Simulation Testing with Simulant

@stuarthalloway

Example-Based Tests (EBT)

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

EBT

setup

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```


EBT

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

inputs

EBT

execution

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```


EBT

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

output

EBT

```ruby
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

validation

EBT

```clojure
(are [x y] (= x y)
    (+) 0
    (+ 1) 1
    (+ 1 2) 3
    (+ 1 2 3) 6

    (+ -1) -1
    (+ -1 -2) -3
    (+ -1 +2 -3) -2

    (+ 2/3) 2/3
    (+ 2/3 1) 5/3
    (+ 2/3 1/3) 1 )
```

EBT

```
(are [x y] (= x y)
    (+) 0
    (+ 1) 1
    (+ 1 2) 3
    (+ 1 2 3) 6

    (+ -1) -1
    (+ -1 -2) -3
    (+ -1 +2 -3) -2

    (+ 2/3) 2/3
    (+ 2/3 1) 5/3
    (+ 2/3 1/3) 1 )
```

no setup

EBT

```
(are [x y] (= x y)
    (+)           0
    (+ 1)         1
    (+ 1 2)       3
    (+ 1 2 3)     6

    (+ -1)        -1
    (+ -1 -2)     -3
    (+ -1 +2 -3)  -2

    (+ 2/3)       2/3
    (+ 2/3 1)     5/3
    (+ 2/3 1/3)   1 )
```

inputs

EBT

```
(are [x y] (= x y)
    (+)            0
    (+ 1)          1
    (+ 1 2)        3
    (+ 1 2 3)      6

    (+ -1)         -1
    (+ -1 -2)      -3
    (+ -1 +2 -3)   -2

    (+ 2/3)        2/3
    (+ 2/3 1)      5/3
    (+ 2/3 1/3)    1 )
```

execution

EBT

```
(are [x y] (= x y)
    (+)            0
    (+ 1)          1
    (+ 1 2)        3
    (+ 1 2 3)      6

    (+ -1)         -1
    (+ -1 -2)      -3
    (+ -1 +2 -3)   -2

    (+ 2/3)        2/3
    (+ 2/3 1)      5/3
    (+ 2/3 1/3)    1 )
```

outputs

EBT

```
(are [x y] (= x y)
    (+)          0
    (+ 1)        1
    (+ 1 2)      3
    (+ 1 2 3)    6

