

Системне Програмування

З використанням мови програмування **Rust.**
Application Structure. Testing. Ecosystem

Створення програми – це лише частина роботи програміста. Написати код, який виконує поставлене завдання, безумовно, важливо, але ще важливіше переконатися, що цей код працює правильно, ефективно і надійно. Саме для цього існує тестування, яке є невід’ємною частиною розробки програмного забезпечення.

Часто розробники приділяють основну увагу лише написанню функціональності, нехтуючи тестами або сприймаючи їх як другорядний процес. Проте без якісного тестування навіть найкращий алгоритм може виявитися непередбачуваним у реальному використанні. Помилки в коді можуть призводити до збоїв, втрати даних та навіть серйозних фінансових втрат. Тому тестування не просто важливе – воно може бути критично необхідним.

Ще один аспект сучасного програмування – це повторне використання коду. Ми рідко пишемо все з нуля. Замість цього ми використовуємо бібліотеки, фреймворки та інші готові рішення, які допомагають нам значно скоротити час розробки. Це дає змогу зосередитися на бізнес-логіці та унікальних аспектах програми, не витрачаючи зусиль на вирішення вже відомих проблем. Але разом із цим виникає й новий виклик: необхідність тестувати не лише власний код, а й те, як він взаємодіє з зовнішніми бібліотеками.

Отже, тестування – це не просто доповнення до написання коду, а повноцінна частина процесу розробки. І чим раніше воно починається, тим вищою буде якість кінцевого продукту. Сучасний програміст – це не просто автор коду, а інженер, який забезпечує його надійність і ефективність.

Старт програми

Коли ми закінчили написання програми, ми маємо її запустити якимось чином. Як в усіх мовах програмування нам якимось чином потрібно “описати” точку входу. Той момент з якого починається виконання нашого коду.

В Rust так само, але є деякі нюанси.

По-перше, `main` у Rust завжди має повертати `()`, або **`Result<(), E>`**, де **`E`** – це тип помилки. Це дає змогу використовувати оператор `?` безпосередньо в `main`, що значно спрощує обробку помилок у програмах.

Старт програми

```
fn main() -> Result<(), Box<dyn std::error::Error>> {  
    println!("Hello, world!");  
    Ok(())  
}
```

Якщо нам потрібно повернути помилку, яка буде опрацьована операційною системою, в Rust є відповідна функція:

```
fn main() {  
    // ...  
    std::process::exit(code: 1);  
}
```


Тестування

Технічно будь-який код може бути розділений на прості функції

```
fn f1(a: A) -> B
fn f2(a: B) -> C
fn f3(a: C) -> D
fn f4(a: D) -> E
```

Далі наша програма,
це є композиція цих функцій

```
fn app(a: A) -> E {
  let b: B = f1(a);
  let c: B = f2(b);
  let d: B = f3(c);
  let e: B = f4(d);
  e
}
```

Тестування

Але тестувати потрібно наші “мінімальні” функції

```
fn f1(a: A) -> B
```

Припустимо $A = (i32, i32)$, $B = i32$
Тобто наша функція

```
fn f1(a: i32, b: i32) -> i32 {  
    a + b  
}
```


Тестування

Потрібно перевірити, що наша функція повертає правильні результати для всіх(*) вхідних значень.

Виходить так, що нам потрібні додаткові точки входу в програму, які будуть запускати наші тести.

Rust надає такі можливості.

Анотація #[test]

1. Ми маємо додаткові точки входу в програму.
2. Ми бачимо гарну підтримку IDE. Можемо одним "кліком" запустити тест.
3. Маємо можливість запускати один або декілька, або всі тести одним "кліком".



The image shows a dark-themed code editor with two snippets of Rust code. Each snippet is preceded by a green right-pointing triangle, which is a common IDE icon for running a test. The first snippet is:

```
#[test]
fn test_1() {
    // ...
}
```

The second snippet is:

```
#[test]
fn test_2() {
    // ...
}
```

Перевірка значень

```
fn f1(a: i32, b: i32) -> i32 {  
    a + b  
}
```



```
#[test]  
fn test_1() {  
    let a: i32 = 3;  
    let b: i32 = 5;  
    let real: i32 = f1(a, b);  
    let expected: i32 = 8;  
  
    assert_eq!(real, expected);  
}
```


Перевірка значень

```
fn f1(a: i32, b: i32) -> i32 {  
    a + b  
}
```



```
#[test]  
fn test_1() {  
    let a: i32 = 3;  
    let b: i32 = 5;  
    let real: i32 = f1(a, b);  
    let expected: i32 = 9;  
  
    assert_eq!(real, expected);  
}
```

Перевірка panic

```
✓ fn div(a: i32, b: i32) -> i32 {  
    a / b  
}
```

```
#[test]
```



```
✓ fn test_div() {  
    let x: i32 = div(a: 3, b: 0);  
    assert_eq!(x, panic)  
}
```


