Even more hindsight on Vim, Helix and Kakoune

editors, productivity-platforms Wed May 24 11:50:00 2023 UTC, by Dimitri Sabadie — feed

Oh my… 2023 has been such a trek so far… If you have missed my previous article about my thoughts about editors and development platforms, I think it's probably the moment to have a look at it.

Today is the end of May 2023. Helix has been my primary editor for months now. I haven't come back to Neovim. In the previous article, I mentioned that I couldn't give a fair opinion about Helix because I had just started using it. Today, I think I have enough experience and usage (5 / 6 months) to share what I think about Helix, and go a little bit further, especially regarding software development in general and, of course, editing software.

However, before starting, I think I need to make a clear disclaimer. If you use Vim, Neovim, Emacs, VS Code, Sublime Text, whatever, and you think that *"Everyone should just use whatever they want"*, then we agree and **this is not the point of this blog article**. The goal is to discuss a little bit more than just speaking out obvious takes, but please do not start waving off the topic because you think *"Everyone should use what they want"*. There is a place for constructive debate even there. If you start arguing, then it means you want to debate, and then you need to be ready to have someone with different arguments that will not necessarily go your way, nor your favorite editor.

Finally, if you think I haven't used Vim / Neovim enough, just keep in mind that I have been using (notice the tense) Vim (and by extension, Neovim) since I was 15, and I'm 31, so 16 years.

I have wrote that blog article already three times before deleting everything and starting over. I think I will make another blog article about how I think about software, but here I want to stay focused on one topic: editors and development environments.

Helix

From a vimmer perspective, Helix is weird. It reverses the verb and the motion (you don't type `di(` to delete inside parenthesis, but you type `mi(d` to first match the parenthesis and then delete). Then, you have the multi-selections as a default way of doing things; Helix doesn't really have the concept of a *cursor*, which is a design it gets from Kakoune, and I'll explain what it means and implies later.

Some vimmers publicly talked about it. ThePrimeagen, for instance, made a Youtube video about it where he basically just scratched the surface of the editor and *“Hell it’s not Vim so it’s not really good”*. He moved the video to private then but I’m sure you can just look for Tweets. Many Vim afficionados react that way, which is not a very serious way of trying out something new, especially if you happen to publicly talk about it to thousands of people. I think it’s not fair to both the tool (here, Helix) and the people reading you, unless you are one of those Vim zealots thinking you know everything better than everyone else and dismissing people’ points of views just because they are not the same as yours.

Anyway, that reputation didn’t hold me from trying out, and the way I try software is simple: just give in and accept to drop your habits and productivity. Of course, at work, I was still using Vim / Neovim, but switched to Helix on my spare-time projects.

There are many aspects to talk about with Helix. The first one is that, contrary to what people who haven’t really and seriously tried it, the difference with Vim is not only “reversed motion/verb and multi-cursors.” The first difference is the *design* and the *direction*.

Design and direction

Helix is an editor that natively integrates many features that are most of the time plugins in others editors (even in VS Code). The best examples here are tree-sitter, surrounding pairs (being able to automatically surround regions of text with `(`, `[`, HTML tags, etc.), LSP UIs, fuzzy pickers, etc. This is a *massive* and important difference because of two main things:

1. Code maintenance.
2. User experience.

About code maintenance, having all of those features natively integrated in the editor means that you are 100% sure that if you get the editor to start, the features will work, and the editor developers will keep that updated within the next iteration of the editor. For instance, having tree-sitter natively implemented (Rust) in Helix means that the editor itself knows about tree-sitter and its grammars, highlights queries and features. The same thing goes for surrounding-pairs or auto-pairs, for instance. If the team decides to change the way text is handled in buffers, then the code for auto-pairs / surrounding-pairs **will have to be updated for the editor to be releasable**.

The user experience will then be better, because you get those nice features without having to start looking around for a plugin doing it, with the problem of chosing the right one among a set of competing plugins. Plus the risk of having your plugin break because it’s not written by the Helix team. Just install the software and start using those features.

