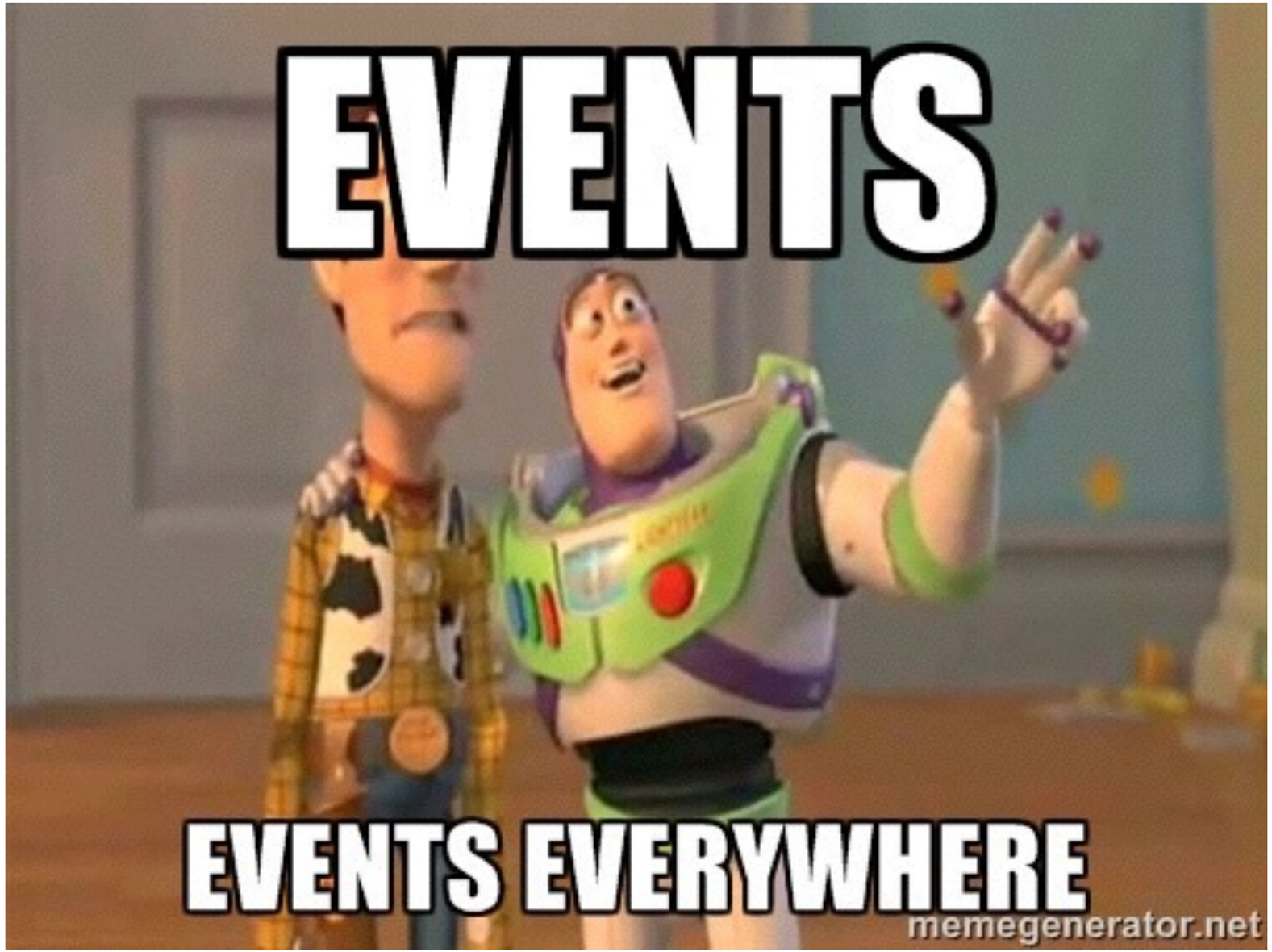
Atlassian

Event sourcing and CQRS from the trenches

SIDNEY SHEK · ARCHITECT · ATLASSIAN · @SIDNEYSHEK


EVENTS
EVENTS EVERYWHERE
memegenerator.net

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

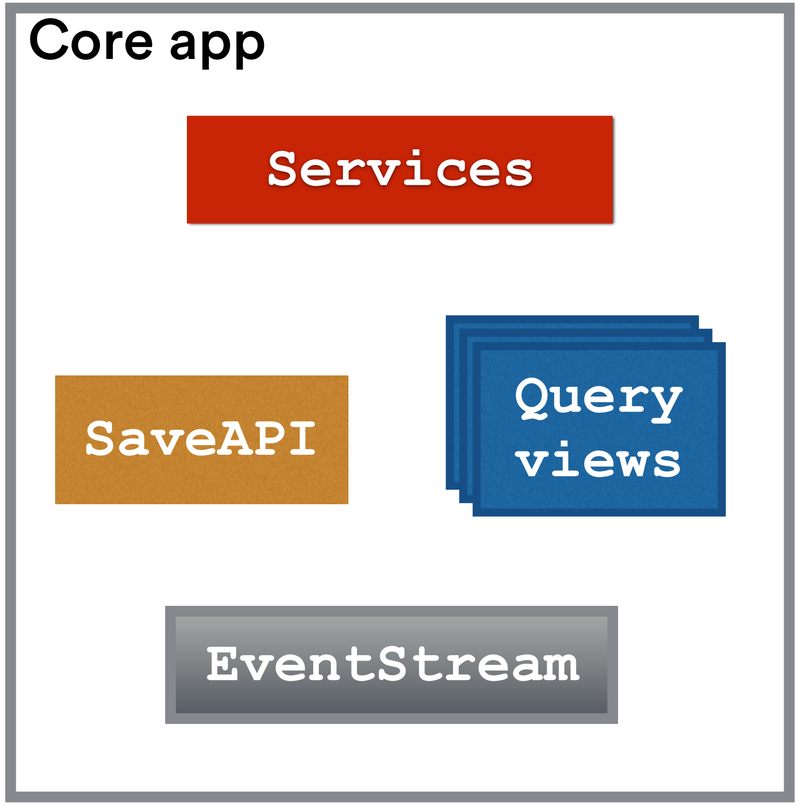
Our Identity System requirements

- 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

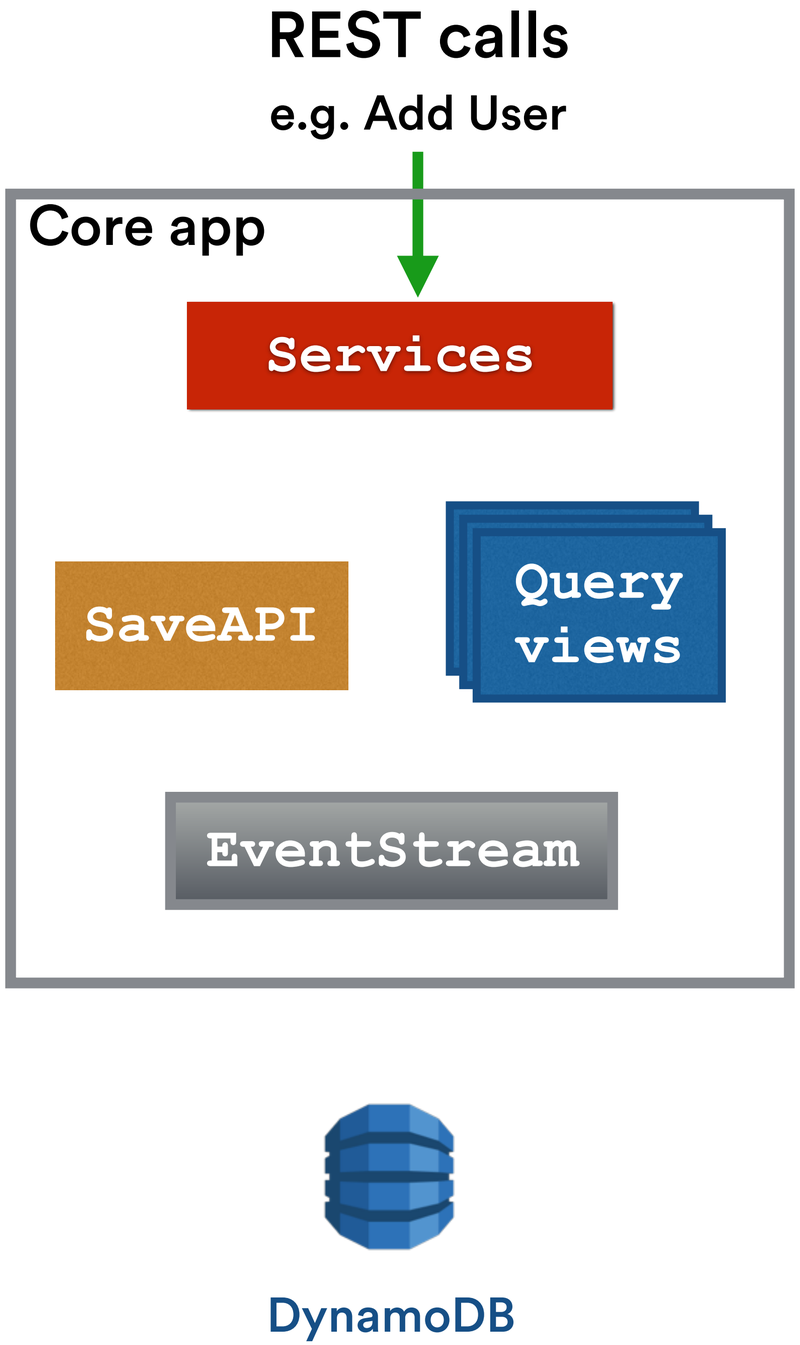
Evolving the architecture

