Serde Driven Reflection

EuroRust 2025

Ohad Ravid, SWE at Wiz

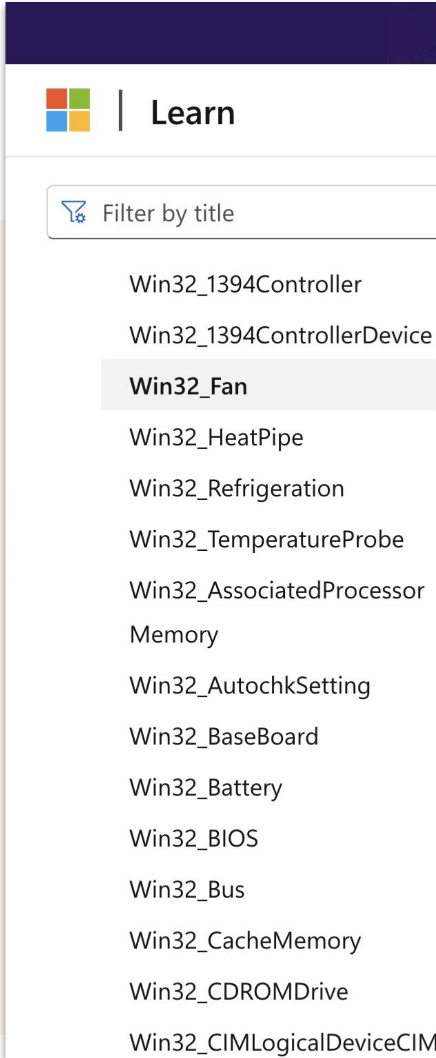
(We do cloud security,
mostly in Go 🙄 but also in Rust 🥳🦀)

Part 0 -
Python, Windows, SQL

An Unexpected Motivation

WMI, a la Python

```python
import wmi # aka the Windows Management Infrastructure.


con = wmi.WMI()


for fan in con.Win32_Fan():
    if fan.ActiveCooling:
        print(f"Fan `{fan.Name}` is \
                running at {fan.Speed}")
```



Learn

Filter by title

Win32_1394Controller

Win32_1394ControllerDevice

Win32_Fan

Win32_HeatPipe

Win32_Refrigeration

Win32_TemperatureProbe

Win32_AssociatedProcessor
Memory

Win32_AutochkSetting

Win32_BaseBoard

Win32_Battery

Win32_BIOS

Win32_Bus

Win32_CacheMemory

Win32_CDROMDrive

Win32_CIMLogicalDeviceCIM

WMI, a la Python

```python
class WMI:
    def __getattr__(self, name: str):
        ...
con = wmi.WMI()


# con.Win32_Fan() == con.__getattr__("Win32_Fan")()
#                 == con.query("SELECT * FROM Win32_Fan")
for fan in con.Win32_Fan():
    if fan.ActiveCooling: # == fan.__getattr__("ActiveCooling") == ..
        print(f"Fan `{fan.Name}` is \
                running at {fan.Speed}")
```

WMI, a la (Raw) Python

```python
class Object:
  def __getattr__(self, name: str):
    return self.get(name)

  def get(self, name: str):
    prop = self.get_raw(name)
    # .. PyCom_PyObjectFromVariant:
    if prop.type == CIMTYPE.BOOL:
      return bool(prop.value)
    elif prop.type == CIMTYPE.UI1:
      return int(prop.value)
    elif prop.type == CIMTYPE.UI2:
      ...
```

```python
class WMI:
  def __getattr__(self, name: str):
    return lambda: \
      self.query(f"SELECT * FROM {name}")

  def query(self, query: str) -> List[Object]:
    # .. PyIDispatch::Invoke(“ExecQuery”, ..)
    ...
```

WMI, a la Rust

yes this is a lot of code,
and we're just warming up!

Brace yourselves!

```rust
// In mod `raw_api`
pub enum Value {
  Bool(bool),
  I1(i8),
  // ..
  UI8(u64),
  String(String),
}
```

```rust
// `raw_api`, cont.
pub struct Object { .. }

impl Object {
  fn get(&self, name: &str) -> Value { .. }
}

fn query(query: &str) -> Vec<Object> { .. }
```

```rust
let object = raw_api::query("SELECT * FROM Win32_Fan")[0];
assert_eq!(object.get("DesiredSpeed"), Value::UI8(100u64));
```

WMI, a la Rust

yes this is a lot of code,
and we're just warming up!

Brace yourselves!

```
// In mod `raw_api`
pub enum Value {
  Bool(bool),
  I1(i8),
  // ..
  UI8(u64),
  String(String),
}
```

```
// `raw_api`, cont.
pub struct Object { .. }

impl Object {
  fn get(&self, name: &str) -> Value { .. }
}

fn query(query: &str) -> Vec<Object> { .. }
```

```
let object = raw_api::query("SELECT * FROM Win32_Fan")[0];
assert_eq!(object.get("DesiredSpeed"), Value::UI8(100u64));
```

WMI, a la Rust

yes this is a lot of code,
and we're just warming up!

Brace yourselves!

```
// In mod `raw_api`
pub enum Value {
  Bool(bool),
  I1(i8),
  // ..
  UI8(u64),
  String(String),
}
```

```
// `raw_api`, cont.
pub struct Object { .. }

impl Object {
  fn get(&self, name: &str) -> Value { .. }
}


fn query(query: &str) -> Vec<Object> { .. }
```

```
let object = raw_api::query("SELECT * FROM Win32_Fan")[0];
assert_eq!(object.get("DesiredSpeed"), Value::UI8(100u64));
```

WMI, a la Rust

yes this is a lot of code,
and we're just warming up!

Brace yourselves!

```rust
// In mod `raw_api`
pub enum Value {
  Bool(bool),
  I1(i8),
  // ..
  UI8(u64),
  String(String),
}
```

```rust
// `raw_api`, cont.
pub struct Object { .. }


impl Object {
  fn get(&self, name: &str) -> Value { .. }
}


fn query(query: &str) -> Vec<Object> { .. }
```

```rust
let object = raw_api::query("SELECT * FROM Win32_Fan")[0];
assert_eq!(object.get("DesiredSpeed"), Value::UI8(100u64));
```

It's RAW

The problem: this a terrible API *for the user:*

```
let res = raw_api::query("SELECT * FROM Win32_Fan");
for obj in res {
    if obj.get("ActiveCooling") == Value::Bool(true) {
        if let Value::String(name) = obj.get("Name") {
            if let Value::UI8(speed) = obj.get("DesriedSpeed") {
                println!("Fan `{name}` is running at {speed}");
            }
        }
    }
}
```

```
struct WMIMapAccess<'a, S, I>
where
    S: AsRef<str>,
    I: Iterator<Item = S>,
{
    fields: Peekable<I>,
    de: &'a Deserializer,
}

impl<'de, 'a, S, I> MapAccess
where
    S: AsRef<str>,
    I: Iterator<Item = S>,
{
    type Error = WMIError;

    // ..
```

Part 1 - What is Reflection, anyway?

