# Dynamic schemas with serde

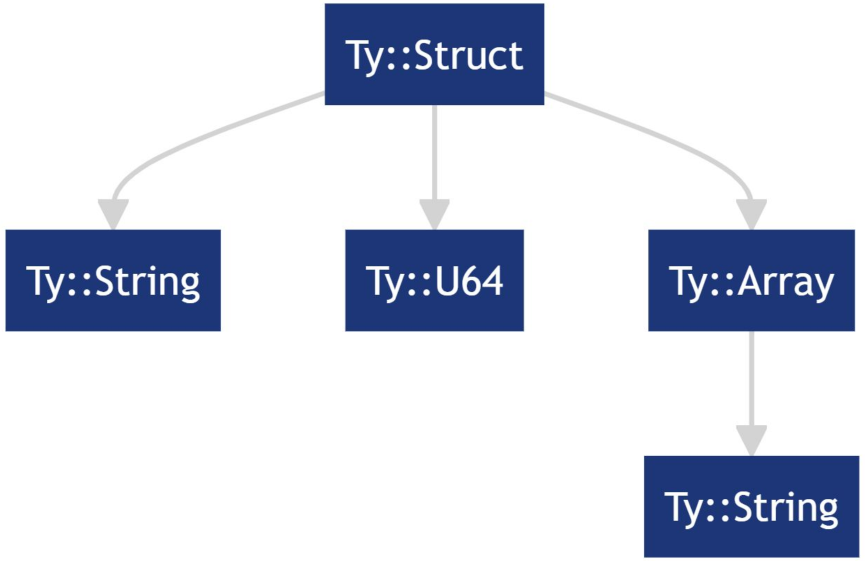
**Alexander Rafferty**

alexanderrafferty.com

github.com/Rafferty97

linkedin.com/in/adrafferty

# Dynamic schema



```rust
pub enum Ty {
    Bool,
    U64,
    I64,
    F64,
    Bytes,
    String,
    Array { inner: Box<Ty> },
    Struct { fields: Box<[Field]> },
}

pub struct Field {
    pub name: Box<str>,
    pub ty: Ty,
}
```

## The mission

Store JSON documents on disk

Validate input against the schema

Take advantage of schema to save space *(e.g. don't store object keys)*

Optimise performance

```rust
let schema = struct_def!({
    "name": Ty::String,
    "age": Ty::U64,
    "hobbies": array_def!(Ty::String),
});
```

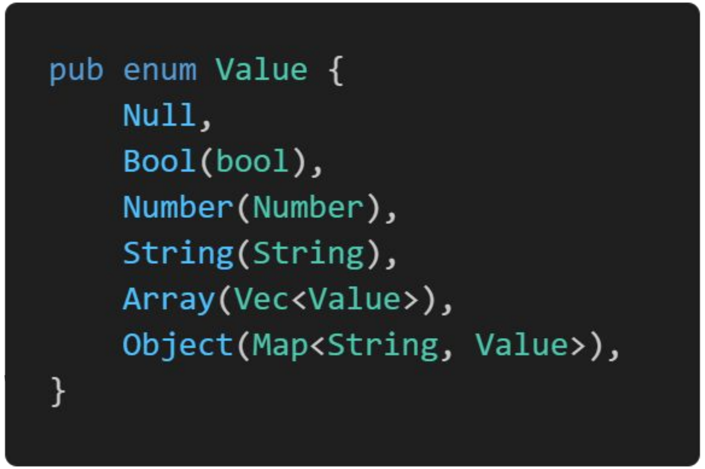
```json
{
  "name": "Alex",
  "age": 27,
  "hobbies": ["coding", "music"]
}
```

