Ownership and lifetimes

How Rust's unique features will help us develop a stable, fast and multi-threaded desktop app

PLM - Paradigm analysis report

Table of Contents

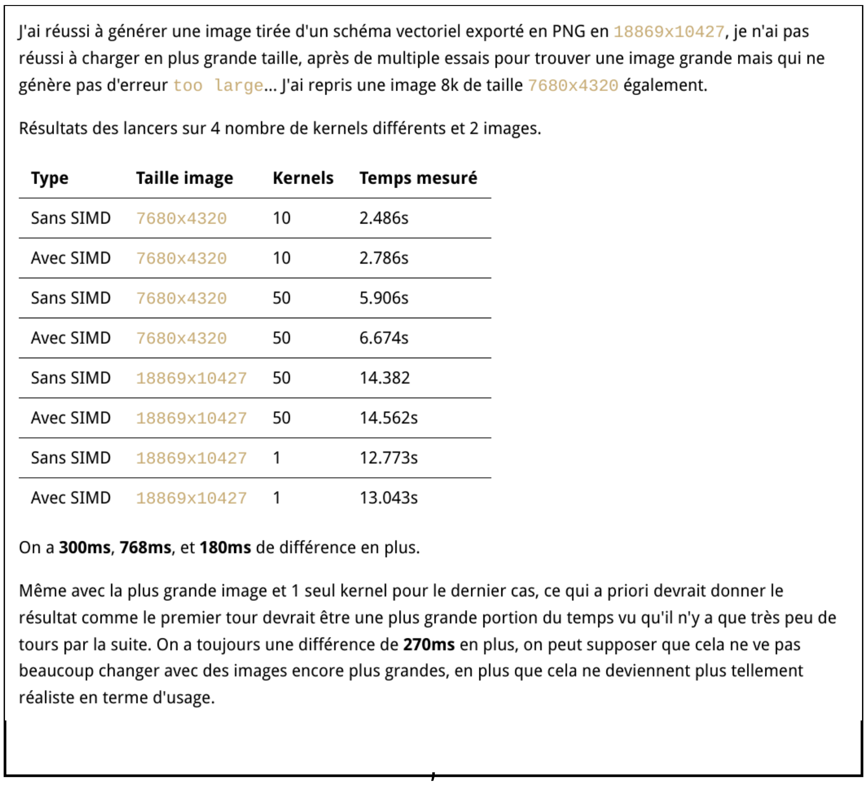
The problem with Markdown

Markdown has the double advantage of being an easy and useful syntax to learn, and has a large support in numerous chat systems and collaboration tools. Redacting Markdown in VSCode or in a web editor is working fine. GitHub and Gitlab show it natively, the editing experience is good, the problems are elsewhere…

Export

First, how do you export your document as PDF if you want to send it to someone that needs this format ?

1. You may use `pandoc` that convert it into Latex and then generate a PDF, but the default style is really not the same as the preview visible in VSCode. In addition, changing the style requires some Latex knowledge. This would imply a lot more learning and setup to change a few visual settings.
2. You could use an easier solution, the `Markdown PDF` extension in VSCode. It provides a simple `Export to PDF` action in the command palette. But what happens if you don't like the default theme ? You have to write pure CSS to change the style of headings, tables, line height, to remove this strange yellow, etc.



J'ai réussi à générer une image tirée d'un schéma vectoriel exporté en PNG en `18869x10427`, je n'ai pas réussi à charger en plus grande taille, après de multiple essais pour trouver une image grande mais qui ne génère pas d'erreur `too large`... J'ai repris une image 8k de taille `7680x4320` également.

Résultats des lancers sur 4 nombre de kernels différents et 2 images.

Type	Taille image	Kernels	Temps mesuré
Sans SIMD	`7680x4320`	10	2.486s
Avec SIMD	`7680x4320`	10	2.786s
Sans SIMD	`7680x4320`	50	5.906s
Avec SIMD	`7680x4320`	50	6.674s
Sans SIMD	`18869x10427`	50	14.382
Avec SIMD	`18869x10427`	50	14.562s
Sans SIMD	`18869x10427`	1	12.773s
Avec SIMD	`18869x10427`	1	13.043s

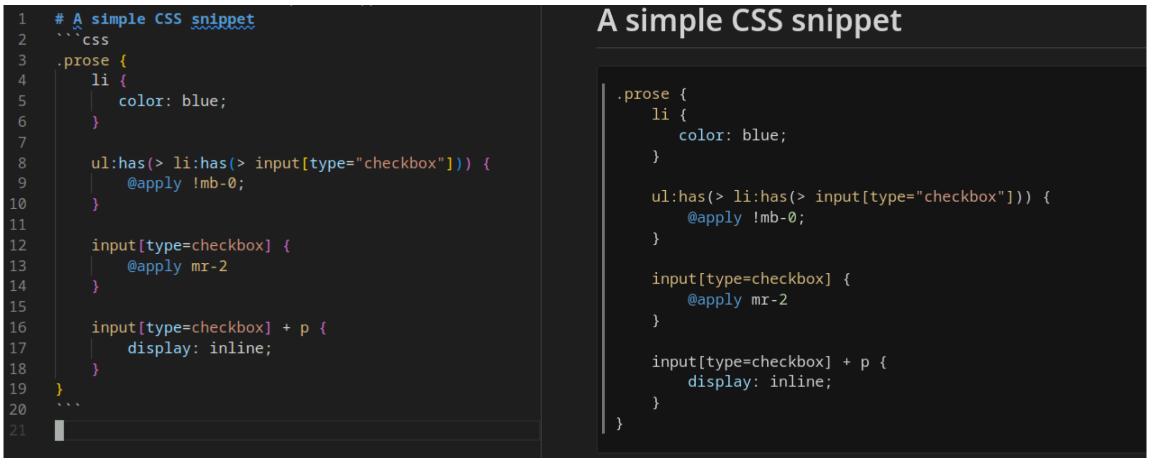
On a **300ms**, **768ms**, et **180ms** de différence en plus.

