Atlassian

Immutable data stores and CQRS for safety, flexibility and profit

SIDNEY SHEK · ARCHITECT · @SIDNEYSHEK · #SCALADAYS

Our exploration

WHAT AND WHY

EVOLVING THE ARCHITECTURE

DELVING INTO CODE

3.5 DESIGN PRINCIPLES

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

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

users_new

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

events

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

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

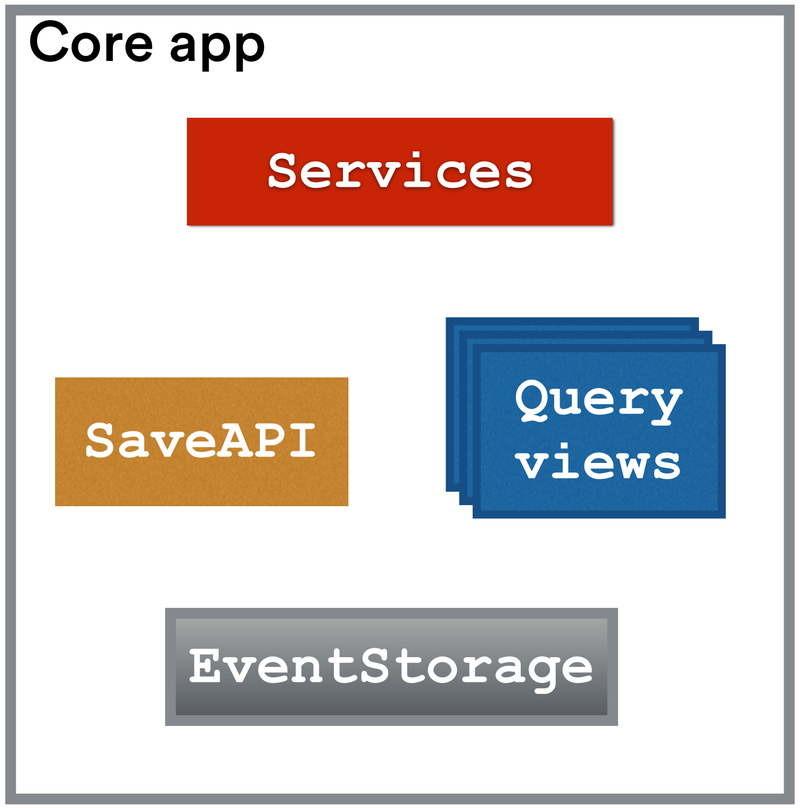
Our Identity System requirements

- 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

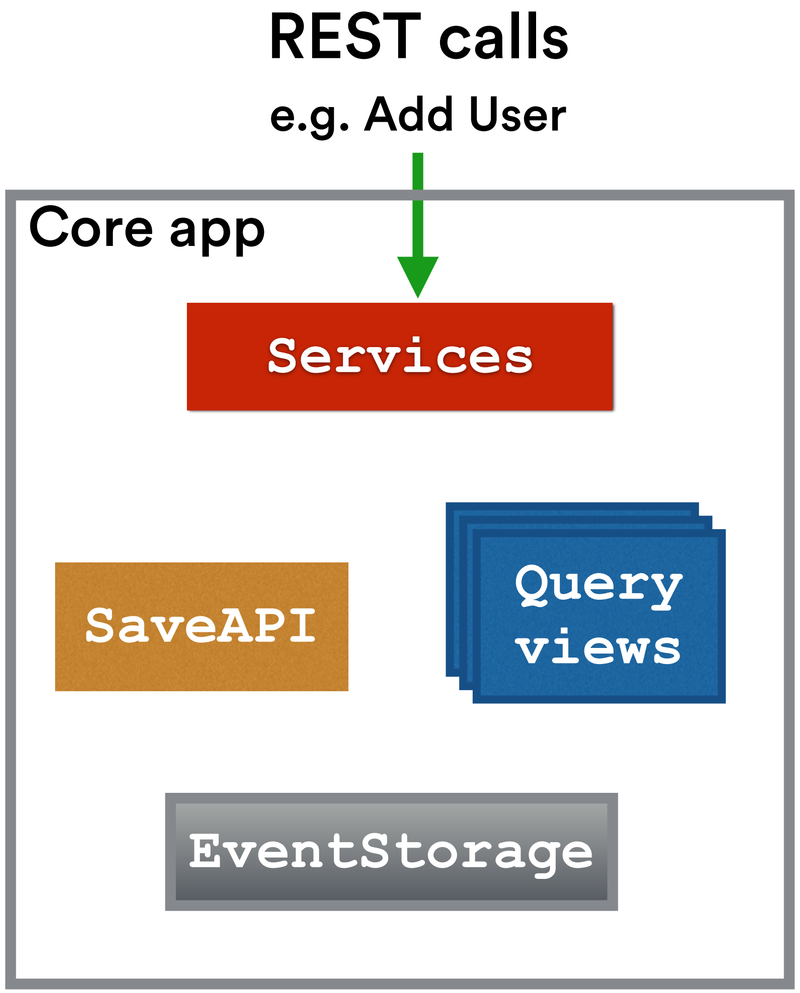
Evolving the architecture

REST calls

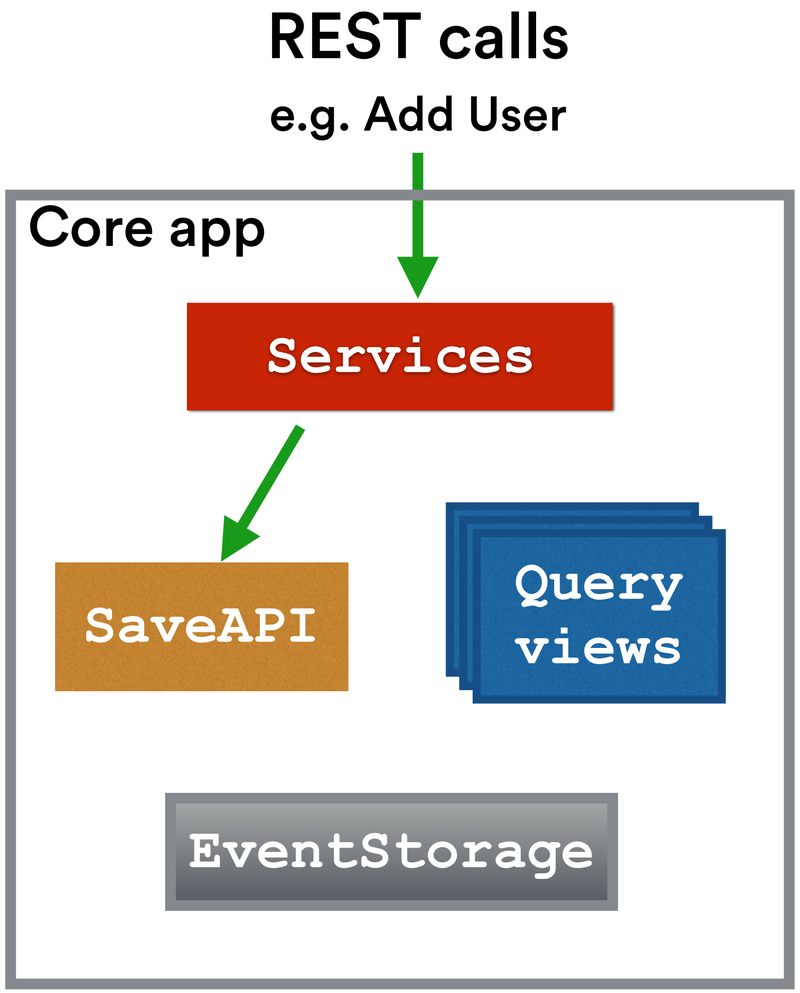
e.g. Add User



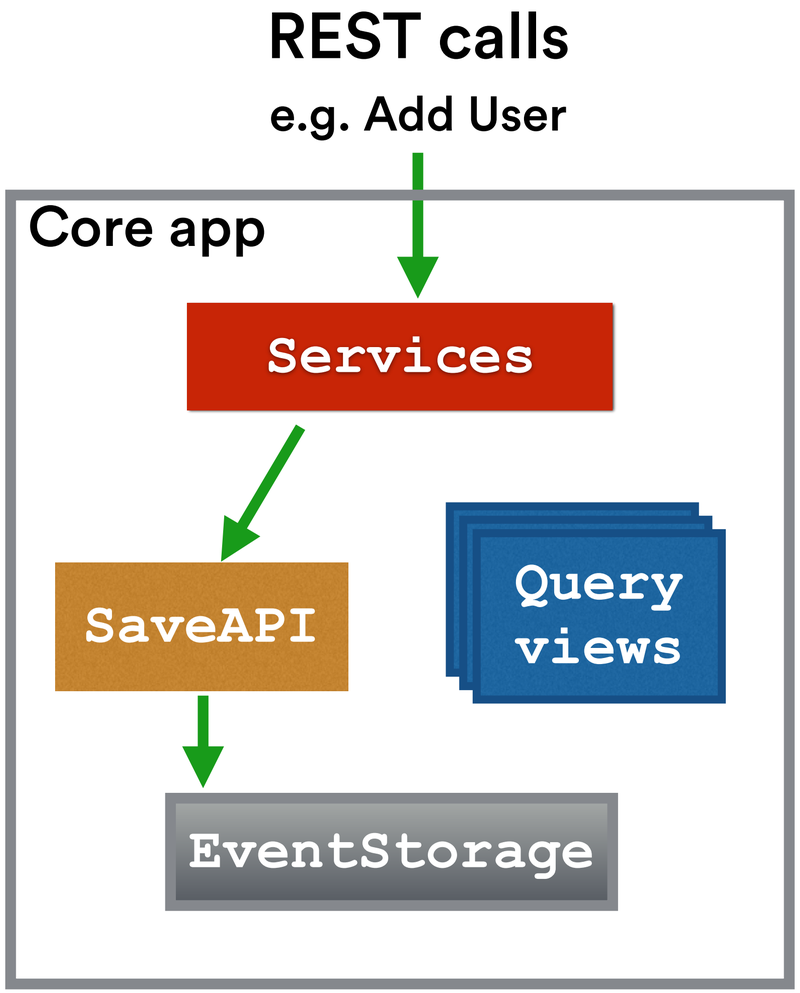
DynamoDB


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

