THE JOURNEY OF HACKING IN A NEW SERDE DATAFORMAT

2024-02-03 / FOSDEM '24 / RUST DEVROOM

ABSTRACT (PARAPHRASED)

> `neo4rs` *use serde. I want to present the journey of building [that].*

WHAT THIS TALK IS NOT

WHAT THIS TALK IS NOT

- Introduction to serde

WHAT THIS TALK IS NOT

- Introduction to serde
- Introduction to neo4rs

WHAT THIS TALK IS NOT

- Introduction to serde
- Introduction to neo4rs
- An actual *deep* deep-dive

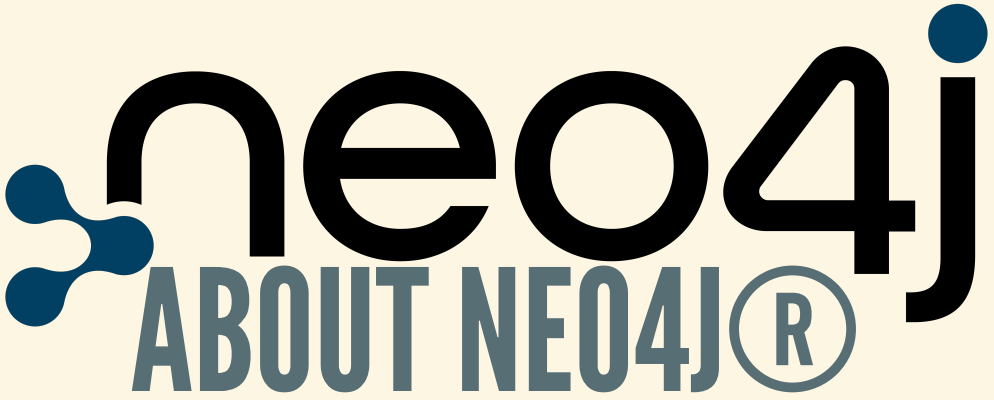
WHAT THIS TALK IS NOT

- Introduction to serde
- Introduction to neo4rs
- An actual *deep* deep-dive
- Discussion about how to pronounce serde

ABOUT ME

- Hi, I'm Paul
- @knutwalker[@hachyderm.io]


neo4j



- A Graph Database written in Not-Rust
- We have a Rust driver: `neo4rs`, written in pure Rust
- Developed under Neo4j Labs
- github.com/neo4j-labs/neo4rs

ABOUT NEO4J®

- Neo4j drivers generally comunicate via Bolt

ABOUT NEO4J®

- Neo4j drivers generally comunicate via Bolt
- Bolt is a binary protocol

ABOUT NEO4J®

- Neo4j drivers generally comunicate via Bolt
- Bolt is a binary protocol
  - packstream for general data types

ABOUT NEO4J®

- Neo4j drivers generally comunicate via Bolt
- Bolt is a binary protocol
  - packstream for general data types
  - "binary JSON-ish"

ABOUT NEO4J®

- Neo4j drivers generally comunicate via Bolt
- Bolt is a binary protocol
  - packstream for general data types
  - "binary JSON-ish"
  - Domain-specific structs (~15)

ABOUT NEO4RS

- Bolt structs as a `BoltType` enum

```
enum BoltType {
    Null,
    Integer(i64),
    String(String),
    List(Vec<BoltType>),
    Node(BoltNode),
    // ...
}
```

neo4rs 0.6

```
let event = node.get::<String>("event").unwrap();
let year = node.get::<u64>("year").unwrap();
```

neo4rs 0.7

```
let event = node.get::<String>("event").unwrap();
let year = node.get::<u64>("year").unwrap();
```

neo4rs 0.7

```rust
#[derive(serde::Deserialize)]
struct Session {
    event: String,
    year: u64,
}
```

neo4rs 0.7

```
let session = node.to::<Session>().unwrap();
```

ABOUT `serde`

- A framework for _**ser**_ializing and _**de**_serializing Rust data structures (serde.rs)
- Jon's decrusting video