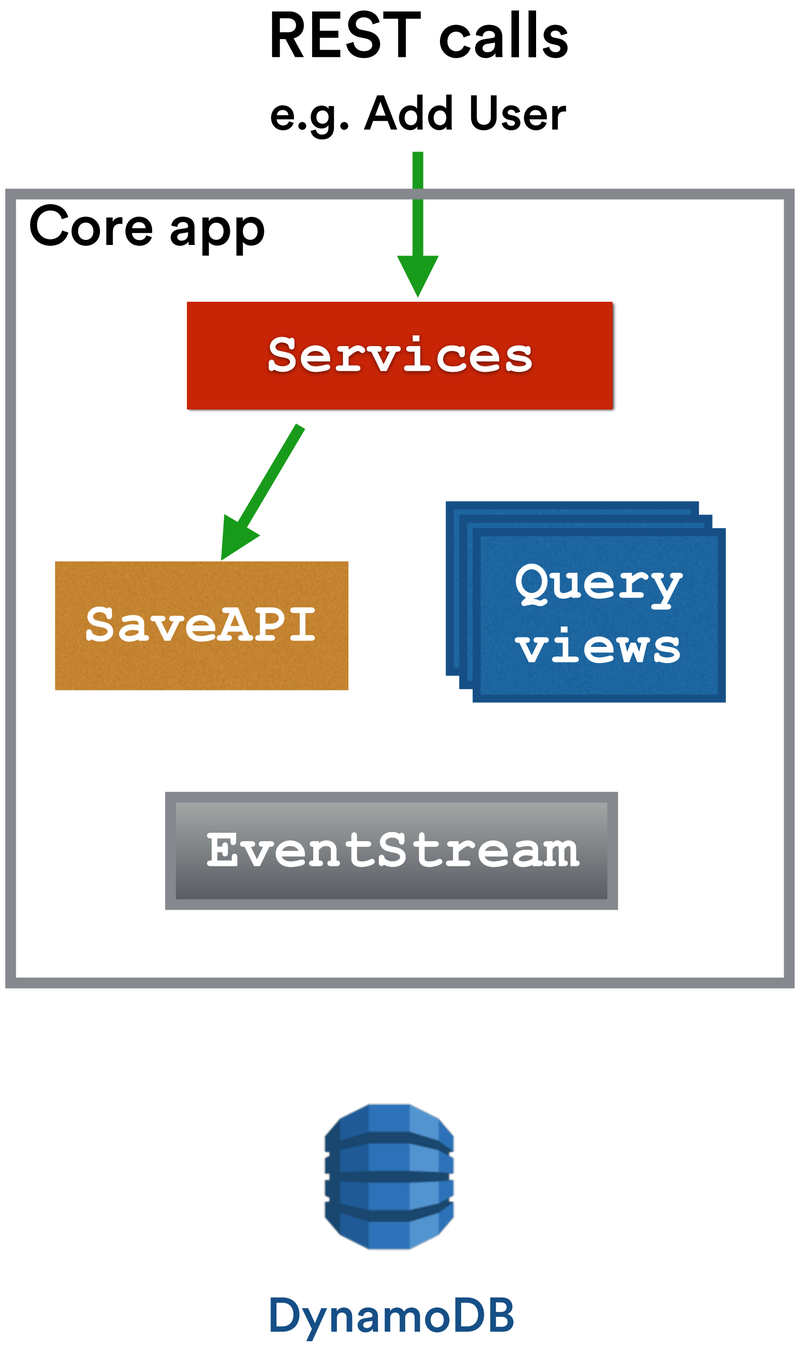
REST calls

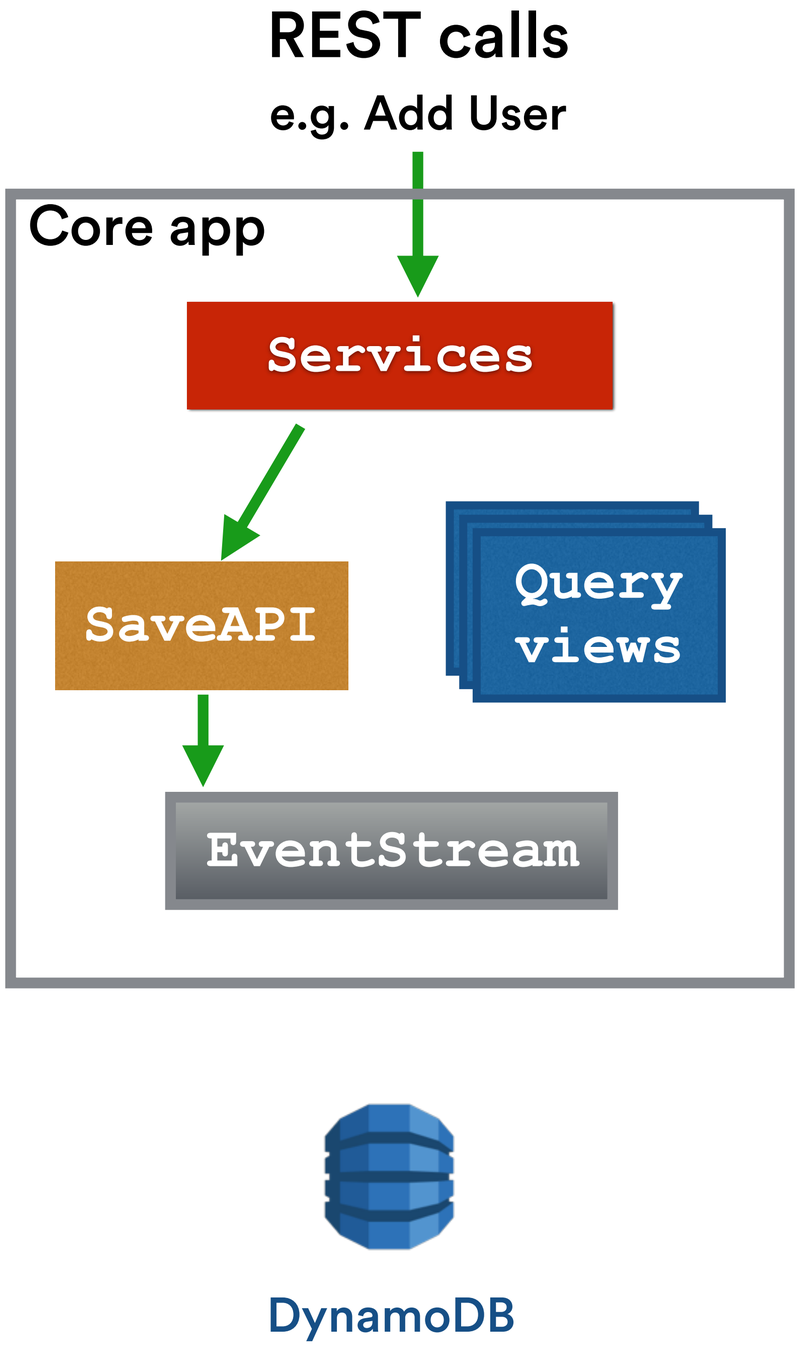
e.g. Add User

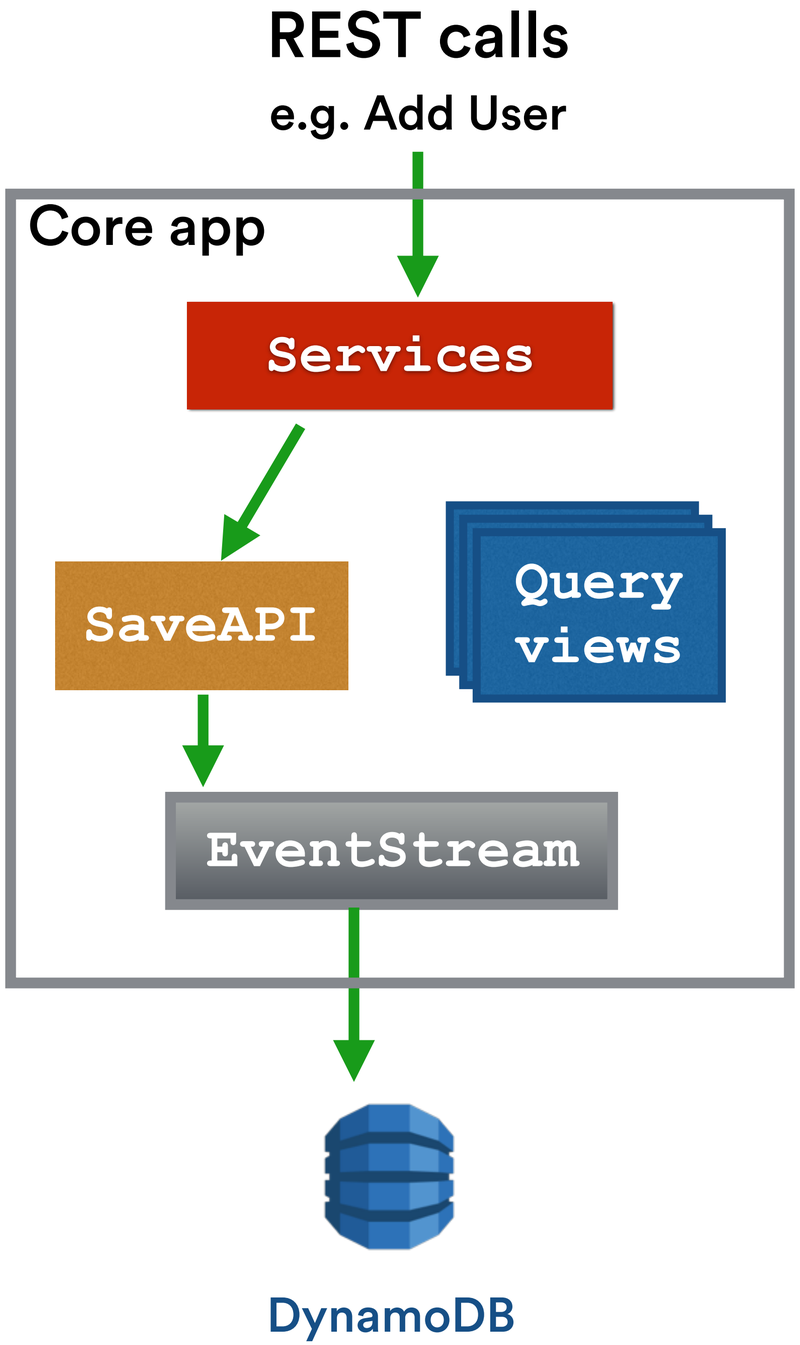


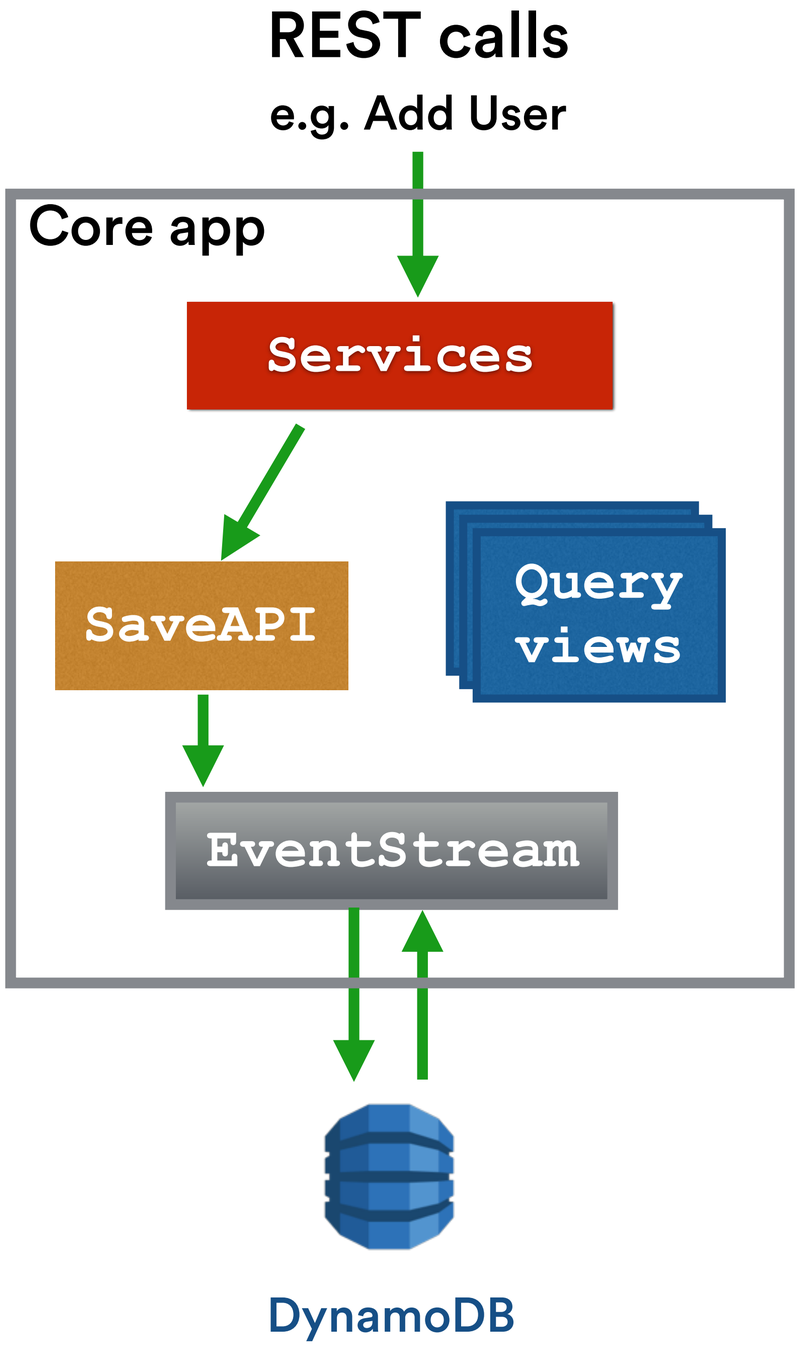
DynamoDB

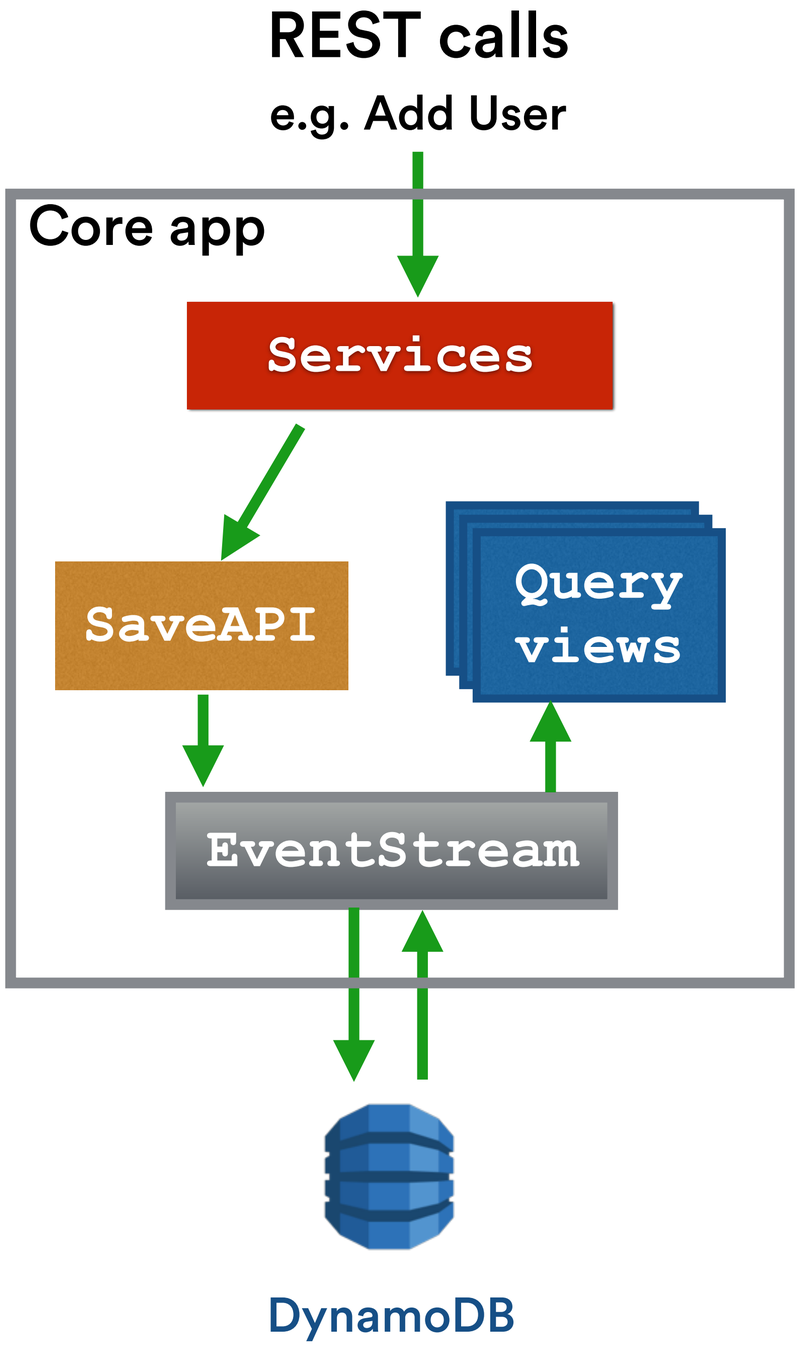

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

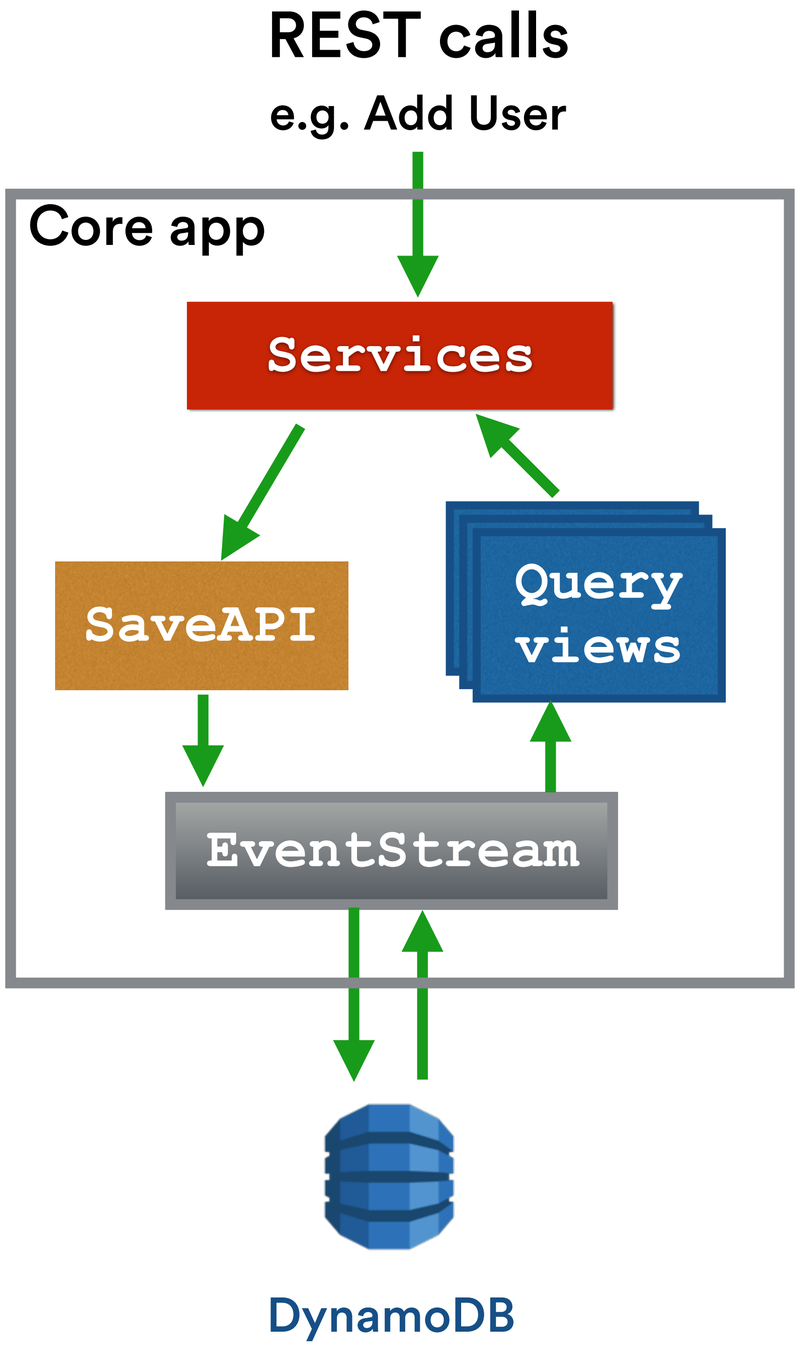

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

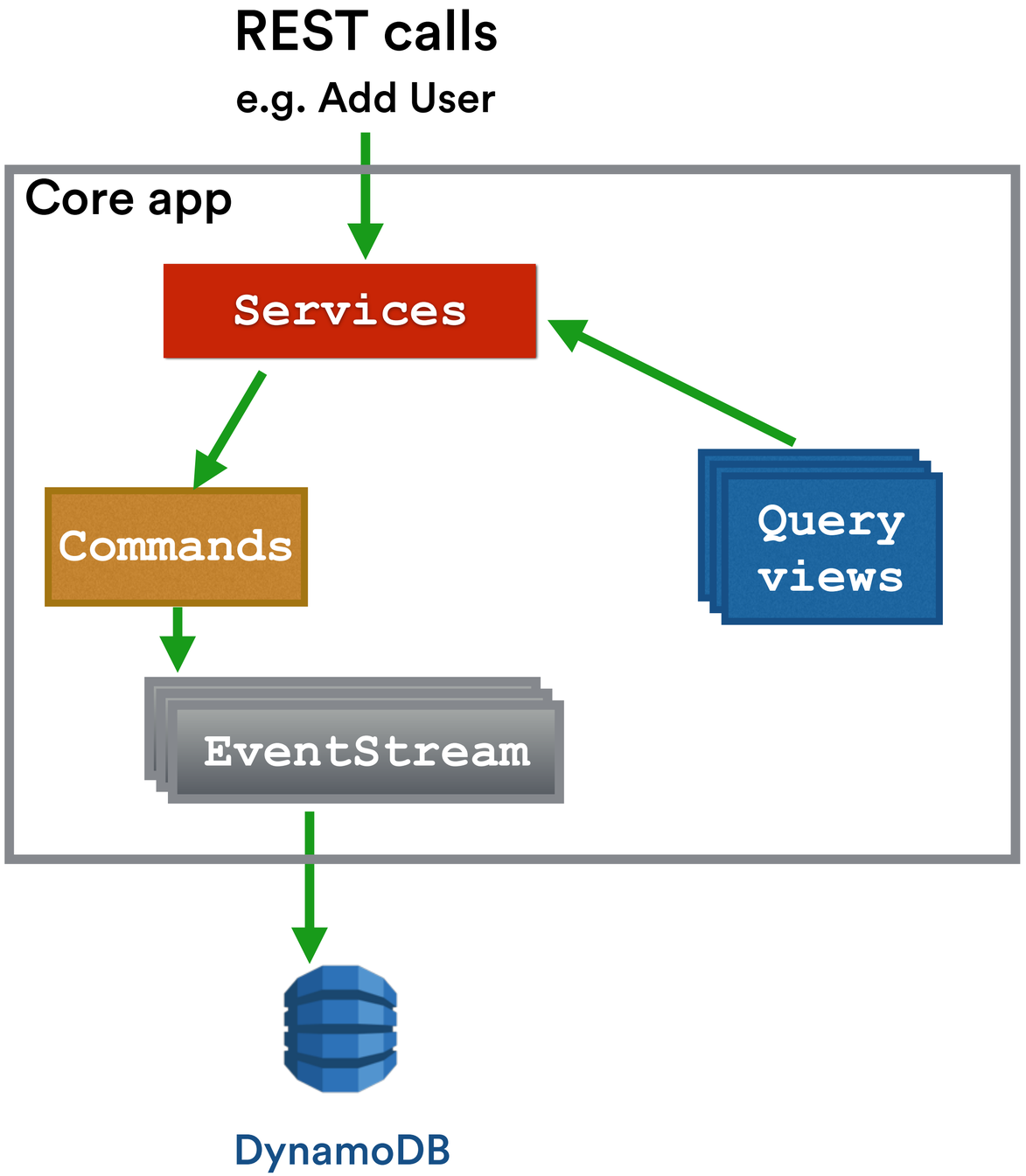

