ProseMirror Guide

This guide describes the various concepts used in the library, and how they relate to each other. To get a complete picture of the system, it is recommended to go through it in the order it is presented in, at least up to the view component section.

Introduction

ProseMirror provides a set of tools and concepts for building rich text editors, using user interface inspired by what-you-see-is-what-you-get, but trying to avoid the pitfalls of that style of editing.

The main principle of ProseMirror is that your code gets full control over the document and what happens to it. This document isn't a blob of HTML, but a custom data structure that only contains elements that you explicitly allow it to contain, in relations that you specified. All updates go through a single point, where you can inspect them and react to them.

The core library is not an easy drop-in component—we are prioritizing modularity and customizeability over simplicity, with the hope that, in the future, people will distribute drop-in editors based on ProseMirror. As such, this is more of a Lego set than a Matchbox car.

There are four core modules, which are required to do any editing at all, and a number of extension modules maintained by the core team, which have a status similar to that of 3rd party modules—they provide useful functionality, but you may omit them or replace them with other modules that implement similar functionality.

The core modules are:

- `prosemirror-model` defines the editor's document model, the data structure used to describe the content of the editor.

- `prosemirror-state` provides the data structure that describes the editor's whole state, including the selection, and a transaction system for moving from one state to the next.

- `prosemirror-view` implements a user interface component that shows a given editor state as an editable element in the browser, and handles user interaction with that element.

- `prosemirror-transform` contains functionality for modifying documents in a way that can be recorded and replayed, which is the basis for the transactions in the `state` module, and which makes the undo history and collaborative editing possible.

In addition, there are modules for basic editing commands, binding keys, undo history, input macros, collaborative editing, a simple document schema, and more under the GitHub prosemirror organization.

The fact that ProseMirror isn't distributed as a single, browser-loadable script means that you'll probably want to use some kind of bundler when using it. A bundler is a tool that automatically finds your script's dependencies, and combines them into a single big file that you can easily load from a web page. You can read more about bundling on the web, for example here.

My first editor

The Lego pieces fit together like this to create a very minimal editor:

```
import {schema} from "prosemirror-schema-basic"
import {EditorState} from "prosemirror-state"
import {EditorView} from "prosemirror-view"

let state = EditorState.create({schema})
let view = new EditorView(document.body, {state})
```

ProseMirror requires you to specify a schema that your document conforms to, so the first thing this does is import a module with a basic schema in it.

That schema is then used to create a state, which will generate an empty document conforming to the schema, and a default selection at the start of that document. Finally, a view is created for the state, and appended to `document.body`. This will render the state's document as an editable DOM node, and generate state transactions whenever the user types into it.

The editor isn't very usable yet. If you press enter, for example, nothing happens, because the core library has no opinion on what enter should do. We'll get to that in a moment.

Transactions

When the user types, or otherwise interacts with the view, it generates 'state transactions'. What that means is that it does not just modify the document in-place and implicitly update its state in that way. Instead, every change causes a *transaction* to be created, which describes the changes that are made to the state, and can be applied to create a *new* state, which is then used to update the view.

By default this all happens under the cover, but you can hook into by writing plugins or configuring your view. For example, this code adds a `dispatchTransaction` prop, which will be called whenever a transaction is created:

```
// (Imports omitted)

let state = EditorState.create({schema})
let view = new EditorView(document.body, {
  state,
  dispatchTransaction(transaction) {
    console.log("Document size went from", transaction.before.content.size,
                "to", transaction.doc.content.size)
    let newState = view.state.apply(transaction)
    view.updateState(newState)
  }
})
```

Every state update has to go through `updateState`, and every normal editing update will happen by dispatching a transaction.

Plugins

Plugins are used to extend the behavior of the editor and editor state in various ways. Some are relatively simple, like the keymap plugin that binds actions to keyboard input. Others are more involved, like the history plugin which implements an undo history by observing transactions and storing their inverse in case the user wants to undo them.

Let's add those two plugins to our editor to get undo/redo functionality:

```
// (Omitted repeated imports)
import {undo, redo, history} from "prosemirror-history"
import {keymap} from "prosemirror-keymap"

let state = EditorState.create({
  schema,
  plugins: [
    history(),
    keymap({"Mod-z": undo, "Mod-y": redo})
  ]
})
let view = new EditorView(document.body, {state})
```

Plugins are registered when creating a state (because they get access to state transactions). After creating a view for this history-enabled state, you'll be able to press Ctrl-Z (or Cmd-Z on OS X) to undo your last change.

Commands

The `undo` and `redo` values that the previous example bound to keys are a special kind of functions called *commands*. Most editing actions are written as commands which can be bound to keys, hooked up to menus, or otherwise exposed to the user.

The `prosemirror-commands` package provides a number of basic editing commands, along with a minimal keymap that you'll probably want to enable to have things like enter and delete do the expected thing in your editor.

```
// (Omitted repeated imports)
import {baseKeymap} from "prosemirror-commands"

let state = EditorState.create({
  schema,
  plugins: [
    history(),
    keymap({"Mod-z": undo, "Mod-y": redo}),
    keymap(baseKeymap)
  ]
})
let view = new EditorView(document.body, {state})
```

At this point, you have a basically working editor.

To add a menu, additional keybindings for schema-specific things, and so on, you might want to look into the `prosemirror-example-setup` package. This is a module that provides you with an array of plugins that set up a baseline editor, but as the name suggests, it is meant more as an example than as a production-level library. For a real-world deployment, you'll probably want to replace it with custom code that sets things up exactly the way you want.

Content

A state's document lives under its `doc` property. This is a read-only data structure, representing the document as a hierarchy of nodes, somewhat like the browser DOM. A simple document might be a `"doc"` node containing two `"paragraph"` nodes, each containing a single `"text"` node. You can read more about the document data structure in the guide about it.

When initializing a state, you can give it an initial document to use. In that case, the `schema` field is optional, since the schema can be taken from the document.

Here we initialize a state by parsing the content found in the DOM element with the ID `"content"`, using the DOM parser mechanism, which uses information supplied by the schema about which DOM nodes map to which elements in that schema:

```
import {DOMParser} from "prosemirror-model"
import {EditorState} from "prosemirror-state"
import {schema} from "prosemirror-schema-basic"

let content = document.getElementById("content")
let state = EditorState.create({
  doc: DOMParser.fromSchema(schema).parse(content)
})
```

Documents

ProseMirror defines its own data structure to represent content documents. Since documents are the central element around which the rest of the editor is built, it is helpful to understand how they work.

Structure

A ProseMirror document is a node, which holds a fragment containing zero or more child nodes.

This is a lot like the browser DOM, in that it is recursive and tree-shaped. But it differs from the DOM in the way it stores inline content.

