**Using Serde in Rust**

[[](https://medium.com/@joshmo_dev?source=post_page-----dc620ce4beff--------------------------------)](https://medium.com/@joshmo_dev?source=post_page-----dc620ce4beff--------------------------------)

[Joshua Mo](https://medium.com/@joshmo_dev?source=post_page-----dc620ce4beff--------------------------------)

·

Follow

7 min read

·

Jan 24

24



Thumbnail for article

In this article we’ll be talking about Serde, how you can use it in your Rust application as well as some more advanced tips and tricks.

**What is serde?**

The serde Rust crate is used to efficiently serialize and deserialize data in many formats. It does this by providing two traits you can use, aptly named Deserialize and Serialize. Being one of the most well-known crates in the ecosystem, it currently supports (de)serialization to over 20 types.

To get started, you’ll want to install the crate into your Rust application:

cargo add serde

**Using serde**

**Deserializing and Serializing data**

The simple way to serialize and deserialize data is by adding the serde derive feature. This adds a macro that you can use to implement Deserialize and Serialize automatically - you can do this with the --features flag (-F for short):

cargo add serde -F derive

Then we can add a macro to any struct or enum that we want to implement Deserialize or Serialize for:

use serde::{Deserialize, Serialize};  
  
#[derive(Deserialize, Serialize)]  
struct MyStruct {  
 message: String,  
 // ... the rest of your fields  
}

This allows us to use any crate with serde support to convert between said formats. As an example, let’s use serde-json to convert to and from JSON format:

use serde\_json::json;  
use serde::{Deserialize, Serialize};  
  
#[derive(Deserialize, Serialize)]  
struct MyStruct {  
 message: String,  
}  
  
fn to\_and\_from\_json() {  
 let json = json!({"message": "Hello world!"});  
 let my\_struct: MyStruct = serde\_json::from\_str(&json).unwrap();  
 assert\_eq!(my\_struct, MyStruct { message: "Hello world!".to\_string());  
 assert!(serde\_json::to\_string(my\_struct).is\_ok());  
}

If you’re interested in using serde-json for your Rust application, we have an article talking about JSON parsing libraries which you can check out [here.](https://www.shuttle.rs/blog/2024/01/18/parsing-json-rust)

We can also deserialize and serialize to/from many sources including from a file stream I/O, a JSON byte array and more!

**Implementing Deserialize and Serialize manually**

In order to better understand how serde works under the hood, we can also implement Deserialize and Serialize manually. This is quite complicated, but for now we will stick with a simple implementation. Here is a simple implementation for serializing an i32 primitive type:

use serde::{Serializer, Serialize};  
  
impl Serialize for i32 {  
 fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>  
 where  
 S: Serializer,  
 {  
 serializer.serialize\_i32(\*self)  
 }  
}

To be able to convert the type, serde internally requires us to use a type that implements Serializer. To implement Serialize for a type that isn’t directly a primitive, we can extend this by serializing into a primitive, then converting into whatever type we want from the primitive. If we want custom serialization for structs, we can also do the same with use of the SerializeStruct trait:

use serde::ser::{Serialize, Serializer, SerializeStruct};  
  
struct Color {  
 r: u8,  
 g: u8,  
 b: u8,  
}  
  
impl Serialize for Color {  
 fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>  
 where  
 S: Serializer,  
 {  
 // 3 is the number of fields in the struct.  
 let mut state = serializer.serialize\_struct("Color", 3)?;  
 state.serialize\_field("r", &self.r)?;  
 state.serialize\_field("g", &self.g)?;  
 state.serialize\_field("b", &self.b)?;  
 state.end()  
 }  
}

Note that to serialize a field, the field type also needs to implement Serialize. If you have a custom type that doesn't implement Serialize, you will either need to implement Serialize or use a Serialize derive macro (if the struct/enum type holds types that all implement Serialize).

The Deserialize trait is a little bit different and is a fair bit more complicated to implement. To be able to deserialize to a type, the type itself needs to implement Sized which means that there are a number of types which can’t use this trait (for example &str) because they are unsized types. To deserialize a type, you also need to use a type that implements the Visitor trait.

