psychic-octo-robot Documentation

Release .1

Ian Hallam, John Kulczak, Sydney Satchwill, and Devon Timaeus

May 19, 2015

Contents

Contents:

CHAPTER 1

Introduction

This project is a collection of modules that brings the power of git to structured documents by students Ian Hallam, Devon Timaeus, and Sydney Satchwill at Rose-Hulman Institute of Technology under the leadership of Associate Professor Sriram Mohan in addition to Wes Winham and Kyle Gibson of PolicyStat.

The goal of gitlit is to bring the power for Git, and source control to the masses for use on documents like .docx, .html, or any other document format that is structured.

CHAPTER 2

Quick Start Guide

2.1 Installing the package

Use an npm install to download and globally install the package:

```
npm install -g gitlit
```

2.2 Create a repository

Create an initial repository off of an HTML page using:

```
gitlit init <file> <outputPath> <repoName>
```

2.3 Commit a modified file

Commit a modified version of the file to the initiallized repository using:

```
gitlit commit <file> <pathToRepository> <commitMessage>
```

2.4 Diff the two previous commits

To get the difference between the two previous commit for display or to merge, run the diff command in gitlit:

```
gitlit diff <repoLocation> <outputLocation>
```

Now open the file that was saved by the diff step in a browser to use the visual diffing tool. Save the output in somewhere easily accessable for the merge step.

2.5 Merge the output of the diff

Run the gitlit merge command to take the changes selected in the diff display and output a commit ready HTML file:

```
gitlit merge <mergeFile> <outputLocation>
```

2.6 Restore an HTML file from a repo

To get an HTML file from the repo run the gilit write command on the directory with the desired output name for the file:

```
gitlit write <directory> <outputFile>
```

CHAPTER 3

Problem Statement

3.1 The Gospel of Git & and Source Control

Software Developers the world over make use of Git, SVN, and other source control systems to keep track of their source code, merge in branches, and generally keep their projects well maintained.

Most source control systems offer many features that make project management a breeze, such as branching,

reverting, committing code changes, and showing differences between revisions of the code. There are some idiosyncrasies between each different system, but the basic functionality mentioned just now is present in all of them. Having a system like this has almost become second nature to project management and development in the world of software development.

3.2 What Everyone else does

For the most part, source control systems are limited to strictly software development fields; it is rare for similar tools to be used by other groups. That being said, the largest innovation that is similar to source control systems and is widely used today are tools like Google Drive, systems where users can edit documents at the same time, and keep one copy synced between the entire group.

Google Drive, and other similar applications, work very similarly to source control at a base level, by keeping one document synced in a group. Many even have the idea of revisions that are present in source control systems, however, most lack the full depth of power that traditional source control systems have. This leaves most casual users with almost no features that are present in a source control system, only the basic functionally of simultaneous editing, viewing, and syncing.

3.3 Who Cares?

Applications like Google Drive could benefit greatly from having more functionality that is present in tools like Git, since it would buy users powers like:

- Showing differences between revisions of a document
- Reverting a document to a different revision in case something goes wrong
- Leaving a “paper” trail of who changed what, when
- Branching to allow for changes that don’t block other users

For the most part, people _could_ just use Git for their documents, even if they aren't code, but for many document types, Git is not suited to managing differences very well, as it just treats all documents as text files (HTML, Microsoft Formats, CSV files, etc.). Additionally, Git is a tool designed by software developers, for software developers, and exists almost purely in a command-line form, or at least, to access most functionality, a command-line must be used. This makes the accessibility of Git to most organizations and people virtually nonexistent, as they would need to learn a tool that has a high burden of knowledge, and get used to using command-line tools, which is foreign to the average user.

3.4 Place your trust in the mighty Gitlit

Our tool is meant to solve some of the problems that crop up with casual users trying to use Git on non-source-code documents.

Specifically, our goal is to make an application that will operate on structured documents, such as HTML and XML, offering all of the same operations and functionality that Git offers, but encapsulated in a way that any user familiar with programs as complex as Microsoft Word would be able to understand and use with proficiency.

Our application will not only create and manage Git repositories of structured documents (one document per repository) but will also provide a UI to allow for users to easily view diffs of revisions, manage merges, and view and manage change requests.

To accommodate the need for client-side Web UI scripting, interfacing with Git repositories, and server side repo-management, the application will be written in Javascript, making use of Node.js and js-git.

CHAPTER 4

Feature List

4.1 Initialize/Create a repository with a structured document

4.1.1 Description

Users should be able to create a Git repository with gitlit by giving it an HTML document, and telling it to create the repository. The user should also be able to configure which remote (i.e. GitHub, vs. private Git repo) to create, as well as whether it should be public or private.