ABOUT `serde`

- data type
- data format
- data model

ABOUT `serde`

- data type
- `#[derive(Serialize, Deserialize)]`
- data format

- data model

ABOUT `serde`

- data type
- `#[derive(Serialize, Deserialize)]`
- data format
- `Serializer` and `Deserializer` traits

- data model

ABOUT `serde`

- data type
- `#[derive(Serialize, Deserialize)]`
- data format
- `Serializer` and `Deserializer` traits
- notice the **r** at the end of the names
- data model

ABOUT `serde`

- data type
- `#[derive(Serialize, Deserialize)]`
- data format
- `Serializer` and `Deserializer` traits
- notice the **r** at the end of the names
- data model
- mostly represented in the API only

ABOUT `serde`

- example data format: JSON

ABOUT `serde`

- example data format: JSON
- data format implementation: `serde_json`

neo4rs MEETS serde

- In neo4rs, data is already available as BoltType

neo4rs MEETS serde

- In neo4rs, data is already available as BoltType
- Implement a data format for BoltType

neo4rs MEETS serde

- In neo4rs, data is already available as BoltType
- Implement a data format for BoltType
- Deserializer only

neo4rs MEETS serde

- In neo4rs, data is already available as BoltType
- Implement a data format for BoltType
- Deserializer only
- Maintain API compatibility, if possible

NODE

```
Node::Structure(
    id::Integer,
    labels::List<String>,
    properties::Dictionary,
)
```

NODE

```
pub struct BoltNode {
    pub id: BoltInteger,
    pub labels: BoltList,
    pub properties: BoltMap,
}
```

EXAMPLE

```
CREATE (n:Session { event: 'FOSDEM', year: 2024 })
RETURN n
```

EXAMPLE

```
CREATE (n
    :Session
    {
        event: 'FOSDEM',
         year: 2024
    }
) RETURN n
```

EXAMPLE

```
CREATE (n
    :Session
    {
        event: 'FOSDEM',
         year: 2024
    }
) RETURN n
```

EXAMPLE

```
CREATE (n
    :Session
    {
        event: 'FOSDEM',
         year: 2024
    }
) RETURN n
```

EXAMPLE

```
CREATE (n
    :Session
    {
        event: 'FOSDEM',
         year: 2024
    }
) RETURN n
```

EXAMPLE

```
#[derive(Deserialize)]
struct Session {
    event: String,
    year: u64,
}
```

EXAMPLE

```
let session = row.get::<Session>("n").unwrap();

let node = row.get::<Node>("n").unwrap();
let session = node.to::<Session>().unwrap();
```

EXAMPLE

```
1 let session = row.get::<Session>("n").unwrap();
2
3 let node = row.get::<Node>("n").unwrap();
4 let session = node.to::<Session>().unwrap();
```

EXAMPLE

```
1 let session = row.get::<Session>("n").unwrap();
2
3 let node = row.get::<Node>("n").unwrap();
4 let session = node.to::<Session>().unwrap();
```

FIRST ATTEMPT

```
#[derive(Deserialize, Serialize)]
pub struct BoltNode {
    pub id: BoltInteger,
    pub labels: BoltList,
    pub properties: BoltMap,
}
```

FIRST ATTEMPT

```
#[derive(Deserialize, Serialize)]
pub struct BoltNode {
    pub id: BoltInteger,
    pub labels: BoltList,
    pub properties: BoltMap,
}
```

FIRST ATTEMPT

```
1 pub fn to<T: Deserialize>(self) -> Result<T, Error> {
2     let value = serde_json::to_value(self)?;
3     let result = serde_json::from_value(value)?;
4     Ok(result)
5 }
```

FIRST ATTEMPT

```
1 pub fn to<T: Deserialize>(self) -> Result<T, Error> {
2     let value = serde_json::to_value(self)?;
3     let result = serde_json::from_value(value)?;
4     Ok(result)
5 }
```

FIRST ATTEMPT

```
1 pub fn to<T: Deserialize>(self) -> Result<T, Error> {
2     let value = serde_json::to_value(self)?;
3     let result = serde_json::from_value(value)?;
4     Ok(result)
5 }
```

FIRST ATTEMPT

```
1 pub fn to<T: Deserialize>(self) -> Result<T, Error> {
2     let value = serde_json::to_value(self)?;
3     let result = serde_json::from_value(value)?;
4     Ok(result)
5 }
```

FIRST ATTEMPT

Are we done?