Ideally, what we want is something like this:

```rust
// 1. The user defines a custom struct for the type of objects to query.
struct Fan { name: String, active_cooling: bool, desired_speed: u64 }

// 2. `query` should return instances of `Fan`, executing "SELECT * FROM Win32_Fan".
let res: Vec<Fan> = api::query();

// 3. Profit.
for fan in res {
    if fan.active_cooling {
        println!("Fan `{}` is running at {}", fan.name, fan.desired_speed);
    }
}
```

Part 1 - What is Reflection, anyway?

Ideally, what we want is something like this:

```
// 1. The us          stom struct for the type of objects to query.
struct Fan                    g: bool, desired_speed: u64 }

// 2. `query` sho                                          FROM Win32_Fan".
let res: Vec<Fan> = api::query();

// 3. Profit.
for fan in res {
    if fan.active_cooling {
        println!("Fan `{}` is running at {}", fan.name, fan.desired_speed);
    }
}
```

```
fields: HashMap<&'static str, ???> = reflect::<Fan>();
fan = Fan { ??? };
```

Just One More Generic, Bro, Trust Me

Usually, if we want to return user-defined types, we need a **generic-return-type**:

```
fn query<T>() -> Vec<T> where T: ??? { ??? }
```

with `T` implementing a trait for:

1. Getting the name of `T`.

2. Constructing a `T` from `Object`.

But, imagine something like `T: From<Object>`.
It forces the user to implement the same code from before, but in a trait.

Like the poster child for Generic Return Types, `Iterator::collect(..)`:

```
fn collect<B>(self) -> B
where
    B: FromIterator<Self::Item>
{ .. }
```

Just One More... proc-macro?

Serde to the Rescue

We *can* create user-defined structs from “dynamic” values, with Serde’s **`derive`**:

```rust
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[serde(rename_all = "PascalCase")]
pub struct Fan {
    name: String,
    active_cooling: bool,
    desired_speed: u64,
}
```

```rust
let fan: Fan = serde_json::from_str(r#"{
    "Name": "CPU1",
    "ActiveCooling": true,
    "DesiredSpeed": 1100,
}"#)?;

println!(
    "Fan `{}` is running at {} RPM",
    fan.name, fan.desired_speed
);
```

So.. how hard can it be to hitch a ride on Serde?

Part 2 - Dipping Our Toes

Our goal is to make our `query()` capable of returning (almost any) `Deserialize`-able type `T`:

```
fn query<T: Deserialize>() -> Vec<T> { todo!() }
```

It should:

1. Infer the needed SQL using `T`'s name, and then
2. Convert the returned *`raw_api`*`::Objects` into `T`s.

To do this, we need to understand how the `Deserialize` trait works under the hood, and how we can use it for "reflection".

Dipping Our Toes

```
fn query<T: Deserialize>()
  -> Vec<T> { .. }
```

It Takes Two to Tango

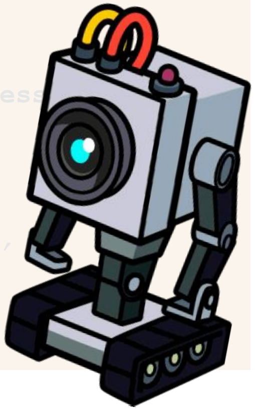
```
let fan: Fan = serde_json::from_str(r#"{ "Name": "CPU1", "Active.."#);



impl Deserialize for Fan                            impl Deserializer for serde_json::Deserializer
┌ fn deserialize(deser)                             │
│   let visitor = FanVisitor {}                     │
│   deser.deserialize_struct(.., visitor) ─calls──▶ ├ fn deserialize_struct(.., visitor)
│                                                   │   let map = serde_json::de::MapAccess::new(..)
│   impl Visitor for FanVisitor                     │
│   ┌ fn visit_map(map) ◀────────calls─────────────────  return visitor.visit_map(map)
│   │   loop {                                         impl MapAccess for serde_json::de::MapAccess
│   │     key = map.next_key() ──────calls─────────▶ ┌ fn next_key()   // { ..,▼"Name": ..
│   │     /* when key is "Name" */                   │
│   │     name: String = map.next_value() ───calls──▶ └ fn next_value() // {          ..:▼"CPU1", ..
│   │   }
◀───────  return Fan { name, ... }
```

It Takes Two to Tango

```
let fan = Fan::deserialize(serde_json::Deserializer::from_str(r#"{ "Name": "CPU1", "Active.."#));

// Generated by `#[derive(Deserialize)]`:           // Provided by `serde_json`:
impl Deserialize for Fan                            impl Deserializer for serde_json::Deserializer
┌ fn deserialize(deser)                             │
│   let visitor = FanVisitor {}                     │
│   deser.deserialize_struct(.., visitor) ─calls──▶ ├ fn deserialize_struct(.., visitor)
│                                                   │   let map = serde_json::de::MapAccess::new(..)
│   impl Visitor for FanVisitor                     │
│   ┌ fn visit_map(map) ◀────────calls──────────────── return visitor.visit_map(map)
│   │   loop {                                          impl MapAccess for serde_json::de::MapAccess
│   │     key = map.next_key() ──────calls───────────▶ ┌ fn next_key()   // { ..,▼"Name": ..
│   │     /* when key is "Name" */                      │
│   │     name: String = map.next_value() ──calls────▶ └ fn next_value() // {          ..:▼"CPU1", ..
│   │   }
◀────── return Fan { name, ... }
```

It Takes Two to Tango

```
let fan = Fan::deserialize(serde_json::Deserializer::from_str(r#"{ "Name": "CPU1", "Active.."#));


// Generated by `#[derive(Deserialize)]`:           // Provided by `serde_json`:
impl Deserialize for Fan                            impl Deserializer for serde_json::Deserializer
┌ fn deserialize(deser)                             │
│   let visitor = FanVisitor {}                     │
│   deser.deserialize_struct(.., visitor) ─calls──▶ ├ fn deserialize_struct(.., visitor)
│                                                   │   let map = serde_json::de::MapAccess::new(..)
│   impl Visitor for FanVisitor                     │
│   ┌ fn visit_map(map) ◀────────calls─────────────────── return visitor.visit_map(map)
│   │   loop {                                          impl MapAccess for serde_json::de::MapAccess
│   │     key = map.next_key() ──────calls────────────▶ ┌ fn next_key()   // { ..,▼"Name": ..
│   │     /* when key is "Name" */                      │
│   │     name: String = map.next_value() ──calls─────▶ └ fn next_value() // {          ..:▼"CPU1",
│   │   }
◀──────  return Fan { name, ... }
```



