Starting with Rust Idioms

Tomáš Jašek

March 7, 2023

Tomáš Jašek

- Rust backend engineer at Sonalake
- LinkedIn, Github

Idiom

Wikipedia:

group of code fragments sharing an equivalent semantic role which recurs frequently across software projects

Why?

- reduce cognitive overhead
- simplify code review
- increase likelyhood of spotting mistakes
- compiler may focus its optimizations on idiomatic code

Groups of Idioms to be Discussed

1. Functional approach with Option & Result
2. Iterator
3. Type Conversions

Part 1: Functional Approach with Option & Result

Part 1: Functional Approach with Option & Result

Introduction

```rust
enum Option<T> {
    Some(T),
    None,
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Part 1: Functional Approach with Option & Result

Motivating Example

- implement `fn (Request) -> parse::Result<String>`
- if parsing failed, return error
- if parseable_user or display_name is not present, return empty string

```rust
struct Request {
    parseable_user: Option<ParseableUser>,
}

struct ParseableUser { ... }

impl ParseableUser {
    fn parse(self) -> parse::Result<User> { ... }
}
```

```rust
struct User {
    display_name: Option<String>,
}

pub mod parse {
    pub struct Error;
    pub type Result<T> = Result<T, Error>;
}
```

Part 1: Functional Approach with Option & Result

Possible Solution

- implement `fn (Request) -> parse::Result<String>`
- if parsing failed, return error
- if parseable_user or display_name is not present, return empty string

```rust
pub fn get_user_display_name(req: Request) -> Result<String> {
    if let Some(parseable_user) = req.parseable_user {
        if let Ok(user) = parseable_user.parse() {
            if let Some(display_name) = user.display_name {
                Ok(display_name)
            } else {
                Ok("".to_string())
            }
        } else {
            Err(parse::Error {})
        }
    } else {
        Ok("".to_string())
    }
}
```

Part I: Functional Approach with Option & Result

Towards functional Approach

- implement `fn (Request) -> parse::Result<String>`
- if parsing failed, return error
- if parseable_user or display_name is not present, return empty string

```
pub fn get_user_display_name(req: Request) -> Result<String> {
    if let Some(parseable_user) = req.parseable_user {
        if let Ok(user) = parseable_user.parse() {




            Ok(user.display_name.unwrap_or("".to_string()))
        } else {
            Err(parse::Error {})
        }
    } else {
        Ok("".to_string())
    }
}
```

[Option::unwrap_or]

Part 1: Functional Approach with Option & Result

“Practically readable” functional approach

- implement `fn (Request) -> parse::Result<String>`
- if parsing failed, return error
- if parseable_user or display_name is not present, return empty string

```
pub fn get_user_display_name(req: Request) -> Result<String> {
    if let Some(parseable_user) = req.parseable_user {
        let user = parseable_user.parse()?;




        Ok(user.display_name.unwrap_or("".to_string()))



    } else {
        Ok("".to_string())
    }
}
```

[question mark]

Part I: Functional Approach with Option & Result

Bonus: Fully functional approach

- implement `fn (Request) -> parse::Result<String>`
- if parsing failed, return error
- if parseable_user or display_name is not present, return empty string

```rust
pub fn get_user_display_name(req: Request) -> Result<String> {
    Ok(req
        .parseable_user
        .map(|pu| pu.parse())
        .transpose()?
        .and_then(|u| u.display_name)
        .unwrap_or("".to_string()))
}
```

Important: does this reduce cognitive overhead for your team?

- `Option::map`
- `Option::transpose`
- `Option::and_then`
- `Option::unwrap_or`

Part 2: Iterator

Part 2: Iterator

Introduction

- returns elements of a collection, allows transforming them
- iterators usually implement [`std::iter::Iterator`]
- collection to iterator
  - *`std::iter::IntoIterator::into_iter` - consumes a collection*
  - *alternative: `iter()` - an immutable reference to an element*
  - *alternative: `iter_mut()` - a mutable reference to an element*
- iterator to collection
  - *`std::iter::Iterator::collect`*
- antipattern: `vec.iter().collect().iter().collect()`
  - ***only use collect to store new collection into memory!***
  - *intermediate results should be passed as iterators*

Part 2: Iterator

Antipattern: iter -> collect -> iter -> collect