FIRST ATTEMPT

Are we done?

```
missing field `event`
missing field `year`
```

FIRST ATTEMPT

```
#[derive(Deserialize, Serialize)]
pub struct BoltNode {
    pub id: BoltInteger,
    pub labels: BoltList,
    pub properties: BoltMap,
}
```

FIRST ATTEMPT

```
#[derive(Deserialize)]
struct SessionNode {
    properties: Session
}
```

SECOND ATTEMPT

```
1 pub fn to<T: Deserialize>(self) -> Result<T, Error> {
2     let value = serde_json::to_value(self.properties)?;
3     let result = serde_json::from_value(value)?;
4     Ok(result)
5 }
```

SECOND ATTEMPT

- Kinda works
- No way to get id and labels

THIRD ATTEMPT

```
#[derive(Deserialize)]
struct Session {
    event: String,
    year: u64,
    id: u64,
    labels: Vec<String>,
}
```

THIRD ATTEMPT

```
#[derive(Deserialize)]
struct Session {
    event: String,
    year: u64,
    id: u64,
    labels: Vec<String>,
}
```

SMALL EXCURSION INTO `serde`

```
impl Deserialize for Session {
    fn deserialize<D>(deserializer: D)
     -> Result<Self, D::Error>
    where
        D: Deserializer,
    {
        todo!()
    }
}
```

SMALL EXCURSION INTO `serde`

```
impl Deserialize for Session {
    fn deserialize<D>(deserializer: D)
     -> Result<Self, D::Error>
    where
        D: Deserializer,
    {
        todo!()
    }
}
```

SMALL EXCURSION INTO `serde`

```
impl Deserialize for Session {
    fn deserialize<D>(deserializer: D)
     -> Result<Self, D::Error>
    where
        D: Deserializer,
    {
        todo!()
    }
}
```

SMALL EXCURSION INTO `serde`

```
impl Deserialize for Session {
    fn deserialize<D>(deserializer: D)
     -> Result<Self, D::Error>
    where
        D: Deserializer,
    {
        todo!()
    }
}
```

SMALL EXCURSION INTO `serde`

```
impl Deserialize for Session {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Erro
        deserializer.deserialize_struct(
            "Session",
            &["event", "year", "id", "labels"],
            todo!("Visitor"),
        )
    }
}
```

SMALL EXCURSION INTO `serde`

```
impl Deserialize for Session {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Erro
        deserializer.deserialize_struct(
            "Session",
            &["event", "year", "id", "labels"],
            todo!("Visitor"),
        )
    }
}
```

SMALL EXCURSION INTO `serde`

```
1 struct SessionVisitor;
2
3 impl Visitor for SessionVisitor {
4     type Value = Session;
5
6     fn expecting(&self, formatter: &mut Formatter) -> fmt::Res
7         Formatter::write_str(formatter, "struct Session")
8     }
9 }
```

SMALL EXCURSION INTO `serde`

```
1 struct SessionVisitor;
2
3 impl Visitor for SessionVisitor {
4     type Value = Session;
5
6     fn expecting(&self, formatter: &mut Formatter) -> fmt::Res
7         Formatter::write_str(formatter, "struct Session")
8     }
9 }
```

SMALL EXCURSION INTO `serde`

```
1 struct SessionVisitor;
2
3 impl Visitor for SessionVisitor {
4     type Value = Session;
5
6     fn expecting(&self, formatter: &mut Formatter) -> fmt::Res
7         Formatter::write_str(formatter, "struct Session")
8     }
9 }
```

SMALL EXCURSION INTO `serde`

```
1 struct SessionVisitor;
2
3 impl Visitor for SessionVisitor {
4     type Value = Session;
5
6     fn expecting(&self, formatter: &mut Formatter) -> fmt::Res
7         Formatter::write_str(formatter, "struct Session")
8     }
9 }
```