Перевірка panic

```
fn div(a: i32, b: i32) -> i32 {  
    a / b  
}  
  
#[test]  
#[should_panic]  
fn test_div() {  
    let x: i32 = div(a: 3, b: 0);  
}
```




Тимчасове вимикання тестів

Наприклад, ми працюємо над тестом, якій

- ще не закінчений
- імплементація не закінчена
- чомусь "ламається"
-

Нам потрібно вимкнути тест (тимчасово)

Тимчасове вимикання тестів



```
#[ignore]
#[test]
fn test_11b() {
    let a:i32 = 3;
    let b:i32 = 5;
    let real:i32 = f1(a, b);
    let expected:i32 = 9;

    assert_eq!(real, expected);
}
```

Автоматичний запуск всіх тестів

Екосистема Rust дуже розвинена і є можливість запустити всі тести, які є в проекті:

```
→ rust-course git:(master) x cargo test
```

Відповідно тести, анотовані `#[ignore]` не будуть виконані

```
test lectures::lec15::t2::main1 ... ok
test lectures::lec15::t2::main2 ... ok
test lectures::lec15::t3::test_11b ... ignored
test lectures::lec14::out2file::test2_write_bin_file1 ... ok
test lectures::lec15::t3::test_11a ... ok
```

але ми про це будемо проінформовані

```
test result: ok. 230 passed; 0 failed; 12 ignored;
```


Висновки

Тобто тестування, це:

1. Також написання коду, який буде виконувати наш “основний” код.
2. Порівняння “реальних” та “очікуваних” результатів роботи програми.

Але є нюанс...

Як тестувати ввід інформації з клавіатури?

Як тестувати вивід на екран?

Як тестувати запис/читання файлу?

Як тестувати відправку e-mail?

Як тестувати http-запит на інший сервер?

Як тестувати речі пов'язані з таймером?

...

Як тестувати будь-які речі, які мають "ефект"

- за кордоном нашої програми

- в операційній системі

- на іншому сервері

...

Висновки

Розробка програмного забезпечення не обмежується лише написанням коду. Тестування є важливим етапом, який допомагає гарантувати коректність, продуктивність та безпеку програми. Чим раніше ми починаємо тестування, тим швидше можемо виявити потенційні проблеми, уникнути помилок та зробити код більш надійним.

Екосистема Rust. Бібліотеки

Екосистема. Бібліотеки 1/3

Сучасна розробка програмного забезпечення рідко починається з чистого аркуша. У більшості випадків програмісти використовують бібліотеки – готові набори функцій, класів та інструментів, які значно спрощують процес створення програм. Це дозволяє швидше вирішувати бізнес-завдання та зосереджуватися на розробці нової функціональності, а не на повторному створенні вже існуючих рішень.

Екосистема. Бібліотеки 2/3

Однією з головних переваг бібліотек є те, що вони зазвичай широко використовуються в спільноті розробників, що означає їхню перевіреність та стабільність. Багато бібліотек проходять ретельне тестування як розробниками, так і великою кількістю користувачів. Це значно зменшує ймовірність критичних помилок і дозволяє створювати більш надійні програми.

Також використання бібліотек сприяє стандартизації коду. Коли розробники застосовують популярні рішення, їхній код стає більш зрозумілим для інших програмістів, що полегшує підтримку та розвиток проєктів. Замість того щоб писати власні алгоритми для роботи з базами даних, веб-запитами або шифруванням, можна використати бібліотеки, які вже вирішують ці завдання ефективно та безпечно.

