# Modeling core financial domains with RESTful + CQRS at Nubank

# Chapter 1: Domain

Avoiding Objects, using Values[1]

[1] https://www.youtube.com/watch?v=ROor6_NGIWU

# Introduction: What is a loan?

A loan is the lending of money (...)
The borrower incurs a debt, and is usually liable to pay interest on that debt until it is repaid, and also to repay the principal amount borrowed.[0]

— What we do at Nubank

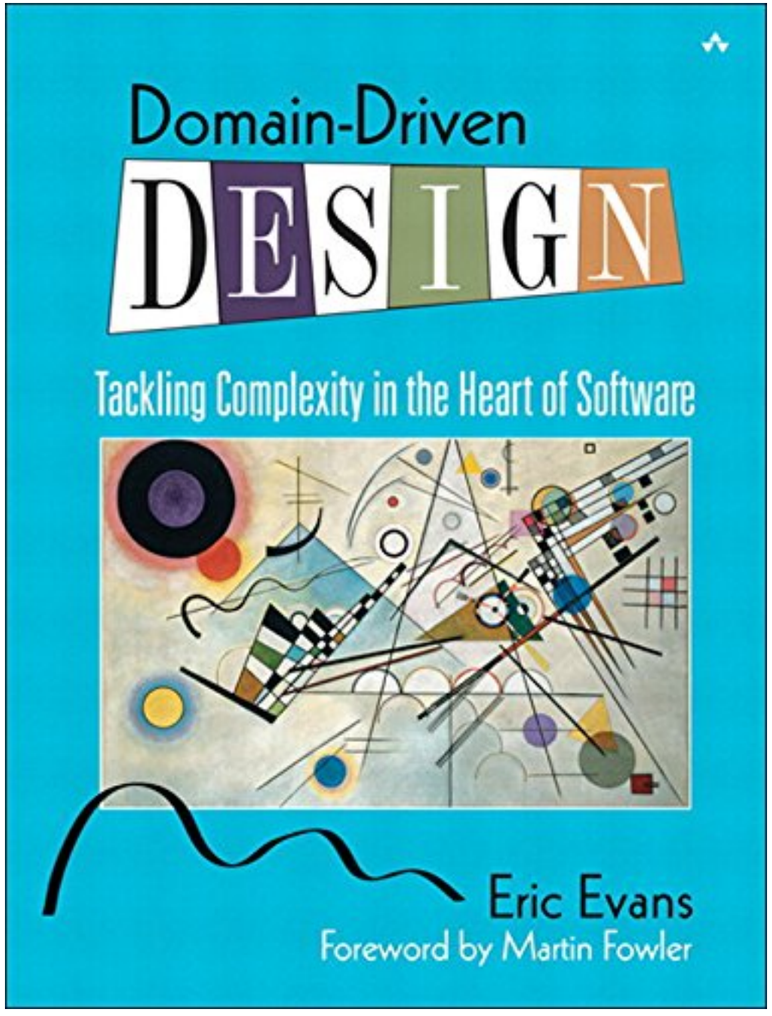
— We better get this right :)

[0] https://en.wikipedia.org/wiki/Loan

# Idea: Use Business Language

A language structured around the **domain model** and **used by all team members** to connect all the activities of the team with the software.[2]

[2] https://en.wikipedia.org/wiki/Domain-driven_design


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

# What is a loan?

— Principal ($)
— Interest Rate (%)
— Late Fee (%)
— Maturity (Date)
— Payment schedule (Bullet, Straight-line, Constant Amortization, ...)
— Purpose (Credit Card, Personal Loan, Mortgage, ...)

# What is a loan?

```
#:loan{:id                   UUID
       :type                 #{:loan.type/personal, :loan.type/credit-card, :loan.type/mortgage, #_...}
       :amortization-schedule #{:loan.amortization-schedule/bullet, :loan.amortization-schedule/price, #_...}
       :installments         Integer
       :contract-date        LocalDate
       :interest-rate        BigDecimal
       :late-interest-rate   BigDecimal
       :late-fee             BigDecimal
       ; ...
       }
```

— Not a class, not an object

— No behaviour implied, no state

— Just data

# **What can** happen **to a loan?**

— Payment
— Late payment
— Anticipation
— Renegotiation
— Cancellation
— "Write-off" (credit default, fraud, ...)

# Idea: Event Sourcing

Capture all **changes** to an application state as a **sequence of events**. [5]

[5] https://martinfowler.com/eaaDev/EventSourcing.html

# **What can** happen **to a loan?**

```
#:loan-event{:id               UUID
             :type             #{:loan-event.type/payment,
                                 :loan-event.type/late,
                                 :loan-event.type/anticipation, #_...}
             :reference-date   LocalDate
             :sort-order       Integer
             :disbursement     BigDecimal
             :discount         BigDecimal
             :interest         BigDecimal
             :late-fee         BigDecimal
             :late-interest    BigDecimal
             ; ...
             }
```

— Remember what happened, not the interpretation

— Explicit date to be relevant, ordered

— Can be stored, queued, logged...

## **What can** happen **to a loan?**

```
(handle-event initial-view issue-event) => view'

(handle-event view' payment-event) => view''

(reduce handle-event initial-view [event ...]) => view''
```

— Rely on pure functions and higher-order functions
— Focus on computation rather than updating objects
— Easy to observe, test

# **What can** happen **to a loan?**

```
(defmethod snapshot-after :loan-event.type/due-payment
  [snapshot-before new-event]
  (let [#_...compute a lot stuff...]
    (-> snapshot-before
        (assoc :debt-snapshot/pv-without-late pv-at-due-date)
        (assoc :debt-snapshot/next-due-date (some-> next-charge :due-date))
        (assoc :debt-snapshot/accruals accruals-after-event)
        (update :debt-snapshot/past-events conj event)
        (update :debt-snapshot/future-charges (comp vec rest))
        (update :debt-snapshot/paid-charges conj paid-charge))))
```

```
#:debt-snapshot{:past-events     [#:loan-event{:reference-date #nu/date "2014-04-01"
                                               :loan           #:loan{:id #uuid "5d586283-14bf-4ed2-a517-456bf0f3194c"}
                                               :type           :loan-event.type/issuance
                                               :disbursement   1000M
                                               :fixed-iof      3.85M
                                               :daily-iof      9.64M
                                               :financed-iof   13.49M
                                               :id             #uuid "5d586283-248d-4feb-8517-f22ac5ad65cc"}
                                  #:loan-event{:id             #uuid "5d5861a2-126d-447a-99c6-3d230fb634aa"
                                               :type           :loan-event.type/late
                                               :reference-date #nu/date "2014-05-01"
                                               :loan           #:loan{:customer-id           #uuid "5d58619b-6662-4416-b773-14a795e
                                                                      :interest-rate         0.11M
                                                                      :yearly-cet            0.15M
                                                                      :installments          6
                                                                      :late-fee              0.02M
                                                                      :type                  :loan.type/personal
                                                                      :due-day               1
                                                                      :contract-date         #nu/date "2014-04-01"
                                                                      :status                :loan.status/pending
                                                                      :id                    #uuid "5d58619b-1b73-4bdb-a44e-1651e5e
                                                                      :initial-days-late     0
                                                                      :late-interest-rate    0.12M
                                                                      :issuer                :loan.issuer/nubank-financeira
                                                                      :holder                :loan.holder/nubank-financeira
                                                                      :first-due-date        #nu/date "2014-05-01"
                                                                      :principal             1000M
                                                                      :grace-days            30
                                                                      :amortization-schedule :loan.amortization-schedule/price}
                                               :charge-index   0}
                                  #:loan-event{:late-fee           -4.7913056686M
                                               :complementary-iof -0.32558288M
                                               :flow-id           "banana:late-payment"
                                               :type              :loan-event.type/late-payment
                                               :loan              #:loan{:customer-id           #uuid "5d58619b-6662-4416-b773-14a7
                                                                         :interest-rate         0.11M
                                                                         :yearly-cet            0.15M
                                                                         :installments          6
                                                                         :late-fee              0.02M
                                                                         :type                  :loan.type/personal
                                                                         :due-day               1
```

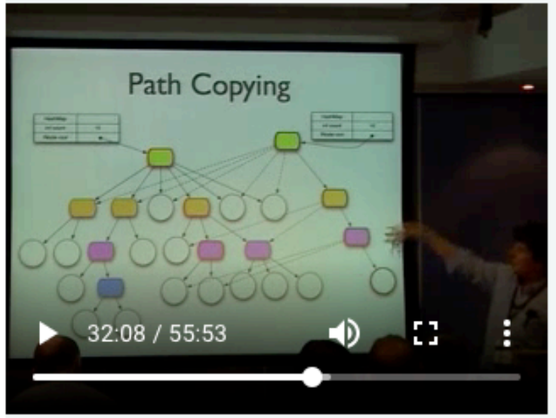
# Idea: Persistent Data Structures [4]

