Atlassian

Immutable data stores for safety, flexibility and profit

SIDNEY SHEK · ARCHITECT · ATLASSIAN · @SIDNEYSHEK

Event Sourcing: What and Why

Universe of Users and Groups

users

Id	Name	Username	APIKey
1	Homer	homer	abcd
2	Bart	bart	f00

users_groups

GroupId	UserId
1	1
1	2

groups

Id	Name
1	Simpsons
2	Flanders

Universe of Users and Groups

users

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

users_groups

GroupId	UserId
1	1
1	2
1	3

groups

Id	Name
1	Simpsons
2	Flanders

Universe of Users and Groups

users

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

users_groups

GroupId	UserId
1	1
1	2
1	3

groups

Id	Name
1	Simpsons
2	Flanders

Universe of Users and Groups

users

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

users_groups

GroupId	UserId
~~1~~	~~1~~
1	2
1	3

groups

Id	Name
1	Simpsons
2	Flanders

If only there was a way to see what changed...

Universe of Users and Groups

users

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

users_groups

GroupId	UserId
1	2
1	3

groups

Id	Name
1	Simpsons
2	Flanders

audit

Id	Event	Time
1	InsertUser(3, Maggie)	0
2	AddUserToGroup(3, 1)	10
3	SetAPIKey(1, d0a)	15
4	RemoveUserFromGroup(1, 1)	20

How many API key changes in the last 6 months?

Universe of Users and Groups

users

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

users_groups

GroupId	UserId
1	2
1	3

groups

Id	Name
1	Simpsons
2	Flanders

audit

Id	Event	Time
1	InsertUser(3, Maggie)	0
2	AddUserToGroup(3, 1)	10
3	SetAPIKey(1, d0a)	15
4	RemoveUserFromGroup(1, 1)	20

Universe of Users and Groups

users

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

users_groups

GroupId	UserId
1	2
1	3

groups

Id	Name
1	Simpsons
2	Flanders

audit

Id	Event	Time
1	InsertUser(3, Maggie)	0
2	AddUserToGroup(3, 1)	10
3	SetAPIKey(1, d0a)	15
4	RemoveUserFromGroup(1, 1)	20

What if…

Instead of audit for reporting only...

Audit became our source of truth

Event Sourcing!

Universe of Users and Groups

users

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

users_groups

GroupId	UserId
1	2
1	3

groups

Id	Name
1	Simpsons
2	Flanders

audit

Id	Event	Time
1	InsertUser(3, Maggie)	0
2	AddUserToGroup(3, 1)	10
3	SetAPIKey(1, d0a)	15
4	RemoveUserFromGroup(1, 1)	20

Universe of Users and Groups

users

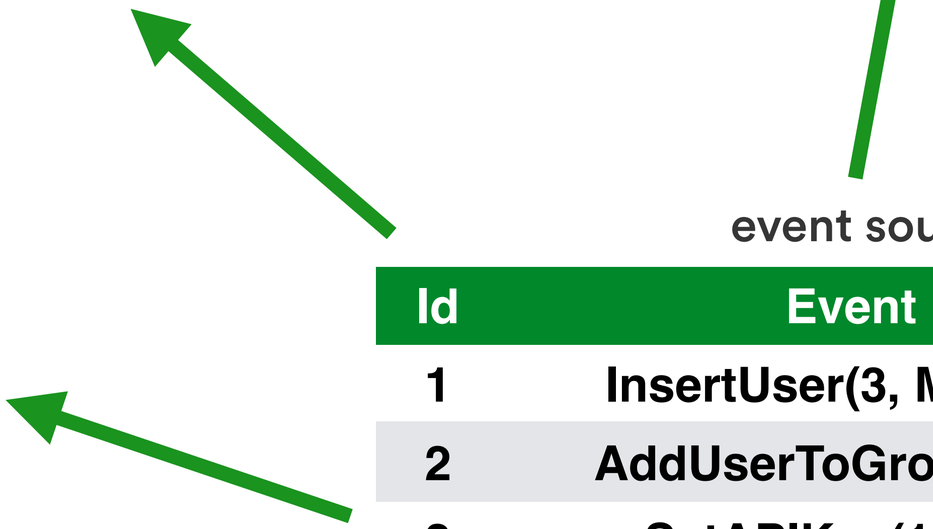
Id	Name	Username	APIKey
1	Homer	homer	d0a
2	Bart	bart	f00
3	Maggie	maggie	baa

users_groups

GroupId	UserId
1	2
1	3

groups

Id	Name
1	Simpsons
2	Flanders



event source

Id	Event	Time
1	InsertUser(3, Maggie)	0
2	AddUserToGroup(3, 1)	10
3	SetAPIKey(1, d0a)	15
4	RemoveUserFromGroup(1, 1)	20