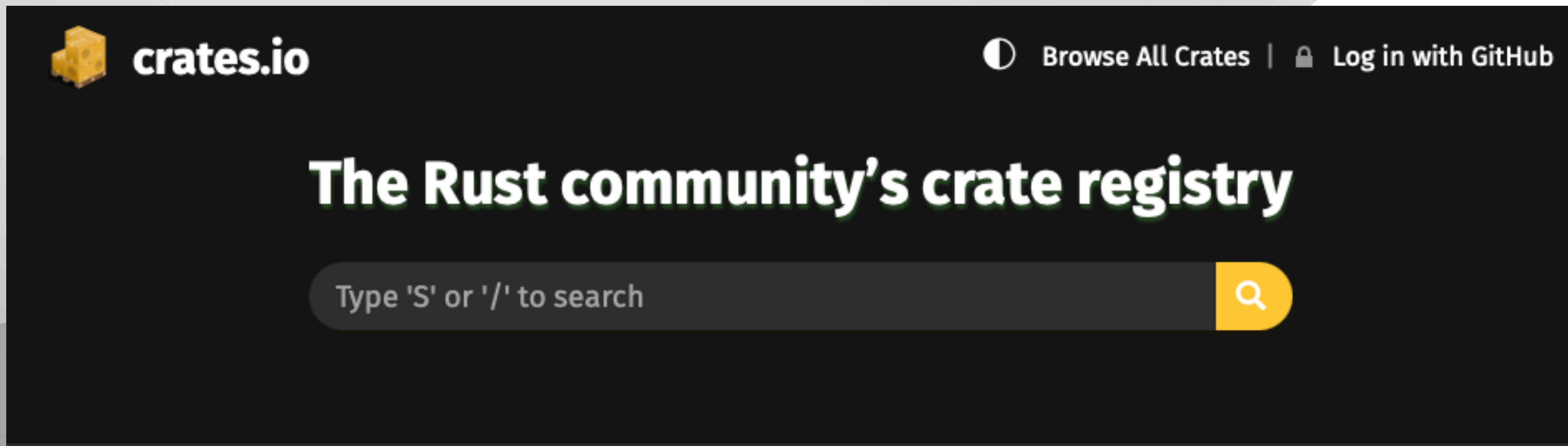
Екосистема. Бібліотеки 3/3

Також використання бібліотек сприяє стандартизації коду. Коли розробники застосовують популярні рішення, їхній код стає більш зрозумілим для інших програмістів, що полегшує підтримку та розвиток проєктів. Замість того щоб писати власні алгоритми для роботи з базами даних, веб-запитами або шифруванням, можна використати бібліотеки, які вже вирішують ці завдання ефективно та безпечно. Однак важливо правильно підходити до вибору бібліотек. Варто звертати увагу на їхню популярність, активність розробки, наявність документації та тестів. Застарілі або маловідомі бібліотеки можуть містити вразливості або бути недостатньо оптимізованими.

Таким чином, використання бібліотек – це не лише спосіб прискорити розробку, але й засіб забезпечити стабільність, безпеку та якість програмного забезпечення. Завдяки ним ми можемо фокусуватися на вирішенні бізнес-завдань, замість того щоб щоразу винаходити колесо.

Rust Crates: єдине місце для бібліотек

У мові програмування Rust екосистема бібліотек організована через crates.io – офіційний репозиторій Rust-бібліотек, або "крейтів". Це єдине місце, де можна знайти, оцінити та використати вже готові рішення для своєї програми.



Основні особливості crates.io:

Зручний пошук – можна шукати бібліотеки за назвою, ключовими словами або категоріями.

Рейтинг та завантаження – кожен крейт має кількість завантажень, що відображає його популярність та довіру з боку спільноти.

Документація – більшість популярних крейтів мають автоматично згенеровану документацію через docs.rs, що дозволяє швидко розібратися з API.

Система версій – дозволяє вибирати стабільні або найновіші версії залежно від потреб проєкту.

Як додавати бібліотеки в Rust

Всі бібліотеки декларуються у файлі **cargo.toml**

```
8 [dependencies]
9  colored = "3.0.0" 3.0.0
10 itertools = "0.14.0" 0.14.0
11 rand = "0.9.0" 0.9.0
12 sysinfo = "0.33.1" 0.33.1
13 request = { version = "0.12.12" 0.12.12, features = ["blocking", "json"] }
14 tokio = { version = "1" 1.43.0, latest: 1.44.0, features = ["full"] }
15 tokio-postgres = "0" 0.7.13
16 time = "0.3.37" 0.3.37, latest: 0.3.39
17 regex = "1.11.1" 1.11.1
18 chrono = "0.4.39" 0.4.39, latest: 0.4.40
```

Висновки

Завдяки централізованій екосистемі, Rust-розробники можуть швидко знаходити та використовувати перевірені рішення, що дозволяє зосередитися на бізнес-логіці, а не на написанні базової інфраструктури з нуля.

Висновки

Сучасні технології дозволяють значно спростити розробку завдяки використанню бібліотек і фреймворків. Однак їх інтеграція вимагає додаткового тестування, оскільки помилки можуть виникати не тільки в нашому власному коді, а й у взаємодії з зовнішніми компонентами. Тому відповідальність програміста полягає не тільки у створенні робочого коду, а й у забезпеченні його якісного тестування.

Висновки

Якісне тестування допомагає не лише знаходити помилки, а й робить код більш підтримуваним, масштабованим та зрозумілим. Це запорука успішного програмного продукту, який працюватиме стабільно, навіть у непередбачуваних умовах. Отже, тестування – це не додаткова робота, а невід’ємна частина процесу програмування, що визначає успіх будь-якого проекту.

**Код з лекцій,
презентації Keypnote,
PDF-файли
знаходяться на GitHub:**

<https://github.com/djnzx/rust-course>
<git@github.com:dnzxr/rust-course.git>