[4] https://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey/

# Persistent Data Structures and Managed References

LIKE 9 BOOKMARKS

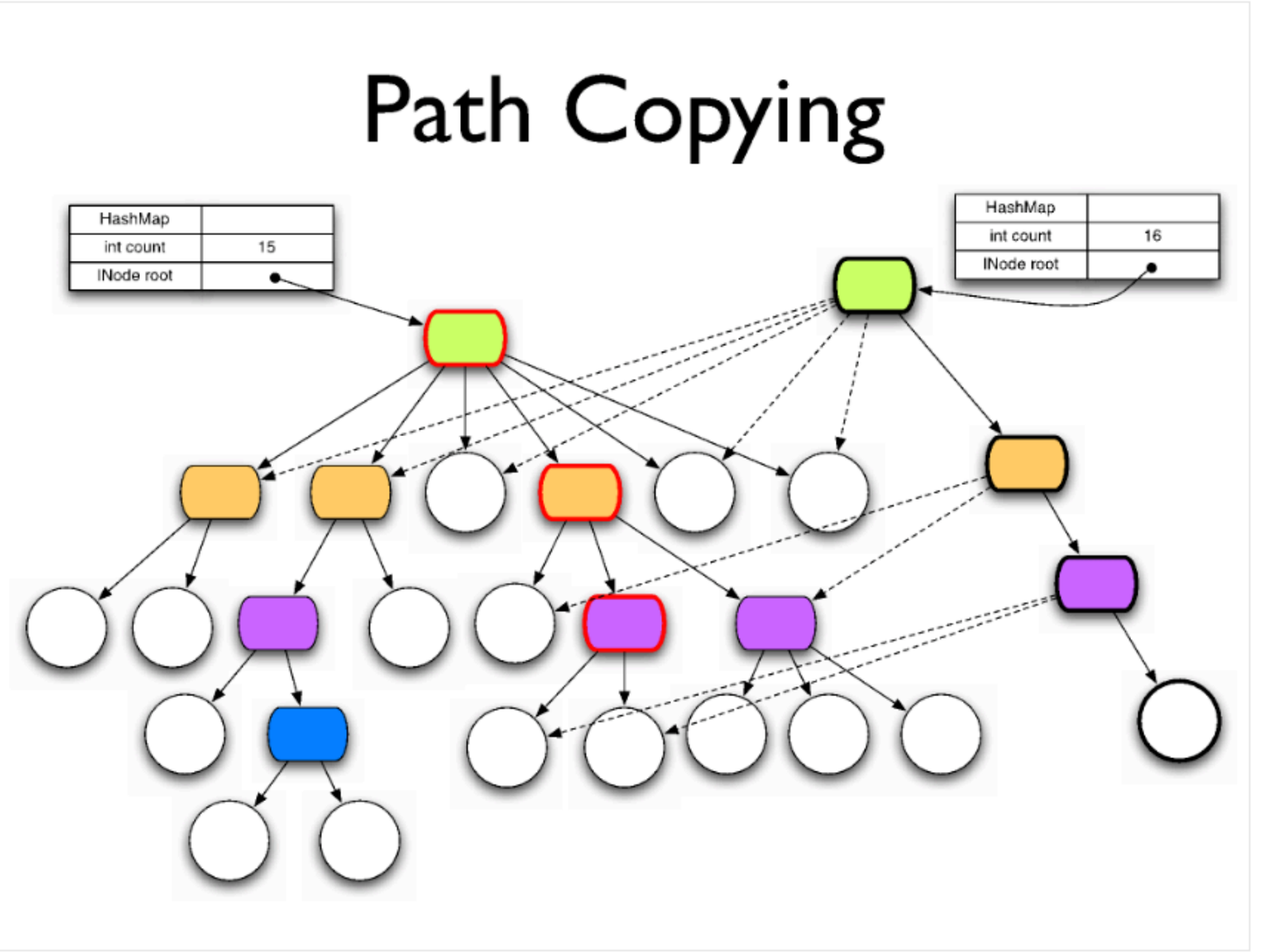
View Presentation Speed: 1X 1.5X 2X




55:53

## Summary

Rich Hickey' presentation is organized around a number of programming concepts: identity, state and values. He explains how to represent composite objects as values and how to deal with change and state, as it is implemented in Clojure.



# Bonus: easy parallelism

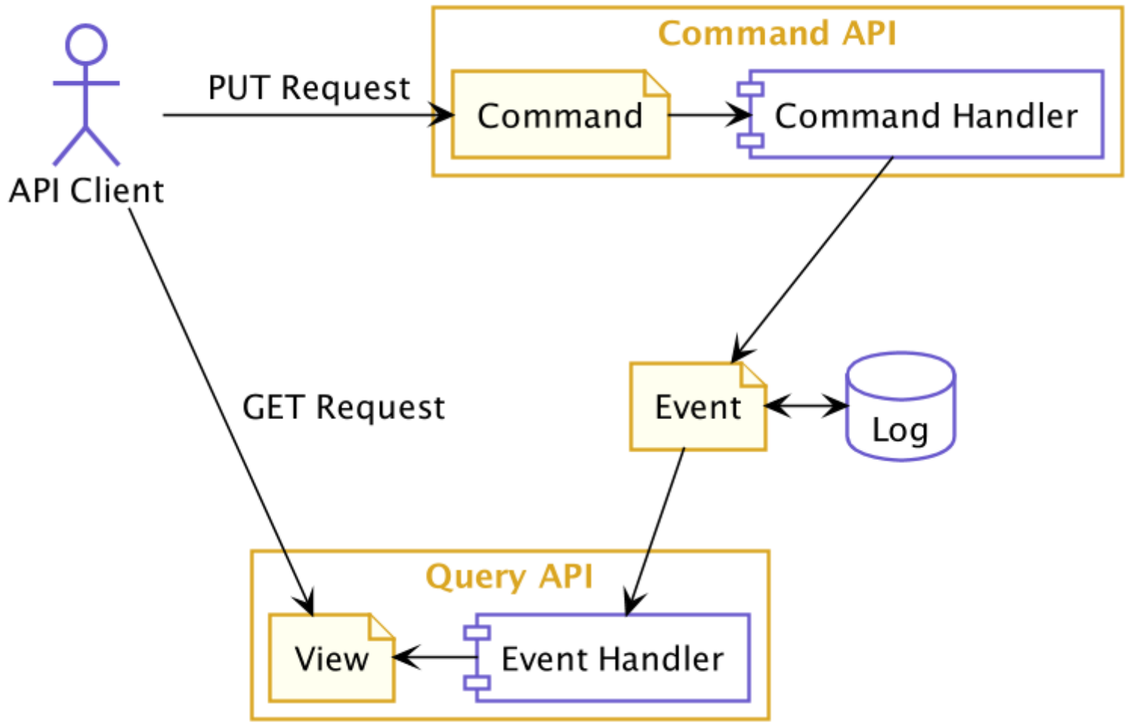
# **What** causes **updates to a loan?**

— Customer borrows a loan
— Customer requests a payment
— Billing system requests an automatic payment
— Billing system detects the loan is late
— Fraud system detects a fraud

# Idea: Command / Query Responsibility Segregation

At its heart is the notion that you can use a **different model** to **update** information than the model you use to **read** information. [6]

[6] https://martinfowler.com/bliki/CQRS.html


Command API
PUT Request
Command
Command Handler
API Client
GET Request
Event
Log
Query API
View
Event Handler

# **What** causes **updates to a loan?**

```
#:command{:id                UUID
          :type              #{:command.type/issuance-request,
                               :command.type/payment-request,
                               :command.type/cancellation-request,
                               #_...}
          :requested-at      LocalDateTime
          :processed-at      LocalDateTime
          :reference-date    LocalDate
          :processing-status #{:command.status/accepted, :command.status/ok, :command.status/failed, #_...}}
```

