Transistor, a bitemporal database client

v1.4
Rust in POA

Julia Naomi

- Rust evangelist @nubank
- Hobbist game developer
- https://github.com/naomijub
- https://twitter.com/GirlGameDev

Otávio Pace

- Bug developer @luizalabs
- Rustacean on spare time
- https://github.com/otaviopace

Agenda

- What is Crux?
- What is transistor?
- What is Edn?
- Why an Edn Ecosystem?
- edn-rs
- edn-derive

What is Crux?

Crux

- Bitemporal
- Schemaless
- It uses Datalog or SQL for queries
- Unbundled/Decoupled

Temporality

insert →

```
{:crux.db/id 1
 :product "cool shoes"
 :quantity 42}
```

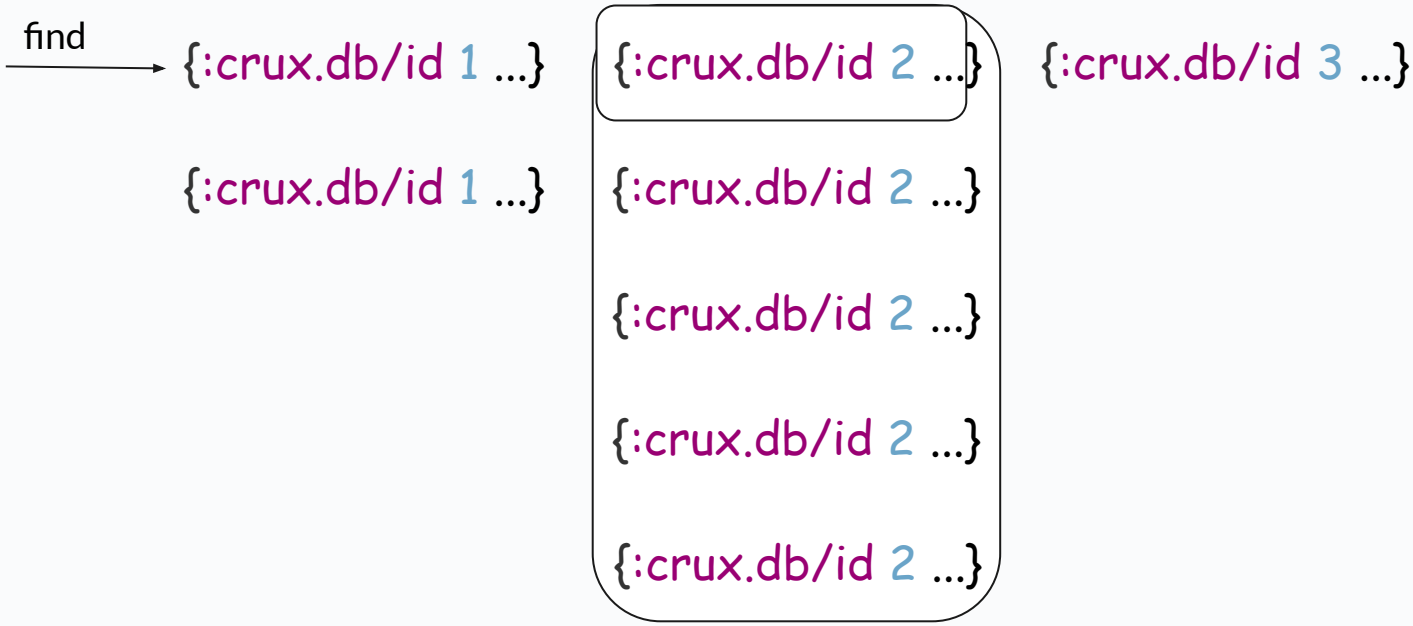
```
{:crux.db/id 1
 :product "cool shoes"
 :quantity 42
 :transaction-time
"2020-07-20T12:00:00.000-00:00"}
```

update →

```
{:crux.db/id 1
 :product "cool shoes"
 :quantity 10}
```

```
{:crux.db/id 1
 :product "cool shoes"
 :quantity 10
 :transaction-time
"2020-07-20T15:00:00.000-00:00"}
```

Temporality



Bitemporality

```
{:crux.db/id 1
 :product “cool shoes”
 :quantity 42
 :transaction-time
“2020-07-20T12:00:00.000-00:00”
 :valid-time “2020-07-20T16:00:00.000-00:00”}
```

Transaction operations

- put: Inserts a new version of a document
- delete: Deletes a document at a given :valid-time
- match: Checks the document state against the given document
- evict: Evicts a document entirely, including all it’s historical versions

Transaction operations

```
enum Action {
    Put(String, Option<DateTime<FixedOffset>>),
    Delete(String, Option<DateTime<FixedOffset>>),
    Match(String, Option<DateTime<FixedOffset>>),
    Evict(String),
}
```

What is transistor?

transistor 1.3.2

Follow

Documentation Repository Dependent crates

Cargo.toml `transistor = "1.3.2"`

Transistor

A Rust Crux Client crate/lib. For now, this crate intends to support 2 ways to interact with Crux:

- ☑ Via `Docker` with a `crux-standalone` version docker-hub. Current Docker image `juxt/crux-standalone:20.07-1.10.0`.
- ☑ Via `HTTP` using the `REST API`.
- ☑ Async support.