The Visitor trait uses the Visitor design pattern in Rust. This means that it encapsulates an algorithm that operates over a collection of same-sized objects. It allows you to write multiple different algorithms for operating over the data, without needing to change any original functionality. You can find out more about this [here.](https://rust-unofficial.github.io/patterns/patterns/behavioural/visitor.html)

Below is an example for a MessageVisitor type that attempts to deserialize multiple types to String:

use std::fmt;  
  
use serde::de::{self, Visitor};  
  
struct MessageVisitor;  
  
impl<'de> Visitor<'de> for MessageVisitor {  
 type Value = String;  
  
 fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {  
 formatter.write\_str("A message that can either be deserialized from an i32 or String")  
 }  
 fn visit\_string<E>(self, value: String) -> Result<Self::Value, E>  
 where  
 E: de::Error,  
 {  
 Ok(value)  
 }  
 fn visit\_str<E>(self, value: &str) -> Result<Self::Value, E>  
 where  
 E: de::Error,  
 {  
 Ok(value.to\_owned())  
 }  
 fn visit\_i32<E>(self, value: i32) -> Result<Self::Value, E>  
 where  
 E: de::Error,  
 {  
 Ok(value.to\_string())  
 }  
}

As you can see, the implementation is quite large! However, it also allows us to make the implementation much simpler. By implementing the Visitor trait, we can pass the type that implements it to our Deserialize method and then deserialize JSON into our struct:

use serde::{Deserialize, Deserializer};  
  
impl<'de> Deserialize<'de> for MyStruct {  
 fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>  
 where  
 D: Deserializer<'de>,  
 {  
 // note: don't use unwrap in production!  
 let message = deserializer.deserialize\_string(MessageVisitor).unwrap();  
 Ok(Self { message })  
 }  
}

There is also documentation on deserializing structs which you can find [here.](https://serde.rs/deserialize-struct.html) However, generally speaking it is recommended that you use the derive feature macros as the manual implementation (as seen on the page itself) is quite large. The implementation involves mostly using a visitor that can visit a map or sequence then iterate through the elements to deserialize it.

**Using serde attributes**

When it comes to serde, the crate also has a number of useful attribute macros that we can use on our types to allow things like field renaming when deserializing a field or serializing to a struct. One of the best examples of this would be when you’re interacting with an API written in a language that may have a key that is a reserved keyword in Rust. You can add a #[serde(rename)] attribute macro like so:

use serde::{Deserialize, Serialize};  
  
#[derive(Deserialize, Serialize)]  
pub struct MyStruct {  
 #[serde(rename = "type")]  
 kind: String  
}

This allows you to get around the issue!

You can also rename all of your fields to another casing by using the rename\_all attribute:

use serde::{Deserialize, Serialize};  
  
#[derive(Deserialize, Serialize)]  
#[serde(rename\_all = "camelCase")]  
pub struct MyStruct {  
 my\_message: String  
}

Now when you serialize this struct, my\_message should automatically turn into myMessage! Perfect for working with APIs written in other languages or with different conventions.

If you’d prefer to not wrap fields in Option, you can also implement default values by using #[serde(default)]. This simply allows fields to be filled in with default values instead of automatically erroring out. You can also use #[serde(default = "path")] to be able to point to functions for providing the automatic default. For example, this struct and function:

use serde::{Deserialize, Serialize};  
  
#[derive(Deserialize, Serialize)]  
pub struct MyStruct {  
 #[serde(path = "my\_function")]  
 my\_message: String,  
}

fn my\_function() -> String {  
 "Hello world!".to\_string()  
}

serde also offers other useful attributes, like being able to deny unknown fields using #[serde(deny\_unknown\_fields)] on top of the struct. This allows you to make sure that the struct is exactly as-is when serializing and deserializing.

**Deserializing and Serializing enums**

Let’s examine this enum type:

use serde::{Deserialize, Serialize};  
  
#[derive(Deserialize, Serialize)]  
enum MyEnum {  
 Data { id: String, data: Value },  
 SomeOtherData { id: i32, name: String }  
}

Note that when converting to and from this enum, it can take two options:

* A String field named id and a JSON value with the key data (this can be a map, a value or anything that the Json value can hold)
* An i32 field named id and a String field named name

You can then match the enum variant for further processing.