    (+ -1)       -1
    (+ -1 -2)    -3
    (+ -1 +2 -3) -2

    (+ 2/3)      2/3
    (+ 2/3 1)    5/3
    (+ 2/3 1/3)  1 )
```

validation

EBT in the Wild

Scales: Unit, Functional, Acceptance

Styles: Test-After, TDD, BDD

Common Idioms: Fixtures, Stubs, Mocks

Deconstructing EBT

Inputs

Execution

Outputs

Validation

Simulation

Model

Outputs

Execution

Inputs

Validation

Simulation

Model

Outputs

kinds of agents
interactions as distributions
stored in a database

Execution

Inputs

Validation

Simulation

Model

Outputs

agents
time-stamped action stream
stored in a database

Inputs

Validation

Simulation

Model

Outputs

Execution

driver program
coordinated through a database
maps agents to processes

Inputs

lidation

Simulation

Model

Outputs

system storage
logs
metrics
... put in all in a database!

Inputs

Validation

Simulation

Model

Outputs

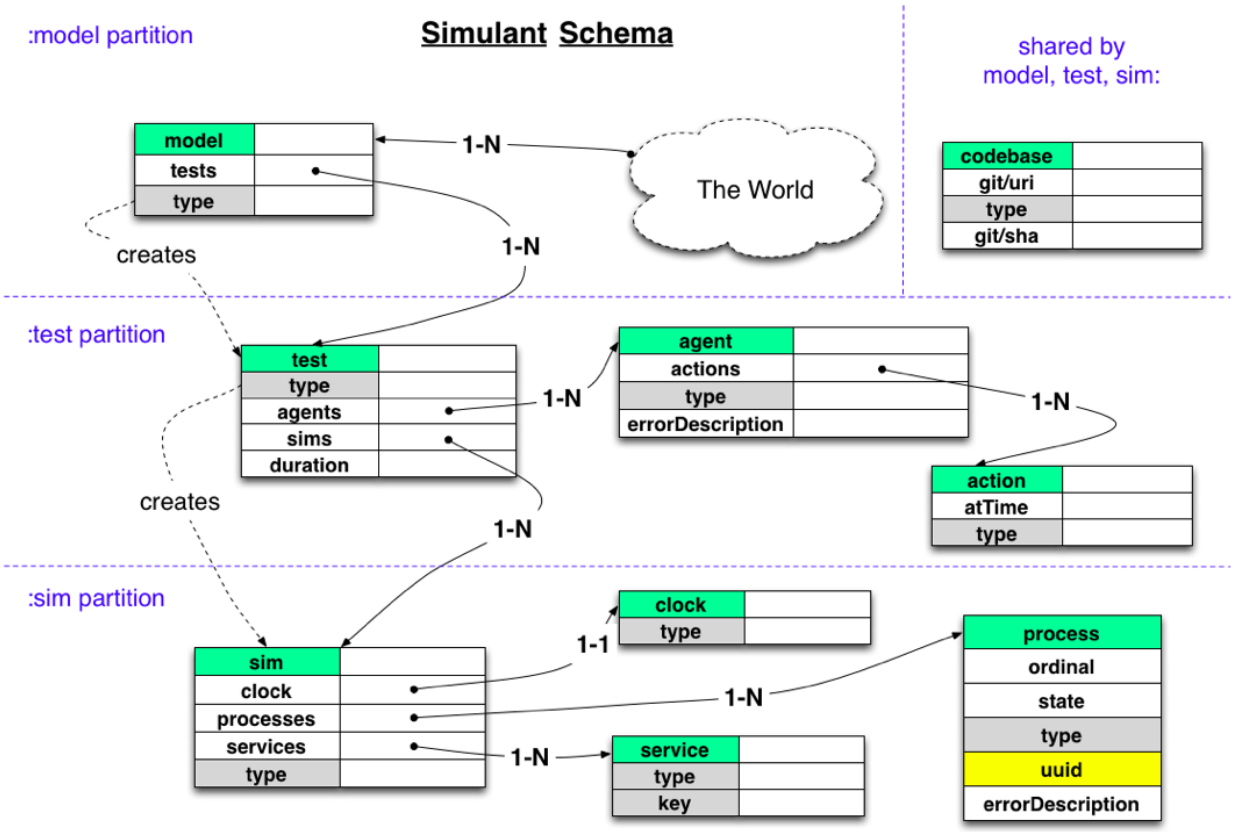
database queries
may be probabilistic

Inputs

Validation

Simulant

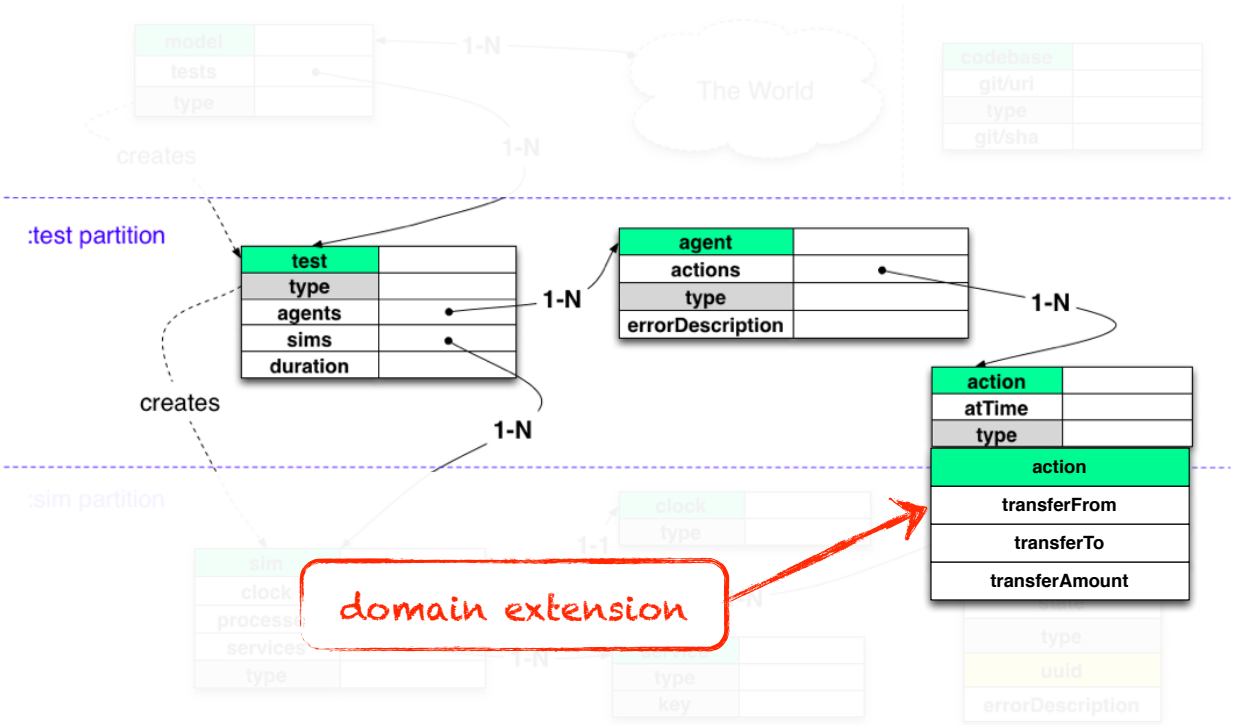
Datomic


:model partition
Simulant Schema
shared by
model, test, sim:
model
tests
type
1-N
The World
codebase
git/uri
type
git/sha
creates
1-N
:test partition
test
type
agents
sims
duration
1-N
agent
actions
type
errorDescription
1-N
action
atTime
type
creates
1-N
:sim partition
clock
type
1-1
sim
clock
processes
services
type
1-N
process
ordinal
state
type
uuid
errorDescription
1-N