In HTML, a paragraph with markup is represented as a tree, like this:

```
<p>This is <strong>strong text with <em>emphasis</em></strong></p>
```

p
"This is "
strong

"strong text with "

em
"emphasis"

Whereas in ProseMirror, the inline content is modeled as a flat sequence, with the markup attached as metadata to the nodes:

paragraph

"This is "

"strong text with "

strong

"emphasis"

strong

em

This more closely matches the way we tend to think about and work with such text. It allows us to represent positions in a paragraph using a character offset rather than a path in a tree, and makes it easier to perform operations like splitting or changing the style of the content without performing awkward tree manipulation.

This also means each document has *one* valid representation. Adjacent text nodes with the same set of marks are always combined together, and empty text nodes are not allowed. The order in which marks appear is specified by the schema.

So a ProseMirror document is a tree of block nodes, with most of the leaf nodes being *textblocks*, which are block nodes that contain text. You can also have leaf blocks that are simply empty, for example a horizontal rule or a video element.

Node objects come with a number of properties that reflect the role they play in the document:

- `isBlock` and `isInline` tell you whether a given node is a block or inline node.
- `inlineContent` is true for nodes that expect inline nodes as content.
- `isTextblock` is true for block nodes with inline content.
- `isLeaf` tells you that a node doesn't allow any content.

So a typical `"paragraph"` node will be a textblock, whereas a blockquote might be a block element whose content consists of other blocks. Text, hard breaks, and inline images are inline leaf nodes, and a horizontal rule node would be an example of a block leaf node.

The schema is allowed to specify more precise constraints on what may appear where—i.e. even though a node allows block content, that doesn't mean that it allows *all* block nodes as content.

Identity and persistence

Another important difference between a DOM tree and a ProseMirror document is the way the objects that represent nodes behave. In the DOM, nodes are mutable objects with an *identity*, which means that a node can only appear in one parent node, and that the node object is mutated when it is updated.

In ProseMirror, on the other hand, nodes are simply *values*, and should be approached much as you'd approach the value representing the number 3. 3 can appear in multiple data structures at the same time, it does not have a parent-link to the data structure it is currently part of, and if you add 1 to it, you get a *new* value, 4, without changing anything about the original 3.

So it is with pieces of ProseMirror documents. They don't change, but can be used as a starting value to compute a modified piece of document. They don't know what data structures they are part of, but can be part of multiple structures, or even occur multiple times in a single structure. They are *values*, not stateful objects.

This means that every time you update a document, you get a new document value. That document value will share all sub-nodes that didn't change with the original document value, making it relatively cheap to create.

This has a bunch of advantages. It makes it impossible to have an editor in an invalid in-between state during an update, since the new state, with a new document, can be swapped in instantaneously. It also makes it easier to reason about documents in a somewhat mathematical way, which is really hard if your values keep changing underneath you. This helps make collaborative editing possible and allows ProseMirror to run a very efficient DOM update algorithm by comparing the last document it drew to the screen to the current document.

Because such nodes are represented by regular JavaScript objects, and explicitly freezing their properties hampers performance, it is actually *possible* to change them. But doing this is not supported, and will cause things to break, because they are almost always shared between multiple data structures. So be careful! And note that this also holds for the arrays and plain objects that are *part* of node objects, such as the objects used to store node attributes, or the arrays of child nodes in fragments.

Data structures

The object structure for a document looks something like this:

Node	
type:	**NodeType**
content:	**Fragment** [**Node** ,

	Node , ...]
attrs:	**Object**
marks:	[**Mark** type: **MarkType** attrs: **Object** , ...]

Each node is represented by an instance of the `Node` class. It is tagged with a type, which knows the node's name, the attributes that are valid for it, and so on. Node types (and mark types) are created once per schema, and know which schema they are part of.

The content of a node is stored in an instance of `Fragment`, which holds a sequence of nodes. Even for nodes that don't have or don't allow content, this field is filled (with the shared empty fragment).

Some node types allow attributes, which are extra values stored with each node. For example, an image node might use these to store its alt text and the URL of the image.

In addition, inline nodes hold a set of active marks—things like emphasis or being a link—which are represented as an array of `Mark` instances.

A full document is just a node. The document content is represented as the top-level node's child nodes. Typically, it'll contain a series of block nodes, some of which may be textblocks that contain inline content. But the top-level node may also be a textblock itself, so that the document contains only inline content.

What kind of node is allowed where is determined by the document's schema. To programatically create nodes, you must go through the schema, for example using the `node` and `text` methods.

```
import {schema} from "prosemirror-schema-basic"
```

```
// (The null arguments are where you can specify attributes, if necessary.)
let doc = schema.node("doc", null, [
  schema.node("paragraph", null, [schema.text("One.")]),
  schema.node("horizontal_rule"),
  schema.node("paragraph", null, [schema.text("Two!")])
])
```

Indexing

ProseMirror nodes support two types of indexing—they can be treated as trees, using offsets into individual nodes, or they can be treated as a flat sequence of tokens.

The first allows you to do things similar to what you'd do with the DOM—interacting with single nodes, directly accessing child nodes using the `child` method and `childCount`, writing recursive functions that scan through a document (if you just want to look at all nodes, use `descendants` or `nodesBetween`).

The second is more useful when addressing a specific position in the document. It allows any document position to be represented as an integer—the index in the token sequence. These tokens don't actually exist as objects in memory—they are just a counting convention—but the document's tree shape, along with the fact that each node knows its size, is used to make by-position access cheap.

- The start of the document, right before the first content, is position 0.
- Entering or leaving a node that is not a leaf node (i.e. supports content) counts as one token. So if the document starts with a paragraph, the start of that paragraph counts as position 1.
- Each character in text nodes counts as one token. So if the paragraph at the start of the document contains the word "hi", position 2 is after the "h", position 3 after the "i", and position 4 after the whole paragraph.

- Leaf nodes that do not allow content (such as images) also count as a single token.

So if you have a document that, when expressed as HTML, would look like this:

```
<p>One</p>
<blockquote><p>Two<img src="..."></p></blockquote>
```

The token sequence, with positions, looks like this:

```
0   1 2 3 4    5
 <p> O n e </p>

5            6   7 8 9 10    11   12            13
 <blockquote> <p> T w o <img> </p> </blockquote>
```

Each node has a `nodeSize` property that gives you the size of the entire node, and you can access `.content.size` to get the size of the node's *content*. Note that for the outer document node, the open and close tokens are not considered part of the document (because you can't put your cursor outside of the document), so the size of a document is `doc.content.size`, **not** `doc.nodeSize`.

Interpreting such position manually involves quite a lot of counting. You can call `Node.resolve` to get a more descriptive data structure for a position. This data structure will tell you what the parent node of the position is, what its offset into that parent is, what ancestors the parent has, and a few other things.