When the first enum variant is written in JSON, you can see that it should correspond with this:

{  
 "Data": {  
 "id": "your\_id\_here",  
 "data": { .. }  
 }  
}

This type of data is “externally tagged” — meaning the data is characterized by the identifier being on the outside of the JSON object. We can add inline tagging so that the identifier is on the inside of the crate — let’s have a look at what this would look like:

use serde::{Deserialize, Serialize};  
  
#[derive(Deserialize, Serialize)]  
#[serde(tag = "type")]  
enum MyEnum {  
 Data { id: String, data: Value },  
 SomeOtherData { id: i32, name: String }  
}

Now the JSON representation looks like this:

{  
 "type": "Data",  
 "id": "your\_id\_here",  
 "data": { .. }  
}

Interested in reading more? The serde documentation has a page on tagging which you can find [here.](https://serde.rs/enum-representations.html)

**Crates that work well with Serde**

**serde\_with**

serde\_with is a crate that provides custom de/serialization helpers to use with serde's with annotation. Normally, you can define a module for the (de)serializer to use that follows a custom module for custom (de)serialization:

#[derive(Deserialize, Serialize)]  
pub struct MyStruct {  
 #[serde(with = "my\_module")]  
 my\_message: String  
}

When using serde\_with, it works by replacing the with annotation with a new one called serde\_as. With this new attribute macro you can do quite a few things:

* De/serializing a type using Display and FromStr traits.
* Support arrays larger than 32 elements.
* Skip serializing empty Option types.
* Deserialize a comma separated list into a Vec<String>.

To use serde\_with, you need to add it to your Cargo.toml either manually or by using the following command:

cargo add serde\_with

Then you need to add serde\_as to the type you want to use it for, like so:

use serde\_with::{serde\_as, DisplayFromStr};  
  
#[serde\_as]  
#[derive(Deserialize, Serialize)]  
struct MyStruct {  
 // Serialize with Display, deserialize with FromStr  
 #[serde\_as(as = "DisplayFromStr")]  
 my\_number: u8,  
}

This struct lets you convert to/from a string but have the type itself in your Rust struct be u8! Pretty useful, right?

This crate also comes with a [guide](https://docs.rs/serde_with/3.5.0/serde_with/guide/index.html) that you can use to fully capitalise on serde\_with. Overall, a strong companion crate for serde.

**serde\_bytes**

serde\_bytes is a crate that allows for optimised handling of &[u8] and Vec<u8> types - while serde is capable of dealing these types by itself, some formats can be de/serialized more efficiently. It’s quite simple to use - you just add it to your Cargo.toml and then add it via the #[serde(with = "serde\_bytes")] annotation like so:

use serde::{Deserialize, Serialize};  
  
#[derive(Deserialize, Serialize)]  
struct MyStruct {  
 #[serde(with = "serde\_bytes")]  
 byte\_buf: Vec<u8>,  
}

Overall, an easy to use and simple crate that improves performance without much knowledge required.

**Finishing up**

I hope you enjoyed reading about Serde! It’s a pretty powerful Rust crate and forms the backbone of most Rust applications.

Read more:

* Read more about how you can get started with the Rocket web framework [here.](https://www.shuttle.rs/blog/2023/12/13/using-rocket-rust)
* Read more about using JSON parsers [here.](https://www.shuttle.rs/blog/2024/01/18/parsing-json-rust)

[Rust](https://medium.com/tag/rust?source=post_page-----dc620ce4beff---------------rust-----------------)

[Rustlang](https://medium.com/tag/rustlang?source=post_page-----dc620ce4beff---------------rustlang-----------------)

[Gui̇de](https://medium.com/tag/gui%CC%87de?source=post_page-----dc620ce4beff---------------gui%CC%87de-----------------)

[Tutorial](https://medium.com/tag/tutorial?source=post_page-----dc620ce4beff---------------tutorial-----------------)