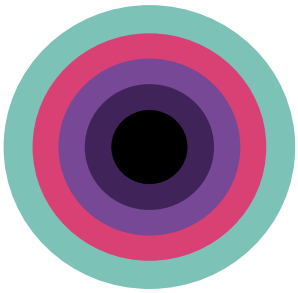


Editoria

Community-owned Book Publishing Platform

This book was written in
a Book Sprint supported
by the Shuttleworth
Foundation

Editoria

Community-owned Book Production Platform

Contents

About This Book

Across three sunny days in the San Francisco fall of 2018, this book was written by fourteen friends, with supporting contributions made by several others remotely. These contributors are each stakeholders within the Editoria community, with their own affiliations to participating organizations. We gathered and produced this book with the shared hope of welcoming, supporting, and enabling other like-minded individuals and organizations within the scholarly communications landscape who are weary of dated proprietary publishing tools and hard-coded linear workflows.

Our appreciation goes to the founders of Editoria—University of California Press, the California Digital Library, and Coko. We recognize generous funding from the Andrew W. Mellon Foundation. We also thank our friends at Book Sprints for their expert guidance in making this book. Thanks also to the Shuttleworth Foundation for funding this event. To all of these organizations and teams and the individual contributors within them: our sincere gratitude!

We welcome new friends all the time to the Editoria community, including Book Sprints as an early adopter, in addition to the American Theological Library Association's ATLA Press, University of North Carolina Press and their affiliate Longleaf Services, University of Michigan's Michigan Publishing and their presentation and preservation platform Fulcrum, and University of Technology Sydney's ePress—to name just a few. There are many others, and there's plenty of room for more, still.

This book is aimed at a wide range of readers within the industry on the publishing, content production, and technology sides; funders and strategic stakeholders making technology or budgeting decisions; project managers, designers, and software engineers; and, really, any lifelong learner with a passion for collaborative publishing technologies!

Acknowledgments

This book was written at the Aspiration offices, the San Francisco Non Profit Technology Center, from 15 to 17 October 2018. We thank Aspiration, Coko's fiscal sponsor, for generously sharing space for us to talk, whiteboard, eat, drink, write, and edit. We also thank Book Sprints CEO Barbara Rühling, who facilitated our process; this book would not have been possible without her. Working remotely, Henrik, Agathe, Raewyn, and the rest of the Book Sprints long-distance support team proved invaluable in keeping the effort on track and moving forward while we slept overnight. Also, Juliana Froggatt copyedited from France for UC Press. Thank you.

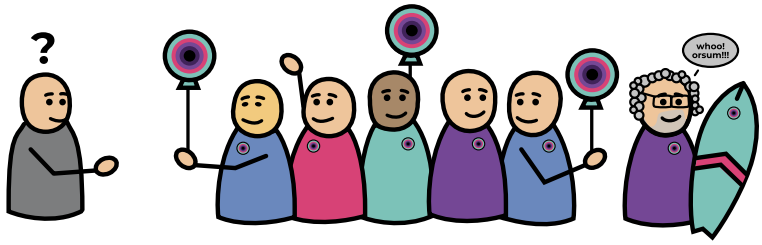
Participating in the Book Sprint were Yannis Barlas (Coko), Kate Warne (UC Press), Justin Gonder (CDL), Christos Kokosias (Coko), Fred Chasen (Paged Media), Alex Theg (Coko), Cindy Fulton (UC Press), Monica Westin (CDL), Adam Hyde (Coko), Jessica Moll (UC Press), Julien Taquet (Coko), Julie Blanc (Paged Media), and Alison McGonagle-O'Connell (Coko). Thanks to all of our organizations for supporting our participation and our time away from the office.

Alex Theg deserves very special thanks here. As the only member of this cohort based in San Francisco, he deftly managed all logistical planning and coordination. We were well fed, supercaffeinated, and generally well rested, as we had beds to sleep in, all without much fuss, thanks to Alex. He even drove some of us around the city.

This book was written about Editoria using Editoria! Here we are drinking our own champagne: Editoria + Book Sprints = zero to book in three days! Further, our plans are to print this book overnight for distribution to attendees at the Editoria Community Meeting on October 18.

Last, we dedicate this book to Alexis, who writes Editoria code. Alexis couldn't be here physically, but he supported us remotely around the clock. We are so grateful.

What Is Editoria?



1

The Skinny

Editoria is a web-based publishing system designed for collaboration on and production of book-length works. Built by publishers for publishers, the system puts control over workflow into your hands and, importantly, takes it out of the hands of third-party vendors.

Leading-edge technology

To return control over workflow to publishers, Editoria enables authoring, collaboration, editing, styling, formatting, review and commentary, automated typesetting, and ultimately multi-format export (including for downstream uses outside the system) all within a web browser. To achieve this functionality, Editoria incorporates "best of breed" technology, being built atop PubSweet, and is an assembly of JavaScript-based modules.

By publishers for publishers

Editoria is open: open source, open community, and an open proposition. This means that the development roadmap is open for your ideas and contributions, as is a vibrant community of forward thinkers from across the scholarly publishing landscape. Participation in the community is critical. The community developed the platform as it exists today, and members are all responsible for architecting and iterating toward the future .

Editoria community members collaborating

Where you belong

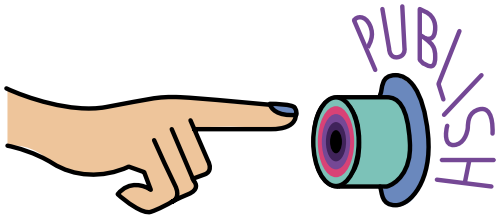
As you hold this book, reading these words, you are already part of the community. Welcome! This is exactly where you belong, and we are delighted you are joining us.

2

Why Should I Care?

As we know, a lack of funds for scholarly communication has meant a curtailing of many forms of publishing. Editoria is here to help by offering a new way to publish niche scholarship efficiently and inexpensively.

Editoria is a browser-based system that facilitates collaboration among editors, authors, and in-house staff in making books. Editoria's in-browser, real-time collaboration has particular advantages over legacy systems, replacing the need to use Track Changes in multiple Word files and keep track of changing versions, and the need to share those files as email attachments or via FTP with potential delays or confusion among users. With Editoria, publishing staff and authors can view, edit, export (as EPUB or PDF), or print a paginated version of a book at any point.



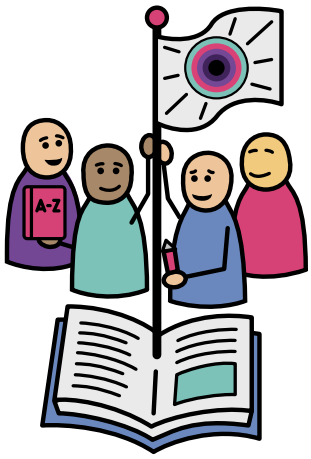
Pushing the button

Consequently, Editoria offers independence from expensive, upgrade-driven systems. You can, for example, do away with Microsoft Word's costly licenses and the need to update macro suites to keep pace with the latest releases of the software or of constantly evolving operating systems. Editoria also frees you from reliance on vendors to produce HTML and other formats from .docx, a process that is costly, time-consuming, and deliberately opaque.

Editoria helps you create and customize professional production workflows that can be scaled to publish many books in ways that suit your needs instead of the needs of the software vendors. For instance, batch copyediting is easier to manage; the Bookbuilder allows all team members to see at a glance who is working on which document and what parts of the book are available for them to work on. And designers can create and deploy CSS style sheets that support particular visual designs and serve specific reader needs. Automated paging from within the system means the ability to avoid the considerable costs of professional typesetting services, including time spent trafficking files and communicating with vendors.

Adopting Editoria gives your team the chance to innovate, to move beyond traditional bookmaking while maintaining professional standards and best practices for editing and typesetting.

But Editoria is not just software—it is a community, and the community drives the evolution of the system as cocreators of its tools. This kind of participation not only gives you the power to improve the product but also means you own your own tools, and you gain the benefit of belonging to a broader knowledge-sharing collective .



Reclaiming the publishing infrastructure

3

Where Did It All Begin?

First and foremost, Editoria is a community. We believe that making good software is a conversation, and we also believe that people making books shouldn't work alone. The Editoria community consists of (but is not limited to) university presses, library publishers, and other folks working in scholarly communication who share a common interest in modernizing book production workflows. We are often motivated, at least in part, by a shared interest in taking back the control of academic production from corporate interests.

Editoria began as a grant proposal jointly submitted by University of California Press (UC Press) and the California Digital Library (CDL), which recognized their shared challenges in supporting scholarly monograph production. University presses in particular struggle to support these valuable and necessary publications—particularly in the humanities and social sciences—because each costs between US$15,000 and US$40,000 to publish and the titles rarely sell enough copies to cover that amount (see "The Costs of Publishing Monographs: Toward a Transparent Methodology," by Nancy L. Maron, Christine Mulhern, Daniel Rossman, and Kimberly Schmelzinger [http://sr.ithaka.org/?p=276785]).

Two areas where costs can be reduced are infrastructure and workflow. University presses are often limited by expensive, outdated infrastructure that does not facilitate collaboration and does not natively support the simultaneous creation of print and digital formats. Library publishers often lack any production infrastructure at all and can't provide the tools needed by campus-based book publishing efforts.

Time and resource expenditures can be reduced if editors are able to move away from exchanging and trafficking files and toward working in a shared, web-based collaborative platform. Finally, copyediting, layout, and proofreading

costs can be substantially reduced by relying on a templated system that collapses the copyediting and proofreading stages and automates page layout.

With generous funding from the Andrew W. Mellon Foundation, UC Press and CDL chose the Collaborative Knowledge Foundation (Coko) as a technology partner to help build both the Editoria platform and its community. Along the way, additional funders and organizations have helped shape the project. For example, the Shuttleworth Foundation funded the development of Paged.js, an export component used by Editoria.

The first release of Editoria was designed to meet the specific needs of UC Press editors. CDL is now working on a flavor of Editoria to suit the specific needs of the library publishing community. Book Sprints also has its own radically fast workflow built on top of Editoria. You can adopt and use any of these configurations "out of the box" or leverage Editoria's modular nature to construct your own ideal publishing platform.

As of press time, we are about to convene the first ever Editoria Community Meeting in San Francisco. Dozens of people from across library publishing, university presses, and scholarly communications will discuss workflow, attend demonstrations of the latest version of Editoria, and ultimately determine the future of book production workflow.

4

Some Key Concepts

Before we go on, there are a few terms we need to define. The following are key concepts for Editoria that will be referred to throughout this book. We have added Editoria-specific detail when appropriate.

Open source

Open source is a type of license that grants liberal rights to anyone to use, modify, and redistribute a piece of software's source code. Editoria uses the MIT license [https://gitlab.coko.foundation/editoria/editoria/blob/master/LICENSE.md] . In effect, this means that anyone can access Editoria's source code for free and change it as they like.

Typically, open source projects also have a style of operating that privileges the contributions of developers over those of other participants. However, Editoria's community is deliberately designed *not* to function like this. We welcome participants of all skill types and prefer input from use-case specialists (sometimes referred to as *end users*). This means we welcome participation from you!

Community

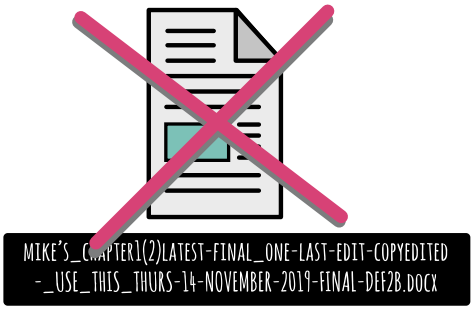
The term *community* gets used a lot in this book! The people who use and contribute to the improvement of Editoria make up our community. We

welcome your participation—again, you do not need to be a developer to participate. We welcome use-case specialists who are enthusiastic and want to use their experience with Editoria to improve it.

Single source

Confusingly, "single source" has nothing to do with source code. Instead, it is the idea that the content of a chapter or other part of a book will look the same to everyone at any give time, no matter which user is altering it. There is only one, canonical, version of the chapter or book, which is shared online with all users who have access to it, and any changes made change that version.

For example, if I open a chapter in Editoria, edit it, and then close it, I have changed the same text that everyone else who comes after me will edit. This is not true of a workflow in which one person changes a Microsoft Word file (for example) and then saves it with a different name and sends that version to the next person to work on. In that case, if the first person edits the original file again, the second person will not have access to those changes.



Single source, multiple output

Linked to single source are advantages to storing your content in HTML. Because Editoria uses one canonical version of the content, and that content's format is HTML, it is possible to export to many, many other formats at the push of a button. HTML, being the most common content format in the world, has the advantage of very many conversion tools built to convert it to other formats. Editoria leverages this so that you can easily export to many formats quickly and cheaply.

Chief among these content conversions in the world of books is output to ePub (which itself uses HTML as an internal format) and to PDF. PDF output in Editoria has been tackled by a special type of magic called Paged.js. This JavaScript library can convert the Editoria HTML content into beautiful, print-ready, book-formatted PDF. It is very magical, and we have documented this conversion extensively in this book.

Automated typesetting

Also known as *automatic typesetting*. There is no such thing as true automated or automatic typesetting in the sense that a design can be applied and every single book will be rendered perfectly without human intervention. However, this term is still useful and describes a process in which books are exported using a standard design template and come out almost perfect, with minimal alterations required. Such is the case with Editoria.

Editoria uses Paged.js, an automatic typesetting tool developed by the clever people of the Paged Media Initiative [https://pagedmedia.org/] . Paged.js, at the push of a button, will take a book from Editoria and render a preview in the browser by paging the content using CSS style. Then the browser print dialog can produce a book-formatted PDF. The results will look very good but will probably require small adjustments. This process is covered later in this book.

Semantic markup

Semantic markup means applying the correct named styles (e.g., Heading 1 or Subtitle) to parts of a text rather than adding arbitrary font sizes and weights to achieve a similar appearance.

You will have used systems that support semantic markup and non-semantic markup. Microsoft Word, for example, uses both. In mixed systems, something that looks like a heading may be either marked up correctly, for example as an H1 (Heading 1), or given arbitrary characteristics, such as 24-point font size and bold weight. The difference is enormous. If the heading is created with a specific named style, that helps with many things, including the targeting of design specifications (specs) to the element. However, if the heading is created with arbitrary characteristics, it is very difficult to automatically apply specs to it (specific font size and weight, for example) when exporting to PDF, etc. Editoria is not a mixed system; it allows only the application of specific named styles. This prevents users from applying arbitrary characteristics and protects the use of the same styles uniformly across the entire book. As an added bonus, uniform application of styling makes books produced with Editoria more accessible!

Note: Sometimes in this book we use the term semantic HTML. This means the same as semantic markup when referring to a HTML file that has been marked up correctly (semantically).

Workflow

Workflow refers to the passage of a text through various processes from creation to completion. Throughout these processes, various people with specific roles (e.g., author, production editor, copy editor) perform tasks at specific moments. Some workflows are more linear and prescriptive than others. A university press, for example, typically has a strongly linear workflow, whereas a Book Sprint or an author working alone has a very flexible workflow.