SMALL EXCURSION INTO `serde`

```
impl Visitor for SessionVisitor {
    fn visit_map<A>(self, mut map: A)
     -> Result<Self::Value, A::Error>
    where
        A: MapAccess,
    {
        todo!()
    }
}
```

SMALL EXCURSION INTO `serde`

```
impl Visitor for SessionVisitor {
    fn visit_map<A>(self, mut map: A)
     -> Result<Self::Value, A::Error>
    where
        A: MapAccess,
    {
        todo!()
    }
}
```

SMALL EXCURSION INTO `serde`

```
impl Visitor for SessionVisitor {
    fn visit_map<A>(self, mut map: A)
     -> Result<Self::Value, A::Error>
    where
        A: MapAccess,
    {
        todo!()
    }
}
```

SMALL EXCURSION INTO `serde`

```
fn visit_map<A: MapAccess>(self, mut map: A) -> Result<Self::V
    let mut event: Option<String> = None;
    let mut year: Option<u64> = None;
    let mut id: Option<u64> = None;
    let mut labels: Option<Vec<String>> = None;
}
```

SMALL EXCURSION INTO `serde`

```
fn visit_map<A: MapAccess>(self, mut map: A) -> Result<Self::
    while let Some(key) = map.next_key::<&str>()? {
        match key {
            "event" => event =
                Some(map.next_value::<String>()?),
            "year" => year = Some(map.next_value::<u64>()?),
            "id" => id = Some(map.next_value::<u64>()?),
            "labels" => labels = Some(map.next_value::<Vec<St
            _ => todo!("unknown field"),
        }
    }
}
```

SMALL EXCURSION INTO `serde`

```
fn visit_map<A: MapAccess>(self, mut map: A) -> Result<Self::
    while let Some(key) = map.next_key::<&str>()? {
        match key {
            "event" => event =
                Some(map.next_value::<String>()?),
            "year" => year = Some(map.next_value::<u64>()?),
            "id" => id = Some(map.next_value::<u64>()?),
            "labels" => labels = Some(map.next_value::<Vec<St
            _ => todo!("unknown field"),
        }
    }
}
```

SMALL EXCURSION INTO `serde`

```
fn visit_map<A: MapAccess>(self, mut map: A) -> Result<Self::
    while let Some(key) = map.next_key::<&str>()? {
        match key {
            "event" => event =
                Some(map.next_value::<String>()?),
            "year" => year = Some(map.next_value::<u64>()?),
            "id" => id = Some(map.next_value::<u64>()?),
            "labels" => labels = Some(map.next_value::<Vec<St
            _ => todo!("unknown field"),
        }
    }
}
```

SMALL EXCURSION INTO `serde`

```
fn visit_map<A: MapAccess>(self, mut map: A) -> Result<Self::
    while let Some(key) = map.next_key::<&str>()? {
        match key {
            "event" => event =
                Some(map.next_value::<String>()?),
            "year" => year = Some(map.next_value::<u64>()?),
            "id" => id = Some(map.next_value::<u64>()?),
            "labels" => labels = Some(map.next_value::<Vec<St
            _ => todo!("unknown field"),
        }
    }
}
```

SMALL EXCURSION INTO `serde`

```
fn visit_map<A: MapAccess>(self, mut map: A) -> Result<Self::
    while let Some(key) = map.next_key::<&str>()? {
        match key {
            "event" => event =
                Some(map.next_value::<String>()?),
            "year" => year = Some(map.next_value::<u64>()?),
            "id" => id = Some(map.next_value::<u64>()?),
            "labels" => labels = Some(map.next_value::<Vec<St
            _ => todo!("unknown field"),
        }
    }
}
```

SMALL EXCURSION INTO `serde`

```
fn visit_map<A: MapAccess>(self, mut map: A) -> Result<Self::
    Ok(Session {
        event: event.ok_or_else(
            || Error::missing_field("event")
        )?,
        year: year.ok_or_else(|| Error::missing_field("year")
        id: id.ok_or_else(|| Error::missing_field("id")),
        labels: labels.ok_or_else(|| Error::missing_field("la
    })
}
```