Take care to distinguish between child indices (as per `childCount`), document-wide positions, and node-local offsets (sometimes used in recursive functions to represent a position into the node that's currently being handled).

Slices

To handle things like copy-paste and drag-drop, it is necessary to be able to talk about a slice of document, i.e. the content between two positions. Such a slice differs from a full node or fragment in that some of the nodes at its start or end may be ‘open’.

For example, if you select from the middle of one paragraph to the middle of the next one, the slice you've selected has two paragraphs in it, the first one open at the start, the second open at the end, whereas if you node-select a paragraph, you've selected a closed node. It may be the case that the content in such open nodes violates the schema constraints, if treated like the node's full content, because some required nodes fell outside of the slice.

The `Slice` data structure is used to represent such slices. It stores a fragment along with an open depth on both sides. You can use the `slice` method on nodes to cut a slice out of a document.

```
// doc holds two paragraphs, containing text "a" and "b"
let slice1 = doc.slice(0, 3) // The first paragraph
console.log(slice1.openStart, slice1.openEnd) // → 0 0
let slice2 = doc.slice(1, 5) // From start of first paragraph
                             // to end of second
console.log(slice2.openStart, slice2.openEnd) // → 1 1
```

Changing

Since nodes and fragments are persistent, you should **never** mutate them. If you have a handle to a document (or node, or fragment) that object will stay the same.

Most of the time, you'll use transformations to update documents, and won't have to directly touch the nodes. These also leave a record of the changes, which is necessary when the document is part of an editor state.

In cases where you do want to 'manually' derive an updated document, there are some helper methods available on the `Node` and `Fragment` types.

To create an updated version of a whole document, you'll usually want to use `Node.replace`, which replaces a given range of the document with a slice of new content. To update a node shallowly, you can use its `copy` method, which creates a similar node with new content. Fragments also have various updating methods, such as `replaceChild` or `append`.

Schemas

Each ProseMirror document has a schema associated with it. The schema describes the kind of nodes that may occur in the document, and the way they are nested. For example, it might say that the top-level node can contain one or more blocks, and that paragraph nodes can contain any number of inline nodes, with any marks applied to them.

There is a package with a basic schema available, but the nice thing about ProseMirror is that it allows you to define your own schemas.

Node Types

Every node in a document has a type, which represents its semantic meaning and its properties, such as the way it is rendered in the editor.

When you define a schema, you enumerate the node types that may occur within it, describing each with a spec object:

```
const trivialSchema = new Schema({
  nodes: {
    doc: {content: "paragraph+"},
    paragraph: {content: "text*"},
    text: {inline: true},
    /* ... and so on */
  }
})
```

That defines a schema where the document may contain one or more paragraphs, and each paragraph can contain any amount of text.

Every schema must at least define a top-level node type (which defaults to the name "doc", but you can configure that), and a "text" type for text content.

Nodes that count as inline must declare this with the inline property (though for the text type, which is inline by definition, you may omit this).

Content Expressions

The strings in the content fields in the example schema above are called *content expressions*. They control what sequences of child nodes are valid for this node type.

You can say, for example "paragraph" for “one paragraph”, or "paragraph+" to express “one or more paragraphs”. Similarly, "paragraph*" means “zero or more paragraphs” and "caption?" means “zero or one caption node”. You can also use regular-expression-like ranges, such as {2} (“exactly two”) {1, 5} (“one to five”) or {2,} (“two or more”) after node names.

Such expressions can be combined to create a sequence, for example "heading paragraph+" means ‘first a heading, then one or more paragraphs’. You can also use the pipe | operator to indicate a choice between two expressions, as in "(paragraph | blockquote)+".

Some groups of element types will appear multiple types in your schema —for example you might have a concept of “block” nodes, that may appear at the top level but also nested inside of blockquotes. You can create a node group by giving your node specs a group property, and then refer to that group by its name in your expressions.

```
const groupSchema = new Schema({
  nodes: {
    doc: {content: "block+"},
    paragraph: {group: "block", content: "text*"},
    blockquote: {group: "block", content: "block+"},
    text: {}
  }
})
```

Here `"block+"` is equivalent to `"(paragraph | blockquote)+"`.

It is recommended to always require at least one child node in nodes that have block content (such as `"doc"` and `"blockquote"` in the example above), because browsers will completely collapse the node when it's empty, making it rather hard to edit.

The order in which your nodes appear in an or-expression is significant. When creating a default instance for a non-optional node, for example to make sure a document still conforms to the schema after a replace step the first type in the expression will be used. If that is a group, the first type in the group (determined by the order in which the group's members appear in your `nodes` map) is used. If I switched the positions of `"paragraph"` and `"blockquote"` in the the example schema, you'd get a stack overflow as soon as the editor tried to create a block node—it'd create a `"blockquote"` node, whose content requires at least one block, so it'd try to create another `"blockquote"` as content, and so on.

Not every node-manipulating function in the library checks that it is dealing with valid content—higher level concepts like transforms do, but primitive node-creation methods usually don't and instead put the responsibility for providing sane input on their caller. It is perfectly possible to use, for example `NodeType.create`, to create a node with invalid content. For nodes that are 'open' on the edge of slices, this is even a reasonable thing to do. There is a separate `createChecked` method, as well as an after-the-fact `check` method that can be used to assert that a given node's content is valid.

Marks

Marks are used to add extra styling or other information to inline content. A schema must declare all mark types it allows in its schema. Mark types are objects much like node types, used to tag mark objects and provide additional information about them.

By default, nodes with inline content allow all marks defined in the schema to be applied to their children. You can configure this with the `marks` property on your node spec.

Here's a simple schema that supports strong and emphasis marks on text in paragraphs, but not in headings:

```
const markSchema = new Schema({
  nodes: {
    doc: {content: "block+"},
    paragraph: {group: "block", content: "text*", marks: "_"},
    heading: {group: "block", content: "text*", marks: ""},
    text: {inline: true}
  },
  marks: {
    strong: {},
    em: {}
  }
})
```

The set of marks is interpreted as a space-separated string of mark names or mark groups—`"_"` acts as a wildcard, and the empty string corresponds to the empty set.

Attributes

The document schema also defines which *attributes* each node or mark has. If your node type requires extra node-specific information to be stored, such as the level of a heading node, that is best done with an attribute.

Attribute sets are represented as plain objects with a predefined (per node or mark) set of properties holding any JSON-serializeable values. To specify what attributes it allows, use the optional `attrs` field in a node or mark spec.

```
  heading: {
    content: "text*",
    attrs: {level: {default: 1}}
  }
```

In this schema, every instance of the `heading` node will have a `level` attribute under `.attrs.level`. If it isn't specified when the node is created, it will default to 1.