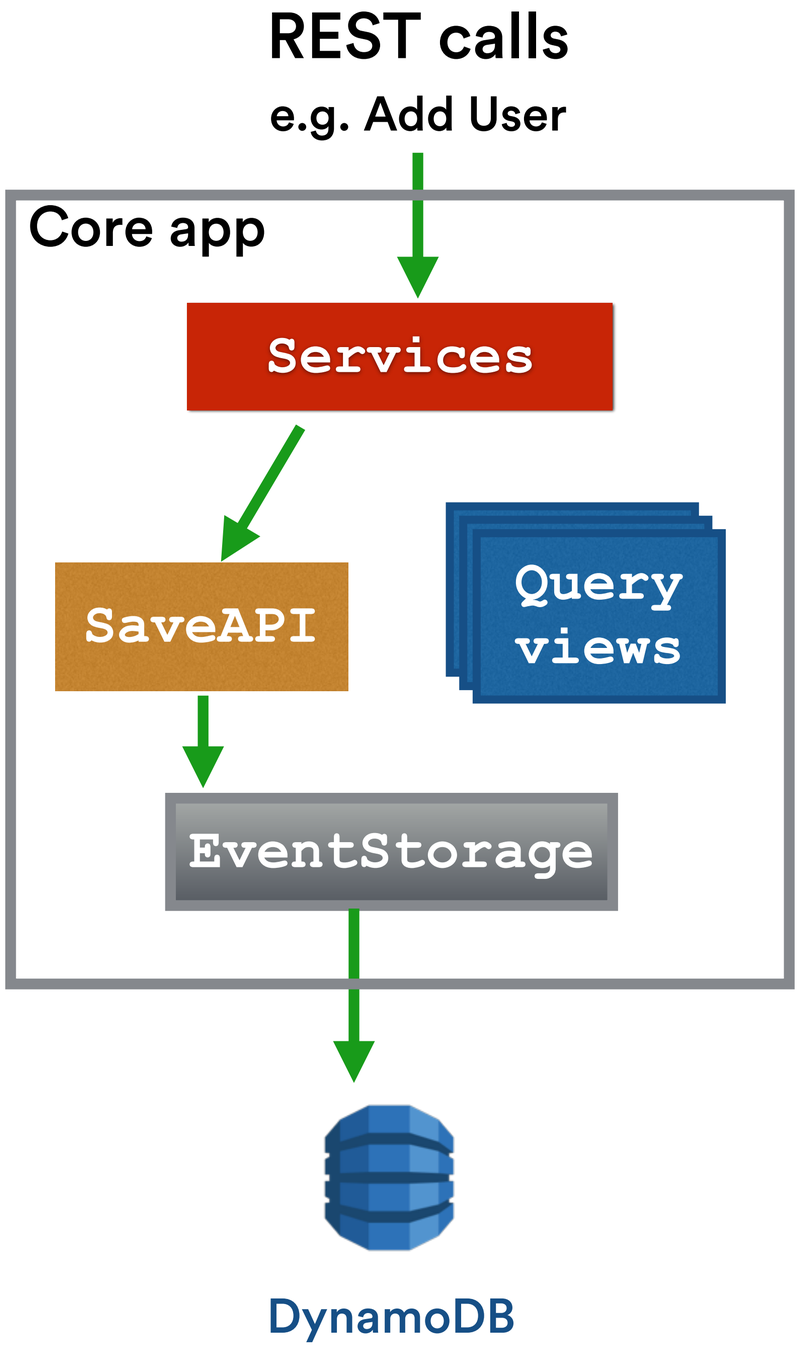
DynamoDB

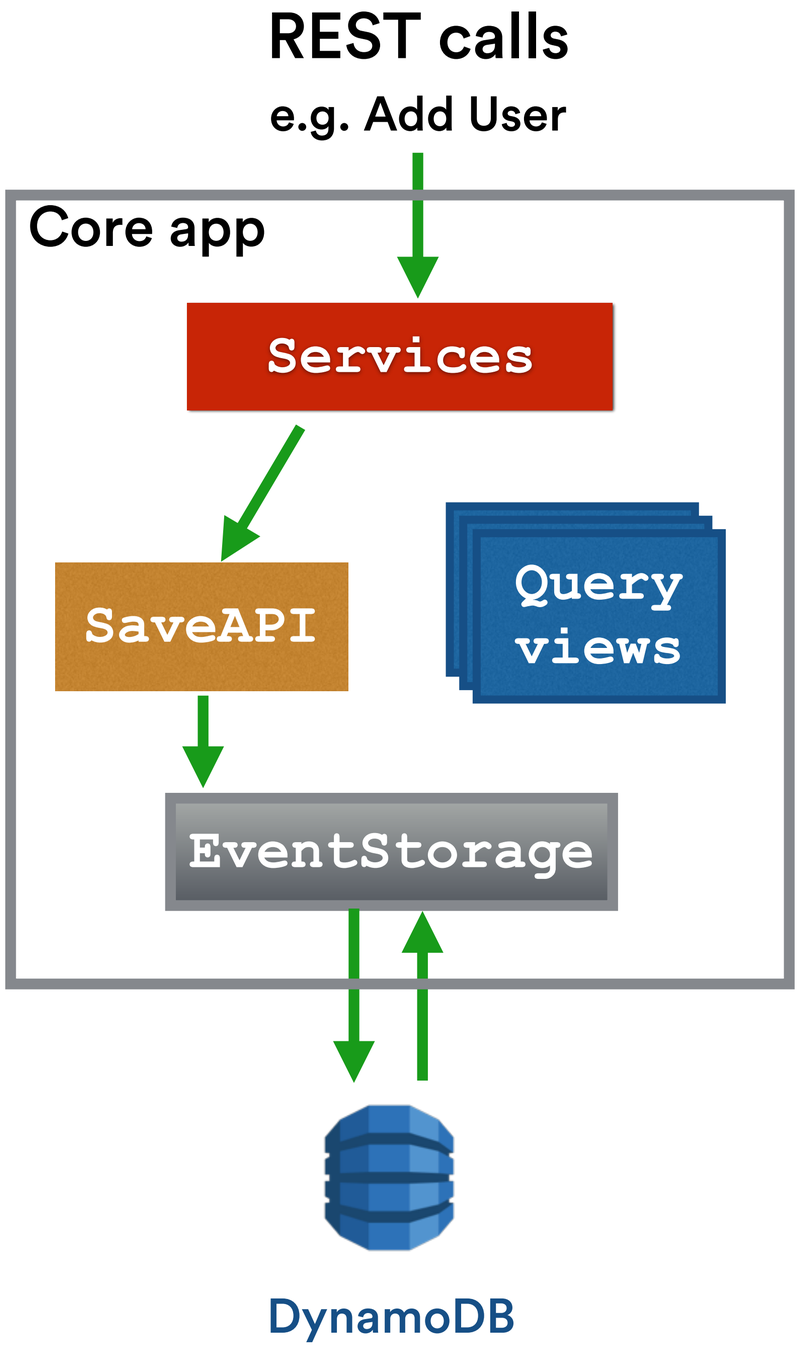

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStorage

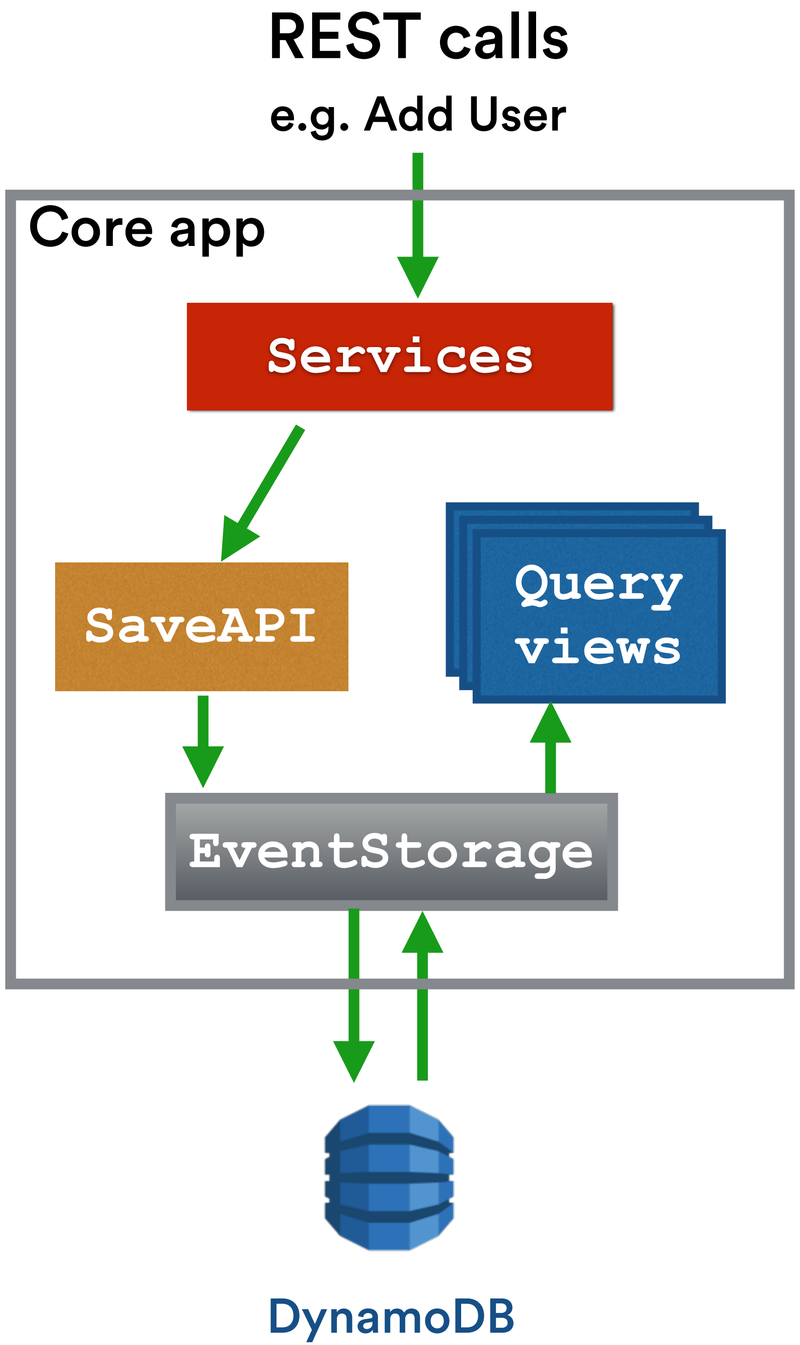
DynamoDB

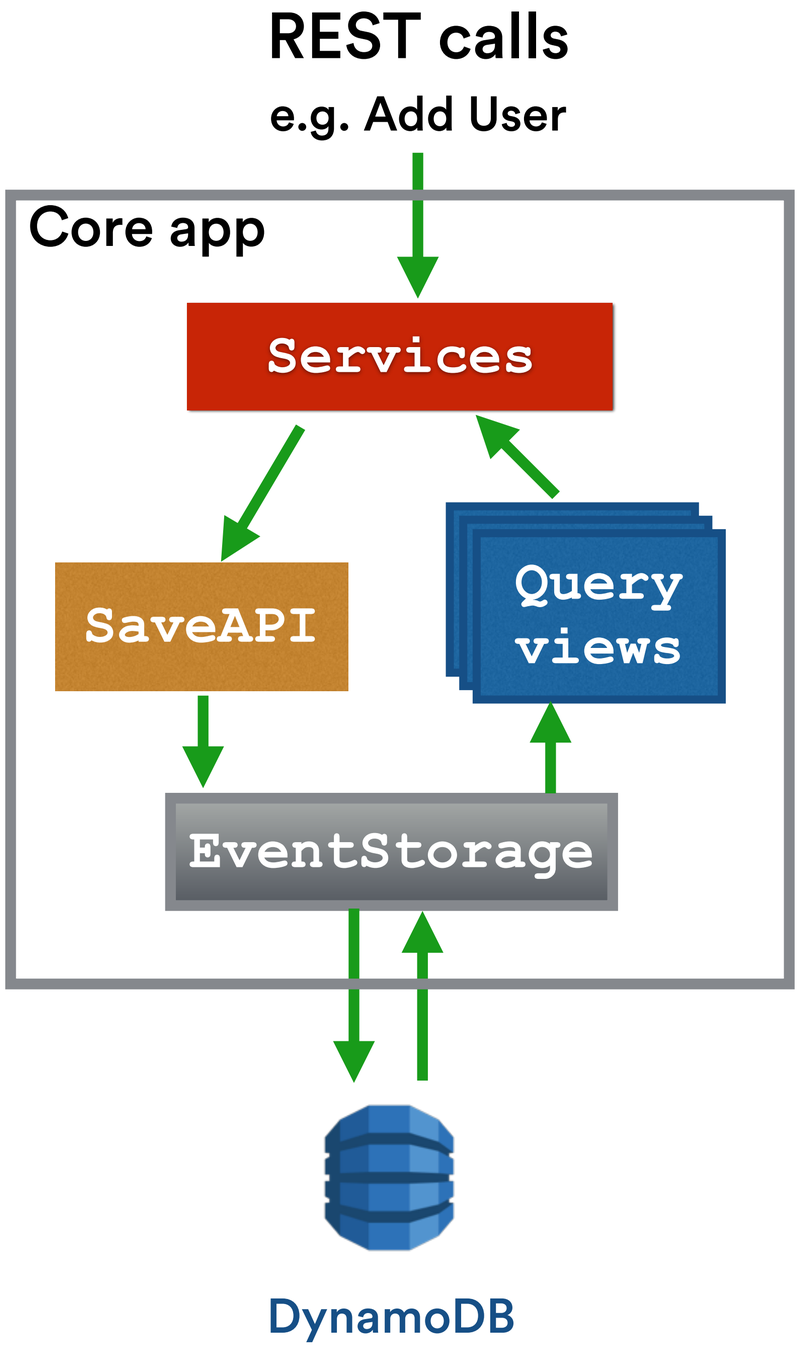

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStorage

DynamoDB

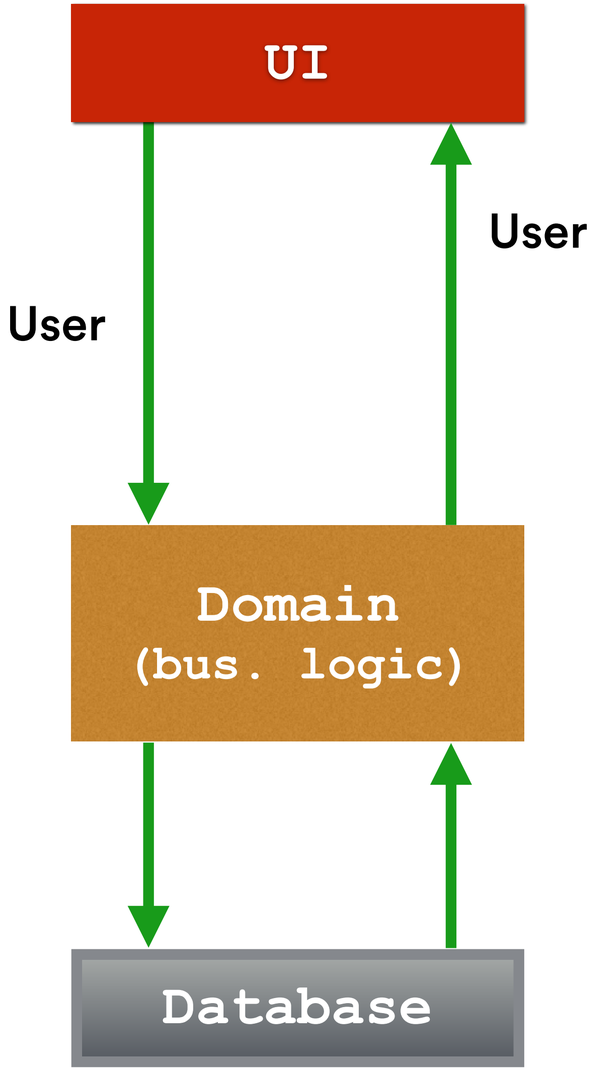

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStorage
DynamoDB

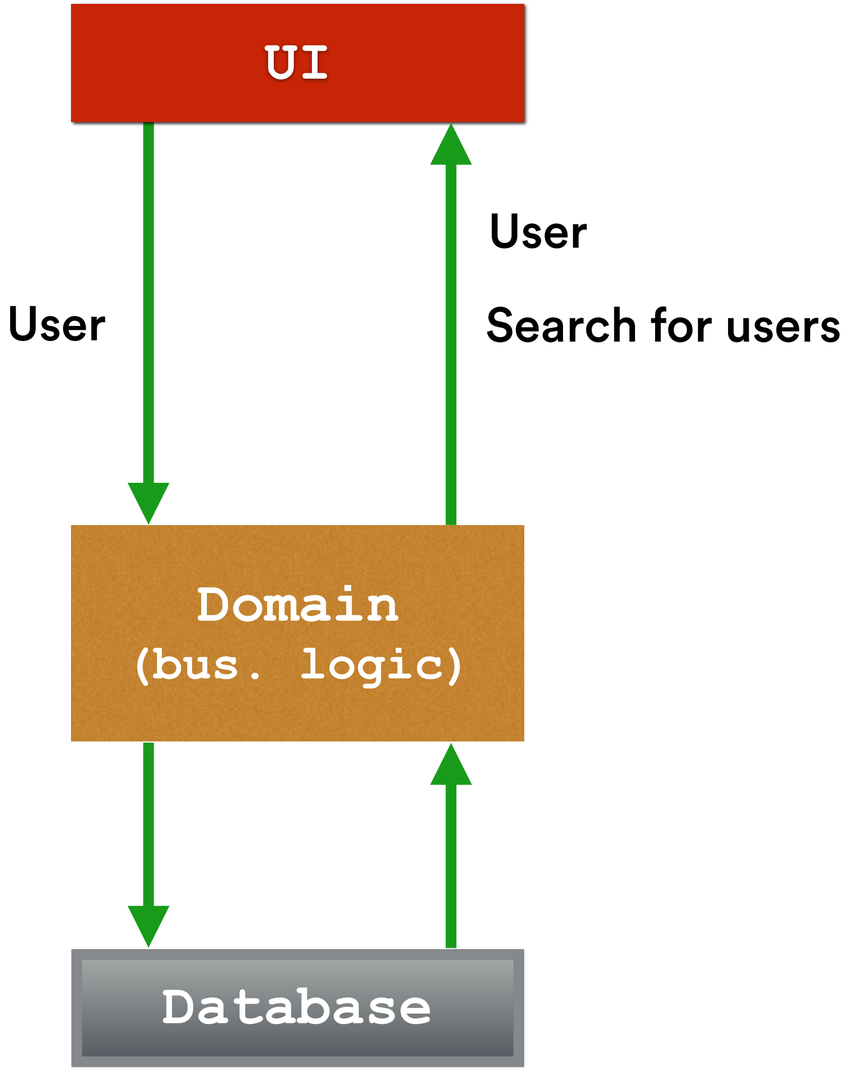

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStorage
DynamoDB

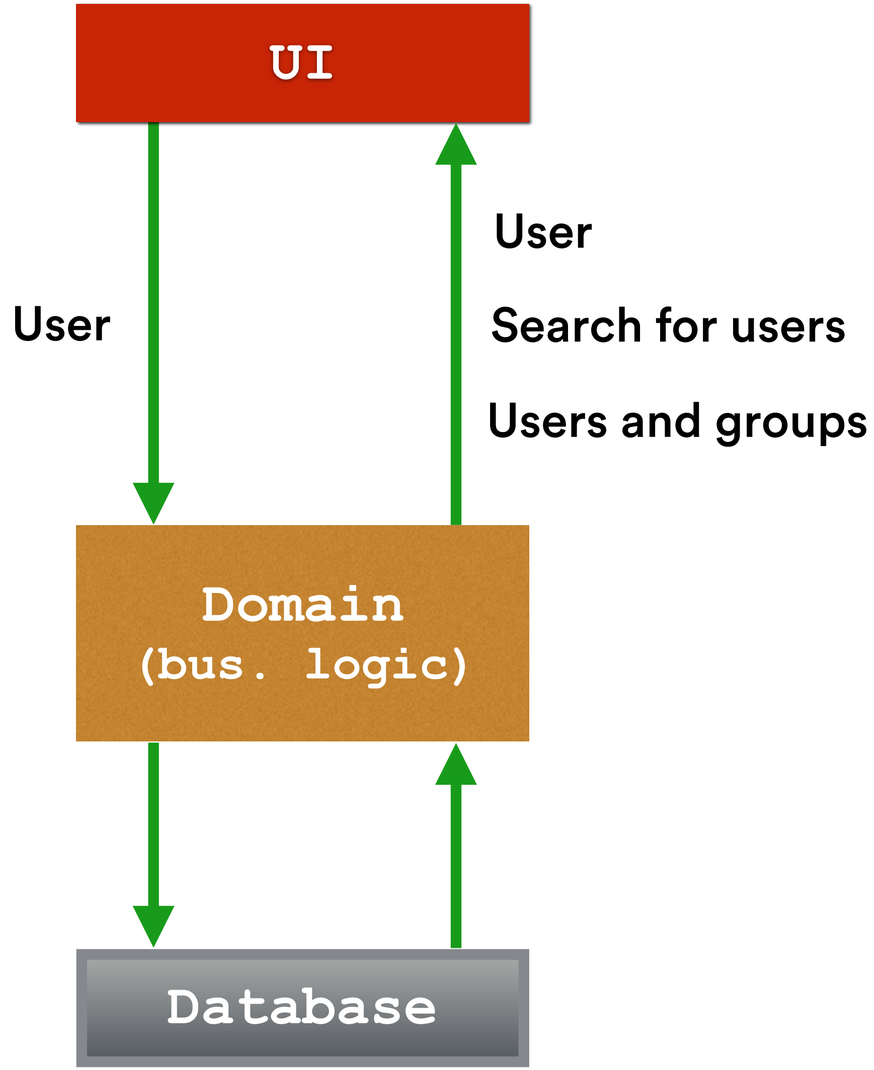

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStorage
DynamoDB

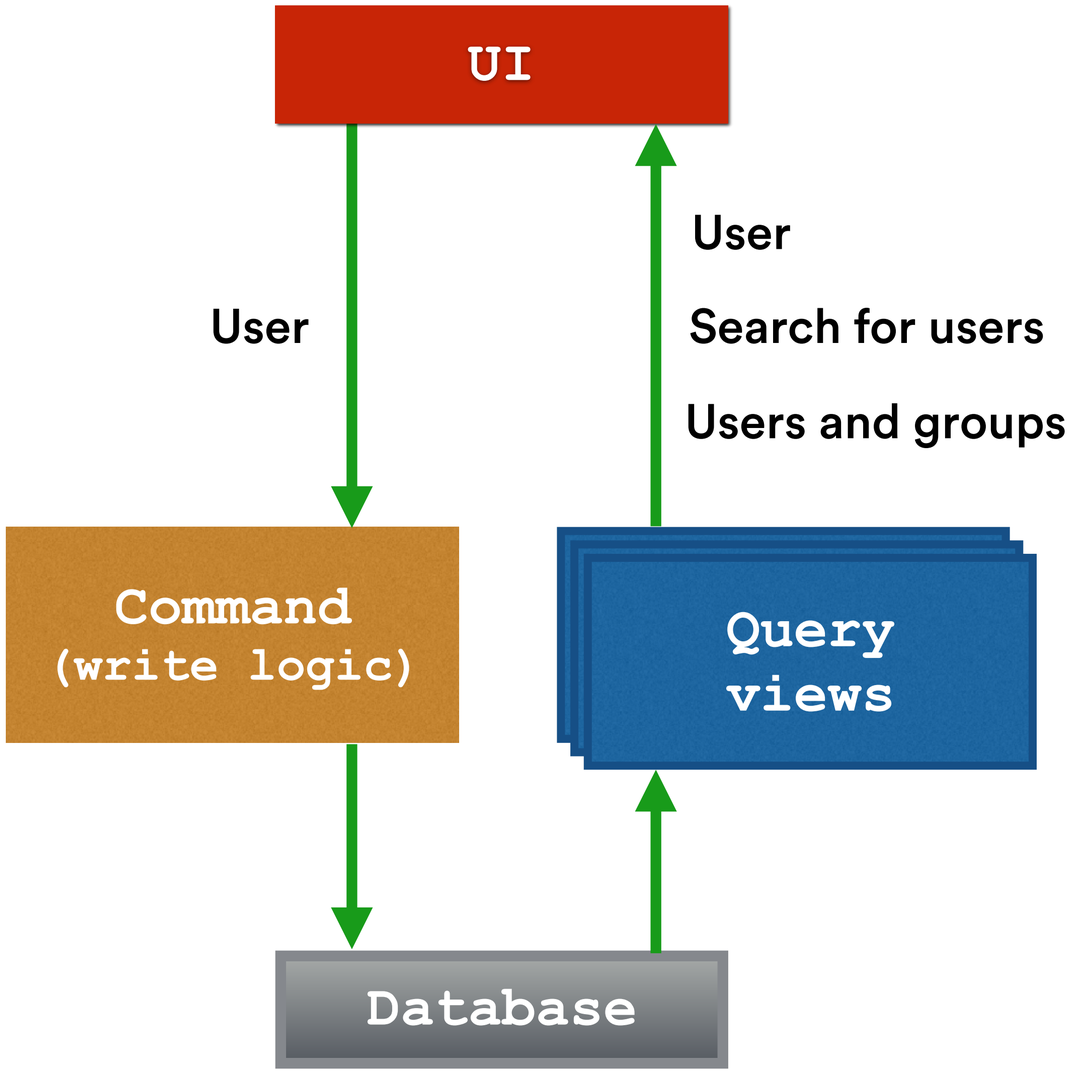

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStorage
DynamoDB

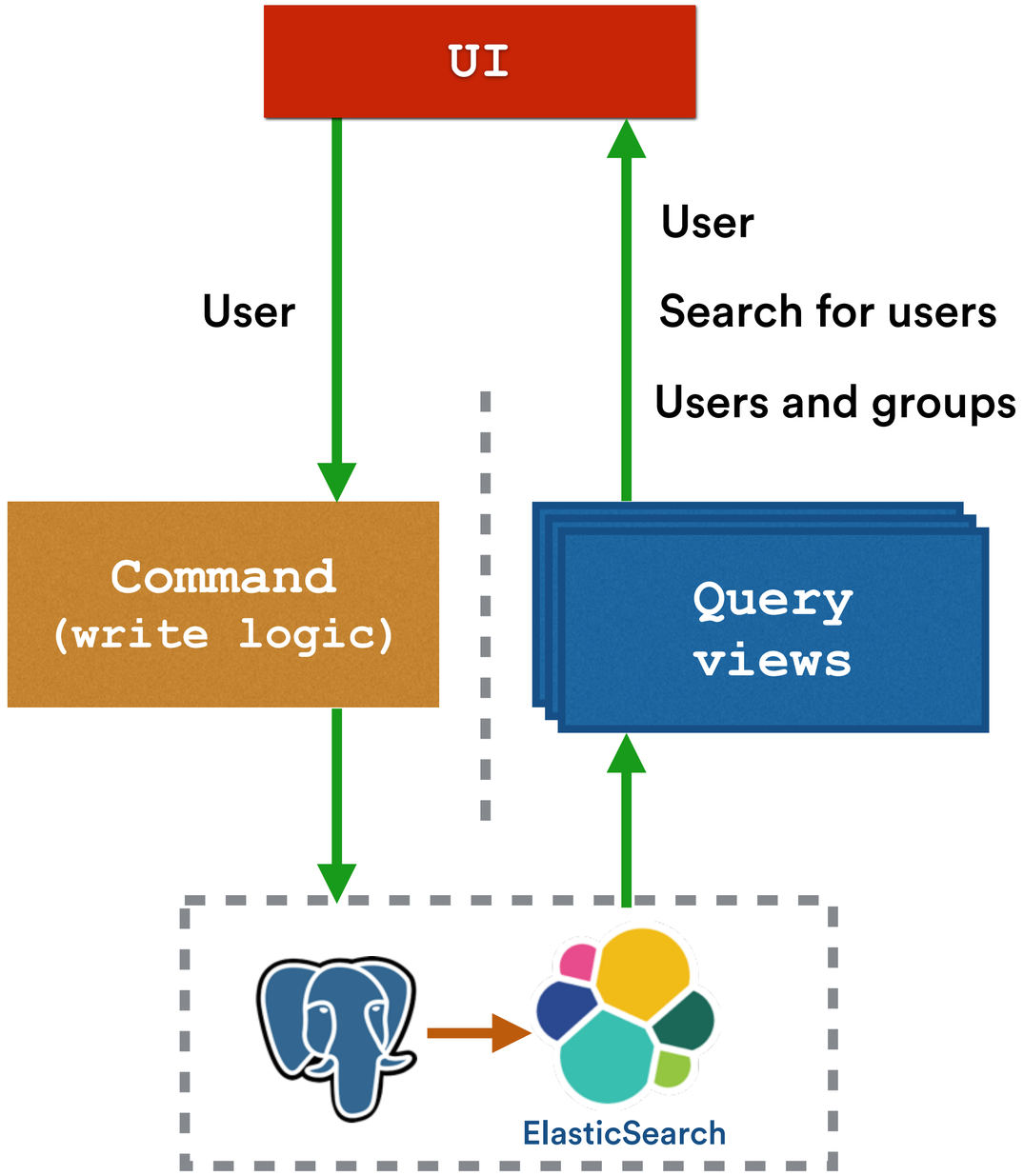
Command Query Responsibility Segregation

