Effective `serde`°

By Writing Less Rust Code

Topics on the Rust Programming Language

Vancouver Rust meetup, 17 April 2019

Daniel J. Pezely
`dpezely` on GitHub, GitLab, Linkedin

° "serde" means ***ser***ialize / ***de***serialize
and is the name of a Rust crate

“My deep hierarchy of data structures is too complicated for auto-conversion.”

--someone *not* using serde

Contents:

Of course, all this applies to far more than just JSON

But JSON is easier for presentation purposes here

Take time to read
https://serde.rs/ *entirely*

before jumping into API docs at
https://crates.io/crates/serde

You'll find it time well-invested!

Spoilers: it's resolved entirely at compile-time, and
without run-time “reflection” mechanisms

The Way Of Serde

Let serde give you superpowers by relying upon:

I. Decorate structs & enums with attributes

II. Write methods of auto-convert traits

III. Coalesce errors via `?` operator

Bonus: Deep or mixed structures? Easy!

I. Decorate structs with attributes

- *Container* attributes for `struct` or `enum` declaration
- *Variant* attributes for each variant of an `enum`
- *Field* attributes for individual `struct` field or within `enum` variant

  See serde.rs/attributes.html

The Way Of Serde

II. Write methods of auto-convert traits

- If writing code handling common patterns:
  - That's probably the wrong approach!
- If writing code to handle name or value conversions:
  - That's probably the wrong approach!
- If checking for existence of nulls or special values:
  - That's probably the wrong approach!

The Way Of Serde

III. Coalesce errors via `?` operator

- Make aggressive use of `?` operator
    - e.g., use `Result` and `ErrorKind` together
- Implement various methods of `From` and `Into` traits
    - compiler reveals exactly what you need
    - so this becomes fairly straight-forward plug-and-chug
- A common Rust idiom-- not just a `serde` thing

The Way Of Serde

IV. Deep or mixed structures? Easy!

- Populate a nested `enum` and their variants from a flattened set
    - i.e., each variant must map to exactly one Enum
    - then, nested Enums may be resolved when decorating with a single attribute
- Ingest minimal data file structures to well-defined structures in Rust
    - e.g., JSON without naming each structural component
    - where keys contain data (*NOT* name of struct)
- Thus, have your idiomatic Rust cake and eat minimalist data files too!

2. A Realistic Example:

a) Each entry may have multiple categories

b) Given as a flattened set in JSON

c) Expand to well-defined structs in Rust

Unpacking Minimalist JSON

```
{
  "energy-preferences": {
    "2000s": ["solar", "wind"],
    "1900s": ["kerosene", "soy", "peanut", "petroleum"],
    "1800s": ["wind", "whale", "seal", "kerosene"]
  }
}
```

Notable:

- Outer structure is an object (*NOT* an array)
- Top-level keys contain information (*NOT* name of structure)
- Inner values within array indicate mixed categories

Starting From The Top

Serde can handle various naming conventions

e.g., snake_case, camelCase, PascalCase, kebab-case, etc.

```rust
#[derive(Serialize, Deserialize, Debug)]
#[serde(rename_all = "kebab-case")]
struct EnergyPreferenceHistory {
    energy_preferences: EnergyPreferences
}

#[derive(Serialize, Deserialize, Debug)]
struct EnergyPreferences (HashMap<Century, Vec<EnergySources>>);
```

See serde.rs/attributes.html

Particularly, serde.rs/container-attrs.html

Avoid Merging Concepts In An `enum`

```rust
enum EnergySources {  // Don't mix categories like this!
    Solar,
    Wind,
    // ...
    Kerosene,
    Petroleum,
    // ...
    PeanutOil,
    SoyOil,
    // ...
    SealBlubber,
    WhaleBlubber,
    // ...
}
```

It would be more idiomatic Rust

grouping them by category, instead

3. Simple Hierarchy Of Enums

Continuing from previous example...

```
enum EnergySources {
    Sustainable(Inexhaustible),
    Animal(Blubber),
    Vegetable(Crop),
    Mineral(Fossil),
}

enum Inexhaustible { Solar, Wind, /* ... */ }

enum Blubber { Seal, Whale, /* ... */ }

enum Crop { Peanut, Soy, /* ... */ }

enum Fossil { Kerosene, Petroleum, /* ... */ }
```

This is more idomatic Rust

But our data file doesn't look anything like this...
Fear not!

