Atlassian

Event sourcing and CQRS from the trenches

SIDNEY SHEK · ARCHITECT · ATLASSIAN · @SIDNEYSHEK

Universe of Users

users

Id	Name	Username	APIKey
1	Homer	homer	d0a
2	Bart	bart	f00
3	Maggie	maggie	baa

Universe of Users

users

Id	Name	Username	APIKey
1	Homer	homer	d0a
2	Bart	bart	f00
3	Lisa Jr	maggie	baa

Universe of Users

users

Id	Name	Username	APIKey
1	Homer	homers	d0a
2	Bart	bart	f00
3	Lisa Jr	maggie	baa

Universe of Users

users

Id	Name	Username	APIKey
1	Homer	homer	d0a
2	Bart	bart	f00
3	Maggie	maggie	baa

events

Seq	Event	Time
123	SetUsername(3, Maggie)	0

Universe of Users

users

Id	Name	Username	APIKey
1	Homer	homer	d0a
2	Bart	bart	f00
3	Lisa Jr	maggie	baa

events

Seq	Event	Time
123	SetUsername(3, Maggie)	0
124	SetName(3, Lisa Jr)	10

Universe of Users

users

Id	Name	Username	APIKey
1	Homer	homers	d0a
2	Bart	bart	f00
3	Lisa Jr	maggie	baa

events

Seq	Event	Time
123	SetUsername(3, Maggie)	0
124	SetName(3, Lisa Jr)	10
125	SetUsername(1, homers)	15

Universe of Users

users_new

Id	Name	Derived
1	Homer	Homer1
2	Bart	Bart2
3	Lisa Jr	Lisa Jr3

users

Id	Name	Username	APIKey
1	Homer	homers	d0a
2	Bart	bart	f00
3	Lisa Jr	maggie	baa

events

Seq	Event	Time
123	SetUsername(3, Maggie)	0
124	SetName(3, Lisa Jr)	10
125	SetUsername(1, homers)	15

Universe of Users

users_new

Id	Name	Derived
1	Homer	Homer1
2	Bart	Bart2
3	Lisa Jr	Lisa Jr3

users

Id	Name	Username	APIKey
1	Homer	homers	d0a
2	Bart	bart	f00
3	Lisa Jr	maggie	baa

events

Seq	Event	Time
123	SetUsername(3, Maggie)	0
124	SetName(3, Lisa Jr)	10
125	SetUsername(1, homers)	15

elastic

Our Identity System requirements

Our Identity System requirements

- Users, groups and memberships

Our Identity System requirements

- Users, groups and memberships
  - Searching for users

Our Identity System requirements

- Users, groups and memberships
  - Searching for users
  - Retrieve by email

Our Identity System requirements

- Users, groups and memberships
  - Searching for users
  - Retrieve by email
- High volume low latency reads

Our Identity System requirements

- Users, groups and memberships
  - Searching for users
  - Retrieve by email
  - Incremental synchronisation
- High volume low latency reads

Our Identity System requirements

- Users, groups and memberships
  - Searching for users
  - Retrieve by email
  - Incremental synchronisation
- Audit trails for changes
- High volume low latency reads

Our Identity System requirements

- Users, groups and memberships
  - Searching for users
  - Retrieve by email
  - Incremental synchronisation
- Audit trails for changes
- High volume low latency reads
- Highly available
  - Disaster recovery
  - Zero-downtime upgrades

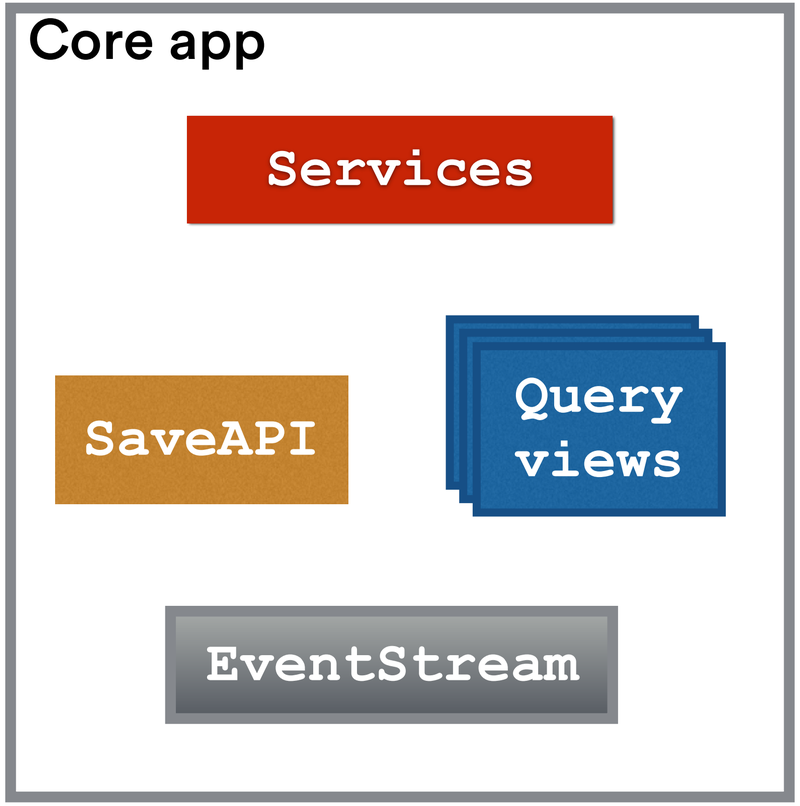
Our Identity System requirements

- Users, groups and memberships
  - Searching for users
  - Retrieve by email
  - Incremental synchronisation
- Audit trails for changes
- High volume low latency reads
- Highly available
  - Disaster recovery
  - Zero-downtime upgrades
- Testing with production-like data

