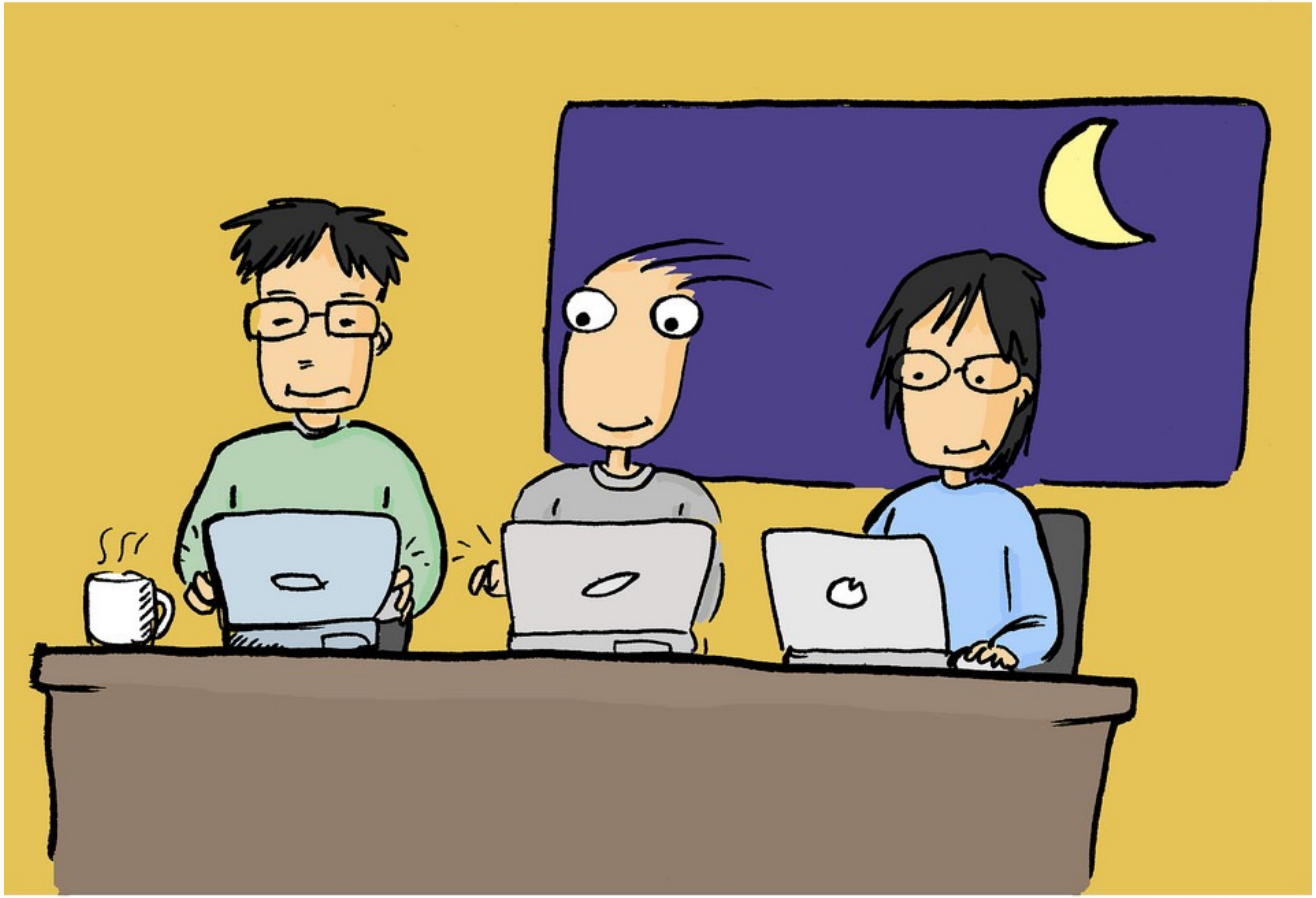
Clients in control

building demand-driven systems with Om Next

Craft Conf 2016

@anmonteiro90



REST: expectations

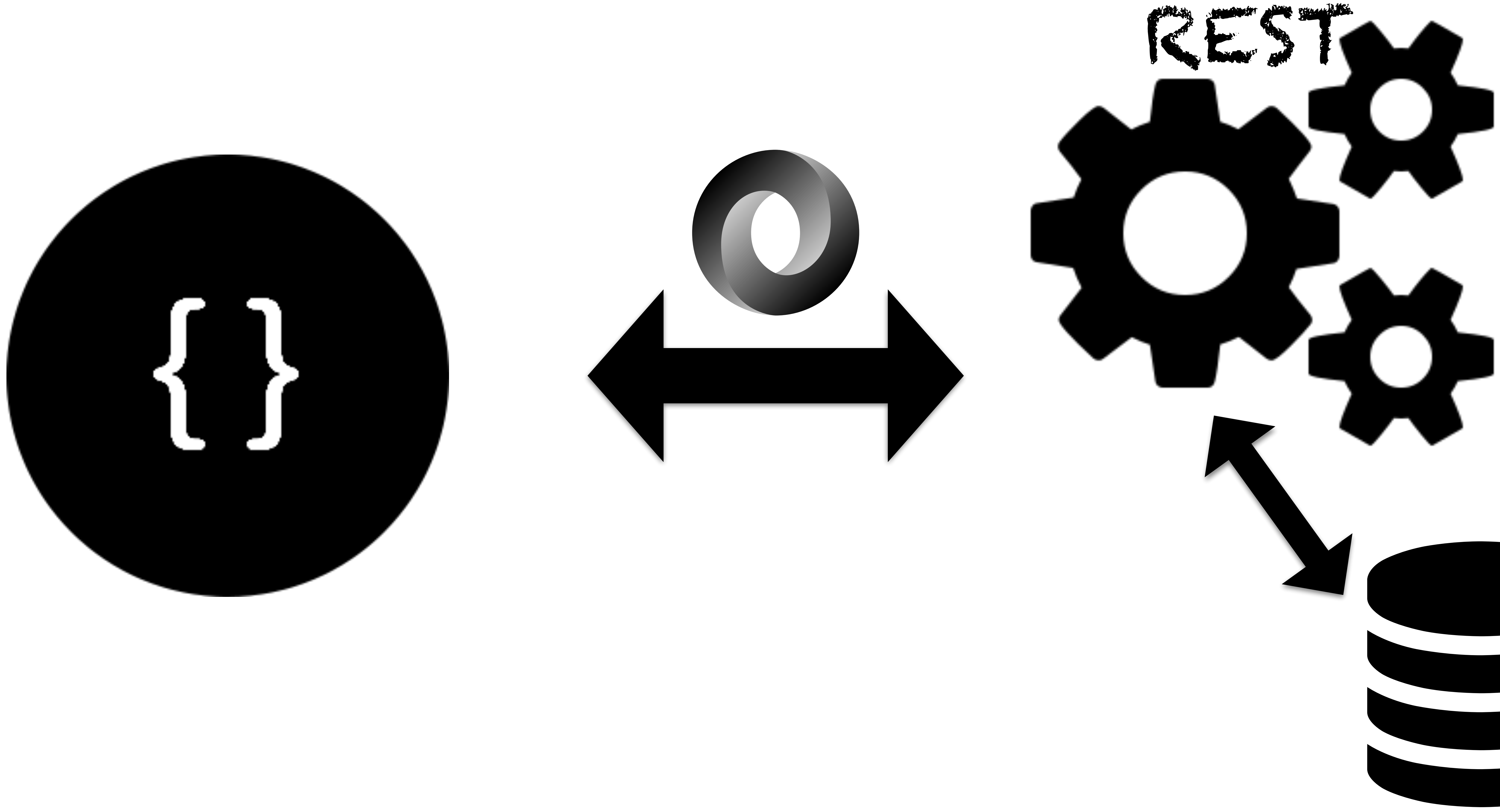
- define logical “resources”
  - identified by URIs
- clients request them