— Remember the request, assign a unique id

— Track status for asynchronous processing

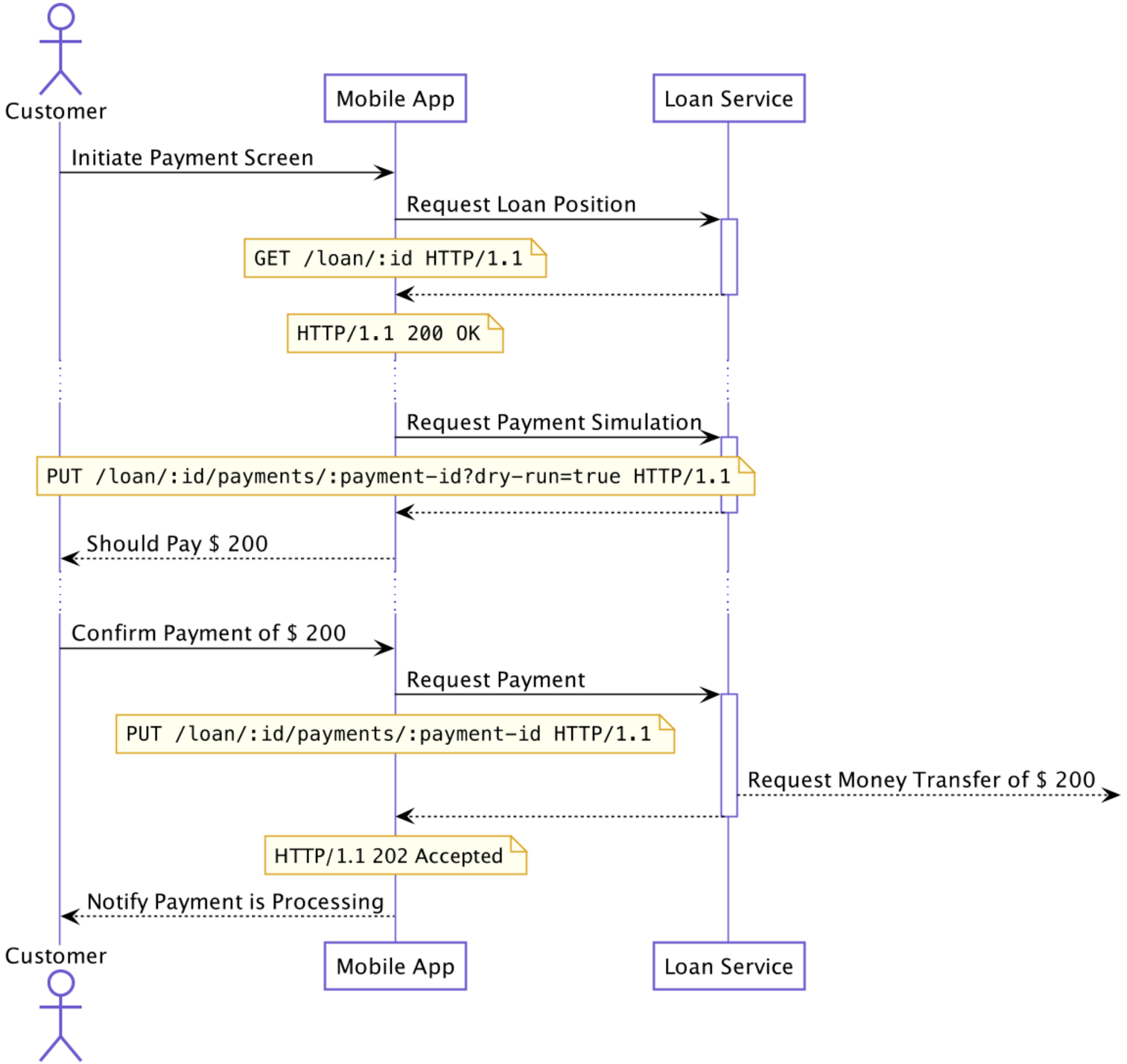
— Can be stored, queued, retried, logged…

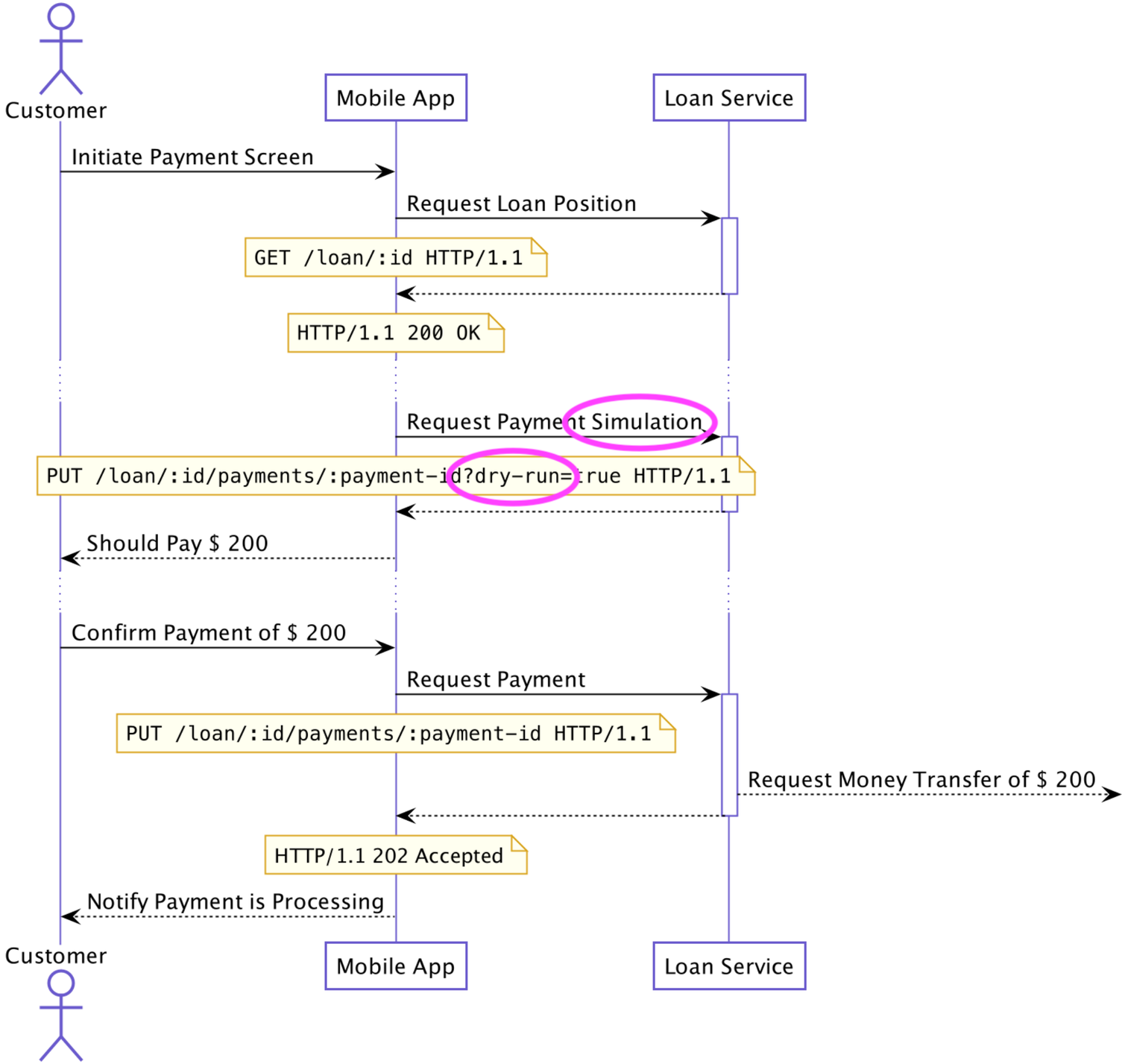
# Chapter 2: Flow

Flows vs. Places [1]

[1] https://www.youtube.com/watch?v=ROor6_NGIWU

# Practical example:
# Designing a Payment API


Customer
Mobile App
Loan Service
Initiate Payment Screen
Request Loan Position
GET /loan/:id HTTP/1.1
HTTP/1.1 200 OK
Request Payment Simulation
PUT /loan/:id/payments/:payment-id?dry-run=true HTTP/1.1
Should Pay $ 200
Confirm Payment of $ 200
Request Payment
PUT /loan/:id/payments/:payment-id HTTP/1.1
Request Money Transfer of $ 200
HTTP/1.1 202 Accepted
Notify Payment is Processing
Customer
Mobile App
Loan Service

