a database. The images and document data is then used by a Spectator application.

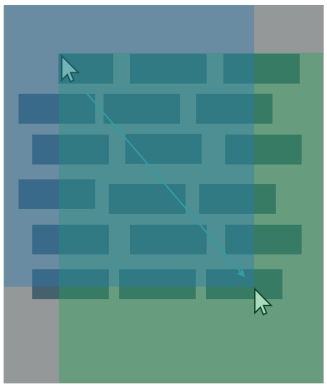

Figure 3: The selection process using an R-Tree

ending characters of the selection. This technique is illustrated in Figure 3. To identify the starting character, the R-tree will be used to find all characters intersecting the green area. Since the characters are indexed, we will take the one with the smallest index to be the starting character. The same technique is used to identify the ending character, but using the blue area and the highest index. It's also worth mentioning that users expect different selections depending on the direction and the sense of the direction dictating the choice of the starting and ending characters.

Annotations are defined by the starting and ending character positions in the text. They can then be stored in a database as separate entries, allowing the user to tag the text for later processing. We note that at no point during this process is the user actively engaging with textual elements and we merely "draw" a coloured box over the selected text. The actual identification of what textual elements were selected is determined behind the scenes but we note that to facilitate efficient operations, we may cache the page's data in the browser to facilitate the efficient implementation of specialized operations (e.g., copy and paste).

Annotations are drawn directly on top of the image of the page in a `SVG` element that matches its size. The `viewbox` property uses the dimensions of the source document. It allows us to use the source positions without having to transpose them. Given an annotation and the indexed sequence of characters, it is possible to compute the enclosing rectangles that would cover the different lines of text

overlapping the annotation. To speed up the computation of these enclosing rectangles, we require the indexed sequence of characters to also contain the line numbers. That way, it makes it trivial to compute the rectangles for the associated lines. Finally, there is the requirement to be able to navigate directly to the different annotations. In order to do that, we also require annotations to contain the (x, y) position of its first character. By converting the positions to percentages of the source document, it is possible to map it back to the document in the browser and navigate directly to the annotation.

We note that it may be tempting to "overlay" the OCR'd text on top of the image (in a transparent manner) to facilitate the selection process. While this does work, it requires potentially more burden on the user's browser to store the invisible text, position it properly and have the selection process work as expected. Moreover, it prevents us from allowing users to zoom in/out on the image as we cannot as easily modify the size of the overlaid text to align with the desired magnification. This particular limitation was a pain point in an earlier version of our document viewer when users wanted to get a closer look at the source text. However, using the positional method described above, mapping the location of highlights on the zoomed text to the original is a matter of relatively straightforward math that we depict at a high-level in Figure 4. We note that approaches that do not actually overlay annotations on the document can utilize more simple structures (e.g., XML), such as those by Baldwin [7] and Thomas and Brailsford [16].

Moreover, the overlaid approach requires annotating each character (or token) with a span to facilitate highlighting (i.e., we can modify a particular set of spans' style to reflect being highlighted). While simple and easy to implement, this can have drastic performance ramifications as each token that is enclosed in a span increases the size of the DOM and puts a greater burden on the user's browser to maintain this and efficiently perform modifications as needed.

2.1 Implementation Details

The entirety of Spectator is written in a combination of JavaScript, TypeScript[4], and React[5] to provide a modern implementation that interested developers will find easy to modify, debug, and understand. In doing so, we hope to make Spectator accessible to interested parties without requiring knowledge of "old" or "niche" web technologies that practitioners may have little knowledge about.

To aid in general user accessibility, our highlighting will alternate between one of 12 predefined colours. These colours were selected to be maximally distinct and allow easy disambiguation [11]. However, this selection process was not perfect and we are open to suggestion on selecting more universally acceptable colours or providing options to accommodate differing preferences and needs. To let users read the text behind the annotations, we used an opacity of 30% and the CSS property `mix-blend-mode` to `multiply`.

2.2 Potential Additions

One of the most obvious additions to Spectator is the ability to search within a document. This was an intentional decision to allow

[4]https://www.typescriptlang.org/
[5]https://reactjs.org/

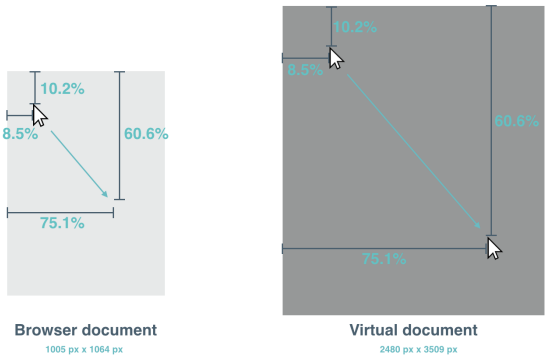

Figure 4: A demonstration of the math behind mapping zoomed document highlights to the original document.

developers and interested parties to use Spectator first and then have a cogent discussion on how best to incorporate search into Spectator. Search (and highlighting) is relatively straightforward to implement if we only care about boolean-based regular expression search as we just map matching text spans to the locations in the containing image. However, we did not want to be prescriptive and bake-in functionality that people may not desire and leave it for the community to help select an appropriate choice.