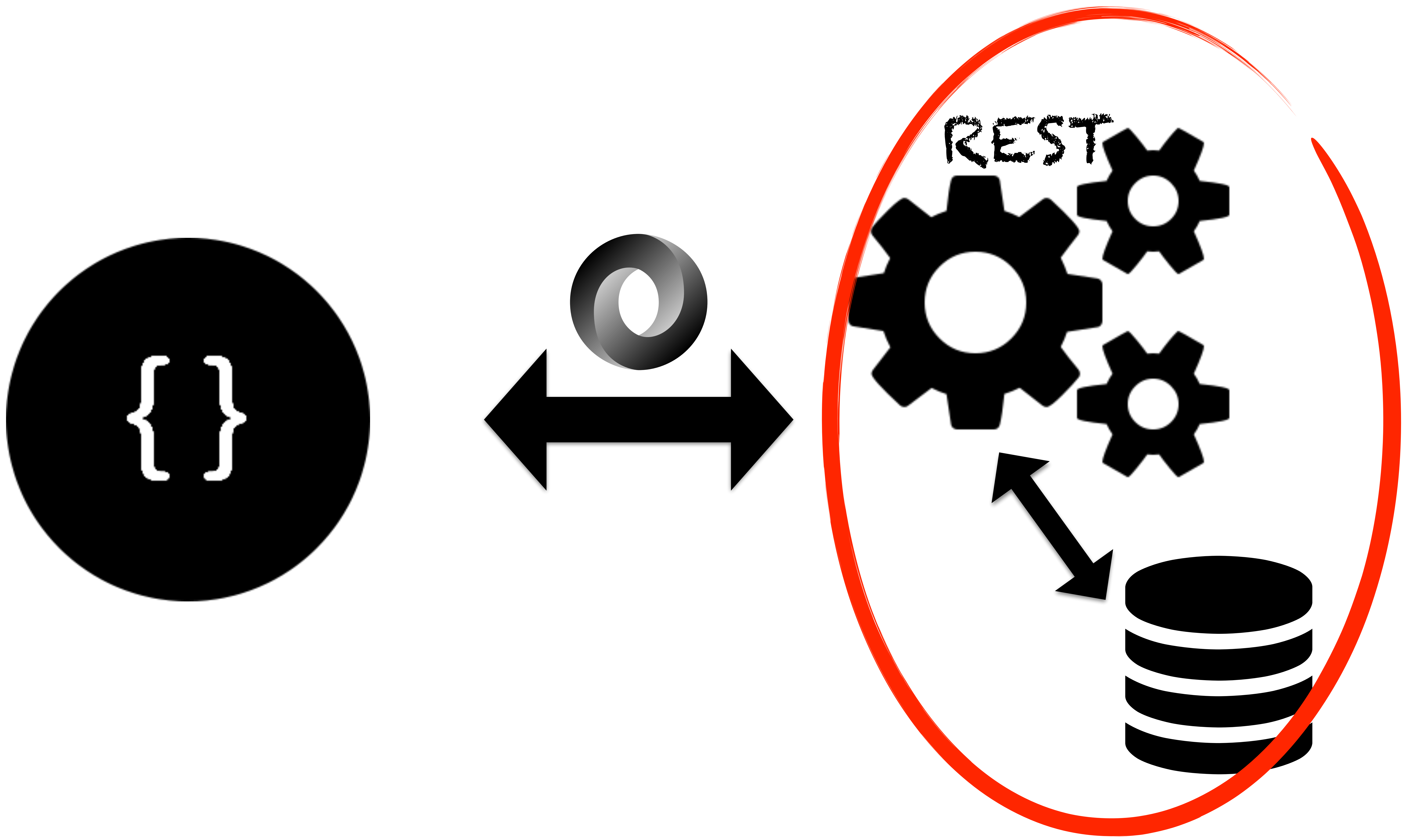
REST: reality

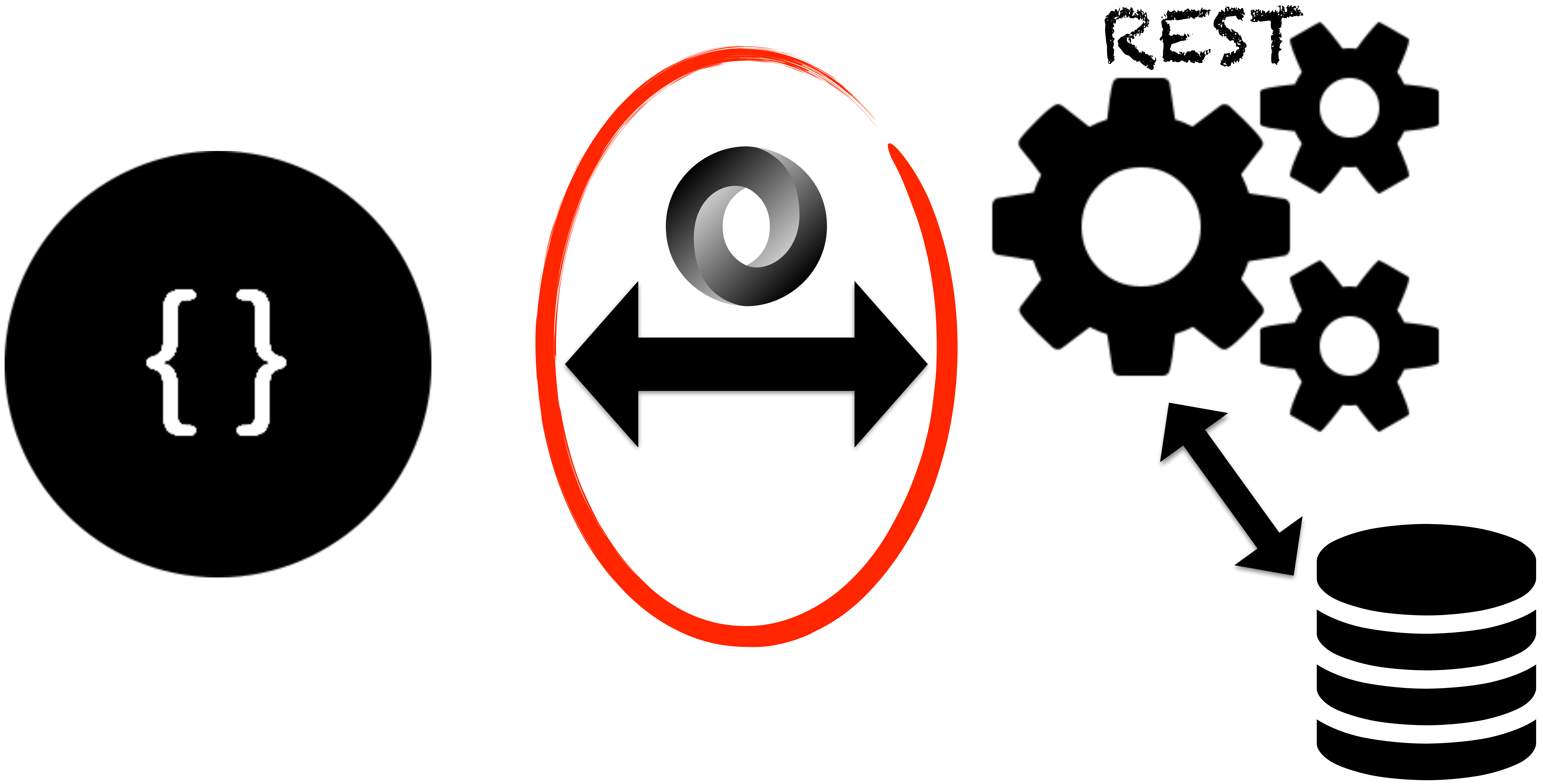
- only able to request trivial data
- “joined” resources
  - bloat endpoint?
  - multiple requests?