For now, Helix ships with a bunch of powerful features that makes it usable in a modern environment:

- tree-sitter support for semantic highligthing, semantic selection and navigation, etc. That also includes a native support for getting grammars / queries from the CLI with `hx --grammar`.
- LSP support, both in terms of features (go-to, references, implementations, incoming / outgoing calls, diagnostics, inlay hints, etc. etc.) and in terms of UI. It’s the same UI for everyone.
- DAP support for debugging (experimental at the time of writing).
- Surrounding pairs.
- Auto pairs.
- Git integration (gutter diff on the right side of your buffer) and Git semantic object (go to next change, etc.).

- Registers and user-defined registers (like in Vim), along with macros (but you won't need them, trust me).
- Native discoverability, including a command palette with *every possible available commands*, tagged with their keybindings.
- Many bundled themes (yes, `catppuccin` themes are there!).
- Snappier than anything else.
- Various integration with your system, including clipboards, external commands, etc.

There is also one (major) thing I want to talk about and that deserves its own section: configuration.

Configuration done right

Helix treats configuration *the way it should be*: as data. Configuration in Helix is done in TOML. There is no scripting implied, it's only a rich object laid out as TOML sections. And this is a *joy* to use. For instance, this is my current Helix configuration (excluding keys remapping, because my keyboard layout is bépo):

```
theme = "catppuccin_macchiato"

[editor]
scroll-lines = 1
cursorline = true
auto-save = false
completion-trigger-len = 1
true-color = true
color-modes = true
auto-pairs = true
rulers = [120]
idle-timeout = 50

[editor.cursor-shape]
insert = "bar"
normal = "block"
select = "underline"

[editor.indent-guides]
render = true
character = "▏"

[editor.lsp]
display-messages = true
display-inlay-hints = true

[editor.statusline]
left = ["mode", "spinner", "file-name", "file-type", "total-line-numbers", "file-encoding"]
center = []
right = ["selections", "primary-selection-length", "position", "position-percentage", "spacer",
```

Notice the tree-sitter and LSP configuration. Yes. *None*.

This is so important to me. Because configuration is data, it is simple for Helix to expose it and present it to the user by reading the TOML file without caring about having any side-effects. Helix has a command line (`:`) where you

can tweak those options dynamically. And you can dynamically reload the configuration as well.

Multi-selection centric

The major difference, even before the reversed motion/verb thing, is the fact that Helix *doesn't really* have a cursor. It has the concept of *selections*. A selection is made of two entities:

- An *anchor*.
- A *cursor*.

The cursor is the part of the selection that moves when you extend the selection. The anchor, as the name implies, is the other part that stays where it is: it's anchored. **By default, you have only one selection** and the anchor is located at the same place as the cursor. It looks similar to any other editor. Things start to change when you begin typing normal commands. Typing `l`, for instance, will move both the anchor and cursor to the right, making them a single visual entity. However, if you type `w`, the cursor will move to the end of the word while the anchor will move to its beginning, visually selecting the word. If you type `W`, the anchor won't move and only the cursor will move, extending the selection. If you press `B`, it will move the cursor back one word, shrinking the selection. You can press `<a-;>` to flip the anchor and the selection, which is useful when you want to extend on the left or on the right.

This concept of selection is really powerful because everything else is based on it. Pressing `J` will move the cursor down one line, leaving the anchor on the current line, extending selected lines. Once something is selected, you can operate on it, with `d`, `c`, `y`, `r`, etc. For instance, `wd` will select the word and delete it. `Jc` will extend the selection with the next line and start changing. Selections in Helix are not just visual helps: they represent what normal editing operations will work on, which is a great design, because you can extend, shrink and reason about them in a much more seamless and natural way.