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

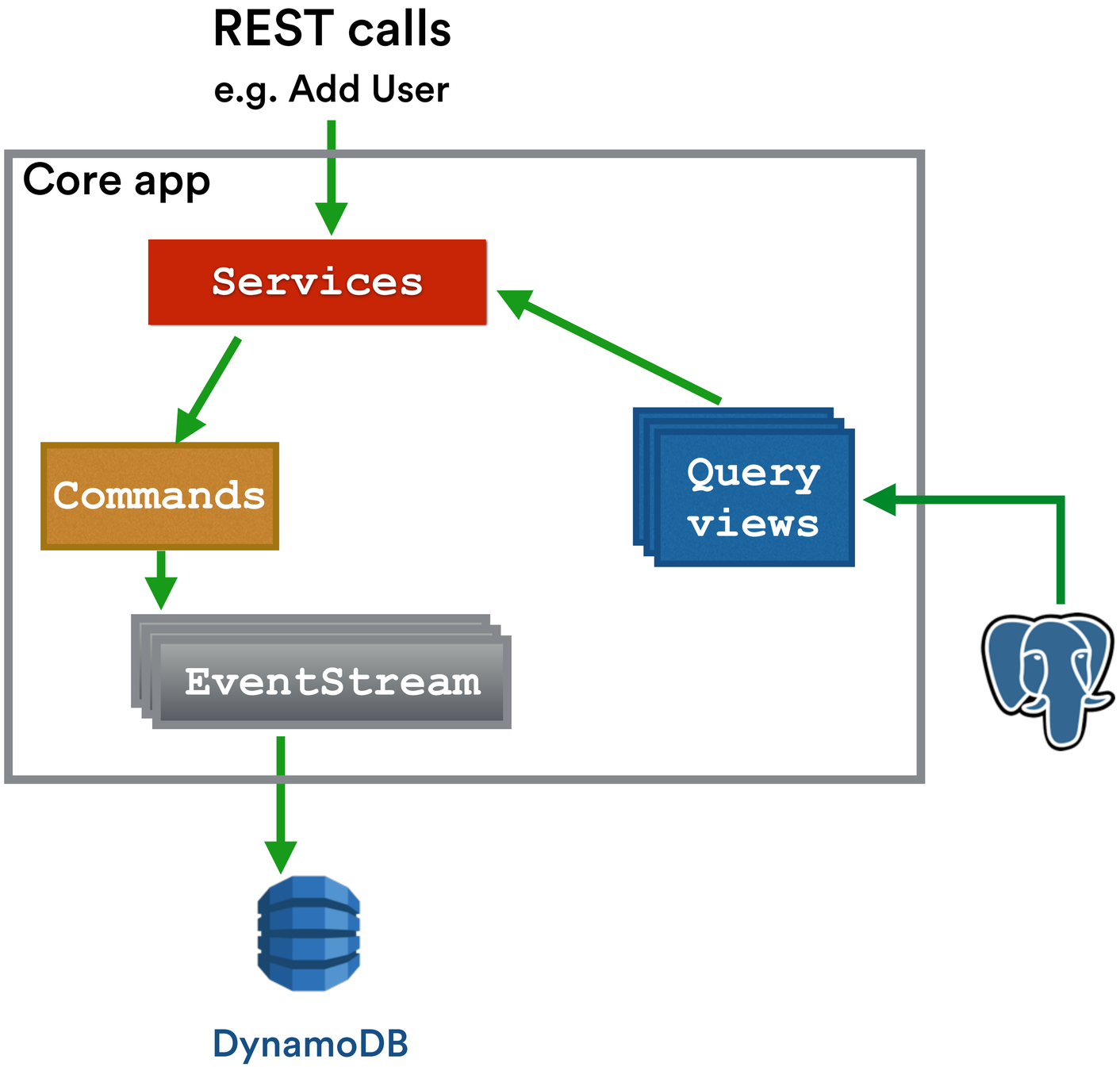

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

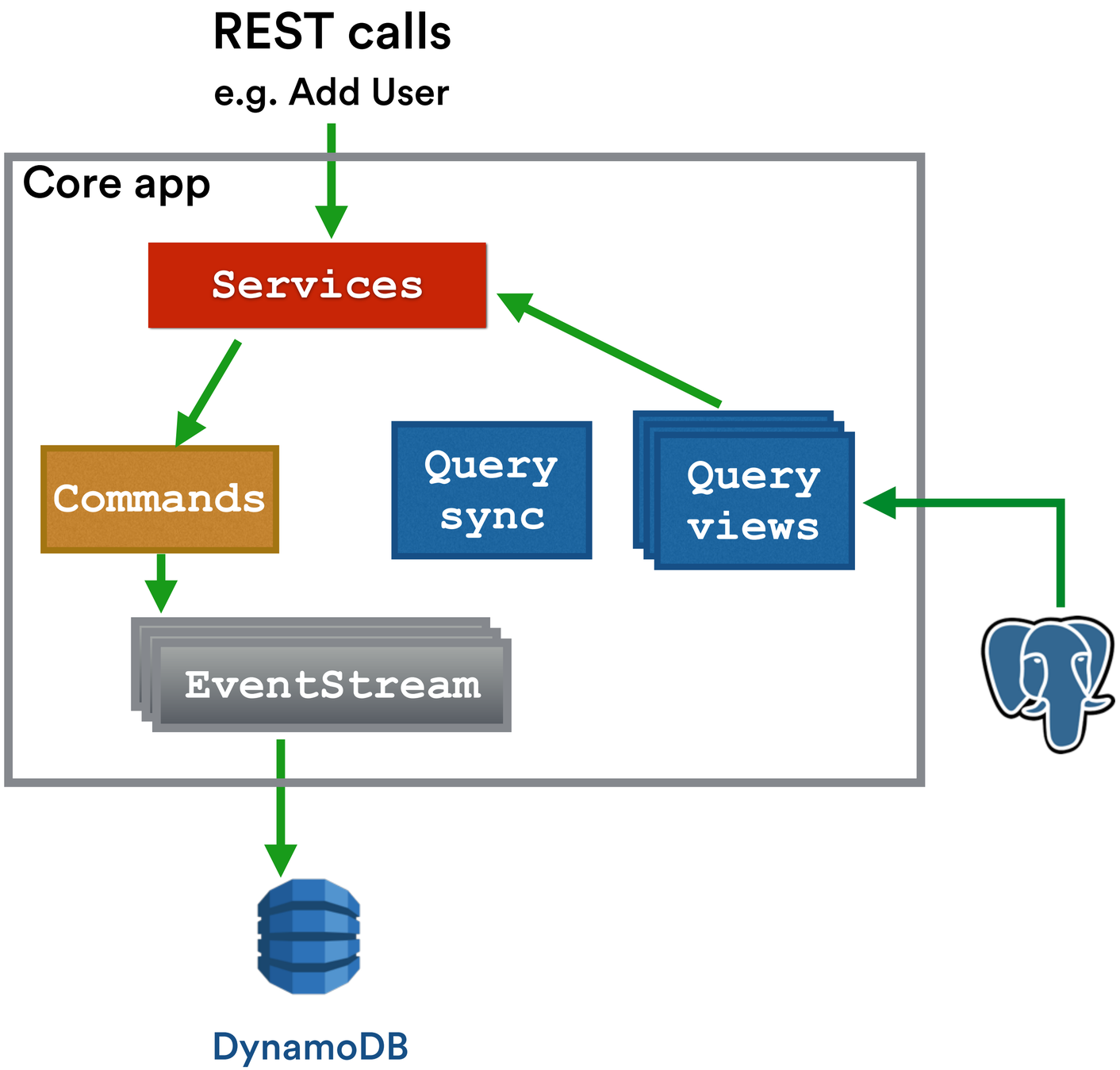

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

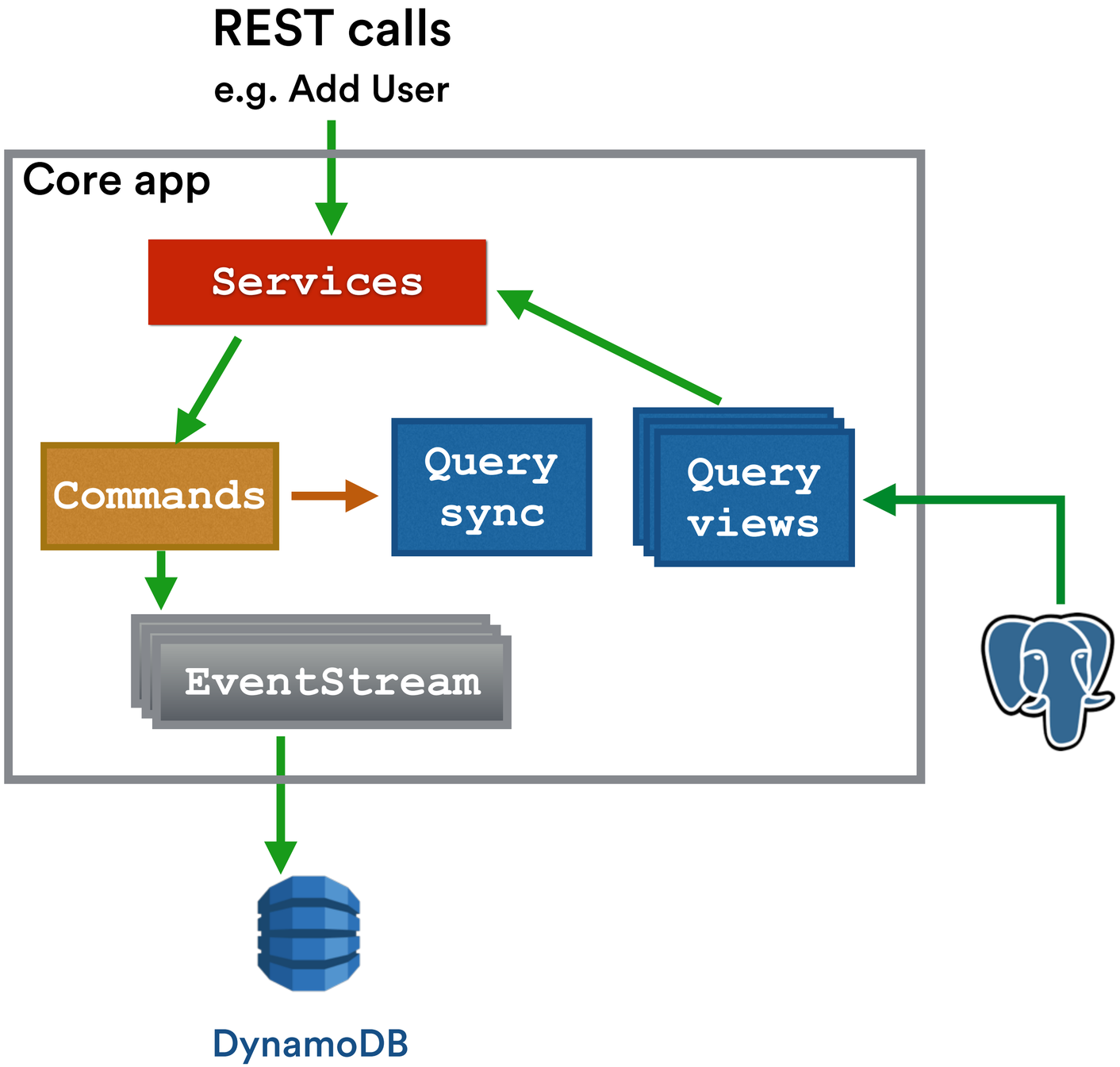

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

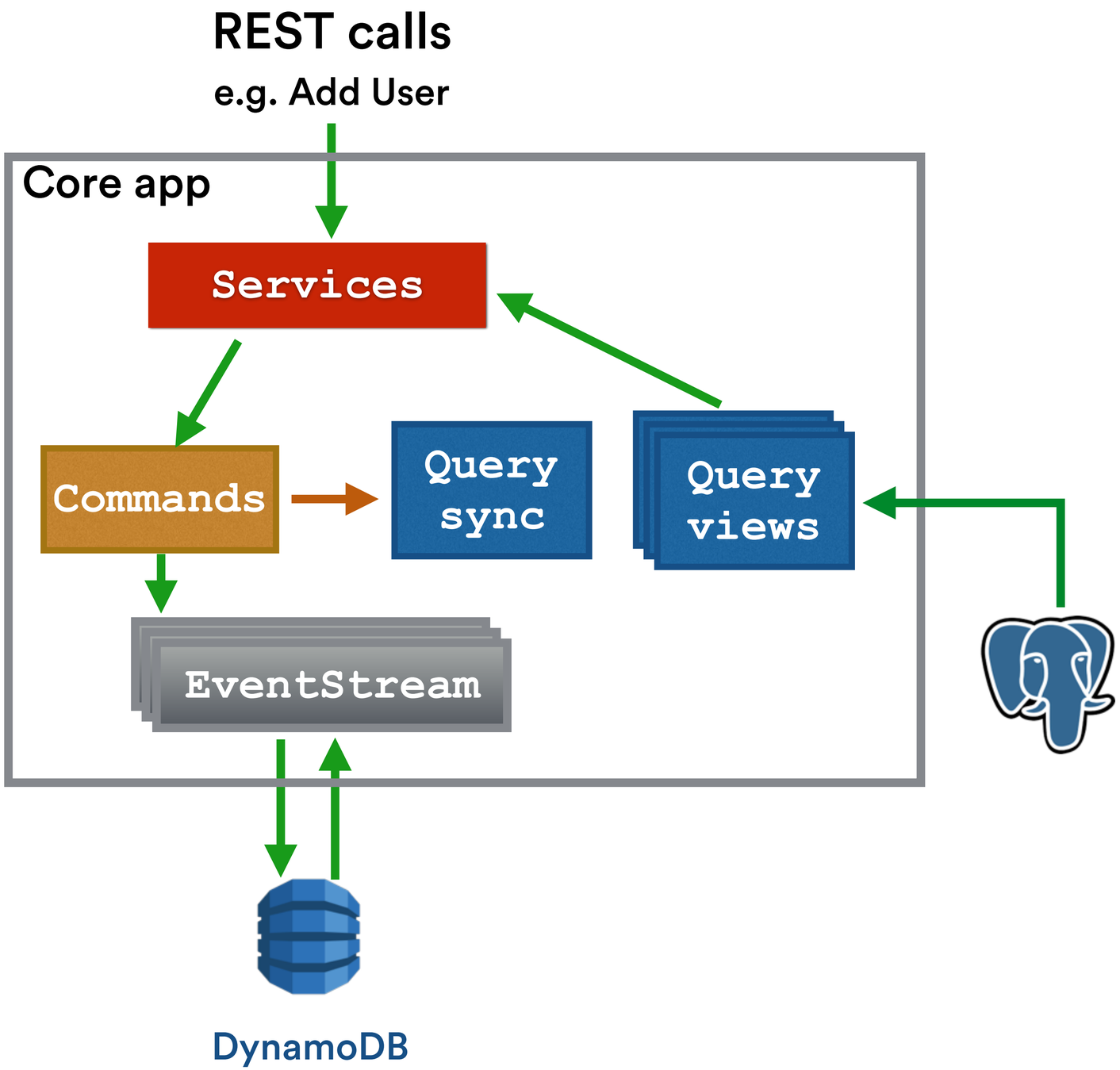

REST calls
e.g. Add User
Core app
Services
SaveAPI
Query views
EventStream
DynamoDB