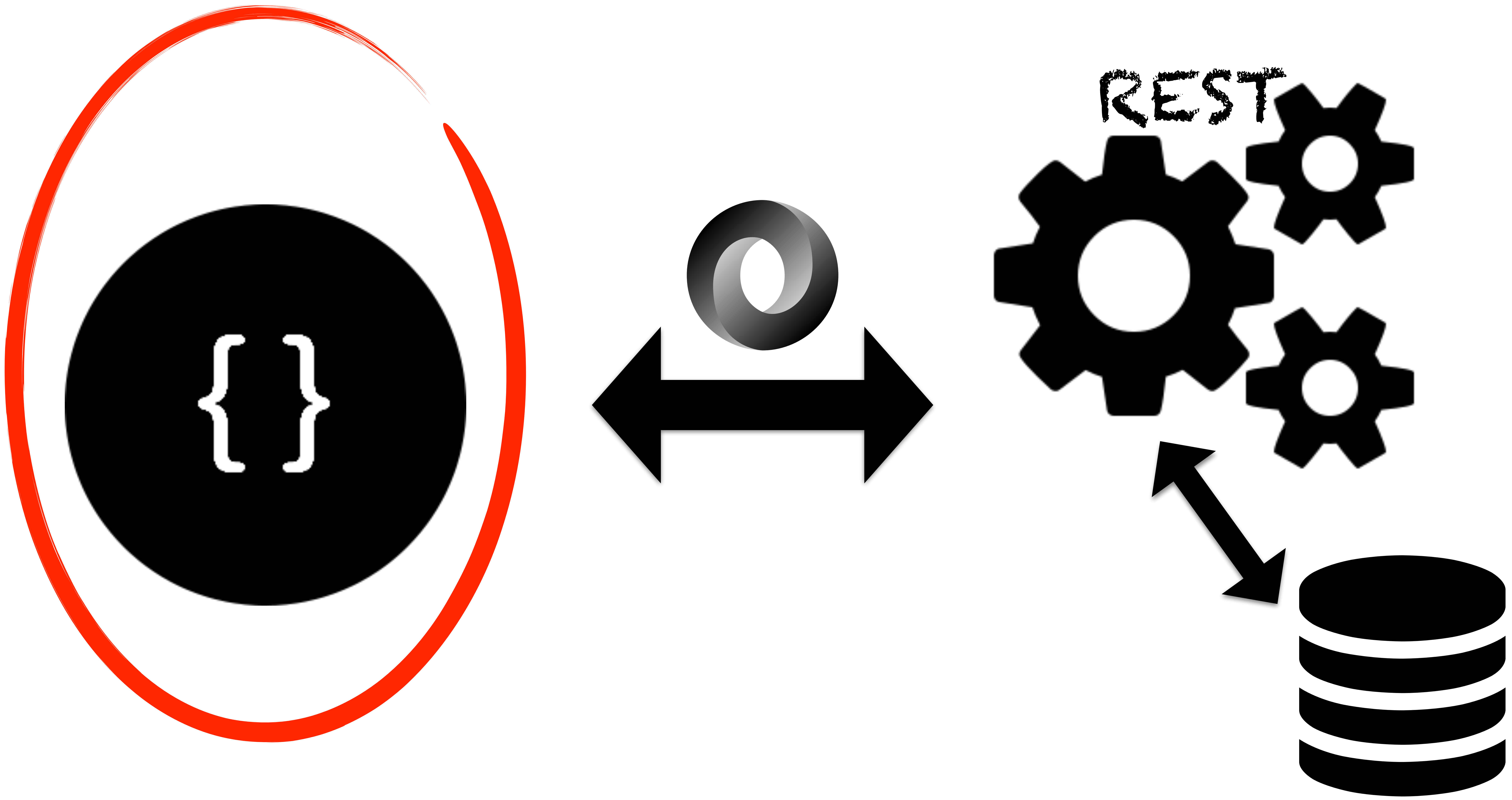
“The REST interface is designed to be efficient for **large-grain hypermedia data transfer**, […] resulting in an interface that is not optimal for other forms of architectural interaction.”