But it's just starting. Remember earlier when I say that **by default, you have only one selection**? Well, you can have many, and this is where Helix starts to really shine to me. The first way to create many selections is to press `C`. `C` will duplicate your current selection on the next line. If you have the anchor at the same position as the cursor, pressing `C` will make it like you have another cursor on the next line below. For instance, consider this text:

```
I love su|shi.
But I also love pizza.
```

The `|` is our cursor (but also anchor). If you press `C`, you will see something like this:

```
I love su|shi.
But I als|o love pizza.
```

But as I had mentioned, `C` duplicates *selections*. Let's say you started like this, `<` being the anchor and `|` the cursor:

```
I love <sus|hi.
But I also love pizza.
```

Pressing `C` now will do this:

```
I love <sus|hi.
But I a<lso| love pizza.
```

Once you have many selections, everything you type as normal commands will be applied to every selections. If I type `f.`, it will set the anchor to the current cursor and extend the cursor to the next `.`, resulting in this:

```
I love sus<hi|.
But I also< love pizza|.
```

Press `<a-;>` to swap anchors and cursors:

```
I love sus|hi<.
But I also| love pizza<.
```

And then pressing `B` will do this:

```
I love |sushi<.
But I |also love pizza<.
```

> Erratum: `B` is not defined to this behavior in a vanilla Helix. I have remapped it to the Kakoune behavior. It doesn't change much to what I'm saying here, though.

Multi-cursor is then not only a nice visual help, but also a completely new way of editing your buffers. Once you get the hang of it, you don't really think in terms of a single cursor but many selections, ranges, however you like to call them.

Getting more cursors

`C` is great, but this not something we usually use. Instead, we use features that don't exist in Vim. I'm not entirely sure how to call those, but I like to call them *selection generators*. They come in many different flavors, so I'll start with the easiest one and will finish with the most interesting and (maybe a bit?) obscure at first.

Matching matching matching!

The `m` key is Helix is a wonderful key. It's the *match* key. It expects a motion and will change all your selections to match the motion. For instance, `mia` *"matches inside arguments"* (tree-sitter). Imagine this context:

```
fn foo(x: i32, y: |i32) {}

fn bar(a: String, |b: bool) {}
```

Press `mia` to get this:

```
fn foo(x: i32, <y: i32|) {}

fn bar(a: String, <b: bool|) {}
```

Then, for instance, press `<a-;>` to flip the anchor and the cursor:

```
fn foo(x: i32, |y: i32<) {}

fn bar(a: String, |b: bool<) {}
```

`F,` to select the previous `,`:

```
fn foo(x: i32|, y: i32<) {}

fn bar(a: String|, b: bool<) {}
```

> Erratum: same thing as with `B`; I have remapped it in my config. The default `B` doesn't extend like this.

And just press `d`:

```
fn foo(x: i32) {}

fn bar(a: String) {}
```

It's so logical, easy to think about and natural. Someting interesting to notice, too, is that contrary to Vim, which has many keys doing mainly the same thing, making things weird and not really well designed. For instance, `vd` selects the current character and delete it. `vc` deletes the current character and puts you into insert mode. `s` does exactly the same. Why would you have a key in direct access doing something so specific? If you want to delete a word, you either press `vwd`, or more simply in Vim, `dw`. All of that is already confusing, but it doesn't end there. `x` deletes the current character, but `x` is actually *cut*, so if you select a line with `V` and press `x`, it will cut the line. Press `Vc` to change a line… or just `S`. What?

All those shortcuts feel like exceptions you have to learn, and it's a good example of a flawed design. On the other side, Helix (which is actually a Kakoune design it's based on), have a single character to delete something: `d`. Since the editor has multi-selections as a native editing entity, all of those situations will imply using the `d` key:

- Deleting the current character: `d`.
- Deleting the next word: `wd`.
- Deleting the current line: `xd` (`x` selects and extend the line).
- Deleting delimiters but not the content: `md(`.
- Etc. etc.

That applies to everything.