When you don't give a default value for an attribute, an error will be raised when you attempt to create such a node without specifying that attribute. It will also make it impossible for the library to generate such nodes as filler to satisfy schema constraints during a transform or when calling `createAndFill`.

Serialization and Parsing

In order to be able to edit them in the browser, it must be possible to represent document nodes in the browser DOM. The easiest way to do that is to include information about each node's DOM representation in the schema using the `toDOM` field in the node spec.

This field should hold a function that, when called with the node as argument, returns a description of the DOM structure for that node. This may either be a direct DOM node or an array describing it, for example:

```
const schema = new Schema({
  nodes: {
    doc: {content: "paragraph+"},
    paragraph: {
      content: "text*",
      toDOM(node) { return ["p", 0] }
    },
    text: {}
  }
})
```

The expression `["p", 0]` declares that a paragraph is rendered as an HTML `<p>` tag. The zero is the 'hole' where its content should be rendered. You may also include an object with HTML attributes after the tag name, for example `["div", {class: "c"}, 0]`. Leaf nodes don't need a hole in their DOM representation, since they don't have content.

Mark specs allow a similar `toDOM` method, but they are required to render as a single tag that directly wraps the content, so the content always goes directly in the returned node, and the hole doesn't need to be specified.

You'll also often need to *parse* a document from DOM data, for example when the user pastes or drags something into the editor. The model module also comes with functionality for that, and you are encouraged to include parsing information directly in your schema with the `parseDOM` property.

This may list an array of *parse rules*, which describe DOM constructs that map to a given node or mark. For example, the basic schema has these for the emphasis mark:

```
parseDOM: [
  {tag: "em"},                 // Match <em> nodes
  {tag: "i"},                  // and <i> nodes
  {style: "font-style=italic"} // and inline 'font-style: italic'
]
```

The value given to `tag` in a parse rule can be a CSS selector, so you can do thing like `"div.myclass"` too. Similarly, `style` matches inline CSS styles.

When a schema includes `parseDOM` annotations, you can create a `DOMParser` object for it with `DOMParser.fromSchema`. This is done by the editor to create the default clipboard parser, but you can also override that.

Documents also come with a built-in JSON serialization format. You can call `toJSON` on them to get an object that can safely be passed to

`JSON.stringify`, and schema objects have a `nodeFromJSON` method that can parse this representation back into a document.

Extending a schema

The `nodes` and `marks` options passed to the `Schema` constructor take `OrderedMap` objects as well as plain JavaScript objects. The resulting schema's `spec.nodes` and `spec.marks` properties are always `OrderedMap`s, which can be used as the basis for further schemas.

Such maps support a number of methods to conveniently create updated versions. For example you could say `schema.markSpec.remove("blockquote")` to derive a set of nodes without the `blockquote` node, which can then be passed as the `nodes` field for a new schema.

The schema-list module exports a convenience method to add the nodes exported by those modules to a nodeset.

Document transformations

Transforms are central to the way ProseMirror works. They form the basis for transactions, and are what makes history tracking and collaborative editing possible.

Why?

Why can't we just mutate the document and be done with it? Or at least create a new version of a document and just put that into the editor?

There are several reasons. One is code clarity. Immutable data structures really do lead to simpler code. But the main thing the transform system does is to leave a *trail* of updates, in the form of values that represent the individual steps taken to go from an old version of the document to a new one.

The undo history can save these steps and apply their inverse to go back in time (ProseMirror implements selective undo, which is more complicated than just rolling back to a previous state).

The collaborative editing system sends these steps to other editors and reorders them if necessary so that everyone ends up with the same document.

More generally, it is very useful for editor plugins to be able to inspect and react to each change as it comes in, in order to keep their own state consistent with the rest of the editor state.

Steps

Updates to documents are decomposed into steps that describe an update. You usually don't need to work with these directly, but it is useful to know how they work.

Examples of steps are `ReplaceStep` to replace a piece of a document, or `AddMarkStep` to add a mark to a given range.

A step can be applied to a document to produce a new document.

```
console.log(myDoc.toString()) // → p("hello")
// A step that deletes the content between positions 3 and 5
let step = new ReplaceStep(3, 5, Slice.empty)
let result = step.apply(myDoc)
console.log(result.doc.toString()) // → p("heo")
```

Applying a step is a relatively straightforward process—it doesn't do anything clever like inserting nodes to preserve schema constraints, or transforming the slice to make it fit. That means applying a step can fail, for example if you try to delete just the opening token of a node, that would leave the tokens unbalanced, which isn't a meaningful thing you can do. This is why `apply` returns a result object, which holds either a new document, *or* an error message.

You'll usually want to let helper functions generate your steps for you, so that you don't have to worry about the details.

Transforms

An editing action may produce one or more steps. The most convenient way to work with a sequence of steps is to create a `Transform` object (or, if you're working with a full editor state, a `Transaction`, which is a subclass of `Transform`).

```
let tr = new Transform(myDoc)
tr.delete(5, 7) // Delete between position 5 and 7
tr.split(5)     // Split the parent node at position 5
console.log(tr.doc.toString()) // The modified document
console.log(tr.steps.length)   // → 2
```

Most transform methods return the transform itself, for convenient chaining (allowing you to do `tr.delete(5, 7).split(5)`).

There are transform methods for deleting and replacing, for adding and removing marks, for performing tree manipulation like splitting, joining, lifting, and wrapping, and more.

Mapping

When you make a change to a document, positions pointing into that document may become invalid or change meaning. For example, if you insert a character, all positions after that character now point one token before their old position. Similarly, if you delete all the content in a document, all positions pointing into that content are now invalid.

We often do need to preserve positions across document changes, for example the selection boundaries. To help with this, steps can give you a *map* that can convert between positions in the document before and after applying the step.

```
let step = new ReplaceStep(4, 6, Slice.empty) // Delete 4-5
let map = step.getMap()
console.log(map.map(8)) // → 6
console.log(map.map(2)) // → 2 (nothing changes before the change)
```

Transform objects automatically accumulate a set of maps for the steps in them, using an abstraction called `Mapping`, which collects a series of step maps and allows you to map through them in one go.

```
let tr = new Transaction(myDoc)
tr.split(10)    // split a node, +2 tokens at 10
tr.delete(2, 5) // -3 tokens at 2
console.log(tr.mapping.map(15)) // → 14
console.log(tr.mapping.map(6))  // → 3
console.log(tr.mapping.map(10)) // → 9
```

There are cases where it's not entirely clear what a given position should be mapped to. Consider the last line of the example above. Position 10 points precisely at the point where we split a node, inserting two tokens. Should it be mapped to the position *after* the inserted content, or stay in front of it? In the example, it is apparently moved after the inserted tokens.