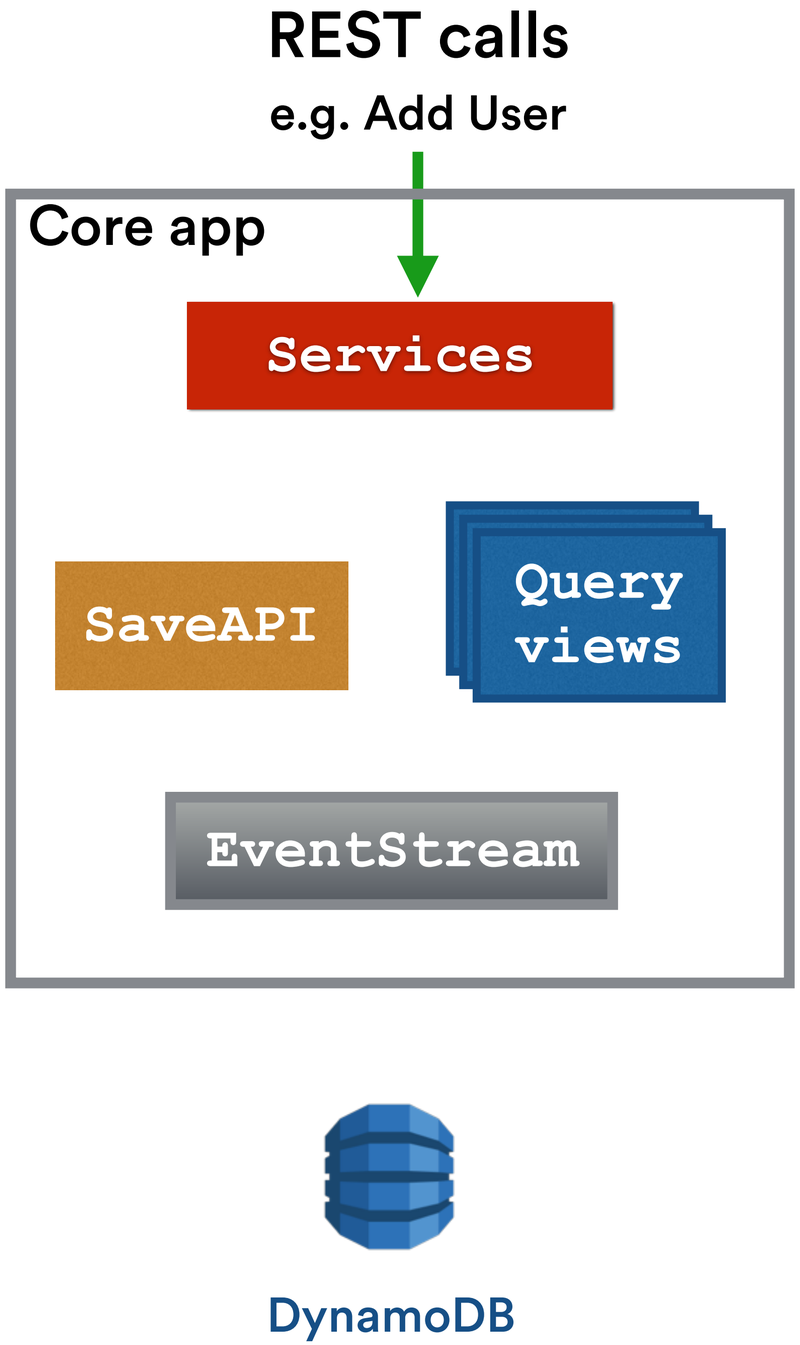
Evolving the architecture

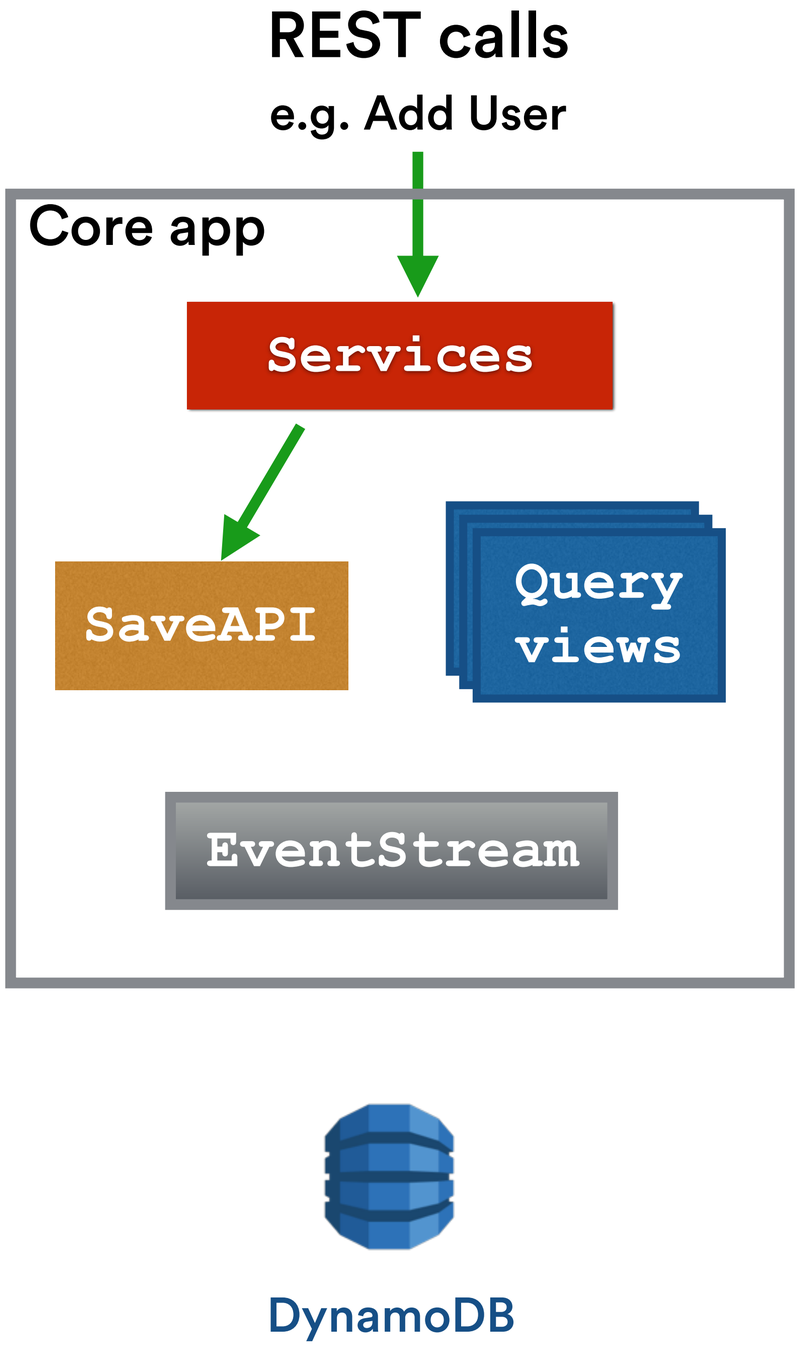
REST calls

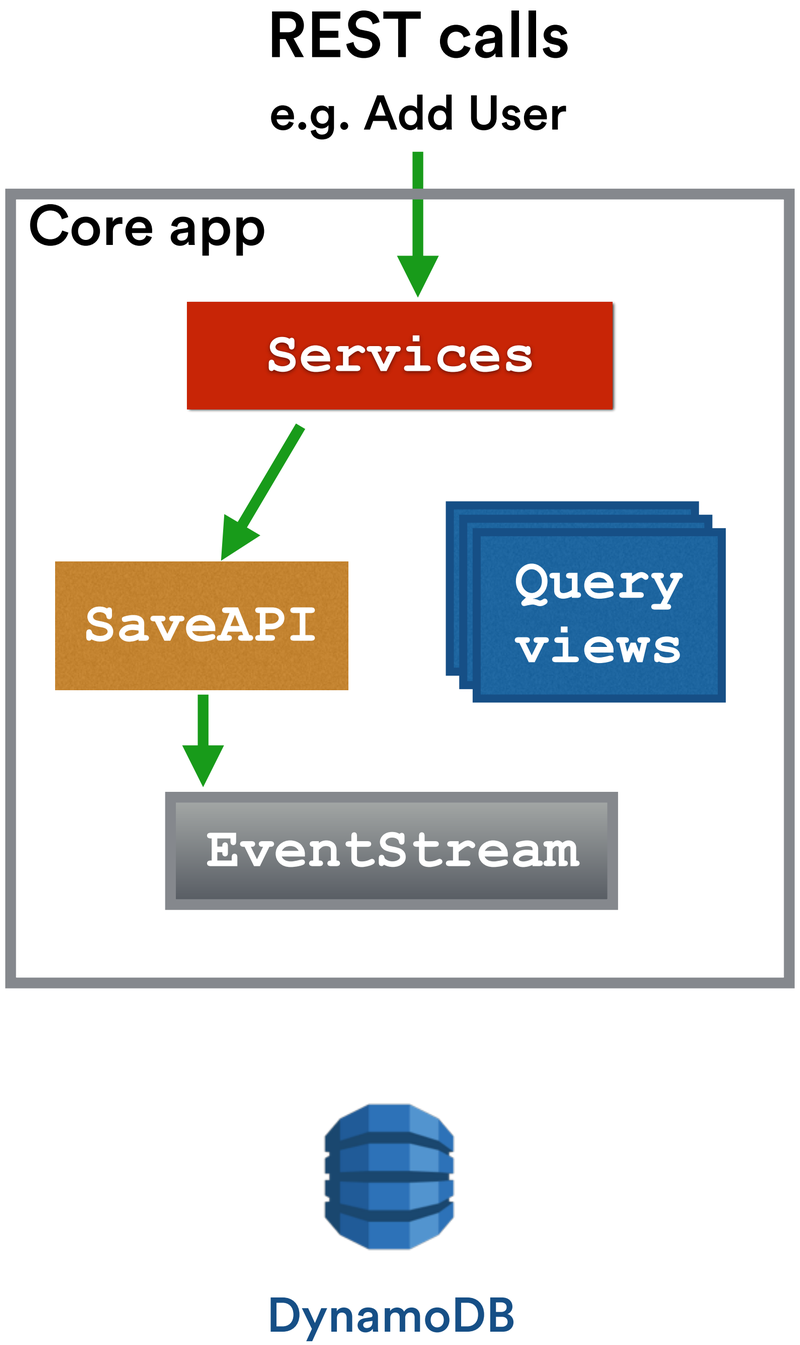
e.g. Add User

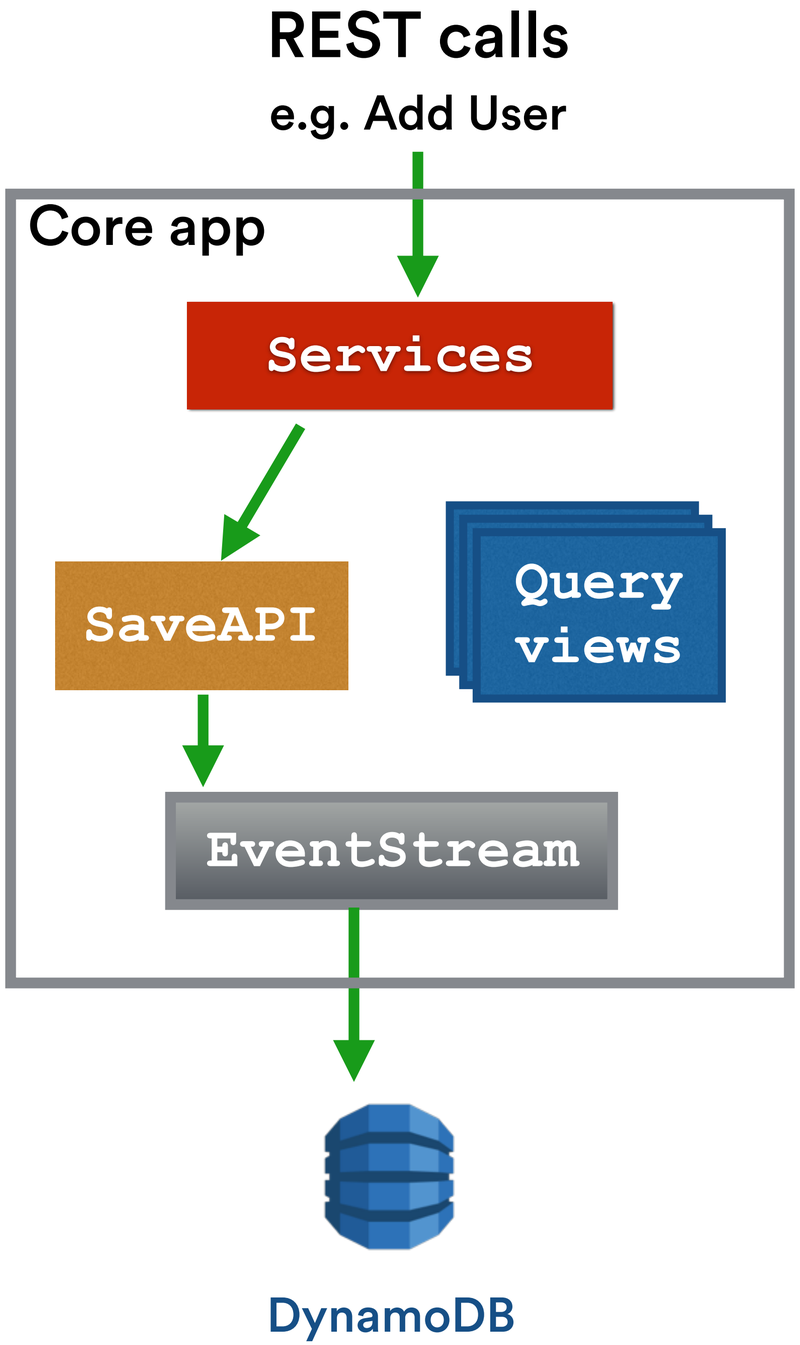


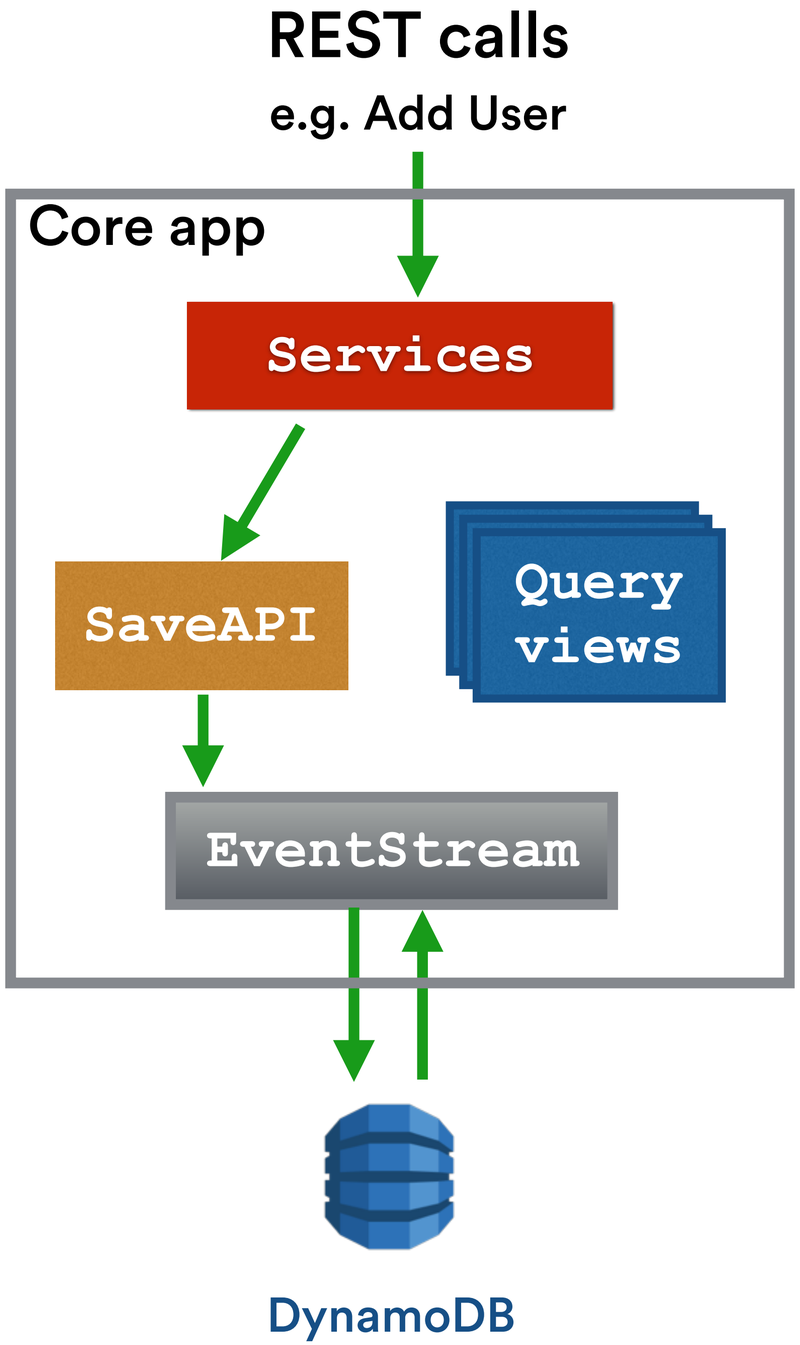
DynamoDB

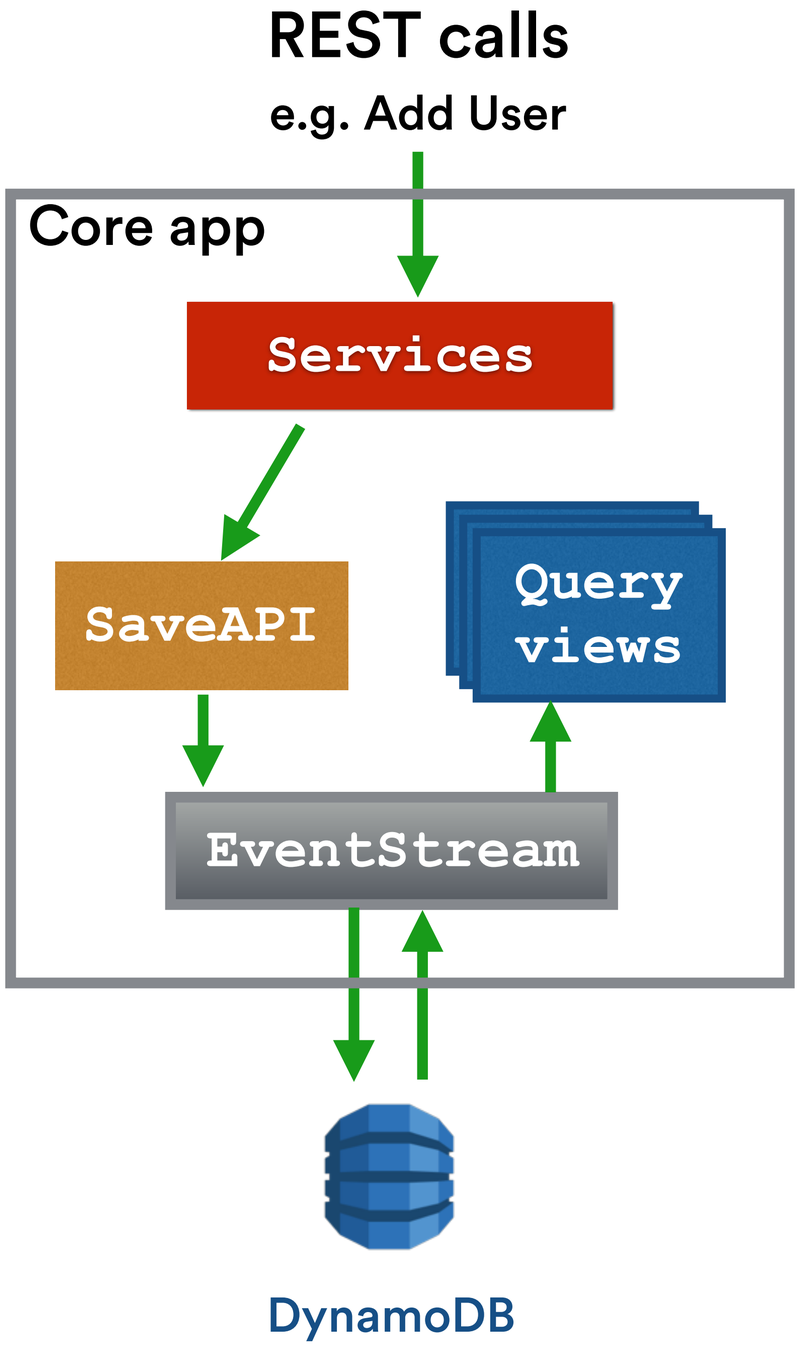

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

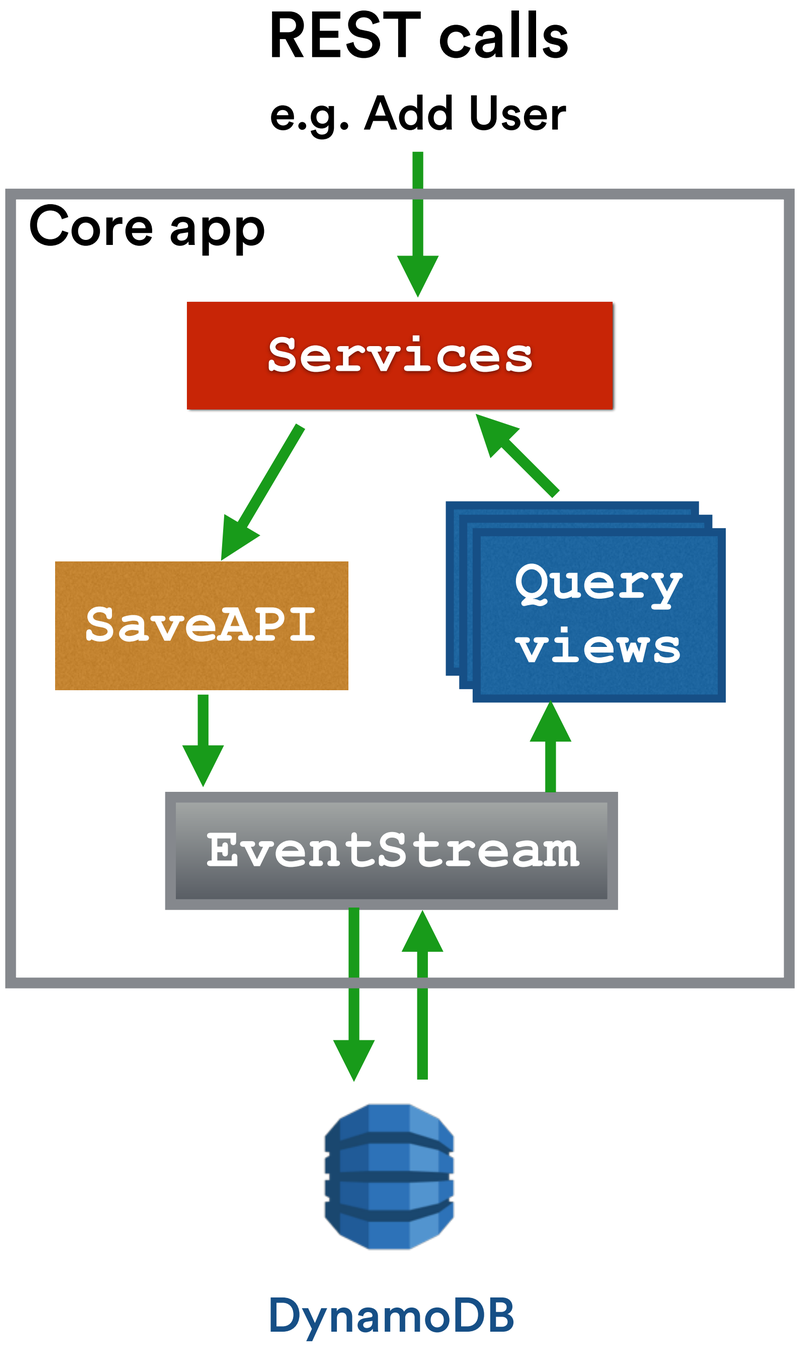

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB


REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