SMALL EXCURSION INTO `serde`

```
fn visit_map<A: MapAccess>(self, mut map: A) -> Result<Self::
    Ok(Session {
        event: event.ok_or_else(
            || Error::missing_field("event")
        )?,
        year: year.ok_or_else(|| Error::missing_field("year")
        id: id.ok_or_else(|| Error::missing_field("id")),
        labels: labels.ok_or_else(|| Error::missing_field("la
    })
}
```

SMALL EXCURSION INTO `serde`

```
fn visit_map<A: MapAccess>(self, mut map: A) -> Result<Self::
    Ok(Session {
        event: event.ok_or_else(
            || Error::missing_field("event")
        )?,
        year: year.ok_or_else(|| Error::missing_field("year")
        id: id.ok_or_else(|| Error::missing_field("id")),
        labels: labels.ok_or_else(|| Error::missing_field("la
    })
}
```

SMALL EXCURSION INTO `serde`

```
1 impl Deserialize for Session {
2     fn deserialize<D>(deserializer: D) -> Result<Self, D::Erro
3         deserializer.deserialize_struct(
4             "Session",
5             &["event", "year"],
6             SessionVisitor,
7         )
8     }
9 }
```

THIRD ATTEMPT

```
struct BoltNodeDeserializer(BoltNode);

impl IntoDeserializer<DeError> for BoltNode {
    type Deserializer = BoltNodeDeserializer;

    fn into_deserializer(self) -> Self::Deserializer {
        BoltNodeDeserializer::new(self)
    }
}
```

THIRD ATTEMPT

```
struct BoltNodeDeserializer(BoltNode);

impl IntoDeserializer<DeError> for BoltNode {
    type Deserializer = BoltNodeDeserializer;

    fn into_deserializer(self) -> Self::Deserializer {
        BoltNodeDeserializer::new(self)
    }
}
```

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    type Error = DeError;

    forward_to_deserialize_any! {
        bool i8 i16 i32 i64 i128 u8 u16 u32 u64 u128
        f32 f64 char str string bytes byte_buf option
        unit unit_struct seq tuple tuple_struct struct
        identifier enum map ignored_any newtype_struct
    }
}
```

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    type Error = DeError;

    forward_to_deserialize_any! {
        bool i8 i16 i32 i64 i128 u8 u16 u32 u64 u128
        f32 f64 char str string bytes byte_buf option
        unit unit_struct seq tuple tuple_struct struct
        identifier enum map ignored_any newtype_struct
    }
}
```

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_any<V>(self, visitor: V)
     -> Result<V::Value, Self::Error>
    where
        V: Visitor,
    {
        visitor.visit_map(MapDeserializer::new(
            self.0.properties.iter()
        ))
    }
}
```

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_any<V>(self, visitor: V)
     -> Result<V::Value, Self::Error>
    where
        V: Visitor,
    {
        visitor.visit_map(MapDeserializer::new(
            self.0.properties.iter()
        ))
    }
}
```

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_any<V>(self, visitor: V)
     -> Result<V::Value, Self::Error>
    where
        V: Visitor,
    {
        visitor.visit_map(MapDeserializer::new(
            self.0.properties.iter()
        ))
    }
}
```

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_any<V>(self, visitor: V)
     -> Result<V::Value, Self::Error>
    where
        V: Visitor,
    {
        visitor.visit_map(MapDeserializer::new(
            self.0.properties.iter()
        ))
    }
}
```

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_any<V>(self, visitor: V)
     -> Result<V::Value, Self::Error>
    where
        V: Visitor,
    {
        visitor.visit_map(MapDeserializer::new(
            self.0.properties.iter()
        ))
    }
}
```

THIRD ATTEMPT

```
1 fn to<T: Deserialize>(self) -> Result<T, DeError> {
2     T::deserialize(self.into_deserializer())
3 }
```

THIRD ATTEMPT

- We now have the same result as before

THIRD ATTEMPT

- We now have the same result as before
- Let's add `id` and `labels`

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_any<V>(self, visitor: V)
     -> Result<V::Value, Self::Error> {
        let properties = self.0.properties.iter();
        let properties = properties.chain([
            ("id", &self.0.id),
            ("labels", &self.0.labels),
        ]);
        visitor.visit_map(MapDeserializer::new(properties))
    }
}
```