But sometimes you want the other behavior, which is why the `map` method on step maps and mappings accepts a second parameter, `bias`, which you can set to -1 to keep your position in place when content is inserted on top of it.

```
console.log(tr.mapping.map(10, -1)) // → 7
```

The reason that individual steps are defined as small, straightforward things is that it makes this kind of mapping possible, along with inverting steps in a lossless way, and mapping steps through each other's position maps.

Rebasing

When doing more complicated things with steps and position maps, for example to implement your own change tracking, or to integrate some feature with collaborative editing, you might run into the need to *rebase* steps.

You might not want to bother studying this until you are sure you need it.

Rebasing, in the simple case, is the process of taking two steps that start with the same document, and transform one of them so that it can be applied to the document created by the other instead. In pseudocode:

```
stepA(doc) = docA
stepB(doc) = docB
stepB(docA) = MISMATCH!
rebase(stepB, mapA) = stepB'
stepB'(docA) = docAB
```

Steps have a `map` method, which, given a mapping, maps the whole step through it. This can fail, since some steps don't make sense anymore when, for example, the content they applied to has been deleted. But when it succeeds, you now have a step pointing into a new document, i.e. the one after the changes that you mapped through. So in the above example, `rebase(stepB, mapA)` can simply call `stepB.map(mapA)`.

Things get more complicated when you want to rebase a chain of steps over another chain of steps.

```
stepA2(stepA1(doc)) = docA
stepB2(stepB1(doc)) = docB
???(docA) = docAB
```

We can map `stepB1` over `stepA1` and then `stepA2`, to get `stepB1'`. But with `stepB2`, which starts at the document produced by `stepB1(doc)`, and whose mapped version must apply to the document produced by `stepB1'(docA)`, things get more difficult. It must be mapped over the following chain of maps:

```
rebase(stepB2, [invert(mapB1), mapA1, mapA2, mapB1'])
```

I.e. first the inverse of the map for `stepB1` to get back to the original document, then through the pipeline of maps produced by applying `stepA1` and `stepA2`, and finally through the map produced by applying `stepB1'` to `docA`.

If there was a `stepB3`, we'd get the pipeline for that one by taking the one above, prefixing it with `invert(mapB2)` and adding `mapB2'` to the end. And so on.

But when `stepB1` inserted some content, and `stepB2` did something to that content, then mapping `stepB2` through `invert(mapB1)` will return `null`, because the inverse of `stepB1` *deletes* the content to which it applies. However, this content is reintroduced later in the pipeline, by `mapB1`. The `Mapping` abstraction provides a way to track such pipelines, including the inverse relations between the maps in it. You can map steps through it in such a way that they survive situations like the one above.

Even if you have rebased a step, there is no guarantee that it can still be validly applied to the current document. For example, if your step adds a mark, but another step changed the parent node of your target content to be a node that doesn't allow marks, trying to apply your step will fail. The appropriate response to this is usually just to drop the step.

The editor state

What makes up the state of an editor? You have your document, of course. And also the current selection. And there needs to be a way to store the fact that the current set of marks has changed, when you for example disable or enable a mark but haven't started typing with that mark yet.

Those are the three main components of a ProseMirror state, and exist on state objects as `doc`, `selection`, and `storedMarks`.

```
import {schema} from "prosemirror-schema-basic"
import {EditorState} from "prosemirror-state"

let state = EditorState.create({schema})
console.log(state.doc.toString()) // An empty paragraph
console.log(state.selection.from) // 1, the start of the paragraph
```

But plugins may also need to store state—for example, the undo history has to keep its history of changes. This is why the set of active plugins is also stored in the state, and these plugins can define additional slots for storing their own state.

Selection

ProseMirror supports several types of selection (and allows 3rd-party code to define new selection types). Selections are represented by instances of (subclasses of) the `Selection` class. Like documents and other state-related values, they are immutable—to change the selection, you create a new selection object and a new state to hold it.

Selections have, at the very least, a start (`.from`) and an end (`.to`), as positions pointing into the current document. Many selection types also distinguish between the *anchor* (unmoveable) and *head* (moveable) side of the selection, so those are also required to exist on every selection object.

The most common type of selection is a text selection, which is used for regular cursors (when `anchor` and `head` are the same) or selected text. Both endpoints of a text selection are required to be in inline positions, i.e. pointing into nodes that allow inline content.

The core library also supports node selections, where a single document node is selected, which you get, for example, when you ctrl/cmd-click a node. Such a selection ranges from the position directly before the node to the position directly after it.

Transactions

During normal editing, new states will be derived from the state before them. You may in some situations, such as loading a new document, want to create a completely new state, but this is the exception.

State updates happen by applying a transaction to an existing state, producing a new state. Conceptually, they happen in a single shot: given the old state and the transaction, a new value is computed for each component of the state, and those are put together in a new state value.

```
let tr = state.tr
console.log(tr.doc.content.size) // 25
tr.insertText("hello") // Replaces selection with 'hello'
let newState = state.apply(tr)
console.log(tr.doc.content.size) // 30
```

`Transaction` is a subclass of `Transform`, and inherits the way it builds up a new document by applying steps to an initial document. In addition to this, transactions track selection and other state-related components, and get some selection-related convenience methods such as `replaceSelection`.

The easiest way to create a transaction is with the `tr` getter on an editor state object. This creates an empty transaction based on that state, to which you can then add steps and other updates.

By default, the old selection is mapped through each step to produce a new selection, but it is possible to use `setSelection` to explicitly set a new selection.

```
let tr = state.tr
console.log(tr.selection.from) // → 10
tr.delete(6, 8)
console.log(tr.selection.from) // → 8 (moved back)
tr.setSelection(TextSelection.create(tr.doc, 3))
console.log(tr.selection.from) // → 3
```

Similarly, the set of active marks is automatically cleared after a document or selection change, and can be set using the `setStoredMarks` or `ensureMarks` methods.

Finally, the `scrollIntoView` method can be used to ensure that, the next time the state is drawn, the selection is scrolled into view. You probably want to do that for most user actions.

Like `Transform` methods, many `Transaction` methods return the transaction itself, for convenient chaining.

Plugins

When creating a new state, you can provide an array of plugins to use. These will be stored in the state and any state that is derived from it, and can influence both the way transactions are applied and the way an editor based on this state behaves.

Plugins are instances of the `Plugin` class, and can model a wide variety of features. The simplest ones just add some props to the editor view, for example to respond to certain events. More complicated ones might add new state to the editor and update it based on transactions.