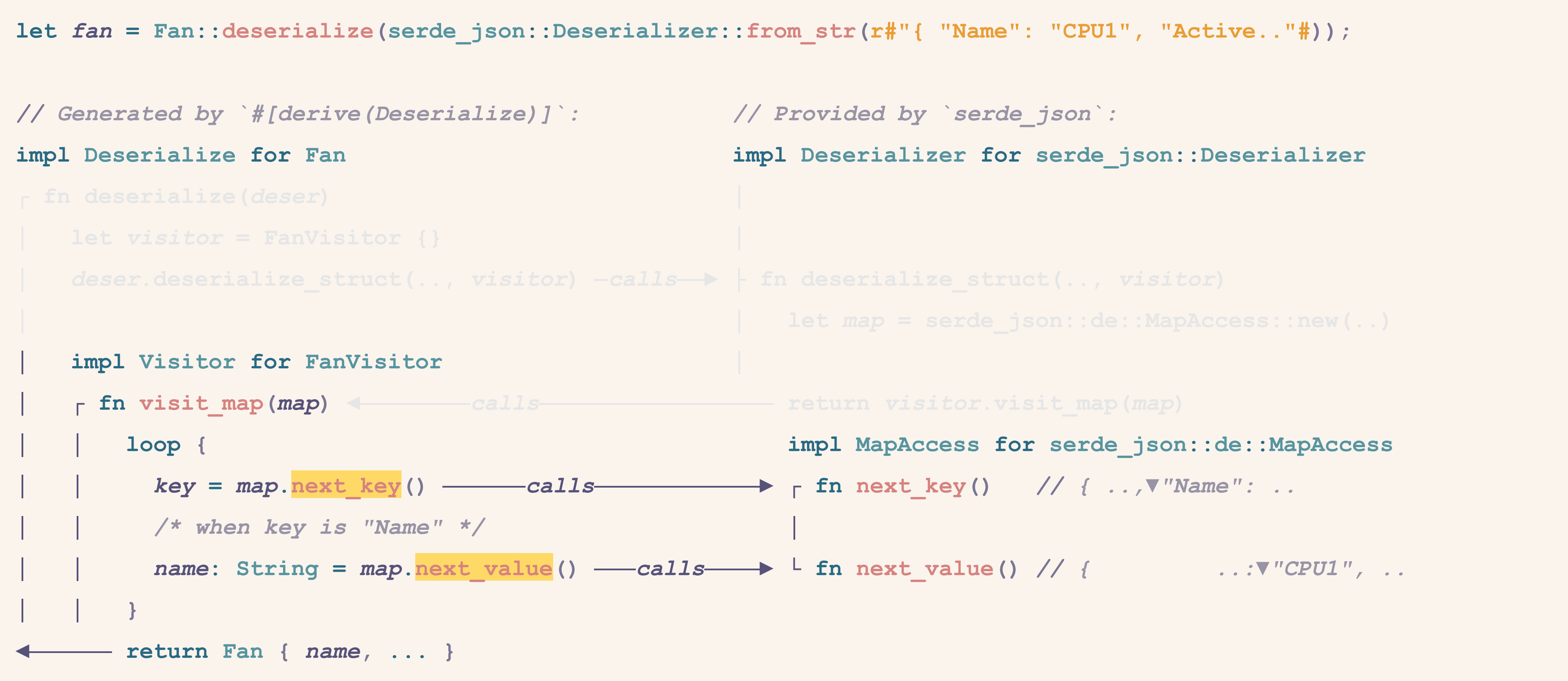
It Takes Two to Tango

```
let fan = Fan::deserialize(serde_json::Deserializer::from_str(r#"{ "Name": "CPU1", "Active.."#));


// Generated by `#[derive(Deserialize)]`:          // Provided by `serde_json`:
impl Deserialize for Fan                            impl Deserializer for serde_json::Deserializer
┌ fn deserialize(deser)                             │
│   let visitor = FanVisitor {}                     │
│   deser.deserialize_struct(.., visitor) —calls——→ ├ fn deserialize_struct(.., visitor)
│                                                   │   let map = serde_json::de::MapAccess::new(..)
│   impl Visitor for FanVisitor                     │
│   ┌ fn visit_map(map) ←————————calls————————————————  return visitor.visit_map(map)
│   │   loop {                                          impl MapAccess for serde_json::de::MapAccess
│   │     key = map.next_key() ——————calls——————————→ ┌ fn next_key()   // { ..,▼"Name": ..
│   │     /* when key is "Name" */                    │
│   │     name: String = map.next_value() ——calls———→ └ fn next_value() // {          ..:▼"CPU1", ..
│   │   }
←———————return Fan { name, ... }
```

It Takes Two to Tango

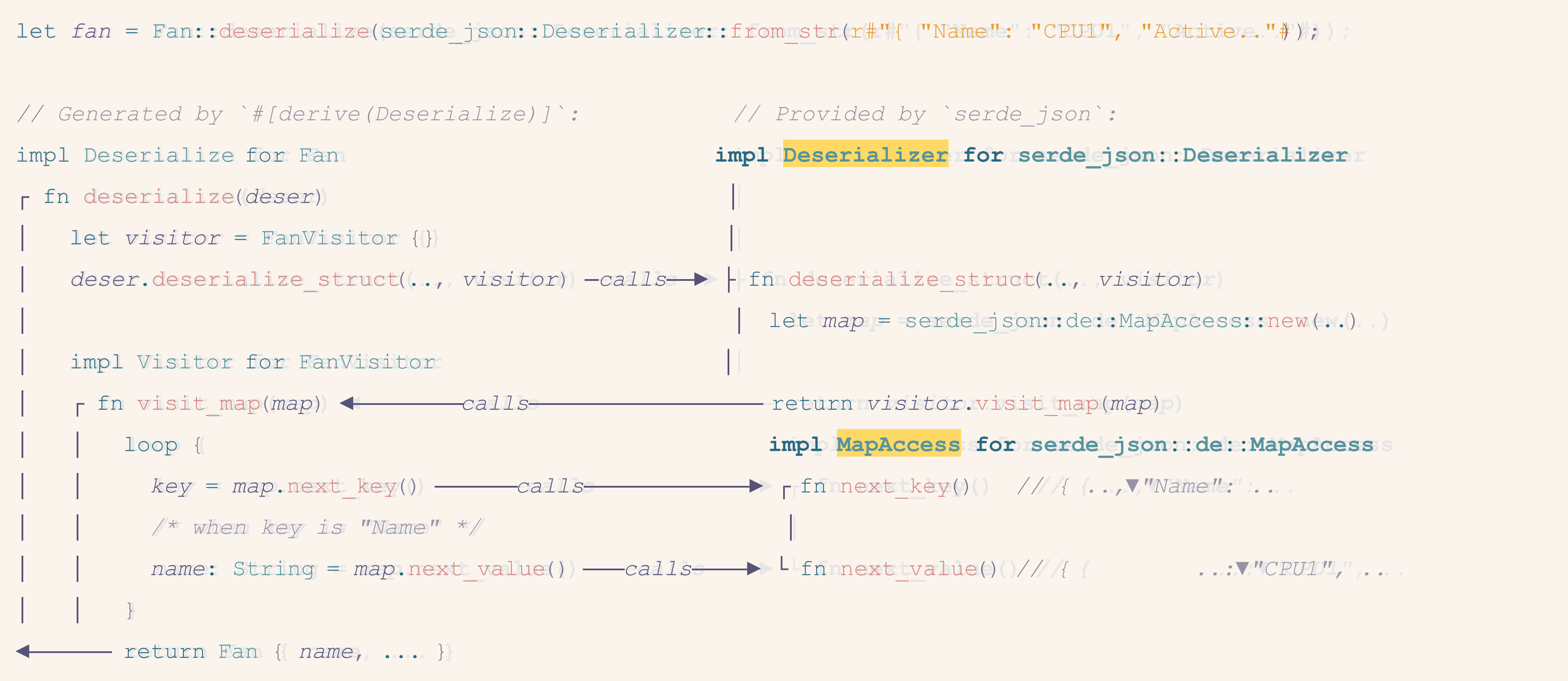
```
let fan = Fan::deserialize(serde_json::Deserializer::from_str(r#"{ "Name": "CPU1", "Active.."#));


// Generated by `#[derive(Deserialize)]`:           // Provided by `serde_json`:
impl Deserialize for Fan                            impl Deserializer for serde_json::Deserializer
┌ fn deserialize(deser)                             │
│   let visitor = FanVisitor {}                     │
│   deser.deserialize_struct(.., visitor) —calls──▶ ├ fn deserialize_struct(.., visitor)
│                                                   │   let map = serde_json::de::MapAccess::new(..)
│   impl Visitor for FanVisitor                     │
│   ┌ fn visit_map(map) ◀────────calls──────────────── return visitor.visit_map(map)
│   │   loop {                                          impl MapAccess for serde_json::de::MapAccess
│   │     key = map.next_key() ──────calls──────────▶ ┌ fn next_key()   // { ..,▼"Name": ..
│   │     /* when key is "Name" */                    │
│   │     name: String = map.next_value() ──calls───▶ └ fn next_value() // {          ..:▼"CPU1", ..
│   │   }
◀────── return Fan { name, ... }
```