Même avec la plus grande image et 1 seul kernel pour le dernier cas, ce qui a priori devrait donner le résultat comme le premier tour devrait être une plus grande portion du temps vu qu'il n'y a que très peu de tours par la suite. On a toujours une différence de **270ms** en plus, on peut supposer que cela ne ve pas beaucoup changer avec des images encore plus grandes, en plus que cela ne deviennent plus tellement réaliste en terme d'usage.

An example of the default style of PDF documents generated by `Markdown PDF`

Syntax highlighting

What about syntax highlighting ? When you integrate code snippets in your reports, you want them to look good, or at least like in your IDE. In VSCode, here is how it would look the editor at left and the native preview at right. As the `@apply` from TailwindCSS is not integrated in the regex system of `highlight.js`, it breaks the style of other parts (see the last block). In the second block, the `ul` and `has` are different types and should not be colored in the same way.

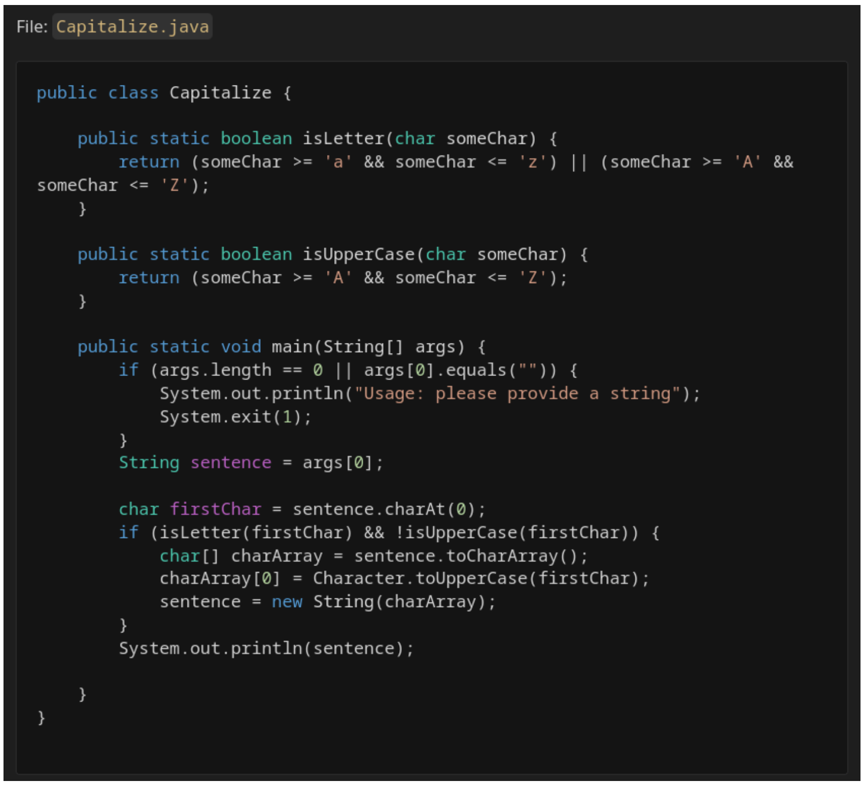


~~~
# A simple CSS snippet
```css
.prose {
    li {
        color: blue;
    }

    ul:has(> li:has(> input[type="checkbox"])) {
        @apply !mb-0;
    }

    input[type=checkbox] {
        @apply mr-2
    }

    input[type=checkbox] + p {
        display: inline;
    }
}
```
~~~

A simple CSS snippet

```css
.prose {
    li {
        color: blue;
    }

    ul:has(> li:has(> input[type="checkbox"])) {
        @apply !mb-0;
    }

    input[type=checkbox] {
        @apply mr-2
    }

    input[type=checkbox] + p {
        display: inline;
    }
}
```

The code preview in VSCode using the library `highlight.js`



File: `Capitalize.java`

```java
public class Capitalize {

    public static boolean isLetter(char someChar) {
        return (someChar >= 'a' && someChar <= 'z') || (someChar >= 'A' &&
someChar <= 'Z');
    }

    public static boolean isUpperCase(char someChar) {
        return (someChar >= 'A' && someChar <= 'Z');
    }

    public static void main(String[] args) {
        if (args.length == 0 || args[0].equals("")) {
            System.out.println("Usage: please provide a string");
            System.exit(1);
        }
        String sentence = args[0];

        char firstChar = sentence.charAt(0);
        if (isLetter(firstChar) && !isUpperCase(firstChar)) {
            char[] charArray = sentence.toCharArray();
            charArray[0] = Character.toUpperCase(firstChar);
            sentence = new String(charArray);
        }
        System.out.println(sentence);

    }
}
```

Another preview of Java code in VSCode. Lots of parts in white are not colorized.