Selecting, splitting, keeping and removing

The design of editing in Helix (Kakoune) is to be interactive and iterative. For instance, consider the following:

```
  pub fn new(
    line_start: usize,
    col_start: usize,
    line_end: usize,
    col_end: usize,
    face: impl Into<String>,
  ) -> Self {
    Self {
      line_start,
      col_start,
      line_end,|
      col_end,
      face: face.into(),
    }
  }
```

Let's say we would like, to begin with, select very quickly every arguments type and switch them to `i32`. Many ways of doing that, but let's see one introducing a great concept: *selecting*. Selecting allows you to create new selections that satisfy a regex. The default keybinding for that is `s` for… select (woah). `s` always applies to the current Selections (notice the use of plural, it will be useful later). We then need to start selecting something. Here, we can just press `mip` to select inside the paragraph, since our cursor is right after `line_end`:

```
< pub fn new(
    line_start: usize,
    col_start: usize,
    line_end: usize,
    col_end: usize,
    face: impl Into<String>,
  ) -> Self {
    Self {
      line_start,
      col_start,
      line_end,
      col_end,
      face: face.into(),
    }
  }|
```

We have the whole thing selected. Press `s` to start selecting with a regex. We want the arguments, so let's select everything with `:` and press `s:` and return:

```
  pub fn new(
    line_start<:| usize,
    col_start<:| usize,
    line_end<:| usize,
    col_end<:| usize,
    face<:| impl Into<String>,
  ) -> Self {
```

```
    Self {
      line_start,
      col_start,
      line_end,
      col_end,
      face<:| face.into(),
    }
  }
```

See how it created a bunch of selections for us. Also, notice that it selected `face: face.into()`, which is not correct. We want to *remove* that selection. Again, several ways of doing it. Something to know is that, Helix (Kakoune) has the concept of *primary* selection. This is basically the selection on which you are going to apply actions first, like LSP hover, etc (it would be a madness to have LSP hover applies to all selections otherwise!). You can cycle the primary selection with `(` and `)`. Once you reach the one you want, you can press `<a-,>` to just drop the selection. However, we don't want to cycle things. We want a faster way.

Let's talk about *removing* selections. The default keybinding is `<A-K>`. However, here, our selections are all about the same content (the `:`). As mentioned before, pressing `x` will select the current line of every selections:

```
  pub fn new(
<   line_start: usize,|
<   col_start: usize,|
<   line_end: usize,|
<   col_end: usize,|
<   face: impl Into<String>,|
  ) -> Self {
    Self {
      line_start,
      col_start,
      line_end,
      col_end,
<     face: face.into(),|
    }
  }
```

Let's filter selection and remove the one matching a pattern. In our case, let's remove selections with a `(` in them: `<A-K>\(` and return:

```
  pub fn new(
<   line_start: usize,|
<   col_start: usize,|
<   line_end: usize,|
<   col_end: usize,|
<   face: impl Into<String>,|
  ) -> Self {
    Self {
      line_start,
      col_start,
      line_end,
      col_end,
      face: face.into(),
```

```
    }
  }
```

Now press `_` to shrink the selections to trim leading and trailing whitespaces:

```
  pub fn new(
    <line_start: usize,|
    <col_start: usize,|
    <line_end: usize,|
    <col_end: usize,|
    <face: impl Into<String>,|
  ) -> Self {
    Self {
      line_start,
      col_start,
      line_end,
      col_end,
      face: face.into(),
    }
  }
```

Imagine that we have changed our mind and now we actually want to change the `usize` to `i32`. We can use the *keep* operator, which is bound to `K` by default. Press `Kusize` and return to get this:

```
  pub fn new(
    <line_start: usize,|
    <col_start: usize,|
    <line_end: usize,|
    <col_end: usize,|
    face: impl Into<String>,
  ) -> Self {
    Self {
      line_start,
      col_start,
      line_end,
      col_end,
      face: face.into(),
    }
  }
```

Another possible way is to press `susize` to only select the `usize` directly, which might be wanted if you want to change them quickly to `i32`, for instance.