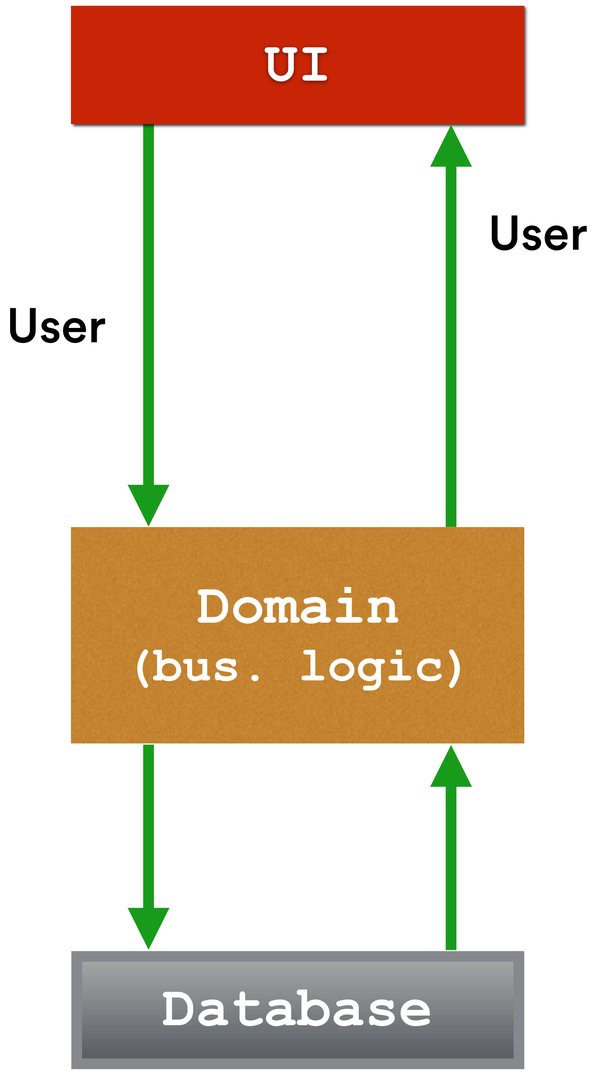

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

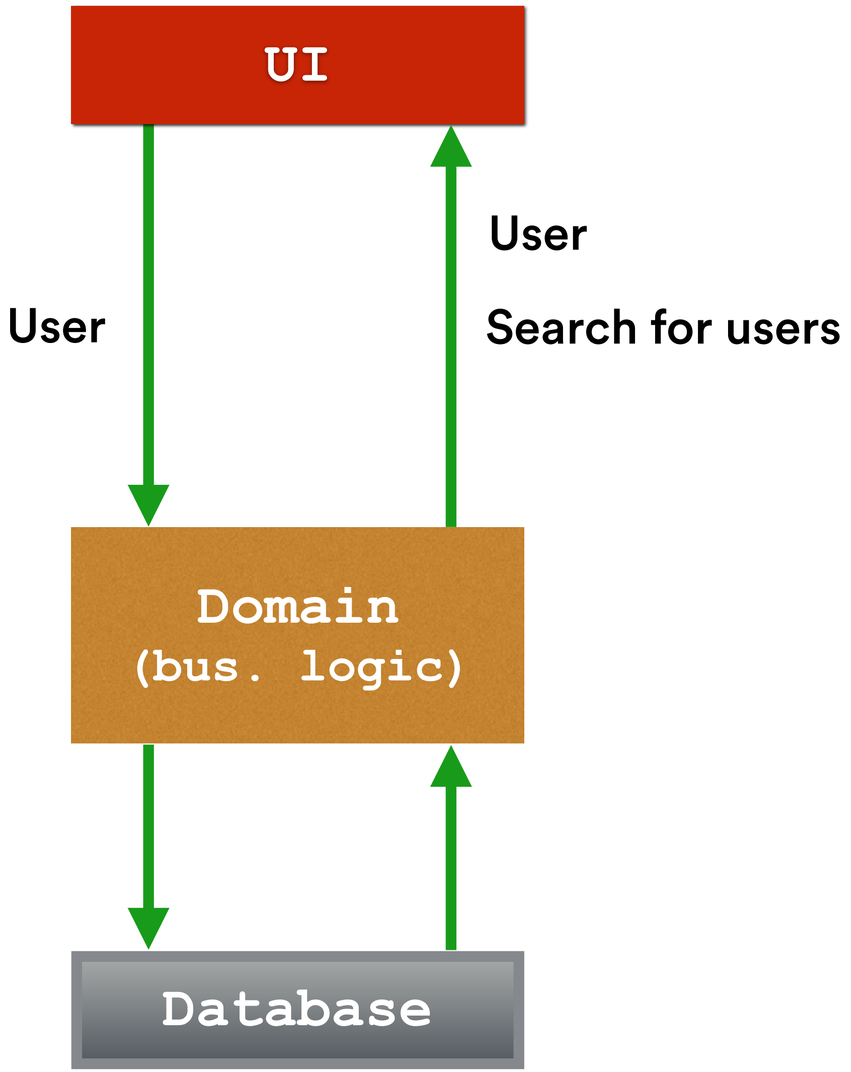

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

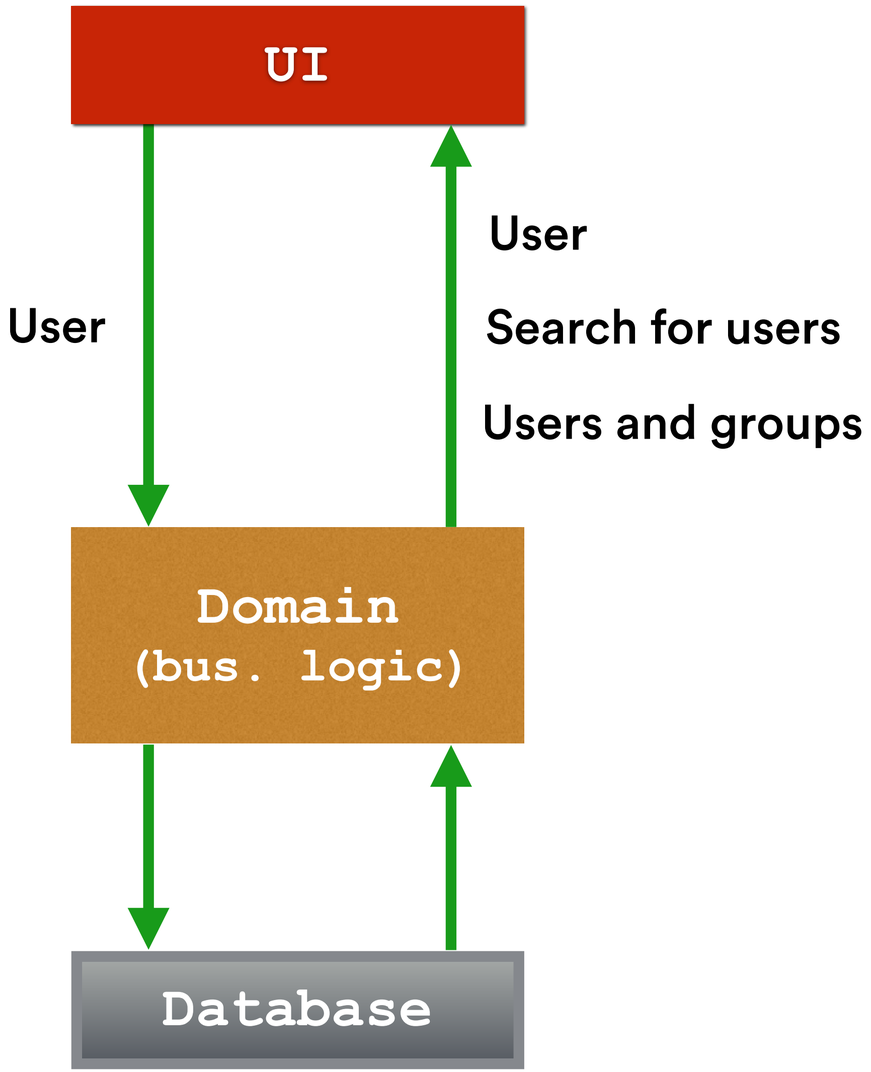

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

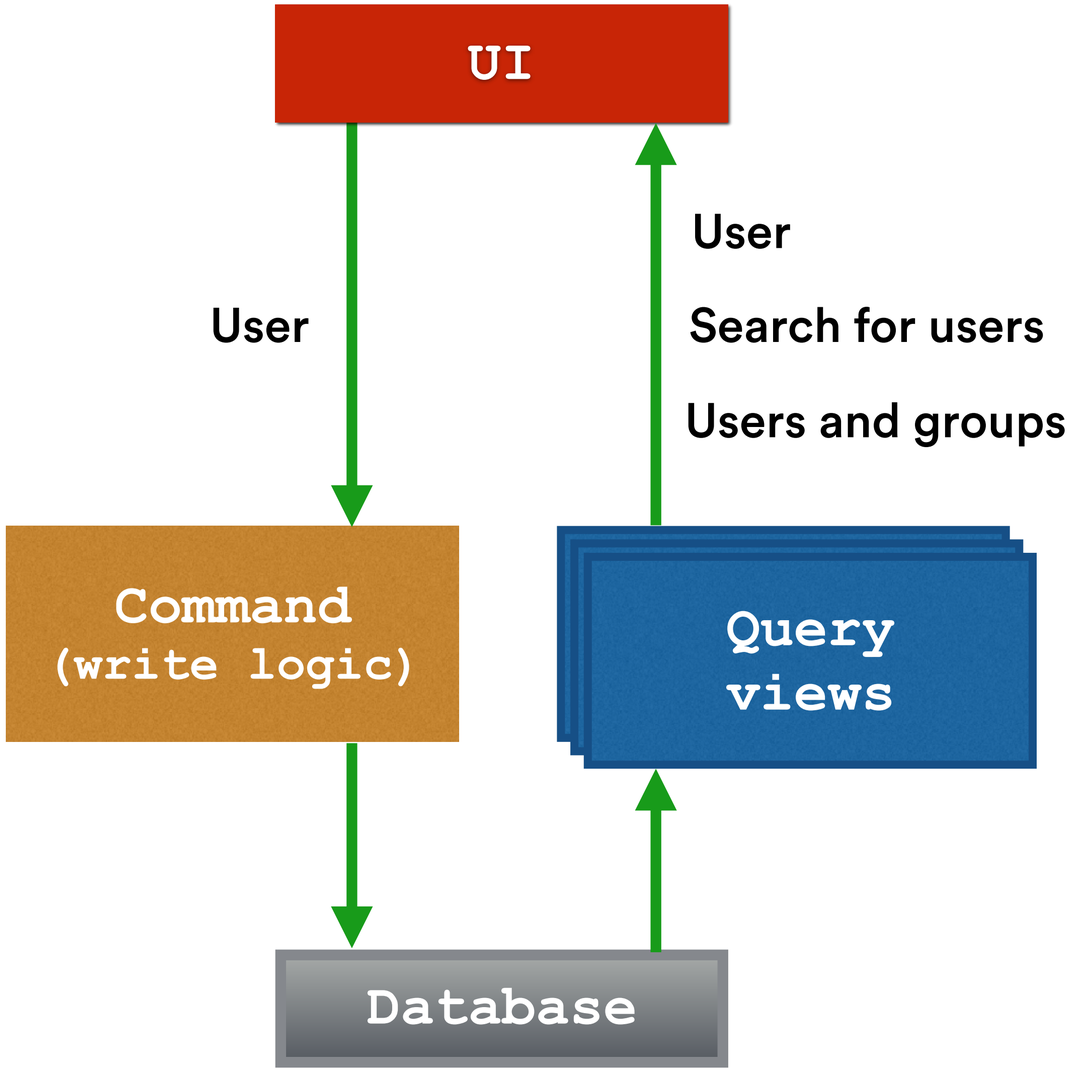

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

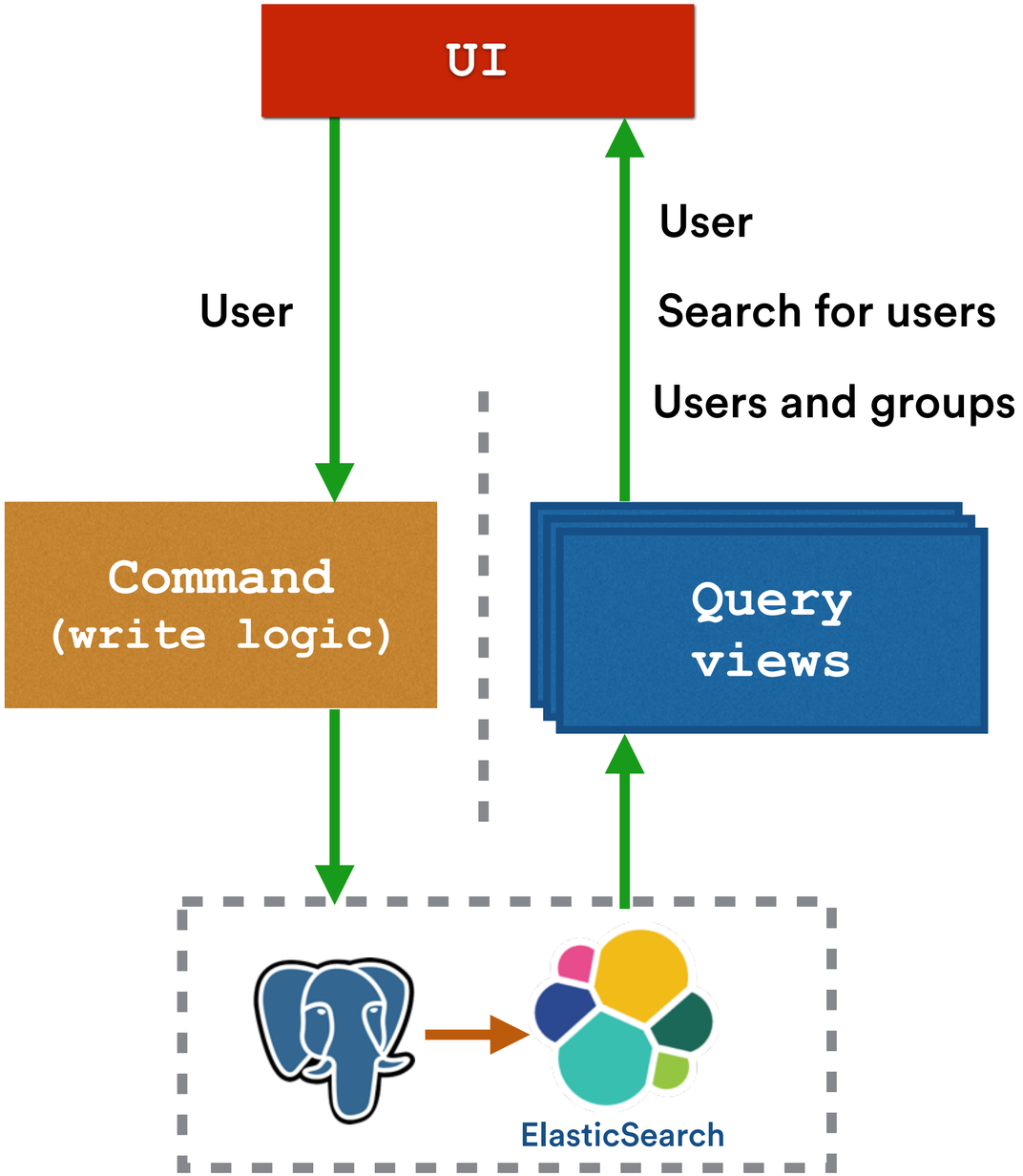
Command Query Responsibility Segregation