Universe of Users and Groups

users

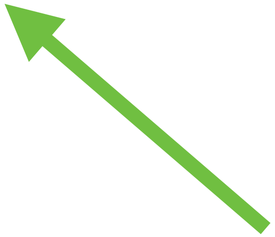
Id	Name	Username	APIKey
1	Homer	homer	d0a
2	Bart	bart	f00
3	Maggie	maggie	baa

users_groups

GroupId	UserId
1	2
1	3

groups

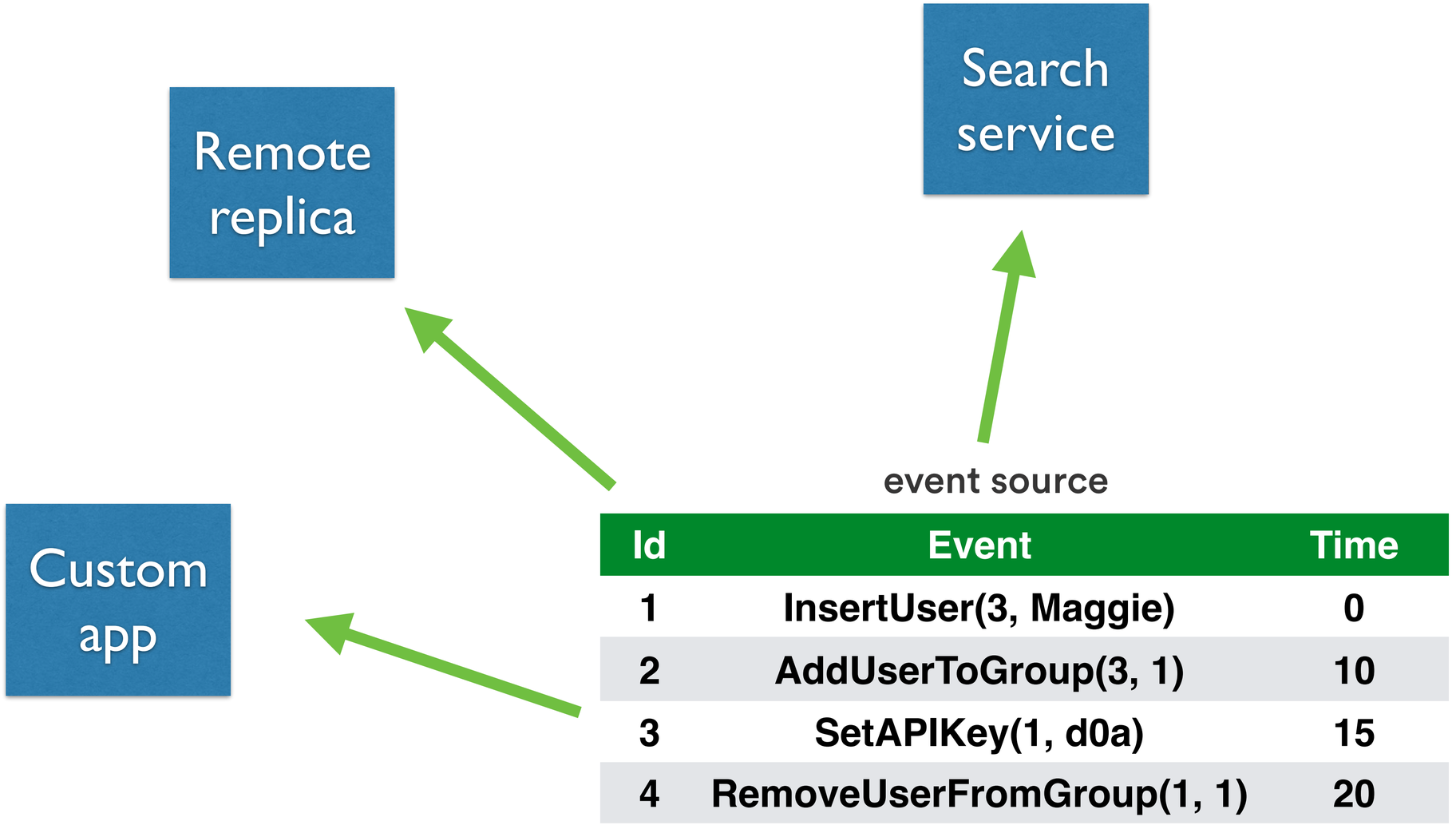
Id	Name
1	Simpsons
2	Flanders



event source

Id	Event	Time
1	InsertUser(3, Maggie)	0
2	AddUserToGroup(3, 1)	10
3	SetAPIKey(1, d0a)	15
4	RemoveUserFromGroup(1, 1)	20

Universe of Users and Groups



Id	Event	Time
1	InsertUser(3, Maggie)	0
2	AddUserToGroup(3, 1)	10
3	SetAPIKey(1, d0a)	15
4	RemoveUserFromGroup(1, 1)	20

How do we store events?