Search

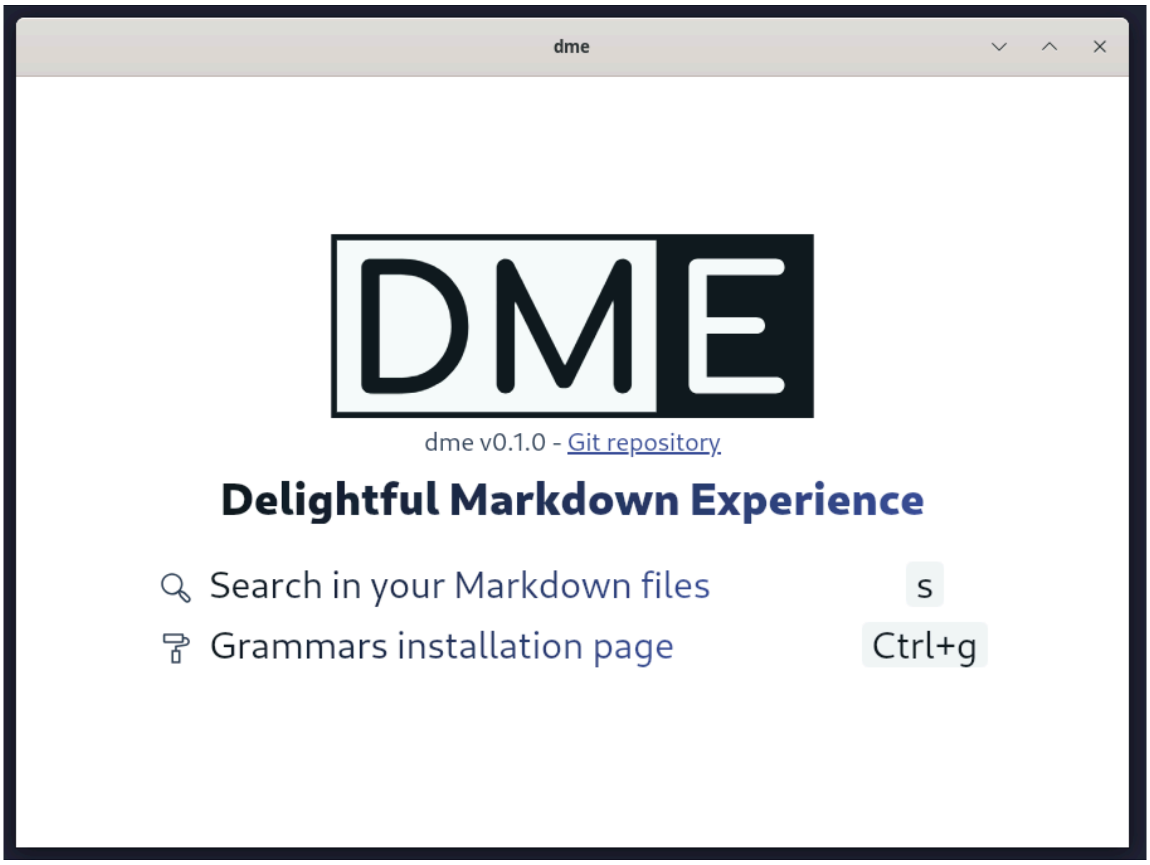
Now, let's say, you don't really remember where you put a very specific note on your disk about some "keyboard shortcuts for my terminal". You have hundreds of notes everywhere for numerous projects. You try to run `grep -r 'keyboard shortcuts for my terminal'` to find all possible matches on all files on your disk. The problem with this approach is that no exact match will be found, because `grep` only find exact matches. We would prefer to have approximative matching. You could search by file path with the `find` command on all files of your disk, and filter by a single keyword, like `find shortcuts`. You'll probably get hundreds of unrelated results or just no result because the filename is too different.

Why DME is the solution ?

Note: the README.md at root contains instructions on how to install DME if you want to follow along.

DME is trying to fix these issues around the search and syntax highlighting experience. We didn't have time to tackle PDF export this semester, but we successfully improved the situation on search and code preview. The PDF style will be the same as the preview style.

When you open DME, either with the `dme` command or via the start menu, you can see the home view.



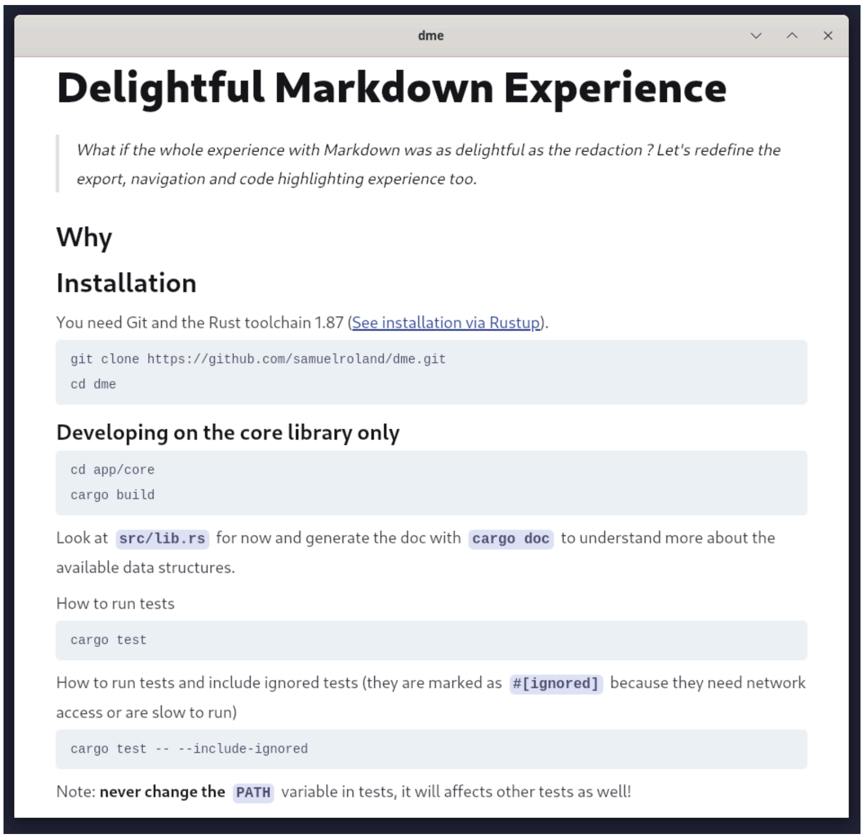
dme v0.1.0 - Git repository

Delightful Markdown Experience

Search in your Markdown files s
Grammars installation page Ctrl+g

The home page of DME, when started without a file as argument

If you opened with a file as the first argument `dme README.md` it would open it and show the preview.



Delightful Markdown Experience

> *What if the whole experience with Markdown was as delightful as the redaction ? Let's redefine the export, navigation and code highlighting experience too.*

Why

Installation

You need Git and the Rust toolchain 1.87 (See installation via Rustup).

```
git clone https://github.com/samuelroland/dme.git
cd dme
```

Developing on the core library only

```
cd app/core
cargo build
```

Look at `src/lib.rs` for now and generate the doc with `cargo doc` to understand more about the available data structures.