service
type
key

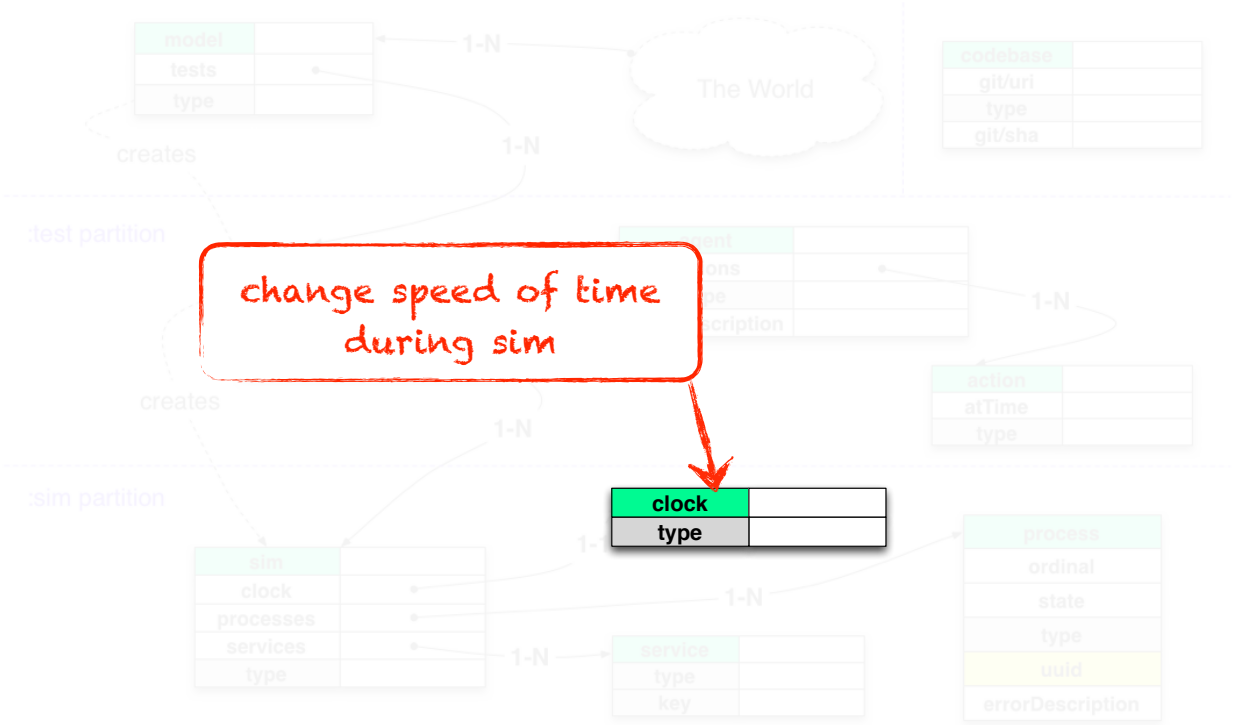
Models

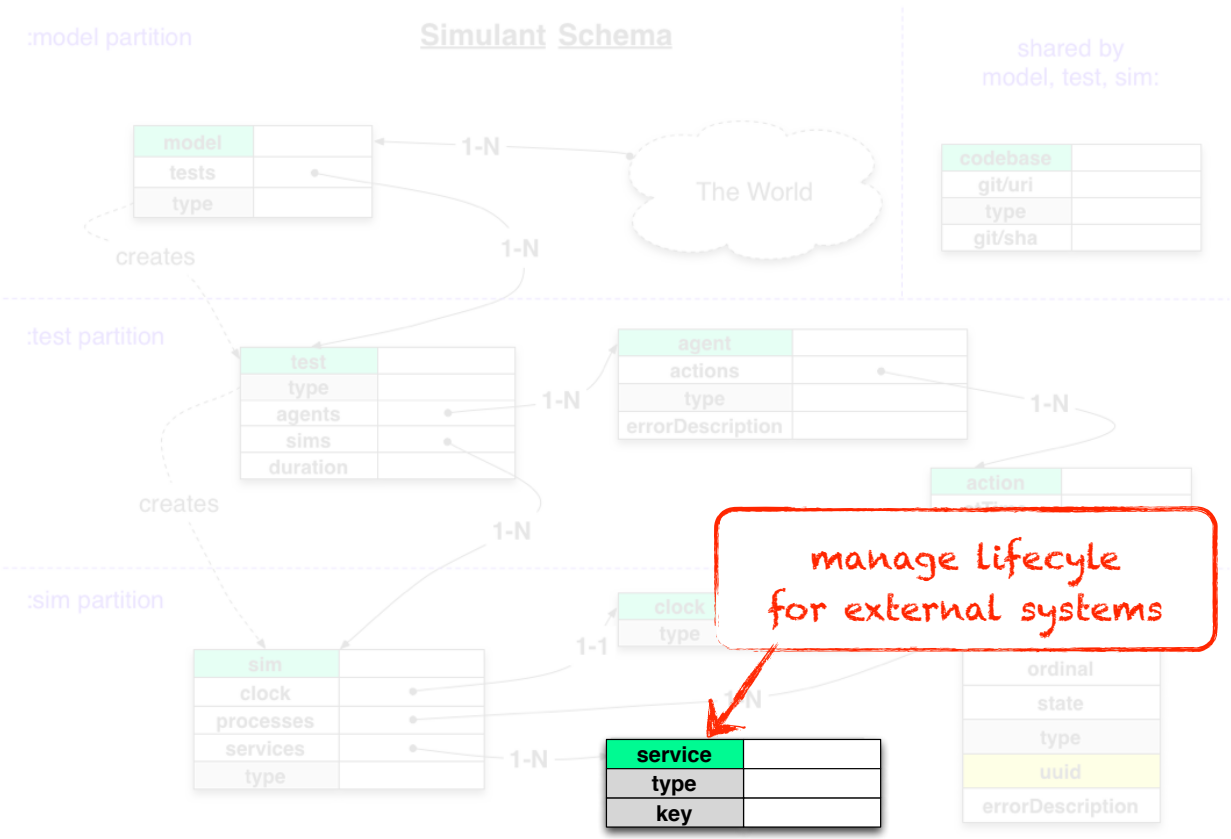

model
tests
type
model
traderCount
initialBalance
meanTradeAmount
meanHoursBetweenTrades
few generic attributes
extended by domain

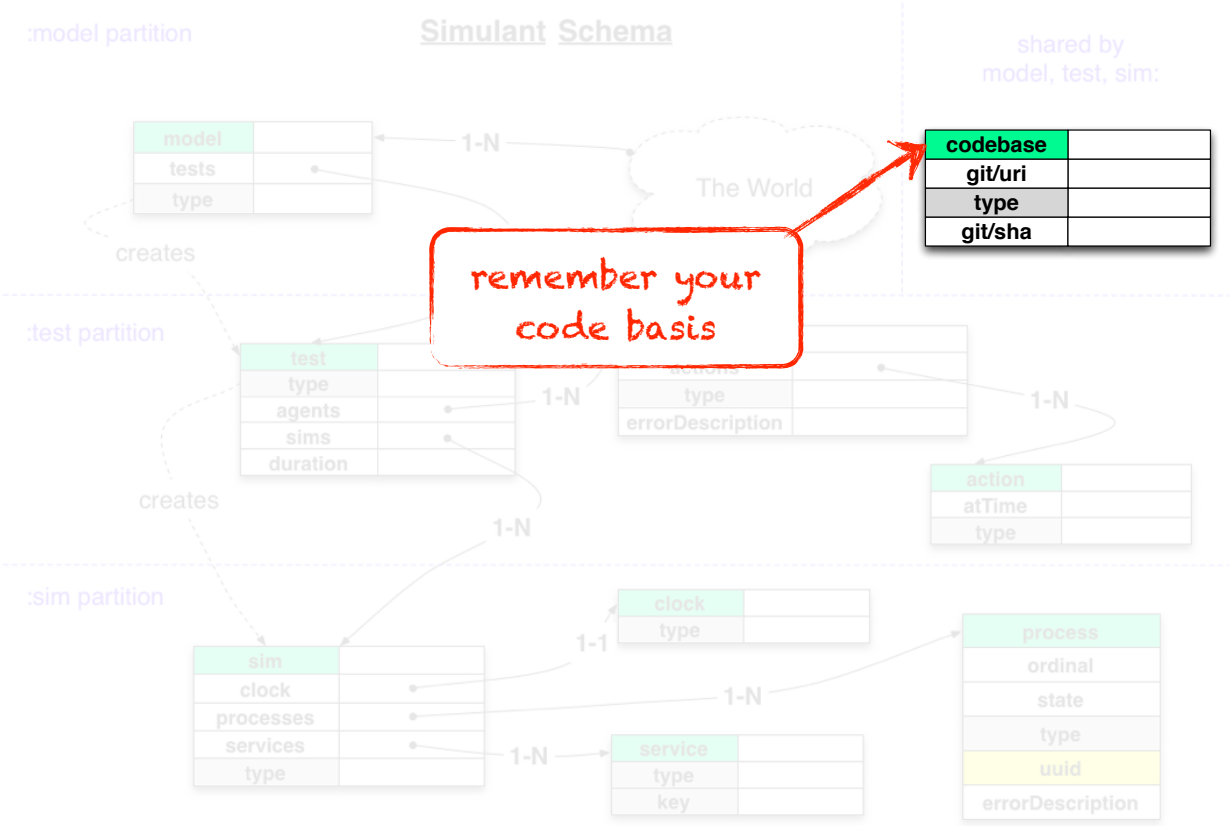
Activity



Clock




:model partition
Simulant Schema
shared by
model, test, sim:
model
tests
type
1-N
The World
codebase
git/uri
type
git/sha
creates
1-N
:test partition
test
type
agents
sims
duration
1-N
agent
actions
type
errorDescription
1-N
action
creates
1-N
manage lifecyle
for external systems
:sim partition
clock
type
1-1
sim
clock
processes
services
type
1-N
1-N
service
type
key
ordinal
state
type
uuid
errorDescription


:model partition
Simulant Schema
shared by
model, test, sim:
model
tests
type
1-N
The World
codebase
git/uri
type
git/sha
creates
remember your
code basis
:test partition
test
type
agents
sims
duration
1-N
type
errorDescription
1-N
action
atTime
type
creates
1-N
:sim partition
clock
type
1-1
sim
clock
processes
services
type
1-N
1-N
service
type
key
process
ordinal
state
type
uuid
errorDescription

Demo

data.generators

Objectives

Generate all kinds of data

Various distributions

Predictable

Approach

Generator fns shadow related fns in clojure.core

Default integer distributions are uniform on range

Other defaults are arbitrary

Repeatable via dynamic binding of *rnd*

Scalar Generators