Ordered ‘Sequence’ (key)

event source

Id	Event	Time
1	InsertUser(3, Maggie)	0
2	AddUserToGroup(3, 1)	10
3	SetAPIKey(1, d0a)	15
4	RemoveUserFromGroup(1, 1)	20

Flexible Value

We don’t need an RDBMS

We just need a key-value store!

Building an event sourcing library

event sourcing lib:

bitbucket.org/atlassianlabs/eventsrc

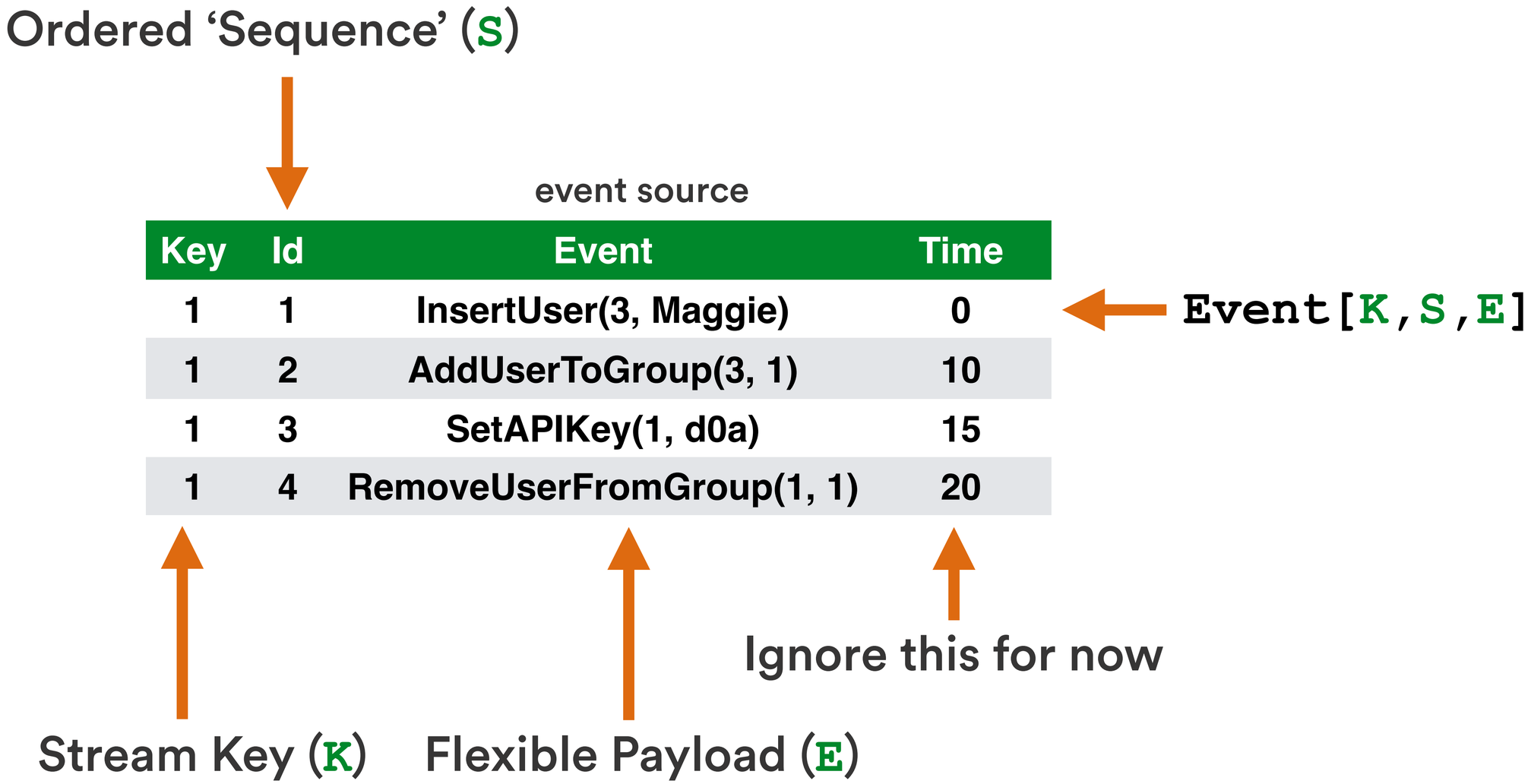
Step 1. Modelling the events table

Ordered ‘Sequence’ (S)