It Takes Two to Tango



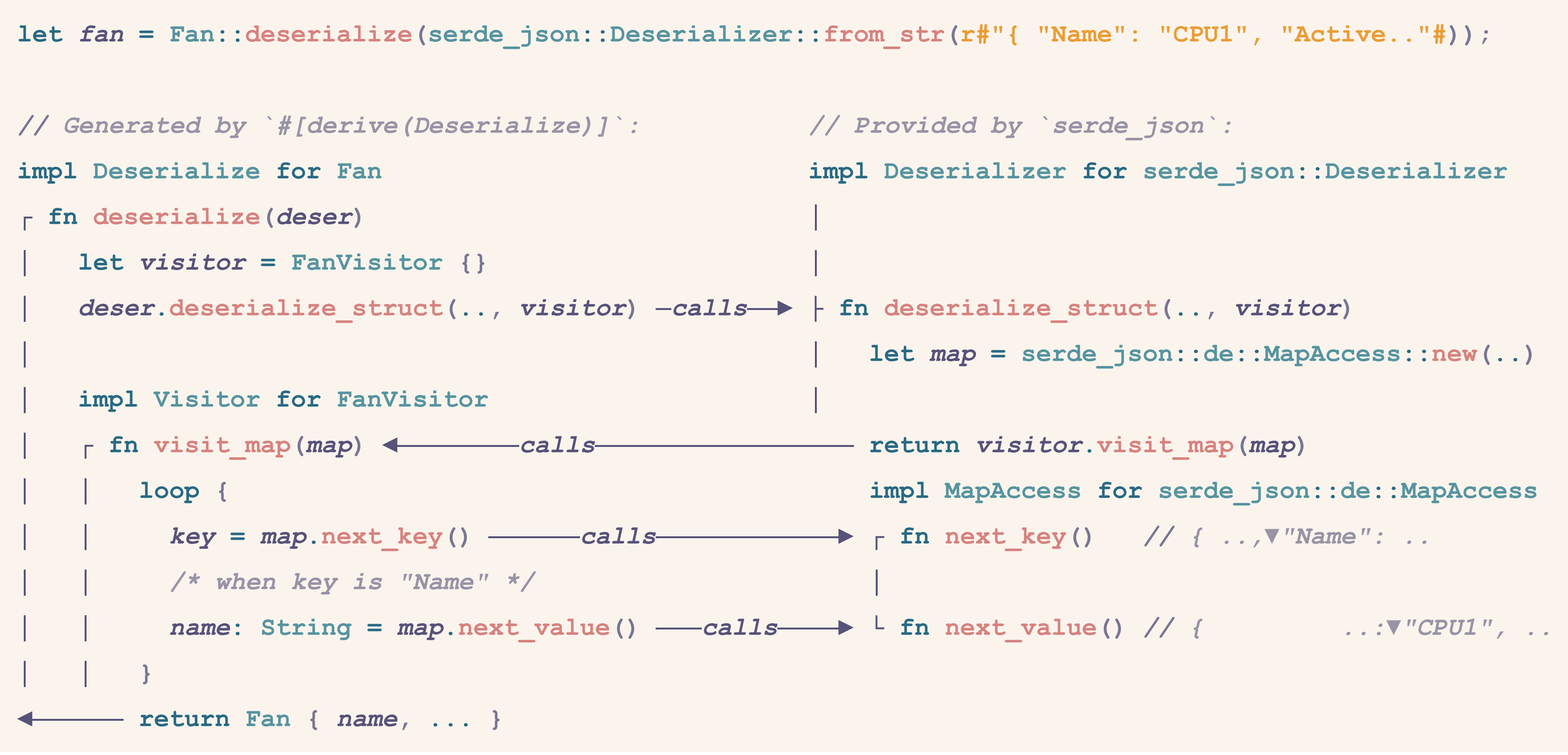
```
let fan = Fan::deserialize(serde_json::Deserializer::from_str(r#"{ "Name": "CPU1", "Active.."#));


// Generated by `#[derive(Deserialize)]`:           // Provided by `serde_json`:
impl Deserialize for Fan                            impl Deserializer for serde_json::Deserializer
┌ fn deserialize(deser)                             │
│   let visitor = FanVisitor {}                     │
│   deser.deserialize_struct(.., visitor) ─calls───▶ ├ fn deserialize_struct(.., visitor)
│                                                   │   let map = serde_json::de::MapAccess::new(..)
│   impl Visitor for FanVisitor                     │
│   ┌ fn visit_map(map)  ◀─────────calls───────────────── return visitor.visit_map(map)
│   │   loop {                                          impl MapAccess for serde_json::de::MapAccess
│   │     key = map.next_key() ──────calls──────────▶  ┌ fn next_key()   // { ..,▼"Name": ..
│   │     /* when key is "Name" */                      │
│   │     name: String = map.next_value() ───calls───▶  └ fn next_value() // {         ..:▼"CPU1", ..
│   │   }
◀────── return Fan { name, ... }
```