How to run tests

```
cargo test
```

How to run tests and include ignored tests (they are marked as `#[ignored]` because they need network access or are slow to run)

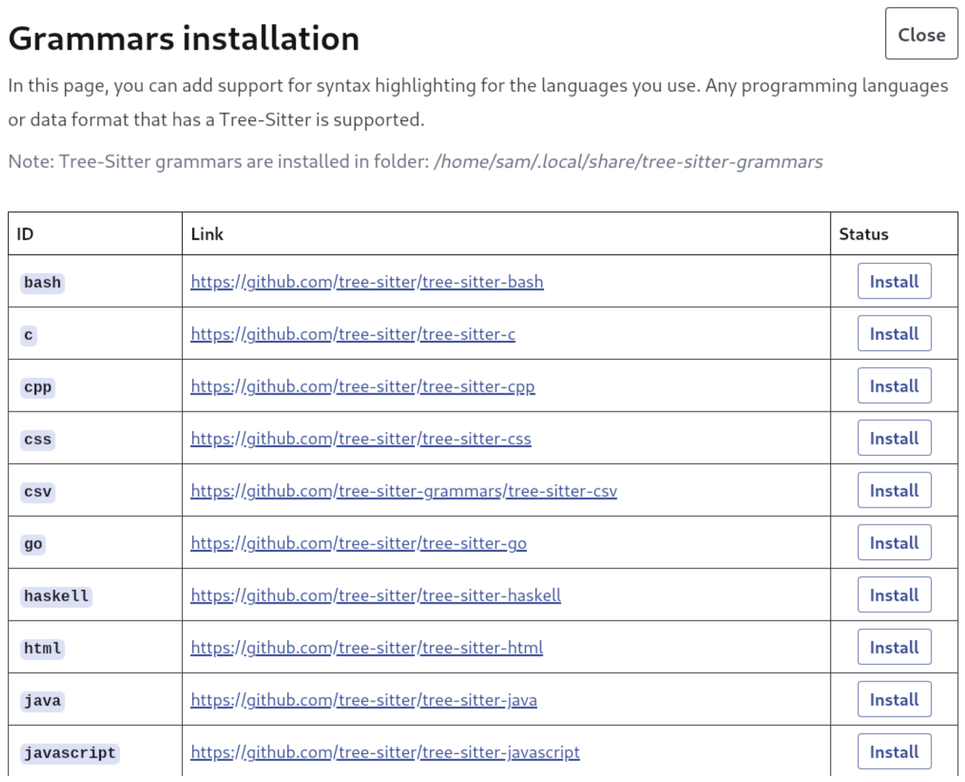
```
cargo test -- --include-ignored
```

Note: **never change the** `PATH` variable in tests, it will affects other tests as well!

Preview of DME’s README with DME itself

The preview doesn’t refresh by itself for now, but if the underlying file has changed, you can press `r` to reload the preview manually. We don’t have colors on these Bash commands yet, because the support for Bash highlighting is not installed.

Just press `Ctrl+g`, it opens this page:



Grammars installation

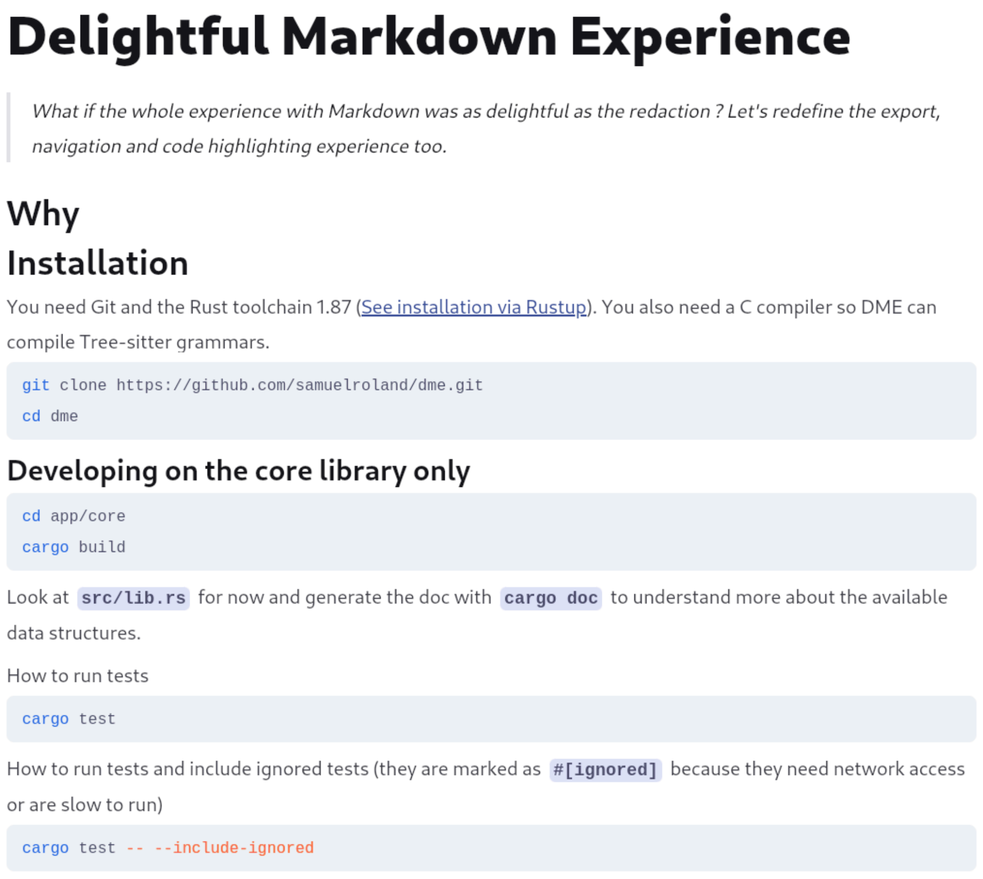
Close

In this page, you can add support for syntax highlighting for the languages you use. Any programming languages or data format that has a Tree-Sitter is supported.

Note: Tree-Sitter grammars are installed in folder: */home/sam/.local/share/tree-sitter-grammars*

ID	Link	Status
`bash`	https://github.com/tree-sitter/tree-sitter-bash	Install
`c`	https://github.com/tree-sitter/tree-sitter-c	Install
`cpp`	https://github.com/tree-sitter/tree-sitter-cpp	Install
`css`	https://github.com/tree-sitter/tree-sitter-css	Install
`csv`	https://github.com/tree-sitter-grammars/tree-sitter-csv	Install
`go`	https://github.com/tree-sitter/tree-sitter-go	Install
`haskell`	https://github.com/tree-sitter/tree-sitter-haskell	Install
`html`	https://github.com/tree-sitter/tree-sitter-html	Install
`java`	https://github.com/tree-sitter/tree-sitter-java	Install
`javascript`	https://github.com/tree-sitter/tree-sitter-javascript	Install

And click `Install` on the row with `bash`. If it works, it will show `Installed` once it's done. You can also delete it afterwards. Hit `Escape` or click on the `Close` button to go back to the previous page. Hit `r` to reload the preview and here it the result.



