Datomic transactions as entities not just meta

In the previous post you saw how datomic data is made up of facts. In this post I will discuss how you add facts to Datomic using transactions.

Just like Clojure code is just data, datomic transactions are also just data. They consist of collections of facts. This is very useful as there is not much to learn. You don't have to learn any particular language just a data format.

I won't go in to detail about how Datomic works, but for running transactions there is a single transactor responsible for running the transactions. For local dev use this is done in memory in the same process. See Datomic's Architecture page for more.

The basics

Here is the example I gave in the last article of facts about me:

```
{:db/id 123 :person/name "Pelle Braendgaard" :location/city "Miami, FL" :co
ntact/phone "+1 305-555-5555"}
```

I add this to the database like this:

```
(datomic.api/transact conn
    [{:db/id #db/id[:db.part/user -1000001] :person/name "Pelle Braendgaard
" :location/city "Miami, FL" :contact/phone "+1 305-555-5555"}])
```

Note as I'm creating a new entity I don't have an entity id. I need to create a temporary entity id which is what `#db/id[:db.part/user -1000001]` does. Don't worry to much about it, there are various ways of doing it, but none are important right now. I spent too much time being confused by this in the beginning.

I can modify an entity by adding new facts

```
(datomic.api/transact conn
    [{:db/id 123123 :location/city "Santiago, Chile" :contact/phone "+56 99
99 9999"}])
```

I could also have done this by adding individual attributes:

```
(datomic.api/transact conn
    [[:db/add 123123 :location/city "Santiago, Chile"]
     [:db/add 123123 :contact/phone "+56 9999 9999"]])
```

You can retract a datom. This doesn't actually remove it, it just marks that fact as no longer valid.

```
(datomic.api/transact conn
    [[:db/retract 123123 :contact/phone]])
```

You can also retract a whole entity, which retracts all facts about an entity:

```
(datomic.api/transact conn
    [[:db/retractEntity 123123]])
```

You can still query past database values and retrieve it.

Creating multiple related entities

Since the transactions just consist of a vector of facts you can easily create multiple entities at once:

```
(datomic.api/transact conn
    [{:db/id #db/id[:db.part/user -1000001] :person/name "Pelle Braendgaard
" :location/city "Miami, FL" :contact/phone "+1 305-555-5555"}
     {:db/id #db/id[:db.part/user -1000002] :person/name "Bob Smith" :locat
ion/city "Coral Gables, FL" :contact/phone "+1 305-555-9999"}])
```

If I wanted to relate them to each other I can use a reference type. I won't go into details on schema yet. But see Datomic's Schema documentation for more.

I start out by creating a temporary id for each entity outside the transaction so I can use them to reference each other. You create this temporary id specifying the partition your data is in. While your just playing use `:db.part/user`. I'm still exploring the benefits of creating multiple partitions and will write that up later.

```
(let [ pelle (datomic.api/tempid :db.part/user)
       bob (datomic.api/tempid :db.part/user) ]
  (datomic.api/transact conn
    [{:db/id pelle :person/name "Pelle Braendgaard" :location/city "Miami,
FL" :contact/phone "+1 305-555-5555" :role/friends bob}
     {:db/id bob :person/name "Bob Smith" :location/city "Coral Gables, FL"
 :contact/phone "+1 305-555-9999" :role/friends pelle}]))
```

Database functions

Database functions are clojure or java functions you add to the schema of the database. I haven't touched on the schema yet but here is a very simple example of a function that increments an attribute:

```
{ :db/id #db/id [:db.part/user]
    :db/ident :inc
    :db/fn #db/fn { :lang "clojure"
                    :params [db id attr amount]
                    :code "(let [ e (datomic.api/entity db id)
                                  orig (attr e 0) ]
                             [[:db/add id attr (+ orig amount) ]])"}}
```

The clojure function it installs is basically:

```
(defn inc [db id attr amount]
  (let [ e (datomic.api/entity db id)
         orig (attr e 0) ]
       [[:db/add id attr (+ orig amount) ]]))
```

It is based the value of the database as it is at the moment of the transaction and then any other parameters you want to pass it. It should return a vector of data elements just like you would when creating a transaction manually. They can also call other functions.

This is can be called by adding it to your transactional data like this:

```
(datomic.api/transact conn
    [[:inc 123123 :person/age 1]])
```

Database functions are especially needed when you want to create facts based on existing facts. For example increasing a value in the database.

You could also use it for validation. In this case you throw an exception in your function if something is invalid. The whole transaction will rollback.

It may also be a neat way of abstracting out the creation of common elements within a transaction.

Transactions as entities

Each transaction has an entity created for it. This by default just has a :db/txInstant timestamp attribute value. But you can add as much information about your transaction that you wish.

This can be useful for auditing by adding user ids, ip addresses etc. But taking it to the extreme lets look at this simple bank application where you transfer funds from one account to the other.

```
(let [txid (datomic.api/tempid :db.part/tx)]
    (datomic.api/transact conn [[:transfer from to amount]
                                {:db/id txid, :db/doc note :ot/from from :ot/to to :o
t/amount amount}]))
```

The `[:transfer ...]` section is a database function performing a transfer between accounts.

The important thing here is that we create a new tempid in the `:db.part/tx partition`. We can add as many facts as we want to this id which will represent the transaction.

This can be queried as if it was just a regular entity:

```
(datomic.api/q '[:find ?tx :in $ :where [?tx :ot/amount _]] (datomic.api/db
 conn))
```

The above returns any entity with a `:ot/amount` attribute.

This transaction anotation is extremely powerful. Instead of creating your own activity tables you can just annotate the transaction instead.

Complete example

I've written a complete example showing most uses of Datomic transactions in a bank like application that you can look at.

The schema contains 2 database functions `:transfer` and `:credit`. `:transfer` calls `:credit` on two accounts with oposite amounts. `:credit` throws an exception to ensure sufficient funds in an account.

The transfer function calls the above transfer function and annotates the transaction with

About me

I am Pelle Braendgaard. Pronounce it like Pelé the footballer (no relation). I live in wonderful Miami Beach. I like clojure, ruby, building stuff, travel, food, wine, rum and trying to help build a new financial system from the grass roots up.

Consulting

I am available to help you out with your project for shorter projects involving Clojure, Datomic, Ruby, OAuth, payments and/or Identity. Please contact me at pelle@stakeventures.com

I normally work remote but may be available for a couple of days onsite in the US, Europe and Latin America if you are prepared to pay expenses. My base rate is $155/h with discounts available for ongoing gigs and for clojure as I'm relatively new in the field.

Published

09 July 2012

Tags

datomic [2]

transactions [1]

More about me

Bio and contact info
Selected clients
Open Source projects
My Agree2 Profile
My personal time currency
My GitHub Account
My Linked In profile
Paylio.be My blog on payments
My Flickr stream

Current projects and startups

PicoMoney
Agree2
TimeCert
OpenTransact
OAuth

information about the transaction.

← Previous | Archive | Next →