The last operation that I want to mention is *splitting*. It's introduced with `S` and will spawn several cursors separated by a regex. For instance, consider:

```
let |array = [1.0, 20.32, 3., 4.35];
```

Let's say you'd like to select the numbers. With the methods described above, it's probably challenging. With the splitting command, it's much easier. Put your cursor anywhere in the list and press `mi[` to select inside of it:

```
let array = [<1.0, 20.32, 3., 4.35|];
```

Then simply press `S,` to split the selection into selections separated by commas. You should end up with this:

```
let array = [|1.0>,| 20.32>,| 3.>,| 4.35>];
```

Pressing `_` will shrink the selections to remove leading and trailing spaces:

```
let array = [|1.0>, |20.32>, |3.>, |4.35>];
```

And here you have it! Now remember that you can combine all of this methods with semantic objects, like `mif` for *inside functions*, `mat` for *around type*, `mia` for *inside arguments*, and many more. Recall that you can do that *on each selection*, allowing for really powerful workflows.

Do I really use all that at work / spare time projects?

Hell yes. It's a muscular memory thing. For instance, I oftentimes the need to not only replace occurrences of patterns, like `fooN` — with `N` being a number — into `logger.fooN`, **but I often need to change the structure around those occurrences.** And here, Helix really stands out. In Vim, you'd have to use a super ugly regex, completely blind, and eventually a macro. The interactive and iterative approach of Helix is so much more powerful to me. For instance, for the case described above: `%` to select the whole buffer, `sfoo.` to select `foo` with a single character afterwards, then return, `clogger.foo` to replace with `logger.foo`, and still in insert mode, `<C-r>"` to paste what was yanked by the `c` operation. Here, the default register, `"`, makes a *lot* of sense, because this register is local to each selection, making this replace operation trivial and interactive.

Another example is something like this:

```
const COLORS: [Color; 3] = [
  Color {
    r: 255,
    g: 0,
    b: 0,
  },
  Color {
    r: 0,
    g: 255,
    b: 0,
  },
  Color {
    r: 0,
    g: 0,
    b: 255,
  },
];
```

Imagine that you want to change every `Color` constructor to a function call, that does something like `Color::rgb(r, g, b)`. Doing that interactively and iteratively in Helix is so easy. I'd put my cursor anywhere in

that block, press `mi[` to select everything inside `[]`, then `sColor<cr>` to create three cursors on the `Cursor`, and from that moment, it's just ping-pong-ing between normal mode and insert mode like you would do with a single selection. You will be using `f`, `t`, `miw` etc. select things but the idea is the same, and the three occurrences will be updated at once.

A simpler editor overall

Contrary to other famous editors and IDEs, Helix is not supposed to be extendable; it doesn't try to solve more problems than it should (and you will see in the Kakoune section that we can even push that to another extreme). Something like Neovim is a bit of a disguised IDE. Yes, a vanilla Neovim with no plugins and no configuration is just a very basic (and I would dare say *featureless*) editor. It won't have LSP working. It won't have tree-sitter working either. Nothing for git integrated. Nothing for delimiters, nothing for pickers, nothing for any modern development. The power of something like Emacs, Neovim etc. is to build on *extensibility*.

I used to enjoy that, until I came to the realization that, *perhaps*, it would be great to put things into perspective: is extensibility something we actually want? What do we try to solve with it? Well, we extend tools to add new features and new behaviors. We extend things so that the native / core tool ships with minimal features but doesn't prevent people from adding specific and customized capabilities.

Extensibility

But is extensibility the only way to achieve that goal? The thing with extensibility is that:

1. You have to build in advance for it. You cannot ship an editor without any extensibility support and expect it to be an emergent feature. You have to add a basic support for it, whether it's a plugin system, a dynamic / relocatable system (`.so` / `.dylib` / `.dll`), a scripting language, a JIT, etc.
2. Extensibility is always walled.

The last point is important. Extending a software **requires the environment to adapt to the specifities of what you're extending**, and that will require the environment to know about the specifities. Here, the environment could be anything **outside of Neovim**. What it means is that, you're not going to use external tools, but you are going to use the interfaces, scripting languages, DSLs etc. of the tool you want to extend.

For instance, people might argue that extending Neovim is great because it only requires learning Lua, which is not specific to Neovim, but that it is not actually true nor acurate. You have to learn *"Neovim Lua"*, which is basically its own beast. It's like the standard library, but for Neovim. It will provide you with APIs you can use — and only use — to add new features to Neovim.

The same argument can be made to *any extensible* editor. VS Code, Emacs, etc.