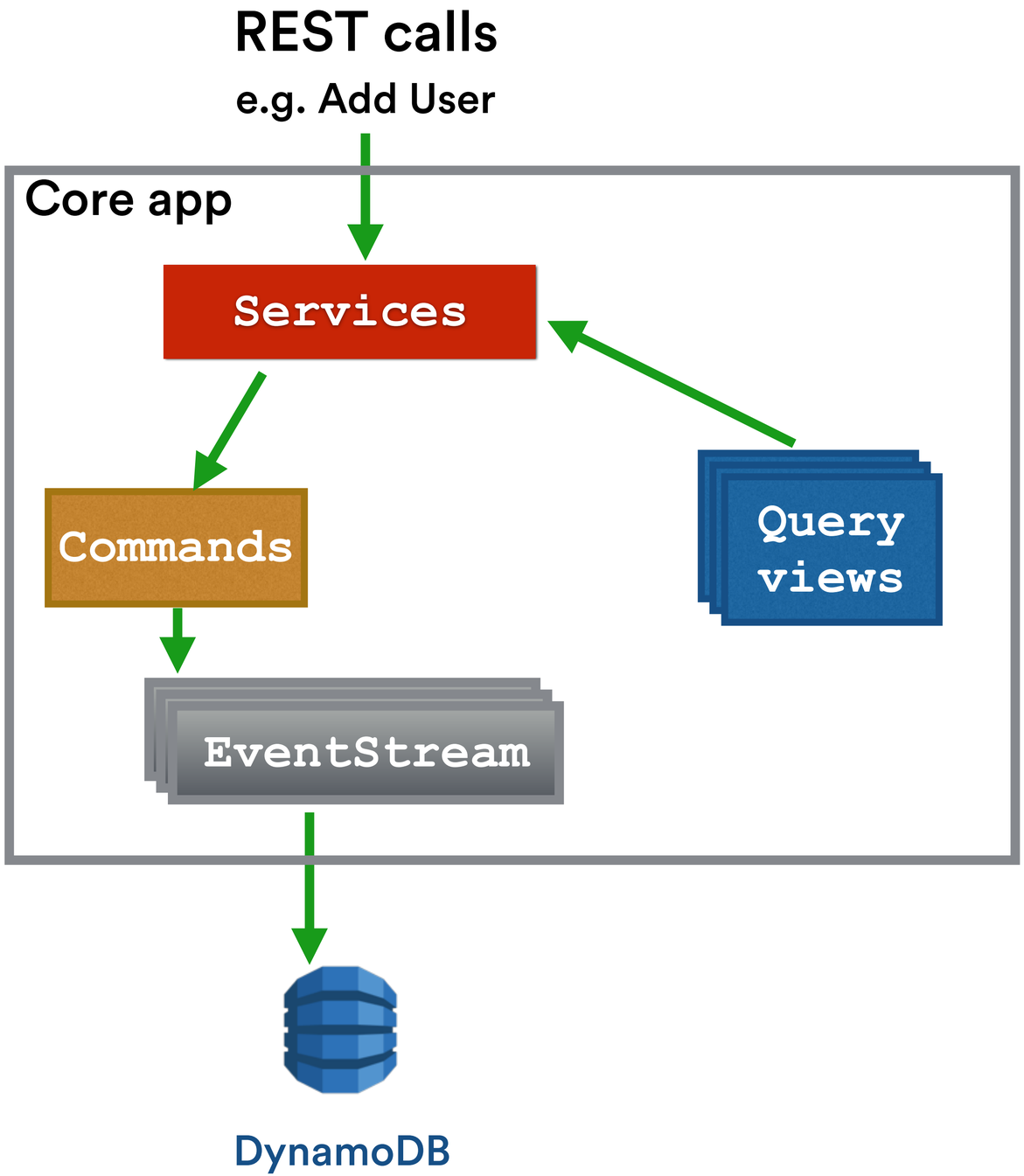

UI
User
User
Domain
(bus. logic)
Database

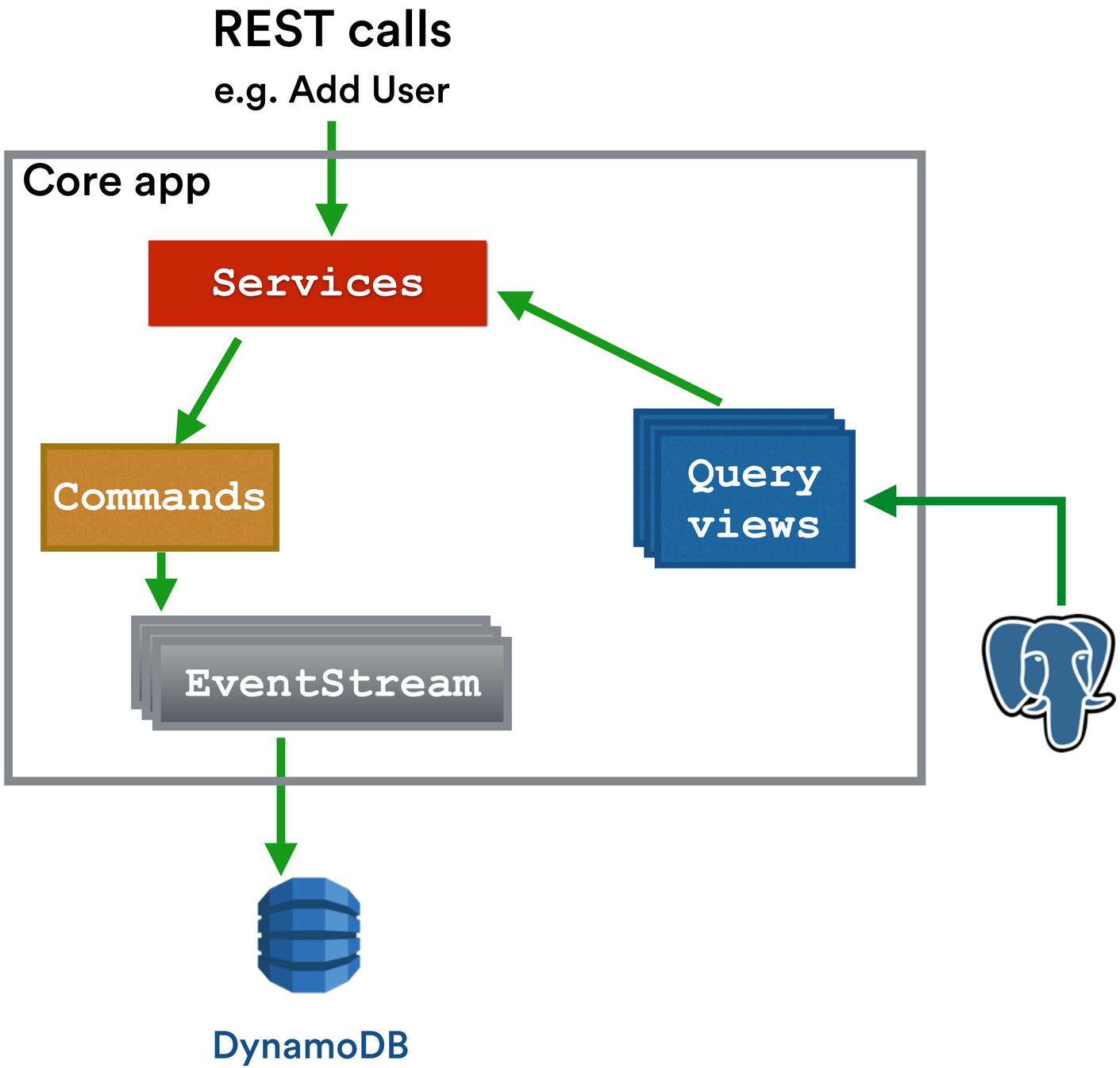

UI
User
User
Search for users
Domain
(bus. logic)
Database

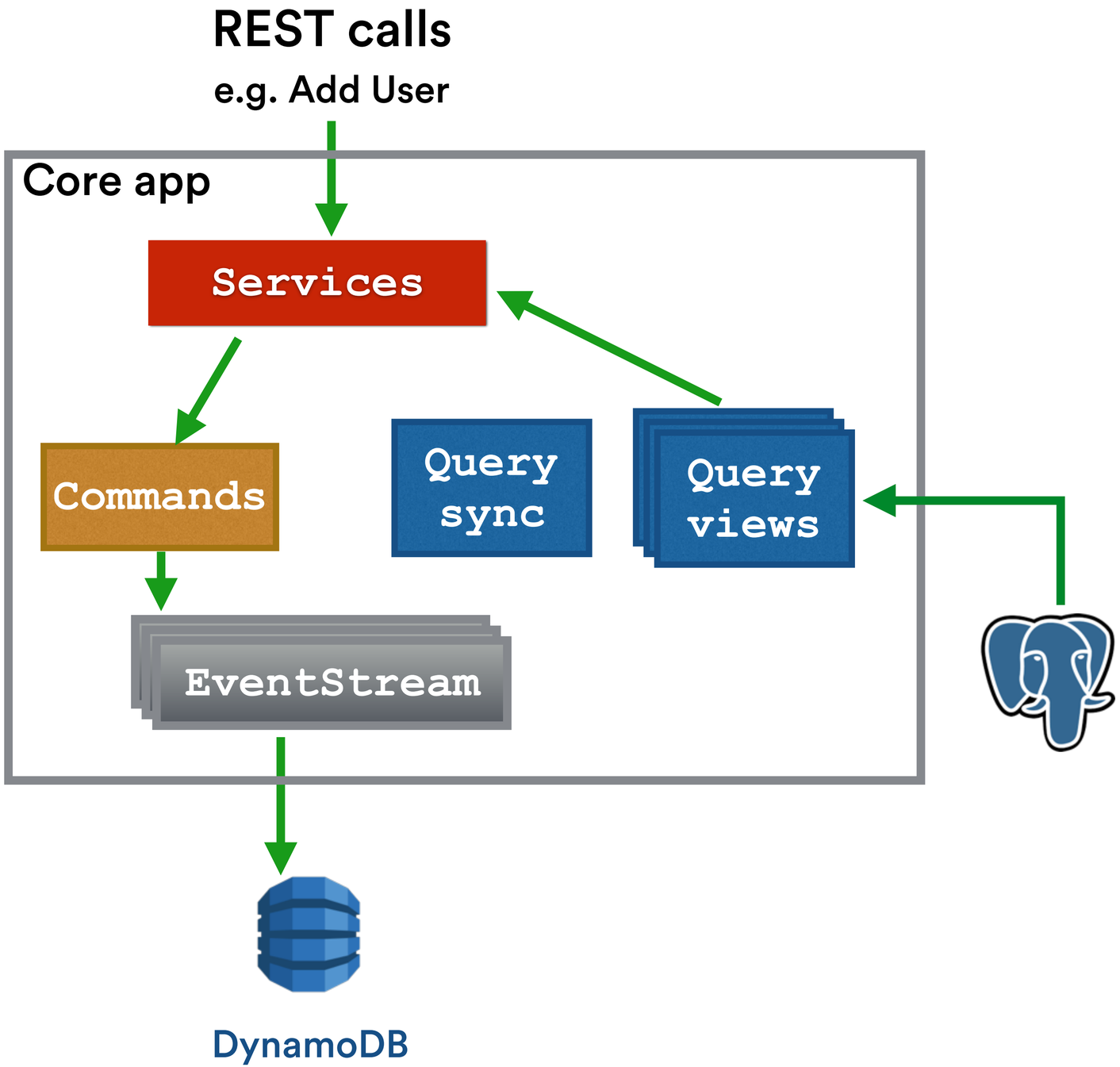

UI
User
User
Search for users
Users and groups
Domain
(bus. logic)
Database

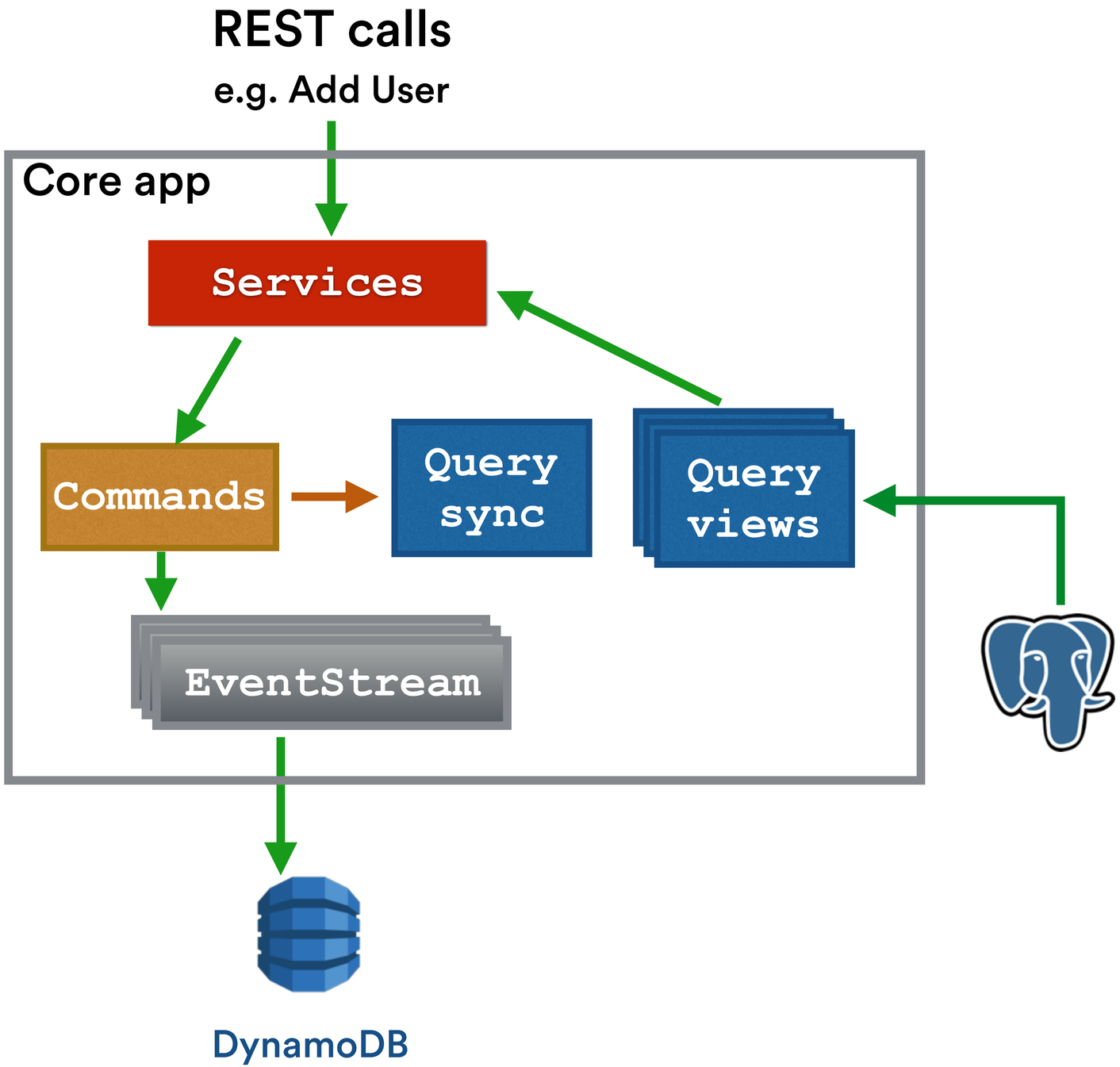

UI
User
Command
(write logic)
Database
User
Search for users
Users and groups
Query
views

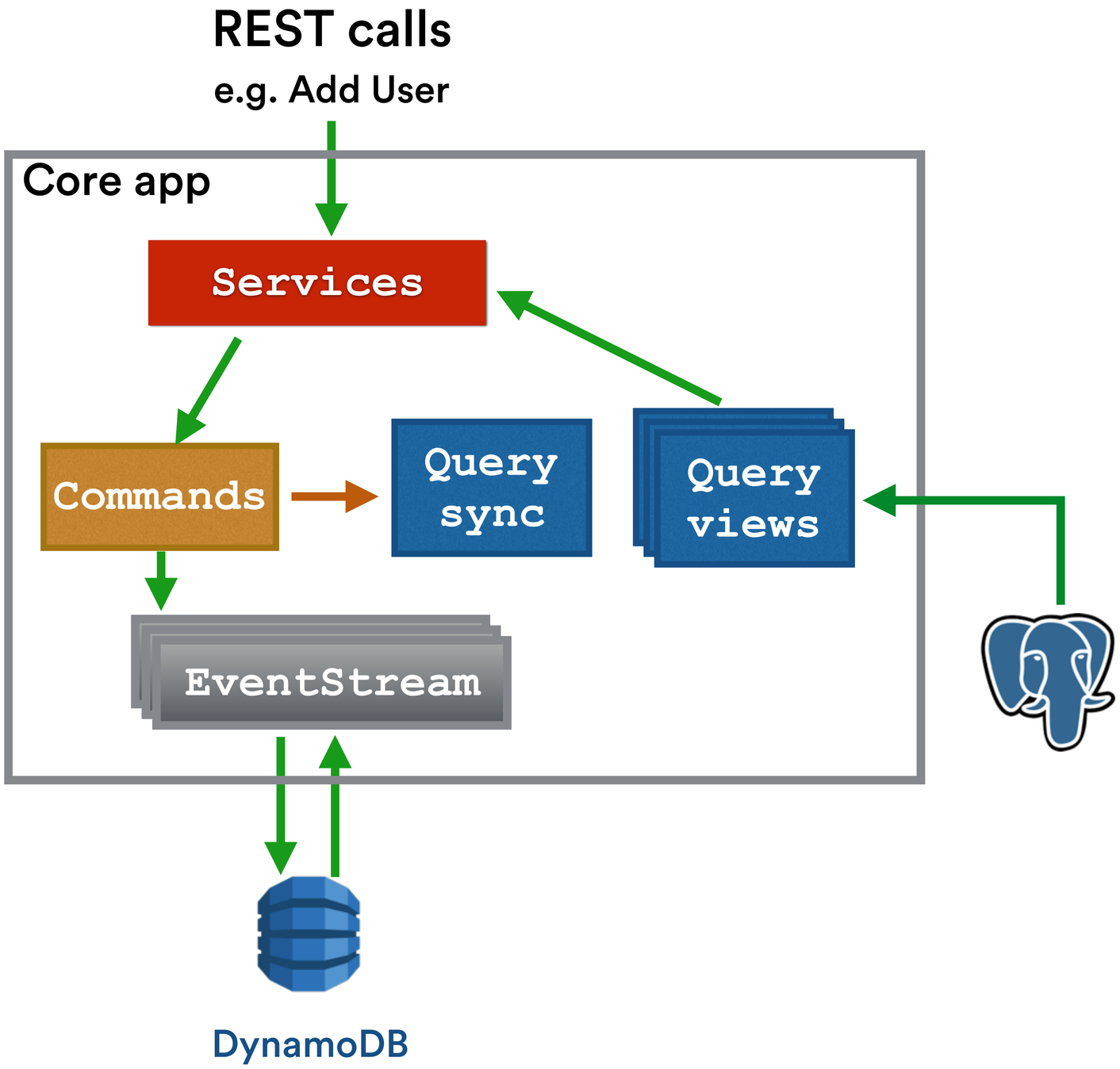

UI
User
User
Search for users
Users and groups
Command
(write logic)
Query
views
ElasticSearch

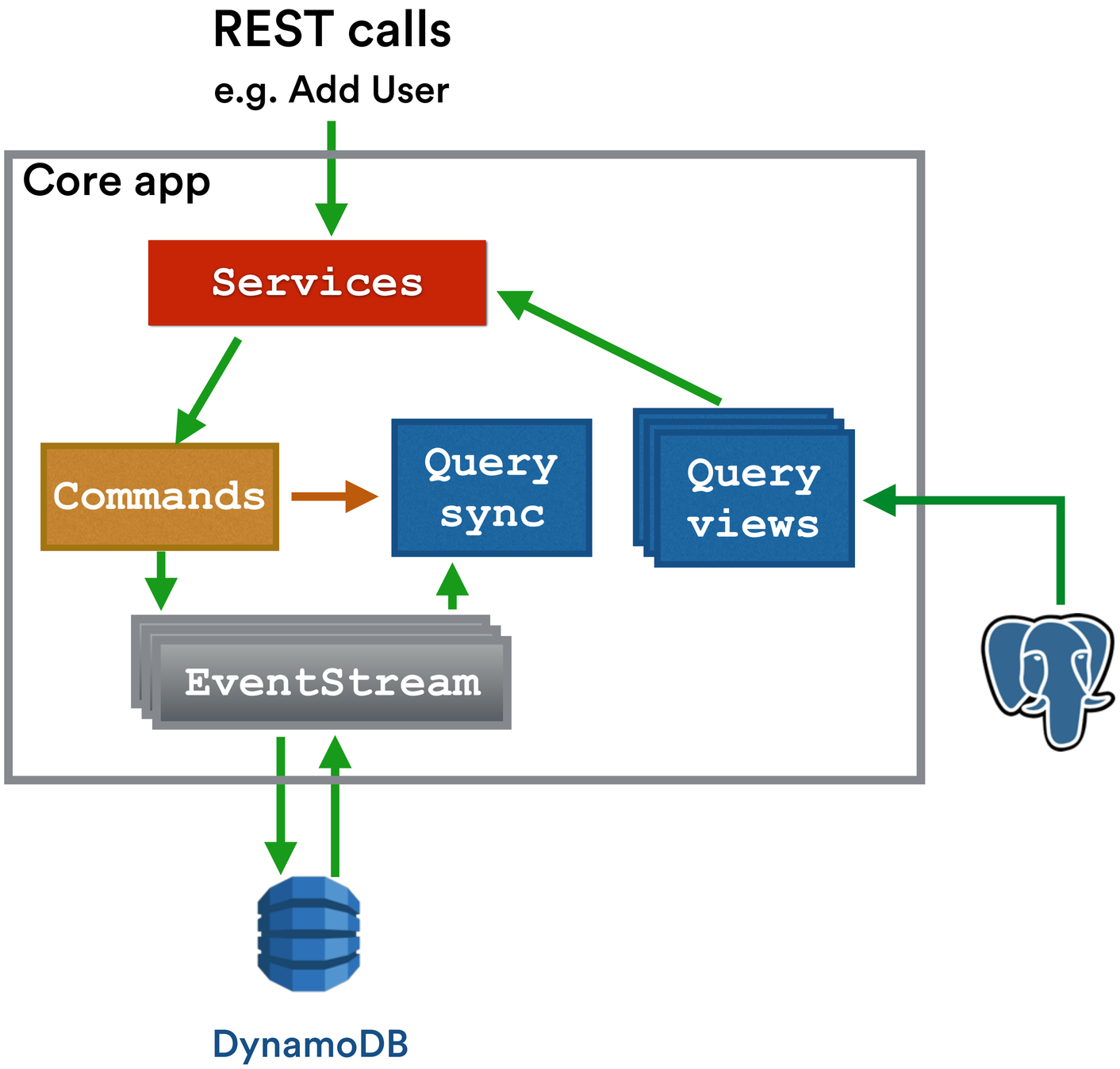

REST calls
e.g. Add User
Core app
Services
Commands
Query views
EventStream
DynamoDB

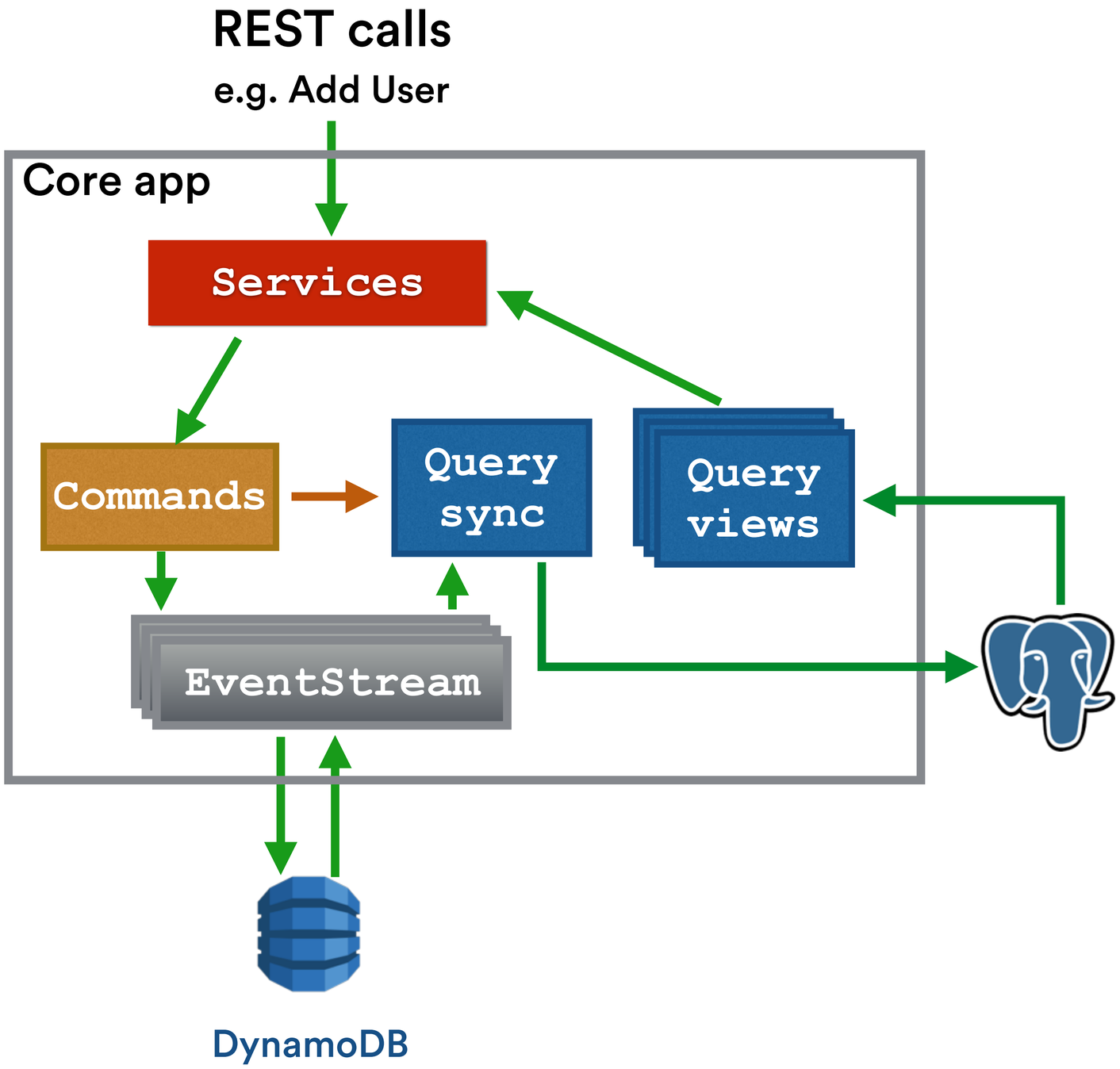

REST calls
e.g. Add User
Core app
Services
Commands
EventStream
Query views
DynamoDB