It Takes Two to Tango

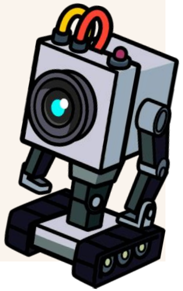


```
let fan = Fan::deserialize(serde_json::Deserializer::from_str(r#"{ "Name": "CPU1", "Active.."#));


// Generated by `#[derive(Deserialize)]`:              // Provided by `serde_json`:
impl Deserialize for Fan                              impl Deserializer for serde_json::Deserializer
┌ fn deserialize(deser)                                │
│   let visitor = FanVisitor {}                        │
│   deser.deserialize_struct(.., visitor) —calls——▶ ├ fn deserialize_struct(.., visitor)
│                                                      │  let map = serde_json::de::MapAccess::new(..)
│   impl Visitor for FanVisitor                        │
│   ┌ fn visit_map(map) ◀——————calls——————————————————   return visitor.visit_map(map)
│   │   loop {                                            impl MapAccess for serde_json::de::MapAccess
│   │     key = map.next_key() ——————calls——————————▶ ┌ fn next_key()   // { ..,▼"Name": ..
│   │     /* when key is "Name" */                     │
│   │     name: String = map.next_value() ——calls——▶ └ fn next_value() // {        ..:▼"CPU1", ..
│   │   }
◀———— return Fan { name, ... }
```

The Deserialize Trait, Play by Play



```
let fan = Fan::deserialize(serde_json::Deserializer::from_str(r#"{ "Name": "CPU1", "Active.."#));


// Generated by `#[derive(Deserialize)]`:          // Provided by `serde_json`:
impl Deserialize for Fan                           impl Deserializer for serde_json::Deserializer
┌ fn deserialize(deser)                             |
|   let visitor = FanVisitor {}                     |
|   deser.deserialize_struct(.., visitor) ─calls──→ ├ fn deserialize_struct(.., visitor)
|                                                   |   let map = serde_json::de::MapAccess::new(..)
|   impl Visitor for FanVisitor                     |
|   ┌ fn visit_map(map) ←────────calls─────────────────  return visitor.visit_map(map)
|   |   loop {                                          impl MapAccess for serde_json::de::MapAccess
|   |     key = map.next_key() ──────calls──────────→   ┌ fn next_key()   // { ..,▼"Name": ..
|   |     /* when key is "Name" */                      |
|   |     name: String = map.next_value() ──calls───→   └ fn next_value() // {         ..:▼"CPU1", ..
|   |   }
←────── return Fan { name, ... }
```



Current State of Affairs

```rust
fn query<T: Deserialize>()
  -> Vec<T> { .. }
```

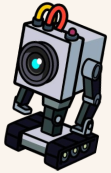
```rust
impl<'de> MapAccess<'de> for
ObjectMapAccess { .. }
```

```rust
impl<'de> Deserializer<'de>
for ObjectDeserializer { .. }
```

Part 3 - Jump In at the Deep End

Zooming in, the `FanVisitor` expects *our* `map` to work like this:

```
key: &str = map.next_key()?;
if key == "Name" {
  // 1. Should return `"CPU1"`.
  name: String = map.next_value()?;
}

key: &str = map.next_key()?;
if key == "ActiveCooling" {
  // 2. But now, it should return `true`.
  active_cooling: bool = map.next_value()?;
}
return Fan { name, active_cooling, .. };
```



```
// We need to provide a `map` ~ {
//   "Name": "CPU1",
//   "ActiveCooling": true,
//   .. }, and that implements:
trait MapAccess<'de> {
  fn next_key(&mut self) -> ..
  fn next_value(&mut self) -> ..
}
```

A Map to Nowhere

Assuming we've solved all the other problems,
our `MapAccess`-able `struct` will look like this:

```
impl<'de> MapAccess<'de> for ObjectMapAccess {
    // ..
    fn next_value<V>(&mut self) -> Result<V>
    where V: Deserialize<'de>, // [1]
    {
        let current_value: Value = /* .. */;
        // Hmm.
    }
}
```

[1]: Actually, we need to implement `next_value_seed`, which uses `DeserializeSeed`, but the idea is the same.

Our First Deserializer

Simplified, we need to be able to do:

```
let name_value = obj.get("Name"); // == Value::String("CPU1".to_string())

struct ValueDeserializer { value: Value }

let name: String = Deserialize::deserialize(ValueDeserializer { value: name_value })?;
```

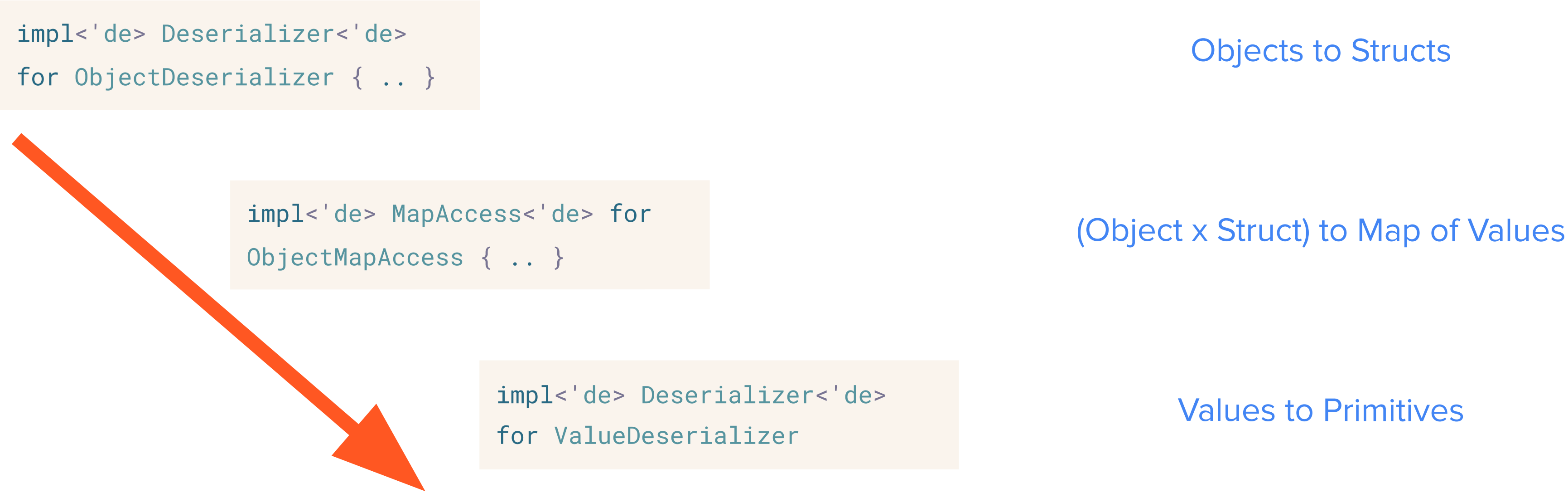
We *also* need to implement `Deserializer` for `ValueDeserializer`.