4.1.2 Risks

This could run into issues with private repos, as users may have too many repos to have one document per repo, especially if they are going to be made private.

Risk Level: Medium (Large problem, not likely to happen as we are going to address it beforehand)

4.1.3 Priority

1st

This feature is necessary for any of the other features to work properly, since we need to set up the repo in such a way that all the other operations will be supported.

4.2 Commit a new revision provided a new document

4.2.1 Description

A user should be able to submit a new version of the document (with changes made as they desire) as a commit, and gitlit will convert that into the appropriate commit changes in Git.

4.2.2 Risks

Having this work when branching off a section could be interesting, as it would require that certain sections be the only things that are changed. Though this should be handled neatly once the organization of the repo directory is decided.

Risk Level: Medium (Should be straightforward how to do it, but possible that there are issues that come up later)

4.2.3 Priority

2nd

This feature is a linchpin feature, without this, the system doesn't really matter.

4.3 Commit a new revision provided a changed subsection?

4.3.1 Description

This may be automatically done in the previous feature, more requirements gathering needs to be done before we can state more about this feature.

4.3.2 Risks

This feature may be marvellously difficult to do, as there are issues with identifying the sections that were changed, as well as determining what to do if there are changes in section that wasn't branched off of.

4.3.3 Priority

Last This feature would be nice to have, but will be rather difficult to pull off; for now, it would be best to focus on the other functionality.

4.4 View differences between revisions of a document

4.4.1 Description

Users should see a well-defined and obvious diff between the old revision and the new revision, ideally rendering the HTML in a manner that lets them see the webpage before and after the change. The diff should also be per-section, not per-line

4.4.2 Risks

There could be some difficulty with *how* we note the changes, as color changes could be unnoticeable depending on what the CSS of the page is meant to be.

4.4.3 Priority

3rd If users can't see the differences between revisions of the document, then the tool doesn't really help with managing documents like Git very well, in fact, it might even be worse.

4.5 Accept/Reject capability for merge conflicts

4.5.1 Description

Users should be able to view any conflicts that come up in an attempted merge, select which version they would like to be used, and then approve the merge, all via a web-based UI.

4.5.2 Risks

Even current merge conflict tools are confusing, so there could be some difficulty in making the UI in such a way that is clear to all users and yet still has the power that is needed to solve merge conflicts.

4.5.3 Priority

6th

This feature matters, but users could always just prevent merge conflicts from ever happening by checking things beforehand.

4.6 Line-by-line Pull Requests

4.6.1 Description

Users should be able to send requests to other contributors of a repository to review and approve their changes made to a document. Reviewers should be able to view the changes, approve, deny, and comment on changes that are in that diff.

4.6.2 Risks

Finding a solid way to turn a GitHub pull request into a form that is easily view-able via gitlit's diff viewer could be an interesting challenge. That being said, there are plenty of options, so not much risk overall.

4.6.3 Priority

5th

Pull requests and reviews are critical to any form of document creation, whether they are source code files or something else. This makes this feature critical right behind basic functionality.

4.7 Commit new revisions of a document to a branch

4.7.1 Description

A major function of Git is the ability to branch off of a repo and make changes that don't block the pipeline for other people. As such, we want to allow users to branch off of individual sections, as well as the document as a whole, and then allow them to make changes to make individual little changes to these branches.

4.7.2 Risks

Deciding what the right action regarding editing sections that are in a branch or not in a branch is a bit complex, so there is some risk that whatever choice we make might not be intuitive to a large number of people that we don't have access to when user-testing.

4.7.3 Priority

4th Branching is super critical to git and gitlit; we have to have it right after basic committing and repository setup.

CHAPTER 5

Using Gitlit

5.1 Getting started

There are two possible use cases for gitlit:

1. Using the tool for its purpose of structured document management.
2. Using the tool to ensure it is working when developing for it.

In the former case, currently, the means of installing it would be through NPM. If NPM is installed on they system, simply run:

```
npm install -g gitlit
```

To run any of the commands discussed here, simply use:

```
gitlit <command info>
```

If contributing to the project, then inside the GitHub project folder, at the root (where the package.json file is), simply run:

```
node ./cli/gitlit.js <command info>
```

Alternatively: ./cli/gitlit.js <command info>

Note: For all of the sections following this, the information given will be what is meant to go into <command info>. So, if you were using the tool for it's purpose, and you wanted to initialize a repository, the command might be given as:

```
init source-file . outputRepo
```

but note that this is prefixed by the necessary command to run the tool based on why you are trying to run it.

