Rust - Day 8

Managing Growing Projects with Packages, Crates, and Modules

- As a project grows, you should organise code by splitting it into multiple modules and then multiple files. **Modularity**
- As a package grows, you can extract parts into separate crates that become external dependencies.
- You can't have two items with the same name in the same scope
- Rust has a number of features that allow you to manage your code's organization, including which details are exposed, which details are private, and what names are in each scope in your programs.
- These features, sometimes collectively referred to as the **module system**, include:
  - **Packages**: A Cargo feature that lets you build, test, share crates
  - **Crates**: A tree of modules that produces a library or executable
  - **Modules**: Lets you control the organization, scope, and privacy of paths
  - **Paths**: a way of naming an item, such as a struct, function or module

Packages and Crates (Libraries)

Crates

- A crate is the smallest amount of code that the Rust compiler considers at a time. Even if you run `rustc` rather than `cargo` and pass a single source code file, the compiler considers that file to be a crate.
- Crates can contain modules, and the modules may be defined in other files that get compiled with the crate, as we'll see in the coming sections.
- A crate can come in one of two forms: a binary crate or a library crate.
- *Binary crates* are programs you can compile to an executable that you can run, such as a command-line program or a server. **Each must have a function called `main` that defines what happens when the executable runs.** All the crates we've created so far have been binary crates.
- *Library crates* don't have a `main` function, and they don't compile to an executable. Instead, they define functionality intended to be shared with multiple projects.
- Most of the time when Rustaceans say "crate", they mean library crate, and they use "crate" interchangeably with the general programming concept of a "library".
- The *crate root* is a source file that the Rust compiler starts from and makes up the root module of your crate.

Packages

- A *package* is a bundle of one or more crates that provides a set of functionality. A package contains a *Cargo.toml* file that describes how to build those crates. Cargo is actually a package that contains the binary crate for the command-line tool you've been using to build your code.
- **A package can contain as many binary crates as you like, but at most only one library crate. A package must contain at least one crate, whether that's a library or binary crate.**

```
$ cargo new my-project
     Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

- In the project directory, there's a *Cargo.toml* file, giving us a package.
- There's also a src directory that contains **main.rs**

- **src/main.rs is the crate root of a binary crate with the same name as the package**
- **Likewise, Cargo knows that if the package directory contains *src/lib.rs*, the package contains a library crate with the same name as the package, and *src/lib.rs* is its crate root.**
- Cargo passes the crate root files to `rustc` to build the library or binary.

> **Here, we have a package that only contains *src/main.rs*, meaning it only contains a binary crate named `my-project`. If a package contains *src/main.rs* and *src/lib.rs*, it has two crates: a binary and a library, both with the same name as the package. A package can have multiple binary crates by placing files in the *src/bin* directory: each file will be a separate binary crate.**

Modules to Control Scope and Privacy

- Before we get to the details of modules and paths, here we provide a quick reference on how modules, paths, the `use` keyword, and the `pub` keyword work in the compiler, and how most developers organize their code. We'll be going through examples of each of these rules throughout this chapter, but this is a great place to refer to as a reminder of how modules work.
- **Start from the crate root**: When compiling a crate, the compiler first looks in the crate root file (usually *src/lib.rs* for a library crate or *src/main.rs* for a binary crate) for code to compile.
-
  - **Declaring modules**: In the crate root file, you can declare new modules; say you declare a "garden" module with `mod garden;`. The compiler will look for the module's code in these places:
    - Inline, within curly brackets that replace the semicolon following `mod garden`
    - In the file *src/garden.rs*
    - In the file *src/garden/mod.rs*
- **Declaring submodules**: In any file other than the crate root, you can declare submodules. For example, you might declare `mod vegetables;` in *src/garden.rs*. The compiler will look for the submodule's code within the directory named for the parent module in these places:
  - Inline, directly following `mod vegetables`, within curly brackets instead of the semicolon
  - In the file *src/garden/vegetables.rs*
  - In the file *src/garden/vegetables/mod.rs*
- **Paths to code in modules**: Once a module is part of your crate, you can refer to code in that module from anywhere else in that same crate, as long as the privacy rules allow, using the path to the code.
  - For example, an `Asparagus` type in the garden vegetables module would be found at `crate::garden::vegetables::Asparagus`.
- **Private vs. public**: Code within a module is private from its parent modules by default. To make a module public, declare it with `pub mod` instead of `mod`. To make items within a public module public as well, use `pub` before their declarations.
- **The `use` keyword**: Within a scope, the `use` keyword creates shortcuts to items to reduce repetition of long paths. In any scope that can refer to `crate::garden::vegetables::Asparagus`, you can create a shortcut with `use crate::garden::vegetables::Asparagus;` and from then on you only need to write `Asparagus` to make use of that type in the scope.
- We create a binary crate named `backyard` that illustrates these rules.