event source

Id	Event	Time
1	InsertUser(3, Maggie)	0
2	AddUserToGroup(3, 1)	10
3	SetAPIKey(1, d0a)	15
4	RemoveUserFromGroup(1, 1)	20

Flexible Payload (E)


Ordered ‘Sequence’ (S)
event source
Key
Id
Event
Time
1
1
InsertUser(3, Maggie)
0
1
2
AddUserToGroup(3, 1)
10
1
3
SetAPIKey(1, d0a)
15
1
4
RemoveUserFromGroup(1, 1)
20
Event[K,S,E]
Ignore this for now
Stream Key (K)
Flexible Payload (E)

Stream of Events

```
trait EventStream {

   type E    // Event payload e.g. InsertUser

}
```

Stream of Events

```
trait EventStream {

  type E    // Event payload e.g. InsertUser

  type S    // Sequence type e.g. Long




}
```

Stream of Events

```
trait EventStream {

  type E    // Event payload e.g. InsertUser

  type S    // Sequence type e.g. Long

  implicit def S: Sequence[S]



}
```

Stream of Events

```
trait EventStream {

  type E    // Event payload e.g. InsertUser

  type S    // Sequence type e.g. Long

  implicit def S: Sequence[S]

  type K    // Stream key e.g. CompanyId

}
```

Stream of Events

```
case class Event[K, S, E](

 payload: E)
```

Stream of Events

```
case class Event[K, S, E](
 id: EventId[K, S],
 payload: E)

case class EventId[K, S](key: K, seq: S)
```

Storing Events

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

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

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

}
```

Storing Events

```
import scalaz.stream.Process

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

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

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

}
```

Stream of Events

```
trait EventStream[F[_]] {
  type E    // Event payload e.g. InsertUser
  type S    // Sequence type e.g. Long
  implicit def S: Sequence[S]
  type K    // Stream key e.g. CompanyId
  type Ev = Event[K, S, E]

  def eventStore: EventStorage[F, K, S, E]
  implicit def M: Monad[F]
}
```

Let’s apply this to our users example

Our User Account Events

```
sealed trait UserAccountEvent
```

Our User Account Events

```
sealed trait UserAccountEvent

case class InsertUser(id: UserId, name: String,
  username: String) extends UserAccountEvent
```

Our User Account Events

```
sealed trait UserAccountEvent

case class InsertUser(id: UserId, name: String,
  username: String) extends UserAccountEvent

case class DeleteUser(id: UserId) extends UserAccountEvent
```

Our User Account Events

```
sealed trait UserAccountEvent

case class InsertUser(id: UserId, name: String,
  username: String) extends UserAccountEvent

case class DeleteUser(id: UserId) extends UserAccountEvent

case class SetAPIKey(id: UserId, apiKey: String)
  extends UserAccountEvent
```

Our User Account Events

```
sealed trait UserAccountEvent

case class InsertUser(id: UserId, name: String,
  username: String) extends UserAccountEvent

case class DeleteUser(id: UserId) extends UserAccountEvent

case class SetAPIKey(id: UserId, apiKey: String)
  extends UserAccountEvent

case class AddUserToGroup(groupId: GroupId, userId: UserId)
  extends UserAccountEvent
```

Our User Account Events

```
sealed trait UserAccountEvent

case class InsertUser(id: UserId, name: String,
  username: String) extends UserAccountEvent

case class DeleteUser(id: UserId) extends UserAccountEvent

case class SetAPIKey(id: UserId, apiKey: String)
  extends UserAccountEvent

case class AddUserToGroup(groupId: GroupId, userId: UserId)
  extends UserAccountEvent

case class RemoveUserFromGroup(groupId: GroupId,
  userId: UserId) extends UserAccountEvent
```

Our User Account EventStream

```
class UserAccountEventStream[F[_]]()
  extends EventStream[F] {




}
```

Our User Account EventStream

```
class UserAccountEventStream[F[_]]()
  extends EventStream[F] {

  type E = UserAccountEvent



  ...
}
```

Our User Account EventStream

```
class UserAccountEventStream[F[_]]()
  extends EventStream[F] {

  type E = UserAccountEvent
  type S = Long  // Sequence[Long] already defined


  ...
}
```

Our User Account EventStream

```
class UserAccountEventStream[F[_]]()
  extends EventStream[F] {

  type E = UserAccountEvent
  type S = Long  // Sequence[Long] already defined
  type K = CompanyId