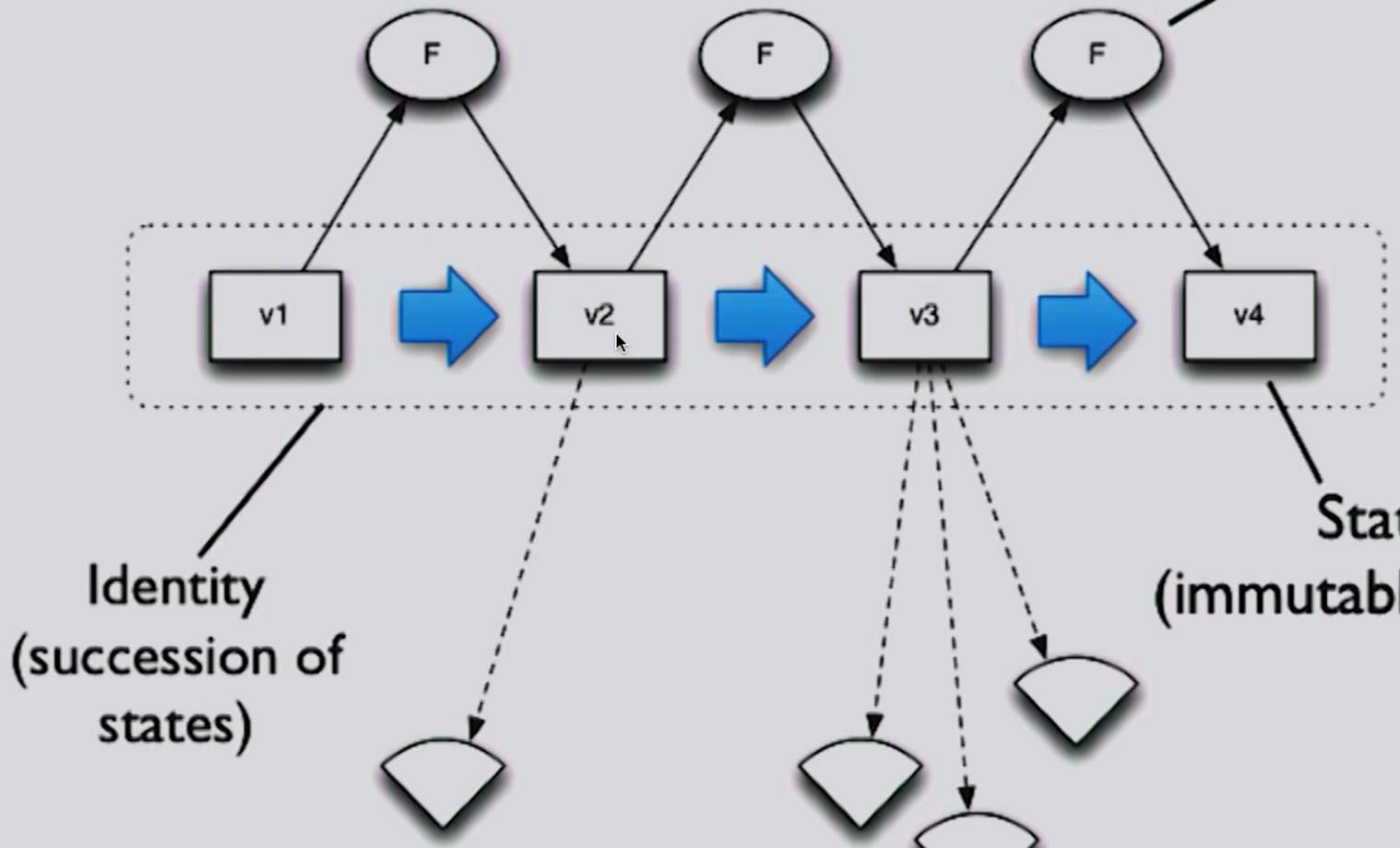

Customer
Mobile App
Loan Service
Initiate Payment Screen
Request Loan Position
GET /loan/:id HTTP/1.1
HTTP/1.1 200 OK
Request Payment Simulation
PUT /loan/:id/payments/:payment-id?dry-run=true HTTP/1.1
Should Pay $ 200
Confirm Payment of $ 200
Request Payment
PUT /loan/:id/payments/:payment-id HTTP/1.1
Request Money Transfer of $ 200
HTTP/1.1 202 Accepted
Notify Payment is Processing
Customer
Mobile App
Loan Service

# Idea:
# Datomic speculative transactions


relevance

2012

# Epochal Time Model



# Quick Detour: Datomic's data model

| entity | attribute | value | tx |
| --- | --- | --- | --- |

# Quick Detour: Datomic's data model

```
[{:name "Jane" :phone "12341234"}]
```

| entity | attribute | value | tx |
|---|---|---|---|
| customer | name | Jane | 1 |
| customer | phone | 12341234 | 1 |

# Quick Detour: Datomic's data model

```
[{:name "Jane" :phone "912341234"}]
```

| entity | attribute | value | tx |
|---|---|---|---|
| customer | name | Jane | 1 |
| customer | phone | 12341234 | 1 |
| customer | phone | 912341234 | 2 |

# Quick Detour: Datomic's data model

```
[{:name "Jane" :phone "912341234"}
 {:name "Joe"}]
```

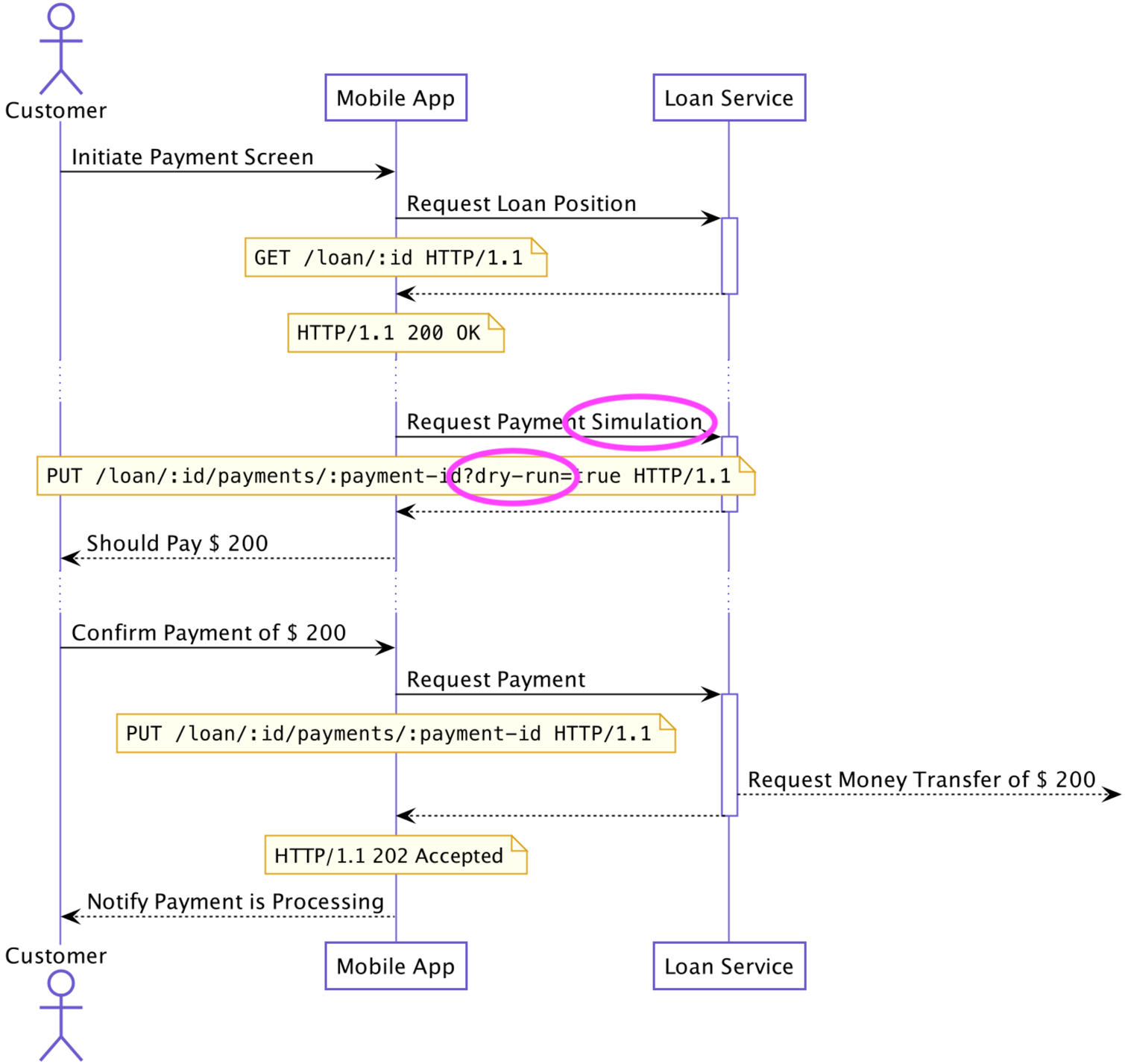
| entity | attribute | value | tx |
|---|---|---|---|
| customer | name | Jane | 1 |
| customer | phone | 12341234 | 1 |
| customer | phone | 912341234 | 2 |
| customer2 | name | Joe | 3 |