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_any<V>(self, visitor: V)
     -> Result<V::Value, Self::Error> {
        let properties = self.0.properties.iter();
        let properties = properties.chain([
            ("id", &self.0.id),
            ("labels", &self.0.labels),
        ]);
        visitor.visit_map(MapDeserializer::new(properties))
    }
}
```

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_any<V>(self, visitor: V)
     -> Result<V::Value, Self::Error> {
        let properties = self.0.properties.iter();
        let properties = properties.chain([
            ("id", &self.0.id),
            ("labels", &self.0.labels),
        ]);
        visitor.visit_map(MapDeserializer::new(properties))
    }
}
```

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_any<V>(self, visitor: V)
     -> Result<V::Value, Self::Error> {
        let properties = self.0.properties.iter();
        let properties = properties.chain([
            ("id", &self.0.id),
            ("labels", &self.0.labels),
        ]);
        visitor.visit_map(MapDeserializer::new(properties))
    }
}
```

THIRD ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_any<V>(self, visitor: V)
     -> Result<V::Value, Self::Error> {
        let properties = self.0.properties.iter();
        let properties = properties.chain([
            ("id", &self.0.id),
            ("labels", &self.0.labels),
        ]);
        visitor.visit_map(MapDeserializer::new(properties))
    }
}
```

THIRD ATTEMPT

- Kinda works

THIRD ATTEMPT

- Kinda works
- We do get id and labels, but:

THIRD ATTEMPT

- Kinda works
- We do get id and labels, but:
- We only map them by the names `id` and `labels`

THIRD ATTEMPT

- Kinda works
- We do get id and labels, but:
- We only map them by the names `id` and `labels`
- We could use something like `___id`, but, meh

THIRD ATTEMPT

- Kinda works
- We do get id and labels, but:
- We only map them by the names `id` and `labels`
- We could use something like `___id`, but, meh
- Let's try something else instead

FOURTH ATTEMPT

```
struct Id(pub u64);
struct Labels(pub Vec<String>);
```

FOURTH ATTEMPT

```
#[derive(Deserialize)]
struct Session {
    event: String,
    year: u64,
    id: neo4rs::Id,
    labels: neo4rs::Labels,
}
```

FOURTH ATTEMPT

```
#[derive(Deserialize)]
struct Session {
    event: String,
    year: u64,
    id: neo4rs::Id,
    labels: neo4rs::Labels,
}
```

FOURTH ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_struct<V>(
        self,
        _name: &'static str,
        fields: &'static [&'static str],
        visitor: V,
    ) -> Result<V::Value, Self::Error>
    where
        V: Visitor,
    {
        visitor.visit_map(MapDeserializer::new(todo!()))
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_struct<V>(
        self,
        _name: &'static str,
        fields: &'static [&'static str],
        visitor: V,
    ) -> Result<V::Value, Self::Error>
    where
        V: Visitor,
    {
        visitor.visit_map(MapDeserializer::new(todo!()))
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_struct<V>(self, fields: &[&str], visitor:
        let property_fields = self.0.properties
            .iter()
            .map(|(k, v)| (k, StructData::Property(v)));

        let additional_fields = fields
            .iter()
            .copied()
            .filter(|f| !self.0.properties.contains_key(*f))
            .map(|f| (f, StructData::Node(self.0)));
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_struct<V>(self, fields: &[&str], visitor:
        let property_fields = self.0.properties
            .iter()
            .map(|(k, v)| (k, StructData::Property(v)));

        let additional_fields = fields
            .iter()
            .copied()
            .filter(|f| !self.0.properties.contains_key(*f))
            .map(|f| (f, StructData::Node(self.0)));
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_struct<V>(self, fields: &[&str], visitor:
        let property_fields = self.0.properties
            .iter()
            .map(|(k, v)| (k, StructData::Property(v)));