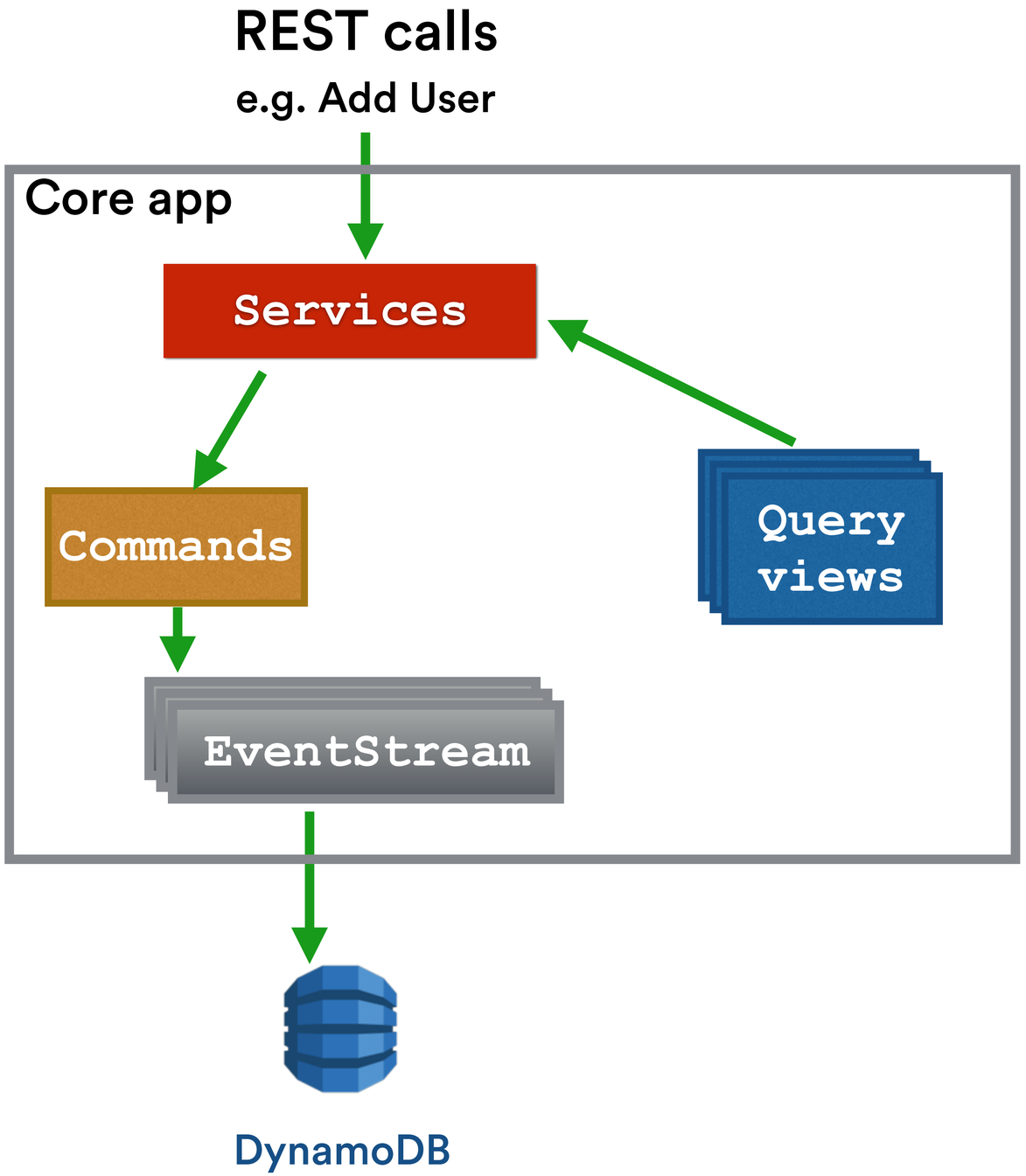

UI
User
User
Domain
(bus. logic)
Database

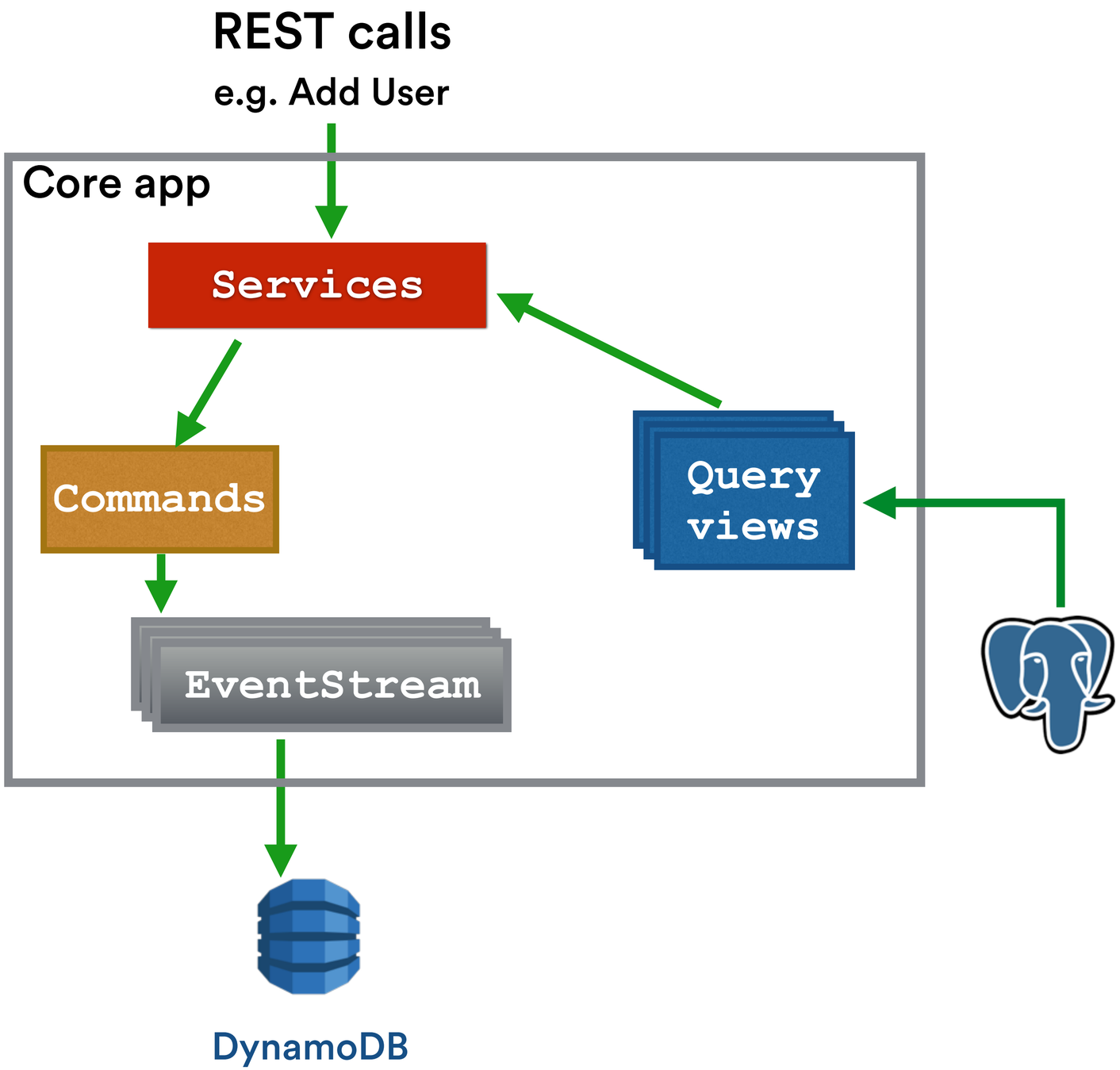

UI
User
User
Search for users
Domain
(bus. logic)
Database

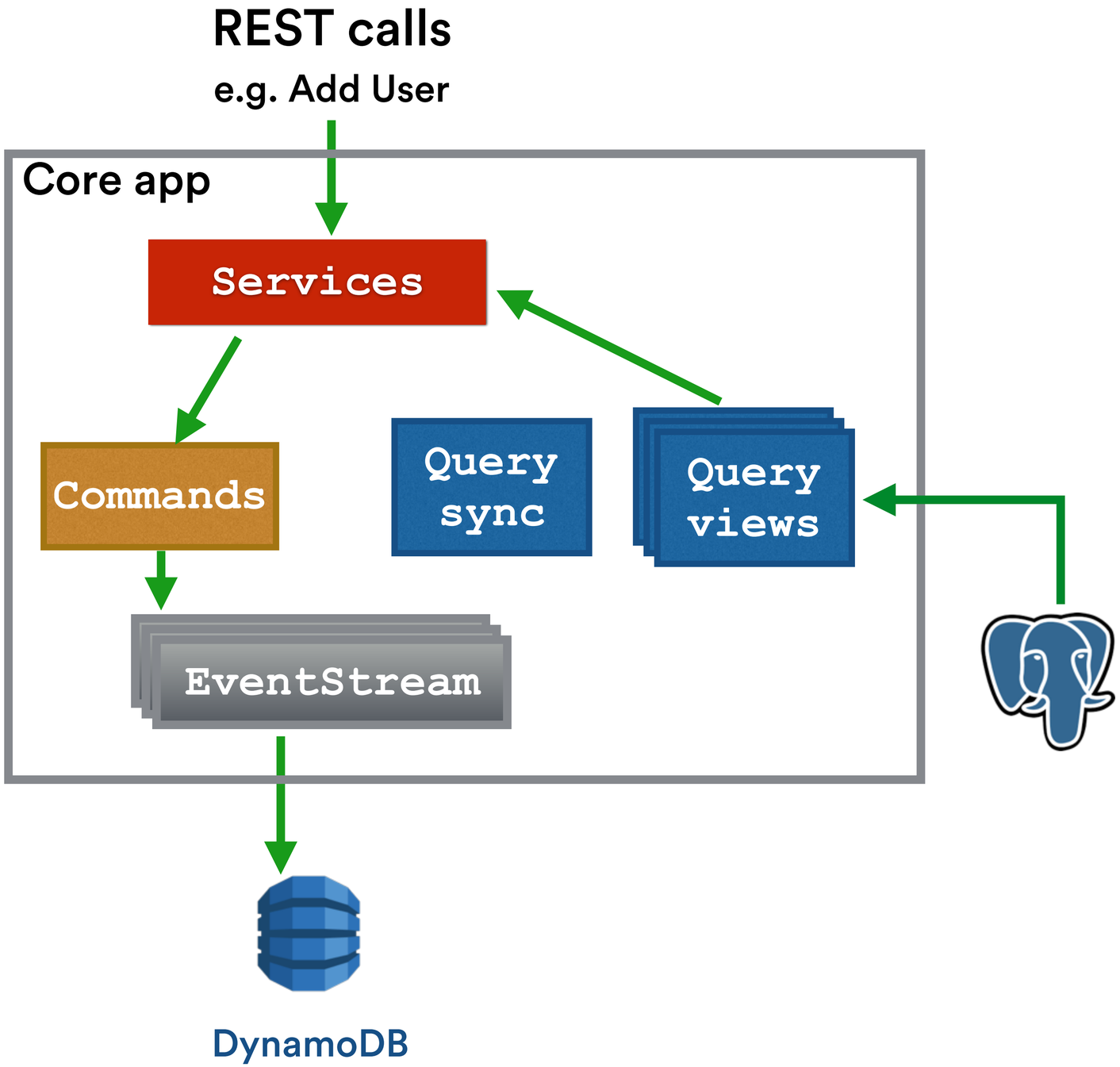

UI
User
User
Search for users
Users and groups
Domain
(bus. logic)
Database

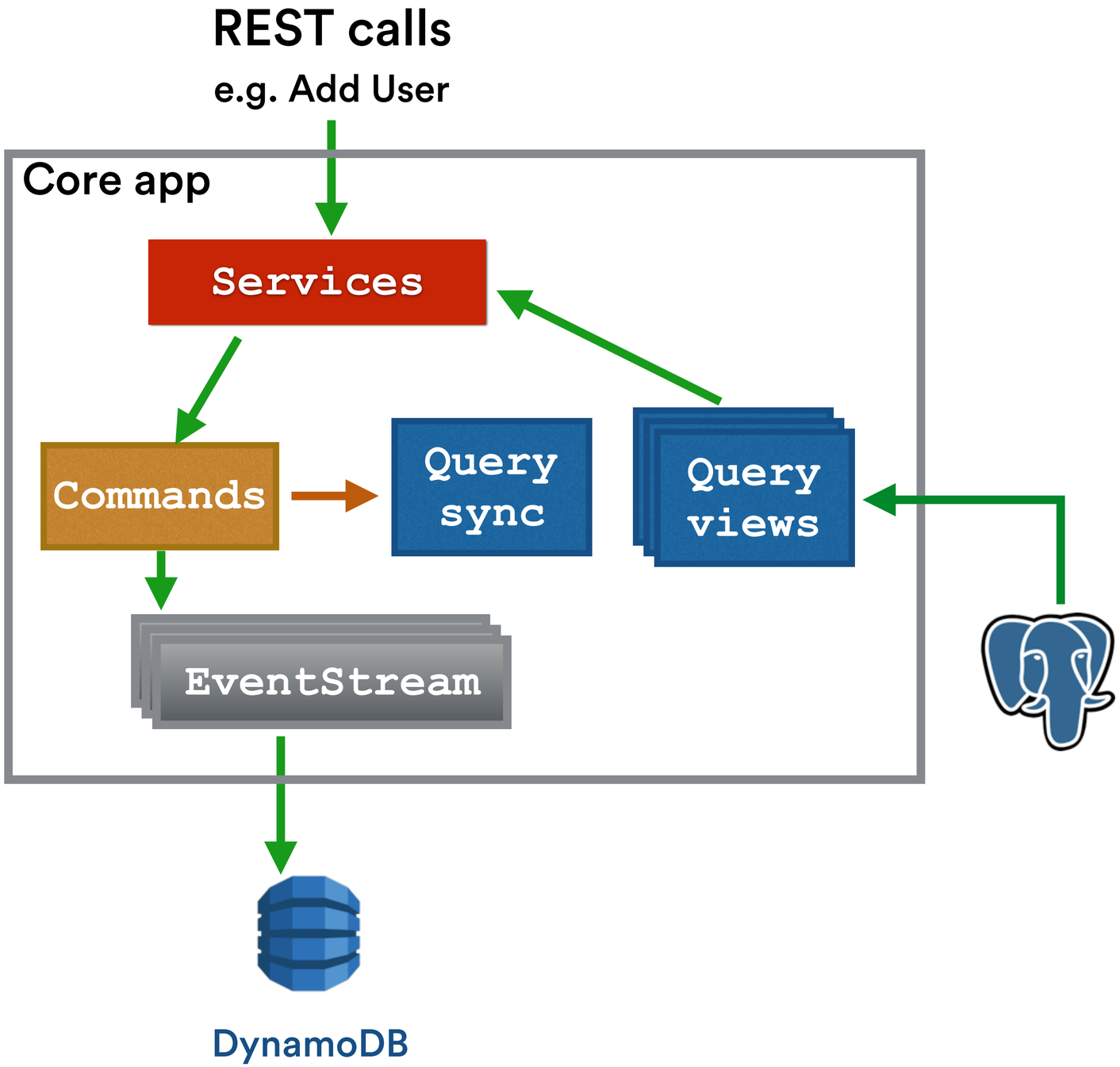

UI
User
User
Search for users
Users and groups
Command
(write logic)
Query
views
Database

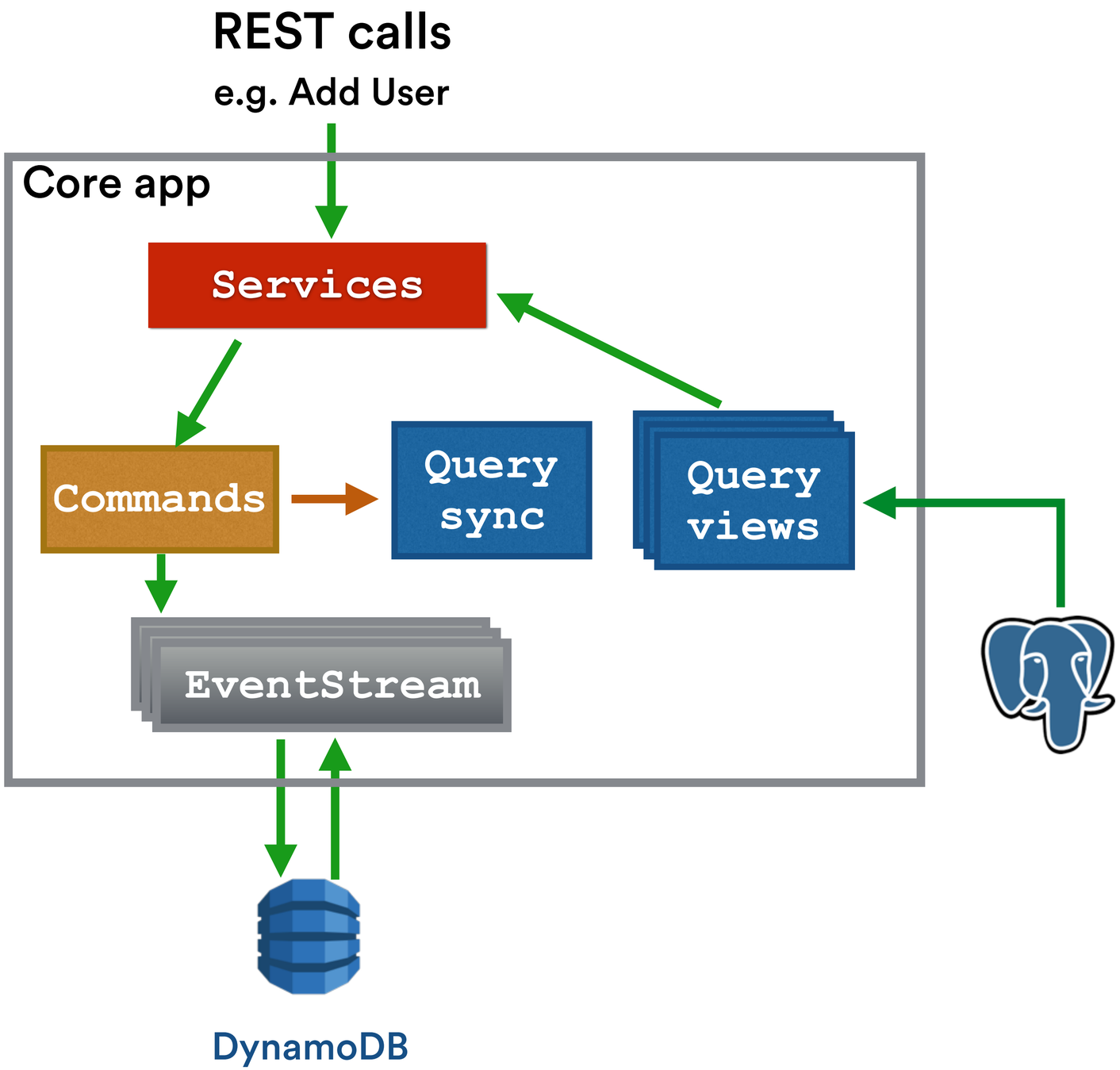

UI
User
User
Search for users
Users and groups
Command
(write logic)
Query
views
ElasticSearch

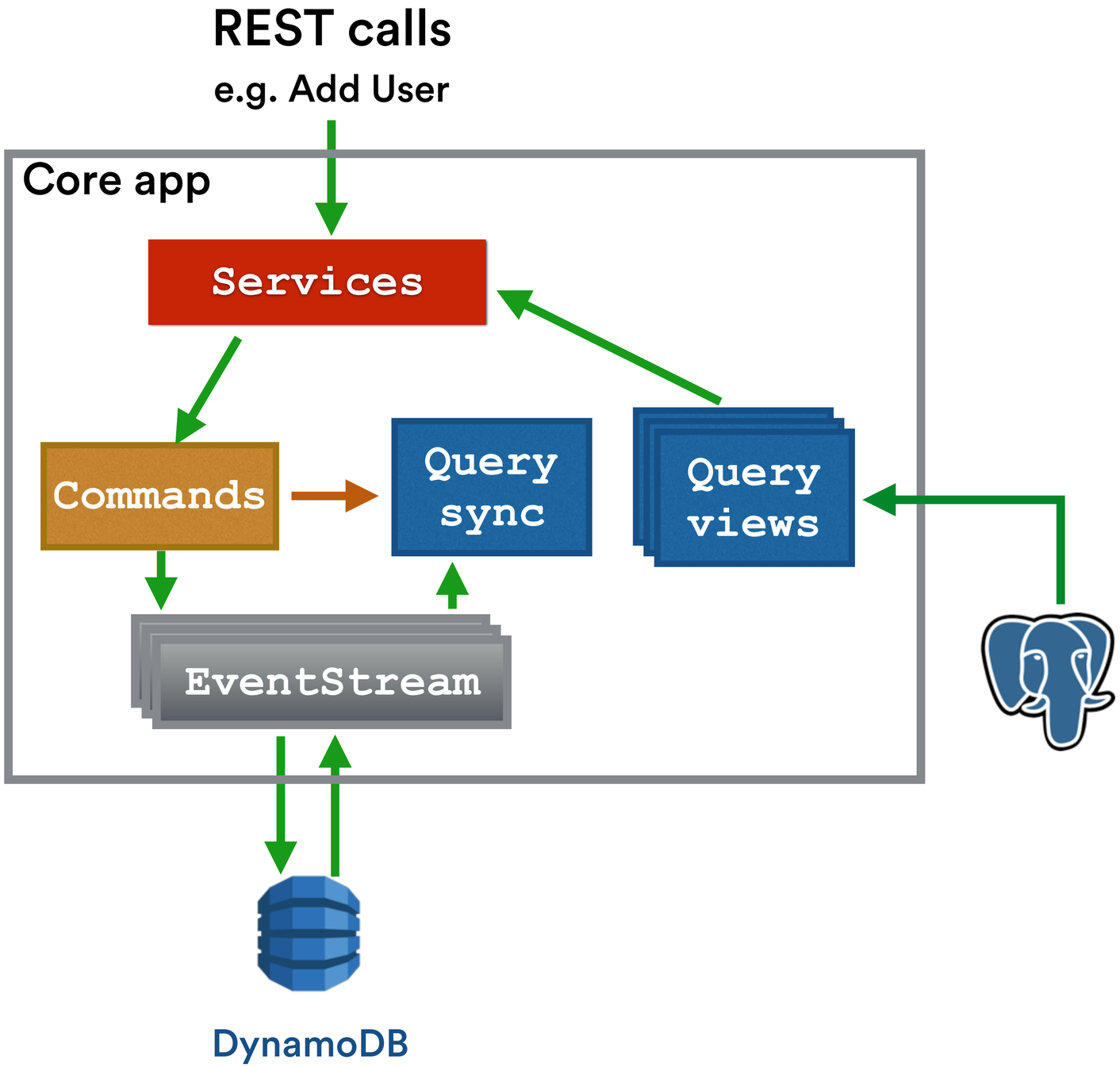

REST calls
e.g. Add User
Core app
Services
Commands
Query views
EventStream
DynamoDB