```rust
pub fn get_nonempty(vec: &Vec<String>) -> Vec<&String> {
    vec.iter().filter(|x| !x.is_empty()).collect()
}

fn transform_vec(vec: &Vec<String>) -> Vec<String> {
    get_nonempty(vec)
        .iter()
        .map(|x| format!("{} ", x))
        .collect()
}

let vec = vec![];
let vec = transform_vec(&vec);
```

Part 2: Iterator

Antipattern FIX: iter -> collect -> iter -> collect

```rust
pub fn get_nonempty(vec: &Vec<String>) -> impl Iterator<Item = &String> {
    vec.iter().filter(|x| !x.is_empty())
}

fn transform_vec(vec: &Vec<String>) -> Vec<String> {
    get_nonempty(vec)
        .map(|x| format!("{} ", x))
        .collect()
}

let vec = vec![];
let vec = transform_vec(&vec);
```

Part 2: Iterator

Example 1: Contest results

Prepare results sheet for a `Contest`. For each contestant, print their score and their name.

```
struct Contestant {
    name: String,
    score: usize,
}

struct Country {
    name: String,
    contestants: Vec<Contestant>,
}

struct Contest {
    countries: Vec<Country>,
}
```

Part 2: Iterator

Example 1: Contest results

Prepare results sheet for a `Contest`. For each contestant, print their score and their name.

```rust
fn results_list(contest: Contest) -> Vec<String> {
    let mut result = vec![];
    for country in &contest.countries {
        for contestant in &country.contestants {
            result.push(format!("{} | {}", contestant.score, contestant.name));
        }
    }
    result
}
```

Part 2: Iterator

Example 1: Contest results

Prepare results sheet for a `Contest`. For each contestant, print their score and their name.

```
fn results_list(contest: Contest) -> Vec<String> {
    let mut result = vec![];
    let contestants = contest.countries.iter().flat_map(|country| country.contestants);
    for contestant in contestants {
        result.push(format!("{} | {}", contestant.score, contestant.name));
    }
    result
}
```

Idiom: [`Iterator::flat_map`]

Notice: hybrid approach – prepare elements in functional way, iterate using `for`

Part 2: Iterator

Example 1: Contest results

Prepare results sheet for a `Contest`. For each contestant, print their score and their name.

```
fn results_list(contest: Contest) -> Vec<String> {
    contest.countries
        .iter()
        .flat_map(|country| country.contestants.iter())
        .map(|contestant| format!("{} | {}", &contestant.score, &contestant.name))
        .collect()
}
```

Idiom: [`Iterator::collect`]()

Part 2: Iterator

Example 1: Contest results

Prepare results sheet for a `Contest`. For each contestant, print their score and their name.

```
fn results_list(contest: Contest) -> impl Iterator<Item = String> {
    contest.countries
        .iter()
        .flat_map(|country| country.contestants.iter())
        .map(|contestant| format!("{} | {}", &contestant.score, &contestant.name))
}
```

Idiom: return iterator

Part 2: Iterator

Example 2: Matrix transpose

Given a matrix $A = (a_{i,j})$, produce a transposed matrix. $A^T = (a_{j,i})$. Example:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

```
struct Matrix {
    rows: Vec<Vec<isize>>,
    row_count: usize,
    col_count: usize,
}
```

- Note: In practice, Matrices are represented using `Vec<isize>` to reduce count of allocations

Part 2: Iterator

Example 2: Matrix transpose

Given a matrix $A = (a_{i,j})$, produce a transposed matrix. $A^T = (a_{j,i})$.

```
impl Matrix {
    fn transpose(&self) -> Matrix {
        let mut result = Matrix::zero(self.col_count, self.row_count);

        for row in 0..self.row_count {
            for col in 0..self.col_count {
                result.set_element_at(col, row, self.element_at(row, col));
            }
        }

        result
    }
}
```

Part 2: Iterator

Example 2: Matrix transpose

Given a matrix $A = (a_{i,j})$, produce a transposed matrix. $A^T = (a_{j,i})$.