Composability

Another way to add features and behaviors is to add them *externally*, by leveraging the tools themselves and compose them instead of pushing more features into them. That vision is not very popular and famous and I'm not entirely sure why. For instance, Vim has vim-fugitive, a Git client for Vim / Neovim. It has 2k commits, between 8k-9k lines of code… and can be used only in Vim and Neovim. Yes, it extends and adds features *inside* those editors, but still. If at some point you decide to switch to another editor, you can forget about this plugin. This is even worse with

something like magit, which is the best Git client I haver ever used. Yet I don't use it anymore, because it's an Emacs plugin. What a shame.

Now repeat that reasoning for all the plugins you use. That has led some people to just install as few plugins as possible and switch to composability.

Composability is the same concept as in code. You have two systems `A` and `B` doing `thingA` and `thingB`, and you write some *piping* glue code / script to connect both. People use many different languages to glue things together. Among the most famous approaches:

- Python.
- Perl, especially when dealing with filtering text.
- The shell, and especially any POSIX.2 compliant shell (no, don't start talking about `fish`).

I have shell functions that I source when I start my shell; for instance, one called `p`, to switch to my spare-time projects:

```
p () {
  proj_dir=${PROJ_DIR:-~/dev}
  project=$(ls $proj_dir | sk --prompt "Switch to project: ")
  [ -n "$project" ] && cd $proj_dir/$project
}
```

This is a great example of composability, which composes the content of a project directory (the `$PROJ_DIR` environment variable, or `~/dev` by default), with the `sk` fuzzy finder, and then changes the directory to whatever the user has picked. I use that script all the time to quickly move to my various projects.

Notice that, `sk`, `fzf`, etc. already are tools that implement fuzzy searching for arbitrary inputs. Tools such as `find` or `fd`, who are so far in my experience the fastest programs to look for stuff, can be composed with shell pipes as well and integrated into your work environment.

Then the question starts to appear: why do editors / plugins re-implement all of that to have it inside the editor? It's pretty apparent in the Neovim community, but it often ends up with abandonware plugins, and harder to maintain editors.

While I was wondering about all that, I was getting pretty productive with Helix, enjoying its simpler design, data configuration, better editing features and overall way more stable experience. I remembered that most of the Helix design came from Kakoune. And I started to think about one (not so) crazy idea: should I have a look at Kakoune?

And let's enter Kakoune.

Kakoune

As mentioned above, Helix is *heavily* inspired by Kakoune. The main difference, from the surface, with distance, is that Helix comes with more bundled features, like LSP, tree-sitter, pickers, etc. However, there are more (drastic) differences that I need to talk about.

The first thing is, again, the design. And for that, I really need to quote a part from the excellent design doc written by @mawww. The part that is the most interesting to me is, obviously, *composability*.

> Being limited in scope to code editing should not isolate Kakoune from its environment. On the contrary, Kakoune is expected to run on a Unix-like system alongside a lot of text-based tools, and should make it easy to interact with these tools.
>
> For example, sorting lines should be done using the Unix sort command, not with an internal implementation. Kakoune should make it easy to do that, hence the `|` command for piping selected text through a filter. The modern Unix environment is not limited to text filters. Most people use a graphical interface nowadays, and Kakoune should be able to take advantage of that without hindering text mode support. For example, Kakoune enables multiple windows by supporting many clients on the same editing session, not by reimplementing tiling and tabbing. Those responsibilities are left to the system window manager.

And this is one of the most important things about Kakoune to me. First, it goes into the direction I've been wanting for years (I'll let you read my previous blog articles about editors and production environments; I'm basically saying the same thing). And second, it ensures that the native editor remains small and true to its scope, ensuring an easier maintenance, hence less bugs and more stable. Also, it doesn't blindly ignore what everyone else is doing.