# Datomic speculative transactions

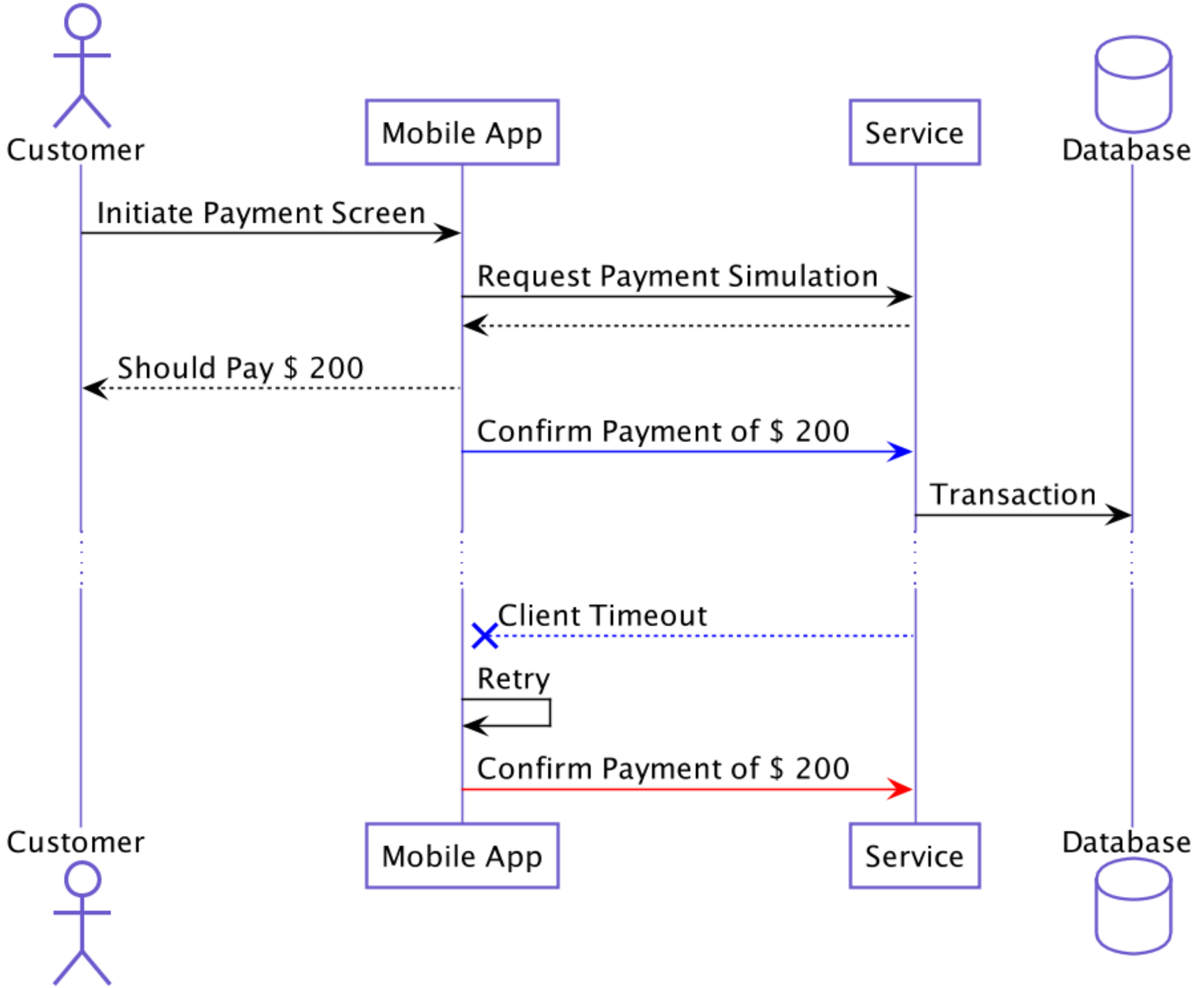
```
(let [events  (command->events command)
      tx-data (prepare-transaction events)]
    (-> (if dry-run?
            (datomic.api/with db tx-data) ;; applies in-memory
            (datomic.api/transact datomic tx-data)) ;; persists to storage
        (get :db-after) ;; lazy representation of entire db
        (proceed-as-usual)))
```

# Datomic speculative transactions

- Database is handled as a (lazy) value
  - Pure functions
- Same code path for simulation or transaction
  - Reduced bugs by avoiding corner cases


Customer
Mobile App
Loan Service
Initiate Payment Screen
Request Loan Position
GET /loan/:id HTTP/1.1
HTTP/1.1 200 OK
Request Payment Simulation
PUT /loan/:id/payments/:payment-id?dry-run=true HTTP/1.1
Should Pay $ 200
Confirm Payment of $ 200
Request Payment
PUT /loan/:id/payments/:payment-id HTTP/1.1
Request Money Transfer of $ 200
HTTP/1.1 202 Accepted
Notify Payment is Processing
Customer
Mobile App
Loan Service

# Problem:
# Idempotent updates

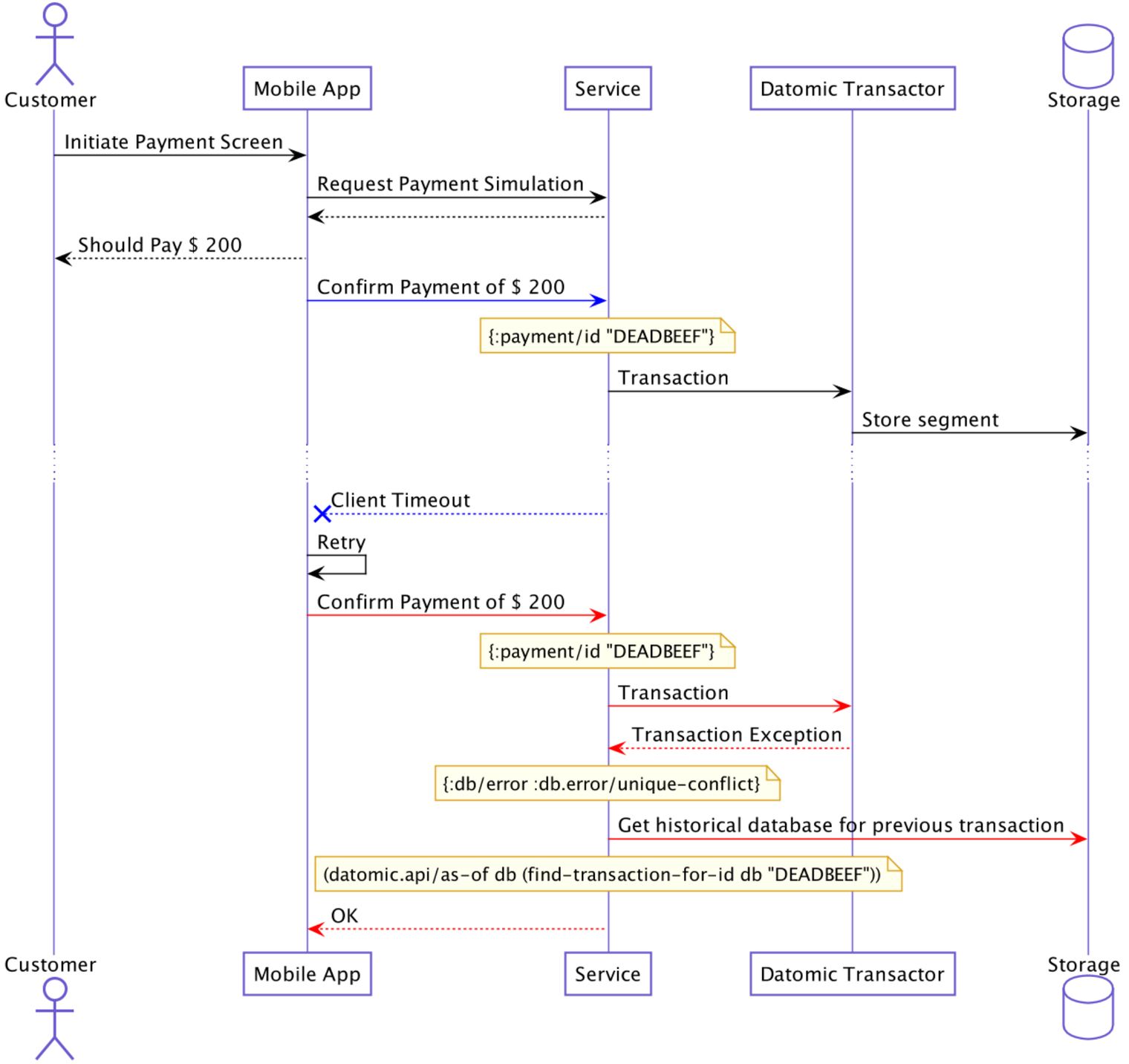

Customer
Mobile App
Service
Database
Initiate Payment Screen
Request Payment Simulation
Should Pay $ 200
Confirm Payment of $ 200
Transaction
Client Timeout
Retry
Confirm Payment of $ 200
Customer
Mobile App
Service
Database