Delightful Markdown Experience

> *What if the whole experience with Markdown was as delightful as the redaction ? Let's redefine the export, navigation and code highlighting experience too.*

Why

Installation

You need Git and the Rust toolchain 1.87 (See installation via Rustup). You also need a C compiler so DME can compile Tree-sitter grammars.

```
git clone https://github.com/samuelroland/dme.git
cd dme
```

Developing on the core library only

```
cd app/core
cargo build
```

Look at `src/lib.rs` for now and generate the doc with `cargo doc` to understand more about the available data structures.

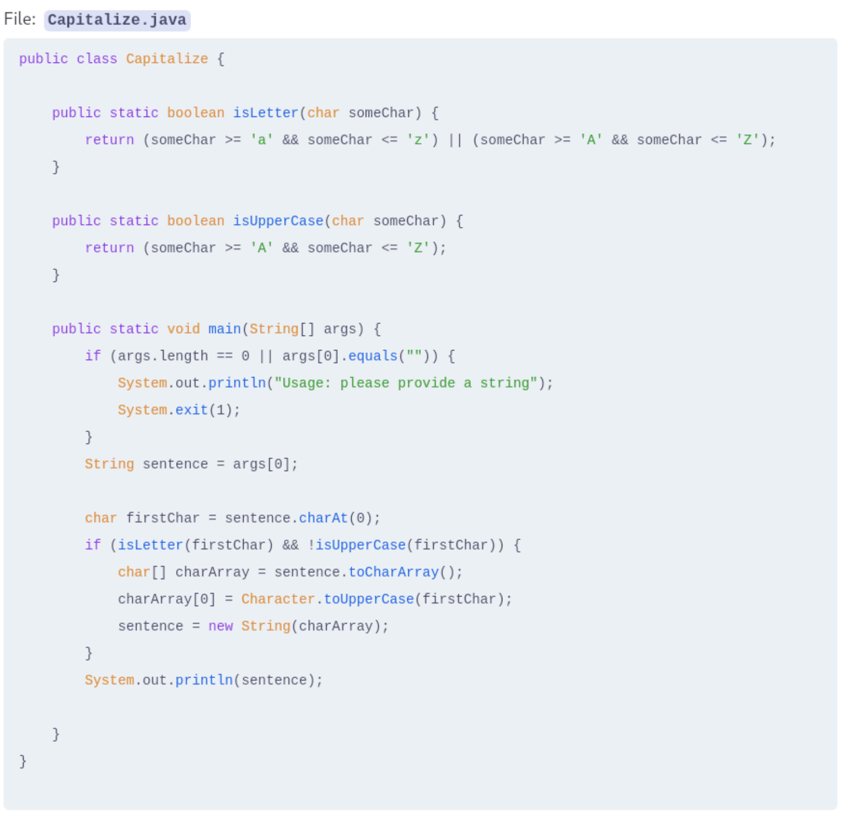
How to run tests

```
cargo test
```

How to run tests and include ignored tests (they are marked as `#[ignored]` because they need network access or are slow to run)

```
cargo test -- --include-ignored
```

The commands are now colorized.

