Mousse Anis Follow

Mar 2 · 4 min read · · Listen

Reading and writing a JSON file in Rust

This article will introduce you to the parsing and writing of a JSON file in RUST.

interchange format, which is easily readable and writable by humans and machines.

JSON has gained popularity over the years due to its simplicity and ease of use with different programming languages. JSON is often used as a prototyping tool to mock data storage needs or directly as data storage.

Here is an example of a JSON file containing some information about **Missy's diet** (If you don't know who Missy is, I invite you to read any of my former articles here. **PS: Please don't tell her that I made public her diet regiment**)

```
{
    "food":[
        {
            "id": 1,
            "name": "steak",
            "missy_comment": "I really like it a lot"
        },
        {
            "id": 2,
            "name": "kibble",
            "missy_comment": "I can live without it"
        },
        {
            "id": 3,
            "name": "chicken",
            "missy_comment": "I really like it a lot"
        }
    ],
    "missy_food_schedule":[
        {
            "date": 20220228,
            "quantity": 1,
            "food": 1,
            "missy_grumpiness": 0
        },
        {
            "date": 20220227,
```

```
            "date": 20220226,
            "quantity": 3,
            "food": 3,
            "missy_grumpiness": 1
        }
    ]
}
```

missy_secrets.json hosted with ❤ by GitHub view raw

Secret information on Missy's daily diet and behavior

As you can see, the file contains a single anonymous object, which contains two arrays, **food,** and **missy_food_schedule.** The **food** array includes all Missy's favorite food items, and **missy_food_schedule** has Missy's diet schedule for the last three days for February 2022 and more top-secret information.

But how can we parse this JSON file to extract some needed information? Update those pieces of information and save our modifications?

In Rust, we can parse a JSON file with two different approaches:

- **A dynamic approach:** the assumption here is that we do not fully understand the data present in the JSON file, so our program will have to check the existence and type of any data fields dynamically.
- **A static approach:** The assumption here is that we fully understand the data present in the JSON file, so we will use deserialization to check for the existence and type of any field.

Dynamic parsing of a JSON file

To dynamically parse a JSON file, we will use the crate **serde_json** (more information here). This crate needs to be added to our **Cargo.toml** under the dependencies section.

```
[package]
name = "medium"
```

```
8  [dependencies]
9  serde_json = "1.0.59"
```

Cargo.toml hosted with ❤ by GitHub view raw

Cargo.toml of the project for dynamic parsing of a JSON file

The program below was created to dynamically parse the file **missy_secrets.json** and double the food quantity consumed by Missy for the last three days of February to reflect a more accurate reality :).

```rust
use serde_json::{Number, Value};

fn main() {
    // Get the filenames from the command line.
    let input_path = std::env::args().nth(1).unwrap();
    let output_path = std::env::args().nth(2).unwrap();

    let mut missy_diet = {
        // Load the first file into a string.
        let text = std::fs::read_to_string(&input_path).unwrap();

        // Parse the string into a dynamically-typed JSON structure.
        serde_json::from_str::<Value>(&text).unwrap()
    };

    // Get the number of elements in the object 'missy_food_schedule'
    let nb_elements = missy_diet["missy_food_schedule"].as_array().unwrap().len();

    for index in 0..nb_elements{
        if let Value::Number(n) = &missy_diet["missy_food_schedule"][index]["quantity"] {
            // Double the quantity for each element in 'missy_food_schedule'
            missy_diet["missy_food_schedule"][index]["quantity"] =
                Value::Number(Number::from_f64(n.as_f64().unwrap() * 2.).unwrap());
        }
    }

    // Save the JSON structure into the other file.
    std::fs::write(
```

main.rs hosted with ❤ by GitHub view raw

main.rs of the project for static parsing of a JSON file

As you can see, we need checks for each element's existence and type before using it. Which can lead to some complex and lengthy lines of code.

Writing and saving a JSON file is a simple operation, as shown from lines 28 to 32 of our example.