  ...
}
```

Our User Account EventStream

```
class UserAccountEventStream[F[_]](
  val eventStore: EventStorage[F, CompanyId, Long,
                               UserAccountEvent])
  extends EventStream[F] {

  type E = UserAccountEvent
  type S = Long  // Sequence[Long] already defined
  type K = CompanyId

  ...
}
```

Step 2. Querying events

Querying event streams

```
trait EventStream[F[_]] {

  ...

  trait QueryAPI[Key, Val] {

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

  }
}
```

Querying event streams

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

    eventStore.get(???)
```

Querying event streams

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

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

Querying event streams

```
import scalaz.stream.process1

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

def streamFold(
  f: (Snapshot[S, Val], Ev) => Snapshot[S, Val]
  )(stream: Process[F, Ev]): F[Snapshot[S, Val]] =
    stream.pipe {
      process1.fold(Snapshot.zero[S, Val])(f)
    }.runLastOr(Snapshot.zero[S, Val])
```

Querying event streams

```
trait QueryAPI[Key, Val] {



  def acc(k: Key)(s: Snapshot[S, Val], e: Ev):Snapshot[S, Val]

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

Querying event streams

```
trait QueryAPI[Key, Val] {

  def toStreamKey: Key => K

  def acc(k: Key)(s: Snapshot[S, Val], e: Ev):Snapshot[S, Val]

  def get(k: Key): F[Option[Val]] =
    streamFold(acc) {
      eventStore.get(toStreamKey(k))
    }.map { _.value }
}
```

Let’s apply this to our users example

Get group members

```
class GroupMembersById extends QueryAPI[CompanyGroupId, List[UserId]] {
  def toStreamKey: CompanyGroupId => CompanyId = _.companyId

  def acc(k: CompanyGroupId)(s: Snapshot[Long, List[UserId]], e: Ev) =









}
```

Get group members

```
class GroupMembersById extends QueryAPI[CompanyGroupId, List[UserId]] {
  def toStreamKey: CompanyGroupId => CompanyId = _.companyId

  def acc(k: CompanyGroupId)(s: Snapshot[Long, List[UserId]], e: Ev) =
  e.payload match {
    case AddUserToGroup(groupId, userId) if k.groupId == groupId =>

  }
}
```

Get group members

```
class GroupMembersById extends QueryAPI[CompanyGroupId, List[UserId]] {
  def toStreamKey: CompanyGroupId => CompanyId = _.companyId

  def acc(k: CompanyGroupId)(s: Snapshot[Long, List[UserId]], e: Ev) =
  e.payload match {
    case AddUserToGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())

  }
}
```

Get group members

```
class GroupMembersById extends QueryAPI[CompanyGroupId, List[UserId]] {
  def toStreamKey: CompanyGroupId => CompanyId = _.companyId

  def acc(k: CompanyGroupId)(s: Snapshot[Long, List[UserId]], e: Ev) =
  e.payload match {
    case AddUserToGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())
      val newList = userId :: currentList.filterNot { _ == userId }









  }
}
```

Get group members

```
class GroupMembersById extends QueryAPI[CompanyGroupId, List[UserId]] {
  def toStreamKey: CompanyGroupId => CompanyId = _.companyId

  def acc(k: CompanyGroupId)(s: Snapshot[Long, List[UserId]], e: Ev) =
  e.payload match {
    case AddUserToGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())
      val newList = userId :: currentList.filterNot { _ == userId }
      Snapshot.value(newList, e.id.seq)




  }
}
```

Get group members

```
class GroupMembersById extends QueryAPI[CompanyGroupId, List[UserId]] {
  def toStreamKey: CompanyGroupId => CompanyId = _.companyId

  def acc(k: CompanyGroupId)(s: Snapshot[Long, List[UserId]], e: Ev) =
  e.payload match {
    case AddUserToGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())
      val newList = userId :: currentList.filterNot { _ == userId }
      Snapshot.value(newList, e.id.seq)
    case RemoveUserFromGroup(groupId, userId) if k.groupId == groupId =>