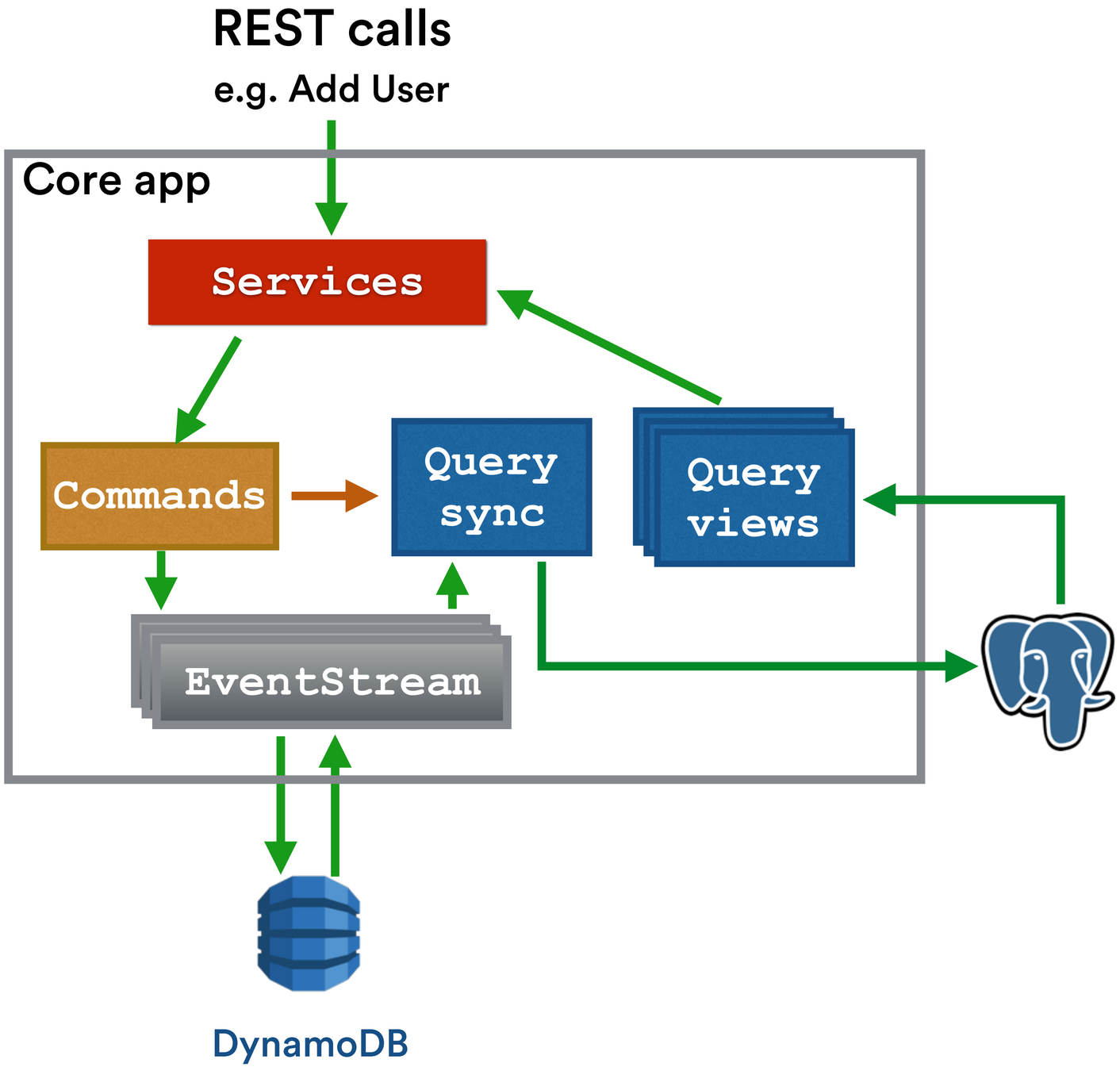

REST calls
e.g. Add User
Core app
Services
Commands
EventStream
Query views
DynamoDB