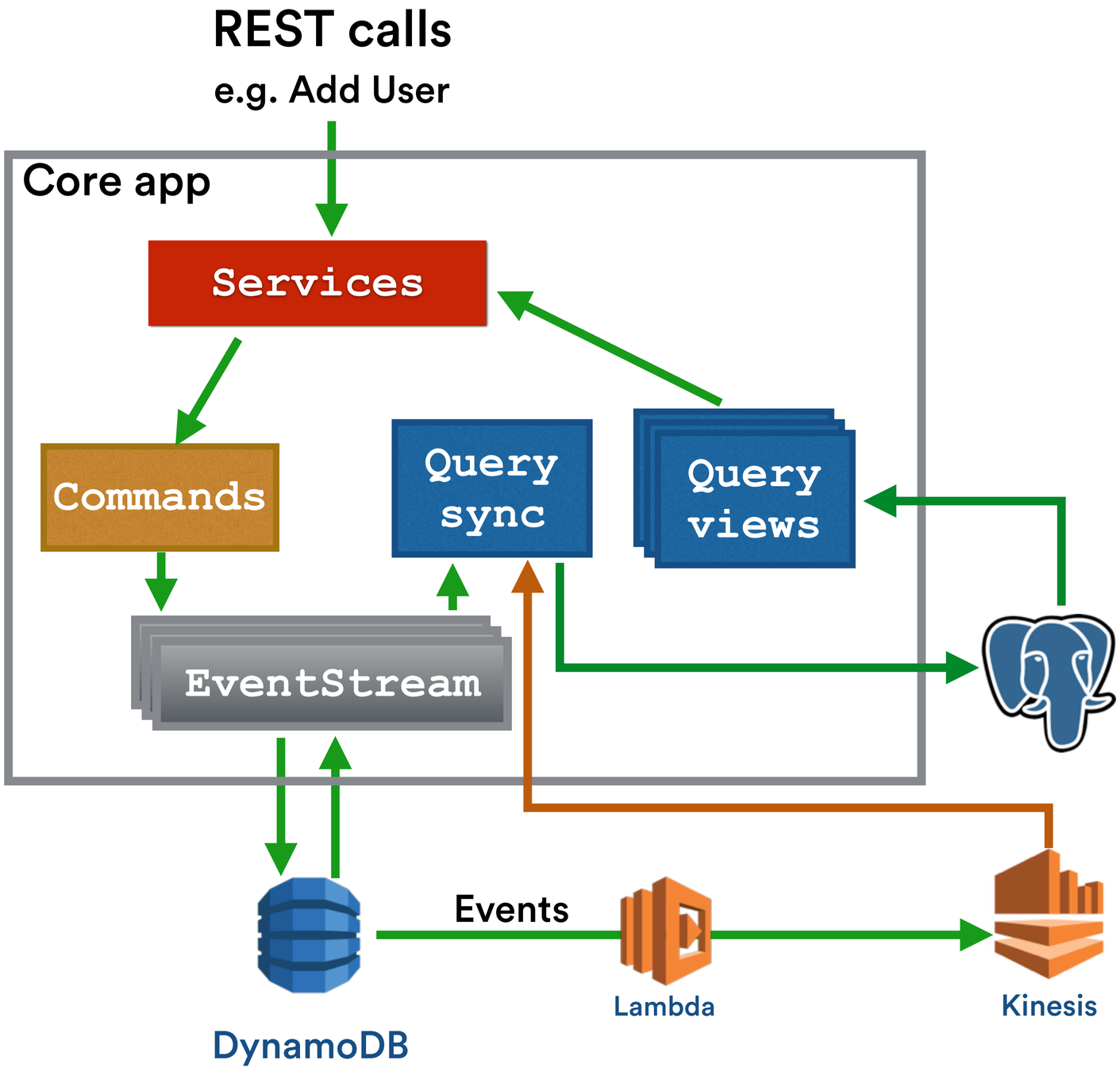

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

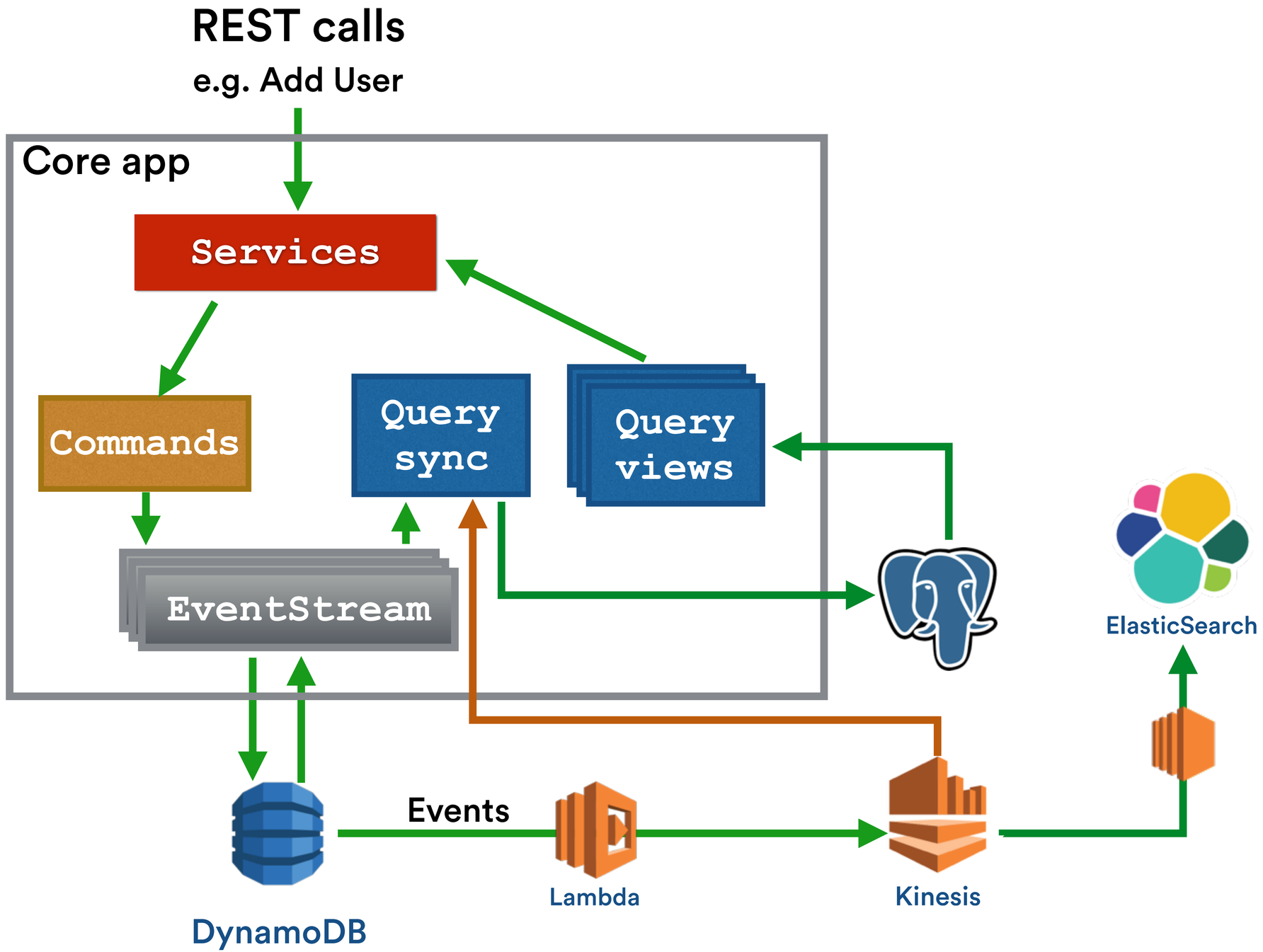

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

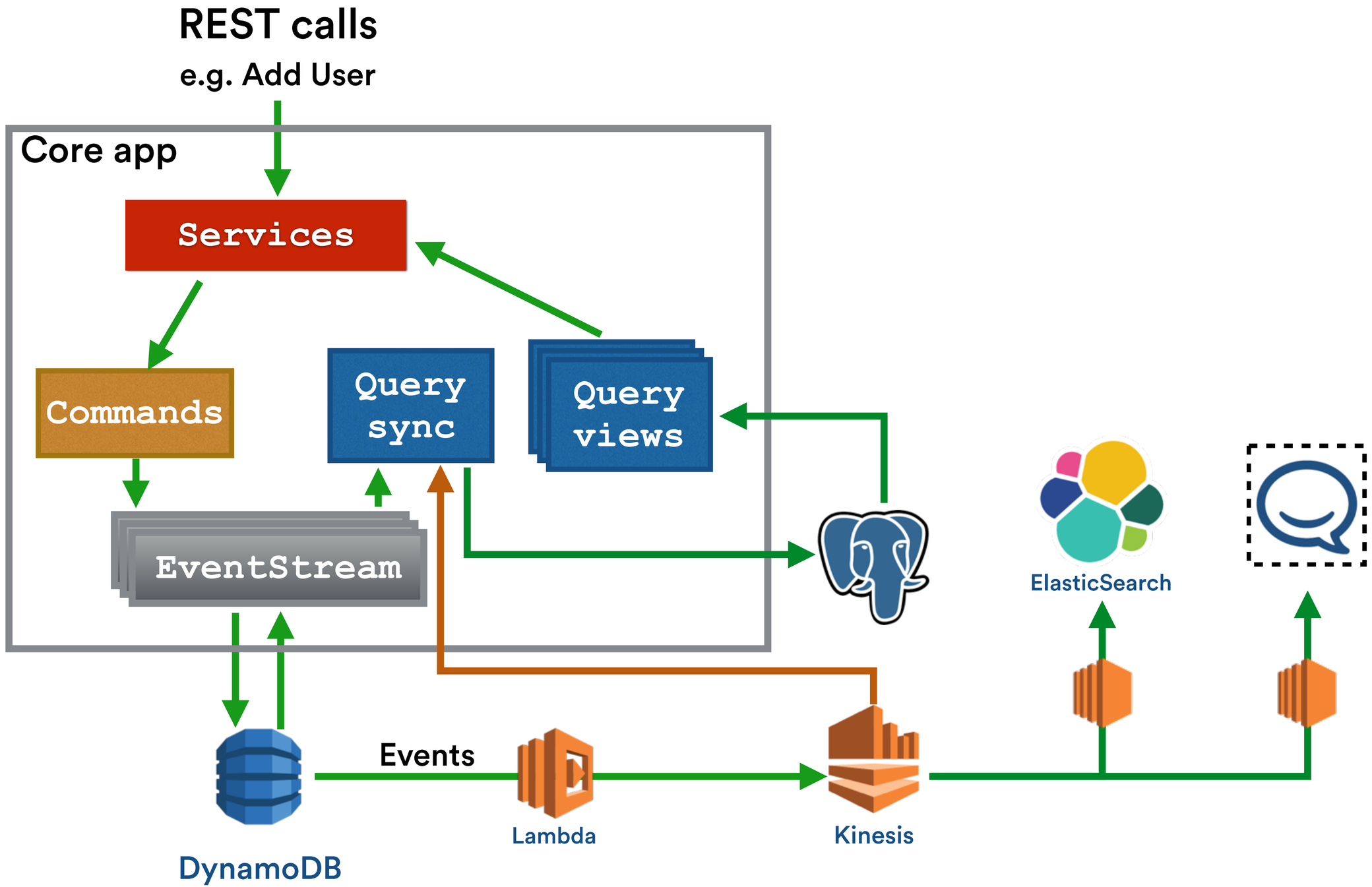

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

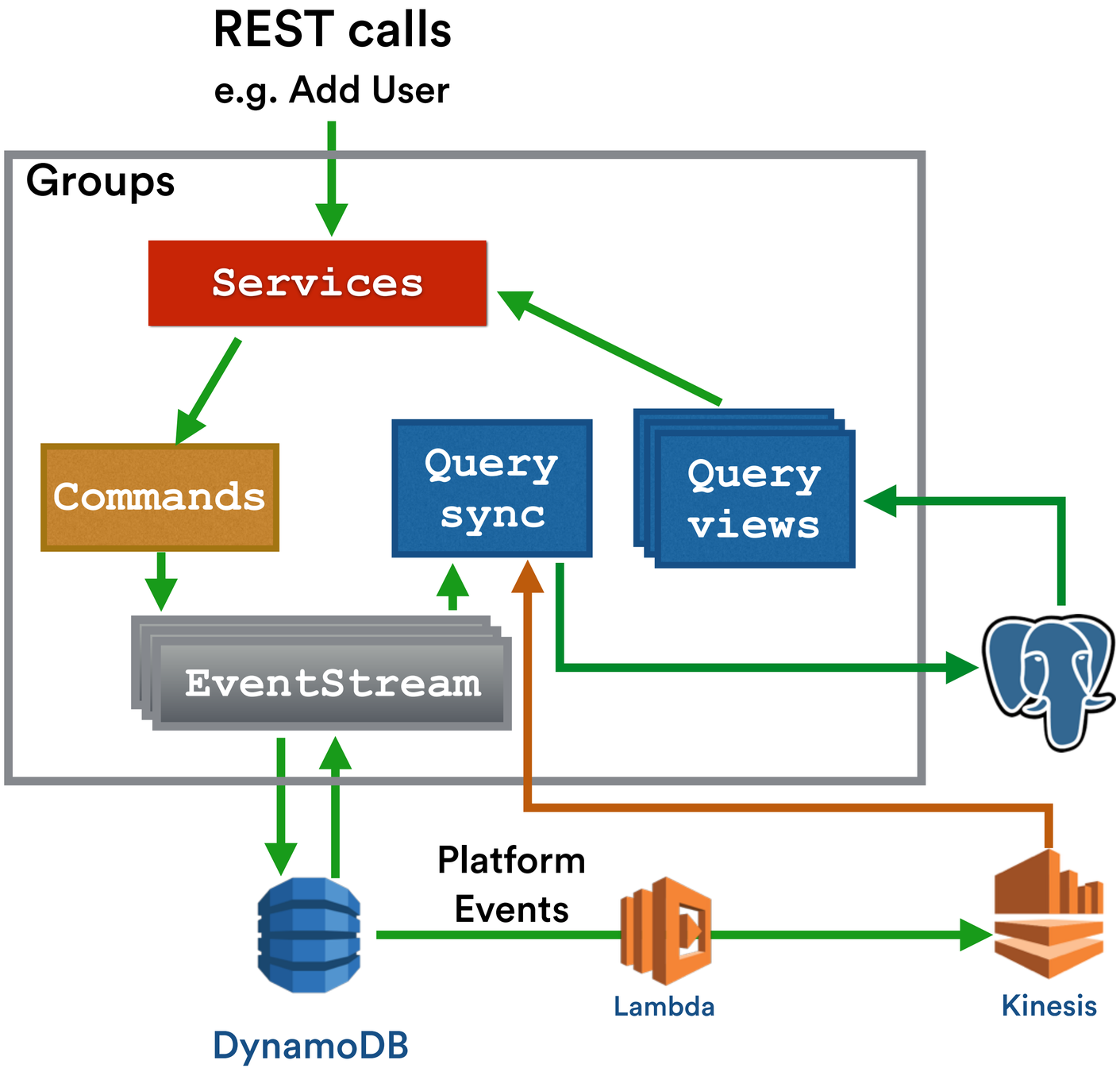

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

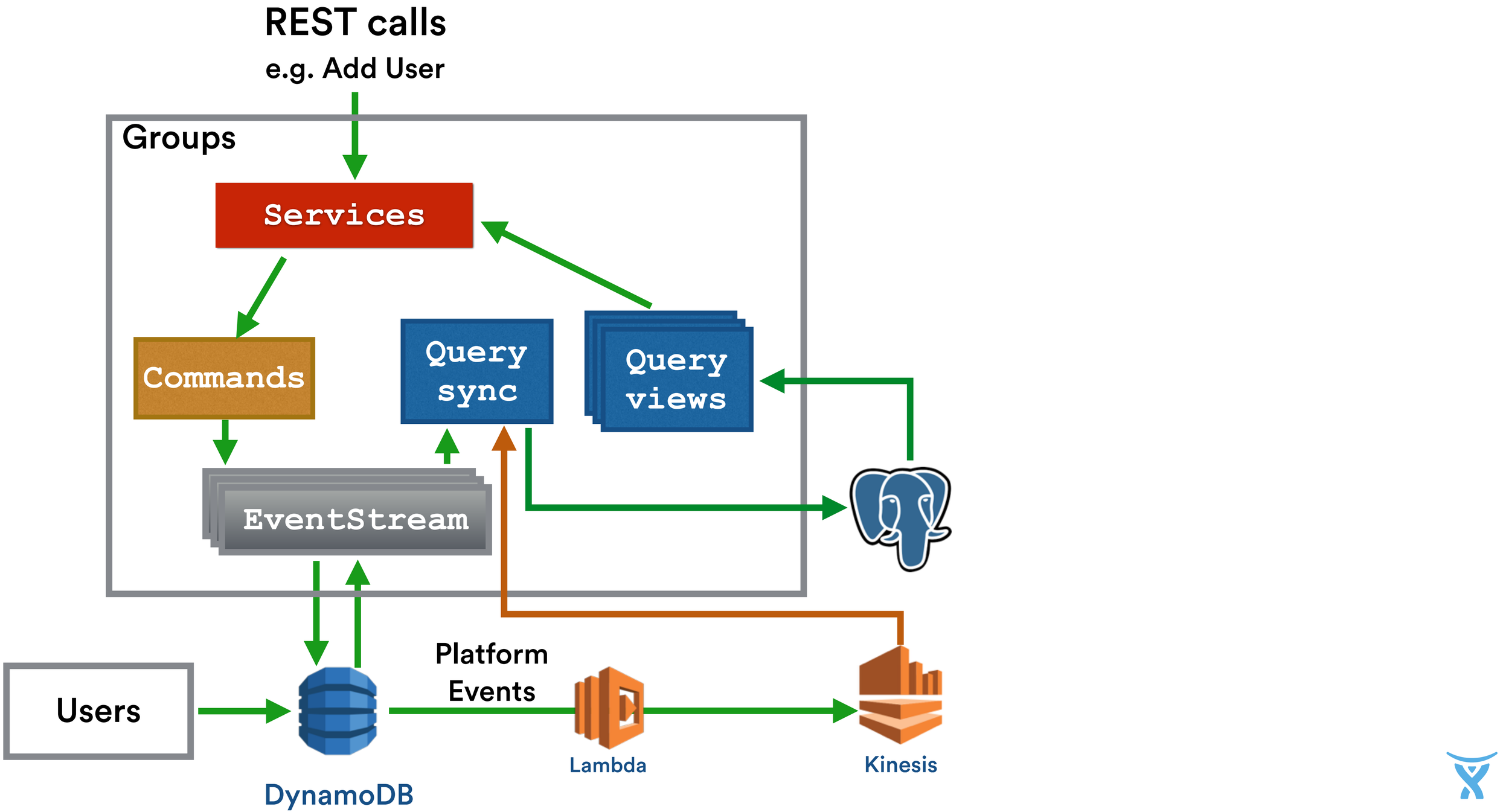

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

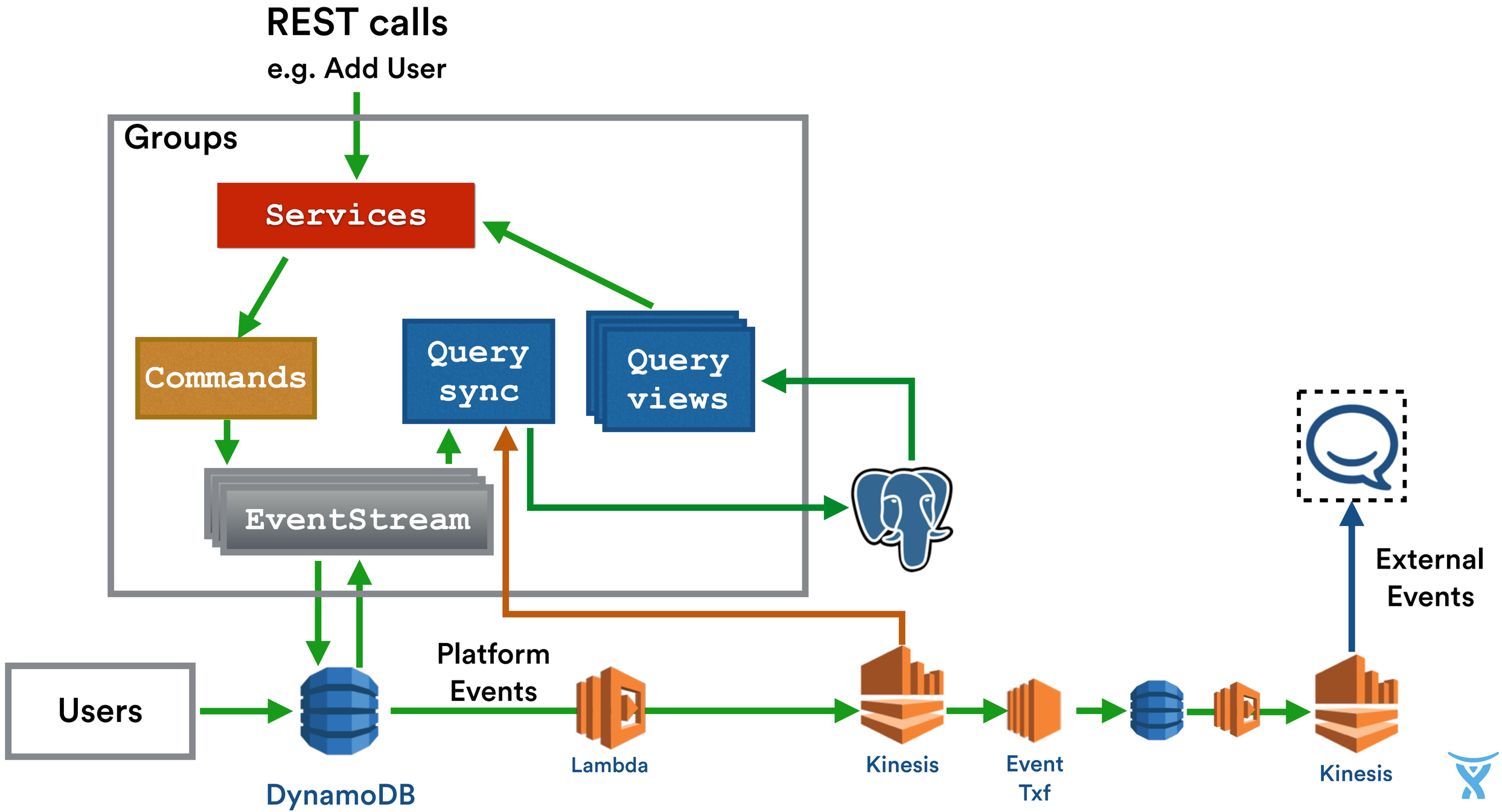

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
Events
Lambda
Kinesis
DynamoDB


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

Delving into code

Modeling Events

Key	Seq	Event	Time
3	123	SetUsername(Maggie)	0
3	124	SetName(Lisa Jr)	10
1	100	SetUsername(homers)	15

Modeling Events

Ordered ‘Sequence’ (key K + seq S)

Key	Seq	Event	Time
3	123	SetUsername(Maggie)	0
3	124	SetName(Lisa Jr)	10
1	100	SetUsername(homers)	15

Modeling Events

Ordered ‘Sequence’ (key K + seq S)

Key	Seq	Event	Time
3	123	SetUsername(Maggie)	0
3	124	SetName(Lisa Jr)	10
1	100	SetUsername(homers)	15

Flexible Value (E)

Event storage API

```
trait EventStorage[F[_], K, S, E] {

}
```

Event storage API

```
trait EventStorage[F[_], K, S, E] {

  def put(event: Event[K, S, E]):
    F[Error \/ Event[K, S, E]]



}
```

Event storage API

```
trait EventStorage[F[_], K, S, E] {

  def put(event: Event[K, S, E]):
    F[Error \/ Event[K, S, E]]

  def latest(key: K): OptionT[F, Event[K, S, E]]



}
```

Event storage API

```
trait EventStorage[F[_], K, S, E] {

  def put(event: Event[K, S, E]):
    F[Error \/ Event[K, S, E]]

  def latest(key: K): OptionT[F, Event[K, S, E]]

  def get(key: K): Stream[Event[K, S, E]]?

}
```

Event storage API

```
import scalaz.stream.Process

trait EventStorage[F[_], K, S, E] {

  def put(event: Event[K, S, E]):
    F[Error \/ Event[K, S, E]]

  def latest(key: K): OptionT[F, Event[K, S, E]]

  def get(key: K): Process[F, Event[K, S, E]]

}
```

Saving events

Saving events

```
case class SaveAPI[K, S, E](store: EventStorage...){
  def save(k: K, e: E): F[SaveResult[S]] =




}
```

Saving events

```
case class SaveAPI[K, S, E](store: EventStorage...){
  def save(k: K, e: E): F[SaveResult[S]] =
    for {
      old <- store.latest(k).run

    } yield ???
}
```

Saving events

```
case class SaveAPI[K, S, E](store: EventStorage...){
  def save(k: K, e: E): F[SaveResult[S]] =
    for {
      old <- store.latest(k).run
      putResult <- eventStore.put(k, Event.next(old, e))
    } yield ???
}
```

Saving events

```
case class SaveAPI[K, S, E](store: EventStorage...){
  def save(k: K, e: E): F[SaveResult[S]] =
    for {
      old <- store.latest(k).run
      putResult <- eventStore.put(k, Event.next(old, e))
      saveResult <- putResult match {
        case \/-(ev) =>
          SaveResult.success(ev.id.seq)




      }
    } yield saveResult
}
```

Saving events

```
case class SaveAPI[K, S, E](store: EventStorage...){
  def save(k: K, e: E): F[SaveResult[S]] =
    for {
      old <- store.latest(k).run
      putResult <- eventStore.put(k, Event.next(old, e))
      saveResult <- putResult match {
        case \/-(ev) =>
          SaveResult.success(ev.id.seq)
        case -\/(Error.DuplicateEventId) =>
          save(k, e)  // With back off + jitter


      }
    } yield saveResult
}
```

Saving events

```
case class SaveAPI[K, S, E](store: EventStorage...){
  def save(k: K, e: E): F[SaveResult[S]] =
    for {
      old <- store.latest(k).run
      putResult <- eventStore.put(k, Event.next(old, e))
      saveResult <- putResult match {
        case \/-(ev) =>
          SaveResult.success(ev.id.seq)
        case -\/(Error.DuplicateEventId) =>
          save(k, e)  // With back off + jitter
        case -\/(Error.Rejected(reasons)) =>
          SaveResult.reject(reasons)
      }
    } yield saveResult
}
```

What about data constraints?

Operation: Constraint as a type

```
case class Operation[S, E](apply: Option[S] => OpResult[E])
```

Operation: Constraint as a type

```
case class Operation[S, E](apply: Option[S] => OpResult[E])


sealed trait OpResult[E]
case class Success[E](e: E) extends OpResult[E]
case class Reject[E](reasons: List[Reason]) extends OpResult[E]
```

Operation: Constraint as a type

```
object Operation {
 def ifSeq(seq: Option[S], e: E): Operation[S, E] =
    Operation { s =>
      if (s === seq) Success(e)
      else Reject(List(Reason("Sequence mismatch")))
    }

  ...
}
```

Save without Constraints

```
case class SaveAPI[K, S, E](store: EventStorage...) {
  def save(k: K, e: E): F[SaveResult[S] =
    for {
      old <- store.latest(k).run

      putResult <- eventStore.put(k, Event.next(old, e))



      saveResult <- putResult match {
        case \/-(ev) =>
          SaveResult.success(query.acc(old, ev).value)
        case -\/(Error.DuplicateEventId) =>
          save(k, e)