– Roy T. Fielding, PhD


{ }
REST


REST


REST


{ }
REST

“The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a **standardized form** rather than one which is **specific to an application's needs**.”

– Roy T. Fielding, PhD

Luke Wroblewski

How to write a service that meets the varying demands of heterogeneous clients?

METEOR

Parse

METEOR
Par e

METEOR

Par e

Let’s keep looking…

Desirable properties

- clients can request the **exact total** response they need
- clients can communicate novelty atomically
  - without sacrificing relational queries on the server

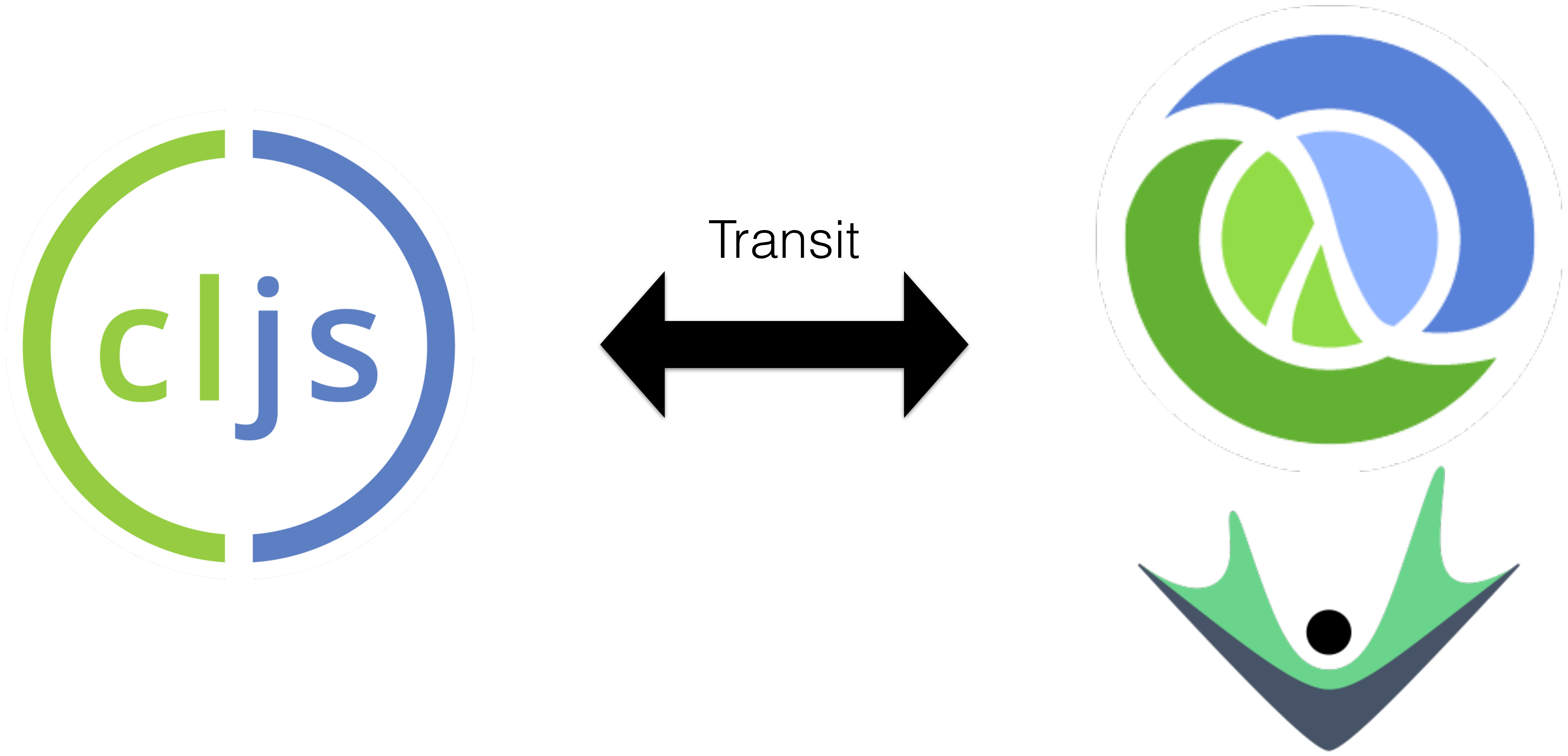
This is the story of how

NETFLIX

eliminated 90% of the networking code in our app.