```
backyard
├── Cargo.lock
├── Cargo.toml
└── src
    ├── garden
    │   └── vegetables.rs
    ├── garden.rs
    └── main.rs
```

- The crate root file in this case is *src/main.rs*, and it contains:

```
use crate::garden::vegetables::Asparagus;

pub mod garden;
```

```
fn main() {
    let plant = Asparagus {};
    println!("I'm growing {plant:?}!");
}
```

- The `pub mod garden;` line tells the compiler to include the code it finds in *src/garden.rs*, which is:

```
pub mod vegetables;
```

- Here, `pub mod vegetables;` means the code in *src/garden/vegetables.rs* is included too. That code is:

```
#[derive(Debug)]
pub struct Asparagus {}
```

- Now let's get into the details of these rules and demonstrate them in action!

Grouping Related Code in Modules

- *Modules* let us organize code within a crate for readability and easy reuse.
- Modules also allow us to control the *privacy* of items because code within a module is private by default.
- Private items are internal implementation details not available for outside use.
- We can choose to make modules and the items within them public, which exposes them to allow external code to use and depend on them.
- As an example, let's write a library crate that provides the functionality of a restaurant. We'll define the signatures of functions but leave their bodies empty to concentrate on the organization of the code rather than the implementation of a restaurant.
- In the restaurant industry, some parts of a restaurant are referred to as *front of house* and others as *back of house*. Front of house is where customers are; this encompasses where the hosts seat customers, servers take orders and payment, and bartenders make drinks. Back of house is where the chefs and cooks work in the kitchen, dishwashers clean up, and managers do administrative work.
- To structure our crate in this way, we can organize its functions into nested modules. Create a new library named `restaurant` by running `cargo new restaurant --lib`. Then enter the code below into *src/lib.rs* to define some modules and function signatures; this code is the front of house section.

```
mod front_of_house {
        mod hosting {
                fn add_to_waitlist() {}
                fn seat_at_table() {}
        }

        mod serving {
                fn take_order()
                fn serve_order()
                fn take_payment()
        }
}
```

- We define a module with the `mod` keyword followed by the name of the module (in this case, `front_of_house`).
- The body of the module then goes inside curly brackets. Inside modules, we can place other modules, as in this case with the modules `hosting` and `serving`.
- Modules can also hold definitions for other items, such as structs, enums, constants, traits, and functions.
- **By using modules, we can group related definitions together and name why they're related.**
- Earlier, we mentioned that *src/main.rs* and *src/lib.rs* are called crate roots. The reason for their name is that the contents of either of these two files form a module named `crate` at the root of the crate's module structure, known as the *module tree*.

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

- **Just like directories in a filesystem, you use modules to organize your code. And just like files in a directory, we need a way to find our modules.**

Better Example

- Let's explore how to keep code organised and clean with the help of a demo project

```
cargo new project_structure
```

- Create a new src/lib.rs file
- It's a good practice to keep all functionalities in the library crate and use the same in main.rs
- To create a project with just library crate.