# Idea:
# Datomic's historical database

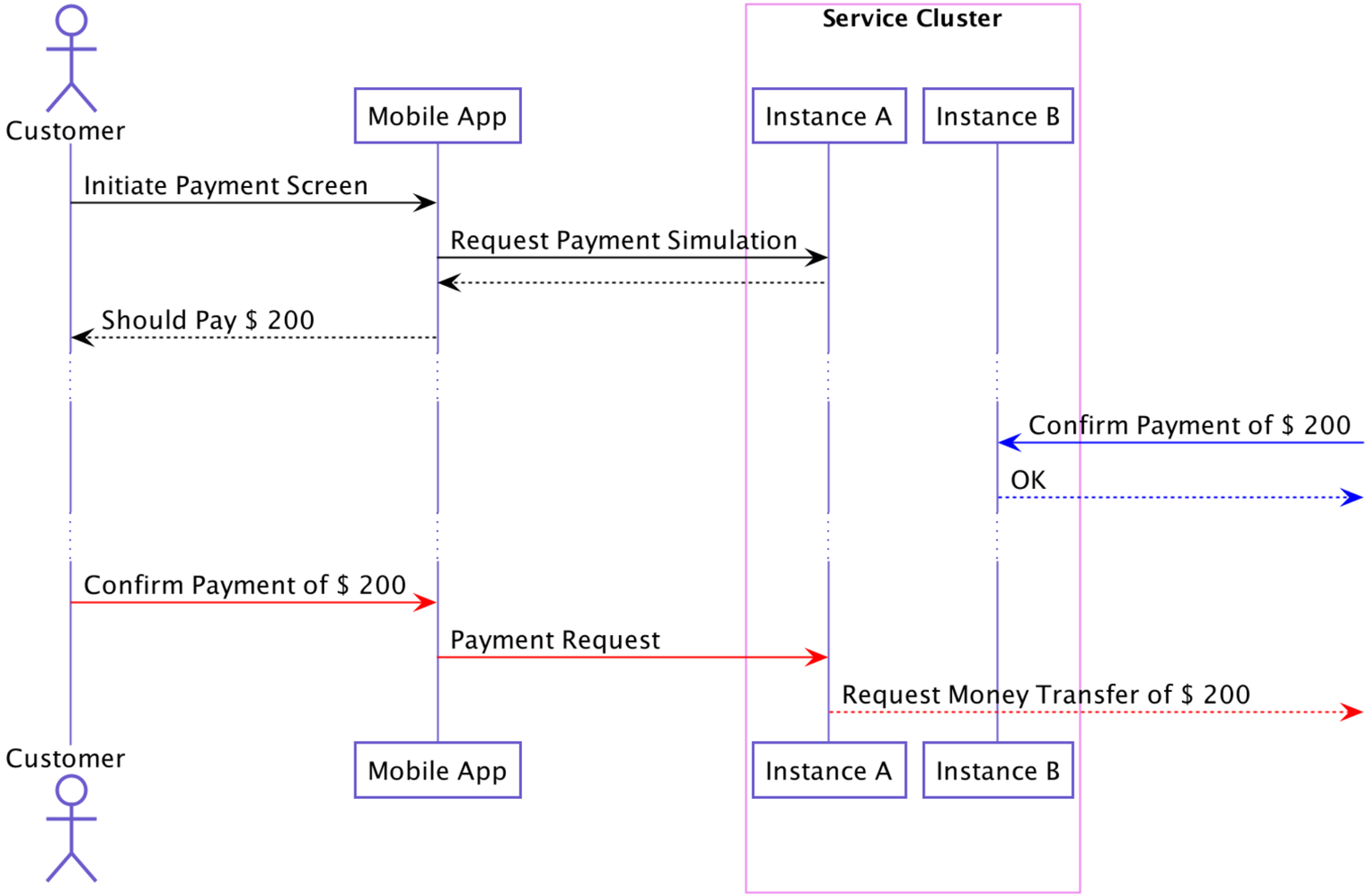
# as-of

function
Usage: (as-of db t)
Returns the value of the database as of some point t, inclusive.
t can be a transaction number, transaction ID, or Date. [11]

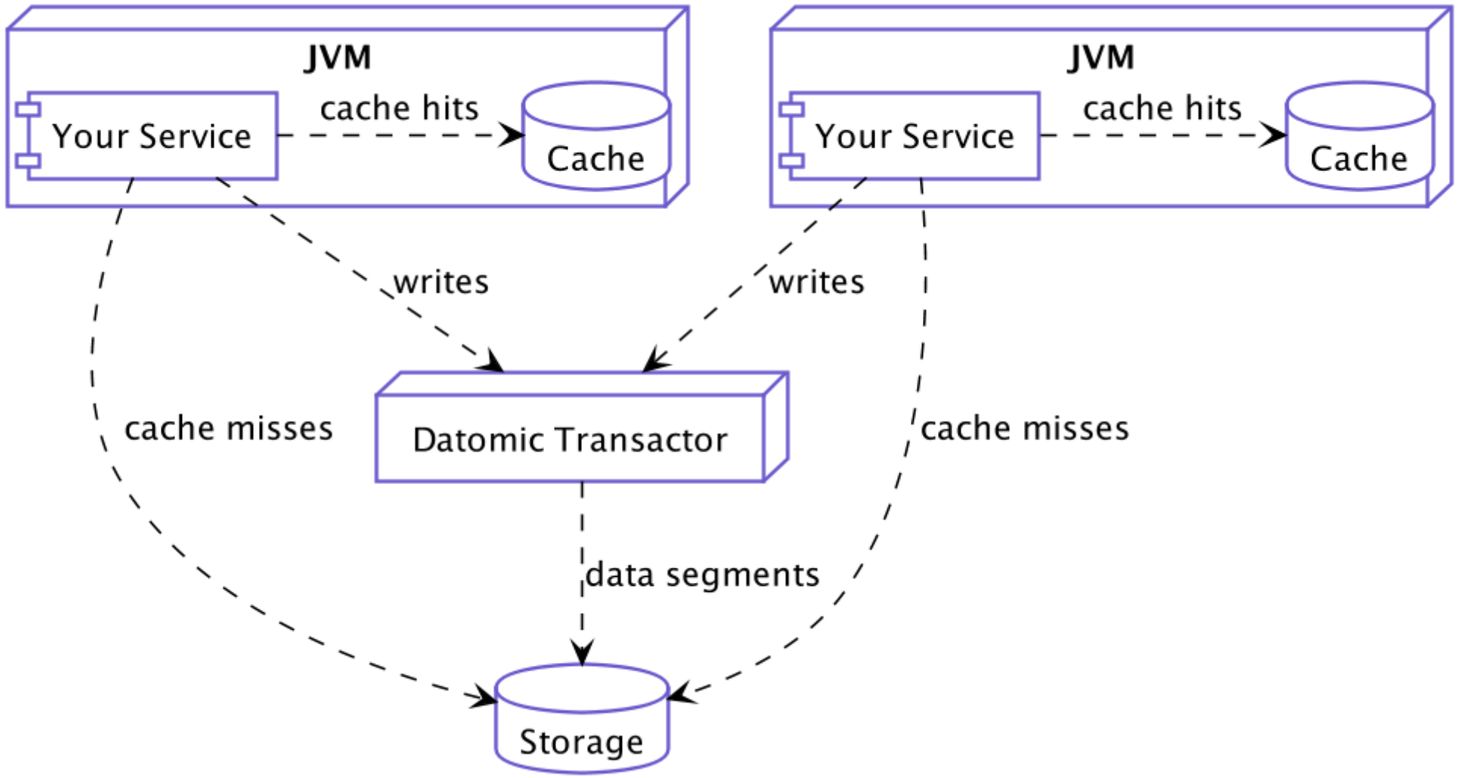
[11] https://docs.datomic.com/on-prem/clojure/index.html#datomic.api/as-of


Customer
Mobile App
Service
Datomic Transactor
Storage
Initiate Payment Screen
Request Payment Simulation
Should Pay $ 200
Confirm Payment of $ 200
{:payment/id "DEADBEEF"}
Transaction
Store segment
Client Timeout
Retry
Confirm Payment of $ 200
{:payment/id "DEADBEEF"}
Transaction
Transaction Exception
{:db/error :db.error/unique-conflict}
Get historical database for previous transaction
(datomic.api/as-of db (find-transaction-for-id db "DEADBEEF"))
OK
Customer
Mobile App
Service
Datomic Transactor
Storage