To execute this program, we pass as the first parameter the **missy_secrets.json** file, and the second parameter will be the name of the new JSON file (**miss_secrets_dynamic.json**) created after updating the daily quantity of food consumed by Missy.

```
cargo run missy_secrets.json miss_secrets_dynamic.json
```

The outcome of the execution of the program will be the following JSON file:

```
{
  "food": [
    {
      "id": 1,
      "missy_comment": "I really like it a lot",
      "name": "steak"
    },
    {
      "id": 2,
      "missy_comment": "I can live without it",
      "name": "kibble"
    },
    {
      "id": 3,
```


```
    {
      "date": 20220228,
      "food": 1,
      "missy_grumpiness": 0,
      "quantity": 2.0
    },
    {
      "date": 20220227,
      "food": 1,
      "missy_grumpiness": 10,
      "quantity": 4.0
    },
    {
      "date": 20220226,
      "food": 3,
      "missy_grumpiness": 1,
      "quantity": 6.0
    }
  ]
}
```

miss_secrets_dynamic.json hosted with ❤ by **GitHub** **view raw**

JSON file after update following a dynamic parsing

As you may have already realized, the '**quantity**' value has doubled for each item within the object '**missy_food_shcedule**' as expected.

Static parsing a JSON file

We **should** use a static type technique to parse our file when we are sure of the JSON file structure. Static parsing will directly convert the data present in a JSON file into some Rust objects.

To achieve such goal, we will be using the crates **serde** (more info here), **serde_derive** (more info here), and **serde_json**(crate used for the dynamic parsing of a JSON file).

```
[package]
name = "medium"
```


```
[dependencies]
serde = "1.0.117"
serde_derive = "1.0.117"
serde_json = "1.0.59"
```


Cargo.toml of the project for static parsing of a JSON file

Here is the source code to perform a static parsing of the same JSON file and, like before, the program aims to double the quantity of food consumed by Missy.

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

#[derive(Deserialize, Serialize, Debug)]
struct Food {
    id: u32,
    name: String,
    missy_likeness: String,
}

#[derive(Deserialize, Serialize, Debug)]
struct Schedule {
    date: i64,
    quantity: f64,
    food: u32,
    missy_grumpiness: u32,
}

#[derive(Deserialize, Serialize, Debug)]
struct MissyFoodSchedule {
    food: Vec<Food>,
    missy_food_schedule: Vec<Schedule>,
}

fn main() -> Result<(), std::io::Error> {
    let input_path = std::env::args().nth(1).unwrap();
    let output_path = std::env::args().nth(2).unwrap();
```

```
    };

    // Double the quantity for each element in 'missy_food_schedule'
    for index in 0..missy_secrets.missy_food_schedule.len() {
        missy_secrets.missy_food_schedule[index].quantity *= 2.;
    }

    // Save the JSON structure into the output file
    std::fs::write(
        output_path,
        serde_json::to_string_pretty(&missy_secrets).unwrap(),
    )?;

    Ok(())
}
```

main.rs hosted with ❤ by GitHub view raw

main.rs of the project for dynamic parsing of a JSON file

As you may have realized, we created three structs, one for every object type in the **missy_secrets.json** file.

Structs are in Rust the equivalent of classes from an object-oriented programing perspective. Any behavior linked to a struct or type is introduced via Rust's **Traits** (More on this in one of my past articles here).

In our code source, each Struct is preceded by the following attribute:

```
#[derive(Deserialize, Serialize, Debug)]
```

- The **Deserialize** trait is required to parse (that is, read) JSON strings into this Struct.
- The **Serialize** trait is required to format (that is, write) this Struct into a JSON string.
- The **Debug** trait is for printing a Struct on a debug trace.