Let's start with an example. Yes, Kakoune doesn't have a fuzzy picker to pick your files. However, as mentioned above, it composes well with its environment. It does that via different mechanisms (shell blocks, FIFOs, UNIX sockets, etc.). Here, we can just use whatever we like to get a list of files, and let Kakoune ask the user which files to open. We then simply use the selected value and open it. In order to do that, you need to read the design doc to understand a couple of other things, such as the section about *interactive use and scripting*. Quoting:

> As an effect of both Orthogonality and Simplicity, normal mode is not a layer of keys bound to a text editing language layer. Normal mode **is** the text editing language.

Typing normal commands in a `.kak` file is then the way to go. And then, coming back to the fuzzy picker example, here are three commands defined in my `kakrc`:

```
## Some pickers
define-command -hidden open_buffer_picker %{
  prompt buffer: -menu -buffer-completion %{
    buffer %val{text}
  }
}

define-command -hidden open_file_picker %{
  prompt file: -menu -shell-script-candidates 'fd --type=file' %{
    edit -existing %val{text}
  }
}

define-command -hidden open_rg_picker %{
  prompt search: %{
    prompt refine: -menu -shell-script-candidates "rg -in '%val{text}'" %{
      eval "edit -existing  %sh{(cut -d ' ' -f 1 | tr ':' ' ' ) <<< $kak_text}"
    }
```

```
  }
}
```

As you can see, it's just about composing known and well written tools together. Another example? Alright. Kakoune doesn't have splits, but I still want them. Let's go:

```
## kitty integration
define-command -hidden kitty-split -params 1 -docstring 'split the current window according to t
  kitty @ launch --no-response --location $1 kak -c $kak_session
}

## zellij integration
define-command -hidden zellij-split -params 1 -docstring 'split (down / right)' %sh{
  zellij action new-pane -cd $1 -- kak -c $kak_session
}

define-command -hidden zellij-move-pane -params 1 -docstring 'move to pane' %sh{
  zellij action move-focus $1
}

## tmux integration
define-command tmux-split -params 1 -docstring 'split (down / right)' %sh{
  tmux split-window $1 kak -c $kak_session
}

define-command tmux-select-pane -params 1 -docstring 'select pane' %sh{
  tmux select-pane $1
}
```

The design is not extensible: it's composable, and all in all, it makes so much more sense to me.

Kakoune vs. the rest

If you are used to Helix, then Kakoune with a bit of configuration will feel very similar to Helix. Of course, you will have to look around for LSP and tree-sitter support. The way we do that is by adding external processes to interact with Kakoune servers / clients via UNIX sockets, FIFO pipes, etc.. Kakoune doesn't know anything about LSP or tree-sitter, but you can write a binary in any language you want and send remote commands to control the behavior of Kakoune.

- For LSP: kak-lsp/kak-lsp
- For tree-sitter: phaazon/kak-tree-sitter (I'm stil working on it and it lacks documentation but is usable).

The interesting aspect with those tools is that, in *theory*, we could adapt them to make them editor independent. If more editors adopted the strategy of Kakoune (composing via the shell), we wouldn't even have to write other binaries to add LSP / tree-sitter support, which is an interesting aspect.

I plan on writing a blog article detailing the design of `kak-tree-sitter`, because I think it's a good source of knowledge regarding UNIX and tree-sitter.

Besides that, Kakoune is way more mature than Helix, in the sense that it has some specificities to some edge cases with multi-selection features (such as, what happens when you have multiple cursors inside text looking like function argument lists, and you type `mia` to select them, but some selections are actually not arguments? Kakoune will remove the mismatched selections, which is what we would expect, while Helix…… erm it's complicated! but currently, it will keep the selections around, which is confusing and dangerous).

Kakoune has a *wonderful* feature called **marks**. Marks are different from what you have in Vim. They use a specific register to record the current selections and eventually restore them later, supporting merging selections and editing commands. An example that I love doing; imagine the following snippet:

```
Thank you to [@NAME](https://github.com/NAME) for their contribution.
```

Let's say you want to have a cursor on each `NAME`. Easy, with `s`. You just select the whole thing (you can select the whole line with `x` for instance), then `sNAME`, return, then `c` to start changing with whatever you want.