Checkpoint

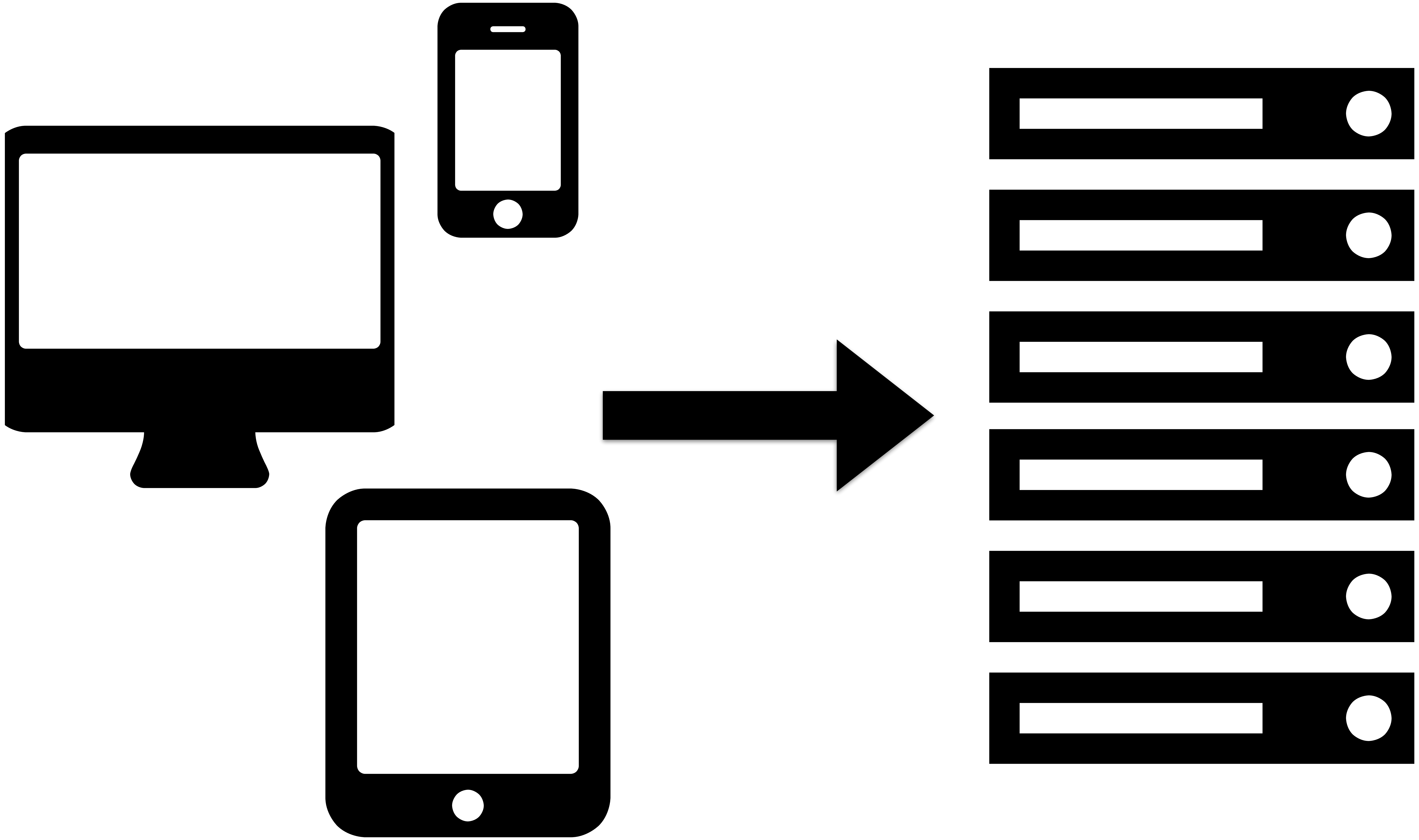
- How to make precise requests?
  - Client
  - Server
- How do clients communicate novelty?
  - Communicate identity back to client?
- Communication over the wire?
- Client-only state
- Testing
- Caching
- Pluggable client / server storage?


cljs
Transit

Enter Om Next…

Om Next *opinions*

- Single source of truth
- Minimize flushing to DOM
- Abstract asynchrony
- No (visible) event model



```
[:person/name]
```

```
(defui Person
  static om/IQuery
  (query [this]
    [:person/name])

  Object
  (render [this]
    ...))
```

Query expressions

```
:person/name

(:person/friends {:sort :asc})

{:person/address
 [:address/street :address/zip]}
```

Query expressions

```
(increment/users!)

(delete/friend! {:me 1 :friend 2})
```

Parser

- Evaluates query expressions
- Hydrates queries
  - no reshaping!

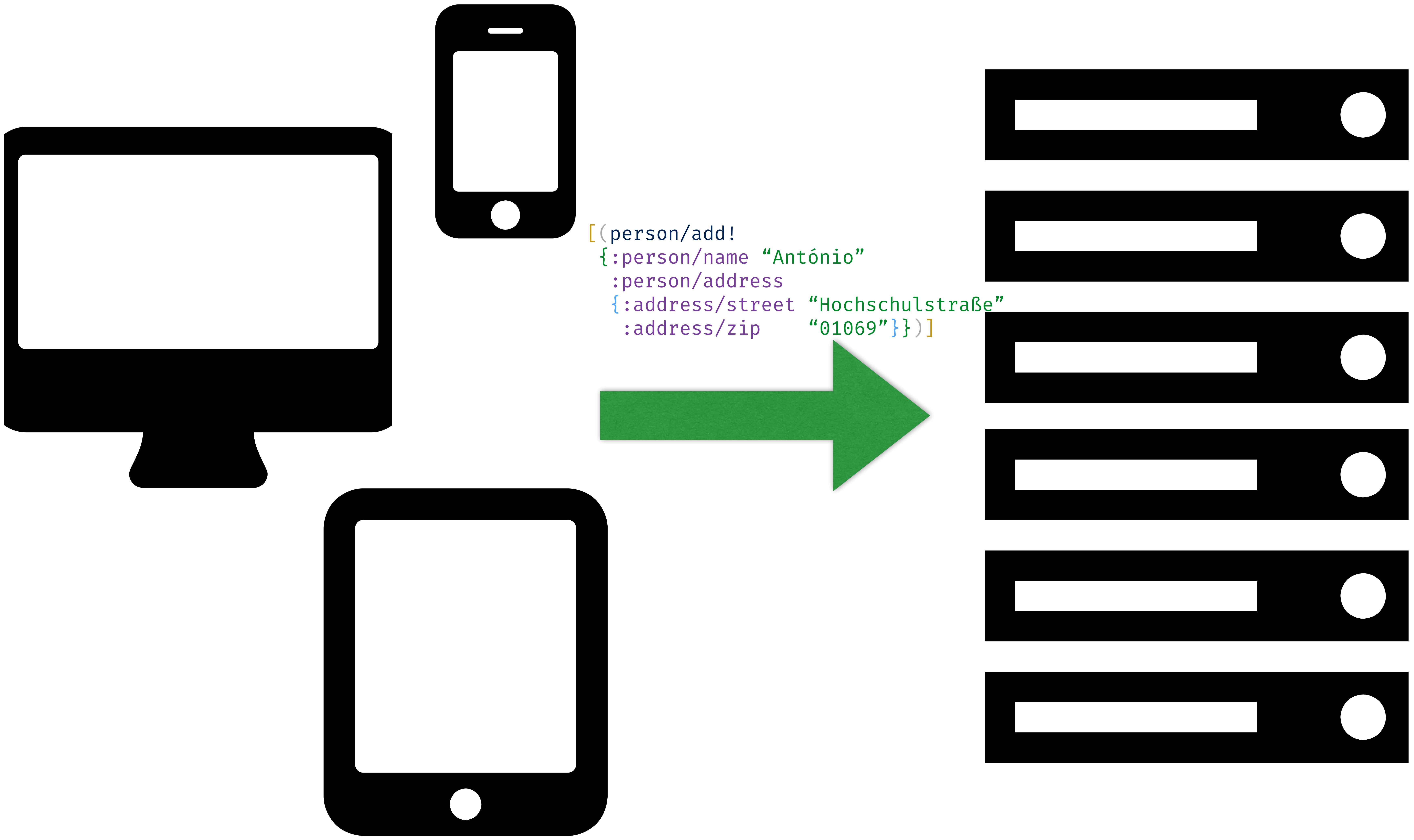
Parser

```
[:person/name]
      ↓
{:person/name "António"}
```

