Contents

1 RUST PROGRAMING LANGUAGE

> This guide is for absolute biggners who have no prior programming experiance but is comfortable with terminal

So let's start by a simple bash script

```
mkdir ~/device

mount /dev/sda1 ~/device && echo "Mounted sucessfully"

touch example.txt

echo "This is an example file." > example.txt

cat example.txt
```

So, how does this work ?

we know that mkdir, mount touch are programs which does specific tasks. every exicutable situated in the *bin directory can be executed via a terminal. Users can setup there custom bin path to* ~.local/bin or ~/.bin thereby the exicuaatbles in those directories will get added to the path so that its acessable from terminal.

when we install a distro some applications will get installed by default and we can use that right.

In rust there is something like this. There is a set of crate/modules which are part of standard library which will be thre after we install rust.

But in programing languages we need to impot

> path in bash is also imported this way via .bashrc

1.1 Structs in Rust

A struct in Rust is a way to create custom data types. It is a collection of fields, each with its own type. Structs are useful for grouping related data together and creating more complex types.

```
struct Book {
    title: String,
    author: String,
    pages: u32,
}
```

In this example, Book is a struct with three fields: title, author, and pages.

1.1.1 Creating Instances of Structs

You can create an instance of a struct by specifying values for each field:

```
let my_book = Book {
    title: String::from("The Rust Programming Language"),
    author: String::from("Steve Klabnik and Carol Nichols"),
    pages: 552,
};
```

-
-
-

1.1.2 Implementing Methods for Structs

The impl block allows you to define methods for a struct. Methods are functions that are associated with a struct. Basic impl Example.

```
impl Book {
```

```
    fn get_summary(&self) -> String {
        format!("{} by {} ({} pages)", self.title, self.author, self.pages)
    }
}
```

Here, $\text{get}_{\text{summary}}$ is a method that returns a summary of the book. * Default Values with Structs

To provide default values, you can implement the Default trait for your struct. This allows you to create an instance with default values for fields you don't specify. Implementing the Default Trait

```
use std::default::Default;

struct Book {
    title: String,
    author: String,
    pages: u32,
}

impl Default for Book {
    fn default() -> Self {
        Self {
            title: String::from("Untitled"),
            author: String::from("Unknown"),
            pages: 0,
        }
    }
}
```

Now you can create a Book instance with default values:

```
let default_book = Book::default();
```

1.1.3 Applying to Your $\text{clipcat}_{\text{notify}}$ Function

Let's define a struct for notification settings and implement methods for it, including default values. Define the Struct

```
use notify_rust::Notification;
use notify_rust::Timeout;
use std::path::Path;

struct NotificationSettings {
    selected_option: String,
    summary: String,
    icon_path: String,
    timeout: Timeout,
}

impl Default for NotificationSettings {
    fn default() -> Self {
        Self {
            selected_option: String::from(""),
            summary: String::from("Notification"),
            icon_path: String::from("default_icon.svg"),
            timeout: Timeout::Milliseconds(2000),
        }
    }
}

impl NotificationSettings {
    fn show(&self) {
        let icon_path = Path::new(&self.icon_path).canonicalize().unwrap_or_else(|_| Pa
        let icon_path_str = icon_path.to_str().unwrap_or_default();
        let body_message = format!("You selected {}", self.selected_option);
        Notification::new()
            .summary(&self.summary)
            .body(&body_message)
            .icon(&icon_path_str)
            .timeout(self.timeout)
            .show()
            .unwrap();
    }
}
```

Using the Struct in Your Function

You can now use this struct in your clipcat$_{\text{notify}}$ function:

```
fn clipcat_notify(selected_option: String, summary: Option<String>) {
    let mut settings = NotificationSettings::default();
    settings.selected_option = selected_option;
    if let Some(summary) = summary {
        settings.summary = summary;
    }
    settings.show();
}
```

Full Integration in Your Main Function

Here's how you would integrate this into your existing main function:

```
fn main() -> Result<(), Box<dyn Error>> {
    // Initialize clipboard context
    let mut clipboard: ClipboardContext = ClipboardProvider::new()?;

    // Get clipboard contents
    let args_str = clipboard.get_contents()?;

    // Connect to the SQLite database
    let b_conn = Connection::open("bookmark.db")?;

    // Create the bookmark table if it doesn't exist
    b_conn.execute(
        "CREATE TABLE IF NOT EXISTS bookmark (
            id   INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            bookmark BLOB
        )",
        (), // empty list of parameters
    )?;

    // Prepare the statement to check if the bookmark exists
    let mut stmt = b_conn.prepare("SELECT COUNT(*) FROM bookmark WHERE name = ?1")?;

    // Query for the count of existing bookmarks
    let count: i32 = stmt.query_row(params![args_str.clone()], |row| row.get(0))?;
```

```
if count == 0 {
    // Insert a new bookmark into the database if it doesn't exist
    let me = BookMark {
        id: 1,
        name: args_str.clone(), // Clone the string to use it here
        bookmark: "Does Steven get add to db".to_string(),
    };
    b_conn.execute(
        "INSERT INTO bookmark (name, bookmark) VALUES (?1, ?2)",
        (&me.name, &me.bookmark),
    )?;
} else {
    println!(
        "Bookmark with the name '{}' already exists. Skipping insertion.",
        args_str.bright_red().bold()
    );
}

// Fetch bookmarks from the database
let mut stmt = b_conn.prepare("SELECT id, name, bookmark FROM bookmark")?;
let bookmark_iter = stmt.query_map([], |row| {
    Ok(BookMark {
        id: row.get(0)?,
        name: row.get(1)?,
        bookmark: row.get(2)?,
    })
})?;

// Collect the names of the bookmarks to display in Rofi
let mut options = Vec::new();
for bookmark in bookmark_iter {
    let bookmark = bookmark?;
    options.push(bookmark.name);
}

// Specify the path to the custom theme file
let theme_file = "rofi.rasi";

// Create a new Rofi instance and run it with the options
```

```
    let rofi_result = Rofi::new(&options).theme(Some(theme_file)).run();

    // Handle the result of the Rofi run
    match rofi_result {
        Ok(selected_option) => clipcat_notify(selected_option, None),
        Err(e) => println!("No option selected or an error occurred: {}", e),
    }

    Ok(())
}

fn clipcat_notify(selected_option: String, summary: Option<String>) {
    let mut settings = NotificationSettings::default();
    settings.selected_option = selected_option;
    if let Some(summary) = summary {
        settings.summary = summary;
    }
    settings.show();
}
```

This implementation ensures that your notification settings have default values and can be customized as needed.