4. Untagged Enums

Decorate With *Attributes*:

Continuing from previous example...

```
#[derive(Serialize, Deserialize, Debug)]
#[serde(untagged)]              // <-- Unflatten from compact JSON
enum EnergySources {
    Sustainable(Inexhaustible),
    Animal(Blubber),
    Vegetable(Crop),
    Mineral(Fossil),
}
```

See "Untagged" section in serde.rs/enum-representations.html

5. Renaming Variants

Pretty JSON and prettier Rust

```rust
#[derive(Serialize, Deserialize, Debug, PartialEq, Eq, Hash)]
enum Century {
    #[serde(rename = "1800s")]
    NinteenthCentury,

    #[serde(rename = "1900s")]
    TwentiethCentury,

    #[serde(rename = "2000s")]
    TwentyfirstCentury
}
```

Each has its preferred naming convention

Additional attributes: `PartialEq`, `Eq`, `Hash`

6. Error Handling

Use `?` early and often:

```
fn main() -> Result<(), ErrorKind> {
    let json_string = fs::read_to_string("energy.json")?;

    let sources: EnergyPreferenceHistory =
        serde_json::de::from_str(&json_string)?;

    println!("{:#?}", sources);
    Ok(())
}
```

Note uses of question mark `?` operator above

Implementing just the above, the compiler helpfully tells you exactly which `impl From` methods to add

Example `ErrorKind`

For Use With `Result` Type

Continuing from previous example...

```rust
#[derive(Debug)]
enum ErrorKind {
    BadJson,
    NoJson,
    NoFilePath,
    Unknown,
}
```

Implementing `From` methods

For Use With `?` Operator

```rust
impl From<serde_json::Error> for ErrorKind {

    fn from(err: serde_json::Error) -> ErrorKind {
        use serde_json::error::Category;

        match err.classify() {

            Category::Io => {
                println!("Serde JSON IO-error: {:?}", &err);
                ErrorKind::NoJson
            }

            Category::Syntax | Category::Data | Category::Eof => {
                println!("Serde JSON error: {:?} {:?}",
                         err.classify(), &err);
                ErrorKind::BadJson
            }

        }
    }
}
```

Other Powerful Features Of `serde` By Writing Less Code

7. Flattening

```
#[derive(Serialize, Deserialize)]
struct CatalogueEntry {
    id: u64,

    #[serde(flatten)]                          // <-- Field Attribute
    description: HashMap<String, String>,
}
```

Would ultimately produce the following JSON representation:

```
{
  "id":     1234,
  "size":   "bigger than a car",
  "weight": "less than an airplane"
}
```

All fields rendered to same level within JSON

See serde.rs/field-attrs.html

Write the preceding item to JSON file

```rust
fn populate_catalogue() -> Result<(), ErrorKind> {
    let id = 1234;

    let mut description = HashMap::new();
    description.insert("size".to_string(),
                       "bigger than a house".to_string());
    description.insert("weight".to_string(),
                       "less than an airplane".to_string());

    let catalogue = vec![CatalogueEntry{id, description}];

    fs::write("foo.json", serde_json::to_string(&catalogue)?)?;
    Ok(())
}
```

Nothing special here

Serde handles iterables-- just implement the trait

8. Asymmetric JSON

Populate fields only when non-null

```
struct Thing {
    pub keyword: String,

    #[serde(default="Vec::new")]   // <-- constructor
    pub attributes: Vec<String>,
}
```

This yields an empty `Vec`

instead of Vec with empty string

without wrapping value with `Option`

Incentive To Read serde.rs:

Borrowing data in a derived impl

When data has already been loaded and memory allocated:

Let your deserialized structs track only references