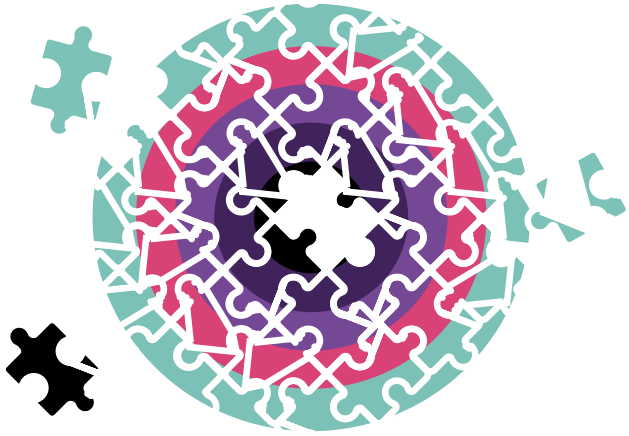
In general, publishers have historically had very few options when designing workflow. Editoria offers many new opportunities for flexible, intentional design.

PubSweet

PubSweet is the back end upon which Editoria is built. For more information about PubSweet and the suite of products that use it, see Technology > Product Suite [https://coko.foundation/product-suite/] on the Coko website.

Track changes

Track changes here refers to a text-editing tool similar to those found in other word processors, such as Microsoft Word. This tool marks deleted or added text with underlining and different colors so that users can easily see it. Depending on their assigned roles within the book team, users can accept or reject these edits and make new, tracked revisions to the text.



Modular

Editoria is not a single, monolithic system. Its file-ingestion, editing, and export components are individual, modular elements within a suite of products that publishers can select à la carte to use or not. This also means that each individual element can be customized or replaced to meet each publisher's needs *without* compromising the rest of the system, and the community can share and collaborate on these component versions.

WYSIWYG

WYSIWYG, the acronym of "what you see is what you get", is frequently used to describe text-editing interfaces. In Editoria, the editor component (called Wax) represents the semantic structure of the document being edited (showing, for example, which text is a heading and which is a block quote). While Wax does not show the design—the final look of the text as it will appear in the printed

book—it does strike a careful balance between visually representing styles and providing enough space for Editoria's powerful collaboration features.

Installation options

Local install

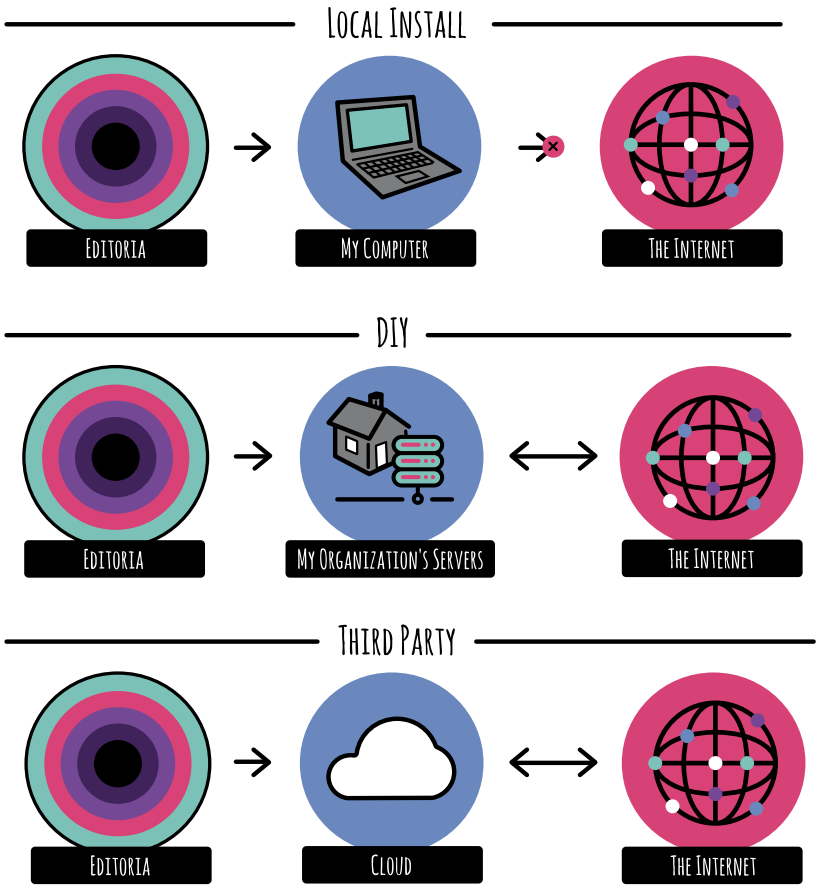
As Editoria code is freely available from GitLab [https://gitlab.coko.foundation/editoria/editoria] , anyone with some tech skills anywhere can install and host it locally on their own computer. In this scenario, the code lives in the local machine, which is generally the only place where this running instance of Editoria can be accessed. The user is responsible for maintenance, security, and updates.

DIY

An organization that has tech resources and has installed Editoria on its own server has performed a DIY installation. The server may be hosted on the organization's premises or remotely. In this case, Editoria can be accessed from anywhere on the internet (unless it is blocked by firewalls). The organization's IT department is responsible for maintenance, security, and updates.

Third Party

Organizations can also decide to delegate full responsibility for hosting and maintaining an instance of Editoria to an outside group. This can include managing maintenance and updates. There are any number of groups that could be an appropriate choice for providing hosted deployment services.



Three types of hosting

5

Quick Start Guide

We know some of you want to get started right away before reading all the documentation. This quick start guide is for you!

Once you sign in, you arrive at the book **Dashboard**, where you see all the books on which you are working. Here you can

- create a book
- delete a book
- rename a book
- open a book to begin working on it, taking you to the Bookbuilder

Within the **Bookbuilder**, you can

- specify team members and assign them roles (project editor, copy editor, author) for this specific book
- upload files, by batch or individually
- set page-break designations
- reorder chapters and parts of the book using drag and drop
- use the status bar to track status and to pass a component from one user to another
- open a part of the book for editing in the Wax editor
- export to a varity of output options

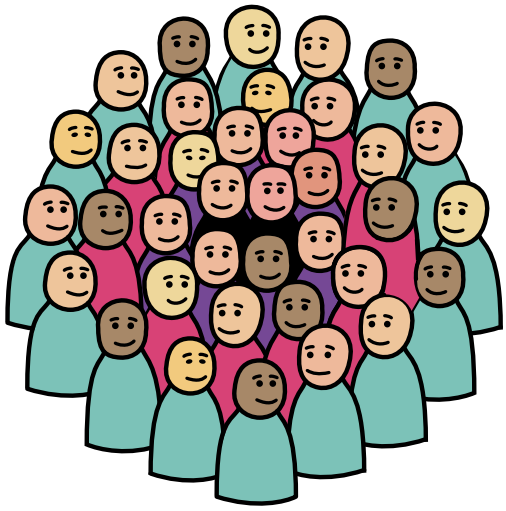
Within the **Wax** editor, you can

- write content
- add styling and formatting
- leverage word processing tools to edit text and review changes

Tip: When the first line of text in a chapter is styled as a title in the Wax editor, it then appears as that chapter's title in the Bookbuilder—check it out!

Good luck! If you get stuck, read the rest of the book or reach out to the community for help.

Community is You, Us, We



6

We Are All In This Together

A philosophy of contribution underlies Editoria's community governance and informs the meaning of *open source* for this shared, distributed project. There are consequently a number of ways you can participate in the Editoria community, described in more detail later in this book.

The open source ethos of Editoria extends beyond the codebase to all those who work with the software to produce and publish books. This can mean sharing code openly for reuse, but it can also mean sharing experiences, expertise, new workflows, working methodologies, and feature proposals.

In a very real sense you're not getting the platform for free—rather, you are also expected to contribute your experience and expertise back to the community. That is, in effect, the cost of being involved with Editoria. You benefit from it, but you also pay into it with your involvement.

As an example, one of the ways that you can participate in the Editoria community as a use-case specialist is to propose a new feature. This process is open to anyone. The community will comment on the ideas you propose, to help improve them. If you have an idea for how Editoria could work better, sharing these ideas, creating feature proposals, and listening to the feedback will not only help you but help others using the platform in the future.

The Editoria team is here to help

There are actually people employed to work on Editoria. The team includes facilitators, community managers, developers, and CSS specialists. These folks

are part of the community and are also there to support it. The Editoria team is happy to help you with everything from demos to introductions, from understanding the code to optimizing workflows—whatever you need. Don't be shy! There is information later in this book about how you can reach out to the team; a good place to start is with the Editoria Community Manager (alison@coko.foundation).

A word on *users*

In the software world, there is a legacy polarization of those who make the software and those who use the software. This is especially prevalent in the open source community, where developers are the primary—sometimes the only—solution providers, and users are those who consume developers' work. The culture and language of open source have largely led to the exclusion of users from the design and production of the tools they need, and consequently they are sometimes seen as parasites on developers' work. This paradigm is not true for the Editoria community. Here the user is, by design, as important as everyone else, perhaps even more important. Users are the people who know best what changes can improve Editoria.

We would like to dispense with the holdover language that devalues the input of the people using the software, but changing language is a difficult task. So we are consciously using the term *user* for someone who is a valued use-case specialist. And by *use-case specialist*, we mean you!

7

Editoria Is Yours Forever

Editoria's code is open source and currently held by two trusted organizations: University of California Press and Coko. Neither of these has a wish to be acquired or to sell Editoria. That's all we need to say about that.

If you wish for more assurance, it is good to know that the open source license cannot be retrospectively revoked. So what you have now, you will always have. At the time of writing, Coko is working toward a further layer of security by locking the requirement for the source code to be open into a new organizational constitution: if the articles of constitution require the code to remain open, then it will be extremely difficult to change the license even if the organization is acquired.

But again, Editoria will be open forever. Some of us have literally slept on couches to get us here; we aren't about to sell out now.

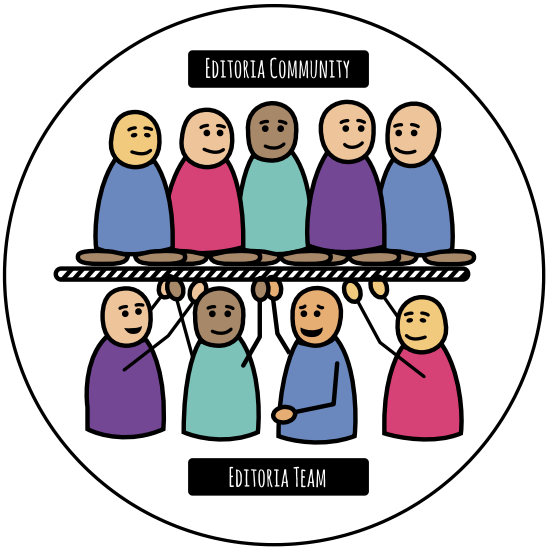
Coko is also currently developing membership models to aid Editoria's long-term sustainability and is conducting an ongoing search for funding. Please consider becoming part of these efforts! A good starting point is to contact the Editoria community manager.

Further, Coko, as the primary steward of Editoria, does not intend to compete with any service provider in its communities, whether the book or the journals community. We want to seed commercial services in the Coko community, not compete with them. We want to help you succeed! So should you wish to start offering Editoria services of any kind, come talk to the Coko folks.

Coko is also invested in growing and strengthening the community by facilitating regular events that bring people together to determine Editoria's future, leaving the processes for development entirely open. Anyone can

participate by adding their own vision—even if you lack development resources. We have designed a f eature proposal process (documented later in the book) that enables people without development resources to get features they want built. This process is designed to be inclusive and transparent.

Editoria and Coko are not storing any personal data, because neither is a provider of services that require personal data. Whoever is responsible for installing Editoria is responsible for storing information and making decisions about data ownership and access. Coko will endorse and work directly with only those services that respect data privacy requirements.



8

How You Can Contribute

The Editoria community is an active arena, and you are encouraged to jump in! Some ways to get involved are listed below. There are sure to be many additional ways to contribute that we haven't even thought of yet. We all help each other out, because when one of us moves forward, we all do—this is the spirit of our community.

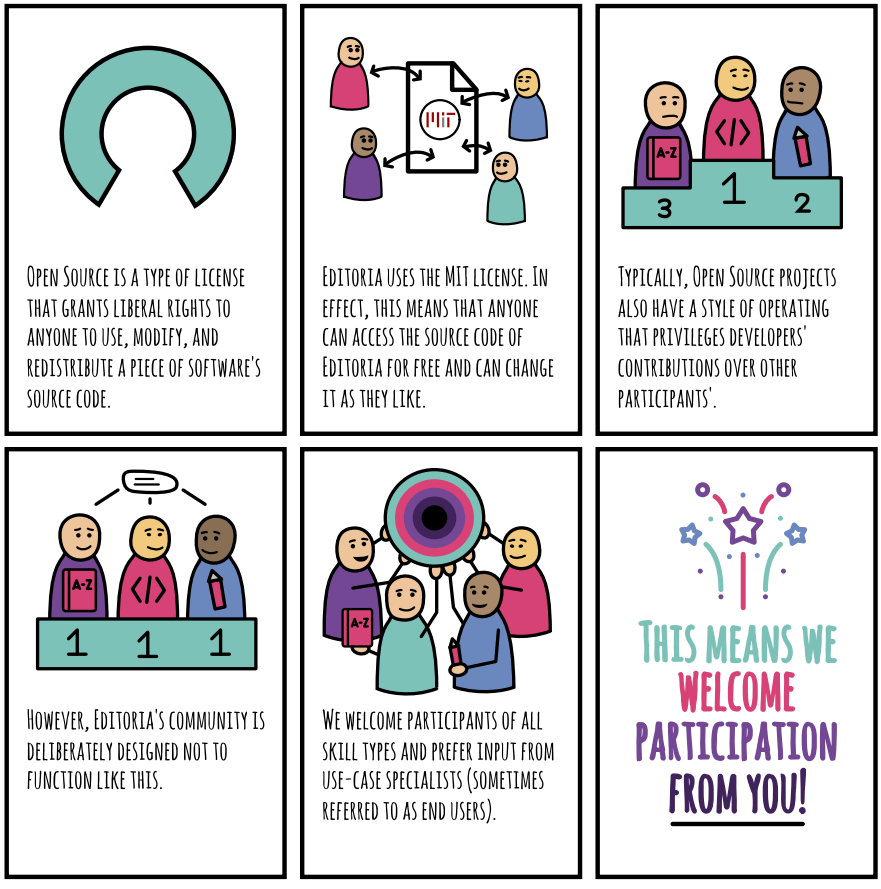
You can contribute to the community by

- attending Editoria events
- hosting an event, including webinars
- sharing meeting space with the community
- blogging about Editoria
- sharing thoughts about Editoria in social channels
- demonstrating Editoria to friends and colleagues
- presenting about Editoria at conferences
- commenting on Editoria issues in GitLab
- proposing features in GitLab
- creating and sharing documentation
- obtaining and sharing funding for continuing development
- helping others on Mattermost [https://mattermost.coko.foundation/]
- contributing code
- contributing CSS templates

Technology-provider organizations can

- build modularly to support integration-use cases
- spread the word broadly within the tech community

The Editoria community manager (alison@coko.foundation) is here to help you in any way possible!