```
impl Matrix {
    fn transpose_iterators(&self) -> Matrix {
        let rows = (0..self.col_count)
            .map(|ncol| {
                self.rows
                    .iter()
                    .map(|row| row[ncol])
                    .collect::<Vec<isize>>()
            })
            .collect();
        Matrix {
            rows,
            row_count: self.col_count,
            col_count: self.row_count,
        }
    }
}
```

Pros: might be more performant

Cons: cognitive overhead, detailed documentation is required

Part 3: Type Conversions

Part 3: Type Conversions

From/Into: Infallible

- infallible conversion
- non-async: just data rearrangement without complex business logic
- processing nested structures one at a time
  - *each impl is concerned with exactly one type*

Part 3: Type Conversions

From/Into: Infallible

- From & Into are dual
- implementing From gives us Into for free

We write

```rust
impl From<Wood> for Paper { ... }
```

Rust implements

```rust
impl Into<Paper> for Wood { ... }
```

Part 3: Type Conversions

From/Into: Infallible

- From & Into are dual
- implementing From gives us Into for free

We write	Rust implements
`impl From<Wood> for Paper { ... }`	`impl Into<Paper> for Wood { ... }`
`impl Into<Paper> for Wood { ... }`	**N/A**

Part 3: Type Conversions

From/Into: Infallible

Example: Convert between representations

```rust
struct Birthday(Date);
struct FullPerson {
    name: String,
    birthday: Birthday,
    birth_number: String,
    address: FullAddress,
}

struct FullAddress {
    street_number: String,
    street: String,
    city: String,
    country: String,
}
```

```rust
struct Age(Duration);
struct PartialPerson {
    name: String,
    age: Age,
    address: PartialAddress,
}

struct PartialAddress {
    city: String,
    country: String,
}
```

Part 3: Type Conversions

From/Into: Infallible

Example: Convert between representations

```rust
impl From<FullPerson> for PartialPerson {
    fn from(fp: FullPerson) -> Self {
        Self {
            name: fp.name,
            age: fp.birthday.into(),
            address: fp.address.into()
        }
    }
}

impl From<FullAddress> for PartialAddress {
    fn from(fa: FullAddress) -> Self {
        Self {
            city: fa.city,
            country: fa.country,
        }
    }
}
```

```rust
impl From<Birthday> for Age {
    fn from(bd: Birthday) -> Self {
        Self(SystemTime::now() - bd.0)
    }
}
```

Part 3: Type Conversions

TryFrom/TryInto: Fallible

- very similar to `From`
- fallible conversion
- non-async: just data rearrangement without complex business logic
- processing nested structures one at a time
  - *each impl is concerned with exactly one type*

Part 3: Type Conversions

TryFrom/TryInto: Fallible

Example: Users must be at least 13 years old to register.

```
struct User {
    name: String,
    age: usize,
}
```

Part 3: Type Conversions

TryFrom/TryInto: Fallible

Example: Users must be at least 13 years old to register.

```
struct User {
    name: String,
    age: usize,
}

struct UserIneligible;

fn register(u: User) -> Result<(), UserIneligible> {
    if u.age < 13 {
        return Err(UserIneligible);
    }

    // TODO: insert user into database

    Ok(())
}
```

Part 3: Type Conversions

TryFrom/TryInto: Fallible

Example: Users must be at least 13 years old to register.

```rust
struct User {
    name: String,
    age: usize,
}

struct UserIneligible;


fn register(u: User) -> Result<(), UserIneligible> {
    let u = EligibleUser::try_from(u)?;

    // TODO: insert user into database

    Ok(())
}
```

```rust
struct EligibleUser(User);

impl TryFrom<User> for EligibleUser {
    type Error = UserIneligible;

    fn try_from(u: User) -> Result<Self, Self::Error> {
        if u.age < 13 {
            return Err(UserIneligible);
        }

        Ok(Self(u))
    }
}
```

Part 3: Type Conversions

FromStr

- fallible conversion from string
- counterpart: `String::parse`, `str::parse`

Non-empty string:

```rust
struct NonEmptyString(String);

impl FromStr for NonEmptyString {
    type Err = ();

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        if s.empty() {
            return Err(());
        }