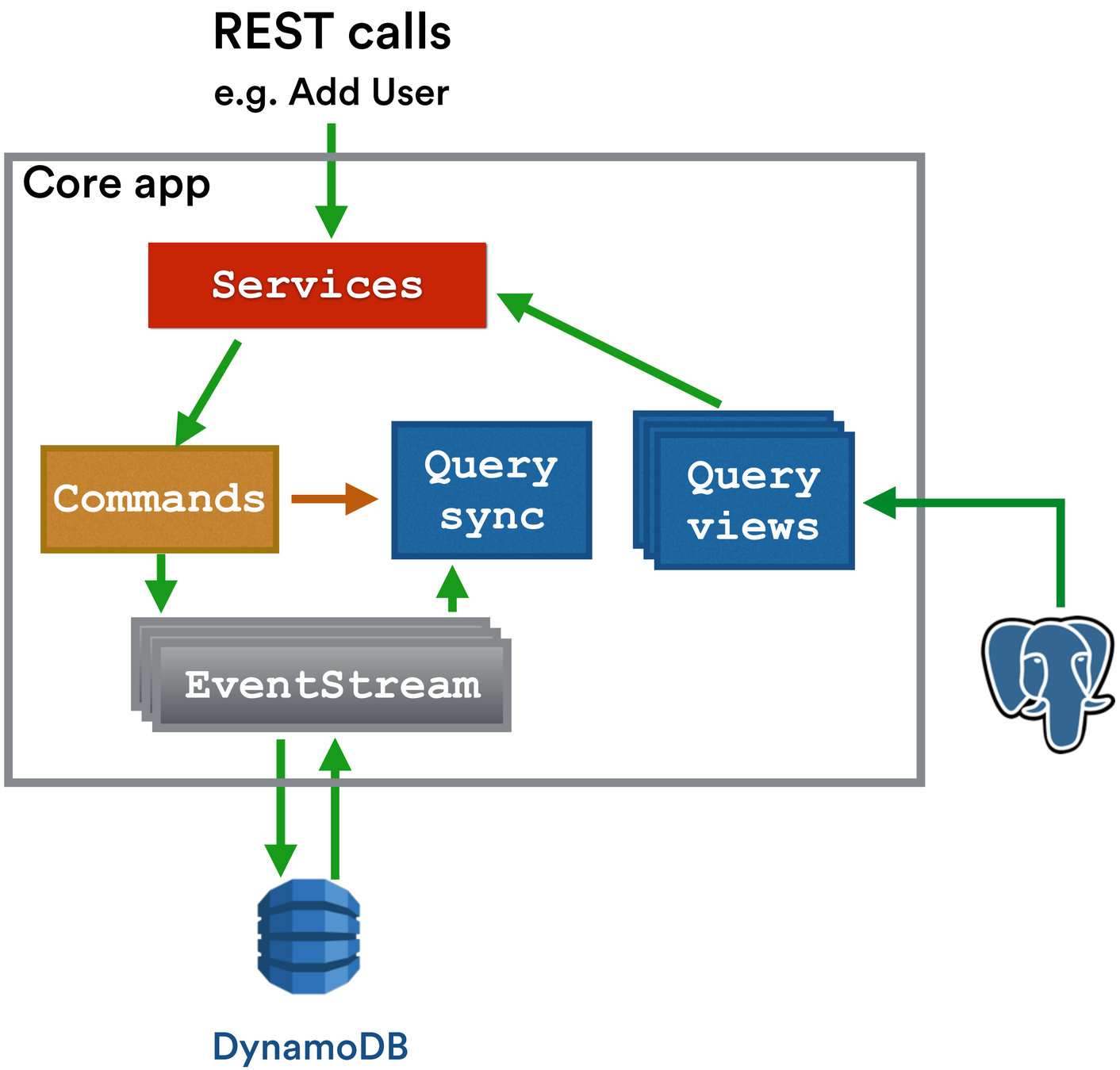

REST calls
e.g. Add User
Core app
Services
Commands
Query views
EventStream
DynamoDB

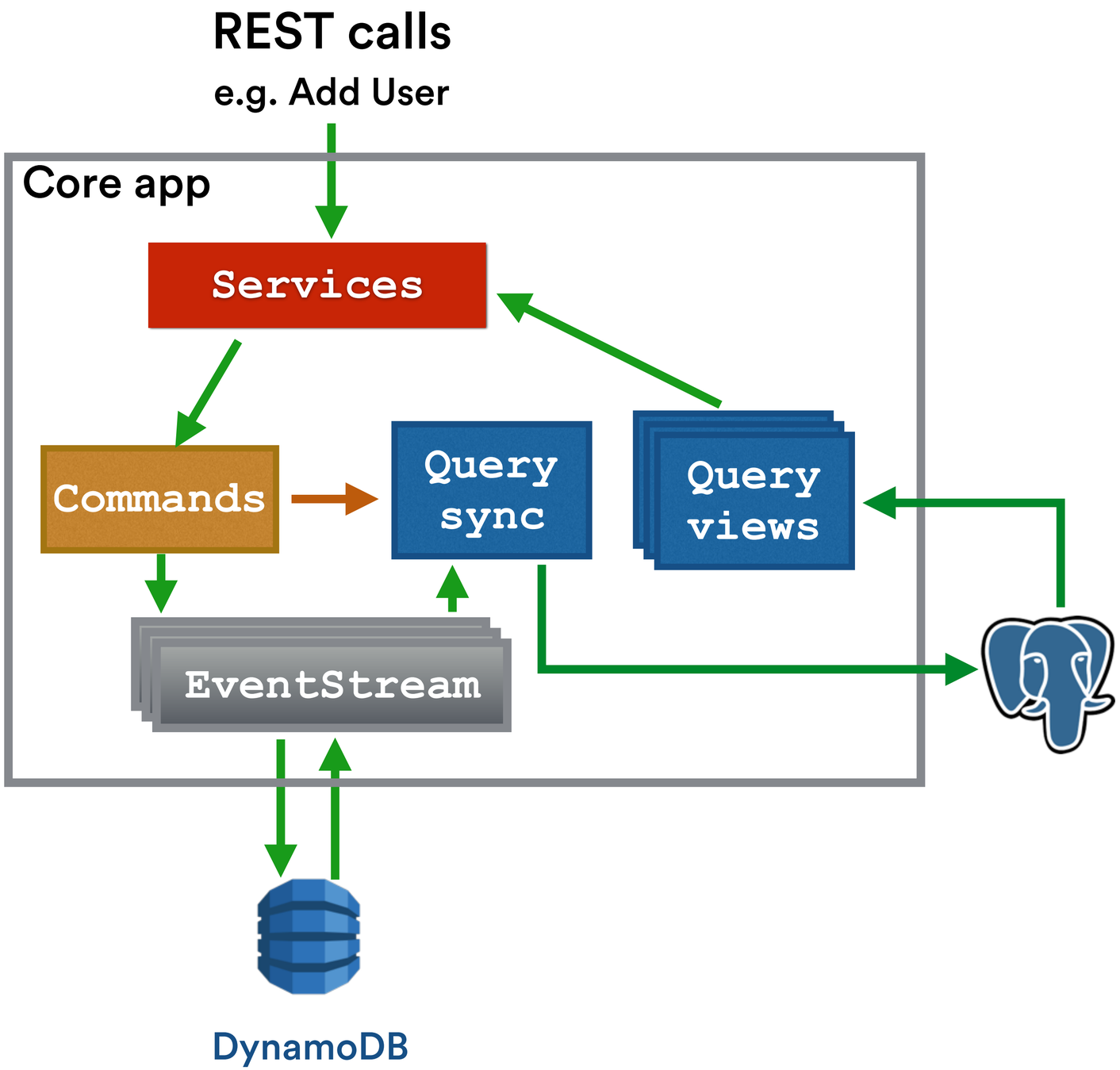

REST calls
e.g. Add User
Core app
Services
Commands
EventStream
Query views
DynamoDB