  }
}
```

Get group members

```
class GroupMembersById extends QueryAPI[CompanyGroupId, List[UserId]] {
  def toStreamKey: CompanyGroupId => CompanyId = _.companyId

  def acc(k: CompanyGroupId)(s: Snapshot[Long, List[UserId]], e: Ev) =
  e.payload match {
    case AddUserToGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())
      val newList = userId :: currentList.filterNot { _ == userId }
      Snapshot.value(newList, e.id.seq)
    case RemoveUserFromGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())



  }
}
```

Get group members

```
class GroupMembersById extends QueryAPI[CompanyGroupId, List[UserId]] {
  def toStreamKey: CompanyGroupId => CompanyId = _.companyId

  def acc(k: CompanyGroupId)(s: Snapshot[Long, List[UserId]], e: Ev) =
  e.payload match {
    case AddUserToGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())
      val newList = userId :: currentList.filterNot { _ == userId }
      Snapshot.value(newList, e.id.seq)
    case RemoveUserFromGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())
      val newList = currentList.filterNot { _ == userId }


  }
}
```

Get group members

```
class GroupMembersById extends QueryAPI[CompanyGroupId, List[UserId]] {
  def toStreamKey: CompanyGroupId => CompanyId = _.companyId

  def acc(k: CompanyGroupId)(s: Snapshot[Long, List[UserId]], e: Ev) =
  e.payload match {
    case AddUserToGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())
      val newList = userId :: currentList.filterNot { _ == userId }
      Snapshot.value(newList, e.id.seq)
    case RemoveUserFromGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())
      val newList = currentList.filterNot { _ == userId }
      Snapshot.value(newList, e.id.seq)

  }
}
```

Get group members

```
class GroupMembersById extends QueryAPI[CompanyGroupId, List[UserId]] {
  def toStreamKey: CompanyGroupId => CompanyId = _.companyId

  def acc(k: CompanyGroupId)(s: Snapshot[Long, List[UserId]], e: Ev) =
  e.payload match {
    case AddUserToGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())
      val newList = userId :: currentList.filterNot { _ == userId }
      Snapshot.value(newList, e.id.seq)
    case RemoveUserFromGroup(groupId, userId) if k.groupId == groupId =>
      val currentList = s.value.getOrElse(List())
      val newList = currentList.filterNot { _ == userId }
      Snapshot.value(newList, e.id.seq)
    case _ => Snapshot.noop(s, e.id.seq)
  }
}
```

Bonus:
Going back in time

Querying event streams...with history

```
def get(k: Key): F[Option[Val]] =
  streamFold(acc(k)) {
    eventStore.get(toStreamKey(k))
  }.map { _.value }

def getAt(k: Key, s: S): F[Option[Val]] =
```

Querying event streams...with history

```
def get(k: Key): F[Option[Val]] =
  streamFold(acc(k)) {
    eventStore.get(toStreamKey(k))
  }.map { _.value }

def getAt(k: Key, s: S): F[Option[Val]] =
  streamFold(acc(k)) {
    eventStore.get(toStreamKey(k)).takeWhile { e =>
      S.order.lessThanOrEqual(e.id.s, s)
    }
  }.map { _.value }
```

Step 3. Saving events

```scala
trait EventStream[F[_]] {

  ...

  trait SaveAPI[Key, Val] {

    def save(k: Key, e: E): F[SaveResult[S, Val]]

  }
}
```

```
trait SaveAPI[Key, Val] {
  def save(k: Key, e: E): F[SaveResult[S, Val]] =
    for {
      old <- getLatestSnapshot(k)

    } yield ???
}
```

```
trait SaveAPI[Key, Val] {
  def save(k: Key, e: E): F[SaveResult[S, Val]] =
    for {
      old <- getLatestSnapshot(k)
      putResult <- eventStore.put(k, Event.next(old.seq, e))
    } yield ???
}
```

```
trait SaveAPI[Key, Val] {
  def save(k: Key, e: E): F[SaveResult[S, Val]] =
    for {
      old <- getLatestSnapshot(k)
      putResult <- eventStore.put(k, Event.next(old.seq, e))
      saveResult <- putResult match {
        case \/-(ev) =>
          SaveResult.success(newValue(old, ev))




      }
    } yield saveResult
}
```

```
trait SaveAPI[Key, Val] {
  def save(k: Key, e: E): F[SaveResult[S, Val]] =
    for {
      old <- getLatestSnapshot(k)
      putResult <- eventStore.put(k, Event.next(old.seq, e))
      saveResult <- putResult match {
        case \/-(ev) =>
          SaveResult.success(newValue(old, ev))
        case -\/(Error.DuplicateEventId) =>
          save(k, e)


