

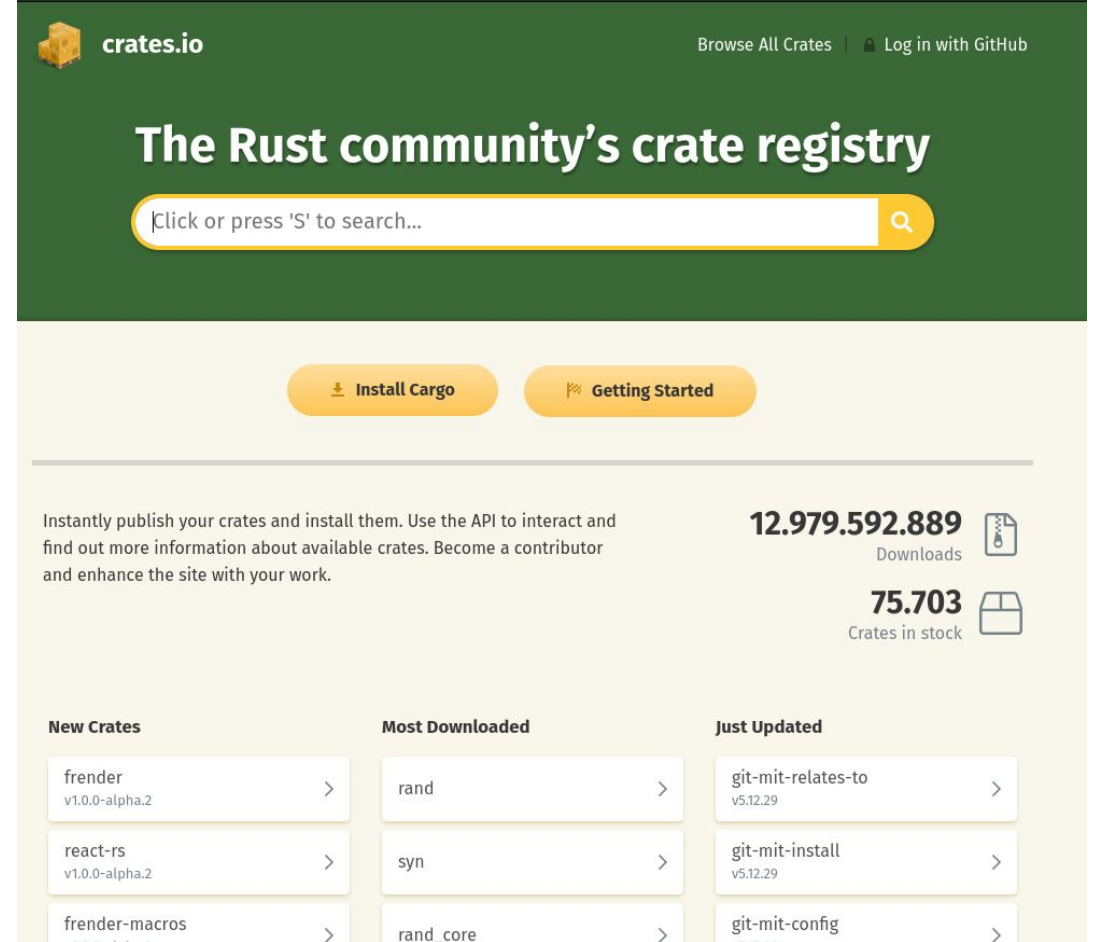
Rust: Crate, Test

Hafta - 3

Crate

Projenize dahil edebileceğiniz Rust kütüphaneleridir.

Crate'lerin bulunduğu adres: crates.io



The screenshot shows the crates.io website, which is the Rust community's crate registry. The header is dark green with the crates.io logo and navigation links. Below the header is a large search bar with a yellow border and a magnifying glass icon. Two yellow buttons, "Install Cargo" and "Getting Started", are positioned below the search bar. The main content area has a light beige background. It features a section with statistics: "12.979.592.889 Downloads" and "75.703 Crates in stock". Below this, there are three columns of crate listings: "New Crates", "Most Downloaded", and "Just Updated". Each listing shows the crate name, version, and a right arrow icon.

crates.io Browse All Crates | Log in with GitHub

The Rust community's crate registry

Click or press 'S' to search...