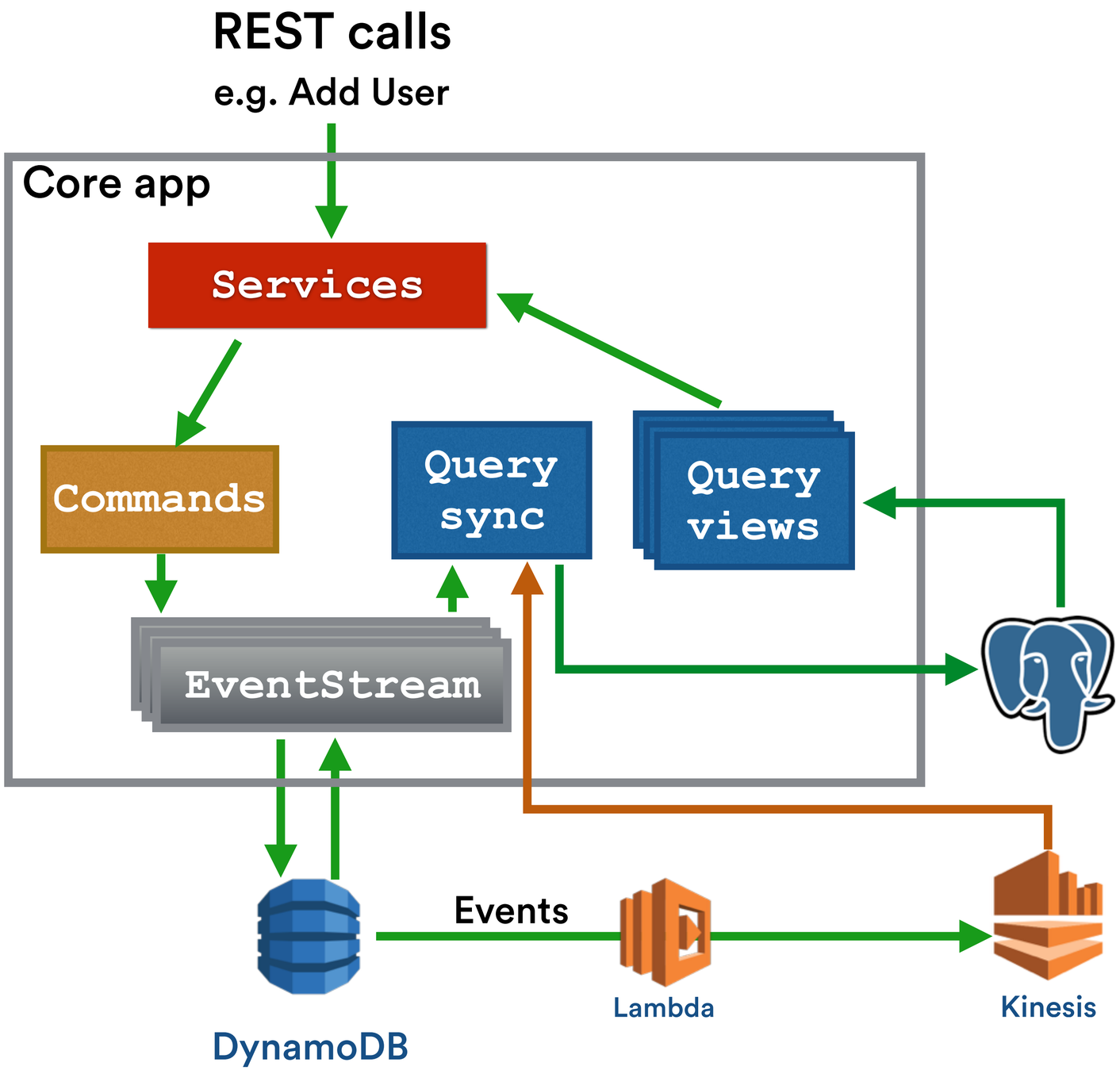

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

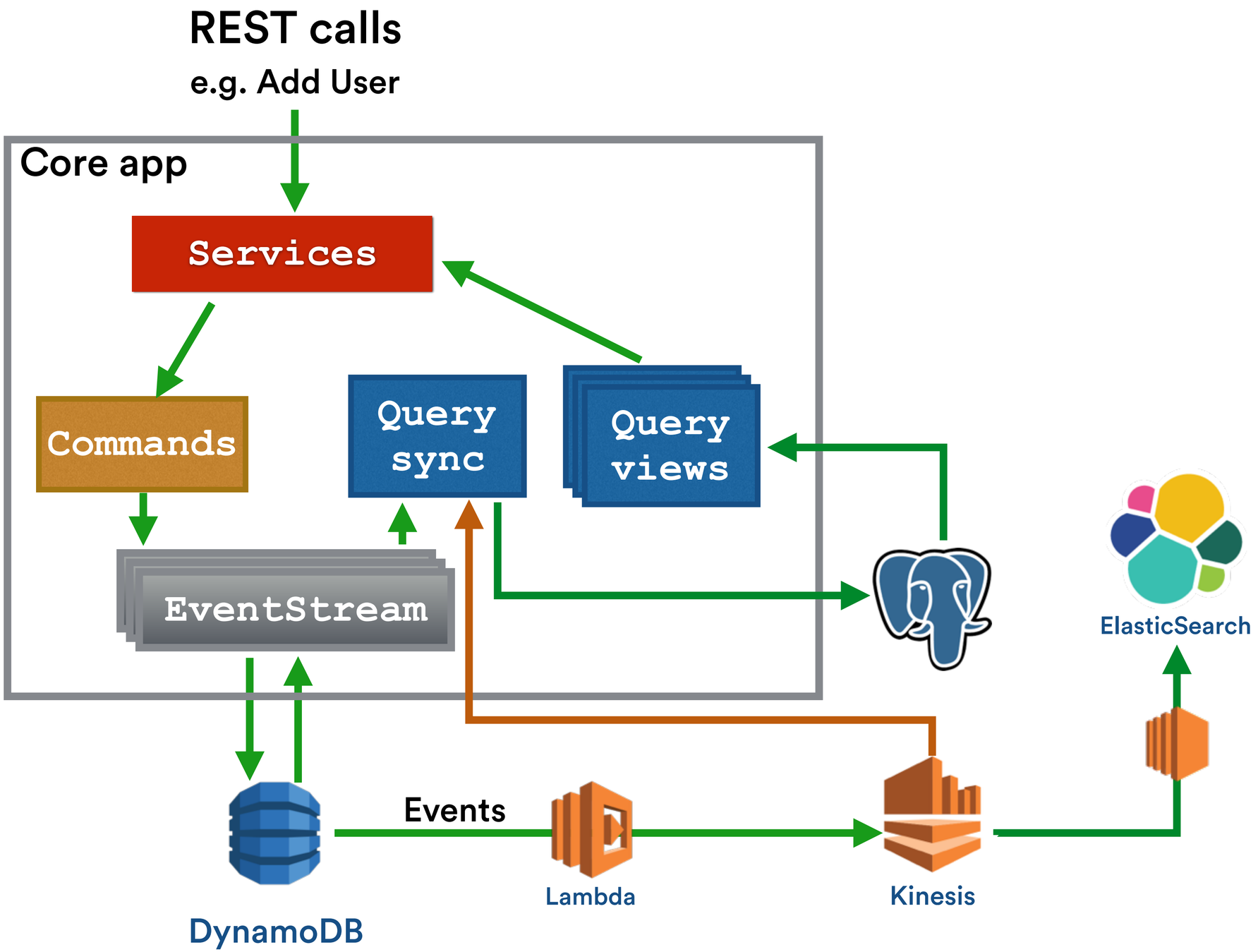

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

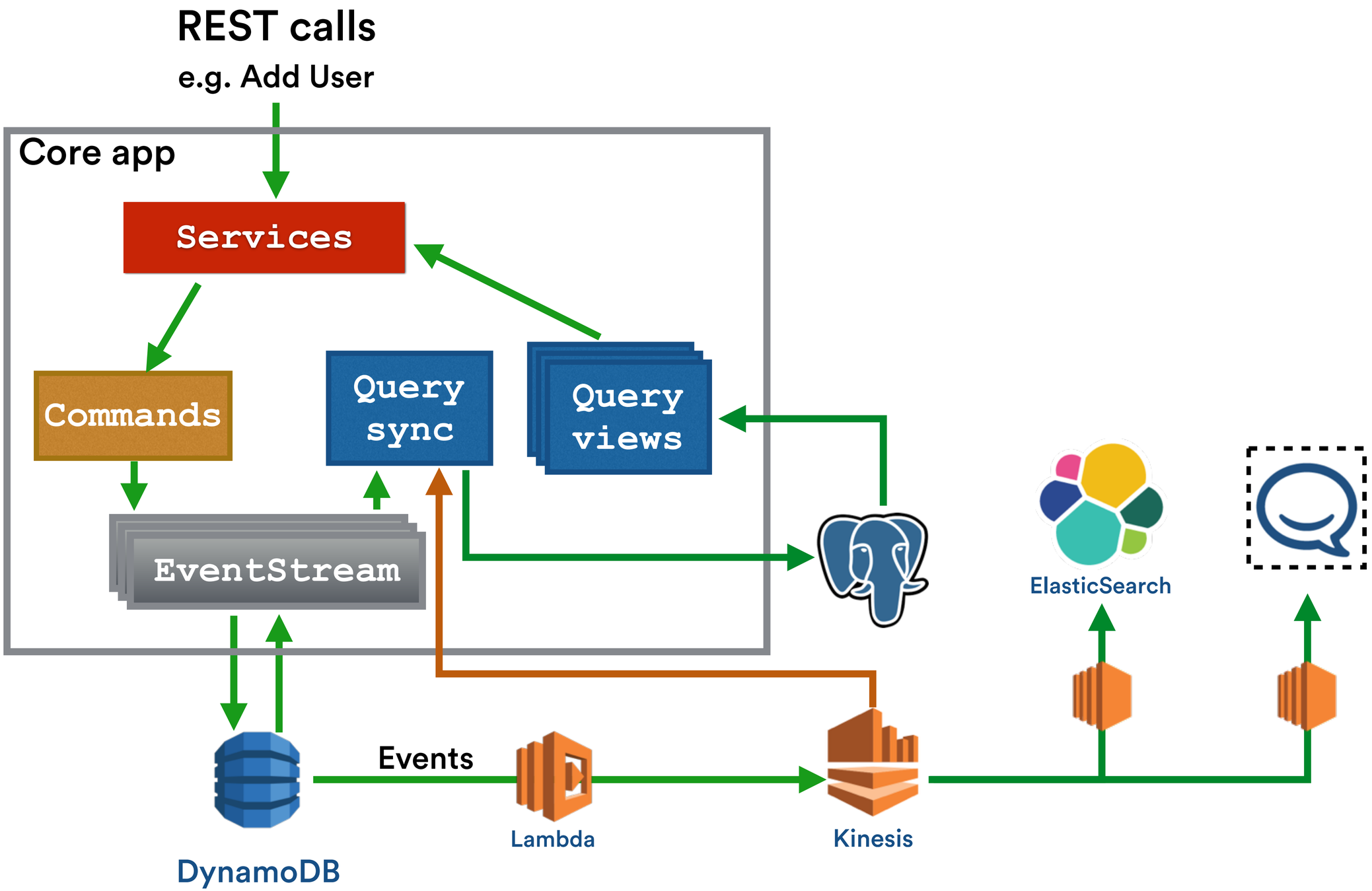

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

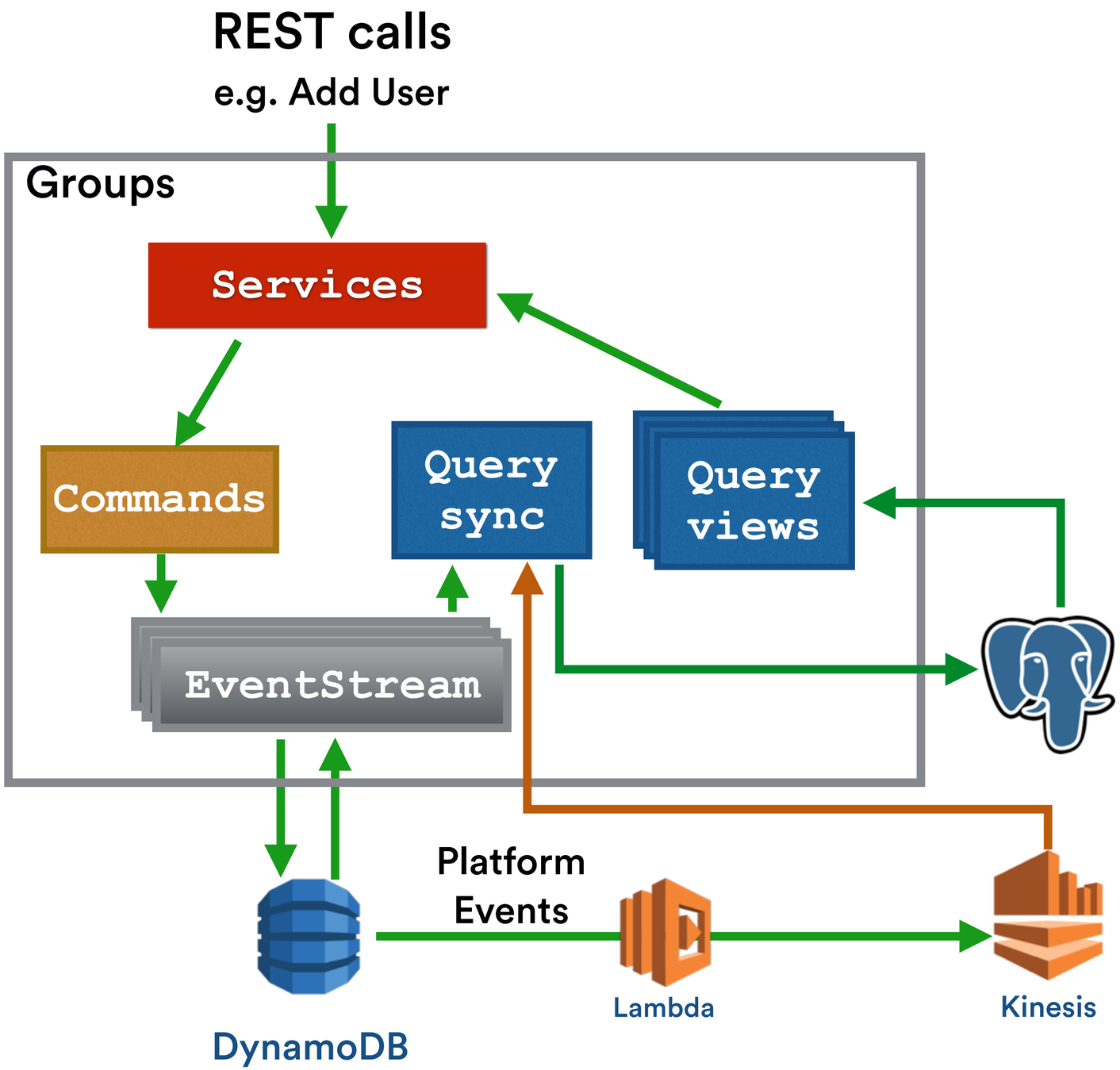

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

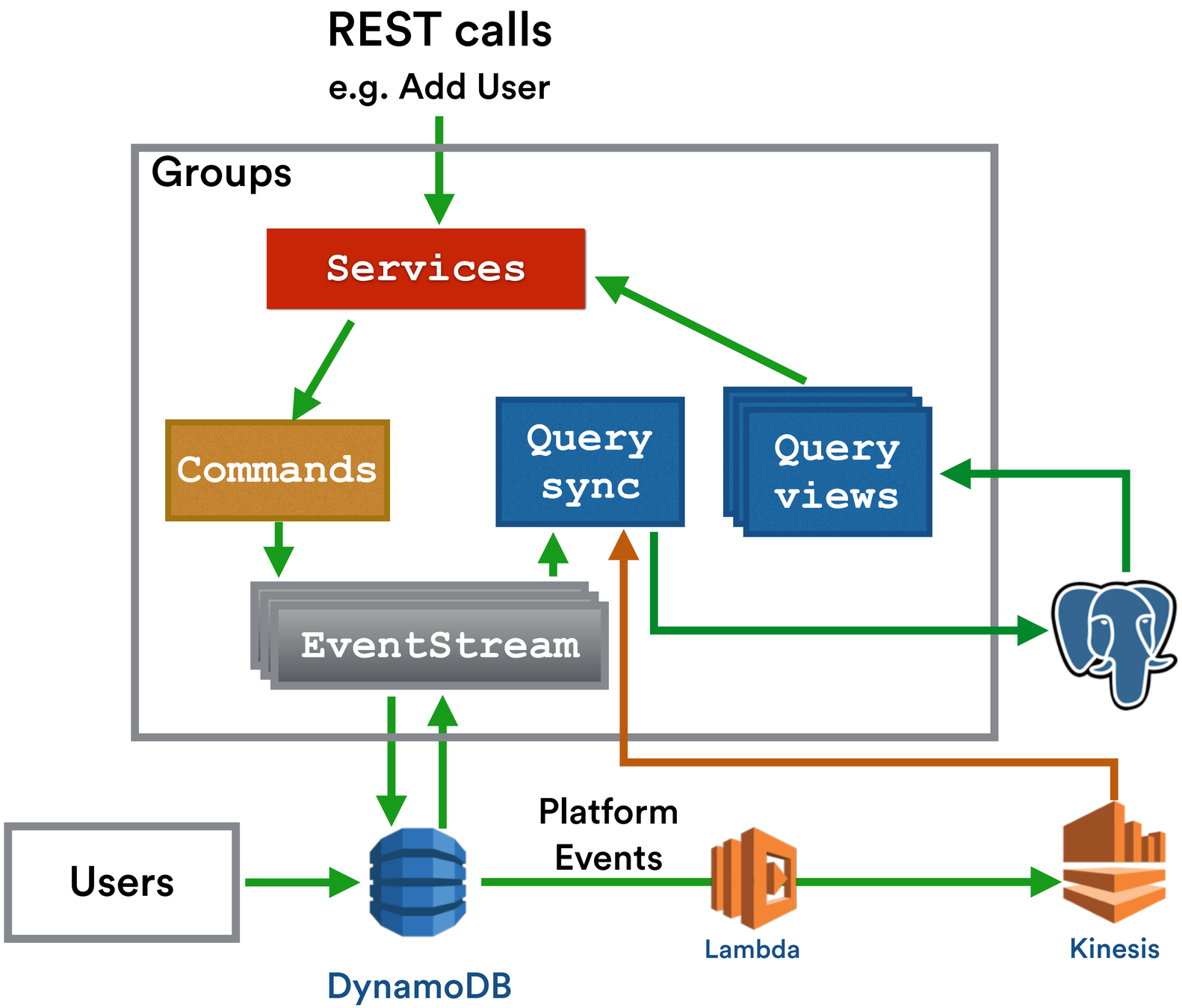

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
DynamoDB

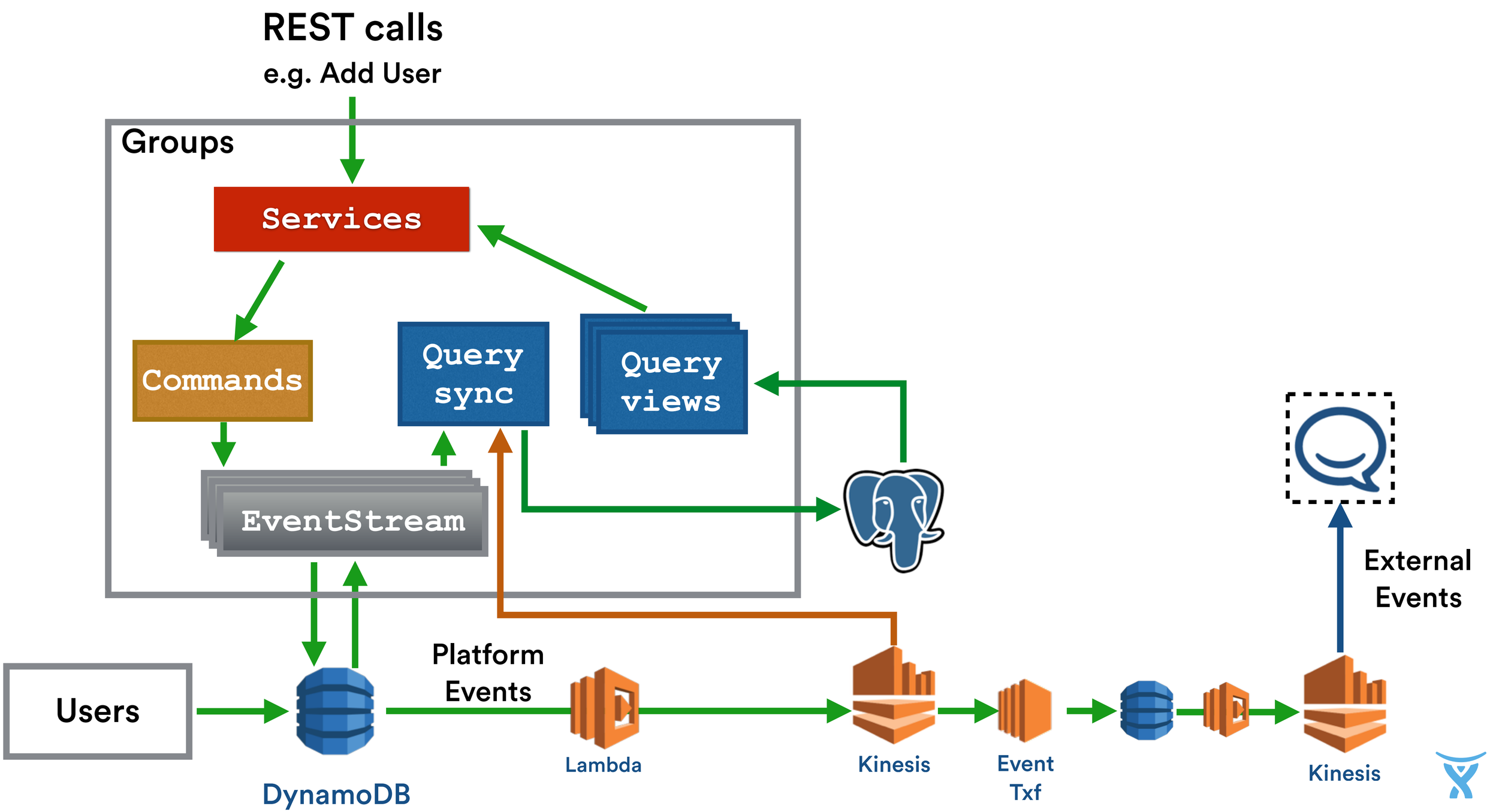

REST calls
e.g. Add User
Core app
Services
Commands
Query sync
Query views
EventStream
Events
Lambda
Kinesis
DynamoDB