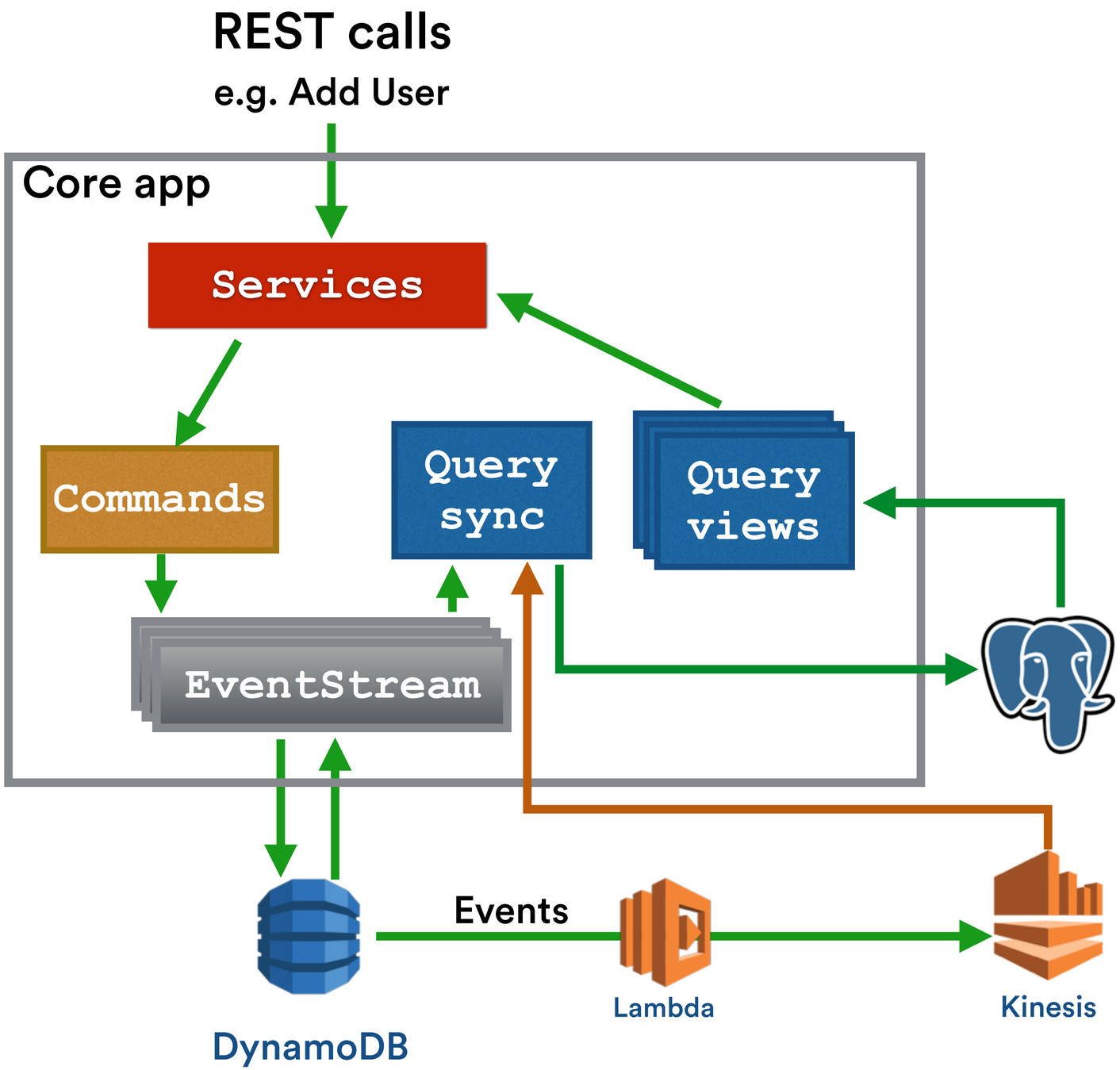

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

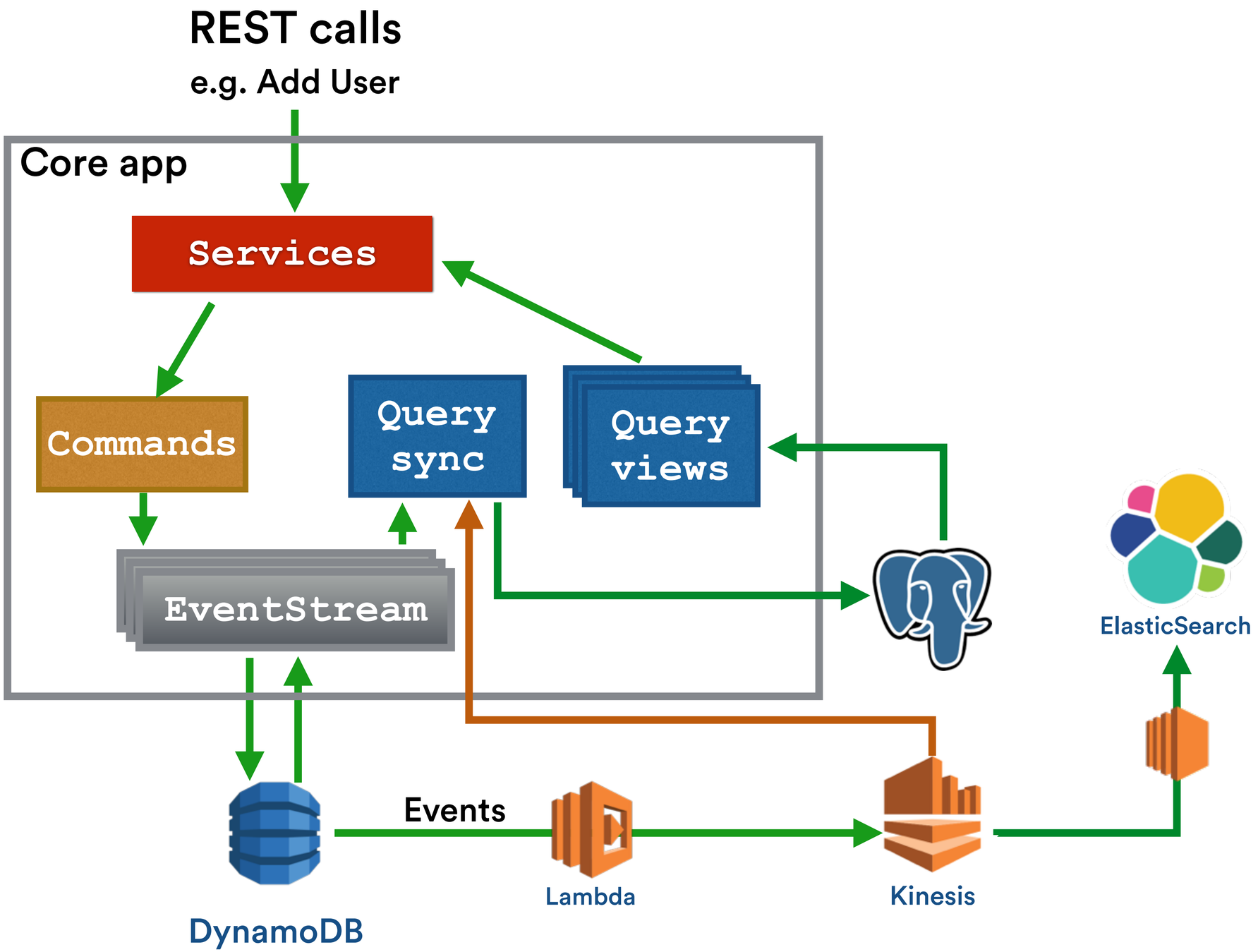

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

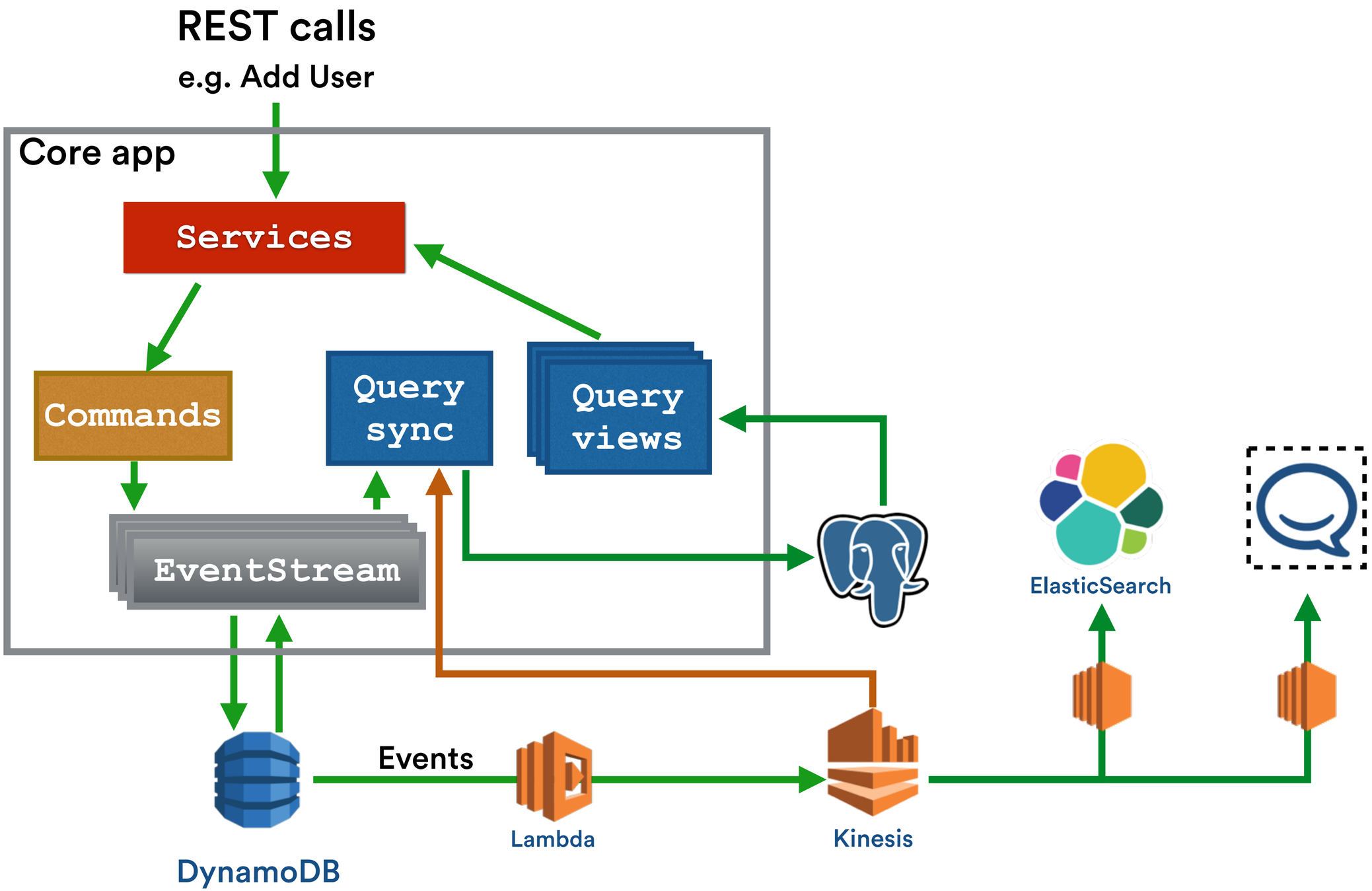

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

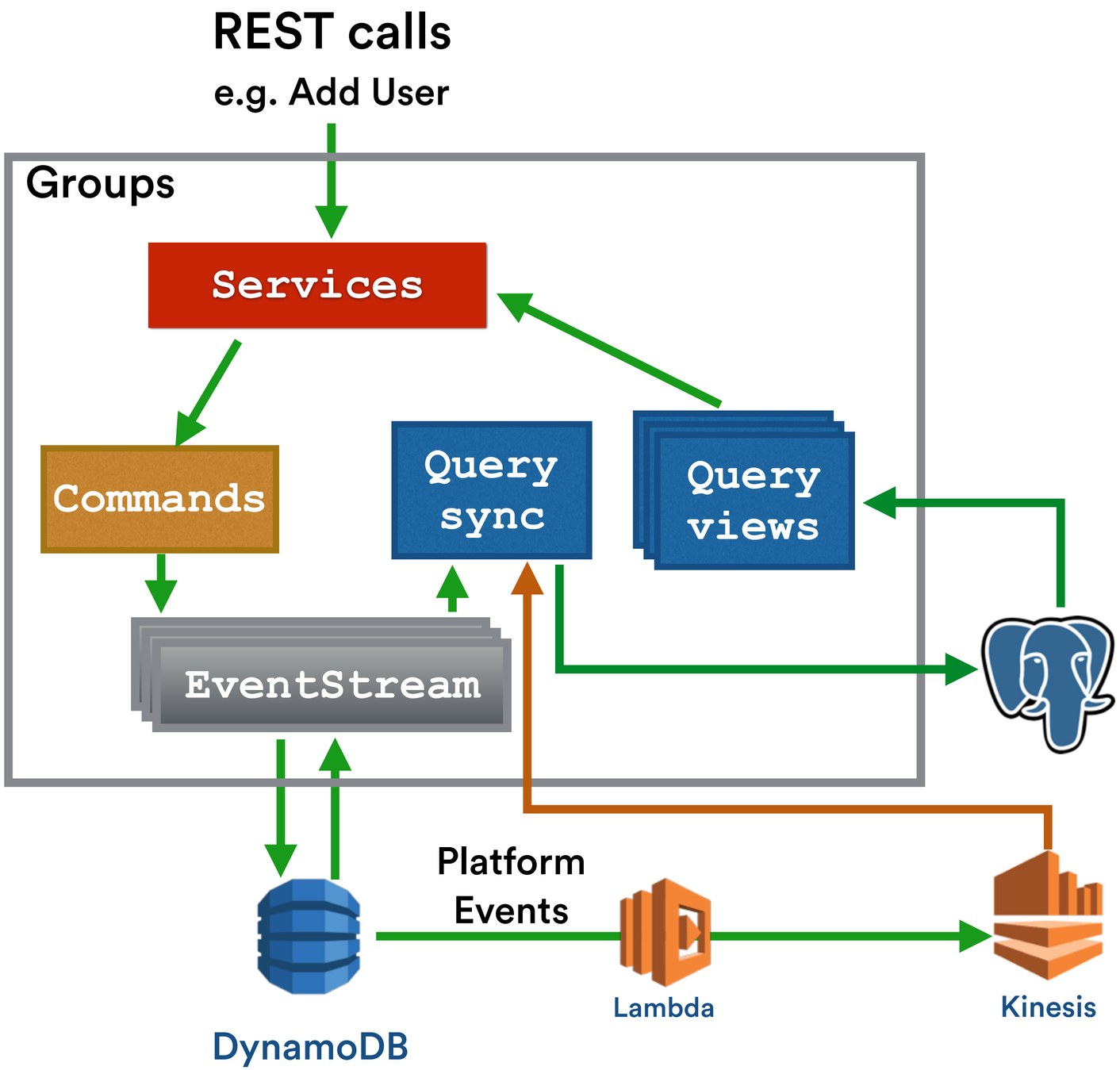

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

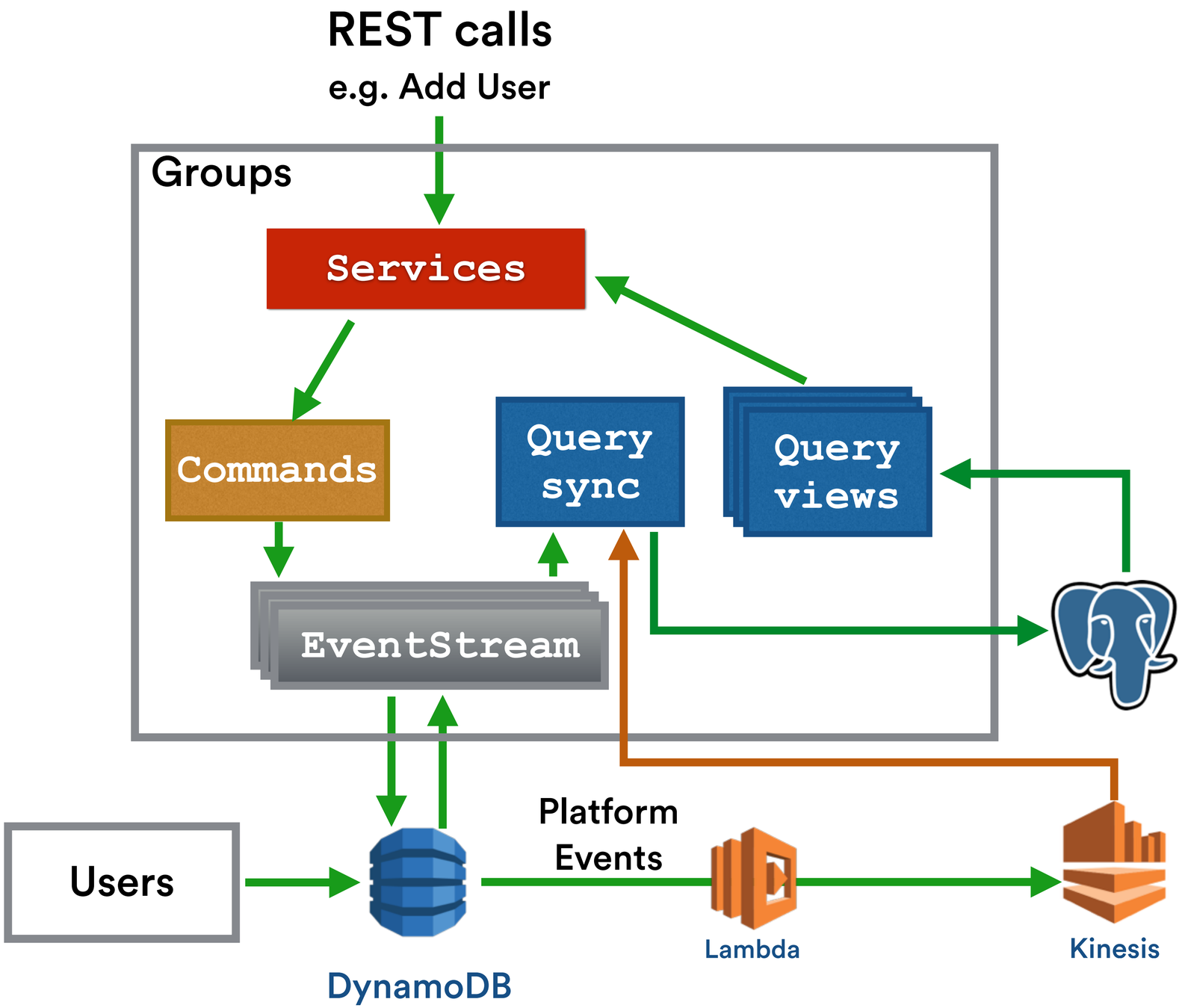

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

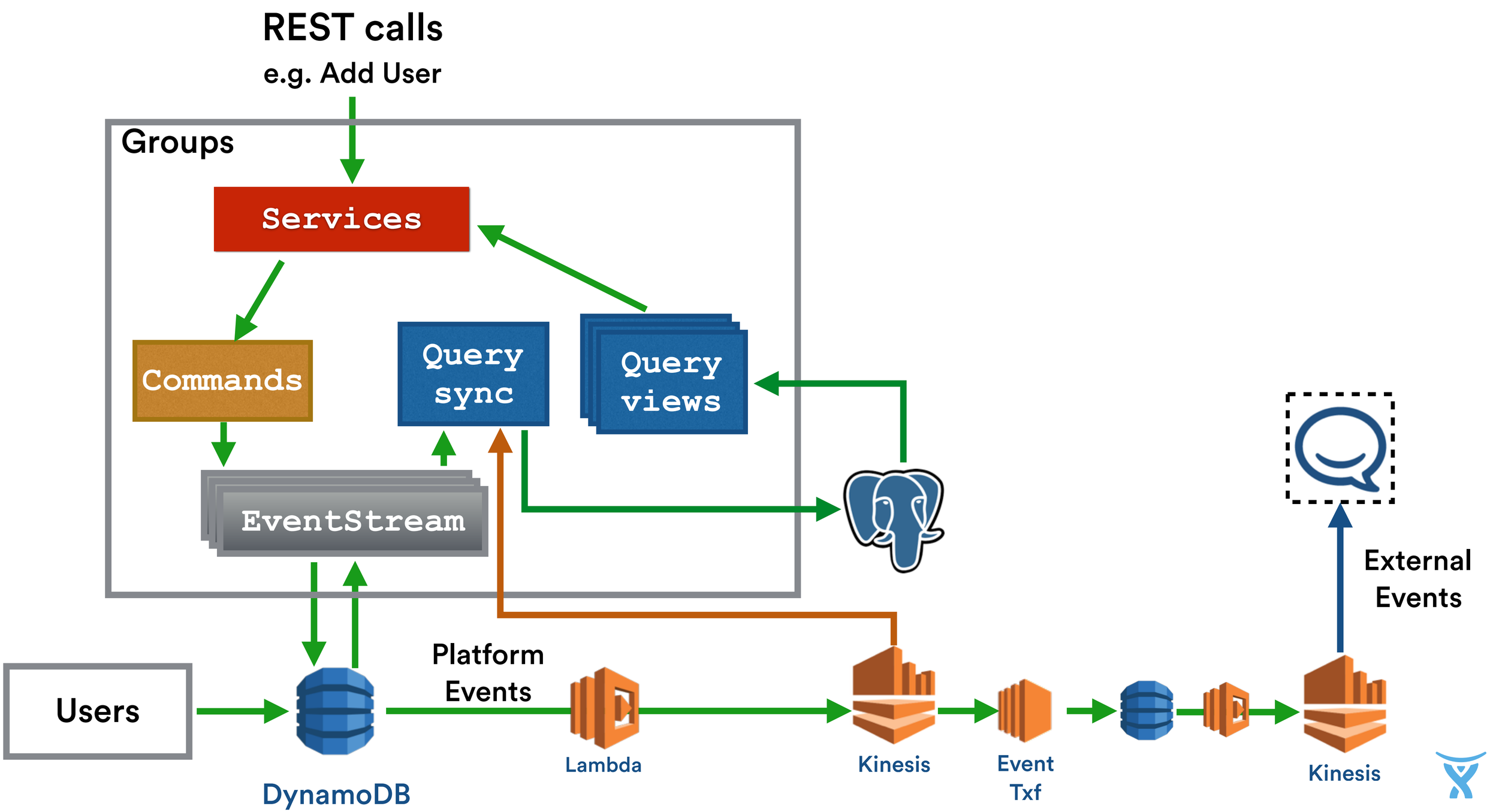

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
Events
Lambda
Kinesis
DynamoDB


REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
ElasticSearch
Events
Lambda
Kinesis
DynamoDB


REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
ElasticSearch
Events
Lambda
Kinesis
DynamoDB


REST calls
e.g. Add User
Groups
Services
Commands
Query sync
Query views
EventStream
Platform Events
Lambda
Kinesis
DynamoDB


REST calls
e.g. Add User
Groups
Services
Commands
Query sync
Query views
EventStream
Users
Platform Events
Lambda
Kinesis
DynamoDB


REST calls
e.g. Add User
Groups
Services
Commands
Query sync
Query views
EventStream
Users
DynamoDB
Platform Events
Lambda
Kinesis
Event Txf
Kinesis
External Events

Events as an API

Insert / Update Delta vs ‘Set’ events

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

UserUpdated(id, email = *Some*(email))

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

UserUpdated(id, email = *Some*(email))

SetUserName(id, name)

SetUserEmail(id, email1)

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

UserUpdated(id, email = *Some*(email))

SetUserName(id, name)

SetUserEmail(id, email1)

SetUserEmail(id, email2)

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

UserUpdated(id, email = *Some*(email))

Fits nicely with CRUD + PATCH

SetUserName(id, name)

SetUserEmail(id, email1)

SetUserEmail(id, email2)

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

UserUpdated(id, email = *Some*(email))

Fits nicely with CRUD + PATCH

Assume insert before update

SetUserName(id, name)
SetUserEmail(id, email1)

SetUserEmail(id, email2)

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

UserUpdated(id, email = *Some*(email))

Fits nicely with CRUD + PATCH

Assume insert before update

SetUserName(id, name)
SetUserEmail(id, email1)

SetUserEmail(id, email2)

Encourages idempotent processing

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

UserUpdated(id, email = *Some*(email))

Fits nicely with CRUD + PATCH

Assume insert before update

SetUserName(id, name)
SetUserEmail(id, email1)

SetUserEmail(id, email2)

Encourages idempotent processing

Single code path for query sync

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

UserUpdated(id, email = *Some*(email))

Fits nicely with CRUD + PATCH

Assume insert before update

SetUserName(id, name)
SetUserEmail(id, email1)

SetUserEmail(id, email2)

Encourages idempotent processing

Single code path for query sync

Minimally sized events to avoid conflict

Insert / Update Delta

UserAdded(id, name, email1)

UserUpdated(id, email = *Some*(email))

Fits nicely with CRUD + PATCH

Assume insert before update

vs

‘Set’ events

SetUserName(id, name)
SetUserEmail(id, email1)

SetUserEmail(id, email2)

Encourages idempotent processing

Single code path for query sync

Minimally sized events to avoid conflict

Single stream

Multiple streams

Single stream	Multiple streams
Transactions and consistent data resolution	

Single stream

Transactions and consistent data resolution

Multiple streams

Sharding for throughput

Single stream

~~Transactions and consistent data resolution~~

Multiple streams

Sharding for throughput

Better availability vs consistency compromise

Rules for splitting streams

1. Place independent events on different streams

Rules for splitting streams

1. Place independent events on different streams

2. Split streams by event type and unique Id

Rules for splitting streams

1. Place independent events on different streams
2. Split streams by event type and unique Id
3. Identify the ‘transactions’ you really need

Rules for splitting streams

1. Place independent events on different streams
2. Split streams by event type and unique Id
3. Identify the ‘transactions’ you really need
4. Use hierarchical streams to maximise number of streams

Rules for splitting streams

1. Place independent events on different streams
2. Split streams by event type and unique Id
3. Identify the ‘transactions’ you really need
4. Use hierarchical streams to maximise number of streams
5. Splitting and joining streams later is possible

But… no guaranteed order between streams

Query views get populated eventually

Query views get populated eventually

- A field should only be updated by a single event stream

Query views get populated eventually

- A field should only be updated by a single event stream
- No foreign key constraints

Query views get populated eventually

- A field should only be updated by a single event stream
- No foreign key constraints
- In general, unique or data constraints ‘enforced’ on write

Let go of transactions and consistency

Why do we need transactions?

Why do we need transactions?

- Enforce business constraints e.g. uniqueness

Why do we need transactions?

- Enforce business constraints e.g. uniqueness

- Guaranteed to see what I just wrote

Write and Read Consistency

But CAP theorem…

CAP or PACELC?

CAP or PACELC?

During a network **Partition**, choose between

Availability versus **Consistency**

CAP or PACELC?

During a network **Partition**, choose between

Availability versus **Consistency**

Else choose between

Latency versus **Consistency**

There is a middle ground...