[Install Cargo](#) [Getting Started](#)

Instantly publish your crates and install them. Use the API to interact and find out more information about available crates. Become a contributor and enhance the site with your work.

12.979.592.889 Downloads

75.703 Crates in stock

New Crates	Most Downloaded	Just Updated
frender v1.0.0-alpha.2	rand	git-mit-relates-to v5.12.29
react-rs v1.0.0-alpha.2	syn	git-mit-install v5.12.29
frender-macros	rand_core	git-mit-config

Cargo.toml olmadan kütüphane oluşturma

kutuphane.rs:

```
pub fn public_fonksiyon() {  
    println!("Kütüphanemin public_fonksiyon'u çalıştı");  
}
```

Terminal

```
$ rustc --crate-type=lib kutuphane.rs  
$ ls lib*  
libkutuphane.rlib
```

Kütüphane dosyamız "**libkutuphane.rlib**" ismiyle oluşturuldu.

Dışarıdan bir .rlib dosyasını dahil etme

main.rs:

```
fn main() {  
    kutuphane::public_fonksiyon();  
}
```

Terminal

```
$ rustc main.rs --extern kutuphane=libkutuphane.rlib  
$ ./main  
Kütüphanemin public_fonksiyon'u çalıştı
```

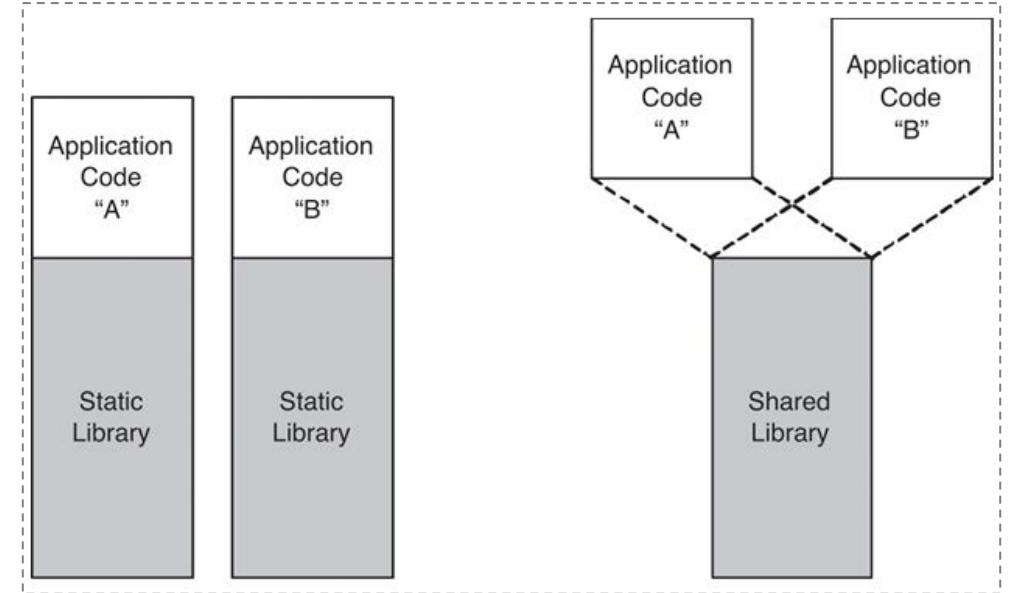
Rust ile oluşturulabilen kütüphaneler

Statik Kütüphaneler (Derleme zamanında programın içine gömülen)

- **.rlib:**
- **.a**
- **.lib:**

Dinamik Kütüphaneler (Çalışma zamanında ortak bellek alanına yüklenen)