Parser

- Runs on the client and server
- Runs **reads** and **mutations**

Demo



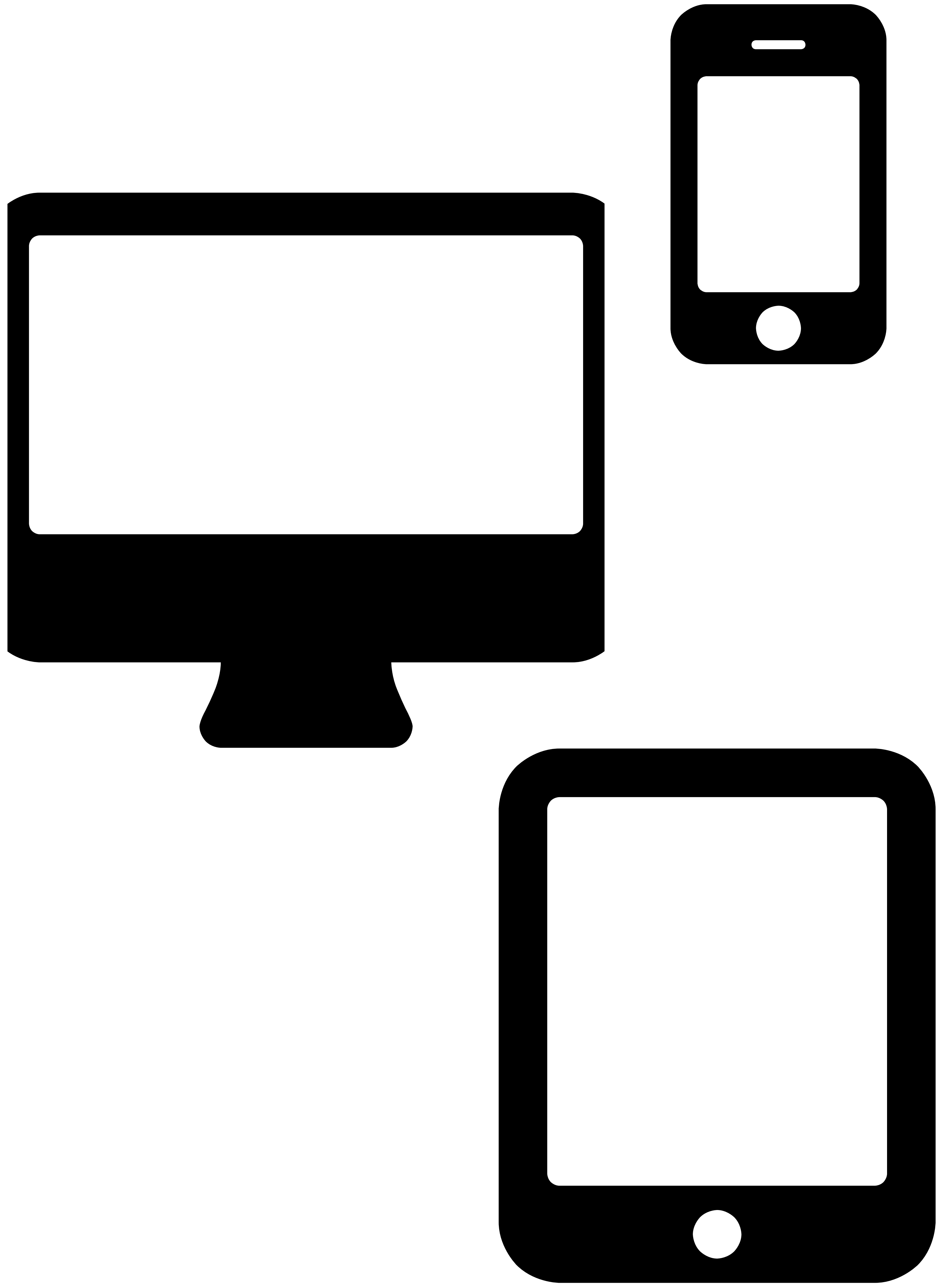
```
[(person/add!
 {:person/name “António”
  :person/address
  {:address/street “Hochschulstraße”
   :address/zip    “01069”}})]
```

```
[(delete/friend! {:me 1
                  :friend 2})
 :friends/list]
```