Our overriding goal to make Spectator was to make it as extensible as possible without sacrificing ease of use and performance. Accordingly, we have left several features out that may be of benefit to the community (e.g., user accounts, main navigation screens) as those are likely to be tailored to individual user needs (e.g., user studies may require different setups). Although, we do plan to release components that can be used to help design and facilitate the creation of such additional functionality.

Additionally, we have left much user behaviour tracking out of the initial release. This was intentionally done to avoid being prescriptive. Our demo application does highlight how one can easily add such tracking through the judicious reuse of components and modern front-end development practices. We are not opposed to baking some of this tracking into the existing components but did not want to be heavy handed in the initial release.

Finally, Spectator has had limited optimization for mobile/tablet use cases due to limited resourcing available to us. While there is some internal demand to better support these interaction styles, we have not had a chance to properly investigate them and design accessible solutions. Insofar as touch/stylus actions are translated by the browser to equivalent mouse events, Spectator will work as advertised. This means that the ability to annotate the documents may be limited as we have not found many touch-based browsers to emulate the `mousedown`/`mouseup` events in a way that facilitates this action. Accordingly, we hope to provide additional support for this in the future to make Spectator more generally accessible.

3 USE CASES

We envision our primary use case to be any situation where individuals want to browse and annotate original source documents (or a rendered variant thereof). By annotating on top of the source text,

users are able to have a consistent and controllable viewing experience. For example, if we were to have users annotate text on a web page, the typical approach might be to strip out all non-visible text (e.g., HTML, and Javascript) and render that in a human readable way. However, this loses much of the contextual information that may be gleaned from the "canonical" in-browser view. By rendering to a canonical format (e.g., using a headless browser and saving to PDF), we ensure a version that is fixed in time to a particular rendering service which also may help increase reproducible experiences.[6] Though several studies [10, 14] have used in-browser annotation extensions to facilitate same goal.

These annotations can then be used to facilitate training machine learning systems, evaluating Information Retrieval systems, or for providing feedback to other users (e.g., clarifications and corrections). The first is the primary use case for our internal use of Spectator (i.e., annotations are used to train the system to identify similar pieces of text). By referencing the original document, we are able to use positional features (e.g., "the token above this one," whether a token is centered on a page) to help train the classifier to recognize the desired text. Spectator can also be used to allow users to provide feedback on automatically generated passage-level annotations on a canonical document rather than a stripped version (i.e., one can replicate systems like HiCal [6] in our framework).

Another beneficial use to Spectator is being able to annotate source documents easily means that generating test collections for (sub)passage-level relevance (e.g., question answering, faceted retrieval) may be expedited and be able to generate a canonical reference back to the source rather than using character/word overlap between annotated passages and user identified ones.

There has also been work showing that teaching/training individuals with high quality annotations or providing tools to make such annotations can yield positive downstream behaviours [8, 12, 13, 17]. Spectator can provide a document type agnostic platform to facilitate this as well as localizing such annotations into a single source for easy use and updating. Internally, we have seen the benefits of combining high quality manual annotations and noisier machine learned annotations to help less experienced users learn about various concepts and topics.

Finally, as an aspirational use case, we envision Spectator (or an application derived from it) being used to help facilitate analysis and research in general. That is, by providing a high-quality document viewer to the community, we hope to remove some of the barriers that arise when researchers are relegated to using commercial software (e.g., Nvivo) to perform annotation/coding and analysis of research data. Such reliance on commercial software can mean that knowledge sharing and transfer can be limited to those researchers with the finances to afford it. Accordingly, it is our hope that the community runs with Spectator (or a tool like it) to build more inclusive and available research platforms and experimental systems.

4 LIMITATIONS

The most obvious limitation of Spectator is that it relies on rendered images and OCR results to present information to users and facilitate annotations. This has several technical constraints in that new document types are not "plug and play" but require rendering to images before they can be used. Moreover, it means that updated documents must be reprocessed and annotations potentially moved over by hand. This limitation results from our production version of Spectator being required to handle scanned documents and not just digital documents. In so doing, we avoid some pitfalls [9] of dealing with the raw documents but create others for those interacting with constantly changing documents.

The other main limitation of Spectator is that it a web-based document viewer and this may limit how it can be used. This is mitigated by the fact that there are technologies that enable embedding web applications as desktop ones (e.g., Electron[7]). While such technologies are not always optimal, they are constantly improving and we believe make Spectator viable even in a desktop-only mode.