      }
    } yield saveResult
}
```

```
trait SaveAPI[Key, Val] {
  def save(k: Key, e: E): F[SaveResult[S, Val]] =
    for {
      old <- getLatestSnapshot(k)
      putResult <- eventStore.put(k, Event.next(old.seq, e))
      saveResult <- putResult match {
        case \/-(ev) =>
          SaveResult.success(newValue(old, ev))
        case -\/(Error.DuplicateEventId) =>
          save(k, e)
        case -\/(Error.Rejected(reasons)) =>
          SaveResult.reject(reasons)
      }
    } yield saveResult
}
```

```
abstract class SaveAPI[Key, Val](query: QueryAPI[Key, Val]) {
  def save(k: Key, e: E): F[SaveResult[S, Val]] =
    for {
      old <- query.getLatestSnapshot(k)
      putResult <- eventStore.put(k, Event.next(old.seq, 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
}
```

What about data constraints?

Operation: Constraint as a type

```
case class Operation[S, Val, E](
  run: Snapshot[S, Val] => OpResult[E])
```

Operation: Constraint as a type

```
case class Operation[S, Val, E](
  run: Snapshot[S, Val] => 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 ifNew(e: E): Operation[S, Val, E] =
    Operation { _.value match
      case None    => Success(e)
      case Some(_) => Reject(List(Reason("Duplicate value")))
    }

  ...
}
```

Operation: Constraint as a type

```
object Operation {
  def ifNew(e: E): Operation[S, Val, E] =
    Operation { _.value match
      case None    => Success(e)
      case Some(_) => Reject(List(Reason("Duplicate value")))
    }

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

  ...
}
```

Save without Constraints

```
abstract class SaveAPI[Key, Val](query: QueryAPI[Key, Val]) {
  def save(k: Key, e: E): F[SaveResult[S, Val]] =
    for {
      old <- query.getLatestSnapshot(k)

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

```
abstract class SaveAPI[Key, Val](query: QueryAPI[Key, Val]) {
  def save(k: Key, op: Operation[S, Val, E]): F[SaveResult[S, Val]] =
    for {
      old <- query.getLatestSnapshot(k)

      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

```
abstract class SaveAPI[Key, Val](query: QueryAPI[Key, Val]) {
  def save(k: Key, op: Operation[S, Val, E]): F[SaveResult[S, Val]] =
    for {
      old <- query.getLatestSnapshot(k)
      opResult = op.run(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

```
abstract class SaveAPI[Key, Val](query: QueryAPI[Key, Val]) {
  def save(k: Key, op: Operation[S, Val, E]): F[SaveResult[S, Val]] =
    for {
      old <- query.getLatestSnapshot(k)
      opResult = op.run(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
}
```

Let’s apply this to our users example

Saving a user record

```
trait DataAccess {
  def saveUser(u: User): F[SaveResult[Long, User]]
}
```

Saving a user record

```
def eventSourcedDataAccess(stream: UserAccountEventStream)
  (saveAPI: stream.SaveAPI[???, User]): DataAccess =
  new DataAccess {
    def saveUser(u: User): F[SaveResult[Long, User]] = {

    }
}
```

Saving a user record

```
def eventSourcedDataAccess(stream: UserAccountEventStream)
  (saveAPI: stream.SaveAPI[???, User]): DataAccess =
  new DataAccess {
    def saveUser(u: User): F[SaveResult[Long, User]] = {
      val event = InsertUser(u.id, u.name, u.username)

    }
}
```

Saving a user record

```
def eventSourcedDataAccess(stream: UserAccountEventStream)
  (saveAPI: stream.SaveAPI[???, User]): DataAccess =
  new DataAccess {
    def saveUser(u: User): F[SaveResult[Long, User]] = {
      val event = InsertUser(u.id, u.name, u.username)
      val operation = Operation[Long, User] {
        }
      saveAPI.save(???, operation)
    }
}
```

Saving a user record

```
def eventSourcedDataAccess(stream: UserAccountEventStream)
  (saveAPI: stream.SaveAPI[CompanyUsername, User]): DataAccess =
  new DataAccess {
    def saveUser(u: User): F[SaveResult[Long, User]] = {
      val event = InsertUser(u.id, u.name, u.username)
      val operation = Operation[Long, User] {
        }
      saveAPI.save((u.id.company, u.username), operation)
    }
}
```