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

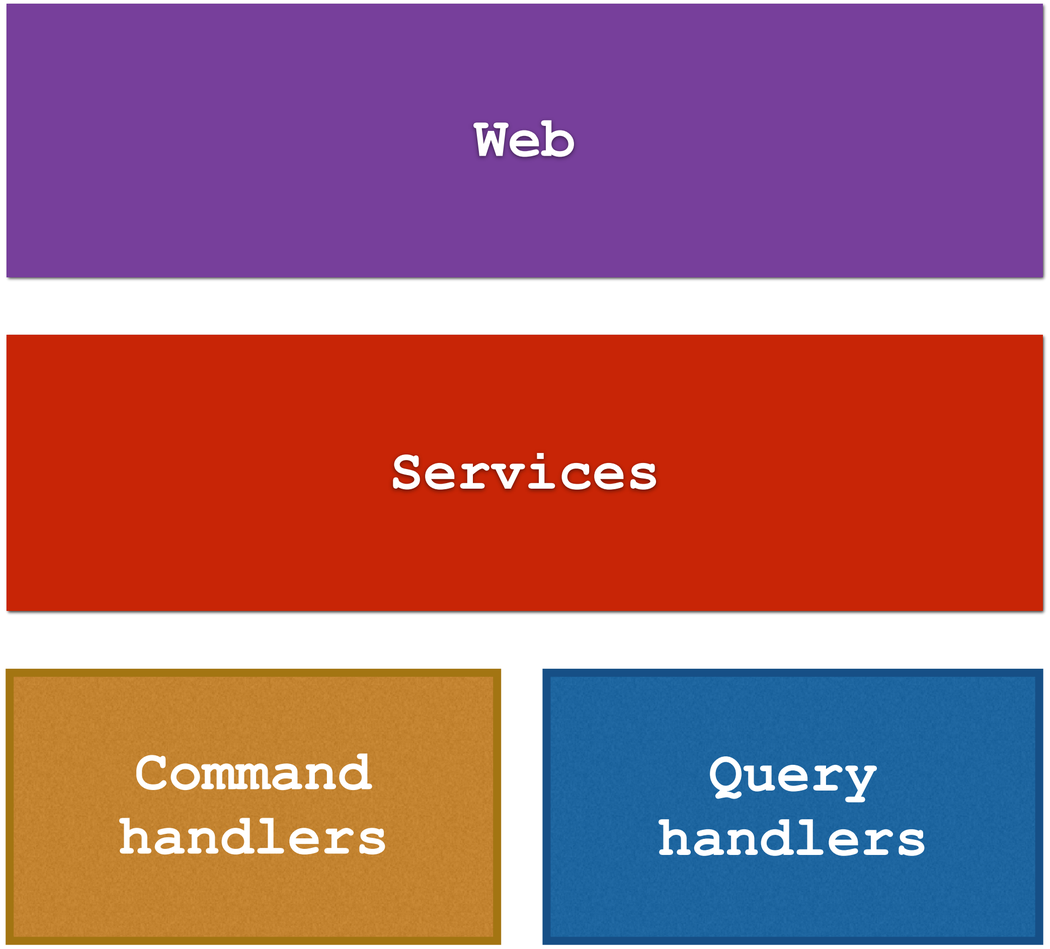

REST calls
e.g. Add User
Groups
Services
Commands
Query sync
Query views
EventStream
Platform Events
Lambda
Kinesis
DynamoDB


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

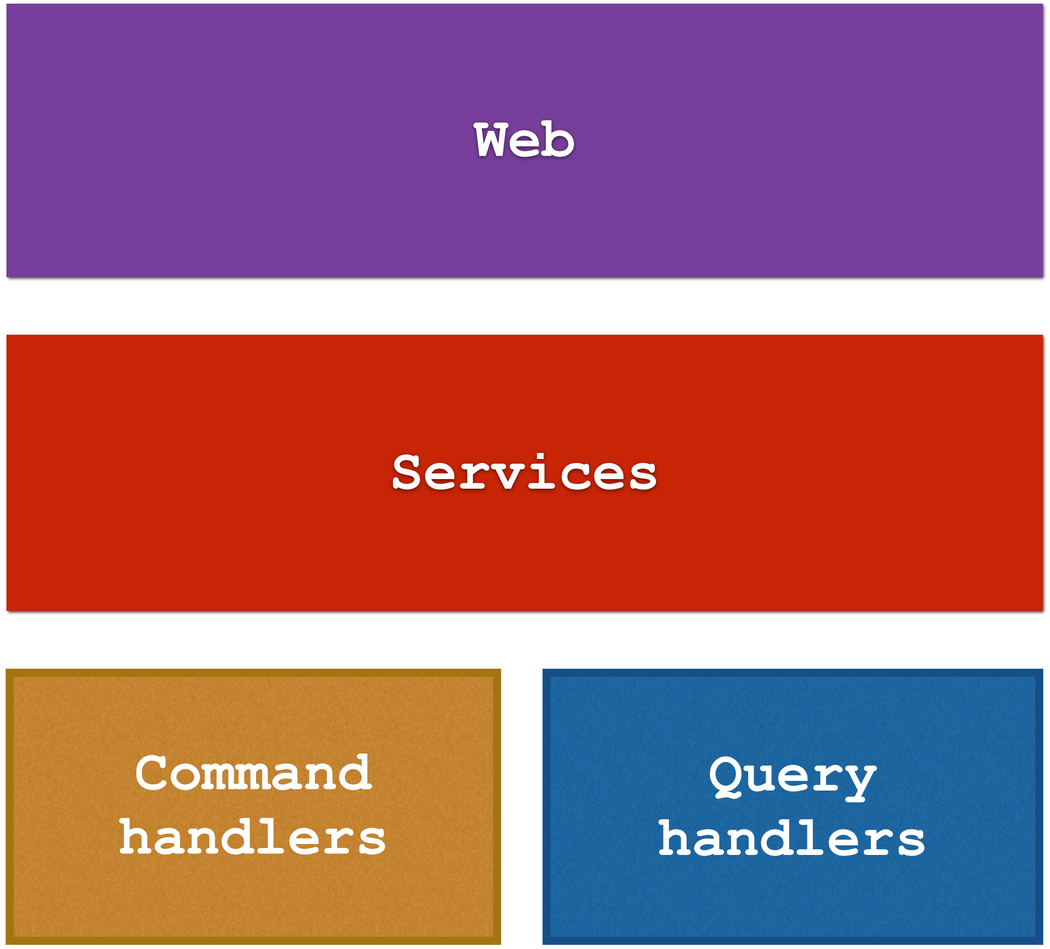

REST calls
e.g. Add User
Groups
Services
Commands
Query sync
Query views
EventStream
Users
DynamoDB
Platform Events
Lambda
Kinesis
Event Txf
Kinesis
External Events

Delving into code

Core app stack

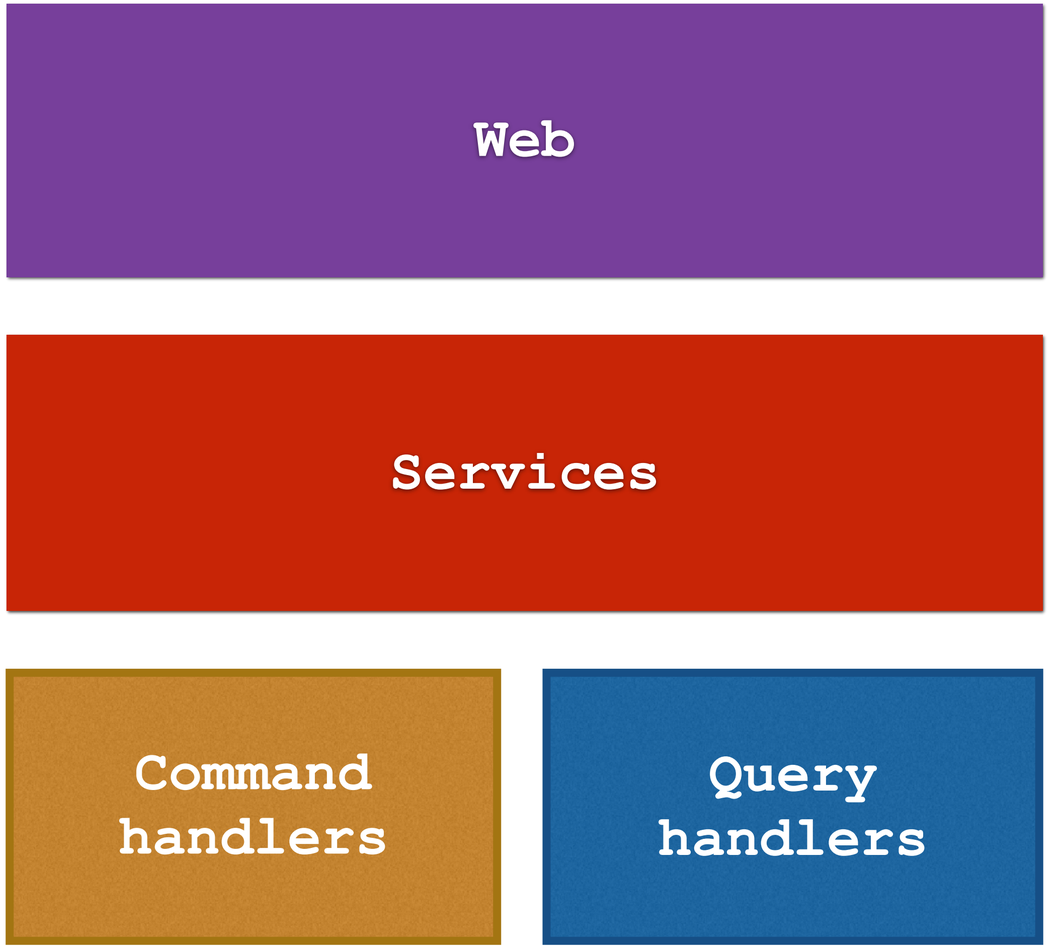


Core app stack




```
case class Save[A](...)
  extends command.Op[A]
case class UserByName[A](...)
  extends query.Op[A]
```

Core app stack

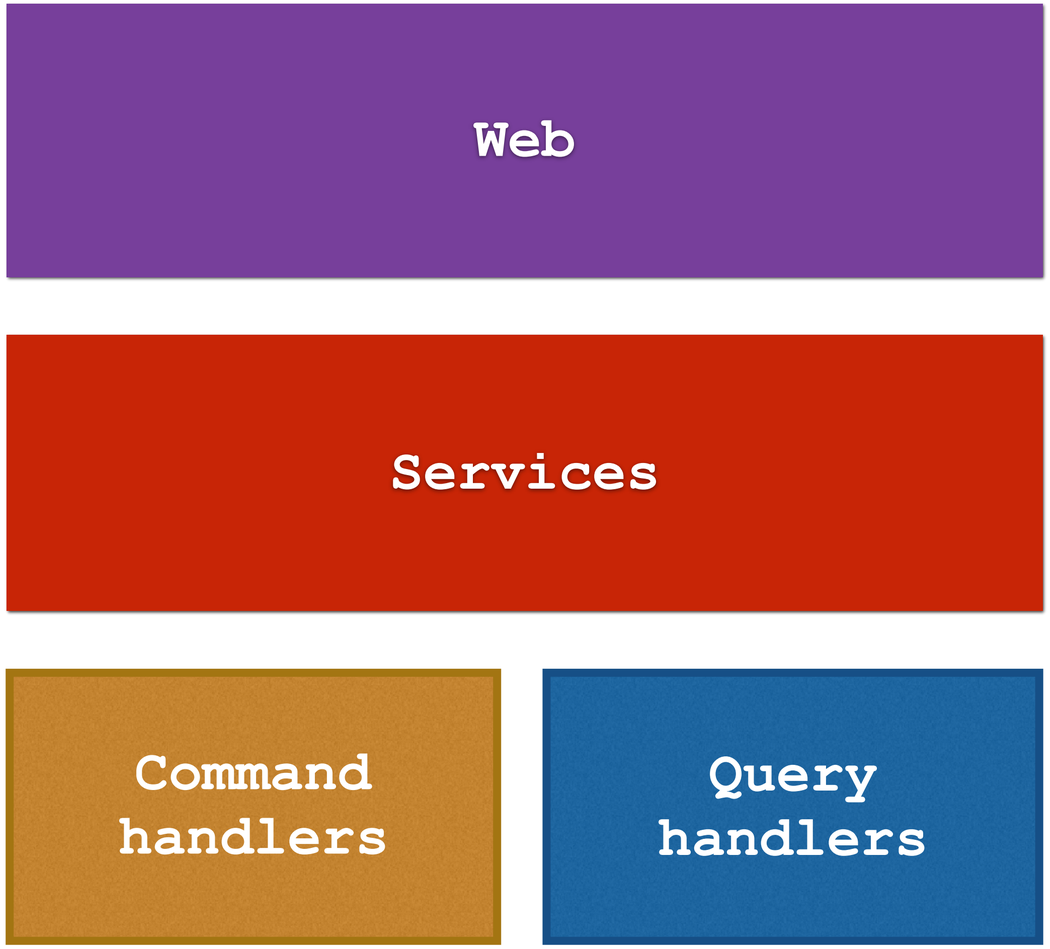



```
def addUser: AppOp[A]

type AppOp[A] = Free[App, A]
type App[A] =
  Coproduct[command.Op, query.Op, A]

case class Save[A](...)
  extends command.Op[A]
case class UserByName[A](...)
  extends query.Op[A]
```

Core app stack




```
post { entity(as[NewUser]) { u =>
  runApi(UserService.addUser(u))
}
```

```
val runApi: AppOp[A] => Route
```

```
def addUser: AppOp[A]
```

```
type AppOp[A] = Free[App, A]
type App[A] =
  Coproduct[command.Op, query.Op, A]
```

```
case class Save[A](...)
  extends command.Op[A]
case class UserByName[A](...)
  extends query.Op[A]
```

Commands save events to streams

```
object UserCommandHandler {
  def apply(streams: Streams): command.Op ~> Result =
    new (command.Op ~> Result) {
       def apply[X](a: command.Op[X]): Result[X] =
         a match {
            case Save(u) =>



            case ...
         }
    }
}
```

Commands save events to streams

```
object UserCommandHandler {
  def apply(streams: Streams): command.Op ~> Result =
    new (command.Op ~> Result) {
      def apply[X](a: command.Op[X]): Result[X] =
        a match {
          case Save(u) =>
            streams.allUsers.save(SingleStreamId, SetName(u.id, u.name))
              .then(streams.userProfile.save(u.id, SetProfile(u))
              .thenReturn(u.id)
          case ...
        }
    }
}
```