        case -\/(Error.Rejected(reasons)) =>
          SaveResult.reject(reasons)
      }
    } yield saveResult
}
```

Save with Constraints

```
case class SaveAPI[K, S, E](store: EventStorage...) {
  def save(k: K, op: Operation[S, E]): F[SaveResult[S] =
    for {
      old <- store.latest(k).run

      saveResult <- putResult match {
        case \/-(ev) =>
          SaveResult.success(query.acc(old, ev).value)
        case -\/(Error.DuplicateEventId) =>
          save(k, e)
        case -\/(Error.Rejected(reasons)) =>
          SaveResult.reject(reasons)
      }
    } yield saveResult
}
```

Save with Constraints

```
case class SaveAPI[K, S, E](store: EventStorage...) {
  def save(k: K, op: Operation[S, E]): F[SaveResult[S] =
    for {
      old <- store.latest(k).run
      opResult = op.apply(old)

      saveResult <- putResult match {
        case \/-(ev) =>
          SaveResult.success(query.acc(old, ev).value)
        case -\/(Error.DuplicateEventId) =>
          save(k, e)
        case -\/(Error.Rejected(reasons)) =>
          SaveResult.reject(reasons)
      }
    } yield saveResult
}
```

Save with Constraints

```
case class SaveAPI[K, S, E](store: EventStorage...) {
  def save(k: K, op: Operation[S, E]): F[SaveResult[S] =
    for {
      old <- store.latest(k).run
      opResult = op.apply(old)
      putResult <- opResult match {
        case Success(e) => eventStore.put(k, Event.next(old.seq, e))
        case Reject(rs) => SaveResult.reject(rs)
      }
      saveResult <- putResult match {
        case \/-(ev) =>
          SaveResult.success(query.acc(old, ev).value)
        case -\/(Error.DuplicateEventId) =>
          save(k, e)
        case -\/(Error.Rejected(reasons)) =>
          SaveResult.reject(reasons)
      }
    } yield saveResult
}
```

Querying by pulling

Querying event streams

```
def get(k: Key): F[Option[Val]] =

    eventStore.get(???)
```

Querying event streams

```
type Ev = Event[K, S, E]

def get(k: Key): F[Option[Val]] =
  streamFold(acc) {
    eventStore.get(???)
  }

def streamFold(
  f: (Option[Val], Ev) => Option[Val]
  )(stream: Process[F, Ev]): F[Option[S, Val]] = ???
```

Querying event streams

```
import scalaz.stream.process1

def get(k: Key): F[Option[Val]] =
  streamFold(acc) {
    eventStore.get(???)
  }

def streamFold(
  f: (Option[Val], Ev) => Option[Val]
  )(stream: Process[F, Ev]): F[Option[Val]] =
    stream.pipe {
      process1.fold(None)(f)
    }.runLastOr(None)
```

Querying event streams

```
import scalaz.stream.process1

def get(toStreamKey: Key => K)(k: Key): F[Option[Val]] =
  streamFold(acc) {
    eventStore.get(toStreamKey(k))
  }

def streamFold(
  f: (Option[Val], Ev) => Option[Val]
  )(stream: Process[F, Ev]): F[Option[Val]] =
    stream.pipe {
      process1.fold(None)(f)
    }.runLastOr(None)
```

Querying event streams

```
case class QueryAPI[F[_], K, S, E, Key, Val](
  toStreamKey: Key => K,
  acc:(k: Key) => (s: Snapshot[S, Val], e: Ev)
                => Snapshot[S, Val],
  store: EventStorage[F, K, S, E]) {

  def get(k: Key): F[Option[Val]] =
    ...
}
```

Querying by pushing

Query synchroniser is a scalaz-stream Channel...

```
class QuerySynchroniser(db: DB) {

  def synchronise(s: Process[Task, StreamEvent]) =









}
```

Query synchroniser is a scalaz-stream Channel...

```
class QuerySynchroniser(db: DB) {

  def synchronise(s: Process[Task, StreamEvent]) =
    (s through writeEvent).run

  private val writeEvent: Channel[Task, StreamEvent, Unit] =
    Process.constant { e: StreamEvent =>




  }
}
```

Query synchroniser is a scalaz-stream Channel...

```
class QuerySynchroniser(db: DB) {

  def synchronise(s: Process[Task, StreamEvent]) =
    (s through writeEvent).run

  private val writeEvent: Channel[Task, StreamEvent, Unit] =
    Process.constant { e: StreamEvent =>
      for {
        r1 <- StreamState.transitionSeq(db, e.id.key, e.id.seq)

      } yield ()
    }
}
```

Query synchroniser is a scalaz-stream Channel...

```
class QuerySynchroniser(db: DB) {

  def synchronise(s: Process[Task, StreamEvent]) =
    (s through writeEvent).run

  private val writeEvent: Channel[Task, StreamEvent, Unit] =
    Process.constant { e: StreamEvent =>
      for {
        r1 <- StreamState.transitionSeq(db, e.id.key, e.id.seq)
        _  <- if (r1.success) db.process(e) else db.rollback
      } yield ()
    }
}
```

...that you pass it an event stream

```
class KinesisProcessor(qs: QuerySynchroniser,
                       es: EventStorage[...]) extends IRecordProcessor {

  override def processRecords(rs: ProcessRecordInput) = {




  }

}
```

...that you pass it an event stream

```
class KinesisProcessor(qs: QuerySynchroniser,
                       es: EventStorage[...]) extends IRecordProcessor {

  override def processRecords(rs: ProcessRecordInput) = {
    // Either emit events from Kinesis, or read events from Dynamo
    val stream = es.get((rs.getRecords...))


  }

}
```