        let additional_fields = fields
            .iter()
            .copied()
            .filter(|f| !self.0.properties.contains_key(*f))
            .map(|f| (f, StructData::Node(self.0)));
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_struct<V>(self, fields: &[&str], visitor:
        let property_fields = self.0.properties
            .iter()
            .map(|(k, v)| (k, StructData::Property(v)));

        let additional_fields = fields
            .iter()
            .copied()
            .filter(|f| !self.0.properties.contains_key(*f))
            .map(|f| (f, StructData::Node(self.0)));
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_struct<V>(self, fields: &[&str], visitor: V
        // ...
        let node_fields =
            property_fields.chain(additional_fields);

        visitor.visit_map(MapDeserializer::new(node_fields))
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_struct<V>(self, fields: &[&str], visitor: V
        // ...
        let node_fields =
            property_fields.chain(additional_fields);

        visitor.visit_map(MapDeserializer::new(node_fields))
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for AdditionalNodeData {
    fn deserialize_newtype_struct<V>(
        self,
        name: &str,
        visitor: V
    ) -> Result<V::Value, Self::Error>
    where
        V: Visitor,
    {
        todo!()
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for AdditionalNodeData {
    fn deserialize_newtype_struct<V>(
        self,
        name: &str,
        visitor: V
    ) -> Result<V::Value, Self::Error>
    where
        V: Visitor,
    {
        todo!()
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for AdditionalNodeData {
    fn deserialize_newtype_struct<V: Visitor>(self, name: &st
        match name {
            "Id" => visitor.visit_newtype_struct(
                self.data.id.into_deserializer()
            ),
            "Labels" => visitor.visit_newtype_struct(
                SeqDeserializer::new(
                    self.data.labels.iter()
                )
            ),
            _ => todo!()
        }
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for AdditionalNodeData {
    fn deserialize_newtype_struct<V: Visitor>(self, name: &st
        match name {
            "Id" => visitor.visit_newtype_struct(
                self.data.id.into_deserializer()
            ),
            "Labels" => visitor.visit_newtype_struct(
                SeqDeserializer::new(
                    self.data.labels.iter()
                )
            ),
            _ => todo!()
        }
    }
}
```

FOURTH ATTEMPT

```
impl Deserializer for AdditionalLinkedData {
    fn deserialize_newtype_struct<V: Visitor>(self, name: &st
        match name {
            "Id" => visitor.visit_newtype_struct(
                self.data.id.into_deserializer()
            ),
            "Labels" => visitor.visit_newtype_struct(
                SeqDeserializer::new(
                    self.data.labels.iter()
                )
            ),
            _ => todo!()
        }
    }
}

}
```

FOURTH ATTEMPT

- Works

FOURTH ATTEMPT

- Works
- There are still some downsides

FOURTH ATTEMPT

- Works
- There are still some downsides
- `#[serde(default)]` doesn't work

#[serde(default)]

```
fn visit_map<A: MapAccess>(self, mut map: A) -> Result<Self::
    let mut year: Option<u64> = None;
    while let Some(key) = map.next_key::<&str>()? {
        match key {
            "year" => year = Some(map.next_value::<u64>()?),
        }
    }
    Ok(Session {
        year: year.unwrap_or_else(|| Default::default())?,
    })
}
```

#[serde(default)]

```
fn visit_map<A: MapAccess>(self, mut map: A) -> Result<Self::
    let mut year: Option<u64> = None;
    while let Some(key) = map.next_key::<&str>()? {
        match key {
            "year" => year = Some(map.next_value::<u64>()?),
        }
    }
    Ok(Session {
        year: year.unwrap_or_else(|| Default::default())?,
    })
}
```

#[serde(default)]

```
impl Deserializer for BoltNodeDeserializer {
    fn deserialize_struct<V>(self, fields: &[&str], visitor: V
        let additional_fields = fields
            .iter()
            .copied()
            .filter(|f| !self.0.properties.contains_key(*f))
            .map(|f| (f, StructData::Node(self.0)));
    }
}
```

FOURTH ATTEMPT

- Works
- There are still some downsides
- `#[serde(default)]` doesn't work

FOURTH ATTEMPT

- Works
- There are still some downsides
- `#[serde(default)]` doesn't work
- Workaround: `Option<T>`

`BoltType`

- Rinse and repeat for `BoltType` and its ~20 variants

`BoltType`

- Rinse and repeat for `BoltType` and its ~20 variants
- Almost...

BoltType::Bytes