5.2 Creating a repository

The command for creating a repository given an HTML document is:

```
init <file> <location for repo to be made> <repo name>
```

So an example command might look like:

```
gitlit init test.html /usr/root/docs first-repo
```

Take note that the 2nd argument needs to be a location that already exists, as what actually happens is a directory is made at that location with the name <repo name>

So, if the above command was run, the directory */usr/root/docs/first-repo* would then be made.

If it existed before, an error will be thrown stating as such.

Note: If there are missing tags in the HTML document, such as a missing opening tag, the parser used by gitlit will take it's best guess as to what was intended. Since there are most likely many possibilities, the possibility chosen may not be what the user meant. If you want to be certain this doesn't happen, ensure that the HTML is valid, preferably through The W3C Validator.

5.3 Generating HTML from a repository

To generate a file from a repository, simply run:

```
write <path to repository directory> <path of output file>
```

An example command might look like:

```
gitlit write /usr/root/docs/first-repo ./test.html
```

Which would generate an HTML file called "test.html" in the current directory, with its source being the repository "first-repo" at */user/root/docs*.

If the output file existed before, its contents will be overwritten so take caution.

Note: The contents of the HTML file will be pretty-printed as reasonably as possible. Due to the complex nature of HTML and the fact that gitlit wants to make *only* meaningful notes about differences between different versions of a document, some quantity of formatting (i.e. whitespace) is lost, and as such, the formatting is not kept *exactly* the same. That being said, the formatting is still decent, and is quite readable.

Note: After generating the file, you likely noticed the **attributes that got added to many of the tags called "por-id"**. These attributes are tracking ID's used to aid in recognition of sections even if the subsections or text is changed. As such, **don't alter or remove the por-id's unless the document is completely finished being developed**, as this will cause gitlit to give very unhelpful diffs when comparing different versions of a document.

5.4 Making a commit

Once a repository is created, a new revision can be tracked in the repository through making a new commit. Just like git, and differences will be stored into a new commit and the repository will be at this new revision, with the history stored in the commit history.

Since gitlit works on a document as a whole, commits are made by providing a new document. This removes the need for users to edit the files split by the tool, and makes it easier to view the document as a whole will editing.

To make a commit, just run:

```
commit <file for new revision> <path to repo directory> <commit message>
```

An example command would look like:

```
gitlit commit ./test.html /usr/root/docs/first-repo "Changed page title"
```

After this command is made, the Git repository that is the basis for that gitlit repository will have a new commit, and the whole directory structure will be in a state the reflects the new revision.

If a write command is run on this repository now, the outputted document will look nearly identical to the input document given in the above command. (Nearly identical because of some likely formatting of the document, and perhaps por-id's added for new sections).

CHAPTER 6

Setting up the Development Environment

6.1 Initial Setup

This assumes that the repository location is already cloned locally, and there is some from of vm software available.

Starting off, download vagrant http://www.vagrantup.com/downloads.html as appropriate to the local system, and install it. Launch a terminal window [if on windows use the git bash], and navigate to the repository location. The correct directory should have a subfolder within it called '.vagrant'. While in that directory call the command 'vagrant up'. This will download the necessary image and set it up with the environment used for testing. Use 'vagrant ssh' after the command has successfully finished to enter the vm and then use 'cd /vagrant/' to get to the directory it was launched from. The command 'exit' will logout from the vagrant vm. When use or testing is complete, run 'vagrant halt' to shut down the vm gracefully.

6.2 After Initial Setup

Once the development has been setup initially, the vm can be re-entered by simply using the 'vagrant ssh' command again, assuming it has not been ended with halt. If it has been ended, calling 'vagrant up' will check for updates and then launch the vm for use.

6.3 Errors in File Location After Initial Setup

There is an error that occasionally arises with vagrant where it points to an incorrect location for the vm. The generic fix for this is to delete or remove the vm from the vm software being used, and use 'vagrant up' to restablish the vm.

CHAPTER 7

HTML to Repository Serialization

In order to store meaningful changes in git, PSO serializes HTML to a folder/file structure. This makes change and move detection behave similar to how git expects and reduces the chances of merge conflicts.

Note: Almost all of these would make great test cases.

7.1 Design Issues

7.1.1 How do we represent text nodes?

For representing text nodes, just having a text file for the information would adequately store what was necessary, and then filenames can be generated arbitrarily, likely using any `id` attributes that already exist.