Now, imagine this instead:

```
Thank you to [@](https://github.com/) for their contribution.
```

How do you insert at the same time at the right of `@` and before the `)`? Getting two selections will be hard (or you will end up writing crazy regexes; remember, we are not using Vim anymore!). A super easy way to do it is to move your selection to `@`:

```
Thank you to [@|](https://github.com/) for their contribution.
```

Press `Z` to register the current selection (you have only one). Then move the cursor to the `)`:

```
Thank you to [@](https://github.com/|) for their contribution.
```

And then press `<a-z>a` to merge the current selection to the one(s) already stored. You end up with this:

```
Thank you to [@|](https://github.com/|) for their contribution.
```

Two cursors at the right places! This is extremely powerful and a feature that should arrive in Helix, but not sure exactly when.

A pretty other important thing to say about Kakoune is that it has a server/client design that allows to share session contexts. That is the main mechanisms used to implement native splits (via Kitty, tmux, whatever), but also many other features, such as project isolation, viewing the same buffers with different highlighters, etc. etc.

…but it's not perfect

There is one important thing I need to mention. I have been playing with Kakoune for a while now, and I have been working on `kak-tree-sitter` for almost as long as I've been working Kakoune (couple of months). And there is one issue with the UNIX approach.

See, tree-sitter, LSP, DAP, git gutters, etc. All those things are pretty *fundamental* to a modern text editor. Externalizing them (Kakoune) is an interesting take, but I'm not entirely sure Kakoune is still doing it completely correctly.

The main problem is *social intelligence*. Because all tooling is now externalized, many people can come up with their own efforts. For instance, `kak-lsp` and `kak-tree-sitter` are completely separate projects, and they should remain that way (for many reasons; scopes, maintenance, dependencies, etc.). However, in order to operate on the editor contents, both programs must interact with the editor. That implies:

- Retrieving buffer contents.
- Performing some actions that can mutate (and overlap / hijack) each other.
- Insert hooks that might be incompatible between them.

This problem is important, because dumping the whole content of a big buffer to an external process is one thing, doing it for every different external processes is a massive overhead. Because I have read a bit the source code of `kak-lsp`, I know that we (`kak-tree-sitter`) are doing similar things: we do use FIFOs to write buffer content and communicate with our servers / daemons without going through the disk. But we are doing it that *twice*. And it's just two projects; any dissociate projects needing to access the buffer content will probably perform similar things.

That is a massive problem to me and I'm not sure how I feel about it. I'm not against sending the content of a buffer via a FIFO to an externalized program — I actually think it's a pretty good design —, but doing it for every integration… I'm not exactly sure what would be the best solution, but maybe something that would snapshot a buffer inside some POSIX shared memory (with mutable lock access if needed) could be one way to go. Honestly, I am not sure.

All of that to say that, the take of Helix is pretty good here, because all of those UNIX problems are not there in that editor: everything runs in the same process, inside the same memory region. I will come back to this problem with my next article on `kak-tree-sitter` and its design.

Conclusion

Today, I'm mainly using both Kakoune and Helix. Helix still has some bugs, even with tree-sitter (which my daemon doesn't have, funnily!), so I sometimes use one or the other tool.

I have learned so many things lately, with both Helix and Kakoune (especially Kakoune, it made me love UNIX even more). All of that echoes the disclaimer I made earlier: yes, Vim and Neovim are good, but Kakoune and Helix are so much better to me. Better designs, better editing experiences, largely snappier, more confidence in the direction of the native code (because a much, much smaller codebase). Helix is written in Rust, Kakoune in C++, but what matters is the actual design. To me, Kakoune is by far the best designed software I have ever seen, and for that, I really admire the work of @mawww and everyone else involved in the project. My contribution to the Kakoune world with kak-tree-sitter is, I hope, something that will help and drive more people in. I will write another blog article about that, with the pros., cons. and tradeoffs. of composability in editors.

In the meantime, have fun and use the editor you love, but remember to have a look around and stay open to change! Keep the vibes!

> I would like to thank @Taupiqueur, who played an important role into making me undersand Kakoune and

eventually fall in love with — the editor, I mean! :)