```
cargo run missy_secrets.json miss_secrets_static.json
```

As you can see, this example is more concise. Creating some Structs to represent each object in the JSON file allows us to directly extract all the data without checking for any element's existence or type.

This approach is more malleable, maintainable, and can be easily extended.

If we change an existing object in our JSON file, we only need to do the same modification in the Struct representing that object in our program.

If we add a new element into our JSON structure, we only need to create a new Struct to represent this new element.

Conclusion

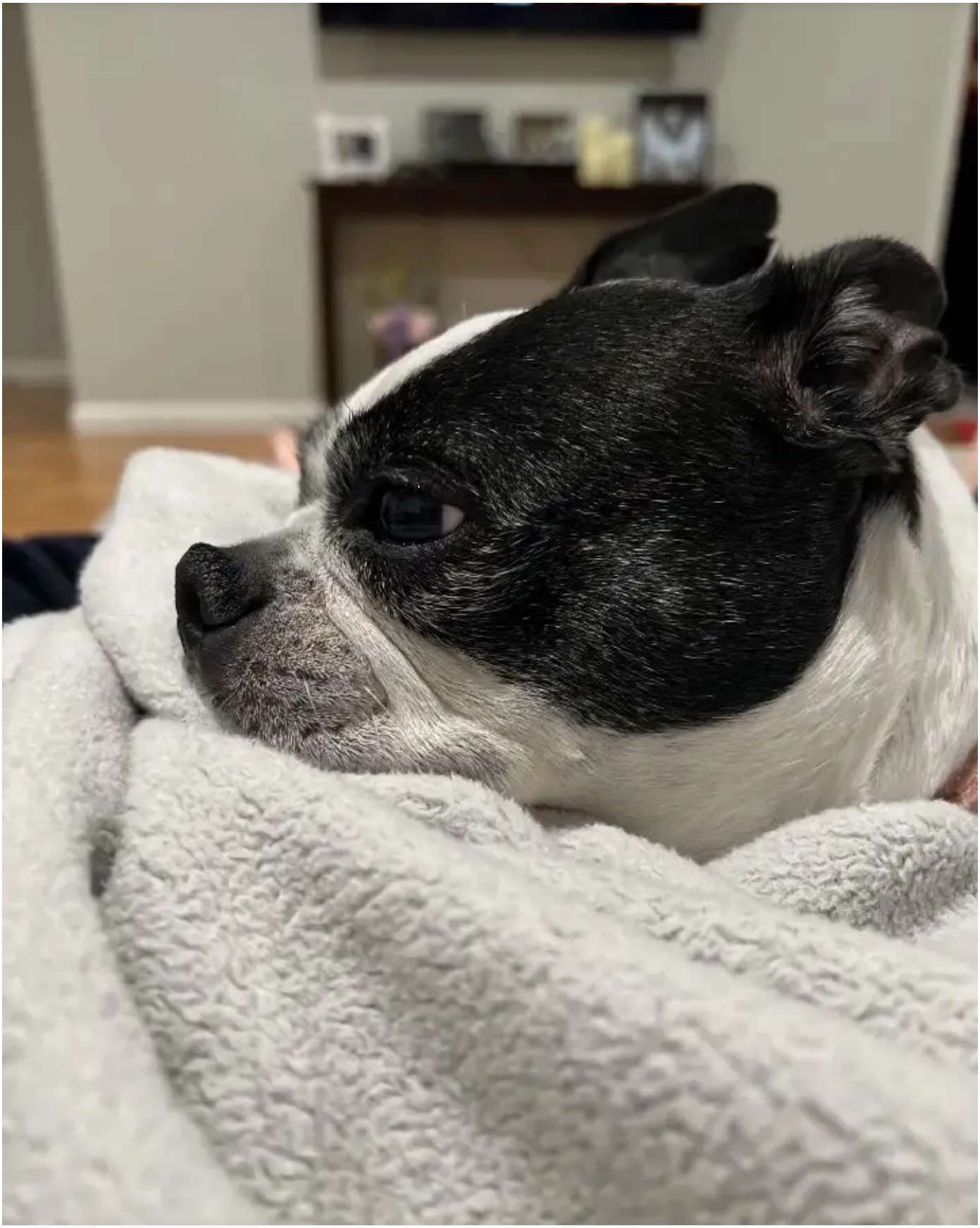
Leveraging a JSON file in Rust is an achievable task thanks to diverse crates, making our life easier.

Writing a JSON file is a straightforward task, but the complexity emerges when we parse and utilize some information within an existing JSON file.

We should always aim to parse a JSON file statically, assuming we thoroughly understand the various items present in the file and the type associated with each element.

This article mentions that JSON files are usually a good starting point to mock any data storage needs. Rust doe have some pretty cool crates that will help you interface with some databases such as Redis (crate redis), PostgresSQL (crate postgres), etc.

Like always, clap, subscribe, comments are welcomed!!

JSON official pages: here.

A basic introduction to JSON: here.

Rust's Structs: here.

Rust's Traits: here.