So, if we had:

```
<p GUID=1>hello <span GUID=2>stuff</span> goodbye </p>
```

This would become something organized like this:

```
* GUID1
    * metadata.json
    * 1.txt
    * GUID2 (this is a dir)
        * 1.txt
    * 2.txt
```

There is the issue of:

- **What if "goodbye" was moved to** be before the span?

But there isn't much that can be done here; it is likely that the parser used for parsing the HTML will result in one file for each of "goodbye" and "hello" so long as the span separates them.

Without the span, the text would be treated as one file, thus, if it was moved, then it would result in a removed file and another file being modified.

To solve this, there would need to be a decision of where to stop in parsing the HTML, so that a file doesn't get removed, but this would require quite a bit of extra information.

This isn't insurmountable, as the old version could just be read, read the file, and keep track of where it "ends" and then just parse to that, or the first child node, and split the files accordingly.

That being said, this could result in some file directories that look very strange, but would likely work better in diffs.

7.1.2 How do we represent ordering of text nodes mixed with other nodes?

This could just be done with naming in lexicographic -based construction, and the table in the table -based construction. Basically, just have the names of directories and the files that hold content be what we use to determine what is what. That is, if a table says:

- GUID1
- GUID2
- GUID3

Regardless if any of those are directories, just recursively build that directory into a node, and then paste it in during construction.

For name based, same idea, but with names.

7.1.3 How do we store node type?

This could be done in the metadata file, since there will be one one either way.

7.1.4 How do we store node attributes (`src`, `href`, etc)?

Same as node type, metadata file would work well for this.

7.2 Simple Paragraph Addition

7.2.1 Initial Content

```
<p>p1</p>
<p>p2</p>
```

After GUID-ization

The first pass adds GUIDs to any nodes that don't have them. Since this is initial, they all get new GUIDs. Further examples will ignore GUIDs. Assume they're there.

```
<p data-por-guid="GUID1">p1</p>
<p data-por-guid="GUID2">p2</p>
```

Lexicographical Representation

Table Representation

7.2.2 Edit 1: Additional 2nd Paragraph

```
<p>p1</p>
<p>p3</p>
<p>p2</p>
```

Lexicographical Representation

Conflicts: No

Table Representation

Conflicts: No

7.2.3 Edit 2: Additional Last Paragraph

From the initial content, we'll add a 4th paragraph.

```
<p>p1</p>
<p>p2</p>
<p>p4</p>
```

Lexicographical Representation

Conflicts: No

Table Representation

Conflicts: No

7.2.4 Merged Edit 1 and Edit 2

Those two edits should merge in without conflicts.

```
<p>p1</p>
<p>p3</p>
<p>p2</p>
<p>p4</p>
```

Lexicographical Representation

Conflicts: No

Table Representation

Conflicts: Yes. In `metadata.json`.

7.3 Nested Nodes

7.3.1 Initial Content

```
<p>p1<span>s1</span>stillp1</p>
<p>p2</p>
```

7.4 Node Moves

7.4.1 Initial Content

```
<p>p1</p>
<p>p2</p>
<p>p3</p>
<p>p4</p>
```

Lexicographical Representation

Table Representation

7.4.2 Edit 1: Last to First

```
<p>p4</p>
<p>p1</p>
<p>p2</p>
<p>p3</p>
```

Lexicographical Representation

Conflicts: No

Table Representation

Conflicts: No

7.4.3 Edit 2: Last to First with content change

```
<p>p4new</p>
<p>p1</p>
<p>p2</p>
<p>p3</p>
```

Lexicographical Representation

Conflicts: No

Table Representation

Conflicts: No

7.4.4 Merged Edit 1 and Edit 2

Those two edits should merge in without conflicts.

```
<p>p4new</p>
<p>p1</p>
<p>p2</p>
<p>p3</p>
```

Lexicographical Representation

Conflicts: No. Not if content and move are separate commits.

Table Representation

Conflicts: No

7.5 Design Decision

Gitlit keeps track of the order that the nodes should be constructed via the table-method. This has the simplest base case, and stays simple even when complexity is added through commits.

For any in-depth discussion of why, see this git issue

CHAPTER 8

Text Node Name-Tracking

When gitlit is tracking a document, there will be files that are generated that are purely made of text, and contain the actual written content of the document.

However, there is an issue that can arise with regards of how to keep track of the differences between each section. Given that we have chosen to keep track of document order via a table, we need a way to consistently generate names so that if a section is left in the document (changed or not), we recognize it, and name the file accordingly so that doing file diffs are done appropriately.