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) extends IRecordProcessor {

  override def processRecords(rs: ProcessRecordInput) = {



  }

}
```

...that you pass it an event stream

```
class KinesisProcessor(qs: QuerySynchroniser) extends IRecordProcessor {

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


  }

}
```

...that you pass it an event stream

```
class KinesisProcessor(qs: QuerySynchroniser) extends IRecordProcessor {

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

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

}
```

Events as an API

Insert / Update Delta vs ‘Set’ events

Insert / Update Delta vs ‘Set’ events

AddUser(id, name, email1)

Insert / Update Delta vs ‘Set’ events

AddUser(id, name, email1)

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

Insert / Update Delta vs ‘Set’ events

AddUser(id, name, email1)

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

SetUserName(id, name)

SetUserEmail(id, email1)

Insert / Update Delta vs ‘Set’ events

AddUser(id, name, email1)

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

SetUserName(id, name)

SetUserEmail(id, email1)

SetUserEmail(id, email2)

Insert / Update Delta vs ‘Set’ events

AddUser(id, name, email1)

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

Fits nicely with CRUD + PATCH

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

SetUserEmail(id, email2)

Insert / Update Delta vs ‘Set’ events

AddUser(id, name, email1)

UpdateUser(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

AddUser(id, name, email1)

UpdateUser(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

AddUser(id, name, email1)

UpdateUser(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

AddUser(id, name, email1)