Check-and-Set writes

Optional forced reads

Check-and-Set writes

Potentially conflicting events on same stream

Optional forced reads

Check-and-Set writes

Potentially conflicting events on same stream

1. Read at seq X
2. Run business rule
3. Stream must be at X to write

Optional forced reads

Check-and-Set writes

Potentially conflicting events on same stream

1. Read at seq X
2. Run business rule
3. Stream must be at X to write

Potential false positives

Optional forced reads

Check-and-Set writes

Potentially conflicting events on same stream

1. Read at seq X
2. Run business rule
3. Stream must be at X to write

Potential false positives

Smaller streams alleviate problems

Optional forced reads

Check-and-Set writes

Potentially conflicting events on same stream

1. Read at seq X
2. Run business rule
3. Stream must be at X to write

Potential false positives

Smaller streams alleviate problems

Optional forced reads

Query view must be at stream seq X

Check-and-Set writes

Potentially conflicting events on same stream

1. Read at seq X
2. Run business rule
3. Stream must be at X to write

Potential false positives

Smaller streams alleviate problems

Optional forced reads

Query view must be at stream seq X

Potential increased latency

Check-and-Set writes

Potentially conflicting events on same stream

1. Read at seq X
2. Run business rule
3. Stream must be at X to write

Potential false positives

Smaller streams alleviate problems

Optional forced reads

Query view must be at stream seq X

Potential increased latency

Do not use as default

Check-and-Set writes

Potentially conflicting events on same stream

1. Read at seq X
2. Run business rule
3. Stream must be at X to write

Potential false positives

Smaller streams alleviate problems

Optional forced reads

Query view must be at stream seq X

Potential increased latency

Do not use as default

Enforce timed waits

Tokens to emulate transactions and consistency

User: homer (id 4)

All Users: Seq 100

User 4: Seq 23

Tokens to emulate transactions and consistency

User: homer (id 4)

All Users: Seq 100

User 4: Seq 23

- Returned on read and write via ETag

Tokens to emulate transactions and consistency

User: homer (id 4)

All Users: Seq 100

User 4: Seq 23

- Returned on read and write via ETag
- Pass as request header for:

Tokens to emulate transactions and consistency

User: homer (id 4)

All Users: Seq 100

User 4: Seq 23

- Returned on read and write via ETag
- Pass as request header for:
  - Condition write (‘transaction’)

Tokens to emulate transactions and consistency

User: homer (id 4)

All Users: Seq 100

User 4: Seq 23

- Returned on read and write via ETag
- Pass as request header for:
  - Condition write (‘transaction’)
  - Force query view update (‘consistency’)

Tokens to emulate transactions and consistency

User: homer (id 4)

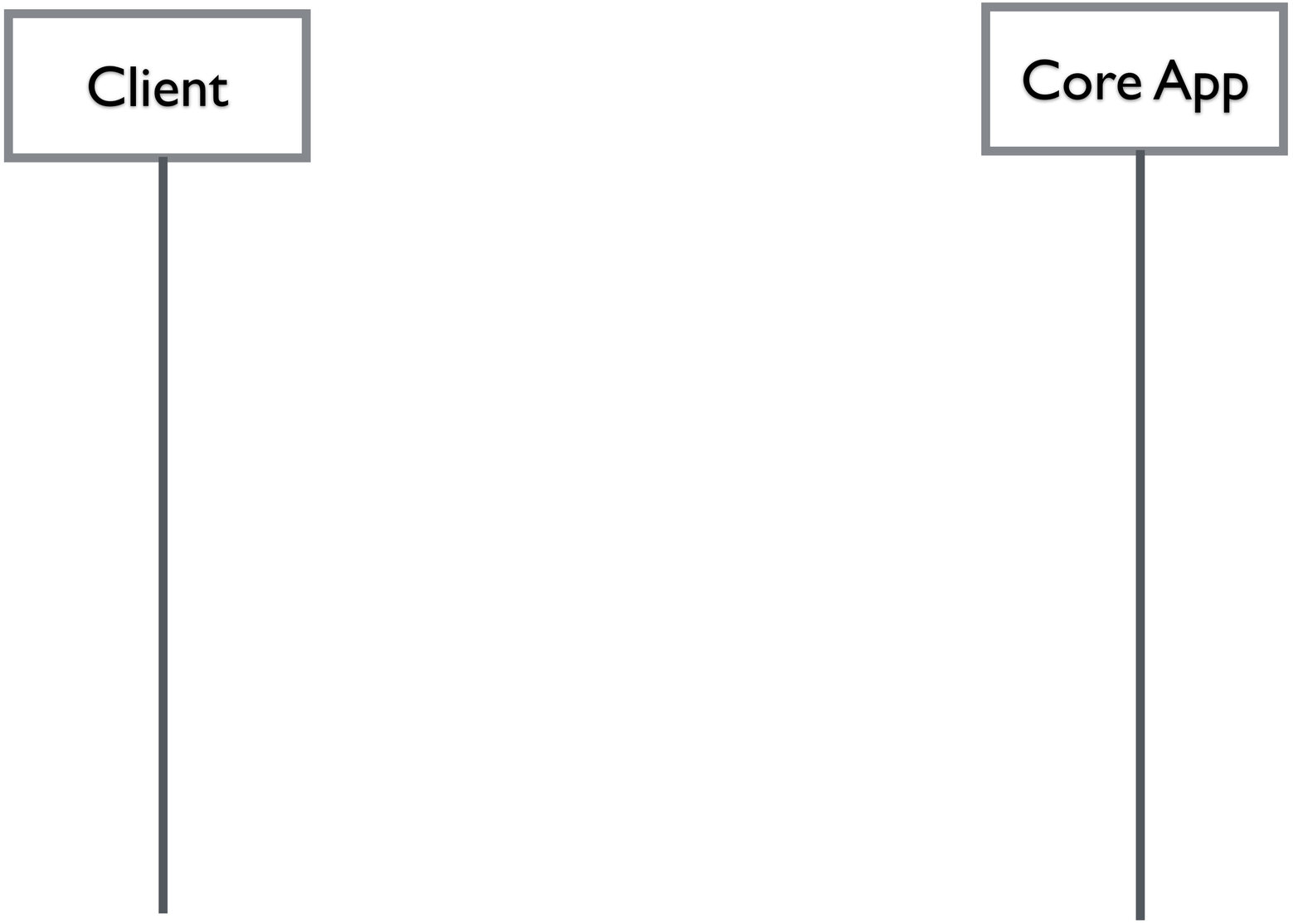
All Users: Seq 100

User 4: Seq 23

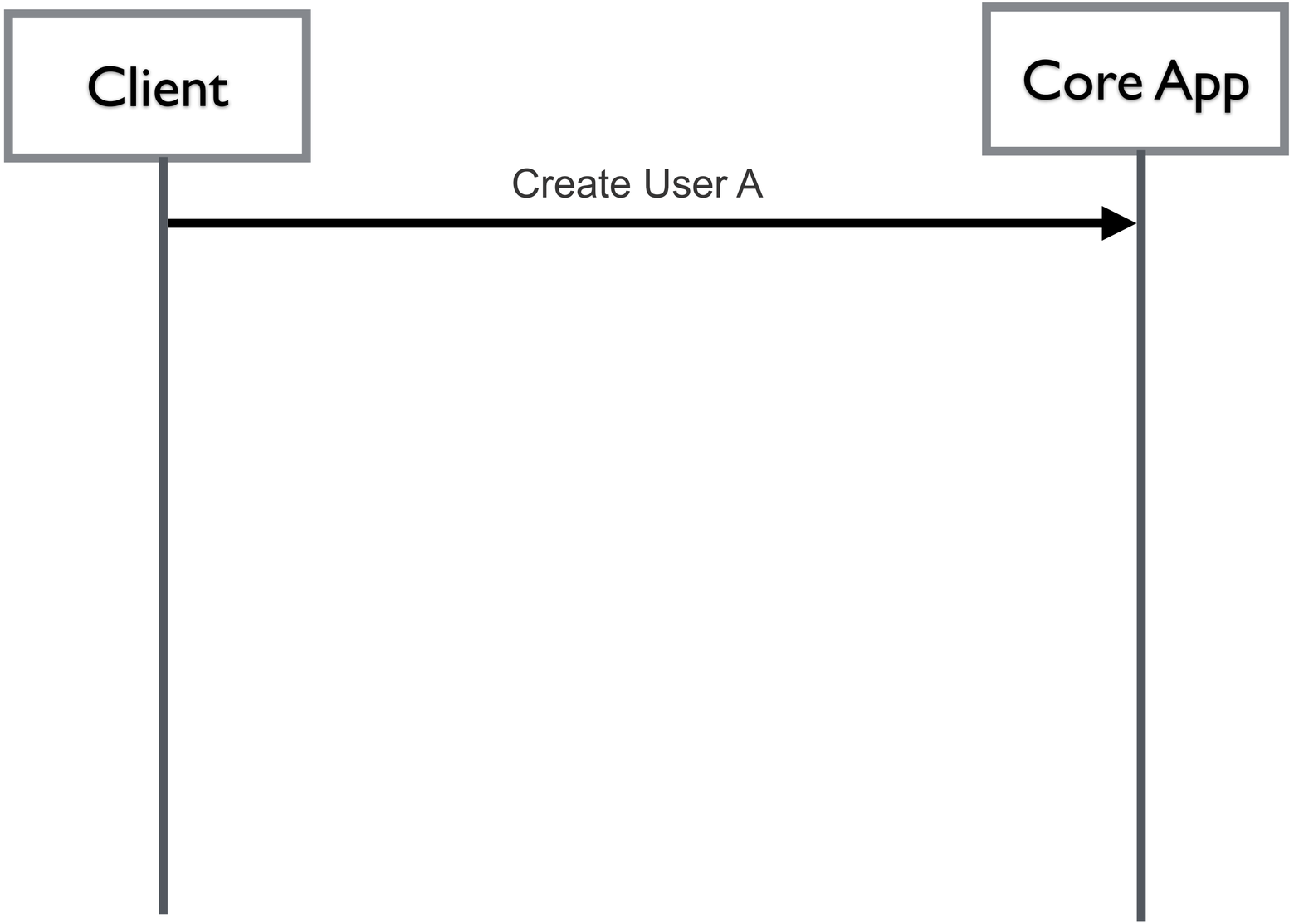
- Returned on read and write via ETag

- Pass as request header for:
  - Condition write (‘transaction’)
  - Force query view update (‘consistency’)
  - Caching