9

Where Is Everyone, Everything?

When you join the community, naturally you'll want to know what support is available as you work in Editoria. Where can you turn for help? What are the available resources for troubleshooting?

Don't worry—we've got you covered.

First, if you are new to the Editoria community, one of the best people to reach out to is the Editoria Community Manager, Alison (alison@coko.foundation). You can also engage via the channels listed below:

Editoria email list

The Editoria email list is useful for sharing info about events we are all at or for quickly calling for broad opinions or help. It can also be used to share other information, ideas, etc., in a way that's more personalized than other channels that are totally public to the web. You can find and subscribe to the email list here [http://lists.coko.foundation/listinfo.cgi/editoria-coko.foundation] .

Mattermost

Mattermost is a online chat platform similar to Slack (if you know it). This is where you can find everyone. The community hangs out here and chats about everything and nothing. Whether we're making small talk or solving problems collaboratively, Coko's Mattermost [https://mattermost.coko.foundation/login] is a window onto the wider community supporting Editoria. Join to see for yourself!

Should you run into trouble, or if you simply have a bug to report or need help setting up the application, this is the place to go. Sign up for the service (which is, of course, free) to participate in the chat. There is a dedicated Editoria channel that anyone can join. Simply shout when you join, and your new community will be happy to help you out!

Readme

Instructions on how to get the code and run Editoria can be found in the repository's readme file [https://gitlab.coko.foundation/editoria/editoria/blob/master/README.md] . You can run the application either on a server or locally on your computer (the latter is usually more useful for developers).

The recommended operating systems for running Editoria are Linux for server deployments and either Linux or Mac OS X for local hosting. Editoria comes with Docker containers to make deployments easier. These are optional, but should you choose to use them, make sure Docker is set up on your server. Refer to the readme. Coko can also quickly set up demo deployments for organizations to try out Editoria before making the decision to jump in.

GitLab

GitLab [https://gitlab.coko.foundation/editoria/editoria] is where developers can find all the code for the different parts of the application. A project's code repository is a great place to assess the health of an open source initiative. Visit ours and review commits, issues, and other key elements of the vibrant community that supports Editoria and other Coko tools.

Also, Editoria has bugs! *Gasp!* But it's okay—so does every software! If you have a bug that you want to submit, simply open an issue on Editoria's GitLab repository. The developers can then pick that up and work on it. Generally, bugs get higher priority than feature proposals.

10

How to Make Feature Proposals

You can propose new features for Editoria. This process is open to everyone, especially to those organizations or individuals without development resources. The Editoria team has specifically designed the feature proposal process to be nontechnical. It is intended primarily for people who use Editoria and want to see it improved to better meet their needs and the needs of others.

Organizations or individuals with development resources that intend to build something should not follow this process but instead open a request for comment (RFC) if the code they want to develop is intended for inclusion in the Editoria core (see the next chapter).

Bug reports are separate from feature proposals and should be logged as such.

To comment on anything below or to put in a request, please visit GitLab [https://gitlab.coko.foundation/editoria/editoria] .

Who can create a feature proposal?

Anyone can propose a new feature: individual or organization, commercial or noncommercial, with closed or open content, libraries, publishers, publishing staff, authors, reviewers . . . Anyone and everyone!

The scope of proposals

There are no hard rules for the scope of a proposal, beyond a best effort to make it easy to understand.

In general, a proposal can be as large or small as you feel it needs to be. If there are numerous distinct features, then these should all be separate proposals. If there is a large development that requires lots of changes to realize, then it is preferable that this be a single feature proposal. For example, numerous small changes to the Bookbuilder should have one feature proposal per item. However, if the proposal is to create an entirely new Bookbuilder interface or a complex new feature with many facets to extend the book export workflow, then these larger developments should, at first, each be written up as a single feature proposal.

Proposals that will affect any part of the Editoria workflow should be made as feature proposals.

Proposal process

Before you can lodge a feature proposal, you must first create an account in the Gitlab repository [https://gitlab.coko.foundation/] .

Then create a new issue here [https://gitlab.coko.foundation/editoria/editoria/issues] .

The proposal title must be created with the prefix “Feature Proposal:” followed by a short title, e.g., *Feature Proposal: Add logout button to Bookbuilder.*

In the description the proposal must state the organization or individual making it. This is to help the Editoria core team keep a fair distribution of attention among participating organizations.

The proposal should then include a short summary that indicates the goal of the feature and the roles affected. These statements provide a sense of who the feature's primary user is and what they are trying to achieve.

The main body of the proposal must describe the feature, preferably from the user's viewpoint. No technical details need be included unless it is a technical proposal (for example, recommended standards for EPUB accessibility should contain the relevant technical details). The aim of this section is to spell out the proposal as clearly as possible, in a very readable format, from a user's perspective. Screenshots or quick sketches of what the feature's user interface will look like (in rough form—a photo of a hand-drawn sketch is completely acceptable, for example) are very much advised if appropriate.

The community manager (Alison) is available to help community members write and submit proposals.

Discussion and voting

The Editoria team encourages the community to comment on proposals (in GitLab, using the comment feature within each proposal). These comments help to improve what is proposed. Subsequently, the community votes on the proposed developments via the up and down votes in GitLab.

Review process

Every three months the Editoria Core team will

1. review the proposals
2. ask for more details where necessary
3. make an initial triage (Adam and Alison)
4. decide what will be included in the next quarter's roadmap (Adam and Alexis)
5. create a table in the Editoria readme documenting the roadmap
6. announce the roadmap on the Editoria.pub site
7. close any feature proposals that did not make the cut for that quarter's roadmap (these may, however, be reopened and resubmitted at a later date)

How decisions will be made

The Editoria core team will consider each proposal on its own merits. There will not be a formal metric system (having the highest number of GitLab yes votes will not automatically guarantee the acceptance of a feature proposal, for instance, although this will have a positive influence). However, the team will look at forming a coherent roadmap, choosing items in a consensus-based discussion with the following factors in mind:

1. Proposals made by organizations or individuals that use Editoria in production and are active in the community will get the highest attention.
2. Proposals made by two or more collaborating organizations or individuals will get more attention that those made by one organization or individual.
3. Proposals that get more community discussion and review (up votes included) and respond to community input will get more attention than those that do not.
4. Clear proposals will be preferred over less clear proposals.
5. The Editoria core team will endeavor to choose proposals for the quarterly roadmap that represent a broad spectrum of the community.

Who makes the decision?

In reviewing and selecting feature proposals, the Editoria core team will be facilitated by Adam Hyde and will include the Editoria community manager (Alison) and the Editoria lead developer (Alexis). The group will also include others as needed (e.g., Christos for decisions involving Wax).

11

Developing a Feature by Yourself

If you are a developer or part of an organization with developers on staff, and you wish to extend or otherwise change Editoria, then you are very welcome to do so. There are several options. One, available to anyone, is to fork the code and create your own ideal version. Another, more community-minded path, is to contribute to the Editoria core code. We believe that most development will occur with the latter intention, so we want to help ensure that your code lands well and is merged quickly into the main repository. There is a simple process that we hope you will follow:

1. Before you start writing any code, create a new issue in the Editoria main repository [https://gitlab.coko.foundation/editoria/editoria/issues] . This must have the prefix "RFC:" (request for comment) and be followed by a short title, e.g., *RFC: New Bookbuilder component.*
2. Describe in as much detail as possible the proposed feature, including wireframes and technical implementation details. This should be written so that the Editoria core team and greater community can quickly evaluate the proposal.

After submitting your RFC, you can expect quick feedback from the Editoria core team. The community at large is also encouraged to comment on the proposal. The idea is to get as much feedback as possible to help you improve your approach before development begins.

This process is designed to assist you in developing features that will match the Editoria approach well, and prevent possible duplication of effort, such as by pairing you with others to lessen your development burden if possible.

We intend this process to be quick and responsive. If you are not getting the feedback you need within the time you allot, then reach out to the Editoria core team directly through the issue in the repository or through the Mattermost chat [https://mattermost.coko.foundation/] . Thanks for considering this option!

12

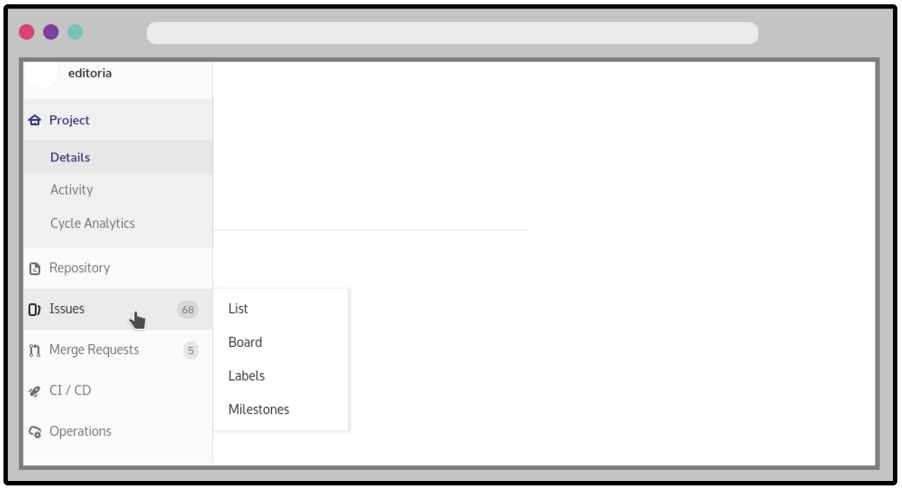
Bug Squashing

A software bug is an error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. One of the skills that it's very good to learn is how not to be frightened of bugs! Bugs are a normal part of software; all software has them.

As a member of the Editoria community, there are two things that you need to learn about dealing with bugs. The first is how to make good bug reports. The second is how to find workarounds so that you can continue working while the bug gets fixed. Of course, if you happen to be a developer, you can also help fix the bug!

Learning how to make an effective bug report

To make a good bug report, you must become a little familiar with GitLab, where all Editoria bugs are recorded. GitLab is a code repository designed for developers, but making bug reports in GitLab is very simple for everyone. First, visit Coko GitLab [https://gitlab.coko.foundation/] and create a new account. Next, go to the Editoria "main repository [https://gitlab.coko.foundation/editoria/editoria/] ." Access issues by clicking the issues link on the left side of the screen, or you can go there directly [https://gitlab.coko.foundation/editoria/editoria/issues] .



Locating the bug reporting screen

Anybody is welcome to create a bug report here. Don't worry about making mistakes (we believe there is no such thing). The community will help you improve how you use these tools as we go.

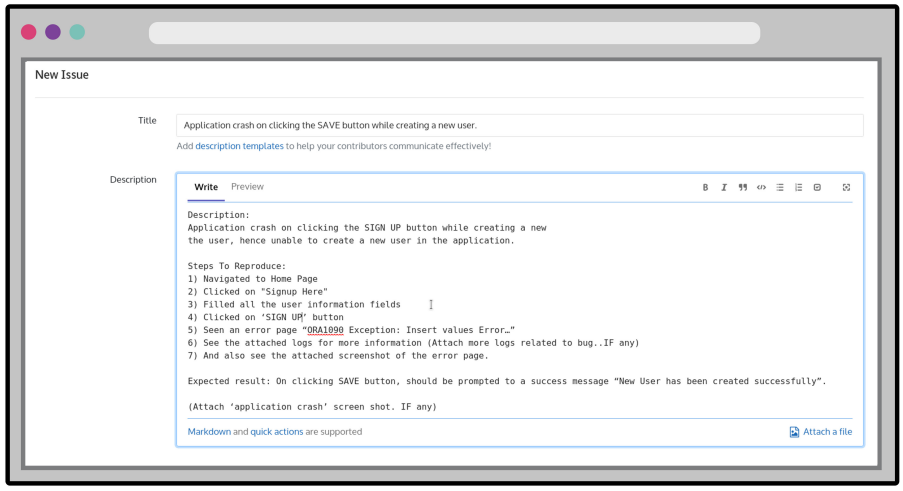
To initiate a bug report, select the green "New issue" button at the top of the page. Enter a title in the title field and a description in the description box. To report a bug, you will use the title naming convention: "Bug:" followed by a short, snappy title describing the bug. For example: "Bug: The entire Bookbuilder page is pink." The title shouldn't be an essay, and should be kept short and snappy. Eight to ten words is best. If you go above or below that, once again, no worries - just report the bug and we'll work it all out.

After you've created a title, you will enter the description of the bug. You don't need to provide any technical detail, unless you know it. You do have to describe what you experience in terms of what you are trying to achieve, what you expected to happen, and what actually happened. A common format that is very useful is as follows:

"As an Author I want to use Bookbuilder to organize my book chapters.

Expected behavior: I click on link to Bookbuilder and I can access the Bookbuilder.

Actual behavior: I click on the link to the Bookbuilder page, and the Bookbuilder page is completely pink. No features are visible. Its all pink."



Creating a new issue

Of course, this example is a little silly, and you'll likely need to communicate more detailed information than this. When doing so, remember you are writing the information to help someone who is trying to help you. They can best help you if you provide clear information so that they can best understand your problem. For example, include contextual information that will help somebody else understand the problem, such as: where in Editoria you are (Bookbuilder, Wax editor..), what role you are when trying to perform the task (author, copyeditor etc), and what you are trying to achieve.

A badly written bug report for the above issue provides none of this context and information. You may have seen these kinds of reports, they look something like this:

"Bookbuilder is broken."

These kind of reports are not very helpful; and above all else, when we file a bug report we are trying as best we can to be helpful.

If possible, giving additional information such as screenshots of the problem (these are especially useful) and the name of the organization providing you an Editoria instance, as well as the deployment's URL - and even the URL of the page where you are experiencing the problem, can be incredibly helpful. Additional information, such as your browser type and version, as well as information about your computer's operating system, can be helpful too. ***Never* give login/password information as these bug reports are public.** When adding screenshots or diagrams (also useful) to the bug report, click the "attach a file" link and upload the image(s). The process of adding screenshots isn't intuitive, but give it a try, and you'll get used to it!