...that you pass it an event stream

```
class KinesisProcessor(qs: QuerySynchroniser,
                       es: EventStorage[...]) extends IRecordProcessor {

  override def processRecords(rs: ProcessRecordInput) = {
    // Either emit events from Kinesis, or read events from Dynamo
    val stream = es.get((rs.getRecords...))

    qs.synchronise(stream).attemptRunFor(...)
  }

}
```

3.5 Design Principles

Principle 1: 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))

UserNameSet(id, name)

UserEmailSet(id, email1)

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

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

UserNameSet(id, name)

UserEmailSet(id, email1)

UserEmailSet(id, email2)

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

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

Fits nicely with CRUD + PATCH

UserNameSet(id, name)

UserEmailSet(id, email1)

UserEmailSet(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

UserNameSet(id, name)

UserEmailSet(id, email1)

UserEmailSet(id, email2)

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

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

UserNameSet(id, name)

UserEmailSet(id, email1)

UserEmailSet(id, email2)

Fits nicely with CRUD + PATCH

Assume insert before update

Encourages idempotent processing

Insert / Update Delta vs ‘Set’ events

UserAdded(id, name, email1)

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

UserNameSet(id, name)

UserEmailSet(id, email1)

UserEmailSet(id, email2)

Fits nicely with CRUD + PATCH

Assume insert before update

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

UserNameSet(id, name)
UserEmailSet(id, email1)

UserEmailSet(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

UserNameSet(id, name)
UserEmailSet(id, email1)

UserEmailSet(id, email2)

Encourages idempotent processing

Single code path for query sync

Minimally sized events to avoid conflict

Principle 2: Split event streams

Single stream

Multiple streams

Single stream

Transactions and consistent data resolution

Multiple streams

Single stream

Transactions and consistent data resolution

Multiple streams

Sharding for throughput

Single stream

Transactions and consistent data resolution

Multiple streams

Sharding for throughput

No ordering between streams

Single stream

~~Transactions and consistent data resolution~~

Multiple streams

Sharding for throughput

No ordering between streams

Better latency 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

Principle 3: 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

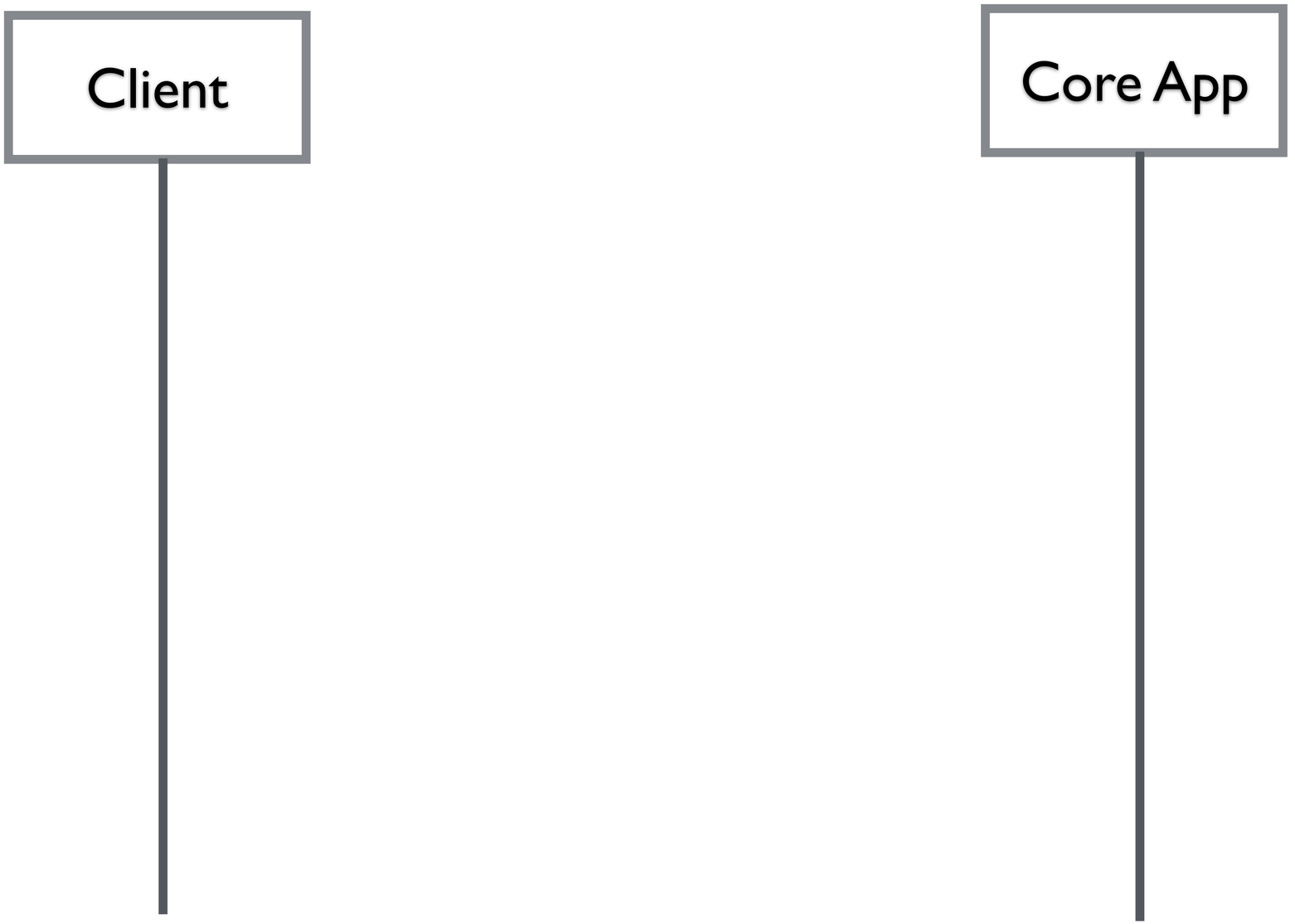
User: homer (id 4)