```
impl Deserializer for BoltTypeDeserializer {
    fn deserialize_any<V>(self, visitor: V) -> Result<V::Valu
        match self.value {
            BoltType::String(v) =>
                visitor.visit_string(v.clone()),
            BoltType::Bytes(v) =>
                visitor.visit_bytes(v.clone()),
            // ...
        }
    }
}
```

BoltType::Bytes

```
impl Deserializer for BoltTypeDeserializer {
    fn deserialize_any<V>(self, visitor: V) -> Result<V::Valu
        match self.value {
            BoltType::String(v) =>
                visitor.visit_string(v.clone()),
            BoltType::Bytes(v) =>
                visitor.visit_bytes(v.clone()),
            // ...
        }
    }
}
```

BoltType::Bytes

```
impl Deserializer for BoltTypeDeserializer {
    fn deserialize_any<V>(self, visitor: V) -> Result<V::Valu
        match self.value {
            BoltType::String(v) =>
                visitor.visit_string(v.clone()),
            BoltType::Bytes(v) =>
                visitor.visit_bytes(v.clone()),
            // ...
        }
    }
}
```

BoltType::Bytes

```
impl Deserializer for BoltTypeDeserializer {
    fn deserialize_any<V>(self, visitor: V) -> Result<V::Value
        match self.value {
            BoltType::Bytes(v) => visitor.visit_seq(SeqDeseria
            // ...
        }
    }
}
```

BoltType::Bytes

```
impl Deserializer for BoltTypeDeserializer {
    fn deserialize_bytes<V>(self, visitor: V) -> Result<V::Va
    where
        V: Visitor,
    {
        if let BoltType::Bytes(v) = self.value {
            visitor.visit_bytes(v.clone())
        } else {
            self.unexpected(visitor)
        }
    }
}
```

BoltType::Bytes

```
impl Deserializer for BoltTypeDeserializer {
    fn deserialize_bytes<V>(self, visitor: V) -> Result<V::Va
    where
        V: Visitor,
    {
        if let BoltType::Bytes(v) = self.value {
            visitor.visit_bytes(v.clone())
        } else {
            self.unexpected(visitor)
        }
    }
}
```

BoltType::Bytes

```
impl Deserializer for BoltTypeDeserializer {
    fn deserialize_bytes<V>(self, visitor: V) -> Result<V::Va
    where
        V: Visitor,
    {
        if let BoltType::Bytes(v) = self.value {
            visitor.visit_bytes(v.clone())
        } else {
            self.unexpected(visitor)
        }
    }
}
```

ARE WE DONE NOW?

- No, but out of time 😅

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Maintain existing API:

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Maintain existing API:
  - Have special cases for converting into chrono/time types

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Maintain existing API:
  - Have special cases for converting into chrono/time types
    - `is_human_readable`

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Maintain existing API:
  - Have special cases for converting into chrono/time types
    - `is_human_readable`
    - different precision for timestamps

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Maintain existing API:

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Maintain existing API:
  - Need to deserialize into itself

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Maintain existing API:
  - Need to deserialize into itself
    - Custom `Deserialize` impl for `BoltType`

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Maintain existing API:
  - Need to deserialize into itself
    - Custom `Deserialize` impl for `BoltType`
    - That impl is not really usable for other data formats

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Allow for unexpected fields

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Allow for unexpected fields
    - `deserialize_ignored_any`

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Allow for zero-copy deserialization

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Allow for zero-copy deserialization
    - Keep `'de` lifetimes around

ARE WE DONE NOW?

- No, but out of time 😅
- Here are more things to consider:
- Allow for zero-copy deserialization
  - Keep `'de` lifetimes around
  - implement all the `_borrowed` methods

THANK YOU!

Q/A?

```
use std::process::exit;

exit(42);
```