# Problem:
# Concurrent updates


Service Cluster
Customer
Mobile App
Instance A
Instance B
Initiate Payment Screen
Request Payment Simulation
Should Pay $ 200
Confirm Payment of $ 200
OK
Confirm Payment of $ 200
Payment Request
Request Money Transfer of $ 200
Customer
Mobile App
Instance A
Instance B

# Quick detour: Datomic's architecture

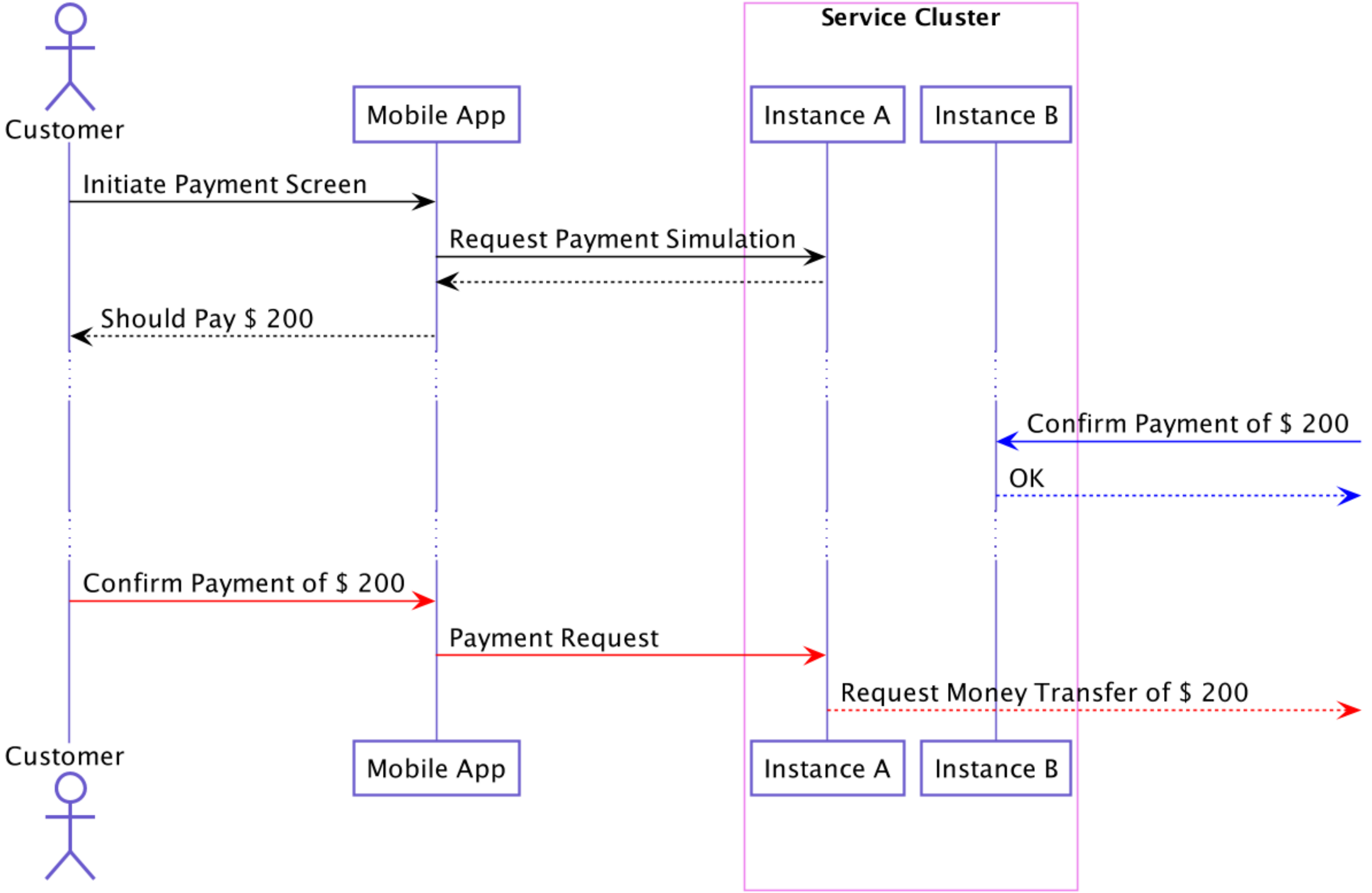


# Quick detour: Datomic's architecture

- High-availability for reads
  - Cache locality
  - Cache eviction handled for you automagically
  - Optional: Memcached layer
- ACID transactions

# Quick detour: Datomic's architecture

- High-availability for reads
  - Cache locality
    - Trade-off: eventually consistent
  - Cache eviction handled for you automagically
  - Optional: Memcached layer
- ACID transactions


Service Cluster
Customer
Mobile App
Instance A
Instance B
Initiate Payment Screen
Request Payment Simulation
Should Pay $ 200
Confirm Payment of $ 200
OK
Confirm Payment of $ 200
Payment Request
Request Money Transfer of $ 200
Customer
Mobile App
Instance A
Instance B

# Idea:
# Datomic Transaction Functions & HTTP Conditional Headers

:db/cas

The compare-and-swap function takes four arguments: an entity id, an attribute, an old value, and a new value.

If the entity has the old value for attribute, then the new value will be asserted.

Otherwise, the transaction will abort and throw an exception. [9]

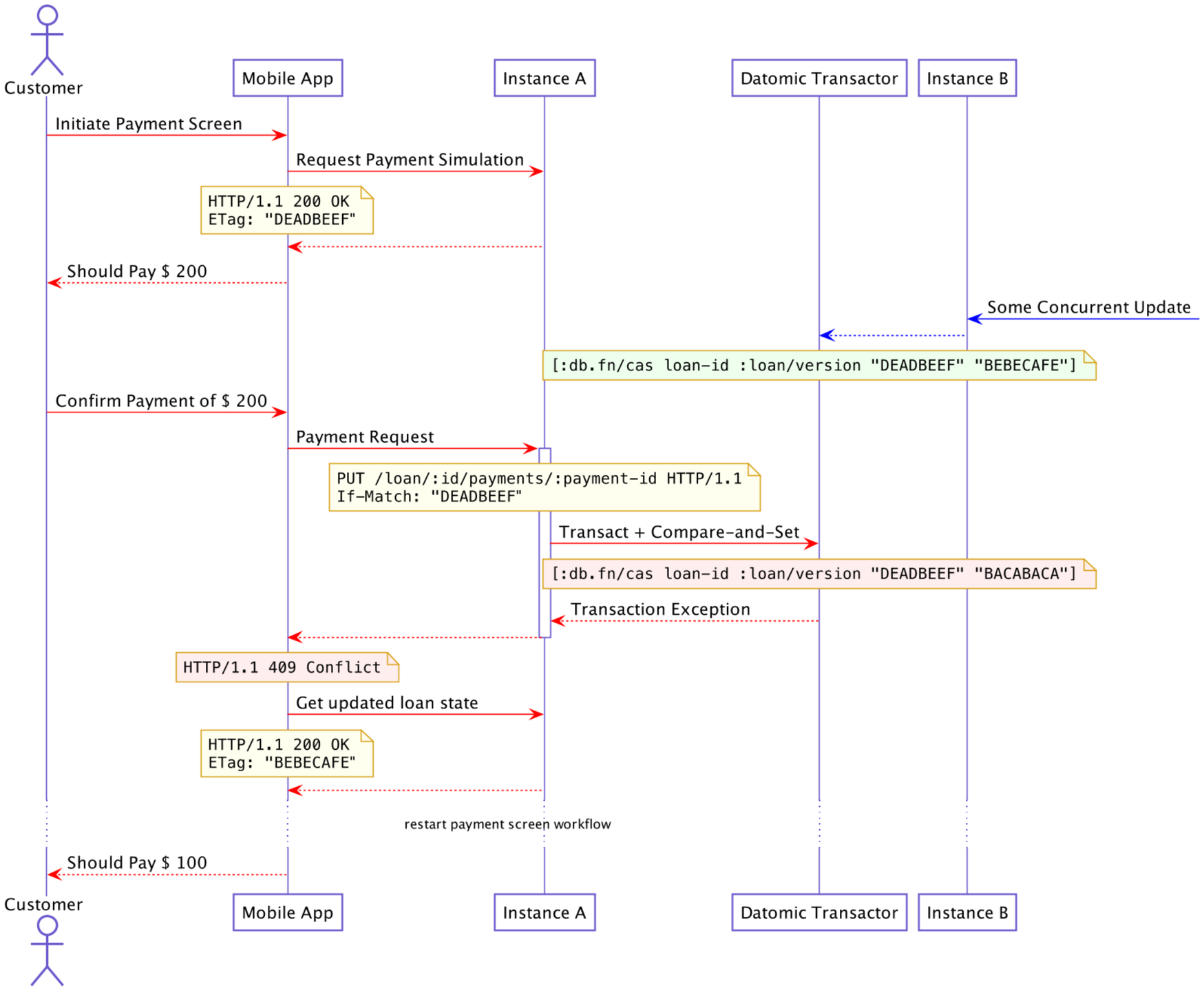
[9] https://docs.datomic.com/cloud/transactions/transaction-functions.html#db-cas

# If-Match

The If-Match HTTP request header makes the request conditional.