For HTML elements, this is easy, since we can just add some arbitrary attribute to each tag that only we will look at. For text nodes, this becomes more difficult. Given something like this:

```
<span>
    First Text
    <div>
        div Text
    </div>
    Second Text
</span>
```

The obvious thing to do for tags is to add an attribute that will keep track of *what* the element is for easier diffing.

```
<span por-id="40E36DB5C0AD13957351">
    First Text
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    Second Text
</span>
```

Now, we are well aware of when the sections get moved or are changed. However, if we generate a filename for the text nodes, we would have no way to map a new commit of this document to the files without doing a text comparison (which we would like to avoid). Thus, we either need a consistent way to name the files (e.g. first file is named 1.txt, second file 2.txt, etc.) or a way to store the filename/id when we generate the document for users to re-commit to us.

If we did a consistent naming format, that would somewhat defeat the purpose of a table, additionally, if there was ever an insert of a new element that had a text node near it, there could be problems with miss-identification with old sections.

This leaves us with trying to store the id of the text node somehow so that in future commits we can identify the name the new files should be.

There are 2 main ways we are considering doing this:

- XML-style Text & Tail

- Custom HTML tags with id information.

The rest of this page will be exploring how each would be done, working through example cases that would need to be handled and deciding on which method gives the best coverage.

Note: Almost all of these would make great test cases.

CHAPTER 9

Method explanation

9.1 Custom HTML Tags with id Info

The first method we thought of to solve the id issue was to create our own HTML tag and use that with our metadata file to keep track of the id of each text node.

For example, the input:

```
<span por-id="40E36DB5C0AD13957351">
    First Text
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    Second Text
</span>
```

Would become something like:

```
<span por-id="40E36DB5C0AD13957351">
    <por por-id="39E1D62AADCF588B873C ">First Text</por>
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
    </div>
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

This works because HTML just ignores any tags that it doesn't recognize, so, this should be fine to do.

9.1.1 Pros

- Very Easy to do
  - Both HTML generation and Repo generation becomes straightforward
- High Consistency
  - Very clear mapping
  - If the id existed before, just compare against the old.
- Easy to understand
- Only needs to be done once
  - Still needs to be read each time

9.1.2 Cons

- Standards are weird
  - HTML might misbehave if we inject our own tags
- Fragile
  - If a user deletes the tag, it will look like an insert.
- Gives the User responsibility
  - The user has to keep track of the new tags
  - More work for the user

9.2 XML-style Text & Tail

The next method of storing the ids for text nodes took some inspiration from XML. In XML, each tag has a `text` attribute and a `tail` attribute. The `text` attribute just has the first text of the node that isn't a child's tail. The `tail` attribute has any text that falls after the current node and before the next tag.

For example, the input:

```
<span por-id="40E36DB5C0AD13957351">
    First Text
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    Second Text
</span>
```

Would have a mapping of something like this:

```
{
    por-id: "40E36DB5C0AD13957351",
    text: "First Text",
    children: {
                por-id: "76F7BEE8A2001AC7144D",
                text: "div Text",
                tail: "Second Text"
               },
    tail: ""
}
```

Alternatively, instead of keeping track of the actual text, it could just note the por-id of the object:

```
{
    por-id: "40E36DB5C0AD13957351",
    text: "39E1D62AADCF588B873C",
    children: {
                por-id: "76F7BEE8A2001AC7144D",
                text: "56ADFDCAACEB1FDBCEA1",
                tail: "3CED92A56695A78653ED"
               },
    tail: ""
}
```

This would provide a good mapping of text nodes to ids while avoiding placing any extra tags around text.

9.2.1 Pros

- Easy to do
- High Consistency
  - Very clear mapping
  - If the id existed before, just compare against the old.
- Only new attributes added to nodes
- Consistent with XML, another markup language
- **More robust**
  - No tags to move around

9.2.2 Cons

- **Still fairly fragile**
  - If the user moves the text and not the id in the text or tail attribute, then no point.

CHAPTER 10

Do we even need tags to track text nodes?

The reason why we need tags to keep track of which text nodes are which is so that we can know which sections are which, and know if they have been moved without having to do a text comparison on the contents of the text node.

This is important because if we were to do a text comparison, we would have to ask, “How accurate/sensitive is good enough?” Because this is complex, if we can avoid text comparison altogether that would be preferable.

That being said, if we don’t care about differentiating between additions, deletions, and moves, then we could just ignore tags and do text-comparison. The reason this is alright is because at some basic level text comparison needs to happen for a diff, but if we don’t care about tracking moves, then we don’t need to keep track of moves *and* changes, thus, if a section was both moved *and* changed, we could just say it was an addition and be done with it.

CHAPTER 11

Case 1: Editing A Text Node

Consider the case of editing a pre-existing text node. The document before the edit might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    First Text
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    Second Text
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    First Text that has been altered
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    Second Text
</span>
```

Note that the first text section has been edited, but not moved or had any other changes applied to it.