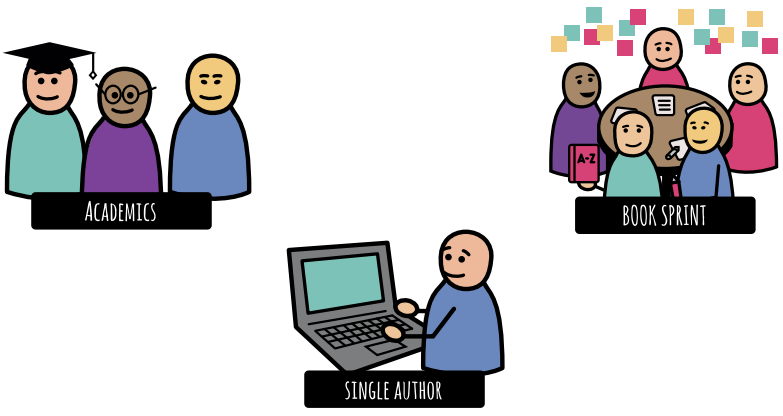
Once again, bug reports are welcomed, and we thank you for giving your best attempt to make them understandable by people who can fix them. If you are someone who can fix bugs, please feel welcome to jump into the main repository and try to fix some.

Bugs are seldom showstoppers. Please try to find workarounds, if you can, and share them (in the comments on the bug) with others while the bug is being identified and fixed. Sometimes there are many ways to do things with Editoria, and they aren't always obvious, so try a few things to see if you can solve your problem using the software in another way. We know this process can be frustrating, so we thank you for your patience.

Please keep an eye on on your issue, particularly looking out for comments, as this is where information related to the bug, including questions for you, may be added. However, this area is not designed for chat or debate, so we try to keep the noise level in this area to a minimum. (Mattermost is great for chatting, though!) In general, if you want to be someone who is involved with bugs overall, monitoring this space and adding helpful detail to bugs (regardless of who entered them) is a valuable contribution. Sharing your expertise and experiences will help move everyone forward.

Finding bugs is the first step to killing them, and submitting good bug reports is extremely valuable. We look forward to your future bug reports!

Optimal Use Cases for Editoria



13

Supported Workflows

Editoria has already been used in two very different workflows that represent the opposite extremes of a spectrum.

On one end of the spectrum is the traditional post-acquisition, linear workflow of a university press. In this workflow, manuscripts accepted for publication pass through sequential steps: project editor assessment, copyediting, author review of editing, cleanup, and, finally, export to the composed book. This was Editoria's first use case (for University of California Press), and when you install it, it is configured for this process.

On the other end of the spectrum is what we call a flat or concurrent workflow, used in Book Sprints, in which many authors work at the same time on various chapters, each having the same capability to write and revise during the same creative session.

Editoria can be used for both of these models, or for other workflows that might combine different aspects of the two. As mentioned earlier, for example, CDL is customizing Editoria to better suit a library use case. If you'd like help figuring out how Editoria can optimize your workflow or how to further extend or customize Editoria, then reach out the the Editoria team and community for help.

14

Types of Content

The word processing capabilities provided by Editoria's Wax editor (see the chapter on editing in Wax) are quite sophisticated. Books created with Editoria can include

- figures and images, with captions
- tables and graphs, as images
- links
- code snippets
- footnotes or backnotes
- in-line notes
- break ornaments
- Unicode and other special characters
- special formatting for blocks of text (e.g., epigraphs, extracts)

Editoria also supports many languages (both left-to-right- and right-to-left-reading). Adding Unicode character sets to the Wax editor is relatively simple.

Editoria and Paged.js provide sophisticated, customizable controls over the exported book in the form of CSS rules applied to the elements (titles, extracts, headings, etc.), but since the exported book is generated in a rules-based manner, custom tweaks to the layout of individual pages require some experience in CSS book design. Future features in the Paged.js pagination engine will make page-specific tweaks easier, and the Paged.js community is organizing workshops to help publishers with this part of their process.

While the Editoria platform can be customized and expanded, if you wish to do the layout in Editoria using Paged.js then it is best suited to text-and image-centric books. In general, the less the content relies on special layouts and highly customized pagination (as in an art book, for example), the better it will fit Editoria's end-to-end workflow at present. However, if necessary it is always good to know you can export to InDesign ICML if necessary to support legacy design workflows (covered later in the book). In time this will become less and less necessary as we add more and more features to Editoria and Paged.js.

15

Authors and Teams

Editoria can be used effectively by teams or individuals for a variety of workflows.

Single user

A single user can use Editoria to author, edit, and export to paginated format. Text can either be uploaded from Word .docx files or composed directly within the Editoria platform.

Production team

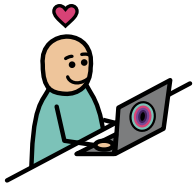
With different permissions configurations, Editoria can be a powerful collaboration tool for teams to manage multiple people with different roles.

Although Editoria does not support multiple users concurrently editing a single chapter, it does allow for multiple users to work on separate chapters at the same time, with each chapter locked to others while it is being edited. In time, concurrent chapter editing will be added but most publishers in the community prefer not to have it.

Editoria also supports multiple users with different roles: author, production editor, and copy editor (covered later in the book). A status bar shows where in the production process each chapter is, and it controls the access each user has

to that chapter at any given moment, helping to prevent miscommunication or stepping on someone else's toes. For instance, when the copy editor is working on a given chapter, the author does not have permission to go in and make changes to it; only when the copy editor has finished and passed the permission to the author can the author make changes to the chapter. Editing and revision tools, such as track changes and margin comments for discussion, allow for communication within the word processor.

Using Editoria



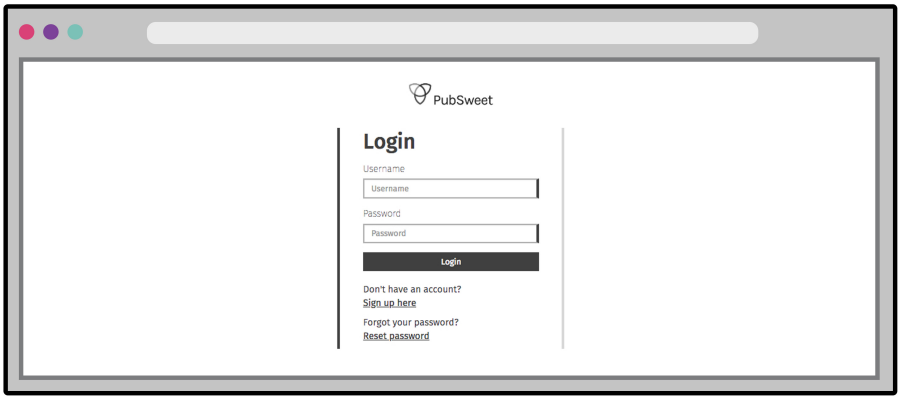
16

How It All Fits Together

Editoria comprises many parts. This chapter is a high-level look at the basics. This is helpful for understanding the documentation that follows, and for adding clarifying detail to any feature proposals you may make.

Login page

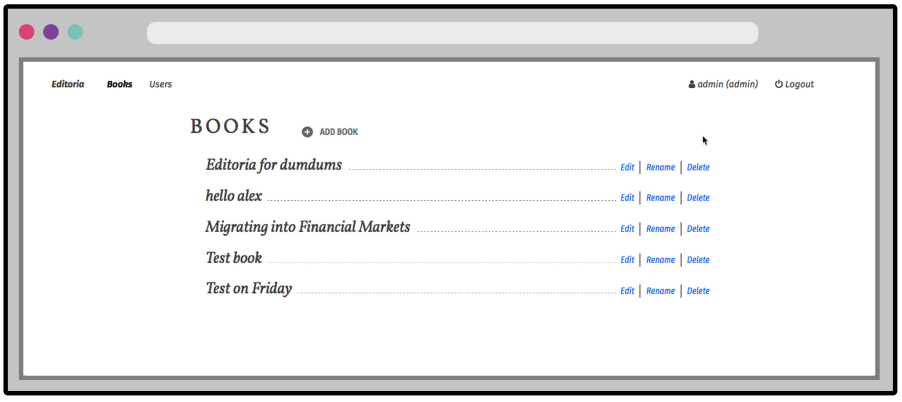
On the login page you can create a new account or log into an existing accoun t.



Login screen

Dashboard

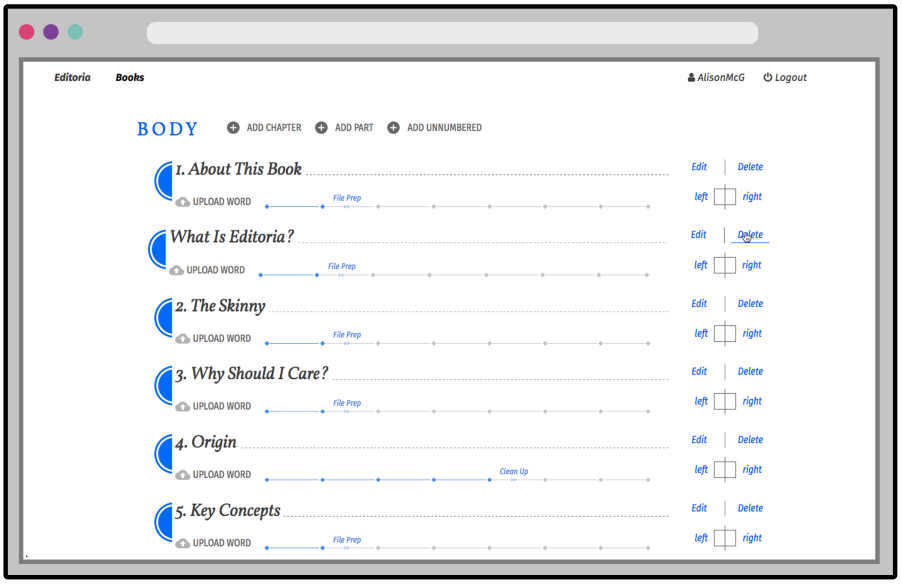
Once logged in, you will be at the Dashboard, which contains a list of all the books to which you have access. If you have the appropriate privileges, you can create, delete, or rename books here.



Dashboard

Bookbuilder

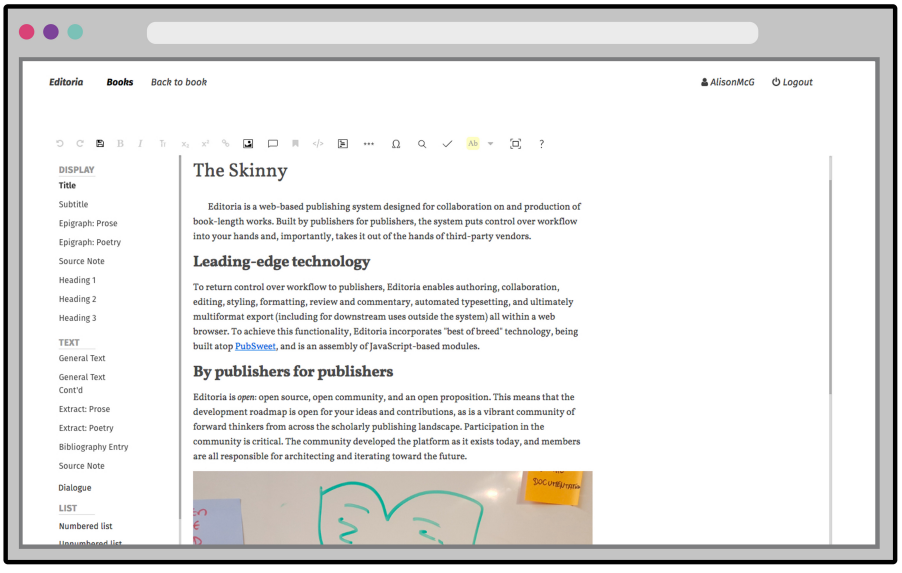
When you click on edit next to the name of a book, you will go to the Bookbuilder, which serves as a platform for uploading and organizing the contents of the book, specifying pagination, and tracking workflow status.



Bookbuilder

Wax

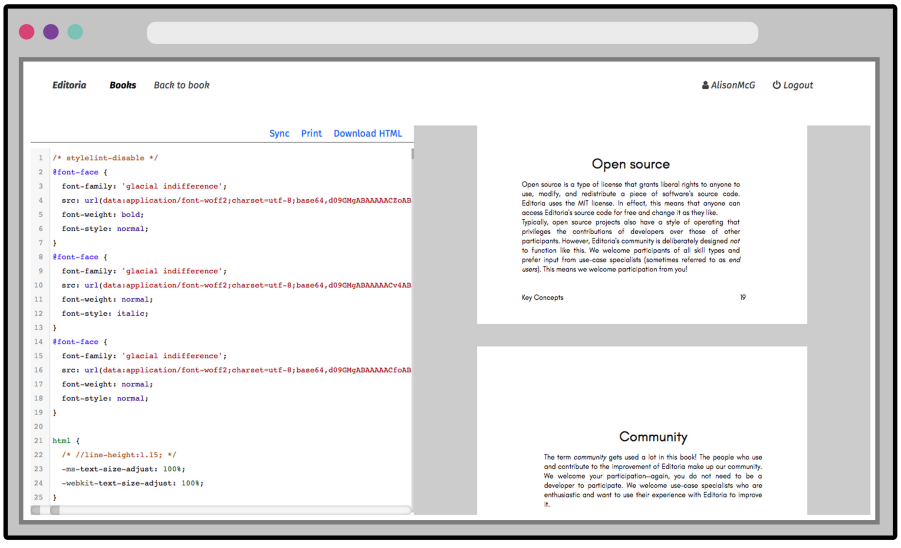
Wax is the content editor for Editoria. Here is where you can edit and format content, place images, and communicate with other members of the book team. You can also create content in Wax.



Wax screen

Paged.js

There are several ways to export your book, but here we mention only Paged.js, as it is the sole tool with custom styling support built into Editoria at present. (Later chapters cover the other formats you can export.) Paged.js is very sophisticated, so we have included an entire section on how to use it later in the book. That section is intended for those who want to or can already understand and use CSS.

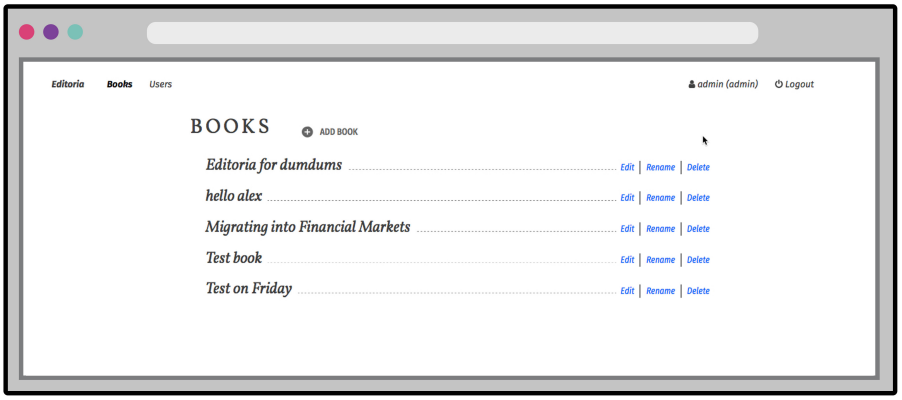


Paged.js screen in Editoria

17

The Mighty Dashboard

Once logged in, you will see the Dashboard. The Dashboard displays the list of books to which you have access in Editoria. From here, users with the proper permissions can edit, rename, or delete books.