For other methods, and in particular for PUT, If-Match can be used to prevent the lost update problem. It can check if the modification of a resource that the user wants to upload (will not override another change that has been done since the original resource was fetched. If the request cannot be fulfilled, the 412 (Precondition Failed) response is returned. [10]

[10] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/If-Match


Customer
Mobile App
Instance A
Datomic Transactor
Instance B
Initiate Payment Screen
Request Payment Simulation
HTTP/1.1 200 OK
ETag: "DEADBEEF"
Should Pay $ 200
Some Concurrent Update
[:db.fn/cas loan-id :loan/version "DEADBEEF" "BEBECAFE"]
Confirm Payment of $ 200
Payment Request
PUT /loan/:id/payments/:payment-id HTTP/1.1
If-Match: "DEADBEEF"
Transact + Compare-and-Set
[:db.fn/cas loan-id :loan/version "DEADBEEF" "BACABACA"]
Transaction Exception
HTTP/1.1 409 Conflict
Get updated loan state
HTTP/1.1 200 OK
ETag: "BEBECAFE"
restart payment screen workflow
Should Pay $ 100
Customer
Mobile App
Instance A
Datomic Transactor
Instance B

# HTTP <-> Datomic Feature Correspondence

| Verb | Req Header | Resp Header | Parameter | Datomic API |
|---|---|---|---|---|
| GET | - | ETag | - | Historical DB (`as-of`) |
| PUT | If-Match | - | - | ACID Transactions + Compare-and-Swap (`:db.fn/cas`) |
| PUT | - | - | dry-run | Virtual transactions (`with`) |

# Chapter 3: System

# Introducing: the actors

— Customer: Borrows and pays loans.

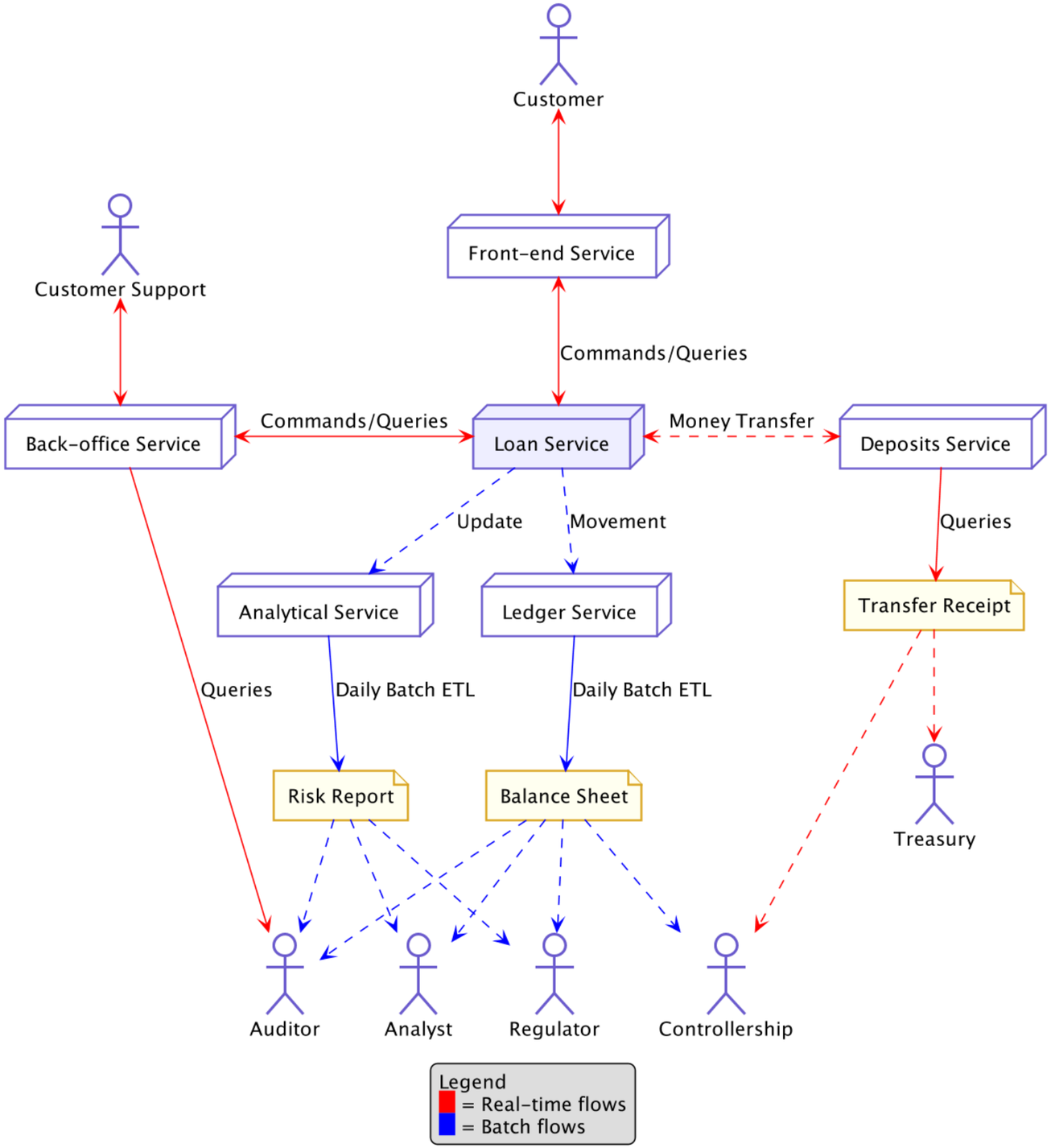
— Treasury: Manage cash stock for daily operation.

— Controllership: Reconciliation, incorporate product financials on balance sheet, collect taxes.

— Analysts: Understanding customer, product and portfolio behaviors.

— Regulator: Ensure country-level financial system health.

— Auditor: Cross-check data from different sources.


Customer
Front-end Service
Customer Support
Commands/Queries
Back-office Service
Commands/Queries
Loan Service
Money Transfer
Deposits Service
Update
Movement
Queries
Analytical Service
Ledger Service
Transfer Receipt
Queries
Daily Batch ETL
Daily Batch ETL
Risk Report
Balance Sheet
Treasury
Auditor
Analyst
Regulator
Controllership
Legend
= Real-time flows
= Batch flows

# Idea: Reporting Databases

The **operational needs** and the **reporting needs** are often quite **different** - with different requirements from a schema and different data access patterns.
(...)
It's often a wise idea to separate the reporting needs into a reporting database, which takes a copy of the essential operational data but **represents it in a different schema**. [7]

[7] https://martinfowler.com/bliki/ReportingDatabase.html

# Data shapes

| desired read shape | Datomic supports via |
|---|---|
| k/v | AVET |
| row | EAVT |
| column | AEVT |
| document | EAVT + pull |
| graph | VAET |

# Keeping sanity

- Separate transactional workflows from reporting workflows
  - Denormalized views with the right granularity required by the actors
- Focused core domain services
  - e.g. thin REST API, easy to test
- Auditing is orthogonal
  - Datomic's reified transactions help with that too [8]

[8] https://www.youtube.com/watch?v=7lm3K8zVOdY

# Conclusion

| Feature | Ideas |
| --- | --- |
| Functional Programming + Values | Use Business Language, Domain Driven Design |
| Persistent Data Structures | Event Sourcing |
| Speculative transactions | Command / Query Responsibility Segregation |
| Reified Transactions | Orthogonal Audit |
| Data shapes | Reporting Databases |

# Conclusion

— Know your stakeholders and its timing requirements (real-time vs. batch, sync vs. async, etc)

— Fight entropy

— Use separate databases and shapes for reporting

— Keep tooling simple

Thank you!

Henrique Alves hcarvalhoalves@gmail.com
Gustavo Barrancos gbarrancos@gmail.com