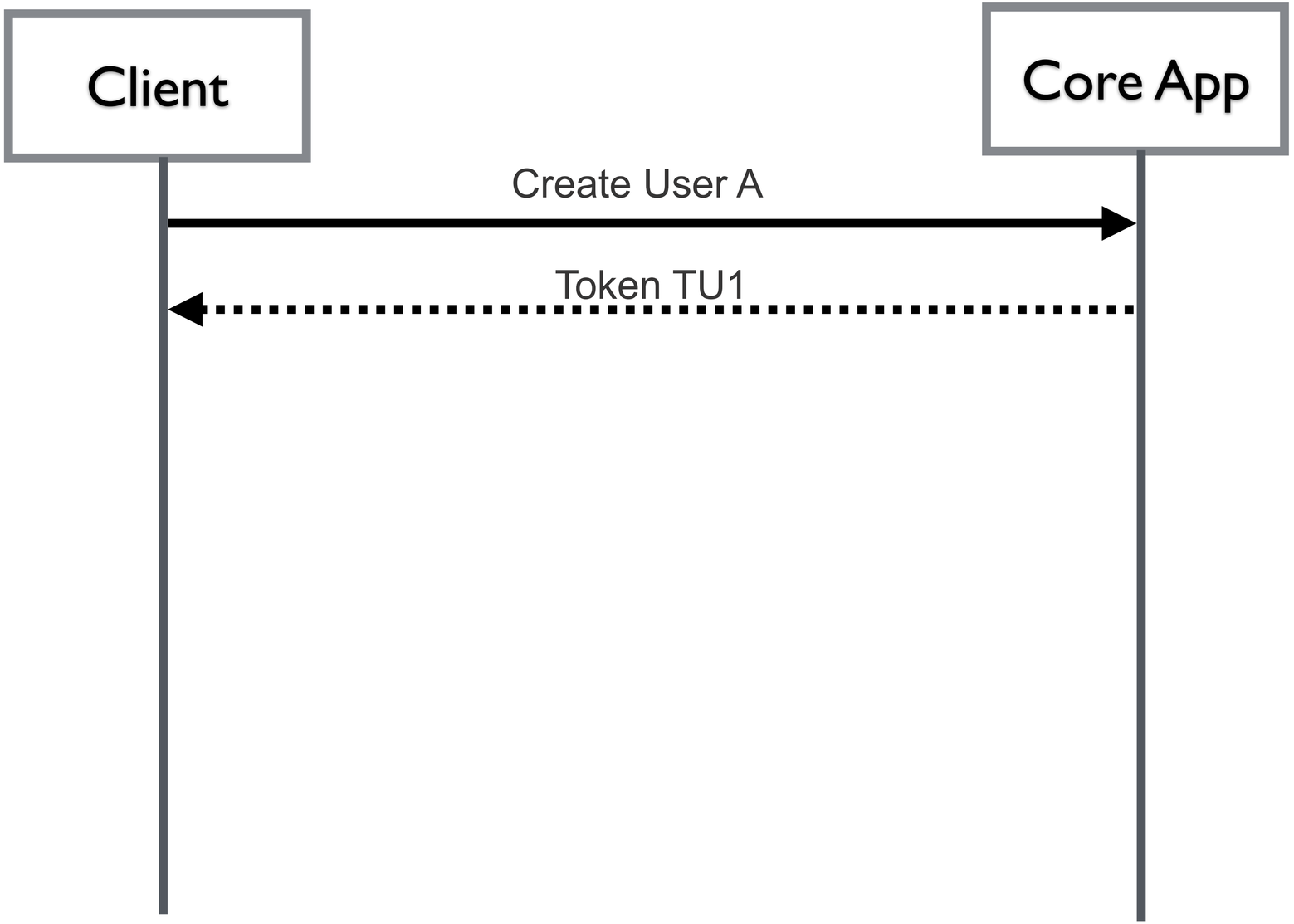
Using tokens to enforce state



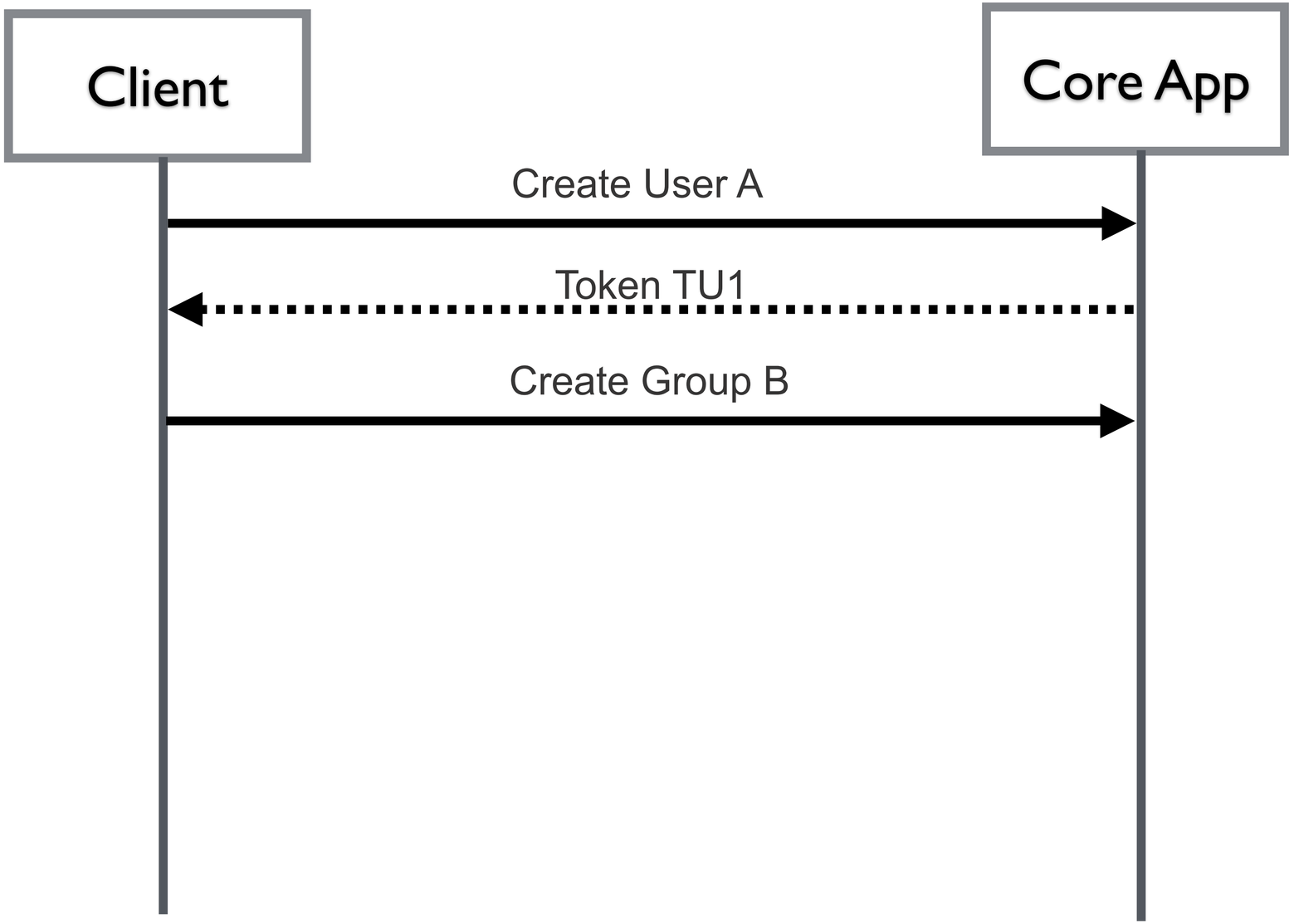
Using tokens to enforce state



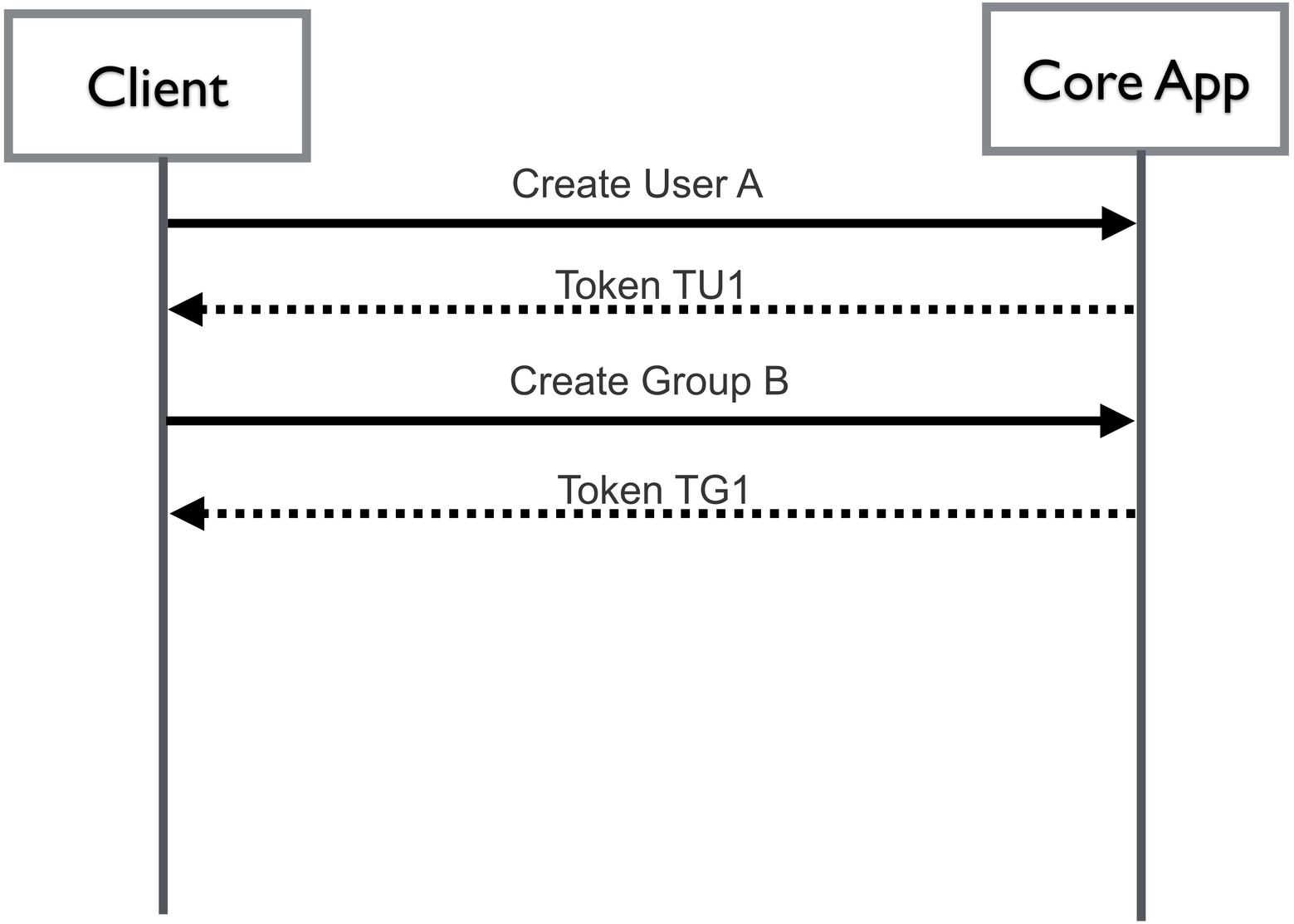
Using tokens to enforce state



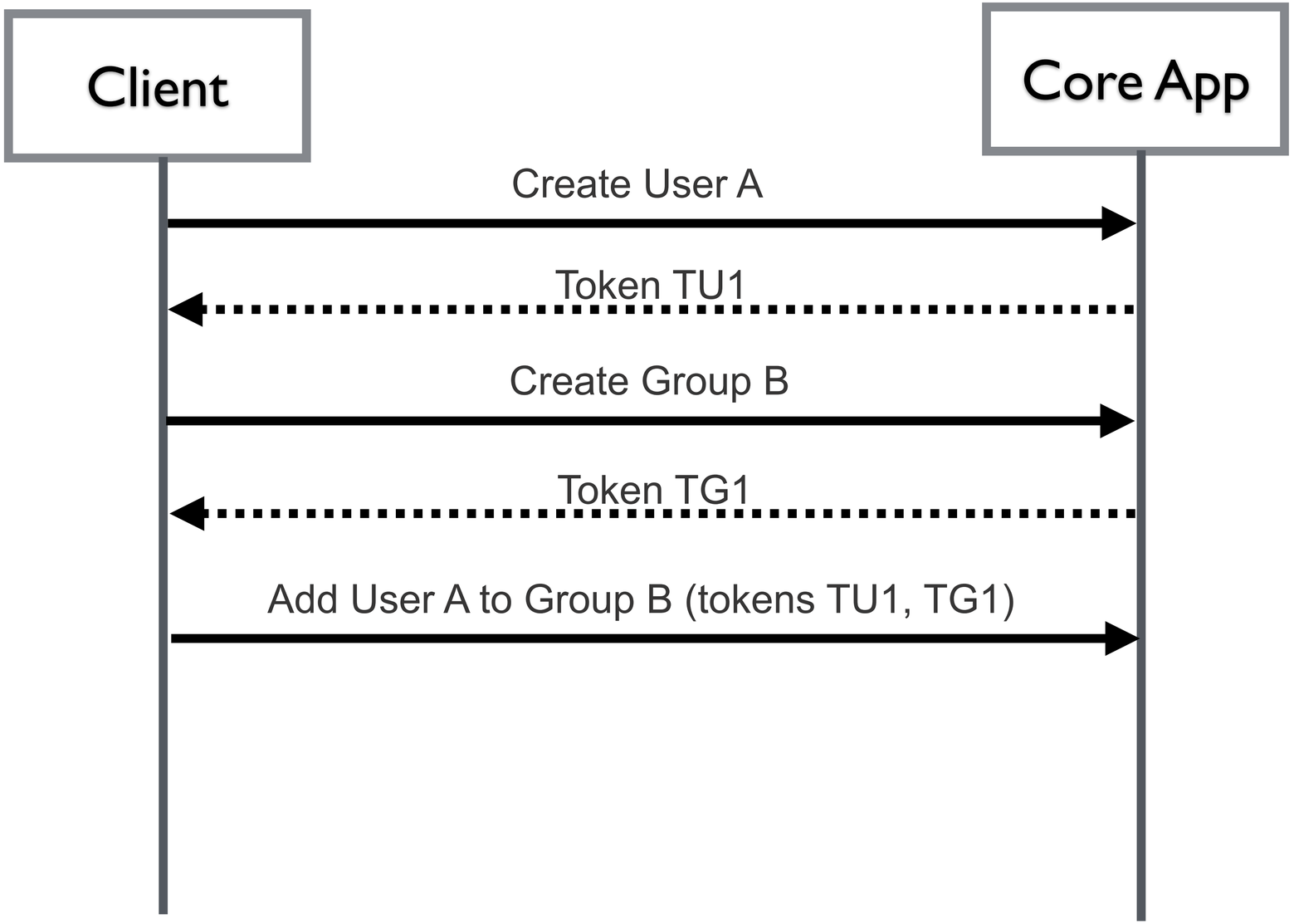
Using tokens to enforce state



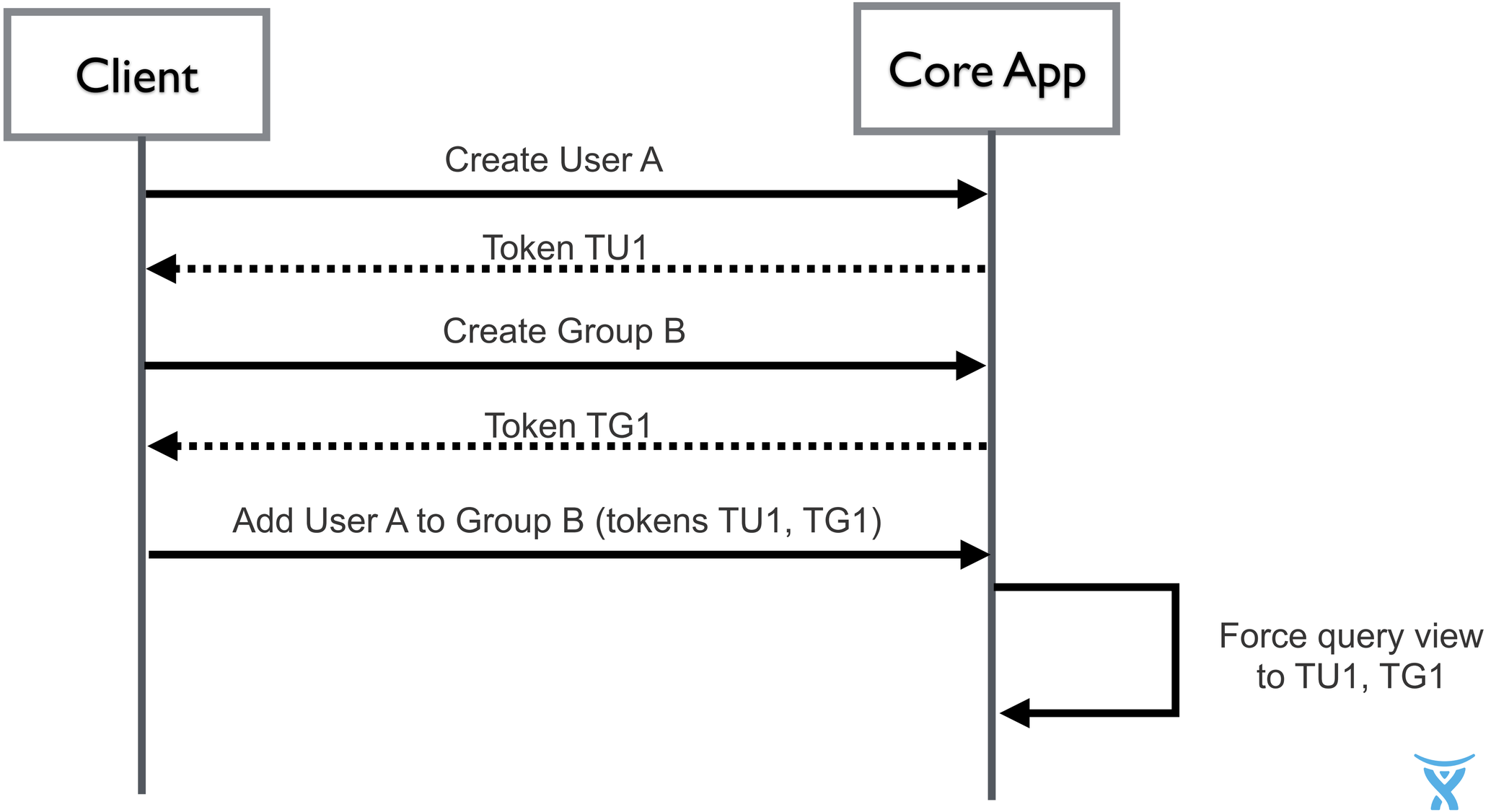
Using tokens to enforce state



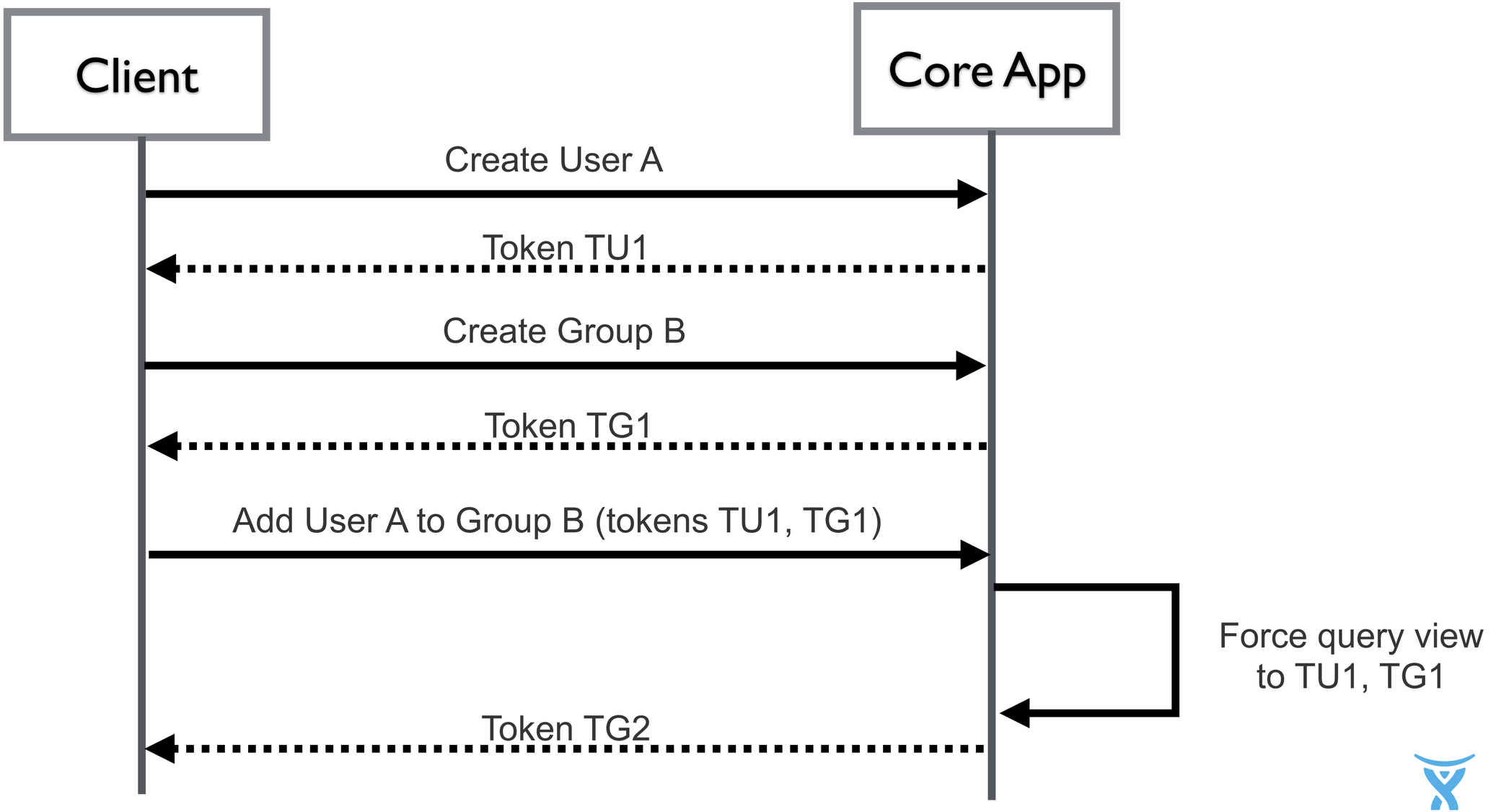
Using tokens to enforce state



Using tokens to enforce state



Using tokens to enforce state



Conflict resolution instead of transactions

Resolve conflicts on query

Resolve conflicts on query

- Conflicting events on same stream

Resolve conflicts on query

- Conflicting events on same stream

- Defined resolution algorithm on replay
  - e.g. Last Write Wins
  - Convergent/Commutative Replicated Data Types (CRDTs)

Resolve conflicts on query

- Conflicting events on same stream

- Defined resolution algorithm on replay
  - e.g. Last Write Wins
  - Convergent/Commutative Replicated Data Types (CRDTs)

No falsely failed transactions

Resolve conflicts on query

- Conflicting events on same stream
- Defined resolution algorithm on replay
  - e.g. Last Write Wins
  - Convergent/Commutative Replicated Data Types (CRDTs)

No falsely failed transactions

More resilient query views

Resolve conflicts on query

- Conflicting events on same stream
- Defined resolution algorithm on replay
  - e.g. Last Write Wins
  - Convergent/Commutative Replicated Data Types (CRDTs)

No falsely failed transactions

More resilient query views

Handles multi-region writes

Resolve conflicts on query

- Conflicting events on same stream
- Defined resolution algorithm on replay
  - e.g. Last Write Wins
  - Convergent/Commutative Replicated Data Types (CRDTs)

No falsely failed transactions

More resilient query views

Handles multi-region writes

Potential temporary glitch

Resolve conflicts on query

- Conflicting events on same stream
- Defined resolution algorithm on replay
  - e.g. Last Write Wins
  - Convergent/Commutative Replicated Data Types (CRDTs)

No falsely failed transactions

More resilient query views

Handles multi-region writes

Potential temporary glitch

Needs to be implemented on all query nodes

Summary

Key takeaways

- Start small and challenge everything!

Key takeaways

- Start small and challenge everything!

- Incremental architecture for incremental demos

Key takeaways

- Start small and challenge everything!
- Incremental architecture for incremental demos
- Think “Events as an API”

Key takeaways

- Start small and challenge everything!
- Incremental architecture for incremental demos
- Think “Events as an API”
- Accept weaker transactions and eventual consistency

“We should using event sourcing more than we do”

Martin Fowler (very loosely paraphrased)

event sourcing lib:
bitbucket.org/atlassianlabs/eventsrc