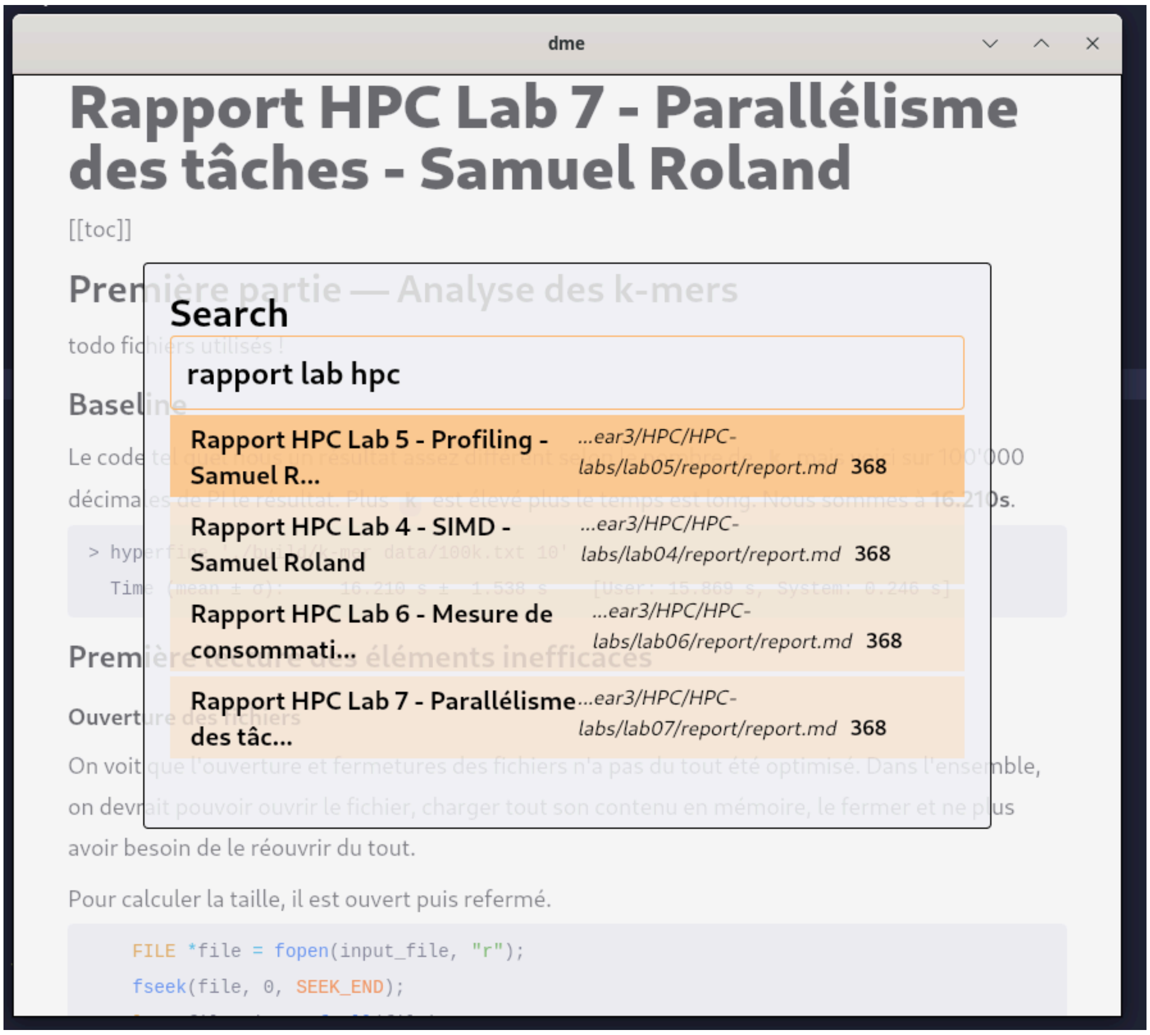

File: `Capitalize.java`

```java
public class Capitalize {

    public static boolean isLetter(char someChar) {
        return (someChar >= 'a' && someChar <= 'z') || (someChar >= 'A' && someChar <= 'Z');
    }

    public static boolean isUpperCase(char someChar) {
        return (someChar >= 'A' && someChar <= 'Z');
    }

    public static void main(String[] args) {
        if (args.length == 0 || args[0].equals("")) {
            System.out.println("Usage: please provide a string");
            System.exit(1);
        }
        String sentence = args[0];

        char firstChar = sentence.charAt(0);
        if (isLetter(firstChar) && !isUpperCase(firstChar)) {
            char[] charArray = sentence.toCharArray();
            charArray[0] = Character.toUpperCase(firstChar);
            sentence = new String(charArray);
        }
        System.out.println(sentence);

    }
}
```

The same snippet of Java highlighted in DME

The code is now better highlighted than in the VSCode preview, because we used a more advanced highlighting system called Tree-Sitter.

DME is also providing a simple but working search system. By pressing `p`, you can access a search dialogue that will match fuzzily. It means that you can do typos or type keywords in any order, and you will still get some results.


dme

Rapport HPC Lab 7 - Parallélisme des tâches - Samuel Roland

[[toc]]

Première partie — Analyse des k-mers

todo fichiers utilisés !

Baseline

Le code tel quel nous un résultat assez différent selon le nombre de k mais voici sur 100'000 décimales de PI le résultat. Plus k est élevé plus le temps est long. Nous sommes à 16.210s.

```
> hyperfine './build/k-mer data/100k.txt 10'
  Time (mean ± σ):     16.210 s ±  1.538 s    [User: 15.869 s, System: 0.246 s]
```

Première lecture des éléments inefficaces

Ouverture des fichiers

On voit que l'ouverture et fermetures des fichiers n'a pas du tout été optimisé. Dans l'ensemble, on devrait pouvoir ouvrir le fichier, charger tout son contenu en mémoire, le fermer et ne plus avoir besoin de le réouvrir du tout.

Pour calculer la taille, il est ouvert puis refermé.

```
    FILE *file = fopen(input_file, "r");
    fseek(file, 0, SEEK_END);
```