Book dashboard

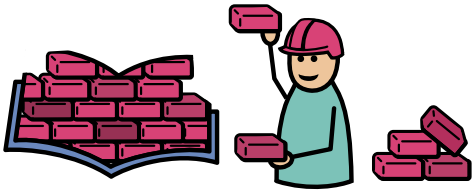
What you see and can do here depends on your role (covered later in this book):

- Administrators can see all books.
- Production editors, copy editors, and authors can see only the books on which they have a role.

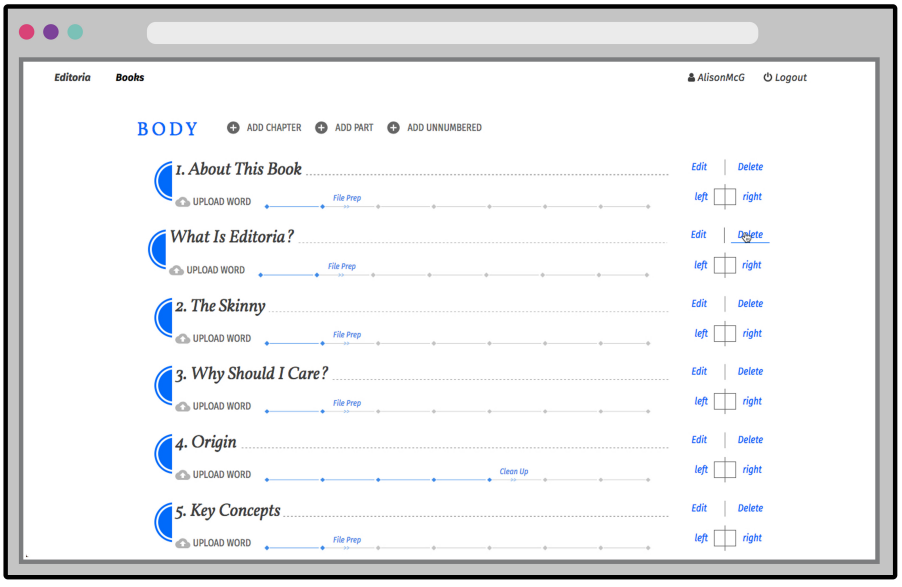
- Administrators can add, rename, and delete any book and access any book's Bookbuilder via the edit button.
- Production editors can add, rename, and delete their books and access their books' Bookbuilder via the edit button.
- Copy editors and authors cannot rename or delete books. They can access the Bookbuilder via the edit button.

18

Building Your Book



When you click on edit next to a book title in the Dashboard, you are taken to the Bookbuilder. The Bookbuilder is where you organize your book. Here you can either upload your Word files or, if you are not working from a previously written manuscript, create fresh, empty chapters directly in the Bookbuilder.



Bookbuilder screen

Adding book components

New items

If you wish to create a new, empty part, chapter, or other book component in Editoria, click add part or add chapter in the Bookbuilder's appropriate section. From there, click edit and start writing! See the chapter on Wax for more details about the editor.

Import

You can also import Microsoft Word .docx manuscripts into Editoria. Files can be uploaded individually or in a batch. If only some of your content is in Word format, it may be best to upload the individual Word files one at a time. To do so, create a new, empty front- or backmatter component or body chapter and select

upload word. After you choose a file, its content will be converted to HTML (this might take a minute or so) and made available in the new component.

Batch file upload

Multiple Microsoft Word XML (.docx) files can also be uploaded via the Bookbuilder. Click on the upload word files button at the top of the page. As soon as you have chosen multiple .docx files, sit back and watch the magic! New chapters appear, forming the structure of the book, and the content is converted from .docx to HTML, ready for editing.

Naming conventions for imported files

The batch upload feature uses file naming conventions to identify both the type of component each file will be (part, numbered chapter, unnumbered component) and which division it will upload into (frontmatter, body, backmatter). To make sure that your components are uploaded into the correct division, you need to change the names of files that you import into Editoria. The rules are as follows:

- Names of .docx files that begin with "a" will be uploaded into frontmatter.
- Names of .docx files that begin with "w" will be uploaded into backmatter.
- File names that start with any other letter will be uploaded into the body.

The file order you created outside the system will be preserved in the Bookbuilder.

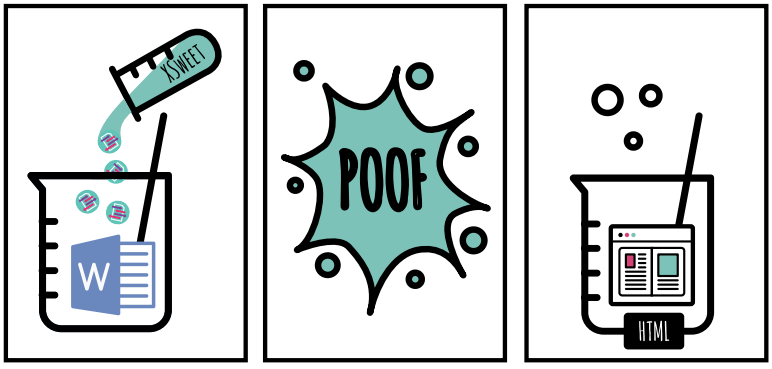
Files uploaded into frontmatter and backmatter are generic book components. The body has three categories of components: numbered chapters, unnumbered chapters, and parts. Further rules can be applied to files that belong to the body, so that they can be correctly categorized. These rules are as follows:

- If there is an "00" anywhere in the file name, the file will become an unnumbered chapter.
- If the file name has a " pt0" anywhere, it will become a part.
- Otherwise, the file will upload as a normal numbered chapter.

The Bookbuilder will display the file names as they are imported. Renaming book components is not handled in the BookBuilder but inside the Wax editor. Refer to the chapter on editing content for more detail.

Replacing MS Word macros

Cleanup operations on content are very often handled by editors through Microsoft Word macros. This section will explain how many common ones are already handled automatically in Editoria during import.



The upload process, the typical first step in creating a book from author files, is done with XSweet, Coko's transformation tool to port the contents of Word documents into Editoria. A Word file's underlying markup is quite messy and complex. Through a series of sequential steps (in discrete XSLT files), XSweet reduces the Word XML to only the useful information—the text, formatting, semantic styles, notes, etc.—and converts it into HTML for Editoria.

However, XSweet does much more than simply extract content from .docx files—it is also a sophisticated tool for enhancing and manipulating text. Many publishers rely heavily on Word macros, which can be a pain point: hard to create, available for only certain machines, and needing maintenance with Word version upgrades. In contrast, XSweet's chain-of-XSLTs architecture is designed to make it easy to add and modify functionality, by editing existing steps,

rearranging their order, or creating new steps. To illustrate what is possible, here's a partial list of enhancements that exist today:

- Recognize plain-text URLs and turn them into hyperlinks
- Apply headings and semantic markup, based on visual formatting such as font size and bold weight
- Apply common copyediting cleanups to text as it is uploaded, such as converting double spaces to single ones
- Convert hyphens between numerals to en dashes
- Remove any number of spaces before or after em dashes
- Convert series of three periods to ellipses
- Replace pairs of adjacent hyphens with em dashes
- Convert en dashes surrounded on both sides by spaces to em dashes
- Surround each equal sign with one space on either side
- Remove spaces adjacent to tabs
- Remove spaces at the beginning and end of paragraphs
- Remove tabs at the end of paragraphs
- Remove empty paragraphs
- Convert single and double straight quotation marks and backticks to smart quotation marks
- Insert hair spaces between single and double quotation marks
- Force punctuation marks to match the formatting of the previous word

For more information and comprehensive documentation about XSweet, visit the XSweet website [http://xsweet.coko.foundation/] .

Indexing

A quick note on building indexes. While the table of contents is generated automatically in Editoria during export by tools such as Paged.js (see later in the book), the program currently has no tools for building an index. At present, indexing must be done in the traditional way from a paginated PDF file. In the future, an indexing tool may be available in the Wax editor to mark up text and automatically generate an index as part of the book's export.

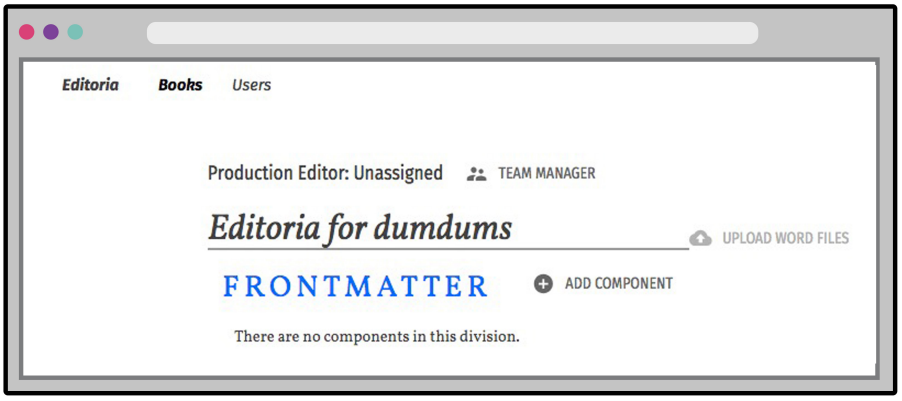
19

Assigning the Book Team

For each book, the project editor assigns users to the team via the Team Manager, determining their roles for that book. A participant's role determines what access they will have to the components throughout the workflow.

The Team Manager

The Team Manager is found in the Bookbuilder, just above the book title.



Team manager

By default, Editoria has the following roles in the system: admin, project editor, copy editor, and author. We believe that these will cover most use cases without any need for customization. However, should you need to accommodate different roles (as well as their corresponding titles) within a specific organization, or the permissions granted to those roles, you can. Turn to the Editoria community if you need assistance.

Admin

When Editoria is set up, a single administrator, or admin, must be created. The admin has access to all parts of the application, from users to chapters and books. The idea is that if you are an admin user, there is nothing you cannot do in Editoria. However, this role is not meant to be an active participant in the book project. It is most likely to be occupied by a member of an organization's IT department or a managing editor, who should be able to have the permissions to help other users out.

Production editor

Sometimes also referred to as project editors, production editors have the highest level of permissions of all the roles that actively participate in Editoria book projects. They can create new books, as well as manage books that are assigned to them.

A production editor who creates a new book automatically becomes the production editor for that book.

Production editors also have the ability to manage teams that are working on their books. They can assign a copy editor or add an author. They also have a certain level of granular control over who can do what when, through the workflow status tool for each chapter. For example, a production editor can make sure that an author does not have write access to a chapter until after the copy editor has done a pass on that chapter. Production editors can also turn on track changes for any stage, so that no changes go unnoticed.

Finally, project editors can at all times create, upload, delete, and add content to components.

Copy editor

Copy editors are assigned to books by production editors. The role is defined at the book level, meaning that a person is a copy editor for only the books to which they have been assigned as a copy editor.

Copy editors have book access similar to that of project editors but are slightly more constrained. They see only the books that they are assigned to, of course. They have write access to book components only during the edit and cleanup stages of each component. Copy editors cannot access the Team Manager, and thus cannot assign other users to their books.

Like project editors, copy editors can add new components to a book and turn track changes on or off.

Author

The role of author is the most restrictive. Like copy editors, authors see on their Dashboards only the books that a project editor has assigned them to. They do not have write access to a book's components until the project editor has advanced the workflow status to "Reviewing." When an author has write access to a component, the changes to it are always tracked.

General access

All users can add comments to a component in the Wax editor, even if they don't have permission to edit the content itself. This way, users can have a discussion around the text regardless of their permissions at any stage of the production process.

To reiterate the point that was made at the beginning of this chapter, the above roles and permissions are what UC Press and the Coko team have found to be sensible defaults. This does not mean that they are written in stone or that they cannot be bent (or changed completely) to fit another organization's different needs. However to do this you will need some help, and the Editoria team and community is here for exactly this.

20

Editing Content

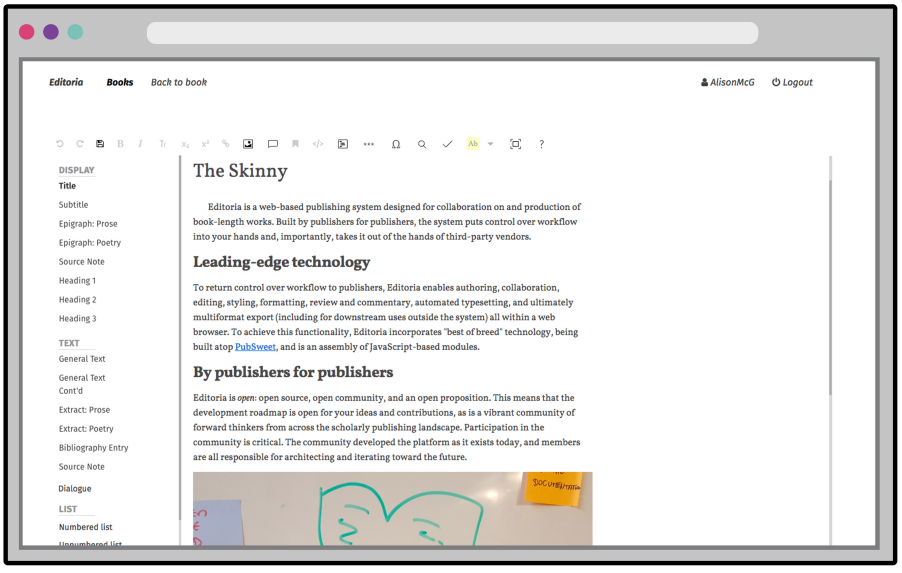
After uploading your files or creating empty components, you are ready to move to the editing stage. In the Bookbuilder you can click edit and you will be taken into Wax, the web-based word processor that is built into Editoria.

If you imported files, all the content and the document structure that the Word-to-HTML conversion picked up (e.g., line breaks, character formatting) will be represented here.

Its important to understand that Wax provides an environment that looks similar to a book's final design (e.g., heading formatting is distinct), but it is not an exact WYSIWYG interface. The main difference is that additional space is present in Wax to accommodate features such as track changes and commenting. On a final export, these features are not needed and will not appear. To see what the finished book will look like, export the book from the Bookbuilder.

Only one user can edit a component (e.g., part, chapter) at a time in Wax, and some tools and functionality might differ depending on the permissions setup and the current role assigned to the user.

So what can you do in Wax?



Wax

Styling

On the left-hand side of the screen is a panel with all the block or paragraph styles that can be applied to your content. These named styles apply the correct semantic markup to each element.

In order for the exporter to correctly identify and display different elements, the correct corresponding styles need to applied in the editor. Applying the correct named styles is both easy and necessary.

Renaming components

Applying the title style will not only mark up the selection, but will also rename the current component in the Bookbuilder.

Component names in the Bookbuilder can only be changed through applying this style in the editor on a selection. This is done so that Editoria can guarantee that what you see in the Bookbuilder is exactly the same as what exists in the content itself.