When creating a plugin, you pass it an object specifying its behavior:

```
let myPlugin = new Plugin({
  props: {
    handleKeyDown(view, event) {
      console.log("A key was pressed!")
      return false // We did not handle this
    }
  }
})

let state = EditorState.create({schema, plugins: [myPlugin]})
```

When a plugin needs its own state slot, that is defined with a `state` property:

```
let transactionCounter = new Plugin({
  state: {
    init() { return 0 },
    apply(tr, value) { return value + 1 }
  }
})

function getTransactionCount(state) {
  return transactionCounter.getState(state)
}
```

The plugin in the example defines a very simple piece of state that simply counts the number of transactions that have been applied to a state. The helper function uses the plugin's `getState` method, which can be used to fetch the plugin state from a full editor state object.

Because the editor state is a persistent (immutable) object, and plugin state is part of that object, plugin state values must be immutable. I.e. their `apply` method must return a new value, rather than changing the old, if they need to change, and no other code should change them.

It is often useful for plugins to add some extra information to a transaction. For example, the undo history, when performing an actual undo, will mark the resulting transaction, so that when the plugin sees it, instead of doing the thing it normally does with changes (adding them to the undo stack), it treats it specially, removing the top item from the undo stack and adding this transaction to the redo stack instead.

For this purpose, transactions allow *metadata* to be attached to them. We could update our transaction counter plugin to not count transactions that are marked, like this:

```
let transactionCounter = new Plugin({
  state: {
    init() { return 0 },
    apply(tr, value) {
      if (tr.getMeta(transactionCounter)) return value
      else return value + 1
    }
  }
})

function markAsUncounted(tr) {
  tr.setMeta(transactionCounter, true)
}
```

Keys for metadata properties can be strings, but to avoid name collisions, you are encouraged to use plugin objects. There are some string keys that are given a meaning by the library, for example `"addToHistory"` can be set to `false` to prevent a transaction from being undoable, and when handling a paste, the editor view will set the `"paste"` property on the resulting transaction to true.

The view component

A ProseMirror editor view is a user interface component that displays an editor state to the user, and allows them to perform editing actions on it.

The definition of *editing actions* used by the core view component is rather narrow—it handles direct interaction with the editing surface, such as typing, clicking, copying, pasting, and dragging, but not much beyond that. This means that things like displaying a menu, or even providing a full set of key bindings, lie outside of the responsibility of the core view component, and have to be arranged through plugins.

Editable DOM

Browsers allow us to specify that some parts of the DOM are editable, which has the effect of allowing focus and a selection in them, and making it possible to type into them. The view creates a DOM representation of its document (using your schema's `toDOM` methods by default), and makes it editable. When the editable element is focused, ProseMirror makes sure that the DOM selection corresponds to the selection in the editor state.

It also registers event handlers for many DOM events, which translate the events into the appropriate transactions. For example, when pasting, the pasted content is parsed as a ProseMirror document slice, and then inserted into the document.

Many events are also let through as they are, and only *then* reinterpreted in terms of ProseMirror's data model. The browser is quite good at cursor and selection placement for example (which is a really difficult problem when you factor in bidirectional text), so most cursor-motion related keys and mouse actions are handled by the browser, after which ProseMirror checks what kind of text selection the current DOM selection would correspond to. If that selection is different from the current selection, a transaction that updates the selection is dispatched.

Even typing is usually left to the browser, because interfering with that tends to break spell-checking, autocapitalizing on some mobile interfaces, and other native features. When the browser updates the DOM, the editor notices, re-parses the changed part of the document, and translates the difference into a transaction.

Data flow

So the editor view displays a given editor state, and when something happens, it creates a transaction and broadcasts this. This transaction is then, typically, used to create a new state, which is given to the view using its `updateState` method.

DOM event

↗ ↘

EditorView

Transaction

↖ ↙

new **EditorState**

This creates a straightforward, cyclic data flow, as opposed to the classic approach (in the JavaScript world) of a host of imperative event handlers, which tends to create a much more complex web of data flows.

It is possible to 'intercept' transactions as they are dispatched with the `dispatchTransaction` prop, in order to wire this cyclic data flow into a larger cycle—if your whole app is using a data flow model like this, as with Redux and similar architectures, you can integrate ProseMirror's transactions in your main action-dispatching cycle, and keep ProseMirror's state in your application 'store'.

```
// The app's state
let appState = {
  editor: EditorState.create({schema}),
  score: 0
}

let view = new EditorView(document.body, {
  state: appState.editor,
  dispatchTransaction(transaction) {
    update({type: "EDITOR_TRANSACTION", transaction})
  }
})

// A crude app state update function, which takes an update object,
// updates the `appState`, and then refreshes the UI.
function update(event) {
  if (event.type == "EDITOR_TRANSACTION")
    appState.editor = appState.editor.apply(event.transaction)
  else if (event.type == "SCORE_POINT")
    appState.score++
  draw()
}

// An even cruder drawing function
function draw() {
  document.querySelector("#score").textContent = appState.score
  view.updateState(appState.editor)
}
```

Efficient updating

One way to implement `updateState` would be to simply redraw the document every time it is called. But for large documents, that would be really slow.

Since, at the time of updating, the view has access to both the old document and the new, it can compare them, and leave the parts of the DOM that correspond to unchanged nodes alone. ProseMirror does this, allowing it to do very little work for typical updates.

In some cases, like updates that correspond to typed text, which was already added to the DOM by the browser's own editing actions, ensuring the DOM and state are coherent doesn't require any DOM changes at all. (When such a transaction is canceled or modified somehow, the view *will* undo the DOM change to make sure the DOM and the state remain synchronized.)

Similarly, the DOM selection is only updated when it is actually out of sync with the selection in the state, to avoid disrupting the various pieces of 'hidden' state that browsers keep along with the selection (such as that feature where when you arrow down or up past a short line, you horizontal position goes back to where it was when you enter the next long line).

Props

'Props' is a useful, if somewhat vague, term taken from React. Props are like parameters to a UI component. Ideally, the set of props that the component gets completely defines its behavior.

```
let view = new EditorView({
  state: myState,
  editable() { return false }, // Enables read-only behavior
  handleDoubleClick() { console.log("Double click!") }
})
```

As such, the current state is one prop. The value of other props can also vary over time, if the code that controls the component updates them, but aren't considered *state*, because the component itself won't change them. The `updateState` method is just a shorthand to updating the `state` prop.

Plugins are also allowed to declare props, except for `state` and `dispatchTransaction`, which can only be provided directly to the view.

```
function maxSizePlugin(max) {
  return new Plugin({
    props: {
      editable(state) { return state.doc.content.size < max }
    }
  })
}
```

When a given prop is declared multiple times, how it is handled depends on the prop. In general, directly provided props take precedence, after which each plugin gets a turn, in order. For some props, such as `domParser`, the first value that is found is used, and others are ignored. For handler functions that return a boolean to indicate whether they handled the event, the first one that returns true gets to handle the event. And finally, for some props, such as `attributes` (which can be used to set attributes on the editable DOM node) and `decorations` (which we'll get to in the next section), the union of all provided values is used.