Search

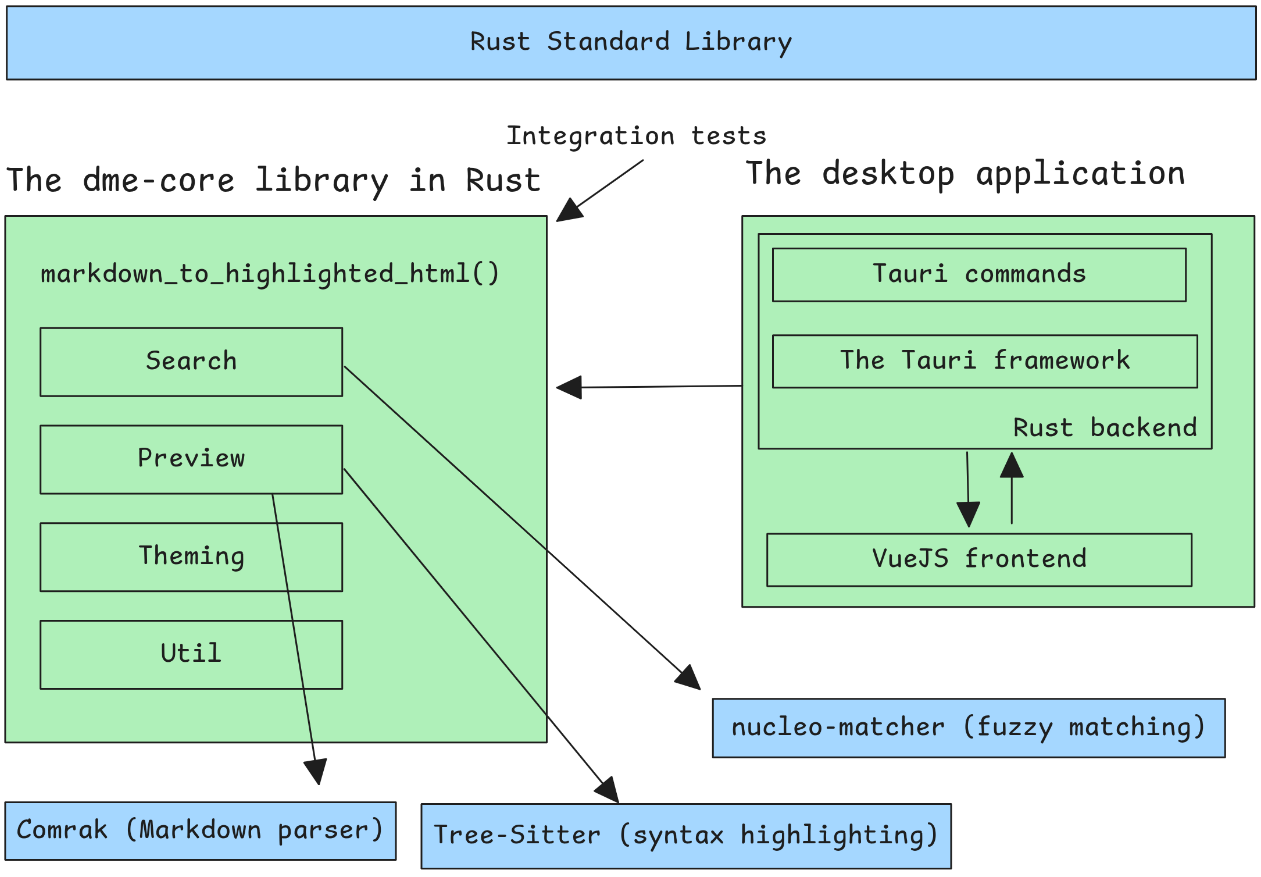
rapport lab hpc

Rapport HPC Lab 5 - Profiling - Samuel R...	...ear3/HPC/HPC-labs/lab05/report/report.md	368
Rapport HPC Lab 4 - SIMD - Samuel Roland	...ear3/HPC/HPC-labs/lab04/report/report.md	368
Rapport HPC Lab 6 - Mesure de consommati...	...ear3/HPC/HPC-labs/lab06/report/report.md	368
Rapport HPC Lab 7 - Parallélisme des tâc...	...ear3/HPC/HPC-labs/lab07/report/report.md	368

Searching for some keywords in incorrect order do matches the expected headings

To avoid very low quality match, it removes the low priority matches, so you have to type around 2 words to see the first results. The results are streamed, you will see the first one quickly, wait a bit and others might come after that. To quit the search, you can hit `Escape` twice. If you hit `Escape` once, you can move the selected entry with `j/k` or arrow down/up and hit enter to open the selected document.

Architecture



High level overview of the architecture

We have `dme-core` crate (under `app/core`) as the library where most of the logic is implemented. It contains several modules.

- `preview` responsible for rendering HTML for a given Markdown file, including syntax highlighting of code snippets
- `theming` responsible to generate CSS based on a given theme
- `search` responsible for Markdown indexing and search
- `export` responsible for PDF export (not implemented)

Then we use this library in the desktop app which is separated in 2 main parts

1. The frontend (`app/src`), is the VueJS application written in TypeScript
2. The backend (`app/src-tauri`) in Rust using Tauri. It defines some commands (some functions accessible by the frontend code) using the core library.

Some integration tests (`app/core/tests`) are testing the core library but only access public interfaces like external code.

Implementation

The search crate contains two core functionalities

- The indexation of Markdown file paths and Markdown titles in all these files, starting at the HOME folder
- The search with these 2 index, with an optional streaming

An important point is that the indexing process is asynchronous, running in separated threads. The main reason is for performance. The second reason is to avoid blocking UI / caller on a start call. Instead, the `DiskResearcher` makes available a `Progress` struct to know how much of the markdowns have been indexed.

Indexing the disk

This is the base algorithm of the indexation of markdown title on disk

```
paths = search all paths on disk ending with .md, excluding hidden folders and specific folders
for dependencies
chunks = split the paths found in N chunks, N depends on the number of CPU cores
for each chunk in chunks
    start a thread
      for each file in chunk
          get file content
          extract title base on content
          lock shared map of title
          write data to shared map of title
          unlock
      end for
    end thread
end for
```

Searching with the index

This is the base algorithm of the search inside the title map build by the indexing process

```
chunks = split the title map
for each chunk in chunks
    start a thread
        evaluate how close the title is to the query
        lock
        write the entry and its proximity score to shared map of results
        unlock
    end thread
end for

search for markdown path matching the query
join and sort results
```

The process of evaluating the proximity of the titles/path and the query use an external crate: `nucleo_matcher`. Each result is attributed a score, with the path receiving a 0.6 factor compared to title to prioritize titles. Those results are then sorted, and a subset is returned.

To ensure the quality of the results, the subset is calculated as followed:

```
get the highest scores in the results
lowerbound of accepted matches = 3/4 of the maximum

filter all matches to keep only entries with score above lowerbound
return the results
```

An important note is that even if we limit the results to 10, we might still get fewer results, if they were deemed not relevant enough.

Concurrency in search

The 2 index are defined under a `Arc` (atomic reference counting) so the last thread with a reference to it, will be able to clean the ressource. Instead of a Mutex we have a `RwLock` which implement a readers/writers pattern, where multiple readers can read lock at the same time, or a single writer can write alone.

```rust
#[derive(Debug)]
pub struct DiskResearcher {
    /// The list of all paths to Markdown files found
    /// Use a RwLock to make searches more optimized during index construction
    markdown_paths_vec: Arc<RwLock<Vec<String>>>,
    /// Each heading found will have an entry with a vector of files where it was found.
    /// Use a RwLock to make searches more optimized during index construction
    title_map: Arc<RwLock<HashMap<String, Vec<String>>>>,

    // ...

    /// The progress counter counting the number of markdown paths where the titles
    /// have been extracted and saved
    /// Use a Mutex because writes will probably be more frequent than reads
    progress_counter: Arc<Mutex<usize>>,
}
```

Because these Mutex and RwLock, we can safely run search during the construction of the index, even if the a dozen of threads are running to index and another dozen dedicated to search run the same time. The parallelisation was pretty quick to implement and didn't took time to verify like the hours spent in PCO course's labs, because of the safety garantees made by the compiler!

Syntax highlighting

Other programs using Tree-Sitter to provide syntax highlighting generally choose a few grammars to package in the program's binary file. It works and simplify the final user experience, which doesn't need to have a C compiler. Some other programs choose to bundle 130 languages to make sure everyone is happy and find its language inside. And then, the binary size explodes in 90 MB…

We don't want to choose which language is the most important, nor take all of them. We took a different approach to solve this size problem. Furthermore, we bundle no grammar at all and the user installs only the grammars needed manually via the UI. The installation is just a `git clone` of one of the proposed grammar and compilation of the C parser auto generated in each repository cloned. Then the Tree-Sitter is able to load dynamically from disk the grammars.