In this case, the functionality we would like is just saying that the first section was changed, with no other perceived changes or moves.

11.1 Custom Tags

Assuming the repository already existed, if we made the change with Custom tags, then there are 2 cases.

1. The text nodes already had custom tags around them
2. The text nodes didn't have any custom tags.

In the second case, the commit would just put custom tags around anything that didn't, in which case, they would be seen as new files if there was a diff (likely).

So, for this case, we really only care about if there were tags already.

```
<span por-id="40E36DB5C0AD13957351">
    <por por-id="39E1D62AADCF588B873C ">First Text</por>
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
    </div>
```

```
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    <por por-id="39E1D62AADCF588B873C ">First Text that has been altered</por>
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
    </div>
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

In this situation, since the text was edited in the same text node (which is defined by the custom tags), the new text would just fall into a file that already exits: 39E1D62AADCF588B873C.txt (or something similar). Because of this, Git would perceive this purely as a change in the file/section, which is what we wanted. Even if we did diff logic ourselves, it would be easy to see that the text was edited, so it is just a text change.

11.2 XML text tail

Assuming the repository already existed, if we made the change with XML text-tail, then the relationships of text & tail would already be stored in the HTML's attributes, otherwise, there would be issues similar to custom tags: the change would be perceived as completely new text.

```
<span por-id="40E36DB5C0AD13957351" text="39E1D62AADCF588B873C">
    First Text
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1" tail="3CED92A56695A78653ED">
        div Text
    </div>
    Second Text
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351" text="39E1D62AADCF588B873C">
    First Text that has been altered
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1" tail="3CED92A56695A78653ED">
        div Text
    </div>
    Second Text
</span>
```

In this case, literally nothing **but** the text got changed. This is as ideal as we can get, as the user then doesn't need to navigate around more tags. Granted, there are attributes to deal with, but this is likely to be seen as less of an issue for users.

CHAPTER 12

Case 2: Deleting A Text Node

Consider the case of deleting a pre-existing text node. The document before the edit might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    First Text
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    Second Text
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    Second Text
</span>
```

Note that the first text section has been deleted, but not moved or had any other changes applied to it.

In this case, the functionality we would like is just saying that the first section was deleted, with no other perceived changes or moves.

12.1 Custom Tags

For this case, we really only care about if there already tags.

```
<span por-id="40E36DB5C0AD13957351">
    <por por-id="39E1D62AADCF588B873C ">First Text</por>
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
    </div>
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    <por por-id="39E1D62AADCF588B873C "></por>
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
```

```
    </div>
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

Or, it might look like this, depending on what the user did

```
<span por-id="40E36DB5C0AD13957351">
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
    </div>
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

In either situation, we could easily identify that there is now no text for the object that was the first text node. This would be identified by it either the node **not being there** or the node containing no text. Both are reasonable to have happen, but the fact that there could be either case means there is a bit more decision making to be made that for editing.

12.2 XML text-tail

Document before change

```
<span por-id="40E36DB5C0AD13957351" text="39E1D62AADCF588B873C">
    First Text
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1" tail="3CED92A56695A78653ED">
        div Text
    </div>
    Second Text
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351" text="39E1D62AADCF588B873C">
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1" tail="3CED92A56695A78653ED">
        div Text
    </div>
    Second Text
</span>
```

The user could also get rid of the text attribute as well

```
<span por-id="40E36DB5C0AD13957351">
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1" tail="3CED92A56695A78653ED">
        div Text
    </div>
    Second Text
</span>
```

Similar to custom tags, we need to have checking to see if the text actually exists if there is a text attribute. If not, then the section was deleted, if there isn't even a text attribute, then if there isn't any text, it was deleted.

In the case of tails, the same idea would happen, which creates 4 cases really:

1. Text
   (a) Tag there but no text
   (b) No text & no tag

2. Tail
   (a) Tag there but no text
   (b) No text & no tag

Note: If there was no tag and then text, in both systems, the text would be recognized as an insertion.