Formatting and inserts

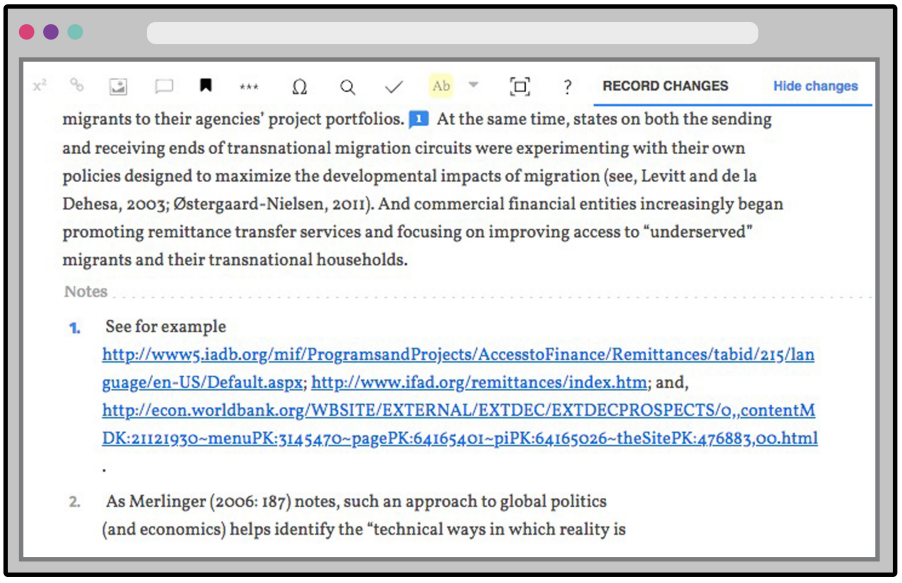
Above the editor is a toolbar with character and in-line styles. The current character formatting options are bold, italics, small caps, superscript, and subscript. There are also tools for inserting hyperlinks, ornaments, and special characters; highlighting text; and adding syntax-highlighted code snippets.

Illustrations

Images (including figures and illustrations) can be added from the toolbar, and each can have a caption. If you want a caption, it is important to use the image's caption interface rather than writing a free sentence under the image. This will enable Editoria to use the correct design specifications when exporting to PDF, etc.

Notes

Notes are displayed at the bottom of the screen. The size of the notes panel can be modified by dragging the horizontal line separating it from the general text.



Notes

The formatting toolbar contains a button for inserting a note where the cursor is in the text. This will create a note reference in the text and open an appropriately numbered note in the notes panel. Deleting a note reference will also delete the note itself. Subsequent notes will automatically renumber. You can also insert note references inside another note's text.

While the Wax interface shows notes at the bottom of the screen, upon export all notes will appear at the back of the finished book rather than at the foot of the page or at the end of each chapter.

Track changes

When turned on (and not everyone can turn track changes on or off—covered later in this book), the track changes feature tracking text that is added, deleted, or moved. Inserted text is blue; deleted text is red and struck through. In file

preparation and copy editing, the feature may be turned off by the production or copy editor.

Hovering over a change displays the name of the user who made the change. A future modification may be to assign different colors to edits according to the users' roles.

To see what the text would look like if all changes were accepted, the markup can be temporarily hidden. This is a purely visual aid and has no effect on the actual content.

Comments

Authors and editors can communicate with each other by using the Wax comments feature. Selecting text will make a comment bubble appe ar to the right. Clicking on it will open a writing area. Type in your text and hit the return key to save your comment (if you click outside the comment window before hitting return, your comment will be lost).

Keyboard shortcuts

There are a number of keyboard shortcuts mapped to different functions of the Wax word processor. Click on the question mark on the formatting toolbar to see the current list.

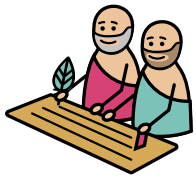
Search and replace

Click on the toolbar's magnifying-glass icon to open Wax's search and replace window.

Full-screen mode

This view provides more space for your writing surface, especially useful for laptops with small screens. Click on the Fullscreen icon on the formatting toolbar to toggle this view.

Wax can be configured to have different combinations of tools and styles or to have custom functionality added to it. Turn to the Editoria community if you need a modified version of the editor.

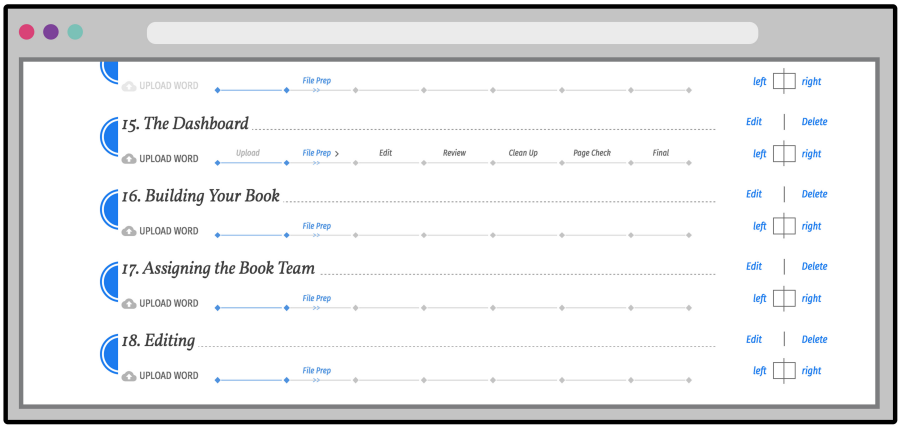


Wax, if you are interested, was named at a Coko team meeting in Athens, Greece, after the reusable wax tablets the ancients greeks used to write on...which are, arguably, the first text editors.

21

Managing Workflow

A status tracker in the Bookbuilder traces the progress of each book component as it moves through an editorial workflow, from upload to final export-ready form. When an assigned participant completes a work stage, they use the tracker to move the component to the next stage, then notify the next workflow participant that they may begin work on that stage. Underlying each stage are permissions that govern who can advance the status. Hover over a component's status tracker to display all the steps in the workflow.



Workflow status tracker

UC Press uses the following workflow stages:

- Upload
- File prep
- Edit
- Review
- Cleanup
- Page check
- Final

Below is a detailed explanation of this workflow and how the status tracker controls user permissions throughout the process.

Baseline permissions

Whenever a user does not have permission to edit a component, the edit button changes to a view button, which opens the Wax editor in read-only mode.

All users can always leave comments, regardless of whether they currently have permission to edit. All users can also always toggle whether tracked changes are visible or hidden (this is not the same as turning tracked changes on or off).

Production editors can always edit a component and can always advance or roll back its status. Production editors and copy editors can style whenever they can edit; authors can never apply styling.

A warning appears anytime a user is about to advance a component's workflow status to a state they cannot roll back.

Workflow stages and permissions

Upload

Only production editors and copy editors can create new components. Both can also batch-upload Word files or add individual components and upload Word files into them. Once an upload is complete, the status tracker automatically advances from Upload to File Prep.

Only production editors can open a new component in the Wax editor and add content—the copy editor can upload but cannot edit directly in this stage. Once a production editor saves some text in a component, the status tracker advances to File Prep.

File Prep

During File Prep, the production editor semantically tags the content using the left-hand styling pane, makes any desired edits, and leaves comments, instructions, or queries for the copy editor or author. The production editor can also toggle tracked changes on and off .

During this stage, copy editors and authors cannot edit the component or advance its status to Edit. Only the production editor can specify that File Prep is complete, by advancing the status to Edit. This grants the copy editor permission to begin working on the component.

Edit

The Edit stage is where the copy editor does the majority of the copyediting, leaving notes or queries for the author as desired. The copy editor can toggle tracked changes on and off, either tracking changes for the author to review or making them silently (small grammatical fixes, for example, don't need to be called out for the author to see). Once the copyediting is complete, the copy editor advances the component's status to Review, at which point they can no longer edit. Simultaneously, the author is granted editing permission and can begin reviewing the copyediting.

The author cannot advance the status from Edit to Review—the copy editor must be the one to certify that this stage is complete.

Review

In the Review phase, the author reviews any tracked changes and comments from the copy editor and the production editor. The author can edit, but tracked changes is locked on in this stage. This ensures that any changes the author makes will be visible to the copy editor and the production editor. The copy editor cannot edit during the Review stage.

As the author reviews the text, the previous and next tracked change commands assist in moving quickly through the edits and accepting changes with which the author agrees. If the author is happy with all the edits in a block of text, they can select and accept multiple changes at once. Accepting a change is the author's confirmation of consenting to the edit.

The author cannot reject tracked changes from the editors; otherwise, the author could undo the editors' work. Instead, an author who does not agree with a proposed change leaves a comment on it to reject the change or to discuss it with the editors. Authors also cannot accept their own changes, as that would defeat the purpose of locking tracked changes on.

When the author's review is complete, they pass the component back to the copy editor by advancing the status to Clean Up. The copy editor can also advance move this stage to complete on behalf of the author, if the author indicates he or she is done, but hasn't advance the status. The author can then no longer edit the component or change its status.

Clean Up

The copy editor regains editing permissions, including toggling track changes, during Clean Up. This is the stage when they review any comments from the author and resolve any outstanding issues. If necessary, the copy editor can roll the status back to Review so the author can make or review further edits.

Once the component is clean, with no tracked changes or comments, the copy editor advances its status to Page Check.

Page Check

Page Check is an open-ended stage at the end of the workflow for final changes or review following the copy editor's cleanup. Only the production editor has editing permission at this stage.

The production editor can roll the status back to Clean Up if necessary, but only the production editor can advance the stage to Final, indicating that the component is ready for export.

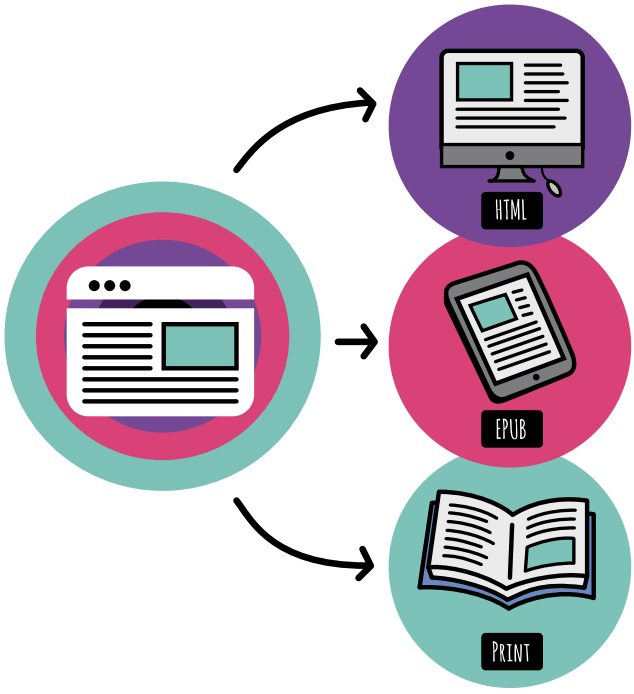
Final

Ready for export! Only the production editor can roll the status back from final.

22

Exporting to Various Book Formats

Editoria already includes the capacity to export to EPUB and to paginated previews that you can save as PDFs. But since it uses semantic HTML to manage the content of the book, it's the perfect starting point for exporting to any file format.



Editoria can export to multiple book formats

PDF

To export a PDF from Editoria, simply select an exporter from the export book dropdown in the Bookbuilder, then select go to preview the book in your browser using either Vivliostyle or Paged.js. In the Vivliostyle preview, use Chrome's browser print dialog to generate a PDF. In the Paged.js preview, use the Print link on the page to trigger Chrome's print dialog for the paginated book. For both, be sure to set "Margins" to "none" and select "Background graphics" for a 1:1 PDF. Then you can save as a PDF. You can also use any pagination engine as an alternative.

EPUB

You can also download an EPUB copy of your book from Editoria. EPUB files are self-contained, with text, images, fonts, and styles. The format is an open standard for electronic books that can be read by e-readers without the need to retrieve content from the web.

Other options

Since EPUB includes all the necessary data, it's also a good starting point for transforming a book into a variety of other file formats, using a tool such as Pandoc, the Swiss Army knife of converting any marked-up file format. If you have book content produced with Editoria and you want to give it to a designer who works in InDesign, after downloading an EPUB copy and installing Pandoc [https://pandoc.org/] , you can transform your EPUB file into an ICML file, Adobe's InCopy file format, which InDesign can use, retaining styles names and markups.

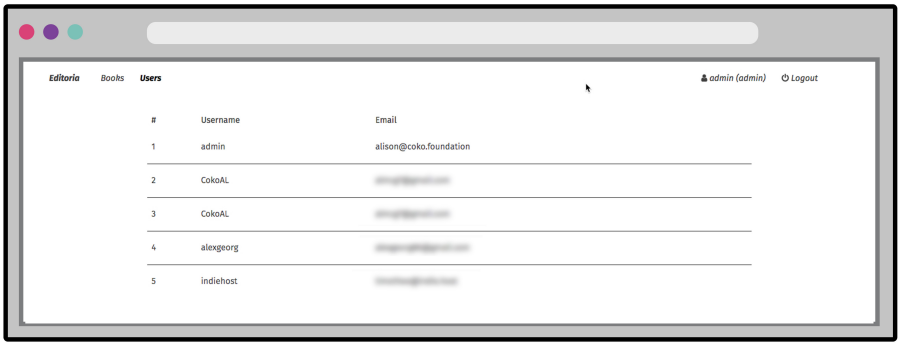
```
pandoc -f epub -t icml -o book.icml epubname
```

Pandoc supports many kinds of exports and retains the formatting for the languages that handle semantics (such as Markdown or HTML) within each export, making it easy to transform the content.

23

Extra Admin Features

On the Dashboard, admins (administrators) have an additional option on the left-hand side of the top navigation bar. (see the chapter on assigning the book team for the basic admin role.) When an admin click on users, a list of all registered users and their email addresses appears. Anyone listed here is available to production editors for assignment to roles on book teams.



Admin view of users

There is a caveat for new installations of Editoria: because roles are specific to books rather than to users, Editoria will not consider a user a production editor (and thus able to create books on the Dashboard) unless that user is assigned to at least one book as a production editor. The easiest way to accomplish this is for the admin to set up a dummy book and add all users who should be the production editor on any book as production editors on this book.

Afterward, these users will be able to add new books to Editoria. As soon as they've each created a book (and been automatically assigned as the production editor for that book), they can be removed from the dummy book and still retain their permissions.

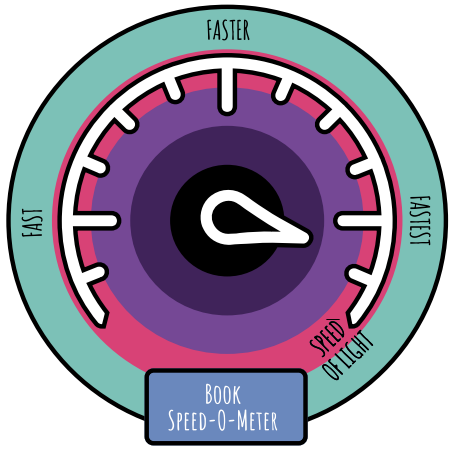
24

A Rapid Book Production Example

The previous chapters in this section describe how Editoria works for the UC Press workflow, which is Editoria's default. This book, however, was created with a version of Editoria supporting a Book Sprint workflow. In this version, permissions are different: no one is assigned the copy editor role, and several features of the Bookbuilder and the Wax editor are turned off (track changes, for example).