- **.dll**
- **.so**
- **.dylib:**



Cargo ile bir kütüphane (crate) oluşturma

```
$ cargo init --lib
```

```
$ tree
```

```
.  
├── Cargo.toml  
└── src  
    └── lib.rs
```

```
1 directory, 2 files
```

```
$ tree
```

```
.  
├── Cargo.toml  
└── src  
    └── lib.rs
```

----> lib.rs:

```
pub fn public_fonksiyon() {  
    println!("Kütüphanemin public_fonksiyon'u çalıştı");  
}  
  
fn private_fonksiyon() {  
    println!("Kütüphanemin private_fonksiyon'u çalıştı");  
}
```

Terminal

```
$ cargo build --release  
$ ls ./target/release/lib*  
libcrateornek.d  
libcrateornek.rlib
```

Cargo ile Crate oluşturma

Kütüphanemizin projesini, programımızın projesi içine taşıyalım:

```
$ tree
.
├── Cargo.toml
├── crateornek
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
└── src
    └── main.rs

3 directories, 4 files
```


Cargo ile Crate oluşturma

```
$ tree
```

```
.
├── Cargo.toml
├── crateornek
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
└── src
    └── main.rs
```

--->

Cargo.toml:

```
[package]
name = "crate-kullanan-ornek"
version = "0.1.0"
edition = "2021"

[dependencies]
crateornek = { path = "./crateornek" } // crateornek'i projemize yerel diziniyle ekledik.
```

```
$ tree
```

```
.
├── Cargo.toml
├── crateornek
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
└── src
    └── main.rs
```

--->

Cargo.toml:

```
[package]
name = "crateornek"
version = "0.1.0"
edition = "2021"

[dependencies]
```

Cargo ile Crate oluşturma

main.rs:

```
use crateornek;  
  
fn main() {  
    crateornek::public_fonksiyon();  
}
```

\$ cargo run

```
Compiling crateornek v0.1.0  
(/home/ef/Rust/crate-kullanan-ornek/crateornek)  
Compiling crate-kullanan-ornek v0.1.0 (/home/ef/Rust/crate-kullanan-ornek)  
  
Finished dev [unoptimized + debuginfo] target(s) in 0.93s  
Running `target/debug/crate-kullanan-ornek`
```

Kütüphanemin public_fonksiyon'u çalıştı

Cargo ile Crate oluşturma

\$ tree

```
.
├── Cargo.toml
├── crateornek
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
└── src
    └── main.rs
```

--->

lib.rs:

```
pub mod ara_modul {
    pub fn public_fonksiyon() {
        println!("Kütüphanemin public_fonksiyon'u çalıştı");
    }

    fn private_fonksiyon() {
        println!("Kütüphanemin private_fonksiyon'u çalıştı");
    }
}
```

\$ tree

```
.
├── Cargo.toml
├── crateornek
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
└── src
    └── main.rs
```

--->

main.rs:

```
use crateornek::ara_modul;

fn main() {
    ara_modul::public_fonksiyon();
}
```