```
cargo new project --lib
```

- Now, write this dummy code in the lib.rs

```
#![allow(dead_code, unused_variables)]

struct Credentials {
    username: String,
    password: String,
}

enum Status {
    Connected,
    Interrupted,
}

fn connect_to_database() -> Status {
    Status::Connected
}

fn get_user() {}

fn login(cred: Credentials) {
    get_user()
}

fn authenticate(cred: Credentials) {
    if let Status::Connected = connect_to_database() {}
}
```

- To be able to call these methods, we should **explicitely** define them as **public** and then import them in other files like:

```
#![allow(dead_code, unused_variables)]

pub struct Credentials {
    username: String,
    password: String,
}

pub enum Status {
    Connected,
    Interrupted,
}

pub fn connect_to_database() → Status {
    Status::Connected
}

pub fn get_user() {}

pub fn login(cred: Credentials) {
    get_user()
}

pub fn authenticate(cred: Credentials) {
    if let Status::Connected = connect_to_database() {}
}
```

- The fields inside the struct are also private so must be explicitly defined as **public**

```
pub struct Credentials {
    pub username: String,
    pub password: String,
}
```

- Now this can be imported in main.rs for use

```
use project_structure::{authenticate, Credentials};

fn main() {
    let cred = Credentials {
        username: String::from("ayroid"),
        password: String::from("password"),
    };

    authenticate(cred);
}
```

- `use` keyword is used to bring things into scope.
- `project_structure` is the name of our current project.
- **Now, let's reorganise our lib.rs code into modules and files**
- Firstly, add the code into related modules

```
#![allow(dead_code, unused_variables)]

mod database {
    pub enum Status {
        Connected,
        Interrupted,
    }

    pub fn connect_to_database() → Status {
        Status::Connected
    }

    pub fn get_user() {}
}

mod auth_utils {
    pub fn login(cred: Credentials) {
        get_user()
    }

    mod models {
        pub struct Credentials {
            pub username: String,
            pub password: String,
        }
    }
}

pub fn authenticate(cred: Credentials) {
    if let Status::Connected = connect_to_database() {}
}
```

- This way the code is more structured with related functions and structs grouped together. **There's an inner module too inside the auth_utils module.**
- To be able to access functions and structures, update the imports

```
#![allow(dead_code, unused_variables)]

mod database {
    pub enum Status {
        Connected,
        Interrupted,
    }

    pub fn connect_to_database() → Status {
        Status::Connected
    }

    pub fn get_user() {}
}

pub mod auth_utils {

    pub fn login(cred: models::Credentials) {
        super::database::get_user()
    }

    pub mod models {
        pub struct Credentials {
            pub username: String,
            pub password: String,
        }
    }
}

pub fn authenticate(cred: auth_utils::models::Credentials) {
    if let database::Status::Connected = database::connect_to_database() {}
}
```

- The `login` method can't access the `get_user()` method and to fix this there are two ways:
    - use super keyword, as they are in the same file but differ in hierarchy
        - **super::database::get_user()**
    - use crate keyword, this allows to use absolute paths from project root
        - **crate::database::get_user()**
- Finally, update the main.rs to fix the imports

```
use project_structure::auth_utils::models::Credentials;
use project_structure::authenticate;

fn main() {
    let cred = Credentials {
        username: String::from("ayroid"),
        password: String::from("password"),
    };

    authenticate(cred);
}
```

- Now, we can split this code into separate files and just use them in the lib.rs
- The convention is to either use - **src/module_name.rs** or **src/module_name/mod.rs**
- src/database.rs

