Rust Workshop

Day 5

The Rust Ecosystem

- Libraries & Documentation
- Idiomatic APIs
- Developer Tools
- Testing
- Continuous Integration & Delivery

Libraries & Documentation

- How to find and use libraries
- How to read their documentation

Finding Libraries

1. blessed.rs
2. lib.rs
3. asking the community
4. crates.io

Not in `std`

- random
- regex
- logging
- time
- http

Using Libraries

demo

Finding Documentation

docs.rs/rand

Idiomatic APIs

Why talk about APIs?

A Rust-API can look *very* different than a C-API.

Case Study #1 – `itertools`

docs.rs/itertools

Case Study #2 – `serde`

docs.rs/serde

Development Tools

Included with `rustup`

- toolchain version manager
- build tool
- package manager
- formatter
- linter
- documentation generator
- LSP

Beyond `rustup` ➡ blessed.rs

- cross compilation
- dependency management & auditing
- snapshot testing
- benchmarking
- release automation
- and more...

FFI / Interop

Lots of great bindings generators, including but not limited to:

- C
- C++
- Python
- Node.js
- and more...

Testing

A function like this can be anywhere.

```
#[test]
fn program_is_correct() {
    assert_eq!(2 + 2, 4, "math has stopped working");
}
```

It will be executed by `cargo test`.

Do your tests have some shared util code?

```
#[cfg(test)]
mod tests {
    fn setup() {}
    fn teardown() {}

    #[test]
    fn program_is_correct() {
        setup();
        // ...
        teardown();
    }
}
```

It is good practice to have a `tests` module.

The module is only compiled during testing, due to the `#[cfg(test)]` attribute.

You can return `Result`s from your tests.

```
#[test]
fn it_works() -> Result<(), String> {
    if 2 + 2 == 4 {
        Ok(())
    } else {
        Err(String::from("two plus two does not equal four"))
    }
}
```

As expected, `Ok` means the test passed, `Err` means it failed.

Test Organization

unit tests	have access to your internals according to normal visibility rules
integration tests	only have access to the public API of your library

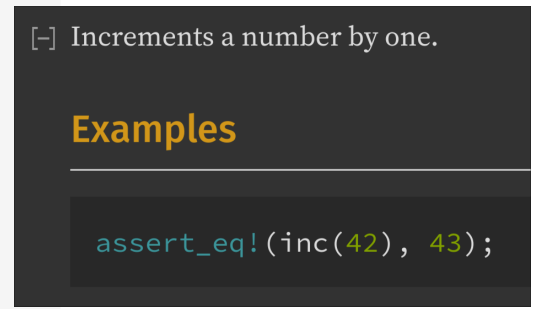
```
├─ src
│  └─ *.rs      <-- unit tests
├─ tests
│  └─ *.rs      <-- integration tests
└─ Cargo.toml
```

Integration tests only work for libraries (`lib.rs`) !

➔ It is common even for binaries to be split into `main.rs` and `lib.rs` ,
with `main.rs` being small and simple.

Documentation Tests

```
1  /// Increments a number by one.
2  ///
3  /// # Examples
4  ///
5  /// ```
6  /// assert_eq!(inc(42), 43);
7  /// ```
8  pub fn inc(x: i32) -> i32 {
9      x + 1
10 }
```



`cargo test` will run this example as a test!

Continuous Integration & Delivery

tutorial in ➡ `rust-exercises/day_4/README.md`

Final Project 

You learned how to write Rust code during the first three days
and you had a healthy dose of practice.

Now, the final project is less about how to write Rust code
and more about getting your Rust-based software into production.

What does production look like?

client	server	storage
CLI Python module web app	HTTP/REST-API WebSocket	file system SQL

Our final project is to create and ship one or a combination of these.

Simulating a Postal Service

This topic works well for many possible combinations of clients and servers.

basic, CLI-only example:

```
$ paekli-cli receive
I don't have any paekli for you! I'm sorry 😢

$ paekli-cli send --express "Ferris Plushie 🦀"

$ paekli-cli receive
{ "content": "Ferris Plushie 🦀", "express": true }
```

Now imagine you send a paekli via CLI and it automatically arrives in the web app, pushed via websocket 🤯

Practice 

`rust-exercises/day_5/README.md`

Work on Projects

paekli-rs	senekor.github.io/paekli-rs
LED-Matrix	github.zhaw.ch/senk/led-matrix-rs-template