Case 3: Inserting A Text Node

Consider the case of inserting a new text node. The document before the edit might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    Second Text
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    First Text
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    Second Text
</span>
```

Note that a text section was added at the beginning, but that no other changes were made.

In this case, the functionality we would like is just saying that the first section was deleted, with no other perceived changes or moves.

Note: If there was text added to a pre-existing section, it would not be recognized as a separate text node. It would just be an edit.

13.1 Custom Tags

Before the edit

```
<span por-id="40E36DB5C0AD13957351">
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
    </div>
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    <por por-id="39E1D62AADCF588B873C ">First Text</por>
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
    </div>
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

Custom tags here behave exactly as we would want and expect. Since there is completely new text where there wasn't a tag, then a tag (and therefore a file) will be made, so it's completely new.

One additionally "cool" thing that *could* be done, is using custom tags for change tracking granularity. As an example, if the insertion was instead after the custom tag with "Second Text", it would be recognized as a new text node, despite it normally not being so. This *could* be useful or something users want, since in further applications (for example, docx files) insertions of new paragraphs might be nicely tracked by allowing something like this.

13.2 XML text-tail

Document before change

```
<span por-id="40E36DB5C0AD13957351">
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1" tail="3CED92A56695A78653ED">
        div Text
    </div>
    Second Text
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351" text="39E1D62AADCF588B873C">
    First Text
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1" tail="3CED92A56695A78653ED">
        div Text
    </div>
    Second Text
</span>
```

The user could also add the text after the div

```
<span por-id="40E36DB5C0AD13957351">
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1" tail="3CED92A56695A78653ED">
        div Text
    </div>
    Second Text
    First Text
</span>
```

In the first case, there would just now be a `text` attribute where there wasn't before, so it's easy to see the insertion. This also applies to if it ended up creating a `tail` attribute.

Unlike with custom tags, there would not be a way to keep track of multiple text nodes in a row. The second case would just be viewed as an edit of that text node.

CHAPTER 14

Case 4: Moving A Text Node

Consider the case of moving a pre-existing text node. The document before the edit might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    First Text
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    <div por-id="placeholder">
    </div>
    Second Text
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    First Text
    <div por-id="placeholder">
    </div>
    Second Text
</span>
```

Note that a text section was moved, without any edits to the content of the text node being made.

In this case, the functionality we would like is just saying that the first section was moved to be after the first div.

Note: Again, if this was moved to be part of another pre-existing text node, it would just be noted as a change to the destination node and a deletion of the old node.

14.1 Custom Tags

Before the edit

```
<span por-id="40E36DB5C0AD13957351">
    <por por-id="39E1D62AADCF588B873C ">First Text</por>
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
    </div>
    <div por-id="placeholder">
```

```
    </div>
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
    </div>
    <por por-id="39E1D62AADCF588B873C ">First Text</por>
    <div por-id="placeholder">
    </div>
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

Custom tags handle this incredibly well, as the only thing to note would be that the order of the nodes is different during the parsing, so the only difference would be a change in the metadata file.

However, one thing to note, the user would need to also move the tag that the text was in. Otherwise, the text node would be shown to be new, and the old tag would say it was edited in some manner, or perhaps deleted.

14.2 XML text-tail

Document before change

```
<span por-id="40E36DB5C0AD13957351" text="39E1D62AADCF588B873C">
    First Text
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1">
        div Text
    </div>
    <div por-id="placeholder" tail="3CED92A56695A78653ED">
    </div>
    Second Text
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351" >
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1" tail="39E1D62AADCF588B873C">
        div Text
    </div>
    First Text
    <div por-id="placeholder" tail="3CED92A56695A78653ED">
    </div>
    Second Text
</span>
```

Similar to custom tags, moves are adequately represented, as the text and tails can be scene to be added or removed. If the text or tail is missing or added, just look for if they were missing elsewhere to match up.

The only downside, is that while in custom tags, users need only move the whole tagged object, here, users need to move the individual attributes. The hard part here, is that there are possibly more things to move, and the users would need to move them to the proper place, which is harder to make clear for the user.

CHAPTER 15

Case 5: Moving an Edited Text Node

Consider the case of moving a text node that has also been edited. The document before the edit might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    First Text
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    <div por-id="placeholder">
    </div>
    Second Text
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    <div por-id="76F7BEE8A2001AC7144D">
        div Text
    </div>
    This is my new First Text
    <div por-id="placeholder">
    </div>
    Second Text
</span>
```

Here, the text was both moved and edited. Because the extent of edits can be quite large (meaning that edits can change as little as one character, or as much as all of the text), attempting to track whether a section was moved would be entirely based on either:

- The section being tagged in some way as being

a section that already existed, thus making *any* amount of change trackable as a move * A heuristic-based text comparison, where we base the decision of if the edit was a move or an insertion based on the quantity of text changed.

In the first case, we would need one of our tracking methods, and would require the user to use the method properly, which might not happen. However, if we chose to do that, the actual process for checking for a move would be quite easy.

In the second case, there would be quite a bit that needs to be decided, such as, how much of the text needs to be the same to count as a move. The process would also take more work, but, if we did it this way, the user just wouldn't need tracking to identify both the move and then change.

15.1 Custom Tags

Before the edit