Decorations

Decorations give you some control over the way the view draws your document. They are created by returning values from the `decorations` prop, and come in three types:

- Node decorations add styling or other DOM attributes to a single node's DOM representation.
- Widget decorations *insert* a DOM node, which isn't part of the actual document, at a given position.

- Inline decorations add styling or attributes, much like node decorations, but to all inline nodes in a given range.

In order to be able to efficiently draw and compare decorations, they need to be provided as a decoration set (which is a data structure that mimics the tree shape of the actual document). You create one using the static `create` method, providing the document and an array of decoration objects:

```
let purplePlugin = new Plugin({
  props: {
    decorations(state) {
      return DecorationSet.create(state.doc, [
        Decoration.inline(0, state.doc.content.size, {style: "color: purple
      ])
    }
  }
})
```

When you have a lot of decorations, recreating the set on the fly for every redraw is likely to be too expensive. In such cases, the recommended way to maintain your decorations is to put the set in your plugin's state, map it forward through changes, and only change it when you need to.

```
let specklePlugin = new Plugin({
  state: {
    init(_, {doc}) {
      let speckles = []
      for (let pos = 1; pos < doc.content.size; pos += 4)
        speckles.push(Decoration.inline(pos - 1, pos, {style: "background:
      return DecorationSet.create(doc, speckles)
    },
    apply(tr, set) { return set.map(tr.mapping, tr.doc) }
  },
  props: {
    decorations(state) { return specklePlugin.getState(state) }
  }
})
```

This plugin initializes its state to a decoration set that adds a yellow-background inline decoration to every 4th position. That's not terribly useful, but sort of resembles use cases like highlighting search matches or annotated regions.

When a transaction is applied to the state, the plugin state's `apply` method maps the decoration set forward, causing the decorations to stay in place and 'fit' the new document shape. The mapping method is (for typical, local changes) made efficient by exploiting the tree shape of the decoration set—only the parts of the tree that are actually touched by the changes need to be rebuilt.

(In a real-world plugin, the `apply` method would also be the place where you add or remove decorations based on new events, possibly by inspecting the changes in the transaction, or based on plugin-specific metadata attached to the transaction.)

Finally, the `decorations` prop simply returns the plugin state, causing the decorations to show up in the view.

Node views

There is one more way in which you can influence the way the editor view draws your document. Node views make it possible to define a sort of miniature UI components for individual nodes in your document. They allow you to render their DOM, define the way they are updated, and write custom code to react to events.

```
let view = new EditorView({
  state,
  nodeViews: {
    image(node) { return new ImageView(node) }
  }
})

class ImageView {
  constructor(node) {
    // The editor will use this as the node's DOM representation
    this.dom = document.createElement("img")
    this.dom.src = node.attrs.src
    this.dom.addEventListener("click", e => {
      console.log("You clicked me!")
      e.preventDefault()
    })
  }

  stopEvent() { return true }
}
```

The view object that the example defines for image nodes creates its own custom DOM node for the image, with an event handler added, and declares, with a `stopEvent` method, that ProseMirror should ignore events coming from that DOM node.

You'll often want interaction with the node to have some effect on the actual node in the document. But to create a transaction that changes a node, you first need to know where that node is. To help with that, node views get passed a getter function that can be used to query their current position in the document. Let's modify the example so that clicking on the node queries you to enter an alt text for the image:

```
let view = new EditorView({
  state,
  nodeViews: {
    image(node, view, getPos) { return new ImageView(node, view, getPos) }
  }
})

class ImageView {
  constructor(node, view, getPos) {
    this.dom = document.createElement("img")
    this.dom.src = node.attrs.src
    this.dom.alt = node.attrs.alt
    this.dom.addEventListener("click", e => {
      e.preventDefault()
      let alt = prompt("New alt text:", "")
      if (alt) view.dispatch(view.state.tr.setNodeMarkup(getPos(), null, {
        src: node.attrs.src,
        alt
      }))
    })
  }
  stopEvent() { return true }
}
```

`setNodeMarkup` is a method that can be used to change the type or set of attributes for the node at a given position. In the example, we use `getPos` to find our image's current position, and give it a new attribute object with the new alt text.

When a node is updated, the default behavior is to leave its outer DOM structure intact and compare its children to the new set of children, updating or replacing those as needed. A node view can override this with custom behavior, which allows us to do something like changing the class of a paragraph based on its content.

```
let view = new EditorView({
  state,
  nodeViews: {
    paragraph(node) { return new ParagraphView(node) }
  }
})

class ParagraphView {
  constructor(node) {
    this.dom = this.contentDOM = document.createElement("p")
    if (node.content.size == 0) this.dom.classList.add("empty")
  }

  update(node) {
    if (node.type.name != "paragraph") return false
    if (node.content.size > 0) this.dom.classList.remove("empty")
    else this.dom.classList.add("empty")
    return true
  }
}
```

Images never have content, so in our previous example, we didn't need to worry about how that would be rendered. But paragraphs do have content. Node views support two approaches to handling content: you can let the ProseMirror library manage it, or you can manage it entirely yourself. If you provide a `contentDOM` property, the library will render the node's content into that, and handle content updates. If you don't, the content becomes a black box to the editor, and how you display it and let the user interact with it is entirely up to you.

In this case, we want paragraph content to behave like regular editable text, so the `contentDOM` property is defined to be the same as the `dom` property, since the content needs to be rendered directly into the outer node.

The magic happens in the `update` method. Firstly, this method is responsible for deciding whether the node view *can* be updated to show the new node at all. This new node may be anything that the editor's update algorithm might try to draw here, so you must verify that this is a node that this node view can handle.

The `update` method in the example first checks whether the new node is a paragraph, and bails out if that's not the case. Then it makes sure that the `"empty"` class is present or absent, depending on the content of the new node, and returns true, to indicate that the update succeeded (at which point the node's content will be updated).

Commands

In ProseMirror jargon, a *command* is a function that implements an editing action, which the user can perform by pressing some key combination or interacting with the menu.

For practical reasons, commands have a slightly convoluted interface. In their simple form, they are functions taking an editor state and a *dispatch function* (`EditorView.dispatch` or some other function that takes transactions), and return a boolean. Here's a very simple example:

```
function deleteSelection(state, dispatch) {
  if (state.selection.empty) return false
  dispatch(state.tr.deleteSelection())
  return true
}
```

When a command isn't applicable, it should return false and do nothing. When it is, it should dispatch a transaction and return true. This is used, for example, by the keymap plugin to stop further handling of key events when the command bound to that key has been applied.