```
(require '[clojure.data.generators :as gen])

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

(gen/rand-nth [:a :b :c])
=> :a
```


Scalar Generators

```
(require '[clojure.data.generators :as gen])

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

(gen/rand-nth [:a :b :c])
=> :a
```

idiomatic ns prefix

Scalar Generators

```
(require '[clojure.data.generators :as gen])

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

(gen/rand-nth [:a :b :c])
=> :a
```

value from platform range

Scalar Generators

```
(require '[clojure.data.generators :as gen])

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

(gen/rand-nth [:a :b :c])
=> :a
```

explicit distribution

Scalar Generators

```
(require '[clojure.data.generators :as gen])

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

(gen/rand-nth [:a :b :c])
=> :a
```

predictable seed
for c.c. methods

Collection Generators

```
(gen/list gen/short)
=> (-8600 -14697 -2382 18540 27481)


(gen/hash-map gen/short gen/string 2)
=> {-7110 "UBL)l",
    11472 "Q5|>^>rQNL9E..y#}IMpw>gnM']jD'<q"}
```

Collection Generators

default size
fairly small

```
(gen/list gen/short)
=> (-8600 -14697 -2382 18540 27481)

(gen/hash-map gen/short gen/string 2)
=> {-7110 "UBL)l",
    11472 "Q5|>^>rQNL9E..y#}IMpw>gnM']jD'<q"}
```


Collection Generators

```
(gen/list gen/short)
=> (-8600 -14697 -2382 18540 27481)

(gen/hash-map gen/short gen/string 2)
=> {-7110 "UBL)l",
    11472 "Q5|>^>rQNL9E..y#}IMpw>gnM']jD'<q"}
```

explicit size
(# or fn)

Composition

```
(gen/one-of gen/long gen/keyword)
=> :OBe0Mkc1g7eqqQnGvcXq0m-McRzl9areH0NwR1

(gen/weighted {gen/long 10 gen/keyword 1})
=> 471803172735646609

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -4909497585342792620
     ...}
```


Composition

```
(gen/one-of gen/long gen/keyword)
=> :OBe0Mkc1g7eqqQnGvcXq0m-McRzl9areH0NwR1

(gen/weighted {gen/long 10 gen/keyword 1})
=> 471803172735646609

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -4909497585342792620
     ...}
```

choose
(equal weights)

Composition

```
(gen/one-of gen/long gen/keyword)
=> :OBe0Mkc1g7eqqQnGvcXq0m-McRzl9areH0NwR1

(gen/weighted {gen/long 10 gen/keyword 1})
=> 471803172735646609

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -4909497585342792620
     ...}
```

explicit weights

Composition

```
(gen/one-of gen/long gen/keyword)
=> :OBe0Mkc1g7eqqQnGvcXq0m-McRzl9areH0NwR1

(gen/weighted {gen/long 10 gen/keyword 1})
=> 471803172735646609

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -4909497585342792620
     ...}
```

any scalar

Composition

```
(gen/one-of gen/long gen/keyword)
=> :OBe0Mkc1g7eqqQnGvcXq0m-McRzl9areH0NwR1

(gen/weighted {gen/long 10 gen/keyword 1})
=> 471803172735646609

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -4909497585342792620
     ...}
```

any collection
(of scalars)

Datalog

Query Anatomy

```
q([:find ...
   :in ...
   :where ...],
   input1,
   ...
   inputN);
```


Query Anatomy

constraints

```
q([:find ...
   :in ...
   :where ...],
   input1,
   ...
   inputN);
```

Query Anatomy

```
q([:find ...
   :in ...
   :where ...],
   input1,
   ...
   inputN);
```

inputs

Query Anatomy

```
q([:find ...
   :in ...
   :where ...],
   input1,
   ...
   inputN);
```

names for inputs

Query Anatomy

```
q([:find ...
   :in ...
   :where ...],
   input1,
   ...
   inputN);
```

variables to return

Variables

```
?customer
                ?product

        ?orderId

                ?email
```

Constants

```
:email
42
"john"
:order/id
#inst "2012-02-29"
```


Extensible Reader

```
:email
42
"john"
:order/id
#inst "2012-02-29"
```

Example Database

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

Data Pattern

Constrains the results returned,
binds variables