Re-read this key

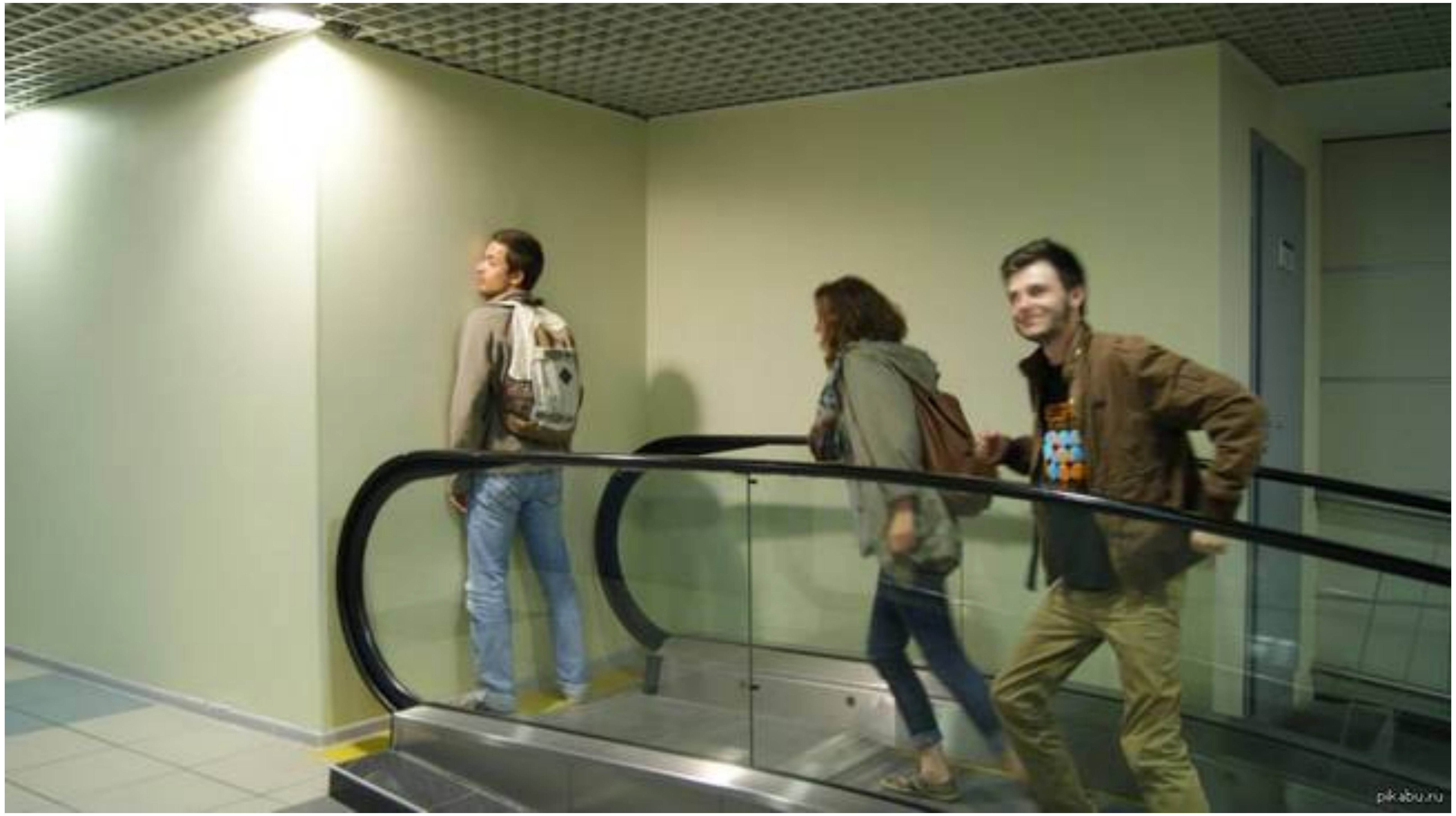
Creating information

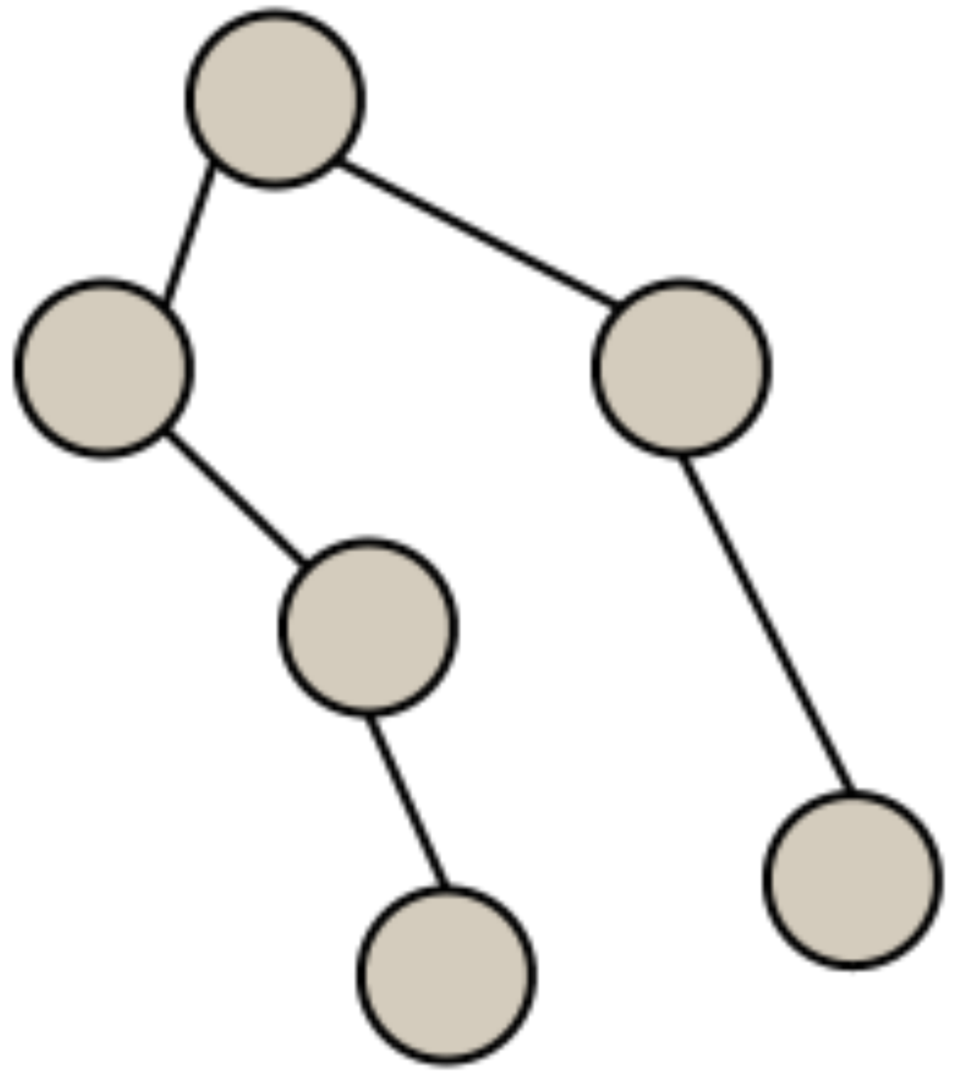
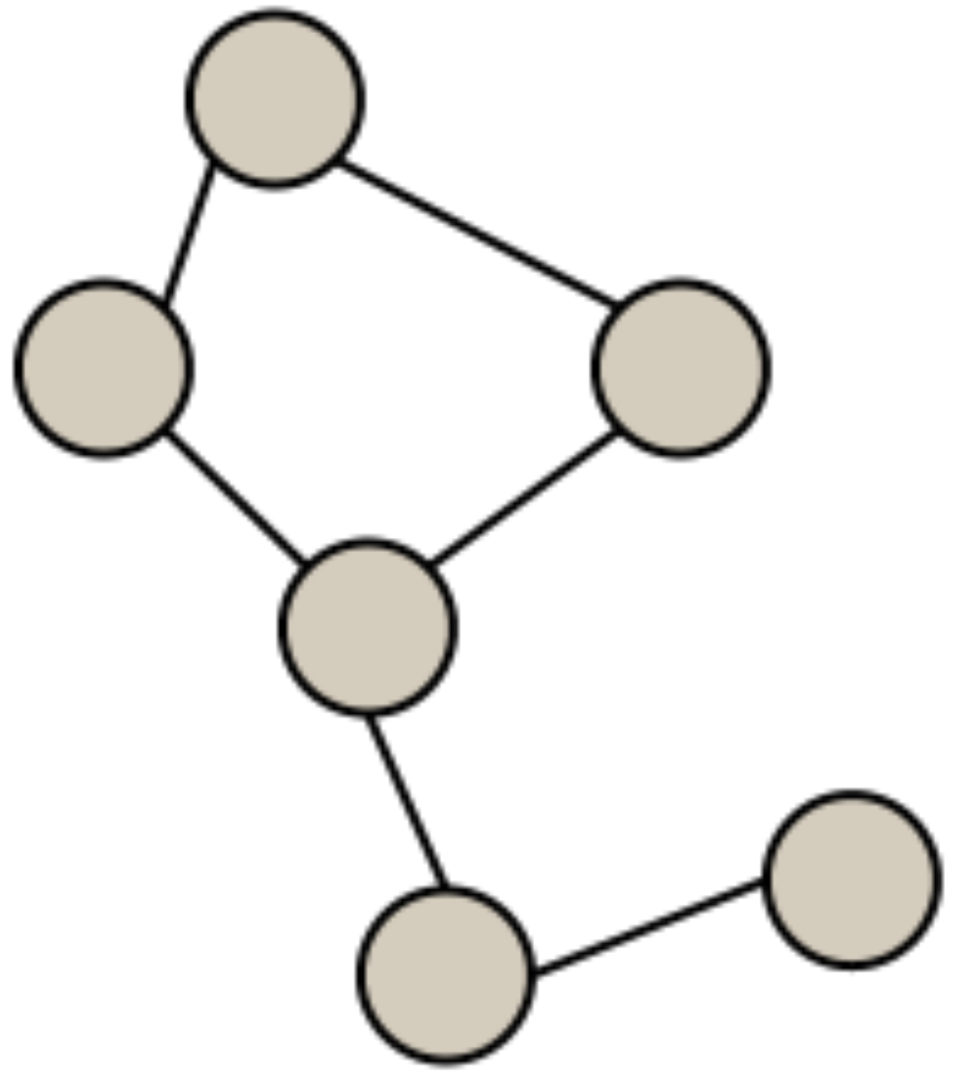
- Create temporary information on client
- Remote mutation hits server
- Server replies with mappings
  - tempids → real ids



Client-only state

- First-class support
- Storage: merged with remote state
- Parser distinguishes local / server
  - knows how to pick remote queries


pikabu.ru

Normalization

- Also in Relay, Falcor
- Om Next can automatically
  - Normalize
  - Denormalize

```
{:people [{:person/name “Alice”
           :person/age 25}
          {:person/name “Bob”
           :person/age 34}]
 :favorites [{:person/name “Bob”
              :person/age 34}]}
```

```
{:people
 [[:person/by-name “Alice”]
  [:person/by-name “Bob”]]
 :favorites
 [[:person/by-name “Bob”]]

 :person/by-name
 {“Alice” {:person/name “Alice”
           :person/age 25}
  “Bob” {:person/name “Bob”
         :person/age 34}}}
```

FOOD GROUPS

Testing

- global app state + immutability = awesome
- Parser abstraction = 1 place
- React = pure function
    - f (data) = UI
- We can just test the UI data tree!

Property-based testing

- example-based
    - specify input / output pairs
- property-based
    - write invariants
    - generate random tests
        - attempt to falsify invariants
        - shrinking

Om Next + test.check

- queries / mutations are data
- generate transactions
  - run against the parser
  - check invariants in resulting state

Demo

Testing recap

1. Generate random transactions
2. Shrink failures
3. Use minimal failure to reproduce bugs

Testing recap

1. ~~Generate random transactions~~
2. ~~Shrink failures~~
3. ~~Use minimal failure to reproduce bugs~~

1. model the user
2. profit

More Om Next

- Recursive UIs
- Heterogeneous UIs
- HTTP Caching
- Custom client side storage
- Streaming

Server

- Clojure preferred / less boilerplate
- Other languages need to implement parser logic
- easier for languages with Transit implementation
- Datomic rocks
  - some people using other DBs

Project status

- very close to beta
-

documentation
    - github.com/omcljs/om/wiki
    - awkay.github.io/om-tutorial/
    - anmonteiro.com

Takeaways

- we can radically simplify UI programming
- Regardless of library / framework
  - your system should support these properties

“Programmers know the benefits of everything and the tradeoffs of nothing”

– Rich Hickey

github.com/anmonteiro/craftconf-demo/

Questions?