To be able to query whether a command is applicable for a given state, without actually executing it, the `dispatch` argument is optional—commands should simply return true without doing anything when they are applicable but no `dispatch` argument is given. So the example command should actually look like this:

```
function deleteSelection(state, dispatch) {
  if (state.selection.empty) return false
  if (dispatch) dispatch(state.tr.deleteSelection())
  return true
}
```

To figure out whether a selection can currently be deleted, you'd call `deleteSelection(view.state, null)`, whereas to actually execute the command, you'd do something like `deleteSelection(view.state, view.dispatch)`. A menu bar could use this to determine which menu items to gray out.

In this form, commands do not get access to the actual editor view—most commands don't need that, and in this way they can be applied and tested in settings that don't have a view available. But some commands do need to interact with the DOM—they might need to query whether a given position is at the end of a textblock, or want to open a dialog positioned relative to the view. For this purpose, most plugins that call commands will give them a *third* argument, which is the whole view.

```
function blinkView(_state, dispatch, view) {
  if (dispatch) {
    view.dom.style.background = "yellow"
    setTimeout(() => view.dom.style.background = "", 1000)
  }
  return true
}
```

That (rather useless) example shows that commands don't *have* to dispatch a transaction—they are called for their side effect, which is *usually* to dispatch a transaction, but may also be something else, such as popping up a dialog.

The `prosemirror-commands` module provides a number of editing commands, from simple ones such as a variant of the `deleteSelection` command, to rather complicated ones such as `joinBackward`, which implements the block-joining behavior that should happen when you press backspace at the start of a textblock. It also comes with a basic keymap that binds a number of schema-agnostic commands to the keys that are usually used for them.

When possible, different behavior, even when usually bound to a single key, is put in different commands. The utility function `chainCommands` can be used to combine a number of commands—they will be tried one after the other until one return true.

For example, the base keymap binds backspace to the command chain `deleteSelection` (which kicks in when the selection isn't empty), `joinBackward` (when the cursor is at the start of a textblock), and `selectNodeBackward` (which selects the node before the selection, in case the schema forbids the regular joining behavior). When none of these apply, the browser is allowed to run its own backspace behavior, which is the appropriate thing for backspacing things out inside a textblock (so that native spell-check and such don't get confused).

The commands module also exports a number of command constructors, such as `toggleMark`, which takes a mark type and optionally a set of attributes, and returns a command function that toggles that mark on the current selection.

Some other modules also export command functions—for example `undo` and `redo` from the history module. To customize your editor, or to allow users to interact with custom document nodes, you'll likely want to write your own custom commands as well.

Collaborative editing

Real-time collaborative editing allows multiple people to edit the same document at the same time. Changes they make are applied immediately to their local document, and then sent to peers, which merge in these changes automatically (without manual conflict resolution), so that editing can proceed uninterrupted, and the documents keep converging.

This guide describes how to wire up ProseMirror's collaborative editing functionality.

Algorithm

ProseMirror's collaborative editing system employs a central authority which determines in which order changes are applied. If two editors make changes concurrently, they will both go to this authority with their changes. The authority will accept the changes from one of them, and broadcast these changes to all editors. The other's changes will not be accepted, and when that editor receives new changes from the server, it'll have to rebase its local changes on top of those from the other editor, and try to submit them again.

The Authority

The role of the central authority is actually rather simple. It must...

- Track a current document version
- Accept changes from editors, and when these can be applied, add them to its list of changes
- Provide a way for editors to receive changes since a given version

Let's implement a trivial central authority that runs in the same JavaScript environment as the editors.

```
class Authority {
  constructor(doc) {
    this.doc = doc
    this.steps = []
    this.stepClientIDs = []
    this.onNewSteps = []
  }

  receiveSteps(version, steps, clientID) {
    if (version != this.steps.length) return

    // Apply and accumulate new steps
    steps.forEach(step => {
      this.doc = step.apply(this.doc).doc
      this.steps.push(step)
      this.stepClientIDs.push(clientID)
    })
    // Signal listeners
    this.onNewSteps.forEach(function(f) { f() })
  }

  stepsSince(version) {
    return {
      steps: this.steps.slice(version),
      clientIDs: this.stepClientIDs.slice(version)
    }
  }
}
```

When an editor wants to try and submit their changes to the authority, they can call `receiveSteps` on it, passing the last version number they received, along with the new changes they added, and their client ID (which is a way for them to later recognize which changes came from them).

When the steps are accepted, the client will notice because the authority notifies them that new steps are available, and then give them *their own* steps. In a real implementation, you could also have `receiveSteps` return a status, and immediately confirm the sent steps, as an optimization. But the mechanism used here is necessary to guarantee synchronization on unreliable connections, so you should always use it as the base case.

This implementation of an authority keeps an endlessly growing array of steps, the length of which denotes its current version.

The `collab` Module

The `collab` module exports a `collab` function which returns a plugin that takes care of tracking local changes, receiving remote changes, and indicating when something has to be sent to the central authority.

```
import {EditorState} from "prosemirror-state"
import {EditorView} from "prosemirror-view"
import {schema} from "prosemirror-schema-basic"
import collab from "prosemirror-collab"

function collabEditor(authority, place) {
  let view = new EditorView(place, {
    state: EditorState.create({
      doc: authority.doc,
      plugins: [collab.collab({version: authority.steps.length})]
    }),
    dispatchTransaction(transaction) {
      let newState = view.state.apply(transaction)
      view.updateState(newState)
      let sendable = collab.sendableSteps(newState)
      if (sendable)
        authority.receiveSteps(sendable.version, sendable.steps,
                               sendable.clientID)
    }
  })

  authority.onNewSteps.push(function() {
    let newData = authority.stepsSince(collab.getVersion(view.state))
    view.dispatch(
      collab.receiveTransaction(view.state, newData.steps, newData.clientID
  })

  return view
}
```

The `collabEditor` function creates an editor view that has the `collab` plugin loaded. Whenever the state is updated, it checks whether there is anything to send to the authority. If so, it sends it.

It also registers a function that the authority should call when new steps are available, and which creates a transaction that updates our local editor state to reflect those steps.

When a set of steps gets rejected by the authority, they will remain unconfirmed until, supposedly soon after, we receive new steps from the authority. After that happens, because the `onNewSteps` callback calls `dispatch`, which will call our `dispatchTransaction` function, the code will try to submit its changes again.

That's all there is to it. Of course, with asynchronous data channels (such as long polling in the collab demo or web sockets), you'll need somewhat more complicated communication and synchronization code. And you'll probably also want your authority to start throwing away steps at some point, so that its memory consumption doesn't grow without bound. But the general approach is fully described by this little example.