*Example schema and conforming JSON*

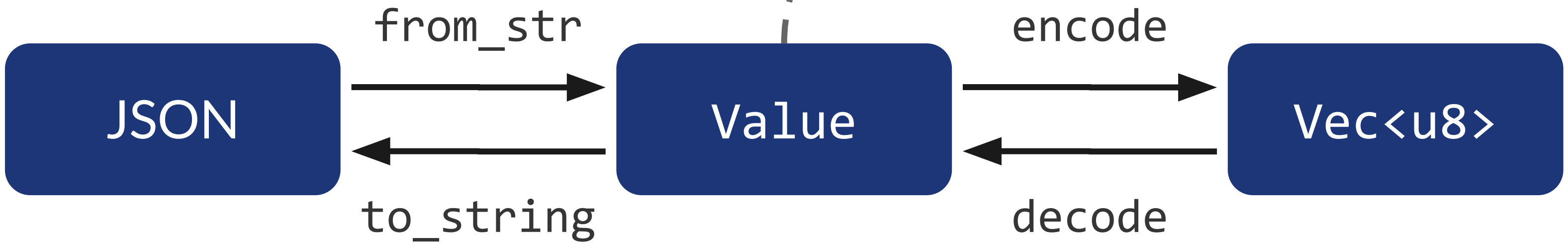
## Simple approach

Use `Value` as intermediary, as it can represent arbitrary JSON

Simple but inefficient



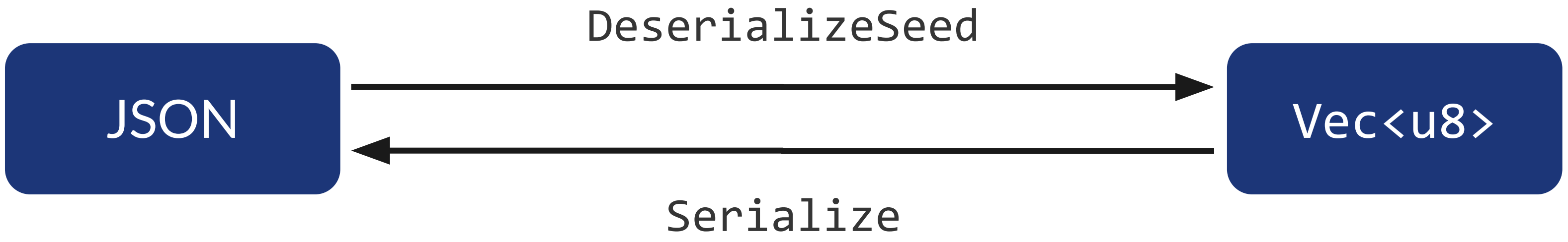
```rust
pub enum Value {
    Null,
    Bool(bool),
    Number(Number),
    String(String),
    Array(Vec<Value>),
    Object(Map<String, Value>),
}
```



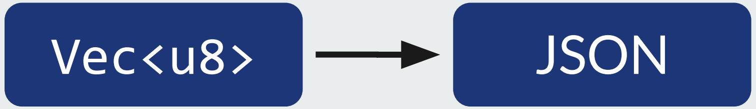
## More efficient approach

Implement `Serialize` and `DeserializeSeed` traits to inject the dynamic schema into the serialization/deserialization process

Allows us to go directly from JSON to encoded bytes and back again



## Serialization



```rust
// 1. Package the data and the schema into a single struct
struct TypedBytes<'a> {
    bytes: &'a [u8], // Binary encoding of document
    ty: &'a Ty,      // How to interpret the encoded data
}

// 2. Then implement `Serialize` on it
impl<'a> Serialize for TypedValue<'a> {
    fn serialize<S: Serializer>(&self, serializer: S) -> Result<S::Ok, S::Error> {
        // Decode `self.bytes` using `self.ty` and pass the result to `serializer`
    }
}
```

## Serialization

$$\begin{tikzcd} \boxed{\text{Vec<u8>}} \arrow[r] & \boxed{\text{JSON}} \end{tikzcd}$$

```rust
fn serialize<S: Serializer>(&self, serializer: S) -> Result<S::Ok, S::Error> {
    match self.ty {
        Ty::U64 => serializer.serialize_u64(self.bytes)
        Ty::Array { inner: ty } => {
            let mut seq = serializer.serialize_seq(None)?;
            while let Some(bytes) = read_element(self.bytes) {
                seq.serialize_element(&TypedBytes { ty, bytes })?;
            }
            seq.end()
        }
        // ...other types
    }
}
```

## Deserialization

$$\begin{tikzcd}\text{JSON} \arrow[r] & \text{Vec<u8>}\end{tikzcd}$$

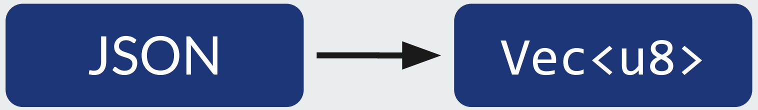
```rust
pub trait Deserialize<'de>: Sized {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: Deserializer<'de>;
}
```

Doesn’t work

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

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

Works!

## Deserialization



```rust
fn deserialize<D: Deserializer<'de>>(self, de: D) -> Result<(), D::Error> {
    match self.ty {
        Ty::Bool => de.deserialize_bool(BoolVisitor { builder }),
        Ty::U64 => de.deserialize_u64(UIntVisitor { builder }),
        Ty::I64 => de.deserialize_i64(IntVisitor { builder }),
        Ty::F64 => de.deserialize_f64(FloatVisitor { builder }),
    }
}
```