5 DEMO APPLICATION

Our demo application is straightforward pipeline to facilitate document annotation of a series of academic papers. Given a set of papers, the pipeline will convert each paper to an image per page (using ImageMagick[8]), run each page through Google's Tesseract OCR software[9] to produce the OCR'd text and necessary positional information for annotation, and will then store all of this information as needed in a lightweight database. A lightweight server is then stood up that serves content for display in Spectator. This directly mimics the flow in Figure 2.

The goal of the demo application is simply to provide a minimal viable example of what can be done with Spectator. Included in this is some basic behavioural tracking of time spent reading a document, amount of content annotated on a document, and the number of annotations made.

Accordingly, we have not gone to great lengths to make a "production ready" application and have done enough to give interested parties a flavour for what is possible and what workflows might look like. In particular, a real workflow would allow processing of different document types and potentially different document qualities (e.g., scans). Such workflows go beyond Spectator and require a more extensive system that we cannot offer at this time.

6 CONCLUSION

We have presented an overview of our document viewer framework, Spectator, and have focused on the design and implementation decisions taken during its implementation. While many of these decisions were motivated by our own internal use cases, we believe that others will find the availability of a high-quality, extensible document viewer to be of benefit. In particular, those cases where developing one's own document viewer is burdensome and secondary to the task at hand. Such use cases include interactive IR systems that have ML/AI components with a human-in-the-loop, document annotation for test collection creation, and providing junior users with pre-annotated examples to learn from.

[6] As a browser's underlying renderer may change over time and so change a page's layout and look.

[7] https://electronjs.org/

[8] imagemagick.org

[9] https://github.com/tesseract-ocr/tesseract

REFERENCES

[1] [n.d.]. collective.documentviewer. https://github.com/collective/collective.documentviewer.

[2] [n.d.]. The NYTimes Document Viewer. https://github.com/documentcloud/document-viewer.

[3] [n.d.]. viewerjs. https://github.com/webodf/ViewerJS.

[4] [n.d.]. web-document-viewer. https://github.com/Atalasoft/web-document-viewer.

[5] [n.d.]. zathura. https://github.com/pwmt/zathura.

[6] Mustafa Abualsaud, Nimesh Ghelani, Haotian Zhang, Mark D. Smucker, Gordon V. Cormack, and Maura R. Grossman. 2018. A System for Efficient High-Recall Retrieval. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval (SIGIR '18)*.

[7] Michael K. Baldwin. 2009. xDOC: A System for XML Based Document Annotation and Searching. In *Proceedings of the 47th Annual Southeast Regional Conference (ACM-SE 47)*.

[8] Aaron Bauer and Kenneth R. Koedinger. 2008. Note-taking, Selecting, and Choice: Designing Interfaces That Encourage Smaller Selections. In *Proceedings of the 8th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL '08)*.

[9] A. J. Bernheim Brush, David Bargeron, Anoop Gupta, and J. J. Cadiz. [n.d.]. Robust Annotation Positioning in Digital Documents. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '01)*.

[10] J. J. Cadiz, Anop Gupta, and Jonathan Grudin. 2000. Using Web Annotations for Asynchronous Collaboration Around Documents. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work (CSCW '00)*.

[11] Mark Harrower and Cynthia A. Brewer. 2003. ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps. *The Cartographic Journal* 40, 1 (2003), 27–37. https://doi.org/10.1179/000870403235002042 arXiv:https://www.tandfonline.com/doi/pdf/10.1179/000870403235002042

[12] Catherine C. Marshall, Morgan N. Price, Gene Golovchinsky, and Bill N. Schilit. 2001. Designing e-Books for Legal Research. In *Proceedings of the 1st ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL '01)*.

[13] Mei-Hua Pan, Naomi Yamashita, and Hao-Chuan Wang. 2017. Task Rebalancing: Improving Multilingual Communication with Native Speakers-Generated Highlights on Automated Transcripts. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17)*.

[14] Beryl Plimmer, Samuel Hsiao-Heng Chang, Meghavi Doshi, Laura Laycock, and Nilanthi Seneviratne. 2010. iAnnotate: Exploring Multi-user Ink Annotation in Web Browsers. In *Proceedings of the Eleventh Australasian Conference on User Interface - Volume 106 (AUIC '10)*.

[15] Ahmed A.O. Tayeh, Payam Ebrahimi, and Beat Signer. 2018. Cross-Media Document Linking and Navigation. In *Proceedings of the ACM Symposium on Document Engineering 2018 (DocEng '18)*.

[16] Peter L. Thomas and David F. Brailsford. 2005. Enhancing Composite Digital Documents Using XML-based Standoff Markup. In *Proceedings of the 2005 ACM Symposium on Document Engineering (DocEng '05)*.

[17] Joanna L. Wolfe. 2000. Effects of Annotations on Student Readers and Writers. In *Proceedings of the Fifth ACM Conference on Digital Libraries (DL '00)*.