```
pub enum Status {
  Connected,
  Interrupted,
}

pub fn connect_to_database() → Status {
  Status::Connected
}

pub fn get_user() {}
```

- src/auth_utils.rs

```
pub fn login(cred: models::Credentials) {
    super::database::get_user()
}

pub mod models {
    pub struct Credentials {
        pub username: String,
        pub password: String,
    }
}
```

- Update the lib.rs to use these new modules

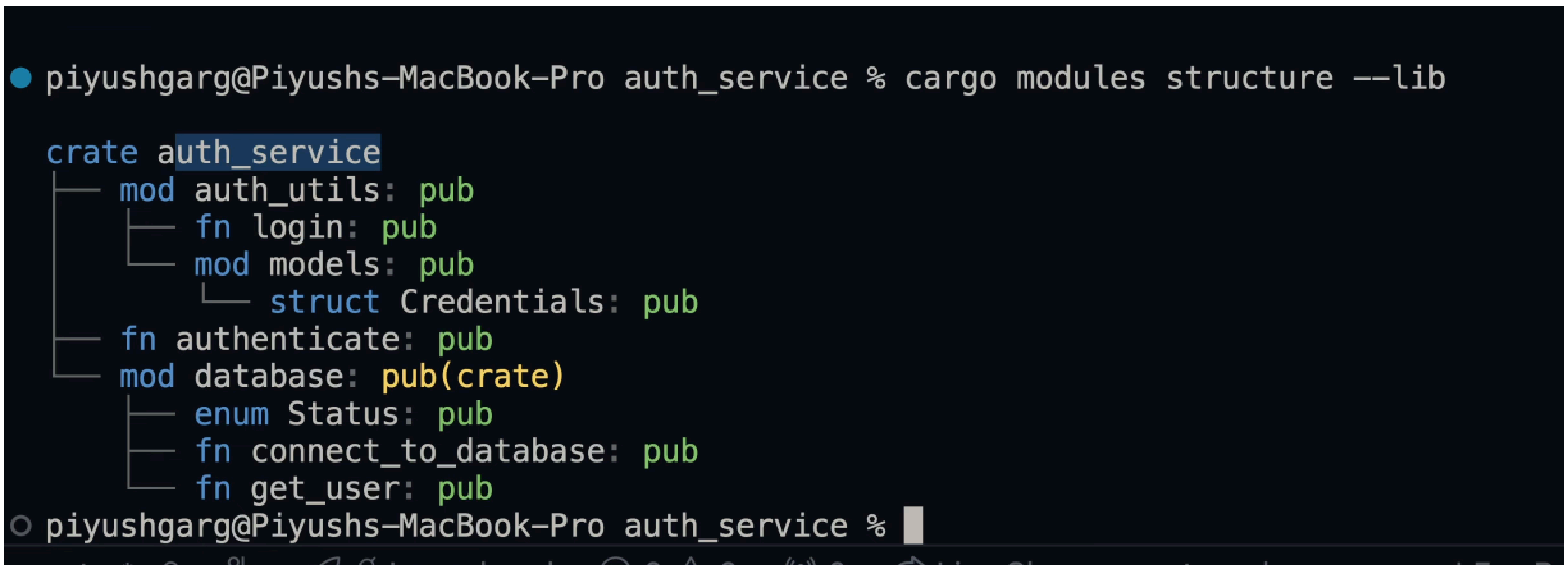
```
#![allow(dead_code, unused_variables)]

pub mod auth_utils;
mod database;

use auth_utils::models::Credentials;
use database::{Status, connect_to_database};