# Deserialization

$$\begin{tikzcd}\text{JSON} \arrow[r] & \text{Vec<u8>}\end{tikzcd}$$

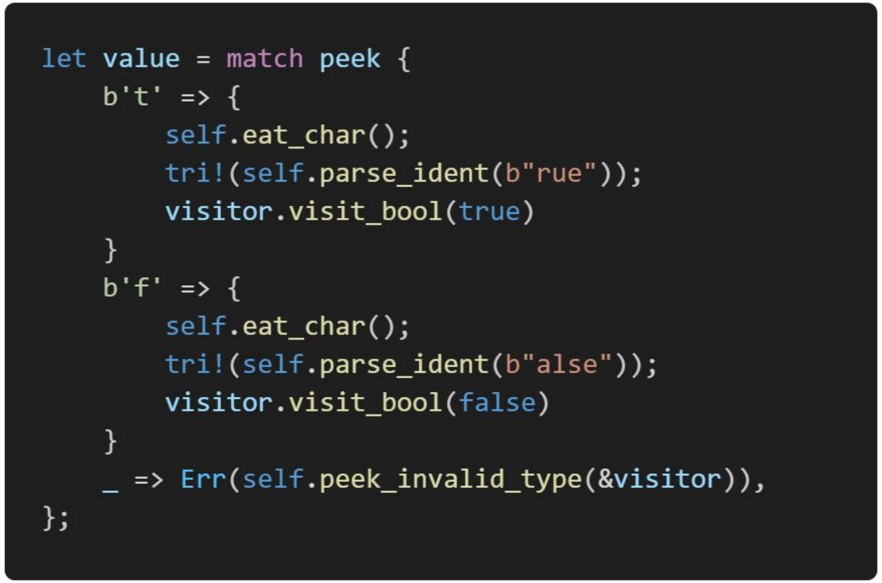
```rust
struct BoolVisitor<'a> {
    pub builder: Builder<'a>,
}

impl<'a, 'de> Visitor<'de> for BoolVisitor<'a> {
    type Value = ();

    fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        write!(formatter, "a boolean")
    }

    fn visit_bool<E: serde::de::Error>(self, value: bool) -> Result<(), E> {
        self.builder.write_bool(value);
        Ok(())
    }
}
```

## Efficient parsing

In the new approach, the schema drives the deserialization

This allows for efficient parsing of the JSON input

For example, booleans are matched with just one `char`



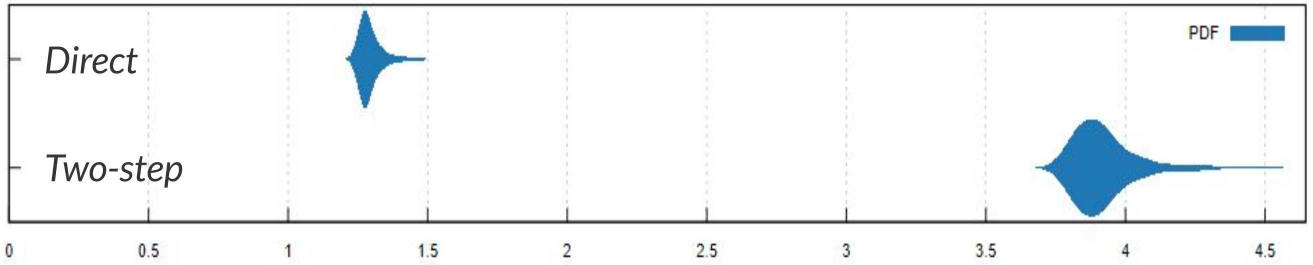
```
let value = match peek {
    b't' => {
        self.eat_char();
        tri!(self.parse_ident(b"rue"));
        visitor.visit_bool(true)
    }
    b'f' => {
        self.eat_char();
        tri!(self.parse_ident(b"alse"));
        visitor.visit_bool(false)
    }
    _ => Err(self.peek_invalid_type(&visitor)),
};
```

`serde_json: src/de.rs:1472-1484`

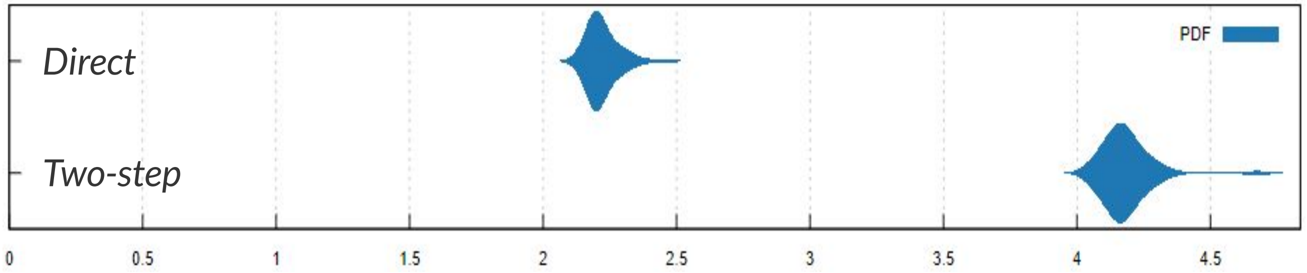
## How much faster is it?

*Average times in microseconds*

**Deserialize**
(JSON to bytes)



**Serialize**
(bytes to JSON)



## Questions?