A reminder:

```
pub enum Value {
  Bool(bool),
  I1(i8),
  // ..
  UI8(u64),
  String(String),
}
```

Values, Maps, Structs

`ValueDeserializer` will create primitives, like `bool`, `u64`, and `String`, While `ObjectDeserializer` and `ObjectMapAccess` will handle structs.



```
impl<'de> Deserializer<'de>
for ObjectDeserializer { .. }
```

Objects to Structs

```
impl<'de> MapAccess<'de> for
ObjectMapAccess { .. }
```

(Object x Struct) to Map of Values

```
impl<'de> Deserializer<'de>
for ValueDeserializer
```

Values to Primitives

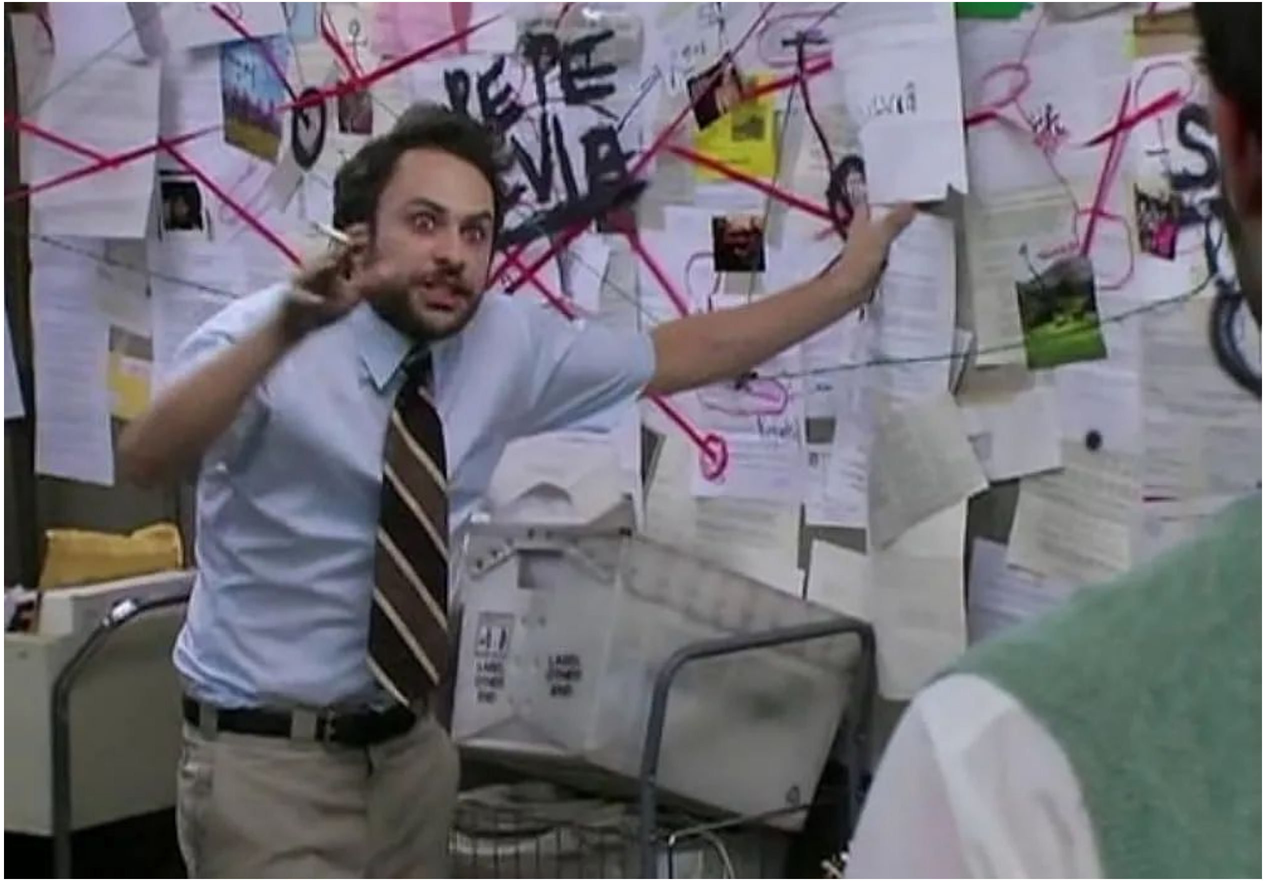
Current State of Affairs

```
fn query<T: Deserialize>()
  -> Vec<T> { .. }
```

```
impl<'de> Deserializer<'de>
for ValueDeserializer
```

```
impl<'de> MapAccess<'de> for
ObjectMapAccess { .. }
```

```
impl<'de> Deserializer<'de>
for ObjectDeserializer { .. }
```


PEPE

Something of Value

```
struct ValueDeserializer { value: Value }
impl<'de> Deserializer<'de> for ValueDeserializer {
    fn deserialize_any<V>(self, visitor: V) -> Result<V::Value>
        where V: Visitor<'de> {
        match self.value {
            Value::Bool(b) => visitor.visit_bool(b),
            // ..
            Value::UI8(v) => visitor.visit_u64(v),
            Value::String(s) => visitor.visit_str(&s), // [2]
        }
    }
    forward_to_deserialize_any! { /* .. */ }
}
```

```
pub enum Value {
  Bool(bool),
  I1(i8),
  // ..
  UI8(u64),
  String(String),
}
```

A self-describing data format

[2]: Actually, `visit_string` is more appropriate here, but this will be slightly more interesting.

A String for a String

Which works like this:

```
let name_value = Value::String("CPU1".to_string());
let name: String = Deserialize::deserialize(ValueDeserializer { value: name_value })?;

impl Deserialize for String                        impl Deserializer for ValueDeserializer
┌ fn deserialize(deser)                            |
|   let visitor = StringVisitor {}                 |
|   deser.deserialize_string(.., visitor) —calls→ | fn deserialize_any(.., visitor)
|                                                      match self.value {
|   impl Visitor for StringVisitor                       ..
|   ┌ fn visit_str(value: &str) ←────────calls──────────  Value::String(s) => visitor.visit_str(&s),
←──────  return Ok(value.to_owned())                    }
```

A String for a String

Which works like this:

```
let name_value = Value::String("CPU1".to_string());
let name: String = Deserialize::deserialize(ValueDeserializer { value: name_value })?;

impl Deserialize for String                        impl Deserializer for ValueDeserializer
┌ fn deserialize(deser)                            │
│   let visitor = StringVisitor {}                 │
│   deser.deserialize_string(.., visitor) ─calls─▶ │ fn deserialize_any(.., visitor)
│                                                      match self.value {
│   impl Visitor for StringVisitor                       ..
│   ┌ fn visit_str(value: &str) ◀────────calls────────── Value::String(s) => visitor.visit_str(&s),
◀────── return Ok(value.to_owned())                    }
```

Our First Achievement

NEW ACHIEVEMENT! You can now deserialize *from* a `HashMap` of `Value`s!

```rust
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[serde(rename_all = "PascalCase")]
pub struct Fan {
    name: String,
    active_cooling: bool,
    desired_speed: u64,
}
```

```rust
use serde::de::IntoDeserializer;

let fan_map = HashMap::from([
    ("Name", Value::String("CPU1".to_string())),
    ("ActiveCooling", Value::Bool(true)),
    ("DesiredSpeed", Value::UI4(1000u32)),
]);

let fan: Fan = Deserialize::deserialize(
    fan_map.into_deserializer()
)?;
```