\$ cargo run

Kütüphanemin public_fonksiyon'u çalıştı

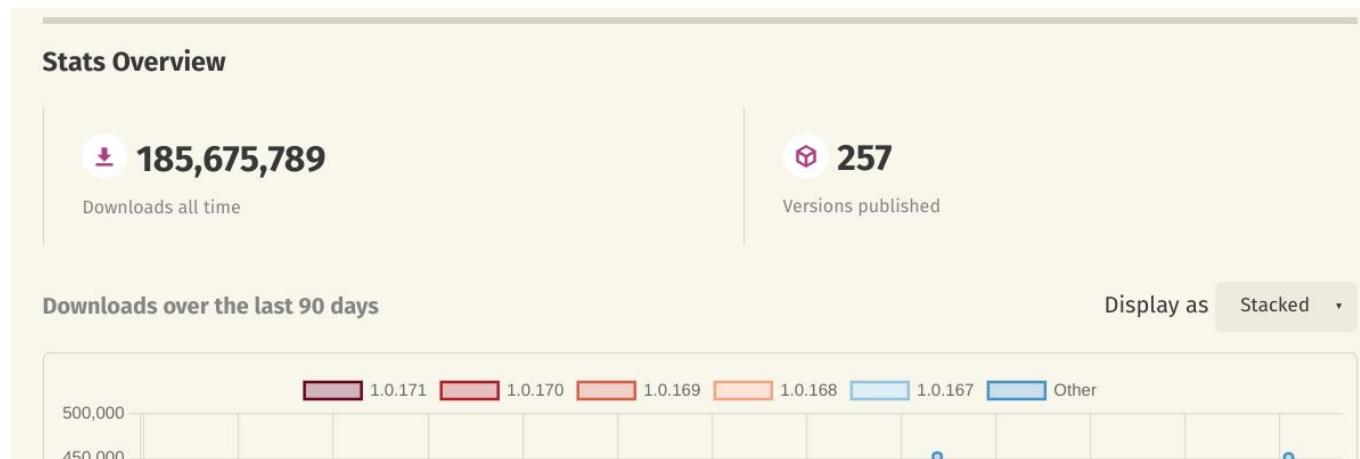
Örnek crate: "serde"

 **serde** 1.0.136

A generic serialization/deserialization framework

#serde #serialization #no_std

<https://crates.io/crates/serde>



Metadata

📅 5 days ago

🔖 v1.19.0

📄 MIT OR Apache-2.0

📦 77.6 kB

Install

Run the following Cargo command in your project directory:

```
cargo add serde
```

Or add the following line to your Cargo.toml:

```
serde = "1.0.171"
```

Homepage

serde.rs

Documentation

docs.rs/serde/1.0.171

Repository

github.com/serde-rs/serde

Örnek crate: "serde"

```
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    // Point struct'ını JSON stringine çevir
    let serialized = serde_json::to_string(&point).unwrap();

    // serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    // JSON string'inden Point struct'ı oluşturur
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    // deserialized = Point { x: 1, y: 2 }
    println!("deserialized = {:?}", deserialized);
}
```

Tests

Testler yazdığımız programın doğru çalışıp çalışmadığını test ettiğimiz kodlardır.

```
$ tree
.
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── testim.rs
```

2 directories, 4 files

Cargo.toml

```
[package]
name = "test-ornek"
version = "0.1.0"
edition = "2021"
```

```
[dependencies]
```

Tests

```
$ tree
```

```
.
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── testim.rs
```

lib.rs:

```
pub mod modul {
    pub fn sayi() -> i32 {
        42
    }
}
```

```
$ tree
```

```
.
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── testim.rs
```

testim.rs:

```
#[cfg(test)]
mod tests {
    use test_ornek::modul;

    #[test]
    fn test_42() {
        assert_eq!(modul::sayi(), 42);
    }
}
```

Tests

```
$ tree
```

```
.  
├── Cargo.toml  
├── src  
│   └── lib.rs  
└── tests  
    └── testim.rs ---->
```

```
testim.rs:
```

```
#[cfg(test)]  
mod tests {  
    use test_ornek::modul;  
  
    #[test]  
    fn test_42() {  
        assert_eq!(modul::sayi(), 42);  
    }  
}
```

```
$ cargo test
```

```
Running tests/testim.rs (target/debug/deps/testim-4cd276d703205a1f)
```

```
running 1 test
```

```
test tests::test_42 ... ok
```


Sıra Sizde

1. `Person { name: String, age: u8, gender: Gender }`
 - Bu struct'ı `serde` ve `serde_json` ile JSON çıktısı haline getirin. Örn: `{"name":"Ahmet","age":18,"gender":"Male"}`
2. Birinci taskın düzgün çalıştığını kontrol eden 3 adet test yazın.
3. JSON haline getirilmiş struct'ı bir `kisi.json` dosyasına yazın (`Serialize`)
4. `kisi.json` dosyasını okuyup JSON metnini tekrar bir struct haline getirin (`Deserialize`)

Yardımcı linkler:

- <https://doc.rust-lang.org/std/fs/struct.File.html>