Last Updated
10 days ago

Crate Size
37 kB

Authors

- Julia Naomi
- Otavio Pace

License

LGPL-3.0

HTTP Client

```
use transistor::client::Crux;

// Simple
let client = Crux::new("127.0.0.1",
"3000").http_client();


// With Authorization
let client = Crux::new("127.0.0.1", "3000")
    .with_authorization("my-auth-token")
    .http_client();
```

Inserting/Updating

```
let julia = Person {
    crux__db___id: CruxId::new("1"),
    name: "Julia".to_string(),
};

let action = Action::Put(edn_rs::to_string(julia), None);

let _ = client.tx_log(vec![action])?;
```

EDN flavored Datalog

```
{:query
    {:find [?p]
     :where [[?p :name "Julia"]]
     :full-results? true}}
```

```
SELECT * FROM
"Persons"
WHERE name = "Julia";
```

- :find -> SELECT
- :where -> WHERE
- :name -> name
- :full-results -> *

Querying

```
let query = Query::find(vec!["?p"])?
    .where_clause(vec!["?p :name \"Julia\""])?
    .with_full_results()
    .build()?;

let julia = client.query(query)?;
```

```
{:query
    {:find [?p]
     :where [[?p :name
"Julia"]]
     :full-results? true}}
```

Entity - find by id

```
let edn_body = client.entity(":hello-entity")?;
// Edn Body = Map(Map({
//      ":crux.db/id": Key(":hello-entity"),
//      ":first-name": Str("Hello"),
//      ":last-name": Str("World"),
// }))
```

Async/await

- The client was first built with **reqwest::blocking**
- Then we implemented the async support for **edn-rs**
- With the **edn-rs** support we were able to start using **reqwest** async
- Then we created a Rust **async** feature in the library

Example

```
let julia = Person {
    crux__db___id: CruxId::new("1"),
    name: "Julia".to_string(),
};

let action = Action::Put(edn_rs::to_string(julia), None);

let _ = client.tx_log(vec![action])?.await;
```

What is EDN?

EDN

- Stands for Extensible Data Notation
- It is a subset of the Clojure Programming Language
- The base can be easily mapped for similar JSON, but it has a lot more constructs/features
- The whole spec is here: https://github.com/edn-format/edn

Examples

```
nil

true
false

"a
string"
 \c

+ - * . ! ...
```

```
:a-keyword

42
3.9

{:crux.db/id 1
 :product "cool shoes"
 :quantity 42}
```

```
("a" "list" "of"
"strings")

["a" "vector" "of"
"strings"]

#{"a" "set" "of"
"strings"}
```

Why an EDN ecosystem?

Libraries we've made

- transistor
- edn-rs
- edn-derive

edn-rs

edn-rs 0.13.2

Follow

Documentation Repository Dependent crates

Cargo.toml

```
edn-rs = "0.13.2"
```

edn-rs

Near Stable

Crate to parse and emit EDN build passing

- **This lib does not make effort to conform the EDN received to EDN Spec.** The lib that generated this EDN should be responsible for this. For more information on Edn Spec please visit: https://github.com/edn-format/edn.

Last Updated
9 days ago

Crate Size
25.9 kB

Authors

- Julia Naomi
- Otavio Pace

License

LGPL-3.0

```
enum Edn {
    Vector(Vector),
    Set(Set),
    Map(Map),
    List(List),
    Key(String),
    Symbol(String),
    Str(String),
    Int(isize),
    UInt(usize),
    Double(Double),
    Rational(String),
    Char(char),
    Bool(bool),
    Inst(String), // #inst "2020-10-12T09:30:00-00:00"
    Uuid(String), // #uuid "d3755af9-7fc5-44a2-8ca5-bb59452ebf42"
    NamespacedMap(String, Map), // :abc{0 5 1 "hello"}
    Nil,
    Empty,
}
```