[3]: Assuming we defined the needed (and trivial) `impl IntoDeserializer<'_> for Value { .. }`

That is So Fetch!

```rust
use serde::de::IntoDeserializer;

let fan_map = HashMap::from([ .. ]);

let fan: Fan = Deserialize::deserialize(
    // A MapDeserializer.
    fan_map.into_deserializer()
)?;
```

Current State of Affairs

```
fn query<T: Deserialize>()
  -> Vec<T> { .. }
```

```
impl<'de> Deserializer<'de>
for ValueDeserializer
```

```
impl<'de> MapAccess<'de> for
ObjectMapAccess { .. }
```

```
impl<'de> Deserializer<'de>
for ObjectDeserializer { .. }
```

Our Second Deserializer

```
struct ObjectDeserializer { obj: Object }

impl<'de> Deserializer<'de> for ObjectDeserializer {
    fn deserialize_struct<V>(
        self,
        name: &'static str,
        fields: &'static [&'static str],
        visitor: V,
    ) -> Result<V::Value, Self::Error>
    where V: Visitor<'de>,
    {
        let map = todo!();
        visitor.visit_map(map)
    }
}
```

A deserialization hint

`deserialize_struct` lets us know the names of all the fields we are expected to produce.

When we started to think about a reflection API in Rust, this wasn't particularly useful - but now we can actually use this!

Our MapAccess

We need to two things in our `MapAccess`-able struct - the *`fields`* and the *`obj`*:

```
struct ObjectMapAccess {
    // what we get by calling `fields.iter().peekable()`.
    fields: Peekable<Iter<'static, &'static str>>,
    obj: raw_api::Object,
}

let map = ObjectMapAccess {
    fields: fields.iter().peekable(),
    obj: self.obj,
};
```

Our MapAccess impl

```
fn next_key<K>(&mut self) -> Result<Option<K>, Self::Error>
    where K: Deserialize<'de> {
    if let Some(field) = self.fields.peek() {
        let field_deser = StrDeserializer::new(field);
        return K::deserialize(field_deser).map(Some);
    }
    Ok(None)
}

fn next_value<V>(&mut self) -> Result<V, Self::Error>
    where V: Deserialize<'de> {
    let current_field = self.fields.next().ok_or(...)?;
    let field_value = self.obj.get(current_field);
    V::deserialize(ValueDeserializer { value: field_value })
}
```

1. `peek()` the next field from the iterator.

2. `next()` the iterator, get the value, and deserialize.

Our Second Achievement

NEW ACHIEVEMENT!

You can now deserialize `Objects` into (most) `Deserialize`-able types!

```
let object: raw_api::Object = /* .. */;

let fan: Fan = Deserialize::deserialize(ObjectDeserializer { obj: object })?;
```

Current State of Affairs

```
fn query<T: Deserialize>()
  -> Vec<T> { .. }
```

```
impl<'de> Deserializer<'de>
for ValueDeserializer
```

```
impl<'de> MapAccess<'de> for
ObjectMapAccess { .. }
```

```
impl<'de> Deserializer<'de>
for ObjectDeserializer { .. }
```

A Missing Piece

Why do we care about the name of the struct, again?

```
// 1. User defines a custom struct for the type of objects to query.
struct Fan { name: String, active_cooling: bool, desired_speed: u64 }

// 2. `query` should return instances of `Fan`,
// executing "SELECT * FROM Win32_Fan".
let res: Vec<Fan> = api::query();

// 3. Profit...
```

A Missing Piece

But didn’t we get the name of the struct somewhere already?

```
struct ObjectDeserializer { obj: Object }

impl<'de> Deserializer<'de> for ObjectDeserializer {
    fn deserialize_struct<V>(
        self,
        name: &'static str,
        fields: &'static [&'static str],
        visitor: V,
    ) -> Result<V::Value, Self::Error>
    where V: Visitor<'de>,
    { .. }
}
```

It’s Not an Issue

The answer? Yet another Deserializer impl, directly from Serde issue #1110:

```rust
struct StructNameDesr<'a> {
  name: &'a mut Option<&'static str>,
}

impl<'de, 'a> Deserializer<'de> for StructNameDesr<'a> { .. }

let mut name = None;
let deser = StructNameDesr { name: &mut name };
```

It's Not an Issue

The answer? Yet another Deserializer impl, directly from Serde issue #1110:

```
struct StructNameDesr<'a> {
  name: &'a mut Option<&'static str>,
}

let mut name = None;
let _ = T::deserialize(
  StructNameDesr { name: &mut name }
);
let inferred_query = format!(
  "SELECT * FROM {}", name?,
);
```

```
fn deserialize_struct<V>(
    self,
    name: &'static str,
    fields: &'static [&'static str],
    visitor: V,
) -> Result<V::Value> where V: Visitor<'de>,
{
    *self.name = Some(name);
    Err(de::Error::custom("no butter"))
}
```

Huzza!

```
fn query<T: Deserialize>()
  -> Vec<T> { .. }
```

```
impl<'de> Deserializer<'de>
 for ValueDeserializer
```

```
impl<'de> MapAccess<'de> for
ObjectMapAccess { .. }
```

```
impl<'de> Deserializer<'de>
for ObjectDeserializer { .. }
```

How It Started vs. How It's Going

Who's bad at reflection now, buddy?

```
let res = raw_api::query("SELECT * FROM Win32_Fan");

for obj in res {
  if obj.get("ActiveCooling") == Value::Bool(true) {
    if let Value::String(name) = obj.get("Name") {
      if let Value::UI8(speed) = obj.get("DesiredSpeed") {
        println!("Fan `{name}` is running at {speed}");
      }
    }
  }
}
```

```
#[derive(Deserialize)]
#[serde(rename = "Win32_Fan")]
#[serde(rename_all = "PascalCase")]
struct Fan {
  name: String,
  active_cooling: bool,
  desired_speed: u64
}

let res: Vec<Fan> = api::query();
for fan in res {
  if fan.active_cooling {
    println!("Fan `{}` is running at {}",
      fan.name, fan.desired_speed);
  }
}
```

What is `DeserializeSeed`?

What about `async`?

What is `'de`?

Why not a proc-macro?

What about custom types?

Questions?

Why not use an ORM crate like diesel or sea-query?

Was that a DCC Reference?

What about `Value::Object(Object)` support?

What about serializing?

What about `enum` support?

Read more at ohadravid.github.io

What is 'de?