        Ok(Self(s.into()))
    }
}

assert_eq!("".parse::<NonEmptyString>(), Err(()));
assert_eq!("hello".parse::<NonEmptyString>(), Ok(NonEmptyString("hello".into())));
```

Part 3: Type Conversions

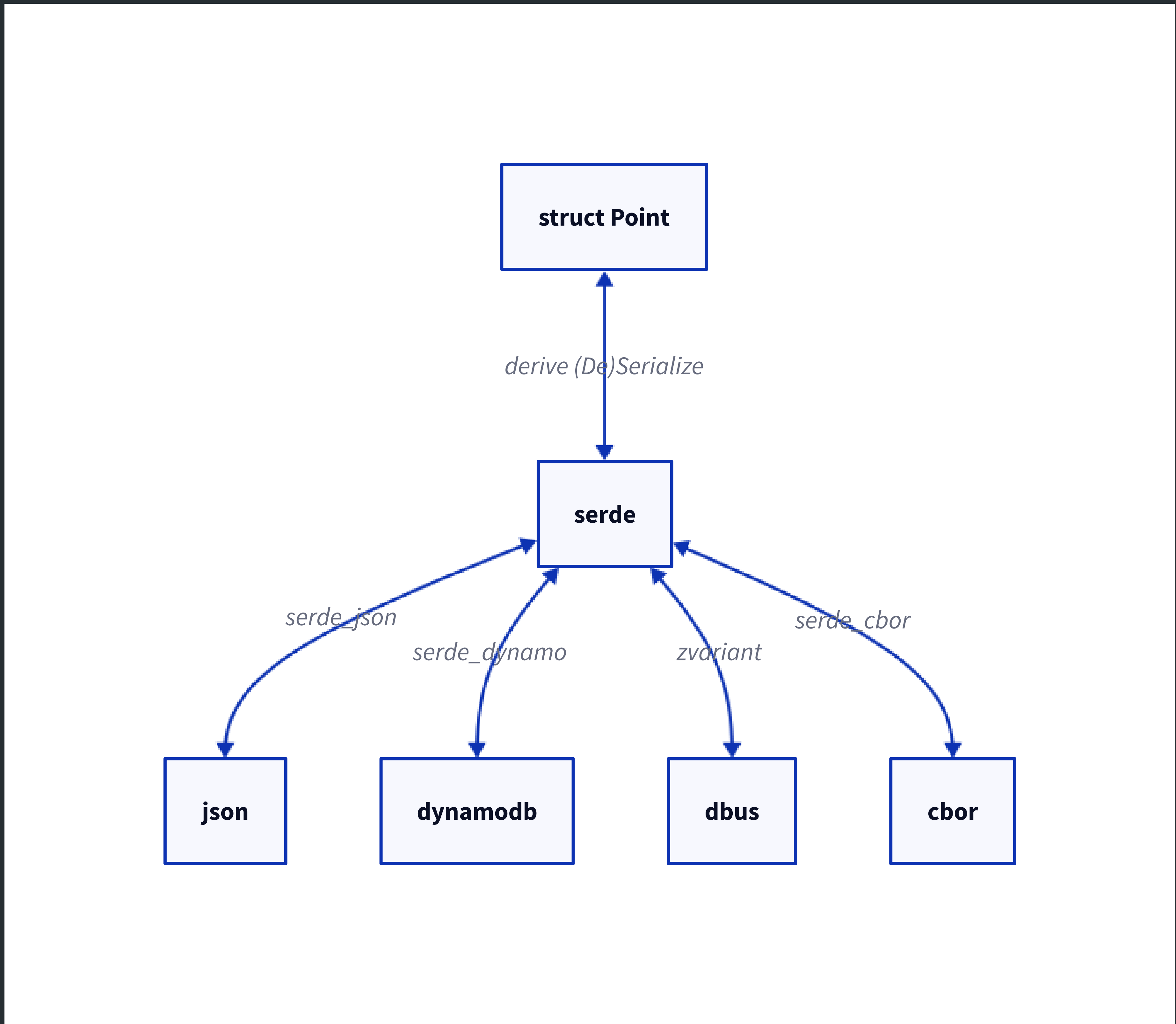
Useful external crates: Serde

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

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

    // Convert the Point to a JSON string.
    let serialized = serde_json::to_string(&point).unwrap();

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

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

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

Credit: serde docs

Part 3: Type Conversions

Useful external crates: [Serde]()



serde

Part 3: Type Conversions

Useful external crates: Strum

```rust
#[derive(Debug, PartialEq, EnumString)]
enum Color {
    Red,
    Green {
        range: usize,
    },

    #[strum(serialize = "blue", serialize = "b")]
    Blue(usize),

    #[strum(disabled)]
    Yellow,

    #[strum(ascii_case_insensitive)]
    Black,
}
```

```rust
let color_variant = Color::from_str("Red").unwrap();
assert_eq!(Color::Red, color_variant);
```

Credit: strum docs

Summary

- do learn about idioms
- do use idiomatic code
- do not blindly apply idioms, consider goals
  - *cognitive overhead*
  - *optimization*