UpdateUser(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

AddUser(id, name, email1)

UpdateUser(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

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

Let go of transactions and consistency

Query views get populated eventually

Query views get populated eventually

- No guaranteed order between events on different streams

Query views get populated eventually

- No guaranteed order between events on different streams
- A field should only be updated by a single event stream

Query views get populated eventually

- No guaranteed order between events on different streams
- A field should only be updated by a single event stream
- No foreign key constraints

Query views get populated eventually

- No guaranteed order between events on different streams
- A field should only be updated by a single event stream
- No foreign key constraints
- Unique or data constraints ‘enforced’ on write

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)

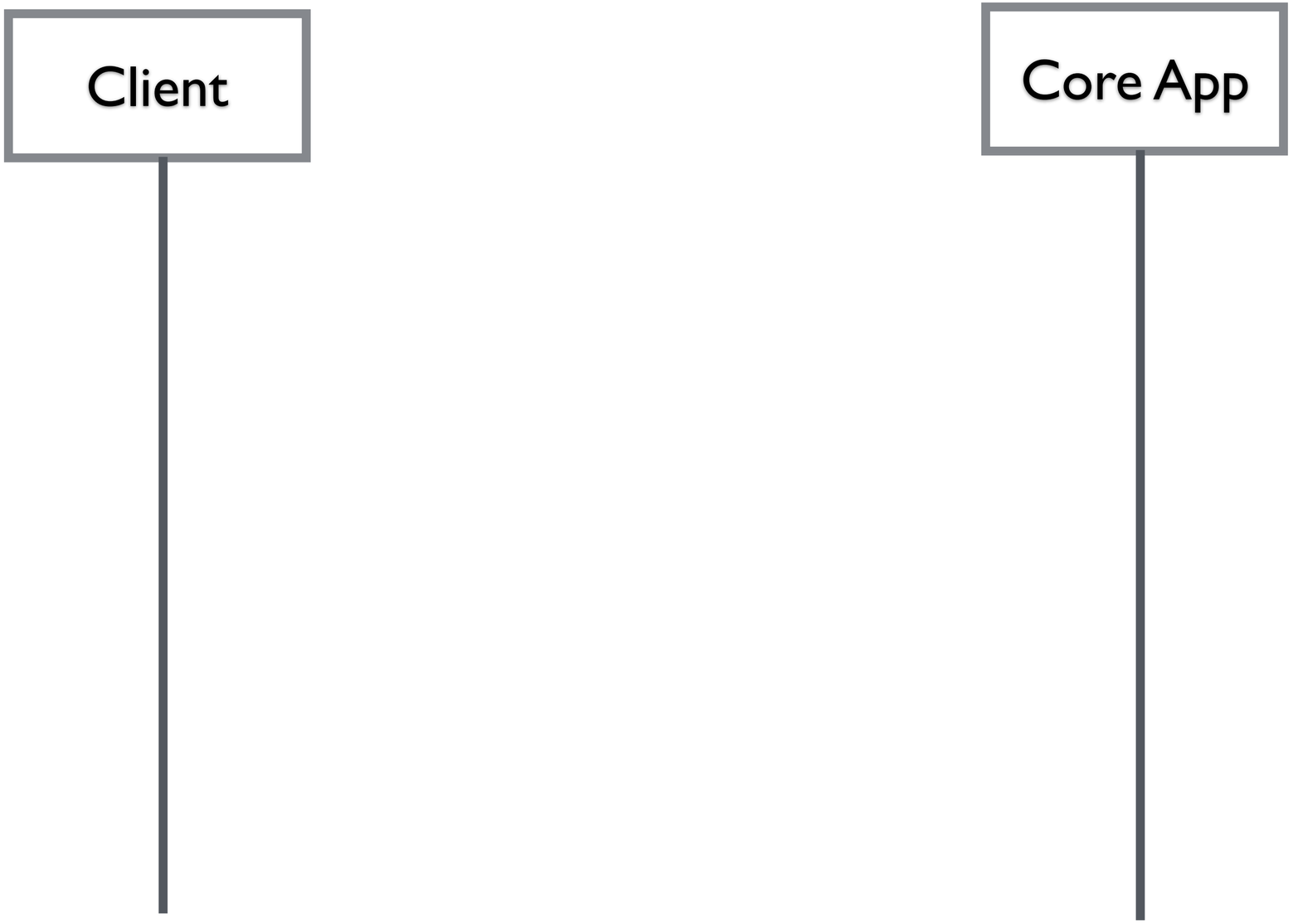
All Users: Seq 100

User 4: Seq 23

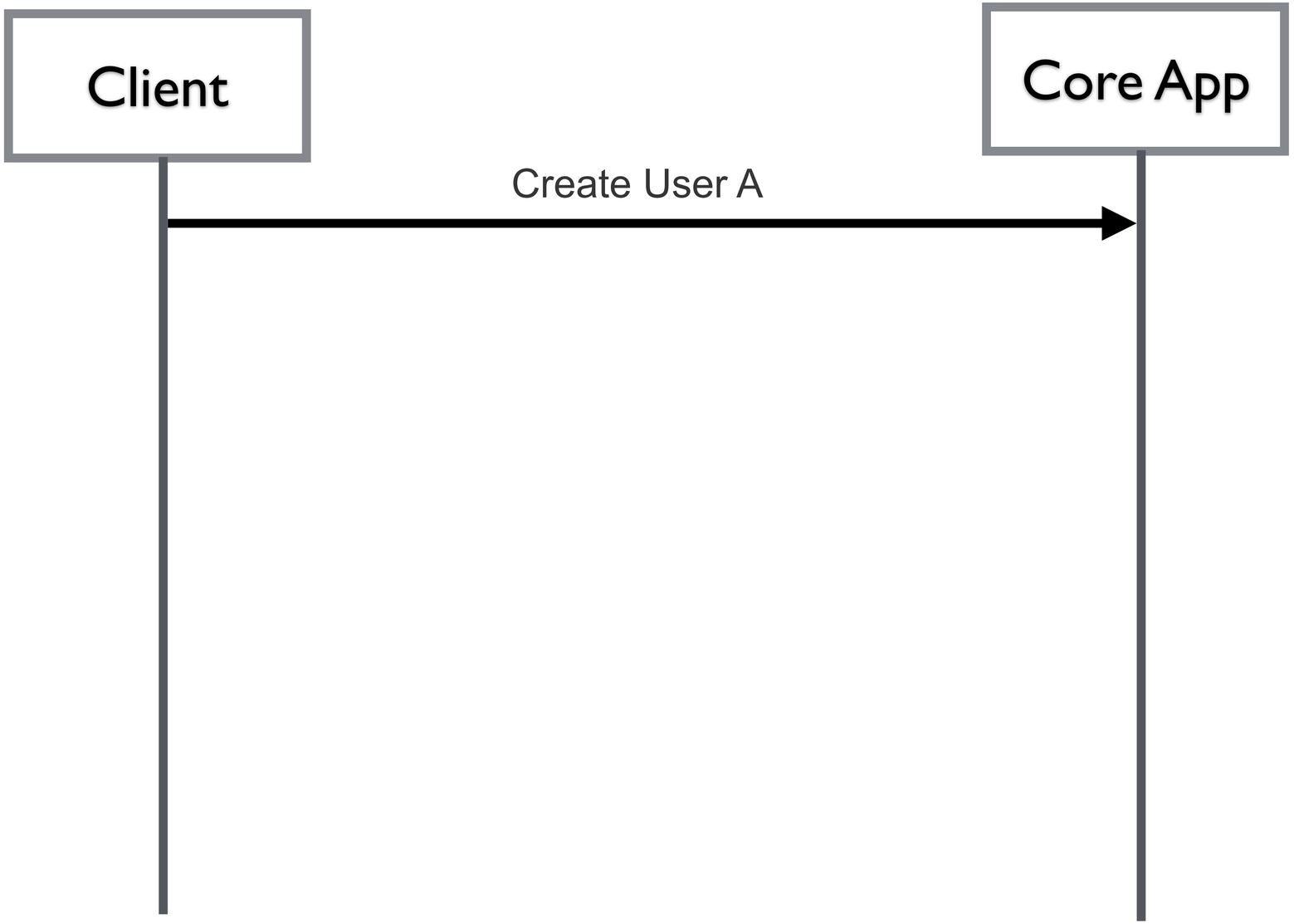
- Returned on read and write via ETag

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

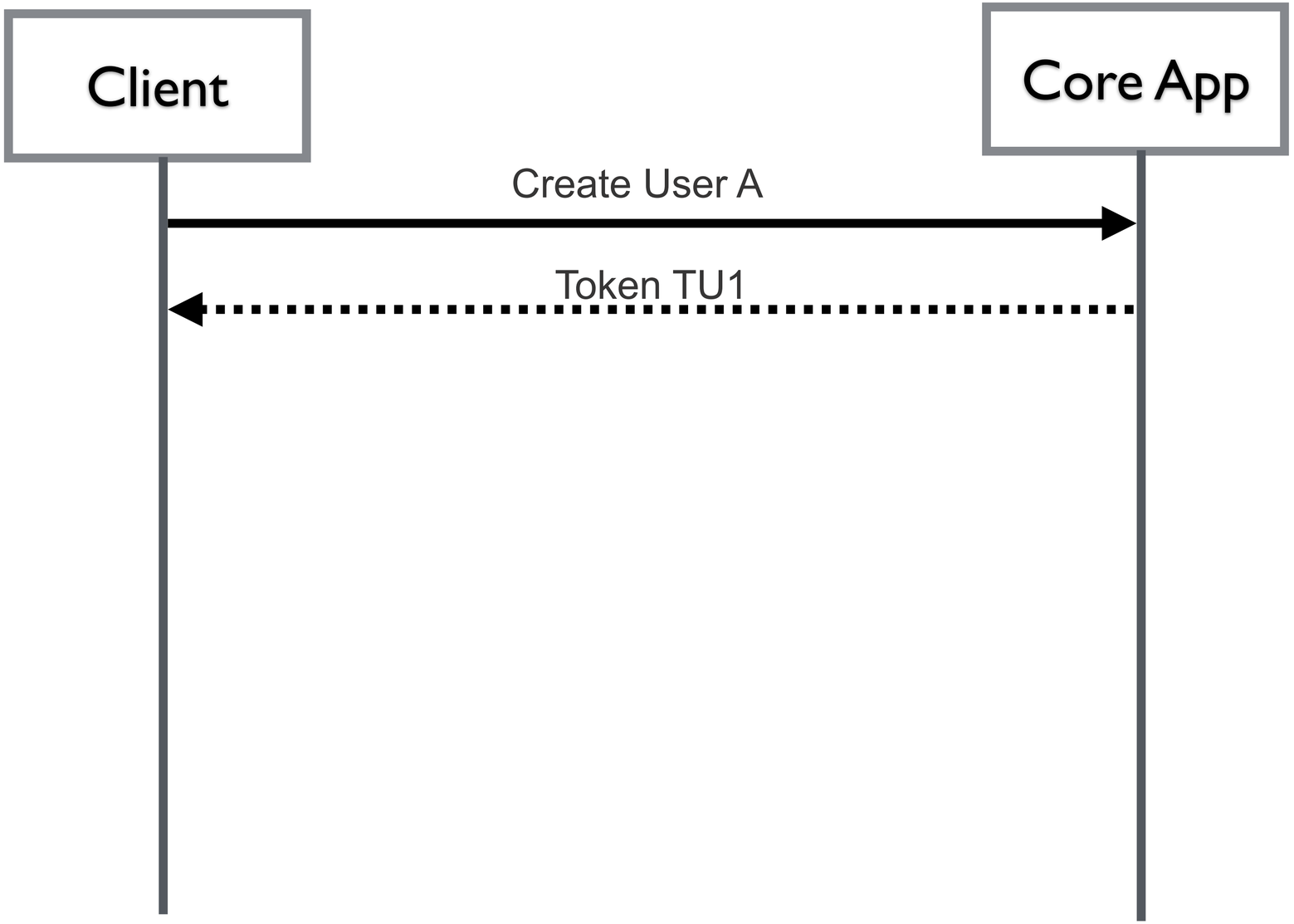
Using tokens to enforce state



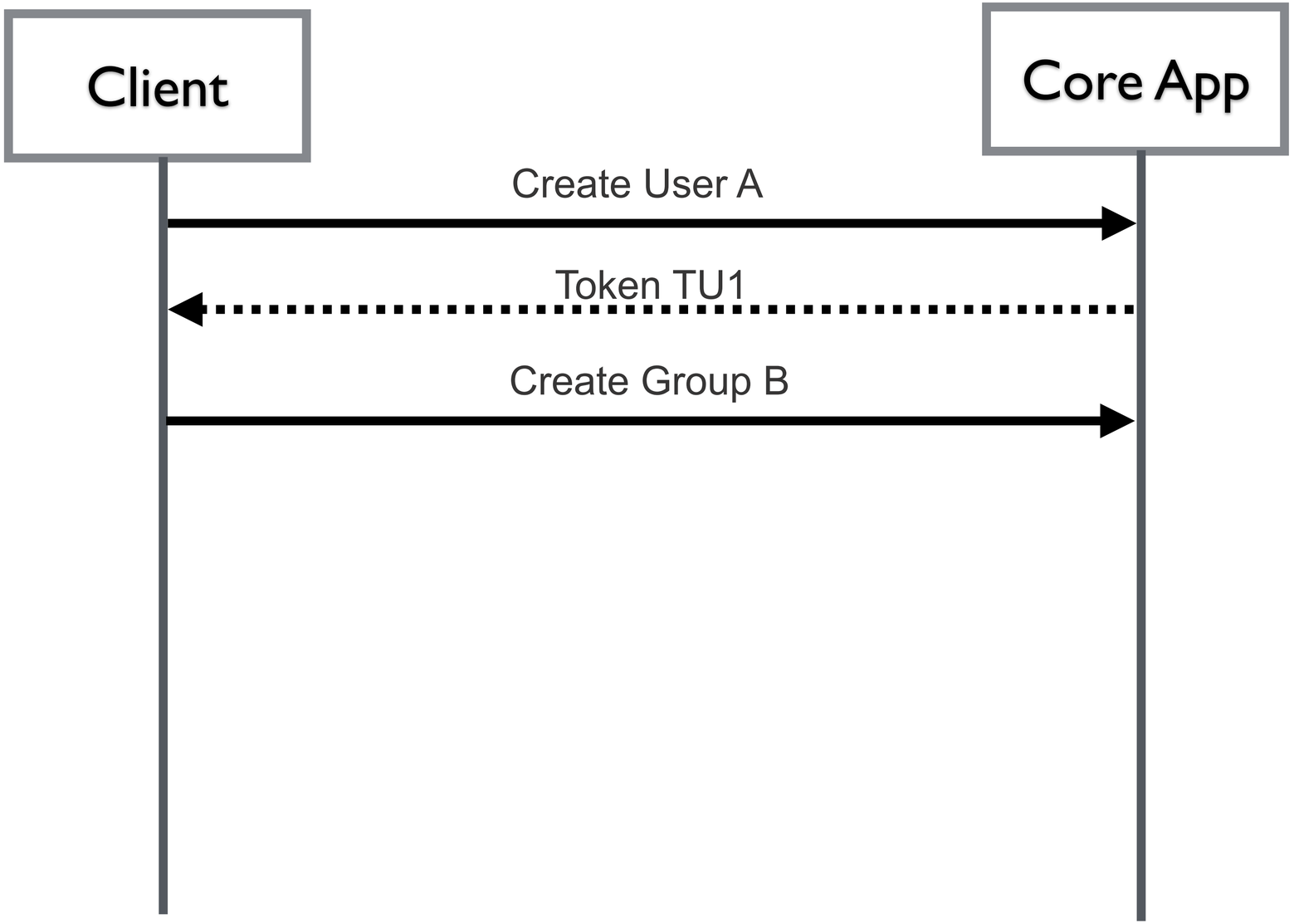
Using tokens to enforce state



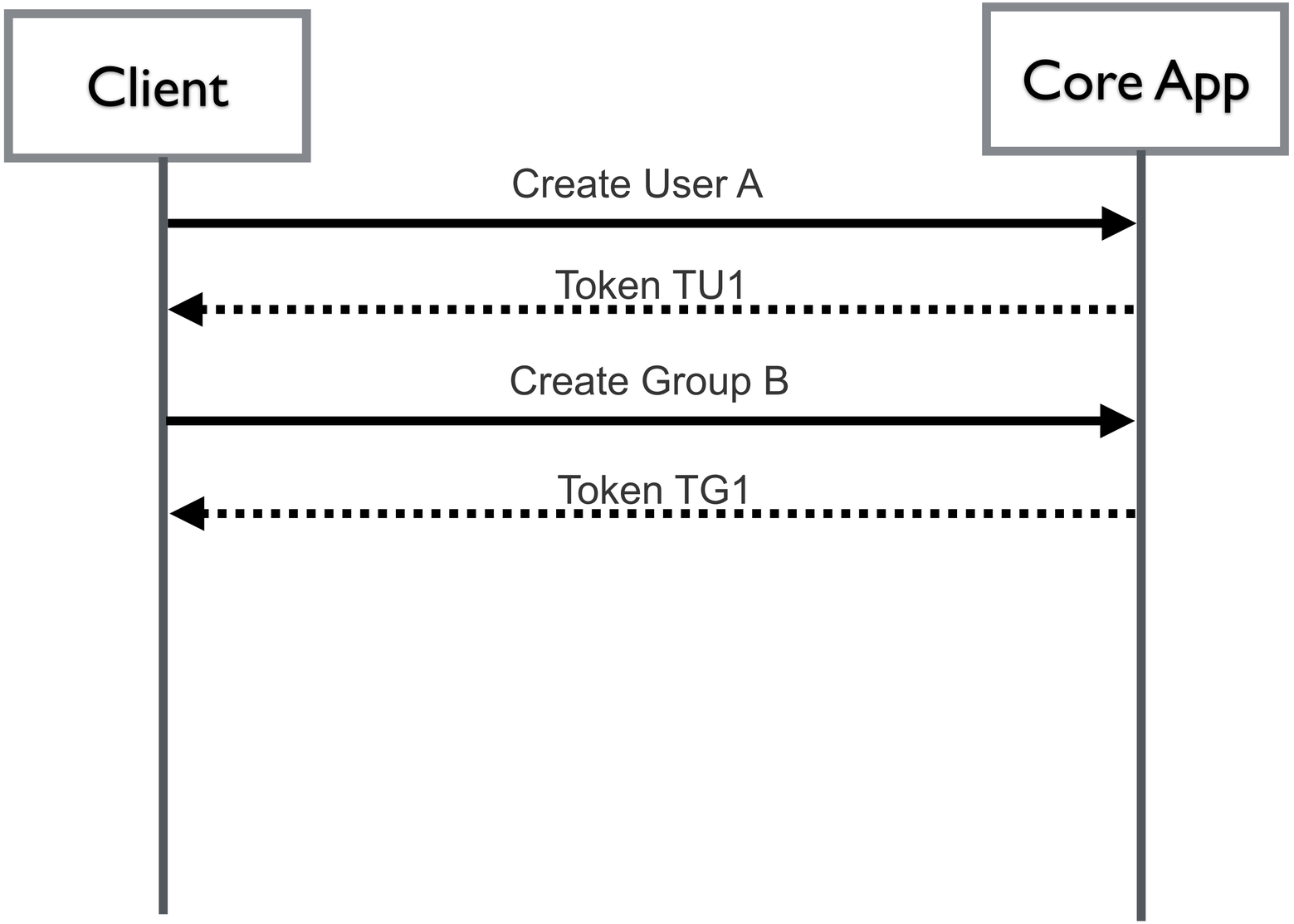
Using tokens to enforce state



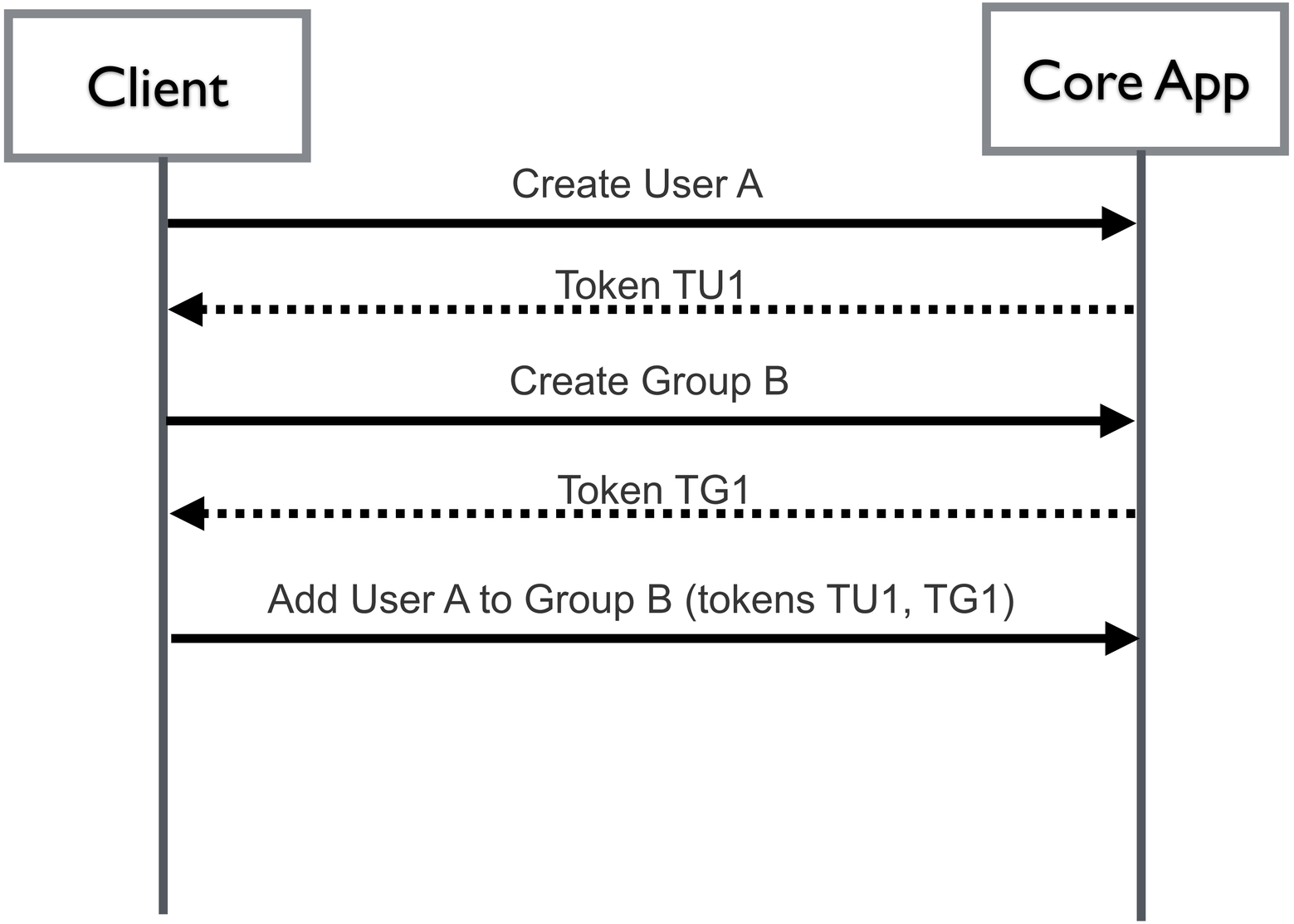
Using tokens to enforce state



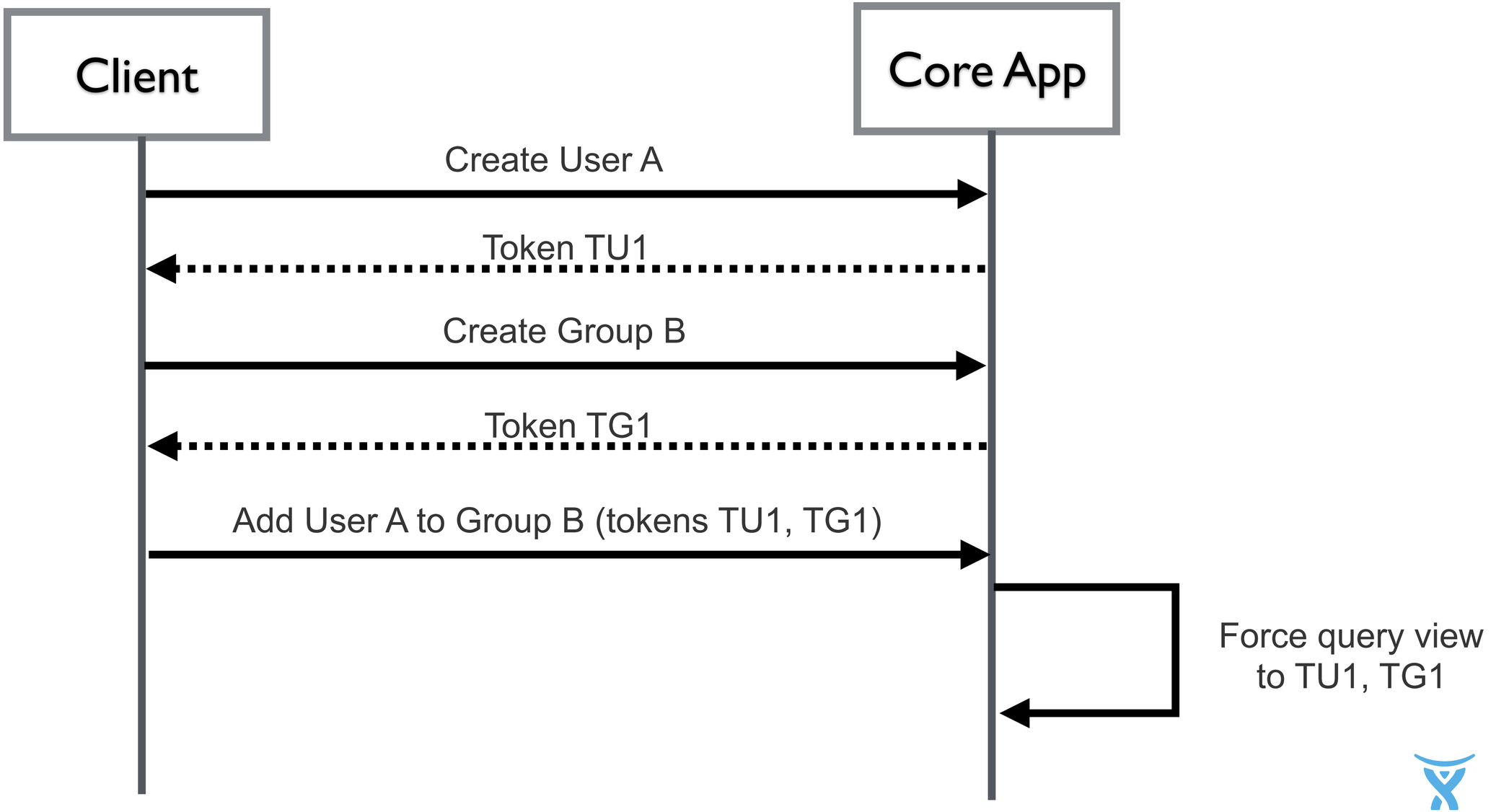
Using tokens to enforce state



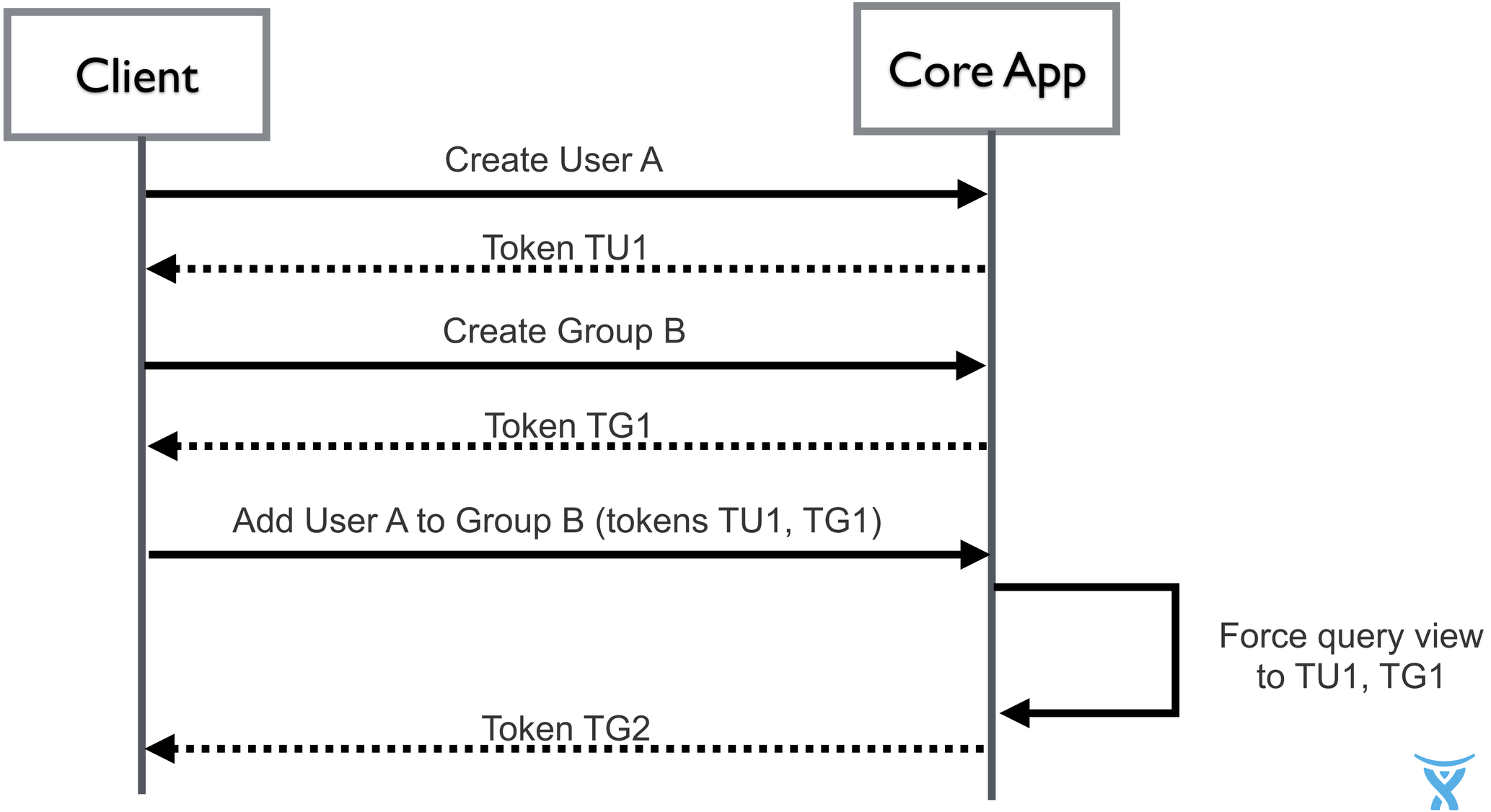
Using tokens to enforce state



Using tokens to enforce state



Using tokens to enforce state



Key takeaways

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