```
[?customer :email ?email]
```

Data Pattern

Constrains the results returned, binds variables

```
[?customer :email ?email]
```

entity attribute value

Data Pattern

Constrains the results returned, binds variables

constant

```
[?customer :email ?email]
```

Data Pattern

Constrains the results returned,
binds variables

variable ↓ variable ↓

```
[?customer :email ?email]
```


entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[?customer :email ?email]
```

Constants Anywhere

"Find a particular customer's email"

```
[42 :email ?email]
```


entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 :email ?email]
```

Variables Anywhere

"What attributes does
customer 42 have?

```
[42 ?attribute]
```


entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 ?attribute]
```

Variables Anywhere

"What attributes and values does customer 42 have?

```
[42 ?attribute ?value]
```

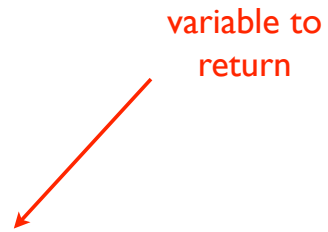

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 ?attribute ?value]
```

Where Clause


```
[:find ?customer
 :where [?customer :email]]
```


Find Clause




```
[:find ?customer
 :where [?customer :email]]
```

Implicit Join

"Find all the customers who have placed orders."

```
[:find ?customer
 :where [?customer :email]
        [?customer :orders]]
```


API

```
import static datomic.Peer.q;

q("[:find ?customer
    :where [?customer :id]
           [?customer :orders]]",
   db);
```

q

```
import static datomic.Peer.q;

q("[:find ?customer
    :where [?customer :id]
           [?customer :orders]]",
   db);
```


Query

```
import static datomic.Peer.q;

q("[:find ?customer
    :where [?customer :id]
           [?customer :orders]]",
   db);
```

Input(s)

```
import static datomic.Peer.q;


q("[:find ?customer
    :where [?customer :id]
           [?customer :orders]]",
   db);
```


In Clause

Names inputs so you can refer to them elsewhere in the query

```
:in $database ?email
```

Parameterized Query

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
   db,
   "jdoe@example.com");
```


First Input

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
   db,
   "jdoe@example.com");
```

Second Input

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
   db,
   "jdoe@example.com");
```


Verbose?

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
   db,
   "jdoe@example.com");
```

Shortest Name Possible

"Find a customer by email."

```
q([:find ?customer
   :in $ ?email
   :where [$ ?customer :email ?email]],
   db,
   "jdoe@example.com");
```


Elide $ in Where

"Find a customer by email."

```
q([:find ?customer
   :in $ ?email
   :where [ ?customer :email ?email]],
   db,
   "jdoe@example.com");
```

no need to specify $

Predicates

Functional constraints that can appear in a :where clause

```
[(< 50 ?price)]
```


Adding a Predicate

"Find the expensive items"

```
[:find ?item
 :where [?item :item/price ?price]
        [(< 50 ?price)]]
```

Functions

Take bound variables as inputs
and bind variables with output

```
[(shipping ?zip ?weight) ?cost]
```


Function Args

```
[(shipping ?zip ?weight) ?cost]
```

bound inputs

Function Returns

```
[(shipping ?zip ?weight) ?cost]
```

bind return values

Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

navigate from customer to zip

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```


Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

get product facts needed *during query*

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

call web service to bind shipCost

Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

constrain price

Calling a Function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

return customer, product pairs

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```


Clojure Wins

700 LOC

Multimethods

Seqs

Laziness

Agents

Datomic Wins

Open schema

Datalog

Time model

Functional

Multi-db queries

Adopting Simulation

Test any target system

Don’t throw our your example-based tests

Comfort with the model comes in ~1 week

Simulation requires time and thought

References

The Simulant open-source library,
https://github.com/datomic/simulant

Simulant Demo,
https://github.com/Datomic/simulant/blob/master/examples/repl/hello_world.clj

Datomic,
http://www.datomic.com/

Clojure,
http://clojure.org/

Relevance,
http://thinkrelevance.com/

Presentations by Stuart Halloway,
https://github.com/stuarthalloway/presentations

@stuarthalloway

https://github.com/stuarthalloway/presentations/wiki