The point is, not only can Editoria be extended with new features, but what is already there can be customized to better suit different workflows. This is both a complex topic and, currently, a complex undertaking, since such tweaks will need the help of a developer, and there are no point-and-click interfaces (yet) to assist with these kinds of configuration. However, it is possible to configure Editoria, and if you wish to do so, then ask the Editoria community how to approach this.

At a later date, there will be simple point-and-click interfaces that will enable customization of Editoria to the nth degree. We will get there, and maybe you will help. But in the meantime, the default setup is capable of supporting a lot of workflows. If you aren't sure about yours, reach out to the team and the community and ask for advice.



Editoria supports rapid book production with Book Sprints

The Magical Paged.js

25

Automated Typesetting inside the Browser

Publishing has come a long way since its early days, when composing a page was about arranging tiny pieces of lead on a printing press to create readable text on a page. However, the paradigm of the printed page still dominates online publishing.

Not only does publishing now not require circulating a hard copy of a document, but digital publishing means that content is not bound to its styling anymore, thanks to standardized markup languages. This is exactly what's happening with web-based publishing: a markup language (HTML) defines the content of a document in a semantic way, and a styling language (CSS) is used to format this content.

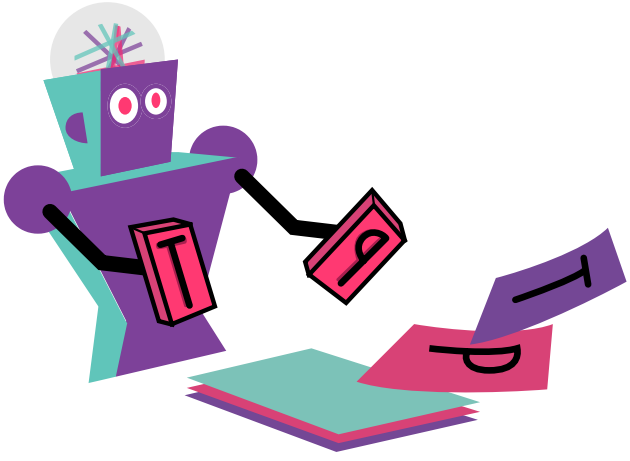
When a browser renders the content of a web page, it's doing a couple of different tasks: It starts by reading the HTML markup and co-relating the styles from a CSS file to the content. Once it does that, it automatically applies the design specs (typesets) to the content on the page.

Consequently, when you look at a web page, you're looking at so-called automated typeset content. And this is what Editoria uses to make layouts in the browser.

Web designers create rules in CSS style sheets that describe the way types of content should appear. These rules can be reused, which can save designers an enormous amount of time that would otherwise be spent repeating the manual application of specs over and over again. However, there are inevitably exceptions to the rules—something doesn't look quite right, etc.—so some

elements of automated typesetting are usually (but not always) overwritten with bespoke rules.

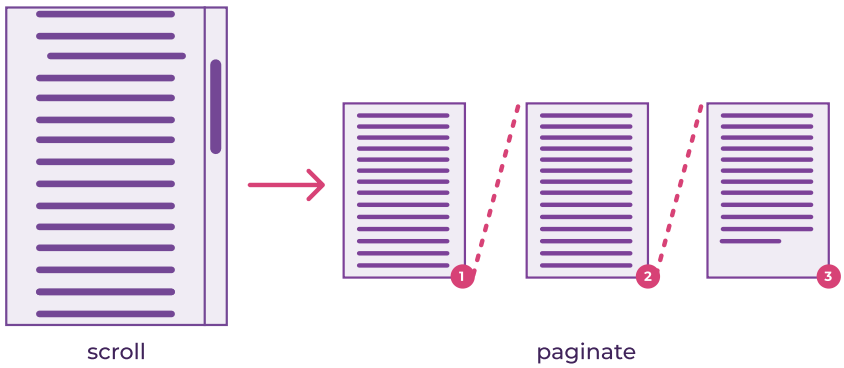
The following chapters explain how to use CSS to design books using Paged.js. This is important information, but please be aware that this content is not for everyone. The target audience includes designers familiar with CSS who wish to design books.



26

Designing with Paged Media

When content is prepared in Editoria, each component is presented as a continuous scroll of text and images. To print this as a book means fragmenting the content into fixed-size pages, flowing the content from page to page.



Scroll to paginated text

Additionally, a CSS style sheet is necessary for designing web pages for print. To define print styles, you must include the `@print` declaration within your HTML file or linked style sheet. The styles declared in this media query will be applied only when the web page is printed from the browser print dialog and

saved as a PDF. All the usual CSS specifications can declare font size, margins, and paddings of elements, colors, styles, etc.

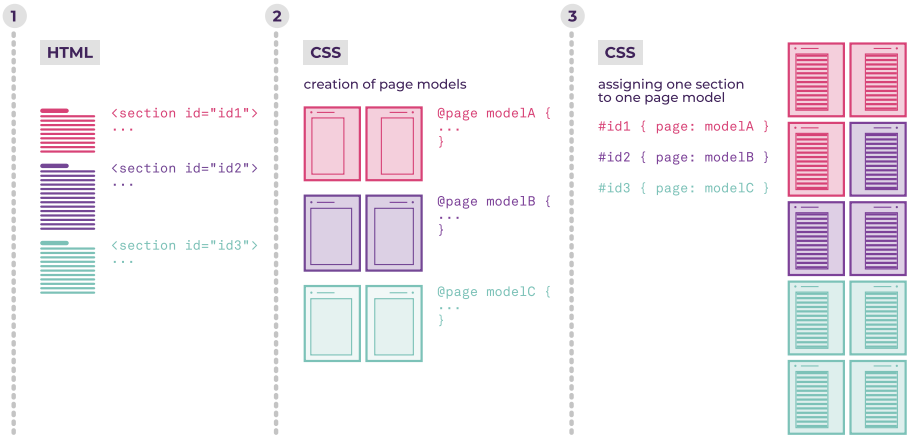
It sounds simple, but designing a book or a print-ready PDF requires thinking about web content in terms of pages. This requires a web designer to perform quite a mental flip, from formatting for the screen to formatting for print PDF.

Pagination

In CSS, the way to fragment the content into pages is described in a World Wide Web Consortium (W3C) specification module called the CSS Paged Media Module Level 3 [https://www.w3.org/TR/css3-page/] .

This module was created to deal with printed matter and Paged Media, and it proposes some basic pagination control features, such as page margins, page size, and orientation. The module also covers other specific book features, outlined below.

The CSS Paged Media Module describes the way a page model partitions a scroll flow into discrete pages. A section of the HTML content is associated with a named page model in the CSS. The specific page layout is applied when this element of the section is encountered. A document can have several different page models. When final output is rendered, pages are automatically created until all the content of the relative sections has been exhausted. It works like this:



How CSS Paged Media works

The page model specifies how a document is formatted within a rectangular area, called the page box, defined with the `@page` rule. This rule allows for specification of various aspects of the page model, such as dimensions, orientation, margins, cropping, and print marks.

Layout

Physical books also have specific print artifacts that aid the reader's navigation: a table of contents, running headers and footers, page numbers, indexes, and so on.

These artifacts are not directly encoded in the content but rather automatically added by the rules defined in the CSS. For print it is also necessary to size and position the content to match the page and to break content in an appropriate way to support the meaning of the text.

These features are spread across several CSS modules:

- CSS Generated Content for Paged Media Module [https://www.w3.org/TR/css-gcpm-3/] defines special requirements for the display of printed document content: running headers and footers, footnotes, generated text for cross-references or a table of contents, PDF bookmarks, etc.
- CSS Fragmentation Module Level 3 [https://www.w3.org/TR/css-break-3/] defines how and where CSS boxes can be fragmented, including across page breaks.
- CSS page floats [https://www.w3.org/TR/css-page-floats-3/#terms] defines how an element is to be removed from the normal flow and placed in a different location on the page.

Print previews

Support for CSS Paged Media and other modules in Editoria is handled by Paged.js, allowing for a preview of the print styles in the browser before printing to PDF. Paged.js is developed and maintained by the Paged Media Initiative [http://www.pagedmedia.org/] , a gathering of people who produce books with web technologies.

Given that browser support does not yet exist for many Paged Media features, Paged.js implements polyfills (code that implements a feature on web browsers that do not support the feature) by parsing the CSS style sheets and fragmenting the content into pages using JavaScript. This also allows for extending supported features with other JavaScript libraries, such as custom hyphenation settings or a math typesetting library.

Thankfully, browser developers have already taken some interest in implementing parts of the Paged Media standards, and `@page` rules have partial support in Chromium and Chrome, giving the minimal needed support to generate PDFs from the Paged.js output.

Working within the browser gives access to extensive debugging tools for live style modifications and testing changes on the fly. While supporting the same W3C specifications, other tools use proprietary desktop rendering engines, forcing you to rerender the entire PDF with each change and making even the

simplest design edit time consuming. When Paged.js is used in browsers, it's possible to have access to the very latest support for CSS features.

27

Paged Media Support with Paged.js

Editoria includes multiple export options. The default method of handling page design and layout uses Paged.js.

The print export of the book's content uses CSS print specifications supported by Paged.js, making it possible to preview the print styles directly in the browser (see the previous chapter for more details). The browser print dialog is then used to create the PDF.

Paged.js is under active development, so the list of supported features is growing quickly. You can find further setup and extension documentation on Paged Media's GitLab [https://gitlab.pagedmedia.org/] , as well as file reports for any issues you may run into. However, the best way to reach out is to join the Paged.js community in the chat room on the Paged Media's Mattermost. [https://mattermost.pagedmedia.org/]

Page rules

The page rules must be set up in the `@print` media query.

```
@print{
  /* write the page rules here */
}
```

Size

The size of the pages in a book can be defined by either width and height (in inches or millimeters) or a paper size such as A5 or Letter. It must be the same for all the pages in the book and will be inferred only from the root `@page`.

```
@page {
  size: A5;
}

# or

@page {
  size: 140mm 200mm;
}
```

Margins

The margin command defines the top, bottom, left, and right areas around the page's content.

```
@page {
  margin: 1in 2in .5in 2in;
}
```

Names

Single pages or groups can be named, for instance as "cover" or "backmatter." Named pages can have their own, more specific, styles and margins, and even different styles from the main rule.

```
@page backmatter {
      margin: 20mm 30mm;
      background: yellow;
}
```

In HTML, these page groups are defined by adding the page name to a CSS selector.

```
section.backmatter {
    page: backmatter;
}
```

Page selectors

Blank pages

The blank selector styles pages that have no content, e.g., pages automatically added to make sure a new chapter begins on the desired left or right page.

```
@page :blank {
    @top-left { content: none; }
}
```

First page and nth page

There are selectors for styling the first page or a specific page, targeted by its number (named `n` in the specification).

```
@page :first {
    background: yellow;
}

@page :nth(5) {
    margin: 2in;
}
```

Left and right or recto and verso

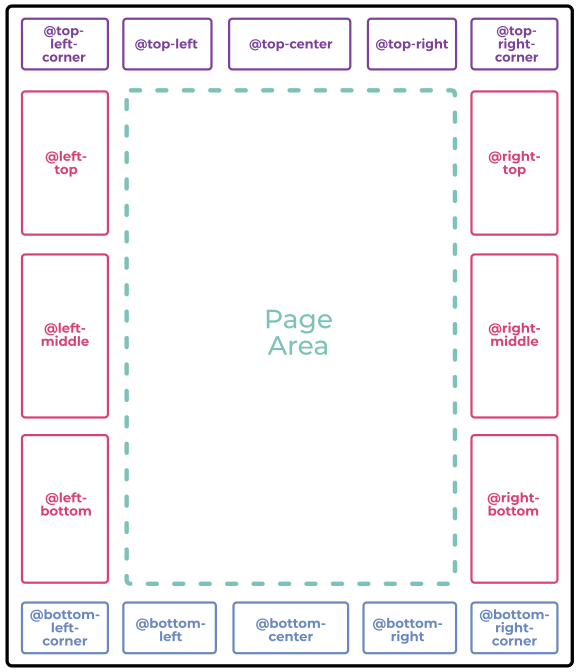
Typically, pages across a spread (a pair of pages) have symmetrical margins and are centered on the gutter. If, however, the inner margin needs to be larger or smaller, the selector to style left and right pages can make that change.

```
@page :left {
    margin-right: 2in;
}
```

```
@page :right {
    margin-left: 2in;
}
```

Margin boxes

The margins of a page are divided into sixteen named boxes, each with its own border, padding, and content area. They're set within the `@page` query. A box is named based on its position: for example, `@top-left`, `@bottom-right-corner`, or `@left-middle` (see all rules [https://www.w3.org/TR/css3-page/#margin-boxes]). By default, the size is determined by the page area. Margin boxes are typically used to display running headers, running footers, page numbers, and other content more likely to be found in a book than on a website. The content of the box is governed by CSS properties.



Automatically generated margin boxes

To select these margin boxes and add content to them, use the following example:

```
@page {
  @top-center {
      content: "Moby-Dick";
  }
}
```

Generated content

CSS counters

`css-counter` is a CSS property that lets you count elements within your content. For example, you might want to add a number before each figure caption. To do so, you would reset the counter for the `<body>` , increment it any time a caption appears in the content, and display that number in a `::before` pseudo-element.

```
body {
  counter-reset: figureNumber;
}

figcaption {
  counter-increment: figureNumber;
}

figcaption::before {
  content: counter(figureNumber)
}
```

Page-based counters

To define page numbers, paged.js uses a CSS counter that gets incremented for each new page.

To insert a page number on a page or retrieve the total number of pages in a document, the W3C proposes a specific counter named `page`. The counters declaration must be used within a `content` property in the margin-boxes declaration. The following example declares the page number in the bottom-left box:

```
@page {
 @bottom-left {
     content: counter(page);
 }
}
```

You can also add a bit of text before the page number:

```
@page {
 @bottom-left {
     content: "page " counter(page);
 }
}
```

To tally the total number of pages in your document, write this:

```
@page {
 @bottom-left {
     content: counter(pages);
 }
}
```

Repeated elements on different pages

Named string

Named strings are used to create running headers and footers: they copy text for reuse in margin boxes.

First, the text content of the element is cloned into a named string using `string-set` with a custom identifier (in the code below we call it "title," but you can name it whatever makes sense as a variable). In the following example, each time a new `<h1>` appears in the HTML, the content of the named string gets updated with the text of that `<h1>`.

```
h1 {  string-set: title content(text) }
```

Next, the `string()` function copies the value of a named string to the document, via the `content` property.

```
@page {
  @bottom-left {
      content: string(title)
  }
}
```

Running elements

Running elements are another way to create running headers and footers. Here the content, complete with style and structure, is copied from the text, assigned a custom identifier, and placed inside a margin box. This is useful for formatted text such as a word in italics.