Here is an example of an error that also shows the value of the borrow checker in this part of the program. It is really not trivial to detect that manually by reading the code. We are constructing the command `git clone --depth 1 --single_branch` to clone a Git repository of a grammar efficiently.

```rust
let only_latest_commits: Option<u32> = Some(1);
let mut args = vec!["clone", git_clone_url];
if let Some(count) = only_latest_commits {
    args.push("--depth");
    args.push(&count.to_string());
}
if single_branch {
    args.push("--single-branch");
}
let output = Self::run_git_cmd(&args, base_directory)?;
```

And the associated compiler error

```
   Compiling dme-core v0.1.0
error[E0716]: temporary value dropped while borrowed
  --> src/util/git.rs:66:24
   |
66 |             args.push(&count.to_string());
   |                        ^^^^^^^^^^^^^^^^^ - temporary value is freed at the end of this
statement
   |                        |
   |                        creates a temporary value which is freed while still in use
...
69 |             args.push("--single-branch");
   |             ---- borrow later used here
   |
   = note: consider using a `let` binding to create a longer lived value
```

The issue is that we need to push a string slice (`&str`) but we have an integer (`u32`), so we call `.to_string()` to convert to a String and we take a reference, which is transformed as a string slice of the whole string. The issue is that this String object is local to the call and immediately freed once the argument has been used. As we continue to use `args` later, `args` contains an entry with a reference to this number that became invalid as the string no longer exist !

The note `consider using a let binding to create a longer lived value` confirms this is a lifetime issue, we need a value that lives longer that `args` to its reference stored inside `args` is always valid.

Another example where we put the paradigm in practice, is the `theme.rs` file with the definition of a `Theme` as reference an array of string slices (reference to parts of strings). This `Theme` struct is also used in struct `Renderer` and we also keep it as a reference. We had to annotate the lifetimes manually here with `'a` . This strange notation indicates that every reference with this annotation will need to live at least as long as the struct itself.

```rust
/// A theme defining colors and modifiers to be used for syntax highlighting.
pub struct Theme<'a> {
  //...
    pub(crate) supported_highlight_names: &'a [&'a str],
}

/// HTML syntax highlighting renderer.
pub struct Renderer<'a> {
    theme: &'a theme::Theme<'a>,
}
```

Why ? Because if the reference `theme` is living shorter than the `Renderer` object, it means we could do `renderer.theme` and access an invalid reference. As the borrow checker is protecting us against invalid pointer dereference, it will not compile without it.

Our experience

Our experience with the paradigm

- **Avoided thousands of possible errors**
  We encountered compile errors related to memory safety, which would have probably been missed in a C codebase.
- **Hard to think about advanced memory references**
  Sometimes, with some of the Tree-Sitter types and mutable references required, it was hard to manage some references where referenced objects didn't lived long enough. We had to refactor some parts of the code
- **No memory crash at runtime** In contrary to all C and C++ projects we did in the past, we had absolutely no memory issues that caused a crash. No segfault, no corrupted data structures, no and buffer overflow!

Our experience with Rust

- **The standard library**
  The numerous types such as `Result`, `Option`, `HashMap`, `Vec` are very useful. If we worked in C, we would miss all this plug and play experience. It also offered us advanced feature that were really helpful like `Arc` or `RwLock`. They were easy to use and well-documented.
- **Tree-Sitter library**
  The Tree-Sitter library was the complete opposite of the standard library of Rust, there was hardly any documentation, often demanding to read the code to understand how it works. Some incoherent behaviours were also detected on 2 ways to do the same thing.
- **Unit and integration testing**
  We were able to write a lot of Unit tests, and even though are program was highly concurrent we did not suffer any side effect linked to concurrency. Our tests were deterministic.
- **Be forced to manage errors** Not having exceptions and having the `Result` enum type, that represents a successful result or an error is a major advantage of the language! The fact that Rust forces to check if it was successful before trying to use the result value, adds a cost during the coding process, but this cost is largely counter balanced with the lack of runtime issues. There is always a way to "go quickly and don't care about errors" if we are okay to see the program panic (crash). We can just call `unwrap()` on `Result`, which is very helpful for unit tests or during prototyping.
- **Compilers contextual errors**
  The Rust compiler is the only one we know that can point so precisely to multiple locations in our code, to indicate not just where the issue is created but the important parts around this issue that are deeply related. When the borrow-checker generate errors, like "could not borrow after move", it gives the location of the move and the borrow and propose a fix like "considering borrowing instead of move by adding a &".
- **Proposed fixes and refactoring**
  The experience in IDE with `rust-analyzer` (the Rust language server) is amazing, there is so many fixes or refactoring proposed, it helps a lot. The compiler itself also generate propositions in its console output.

Future of DME

DME will continue to be developed in the future, here are a few ideas on what to implement next

- **Continue to improve syntax highlighting** with better queries files for Tree-Sitter grammars
- **Develop the PDF export** we didn't have time to develop this semester
- **Making syntax highlighting parallel**: we could make the highlighting even faster, which will become crucial for large documents with a lot of snippets. The work on the HPC course has not been merged yet but show some
- Making **a full text search** and keeping it fuzzy
- Continue improving the **details of rules in the search system** and more testing
- Continue **performance** benchmarking and improvements

Opportunities to dive deeper in the paradigm

For future projects or research, here are a few subjects that could be explored around our paradigm

- **Unsafe Rust**: How it works ? What are the additional possibilities and constraint change ? How is it possible to create safe wrapper around unsafe interfaces ? How a safe wrapper around a C library can be created ?
- **Complex data structures**: How is it possible to define struct self-referential attributes to define graph or network structures ?
- **Conception of advanced types in the standard library**: How `Arc`, `RefCell`, `Cell`, `Rc` are implemented and how to work with or around the borrow checker rules ? When do they need unsafe code and how this unsafe code is reviewed ?
- **Miri** (An interpreter on the internal representation of Rust): How it works ? How it helps to detect unsafe patterns at runtime for unsafe code ? That's a very interesting project that makes Rust safer, even when we are forced to use unsafe Rust.