Macro

```
let edn = edn!( (sym 1.2 3 false :f nil 3/4) );

assert_eq!(
    edn,
    Edn::List(List::new(
        vec![
            Edn::Symbol("sym".to_string()),
            Edn::Double(1.2.into()),
            Edn::Int(3),
            Edn::Bool(false),
            Edn::Key("f".to_string()),
            Edn::Nil,
            Edn::Rational("3/4".to_string()),
        ]
    ))
);
```

Navigation

```
let edn = edn!([sym 1.2 3 {false :f nil 3/4}]);


assert_eq!(edn[1], edn!(1.2));
assert_eq!(edn[1], Edn::Double(1.2f64.into()));
assert_eq!(edn[3]["false"], edn!(:f));
assert_eq!(edn[3]["false"],
Edn::Key(":f".to_string()));
```

Parsing

```
let edn_str = "{ :name \"joana\", :age 290000, }";

let edn: Edn = Edn::from_str(edn_str)?;

assert_eq!(
    edn,
    Edn::Map(Map::new(
        map!{
            ":name".to_string() =>
Edn::Str("joana".to_string()),
            ":age".to_string() => Edn::Int(290000),
        }
    ))
);
```

Emitting

```
let edn = Edn::Map(Map::new(
    map!{
        ":name".to_string() => Edn::Str("joana".to_string()),
        ":age".to_string() => Edn::Int(290000),
    }
));


assert_eq!(
    edn.to_string(), // edn_rs::to_string(edn)
    "{ :name \"joana\", :age 290000, }"
);
```

EDN to Struct

```
let edn = Edn::Map(Map::new(
    map!{
        ":name".to_string() => Edn::Str("joana".to_string()),
        ":age".to_string() => Edn::Int(290000),
    }
));
let person: Person = edn_rs::from_edn(edn)?;
assert_eq!(
    person,
    Person {
        name: "joana".to_string(),
        age: 290000,
    }
);
```

to_string vs to_debug

```
edn.to_string()

// { :name “Raja Kumari”, :age
77, }
```

```
edn.to_debug()

// Edn Body = Map(Map({
//     ":name": Str("Raja
Kumari"),
//     ":age": UInt(77),
// }))
```

edn-derive

edn-derive 0.3.1

Follow

Documentation Repository Dependent crates

Cargo.toml

```
edn-derive = "0.3.1"
```

edn-derive

Edn derive procedural macros for (De)Serialization.

This library still is `pre-alpha`.

Usage

Last Updated
11 days ago

Crate Size
5.39 kB

Authors

- Julia Boeira
- Otavio Pace

License

LGPL-3.0

Serialize

```
#[derive(Serialize)]
struct Person {
    name: String,
    age: usize,
}
let person = Person {
    name: ”joana”.to_string(),
    age: 290000,
}

assert_eq!(
    edn_rs::to_string(person),
    "{ :name \”joana\”, :age 290000, }"
);
```

Deserialize

```
#[derive(Deserialize, Debug, PartialEq)]
struct Person {
    name: String,
    age: usize,
}
let edn_person = "{ :name \"joana\", :age 290000, }";
let person: Person = edn_rs::from_str(edn_person)?;
assert_eq!(
    person,
    Person {
        name: "joana".to_string(),
        age: 290000,
    }
);
```

Serde inspiration

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

#[derive(Serialize,
Deserialize)]
struct Point {
  x: i32, y: i32,
}
let point = Point { x: 1, y: 2
};
let serialized =
```

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

#[derive(Serialize,
Deserialize)]
struct Point {
  x: i32, y: i32,
}
let point = Point { x: 1, y: 2
};
let serialized =
    edn_rs::to_string(point);
```

Using both

```
use edn_derive::{Deserialize as EdnDeserialize, Serialize as
EdnSerialize};
use serde_derive::{Deserialize as SerdeDeserialize, Serialize as
SerdeSerialize};


#[derive(EdnSerialize, EdnDeserialize, SerdeDeserialize,
SerdeSerialize)]
pub struct BrCode {
    ...
}
```

Naming Restrictions

```
{:crux.db/id "d3755af9-7fc5-44a2-8ca5-bb59452ebf42"
  :amount 50
  :operation-type :operation-type/deposit}
```

```
#[derive(Serialize,
Deserialize)]
struct Accont {
    crux__db___id: String
    amount: usize,
    operation_type:
```

```
#[derive(Serialize,
Deserialize)]
enum OperationType {
    Deposit,
    Withdraw,
    Transfer,
```

References

- https://github.com/naomijub/atm-crux
- https://github.com/naomijub/brcode
- https://github.com/otaviopace/smaug
- https://github.com/dtolnay/proc-macro-workshop
- https://opencrux.com/

Thank you