The element's `position` is set:

```
.title {
  position: running(title);
}
```

Then it is placed into a margin box with the `element()` value via the `content` property:

```
@page {
  @top-center {
      content: element(title)
  }
}
```

Controlling text fragmentation with page breaks

Sometimes there is a need to define how content gets divided into pages based on markup. To do so, paged media specifications include `break-before`, `beak-inside`, and `break-after` properties.

`break-before` adds a page break before the element; `break-after` adds a page break after the element.

Here is the list of options:

- `break-before: page` pushes the element (and the following content) to the next available page
- `break-before: right` pushes the element to the next right page
- `break-before: left` pushes the element to the next left page
- `break-before: recto` pushes the element to the next recto page
- `break-before: verso` pushes the element to the next verso page
- `break-before: avoid` ensures that no page break appears between two specified elements

For example, this sequence will create a page break before each `h1` element:

```
h1 {
    break-before: page;
}
```

This code, in contrast, will push the `h1` to the next right page, creating a blank page if needed:

```
h1 {
  break-before: right;
}
```

This snippet will keep any HTML element that comes after an `h1` on the same page as the `h1`, moving them both to the next page if necessary.

```
h1 {
    break-after: avoid;
}
```

The last option is the `break-inside` property, which ensures that the element won't be separated across multiple pages. If you want to be sure that your block quotes will never be divided, write this:

```
blockquote {
  break-inside: avoid;
}
```

Cross-references

To build items such as an index or a table of contents, the export function has to find the pages on which the relevant elements appear inside the book. To do so, paged media specifications include a `target-counter` property.

For cross-references, links are used that target anchors in the book:

```
<p>see the <a href="#anchor-name">Title of the chapter</a></p>
```

Later in the book, the chapter title will appear with the anchor, set using an `ID` property.

```
<h1 id="anchor-name">title of the chapter</h1>
```

The `target-counter` property is used in `::before` and `::after` pseudo-elements and set into the `content` property. As a page counter, it can include some text:

```
a::after {
  content: ", page " target-counter(attr(href), page );
}
```

In the PDF, this code will be rendered as "see the Title of the chapter, page 12".

28

Extending Paged.js

There are several ways to extend the rendering of Paged.js. Selecting the best method will depend on how the code will be called and what it needs to access.

When creating a script or library that is specifically aimed at extending the functionality of paged.js, it is best to use hooks and a handler class.

Paged.js has various points in the parsing of content, transforming of CSS, rendering, and layout of HTML that you can hook into and make changes to before further code is run.

A handler is a JavaScript class that defines functions that are called when a hook in Paged.js is ready to defer to your code. All of the core modules for support of paged media specifications and generated content are implemented as handlers. To create your own handler, you extend this same handler class.

```
class MyHandler extends Paged.Handler {
  constructor(chunker, polisher, caller) {
    super(chunker, polisher, caller);
  }
}
```

The handler also exposes the underlying tools for fragmenting text (`Chunker`) and transforming CSS (`Polisher`)—see below.

Within this class, you can define methods for each of the hooks, and specify when they will be run in the code. A `return` that is asynchronous will delay the next code using `await`.

```
class MyHandler extends Paged.Handler {
  constructor(chunker, polisher, caller) {
    super(chunker, polisher, caller);
  }

  afterPageLayout(pageFragment, page, breakToken) {
    console.log(pageFragment, page, breakToken);
  }
}
```

Paged.js contains the following asynchronous hooks:

Chunker

- `beforeParsed(content` runs on content before it is parsed and given IDs
- `afterParsed(parsed)` runs after the content has been parsed but before rendering has started
- `beforePageLayout(page)` runs when a new page has been created
- `afterPageLayout(pageElement, page, breakToken)` runs after a single page has gone through layout, and allows adjusting the breakToken
- `afterRendered(pages)` runs after all pages have finished rendering

Polisher

- `beforeTreeParse(text, sheet)` runs on the text of the style sheet
- `onUrl(urlNode)` runs any time a CSS URL is parsed.
- `onAtPage(atPageNode)` runs any time a CSS @page is parsed
- `onRule(ruleNode) runs` any time a CSS rule is parsed
- `onDeclaration(declarationNode, ruleNode)` runs any time a CSS declaration is parsed
- `onContent(contentNode, declarationNode, ruleNode)` runs any time a CSS content declaration is parsed

Finally, the new handler needs to be registed in order to be used.

```
Paged.registerHandlers(MyHandler);
```

This can be registered anytime before the preview has started and will persist through any instances of `Paged.Previewer` that are created.

If a JavaScript library, such as MathJax, needs access to the content before it is paginated, you can delay pagination until that script has completed its work. This will give the library full access to the content of the book but has the disadvantage of needing to render the entire book before rendering each page, which can cause a significant delay.

Given that the polyfill will remove the page contents as soon as possible, adding a `window.PagedConfig` will allow you to pass a `Promise` that will delay until it is resolved.

```
let promise = new Promise((resolve, reject) {
  someLongTask(resolve);
});

window.PagedConfig = {
  before: () => {
    return promise;
  }
};
```

It is also possible to delay rendering of the polyfill until called by passing `auto: false`.

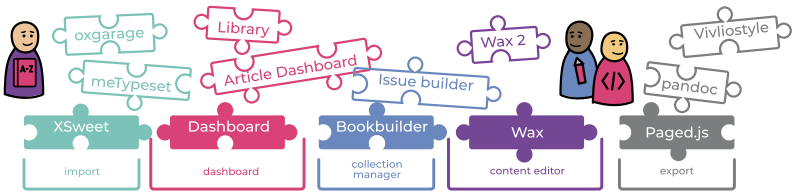
```
window.PagedConfig = {
   auto: false
};

window.PagedPolyfill.preview();
```

When the `Previewer` class is used directly, the `preview()` method can be called at any point that is appropriate.

The Future

29

What's Next for the Technology?



Editoria is excellent as-is, but is also infinitely customizable.

Editoria is already an elegant application: it's simple, with some impressive book production tools. Better yet, the architecture and technologies on which it is built point to an interesting future.

Wax

Until recently, it was not possible to build a sophisticated web-based editing and word processing tool such as Wax. However, web-based editing frameworks are coming of age, and Wax leverages them to deliver finely tuned interaction controls and granular management of the underlying content source necessary for word processing. It is, after all, word processing that publishers need—not just editing. The good news is that the Coko team also built Wax to be modular and configurable.

What's special about Wax is the ability to introduce incremental improvements on top of a sophisticated base framework, which radically reduces the development necessary for new features. The fact that Wax is web-based means that it is networked to many resources beyond its own word processing features. In the future, it may well be developed to allow design modifications, to build indexes, and to create interative elements such as adjustable diagrams.

Paged.js

Of course, there are other tools that can take HTML and render it to PDF, but none, open source or proprietary, takes the smart approach that Paged.js does. This tool follows the CSS standards that are accepted by the W3C but not yet implemented in the browser. The overarching strategy of Paged.js is to make itself redundant by proving the use case of making books in browsers to the people who make the browsers. This will take a while, but thankfully, in the meantime you can continue to use Paged.js to create layouts. Additionally, Paged.js, unlike any other tool, allows you to render the paged content in the browser and through a command-line batch processor, so you can preview the content live in the browser. Style the content, reload, render—as many times as you need to. As with Wax, the architecture of Paged.js is deliberately modular; it can be extended by small modules written in JavaScript to achieve typesetting and layout functions to meet your needs.

XSweet

Other tools used to convert .docx to HTML are written in obscure languages, are incomplete, use overly verbose and complex intermediary file conversion formats, are badly mantained, or are badly architected. The Coko team has looked around this field for many years, hoping for the right tool to mature. Unfortunately, it never appeared, so we had to build it ourselves, enlisting some of the top talent in the file conversion sector to do the job. As a result, XSweet is a sophisticated tool with high conversion fidelity. And yes, it too is modular and extensible.

PubSweet

Editoria is built on top of PubSweet, a cutting-edge, "headless" CMS that enables the construction of any kind of publishing workflow atop its very sturdy foundations. It is the underlying technology common to all the publishing platforms coming out of the Coko community, including Editoria and journal platforms under development by eLife and Hindawi. It is the workhorse of the growing open publishing infrastructure movement. PubSweet is important for Editoria because it allows developers to extend the highly modular architecure at will. Want to plug in a new set of roles and permissions? You can! Want to add a new interface for asset management or title management? Go ahead!

The Future

This technology is the best available today, and it is all open source. You own it. You can do whatever you like with it. You are handed a powerful, extensible framework for publishing any kind of content you like. Editoria handles books but can take you beyond books. It is designed not only to optimize your present workflow but to give you the best foundations for future innovations.

There are many features that we are either working on or planning to work on. Versioning of books is one example—it would be great if you could version chapters and books, navigate easily between editions, and improve or add content while retaining an understanding of the history of how the book has evolved. You could also mark versions as final for an edition or even archive them without removing them from the system. Furthermore, users could manage stored output of a PDF or EPUB to ensure that new editions are based on the precise content that was exported from the system years ago.

Various versions of chat or discussion-list integration have been considered in detail. This could help teams discuss and resolve issues quickly. Connected to this are many possible features that could extend the functionality of comments and annotations to serve a similar purpose.

A newer version of Wax, our word processor, will introduce features that are currently not there, such as creating and editing tables and nested lists. Concurrent editing of chapters has already been named as a wish. We want to include image editors as well.

We also foresee integration with external services such as Title Management and other metadata managers and publishing channels including sales channels like Amazon and open source platforms such as Manifold and Fulcrum. We also want to support a wider variety of file outputs (e.g., .mobi, .odt). Math integration and authoring is on the near horizon too. Asset management and art workflow interfaces are high on our list. What about a translation interface? The sky's the limit!

An interactive administrator panel could let you customize your application's configuration more easily and, more importantly, without the need for a developer in many cases. A profile manager could also provide a useful tool for users to manage their personal data.

Get involved

If any of these ideas piques your interest, get involved in the Editoria community today! Imagine if you and ten other imaginative users immediately started working together on improvements—the system could grow by leaps and bounds very quickly!

Glossary

Bookbuilder — The screen in Editoria where a book's components are assembled and managed.

CMS (Content Management System) — A software application or set of related programs that are used to create and manage digital content.

Coko — Shorthand for Collaborative Knowledge Foundation, the technology and community facilitation providers for Editoria.

Community — Anyone interested in actively participating to move Editoria forward.

CSS (Cascading Style Sheets) — Code that applies formatting to text and elements affecting how those display on a screen or paper.

Dashboard — The screen in Editoria where a user chooses which book to work on.

Editoria — An open source, community-owned, browser-based book production tool. For an extended definition, see the beginning of this book.

Export — Moving from HTML to fixed, publication-ready outputs (examples include EPUB and PDF).

Flat workflow — A workflow with few permission restrictions, enabling rapid collaboration, rather than rigid, granular permission controls (example: Book Sprints).

G&T (Gin & Tonic) — Writers' fuel.

Import — The ingest of files from a desktop or external platform into Editoria. Editoria's process ingests Microsoft Word XML (.docx) files and converts them to HTML for formatting, editing, and further refinement in the Wax editor.

Infrastructure — This generally refers to the digital publishing infrastructure used by publishers to execute publishing functions. It's the foundation of

publishing activity, often described as a pipeline. The Gutenberg press is one of the earliest examples of publishing infrastructure; Editoria is one of the latest.

Library publishing — The set of activities led by college and university libraries to support the creation, dissemination, and curation of scholarly, creative, and educational works. It is distinguished from other publishing fields by a preference for open access dissemination and a willingness to embrace informal and experimental forms of scholarly communication and to challenge the status quo. Learn more on the website of the Library Publishing Coalition.

Linear workflow — A workflow in which permissions are granular and roles are specifically defined (e.g. production editor, copy editor) in relation to a stage in the book production process.

Paged Media — A W3C standard for paginating content in browsers. It is also the name of the initiative working on Paged.js.

Paged Media Initiative — A community sharing experiments and knowledge about printing documents and books using web technology. Learn more on the Paged Media Initiative blog.

Paged.js — A javascript toolkit, developed by the Paged Media Initiative, that is an integrated export option from Editoria. It works as the polyfill for the Paged Media W3C specifications for printing from the browser.

Polyfill — Code that implements a feature on web browsers that do not support the feature.

Presses — Shorthand for university presses or association presses, which tend to differ substantially from library publishers.

Proprietary software — Software with closed code that tends to be typified by rigid, hard-coded workflows and business models that lock publishers in for long periods of time and allow them no meaningful access to their data. Proprietary software can be especially problematic for academic publishers when it is owned by bad actors in the scholarly communication sphere.

Publishers — Anyone doing publishing activities or, alternatively, employed by someone who does.

PubSweet — The open source publishing systems platform upon which Editoria is built. PubSweet is itself built by the Coko community.

Roles — Describes who a person is with respect to a single book (e.g. author, editor) and what they are and are not allowed to do in Editoria.

Scholarly communication — The system through which research articles, monographs, and other scholarly writings are created, evaluated for quality, disseminated to the scholarly community, and preserved for future use. The system includes both formal means of communication, such as publication in peer-reviewed journals, and informal channels, such as electronic listservs.

> *Association of College and Research Libraries, "Principles and Strategies for the Reform of Scholarly Communication 1 [http://www.ala.org/acrl/publications/whitepapers/principlesstrategies]" (2003)*

Wax — A web-based word processor that is configurable and capable of supporting different themes. Wax is a Coko tool and is the editor included with Editoria. Thanks to Editoria's modular design, it can be replaced by another tool if needed.

Workflow — The chain of events undertaken to bring content from initial creation to publication.

WYSIWIG (What You See Is What You Get) — An editing interface where text and other elements that are created or edited appear within the editor the same as they will in final format, whether print or digital.

XSweet — A customizable set of XSLT (extensible stylesheet transforms) that transforms Microsoft Word XML (.docx) content into HTML and beyond. Editoria uses XSweet to import .docx file content into the Wax editor.

Colophon

This book was collaboratively written with Editoria and facilitated by Book Sprints [https://www.booksprints.net/] . The book was rendered to print-ready PDF using Paged.js [https://www.pagedmedia.org/] . The body text of the book is set in Spectral, designed by Production Type for Google and available on Google fonts [https://fonts.google.com/specimen/Spectral] while the headings are composed in Fira Sans and the coding is set in Fira Code, both designed by Erik Spiekermann, and available from Mozilla [https://mozilla.github.io/Fira/] .

The Book Sprint was facilitated by Barbara Rühling.

The book was designed by Agathe Baëz [https://www.agathe-baez.fr/] .

Illustrations are made by Henrik Van Leuween [http://www.henrikvanleeuwen.nl/] .

The book was printed by Imagink [http://imaginkshop.com/] , in San Francisco (California, USA), during the month of October 2018.