pub fn authenticate(cred: Credentials) {
    if let Status::Connected = connect_to_database() {}
}
```

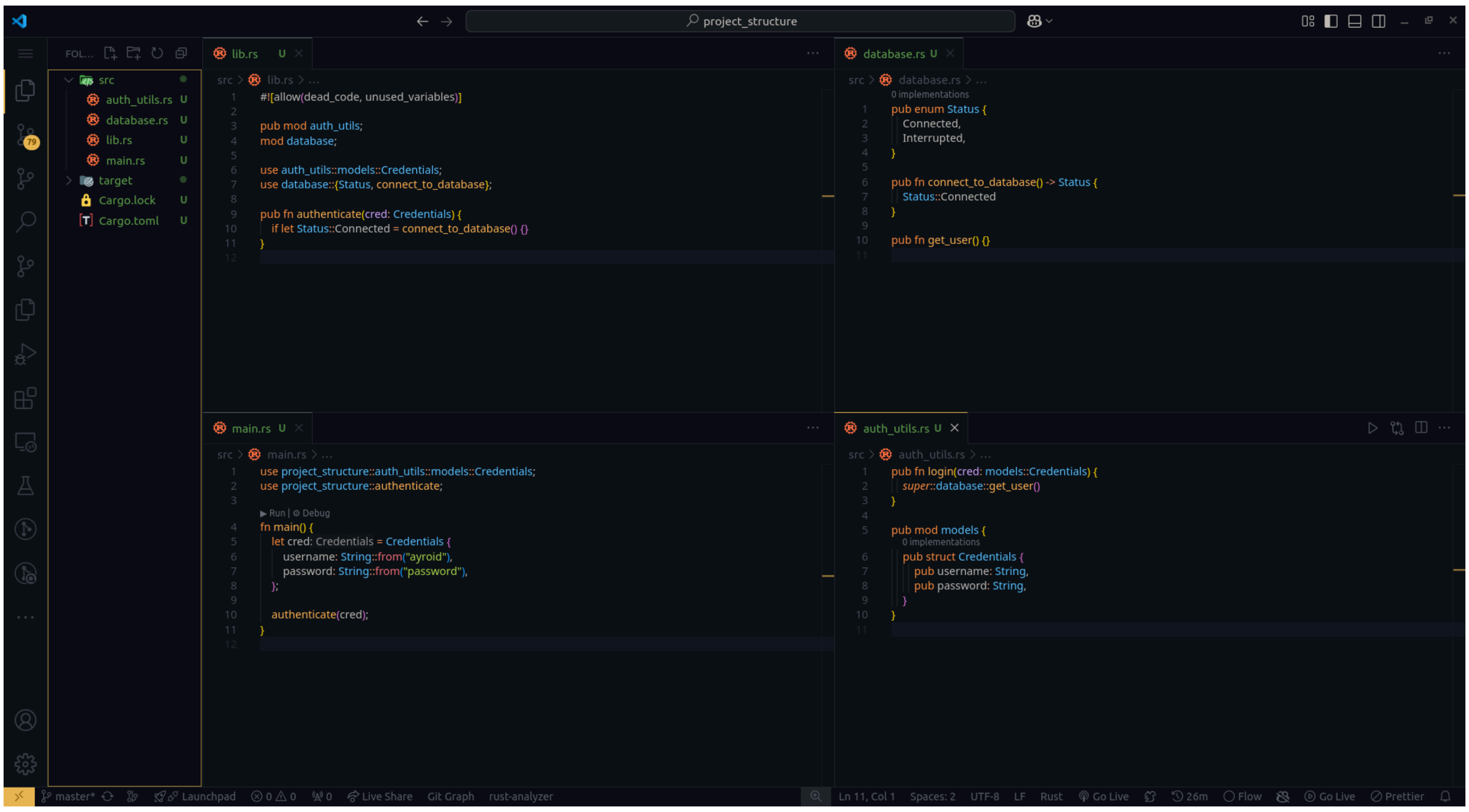
- cargo-modules is a crate that can be used to visualize folder structure with all modules and privacy status.



```
● piyushgarg@Piyushs-MacBook-Pro auth_service % cargo modules structure --lib

  crate auth_service
  ├── mod auth_utils: pub
  │   ├── fn login: pub
  │   └── mod models: pub
  │       └── struct Credentials: pub
  ├── fn authenticate: pub
  └── mod database: pub(crate)
      ├── enum Status: pub
      ├── fn connect_to_database: pub
      └── fn get_user: pub
○ piyushgarg@Piyushs-MacBook-Pro auth_service %
```

Final Structured Project



```
src
  auth_utils.rs
  database.rs
  lib.rs
  main.rs
target
Cargo.lock
Cargo.toml
```

lib.rs

```rust
#![allow(dead_code, unused_variables)]

pub mod auth_utils;
mod database;

use auth_utils::models::Credentials;
use database::{Status, connect_to_database};

pub fn authenticate(cred: Credentials) {
    if let Status::Connected = connect_to_database() {}
}
```

database.rs

```rust
pub enum Status {
    Connected,
    Interrupted,
}

pub fn connect_to_database() -> Status {
    Status::Connected
}

pub fn get_user() {}
```

main.rs

```rust
use project_structure::auth_utils::models::Credentials;
use project_structure::authenticate;

fn main() {
    let cred: Credentials = Credentials {
        username: String::from("ayroid"),
        password: String::from("password"),
    };

    authenticate(cred);
}
```

auth_utils.rs

```rust
pub fn login(cred: models::Credentials) {
    super::database::get_user()
}

pub mod models {
    pub struct Credentials {
        pub username: String,
        pub password: String,
    }
}
```