Rust libs

Command Line Argument Parser (clap)

```
let args = Command::new("poi")
        .arg(
            arg!(--lat <LATITUDE> "Latitude of the location")
                .required(false)
                .default_value("13.0827")
                .value_parser(clap::value_parser!(f64)),
        )
        .arg(..)
        .get_matches();
let lat = *args.get_one::<f64>("lat")
                .expect("Cannot be None because of default val");
let lng = ...
```

Derive Macro

```
#[derive(Parser, Debug)]
#[command(author, version, about)] // read from Cargo.toml
struct Args {
   /// Single arg!
   #[arg(long, default_value=13.0827)]
   lat: f64,
    ...,
   /// Subcommands!
   #[command(subcommand)]
   category: Category,
}
```

.. it can do even more!

- Automatic help generation
- Suggested fixes for users
- Colored output
- Shell completions
- etc..

Reqwest

```
async fn single_req(url: &String) -> Result<NearbyPlacesParser> {
    let response = reqwest::get(url).await?;
    let parsed = match response.status() {
        reqwest::StatusCode::OK => {
            response.json::<NearbyPlacesParser>().await?,
        }
        reqwest::StatusCode::UNAUTHORIZED => {
            panic!("Check your GOOGLE_DEV_API_KEY"),
        }
        _ => return Err(eyre!("Error fetching response from server")),
    };
    Ok(parsed)
    }
```

- Async HTTP client
- Proxies, cookies etc..

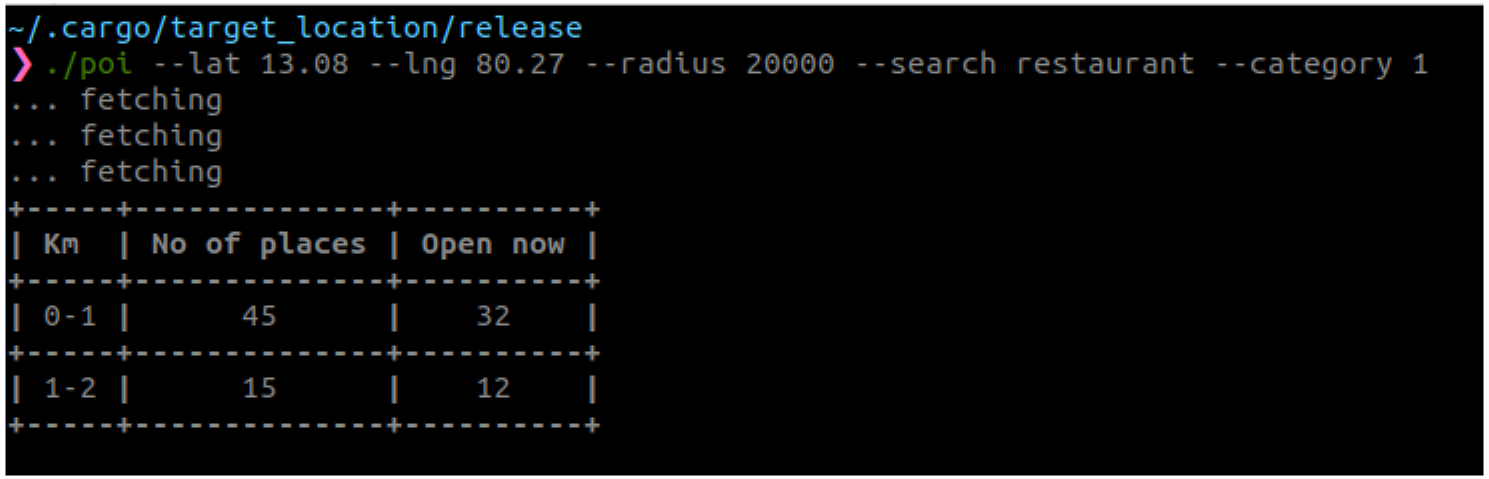
Serde

```rust
#[derive(Debug, Serialize, Deserialize)]
struct NearbyPlacesParser {
    #[serde(alias = "results")]
    places: Vec<Place>,
    next_page_token: Option<String>,
}
#[derive(Debug, Serialize, Deserialize)]
struct Place {
    place_id: String,
    opening_hours: Option<OpeningHours>,
    geometry: Geometry,
}
#[derive(Debug, Serialize, Deserialize)]
struct OpeningHours {
    open_now: bool,
}
...
```

cli_table

```
    let table = total_places.into_iter()
        .zip(open_places.into_iter())
        .map(|((km, total), (_, open))| {
            vec![
                format!("{}-{km}", km - step_size).cell(),
                total.cell().justify(Justify::Center),
                open.cell().justify(Justify::Center),
            ]
        })
        .collect::<Vec<_>>()
        .table()
        .title(vec![
            "Km".cell().bold(true),
            "No of places".cell().bold(true),
            "Open now".cell().bold(true),
        ]);
```

cli_table output



```
~/.cargo/target_location/release
❯ ./poi --lat 13.08 --lng 80.27 --radius 20000 --search restaurant --category 1
... fetching
... fetching
... fetching
+-----+--------------+----------+
| Km  | No of places | Open now |
+-----+--------------+----------+
| 0-1 |      45      |    32    |
+-----+--------------+----------+
| 1-2 |      15      |    12    |
+-----+--------------+----------+
```

Tokio

- Asynchronous runtime, useful if your program needs to wait
- Any number of tasks can be spawned and tokio will manage everything for you..
    - Tasks != threads and instead they're much cheaper to spawn
- Concurrent and Parallel

```
#[tokio::main]
async fn main() {
    ...
    }
```

Spawning Tasks

```rust
let mut handles = Vec::new();
for len in 1..word.chars().count() {
    for perm in word.chars().permutations(len).unique() {
        let perm = String::from_iter(perm);
        let handle = tokio::spawn(is_valid(perm.clone()));
        handles.push(handle);
    }
}

let mut op = Vec::new();
for handle in handles {
    if let Some(word) = handle.await.unwrap() {
        op.push(word);
    }
}
println!("Valid words are: \n{op:?}");
```

… explore!

https://crates.io/