`'de` is the lifetime of the *input data*. See: [Deserializer lifetimes · Serde]

For example, `serde_json::from_str` accepts a `&'de str`.

```
pub struct BorrowedStrDeserializer<'de> { value: &'de str }
impl<'de> BorrowedStrDeserializer<'de> {
    pub fn new(value: &'de str) -> BorrowedStrDeserializer<'de> {
        BorrowedStrDeserializer { value }
    }
}
impl<'de> Visitor<'de> for StrVisitor { // `impl<'de> Deserialize<'de> for &'de str` [*]
    type Value = &'de str;
    fn visit_str<E>(self, v: &str) -> { /* Err! */ }
    fn visit_borrowed_str<E>(self, v: &'de str) -> Result<Self::Value, E> { Ok(v) }
}
```

Value::Object(Object)

Not everything is `deserialize_any` in `ValueDeserializer` after all:

```rust
impl<'de> Deserializer<'de> for ValueDeserializer {
    fn deserialize_struct<V>(
        self,
        name: &'static str,
        fields: &'static [&'static str],
        visitor: V,
    ) -> Result<V::Value, Self::Error>
    where V: serde::de::Visitor<'de>,
    {
        if let raw_api::Value::Object(obj) = self.value {
            let desr = ObjectDeserializer { obj };
            return desr.deserialize_struct(name, fields, visitor);
        }

        Err(Self::Error::custom("only a Value::Object can be deserialized to a struct"))
    }
}
```

DeserializeSeed vs. Deserialize

`DeserializeSeed` is the stateful form of the `Deserialize` trait.

```
pub trait DeserializeSeed<'de>: Sized {
    type Value;

    fn deserialize<D>(self, deserializer: D) -> Result<Self::Value, D::Error>
       where D: Deserializer<'de>;
}

pub trait Deserialize<'de>: Sized {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
       where D: Deserializer<'de>;
}

struct ExtendVec<'a, T: 'a>(&'a mut Vec<T>);
```

Enum Support

EnumAccess is the enum counterpart to MapAccess.

```
fn deserialize_enum<V>(
    self,
    name: &'static str,
    variants: &'static [&'static str],
    visitor: V,
) -> Result<V::Value, Self::Error> where V: Visitor<'de> { .. }

fn deserialize_identifier<V>(
    self,
    visitor: V
) -> Result<V::Value, Self::Error> where V: Visitor<'de>
{
    let class_name = /* ... */;
    visitor.visit_string(class_name)
}
```

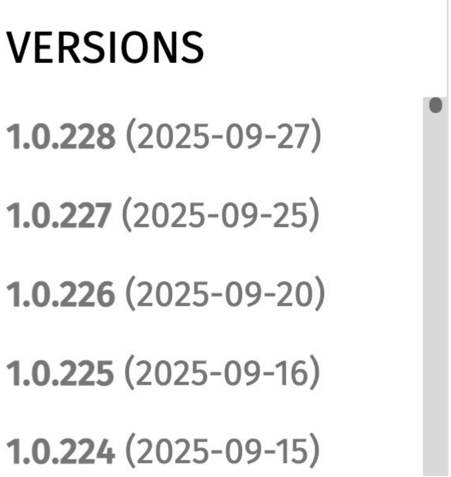
```
#[derive(Deserialize)]
enum Status { OK, Error }

#[derive(Deserialize)]
struct Win32_OperatingSystem {
  status: Status,
}

#[derive(Deserialize)]
enum User {
  #[serde(rename = "Win32_SystemAccount")]
  System(Win32_SystemAccount),
  #[serde(rename = "Win32_UserAccount")]
  User(Win32_UserAccount),
}
```

ORM vs. Serde

In over 100 versions (spanning 7 years), Serde never broke an API.

SeaQuery 0.1.0 is from 2020, currently 1.0.0-rc.14, Diesel was 1.3.3, currently 2.3.2.

WMI is a *very* limited form of SQL (e.g no JOIN support), so scope is reduced.



VERSIONS

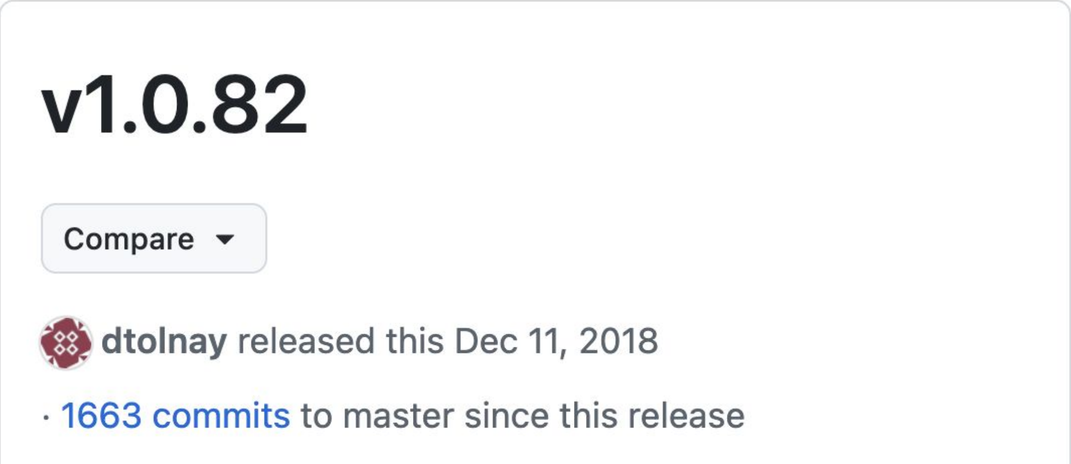
1.0.228 (2025-09-27)

1.0.227 (2025-09-25)

1.0.226 (2025-09-20)

1.0.225 (2025-09-16)

1.0.224 (2025-09-15)



v1.0.82

dtolnay released this Dec 11, 2018

· 1663 commits to master since this release

Why not a proc-macro?

Totally possible, but harder to maintain, esp. when supporting all the different edge cases (like enums, newtypes, Object properties, …)

```
struct __Visitor #impl_generics #where_clause {
    __out: miniserde::#private::Option<#ident #ty_generics>,
}
struct __State #wrapper_impl_generics #where_clause {
    #(
        #fieldname: miniserde::#private2::Option<#fieldty>,
    )*
    __out: &'__a mut miniserde::#private::Option<#ident #ty_generics>,
}
impl #wrapper_impl_generics miniserde::de::Map for __State #wrapper_ty_generics #bounded_where_clause {
    fn key(&mut self, __k: &miniserde::#private::str) -> miniserde::Result<&mut dyn miniserde::de::Visitor> {
        match __k { .. }
    }
}
```

Custom Types

Easy because Serde is built for this type of extensions

```
/// A wrapper type around `time`'s `OffsetDateTime`,
/// which supports parsing from WMI-format strings.
#[derive(Copy, Clone, Eq, PartialEq, ..)]
pub struct WMIOffsetDateTime(
  pub time::OffsetDateTime
);

struct DateTimeVisitor;

impl<'de> Visitor<'de> for DateTimeVisitor {
  type Value = WMIOffsetDateTime;

  fn visit_str<E>(self, value: &str) -> Result<Self::Value>
  {
    value.parse() // Uses the `FromStr` impl.
  }
}
```

```
impl<'de> Deserialize<'de> for WMIOffsetDateTime {
  fn deserialize<D>(deserializer: D) ->
    Result<Self, D::Error>
  where
    D: de::Deserializer<'de>,
  {
    deserializer.deserialize_str(DateTimeVisitor)
  }
}
```