All Users: Seq 100

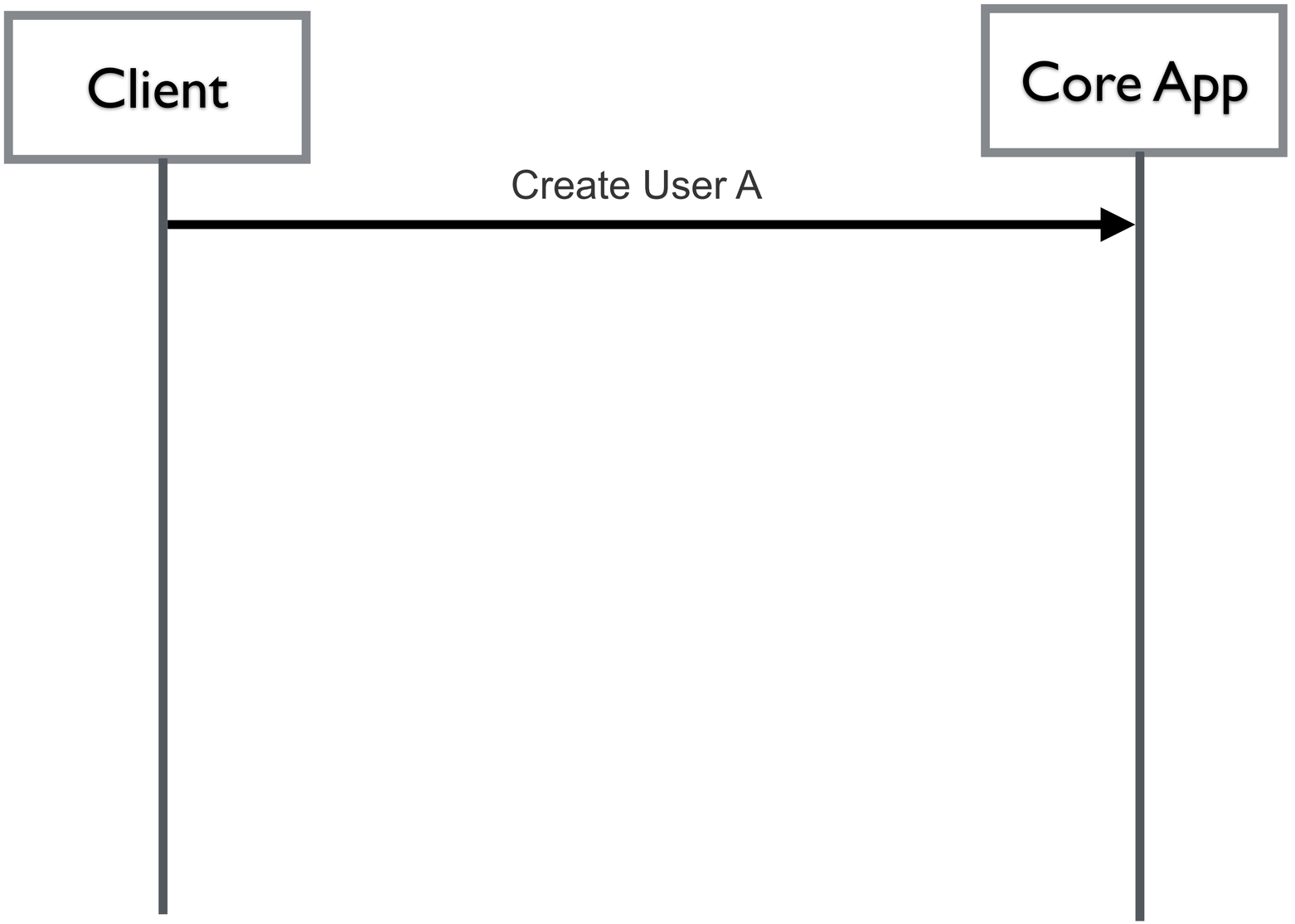
User 4: Seq 23

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

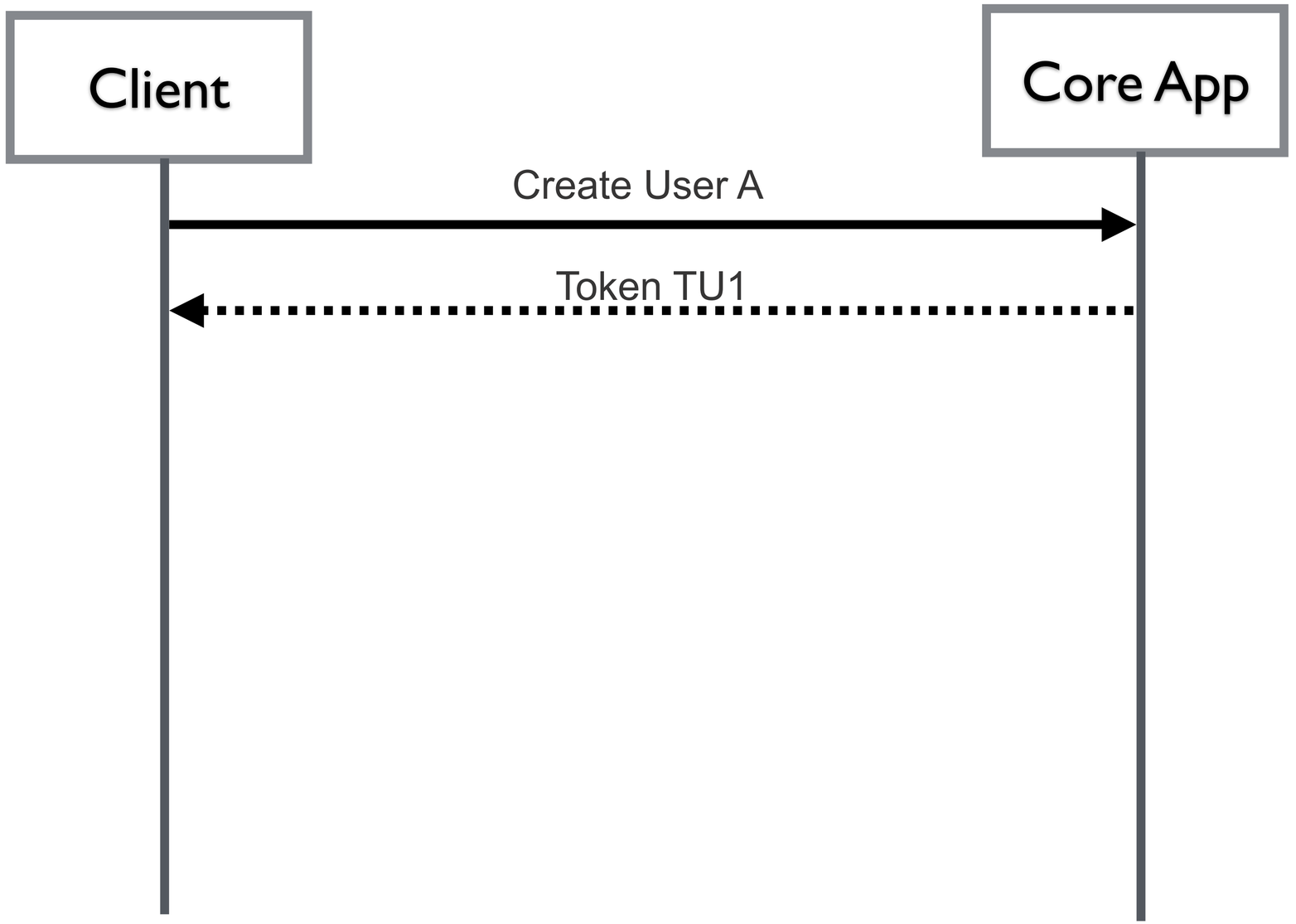
Using tokens to enforce state



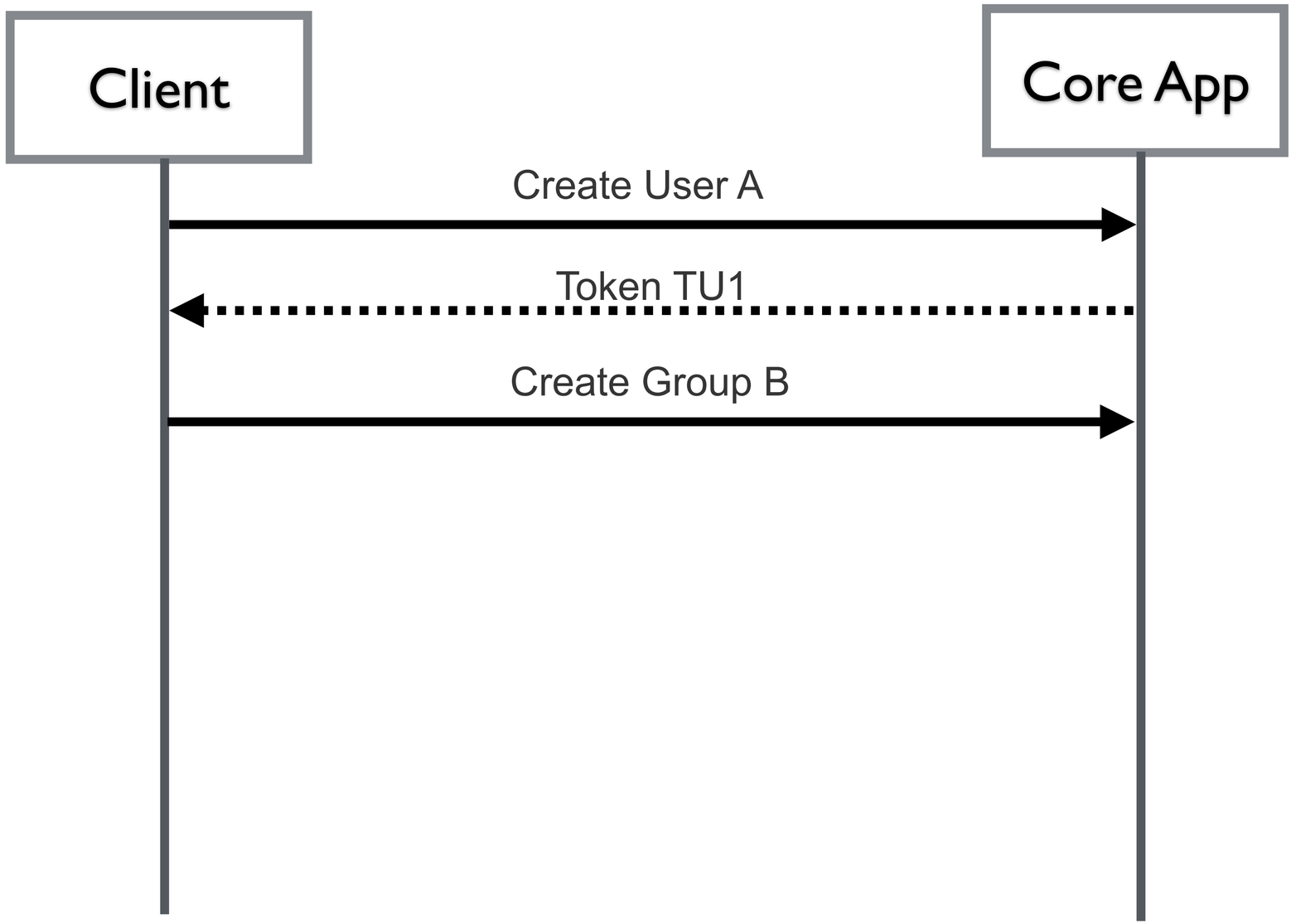
Using tokens to enforce state



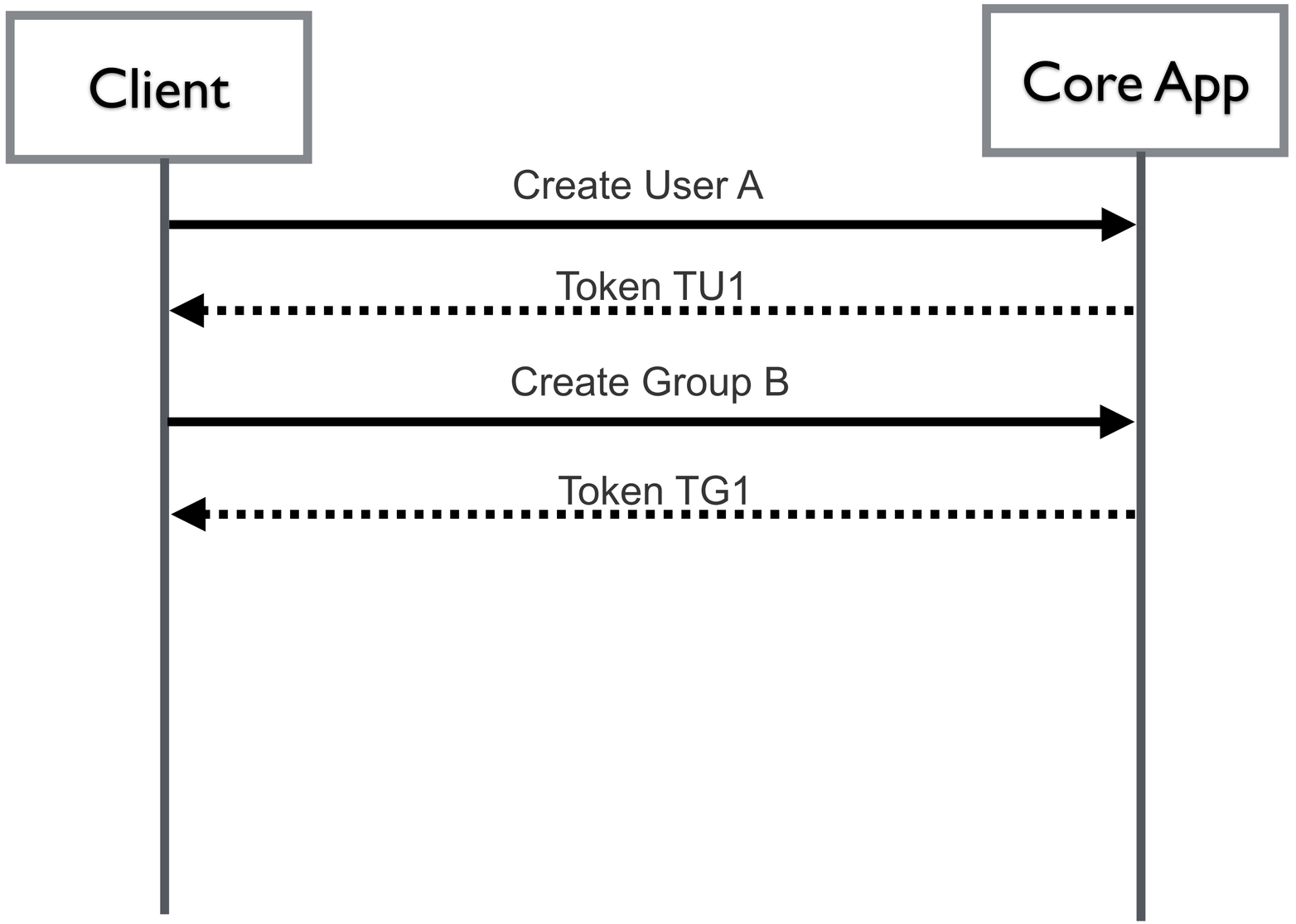
Using tokens to enforce state



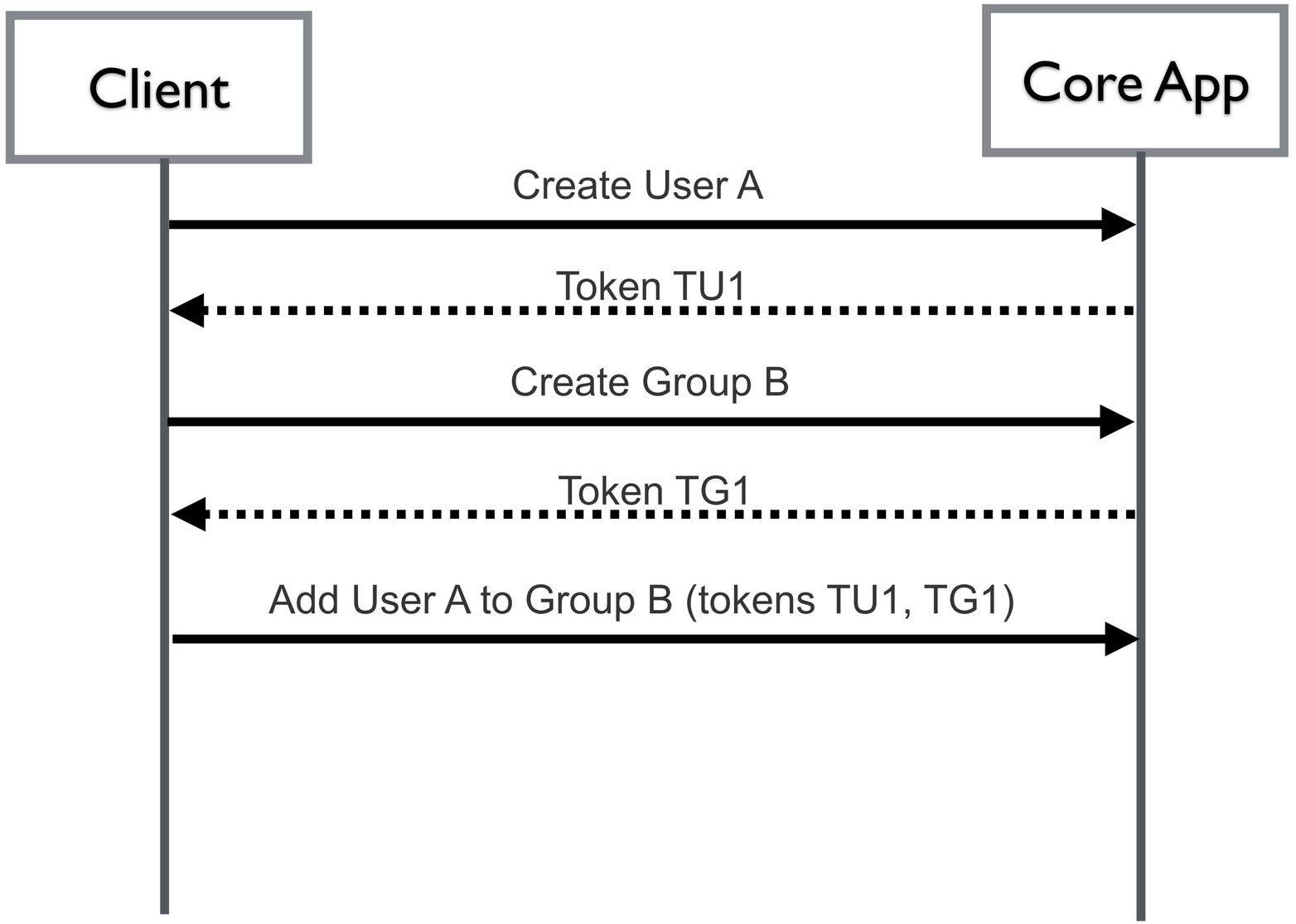
Using tokens to enforce state



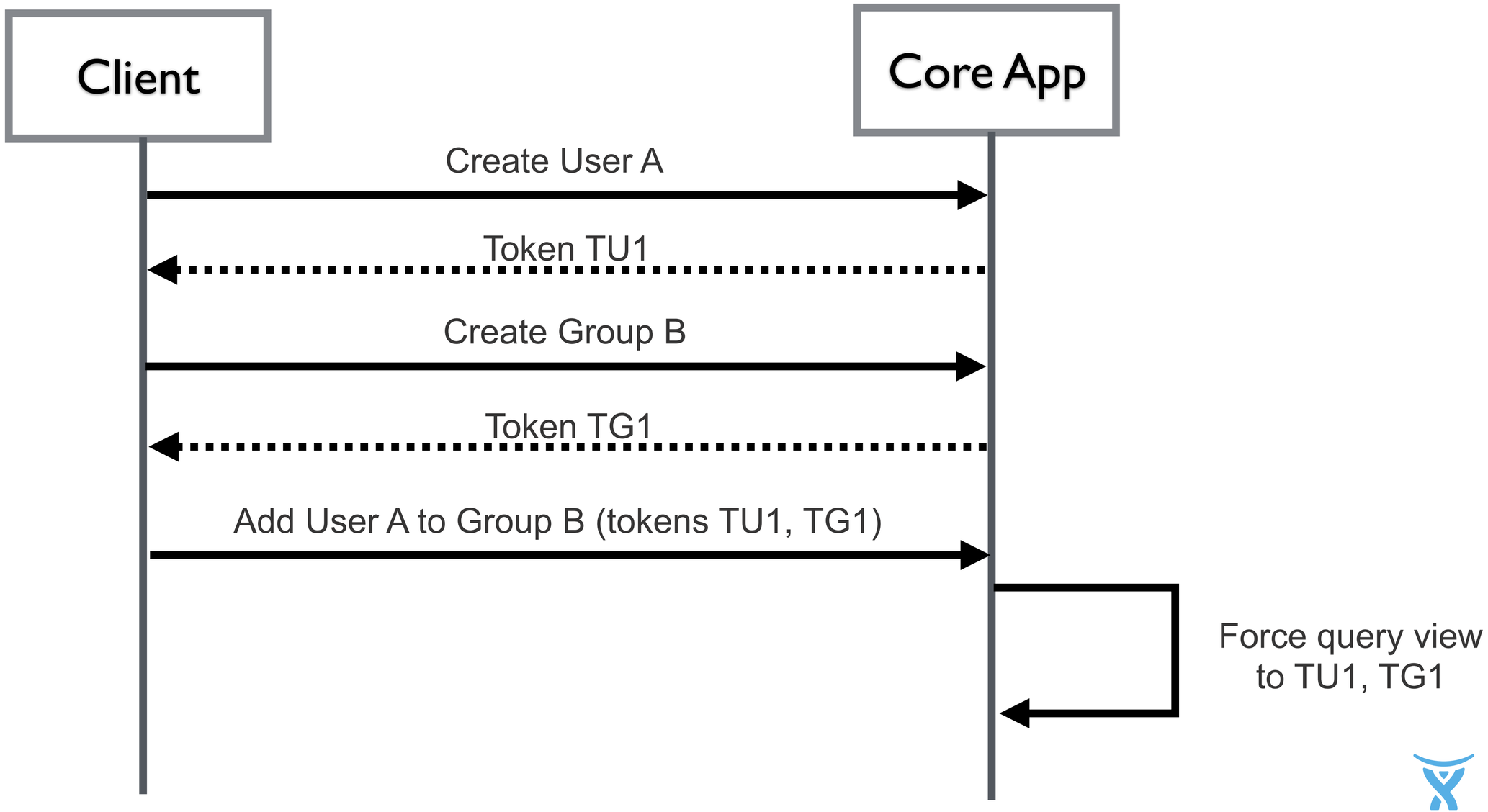
Using tokens to enforce state



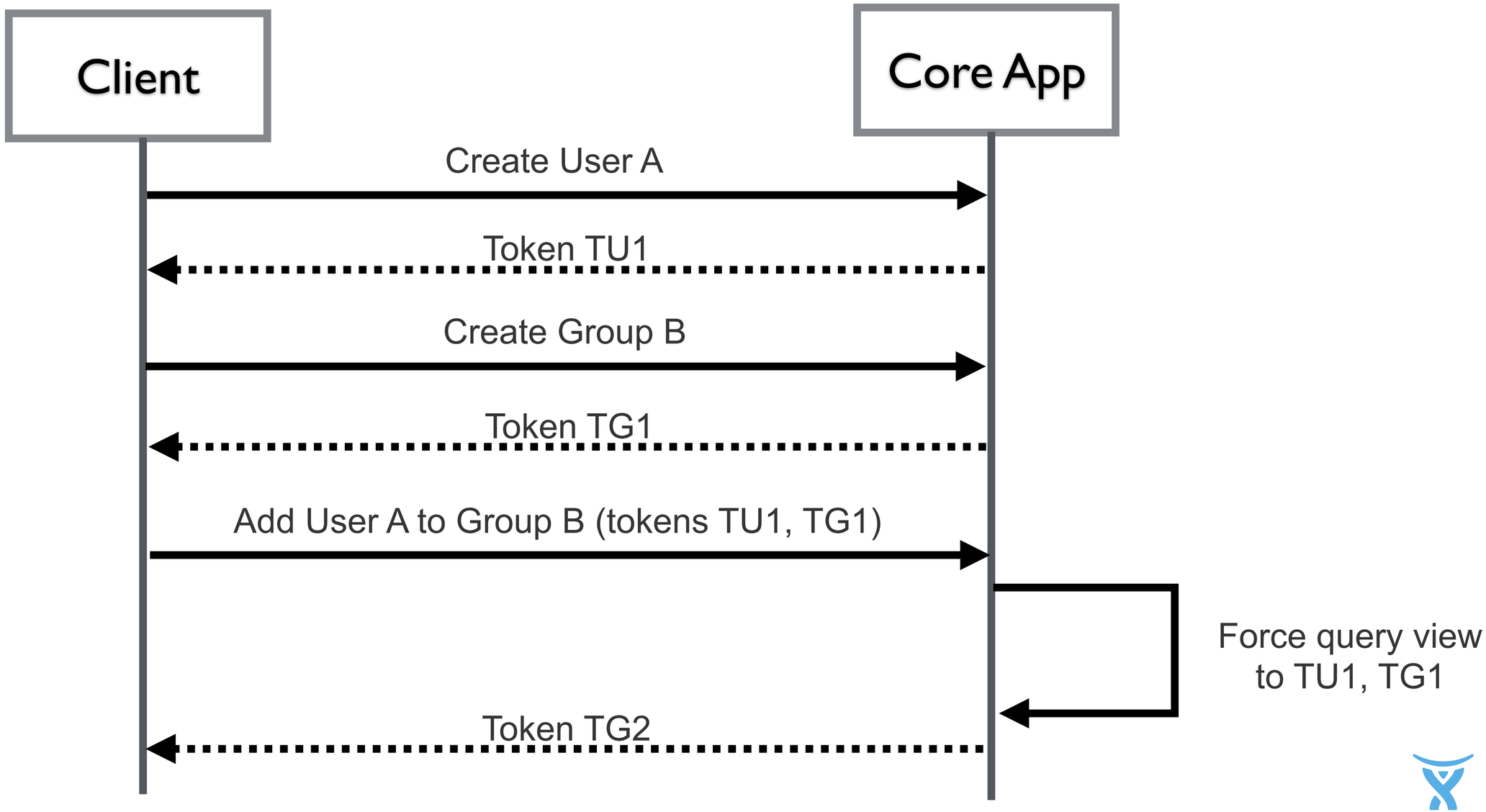
Using tokens to enforce state



Using tokens to enforce state



Using tokens to enforce state



Principle 3.5: 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 glitches

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 glitches

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” - small idempotent events

Key takeaways

- Start small and challenge everything!
- Incremental architecture for incremental demos
- Think “Events as an API” - small idempotent events
- Split streams for reasonable latency vs consistency

Key takeaways

- Start small and challenge everything!
- Incremental architecture for incremental demos
- Think “Events as an API” - small idempotent events
- Split streams for reasonable latency vs consistency
- 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

aws-scala lib:

http://bitbucket.org/atlassian/aws-scala

SCALA DAYS BERLIN 2016

Please

Remember to rate this session

Thank you!

Join the conversation #scaladays


SCALA DAYS
BERLIN 2016