```
<span por-id="40E36DB5C0AD13957351">
    <por por-id="39E1D62AADCF588B873C ">First Text</por>
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
    </div>
    <div por-id="placeholder">
    </div>
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351">
    <div por-id="76F7BEE8A2001AC7144D">
        <por por-id="56ADFDCAACEB1FDBCEA1">div Text</por>
    </div>
    <por por-id="39E1D62AADCF588B873C ">This is my new First Text</por>
    <div por-id="placeholder">
    </div>
    <por por-id="3CED92A56695A78653ED">Second Text</por>
</span>
```

Custom tags track this just fine, but the entire basis of tracking this type of change is that the user *actually* properly moves and keeps the custom tags. If they do that, tracking moves & changes would be straightforward, as we can just look at the section itself and the order of the metadata.

15.2 XML text-tail

Document before change

```
<span por-id="40E36DB5C0AD13957351" text="39E1D62AADCF588B873C">
    First Text
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1">
        div Text
    </div>
    <div por-id="placeholder" tail="3CED92A56695A78653ED">
    </div>
    Second Text
</span>
```

After the edit, the document might look like this:

```
<span por-id="40E36DB5C0AD13957351" >
    <div por-id="76F7BEE8A2001AC7144D" text="56ADFDCAACEB1FDBCEA1" tail="39E1D62AADCF588B873C">
        div Text
    </div>
    This is my new First Text
    <div por-id="placeholder" tail="3CED92A56695A78653ED">
    </div>
    Second Text
</span>
```

Again, text-tail keeps track of this change reasonably, so long as the user moves the tracking information as well. If they don't, then there is no "free" way for us to track this, as we dont' necessarily know what was in each text node without comparing all of the text nodes in the document against each other text node.

CHAPTER 16

A Note about Diff implementation

The only reason why we care about tracking text nodes & what they are named is so that we can reasonably identify moved text nodes across the document whenever we look at doing a diff.

If we didn't care about noting which sections moved, then we could just state that each time there was a missing section, regardless of the content, it was deleted, if there was a new section, it was an addition, and if the sections line up, but are different text-wise, it was a modification. This would still give visibility of the change, it just wouldn't explicitly state it was move.

Because of this, and the fact that we will likely need to do at least *some* text comparison while doing our diff (whether it be git diff, us, or some other library doing it), the tagging information might seem superfluous. In addition to this, it might be smarter to just do a heuristic-based regardless since that is a manner that makes sense, after all: If almost all of the text was changed, and only ~20% of the text is the same, couldn't it just possibly be coincidence?

CHAPTER 17

Design Decision

As of right now, both Custom tags **and** XML-style text-tail would properly track all the changes, considering that they were used properly.

The real issue comes with that last statement: "considering they were used properly". The thing is, we don't have any guarantees that our users *will*.

The best we can hope for is explain *what* the tracking method does, what it's used for, and how to make use of the tracking method so they don't accidentally break the text-node tracking. However, there is still the chance for user error, or misunderstanding.

In the case of custom tags, every time the user wants to move a text node, they would need to also move the surrounding custom tag, which could be a pain to do. However, with text-tail, they would not only have to move the text, but also:

- **The text attribute for the parent node**
  - Also *know* which node the attribute needs to move to
- **The tail attribute if it follows the**
  - Also *when* to add it to a node

Thus, text-tail tends to be a bit cleaner from a user *view*point, but it is much messier to work with, and has a larger chance for user-failure.

Custom tags do have another downside though: they could *possible* break other applications (browsers included) that use HTML, since the tags we add aren't part of the standard. However, there *is* a very simple way to get around this. We just add some sort of "release" feature for the document, where we create the HTML doc, but *don't* add the custom tags for tracking, that way any things that **we use for tracking purposes** (this includes por-ids), would be removed, and thus guaranteeing that the HTML does not contain any unknown elements.

Because of this, its higher visibility, and easier understandability for users, for now, we recommend and are going to go with custom tags.

Note: Again, note that we only *really* need tracking for text nodes if we want to do easy, non-text-comparison-based move detection. If we decided that text-comparison-based move detection is what we should do, or that move detection is not important, we would likely not do any sort of tagging to ensure that there is no possible problems injected into the HTML.

CHAPTER 18

Indices and tables

- genindex
- modindex
- search