Saving a user record

```
def eventSourcedDataAccess(stream: UserAccountEventStream)
  (saveAPI: stream.SaveAPI[CompanyUsername, User]): DataAccess =
  new DataAccess {
    def saveUser(u: User): F[SaveResult[Long, User]] = {
      val event = InsertUser(u.id, u.name, u.username)
      val operation = Operation[Long, User] {
          _.value match {
            case None =>
              OpResult.Success(event)




          }
        }
      saveAPI.save((u.id.company, u.username), operation)
    }
}
```

Saving a user record

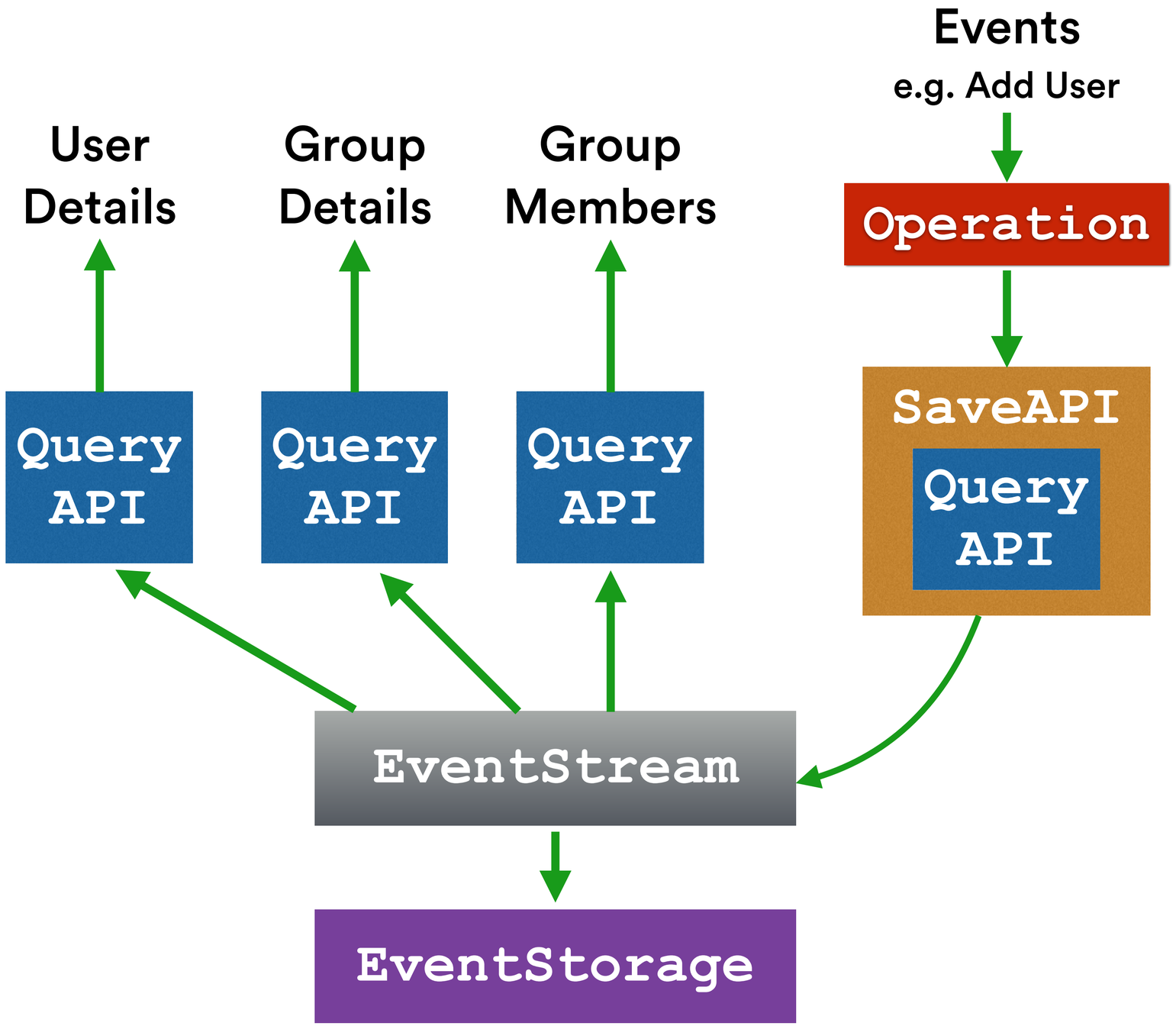
```
def eventSourcedDataAccess(stream: UserAccountEventStream)
  (saveAPI: stream.SaveAPI[CompanyUsername, User]): DataAccess =
  new DataAccess {
    def saveUser(u: User): F[SaveResult[Long, User]] = {
      val event = InsertUser(u.id, u.name, u.username)
      val operation = Operation[Long, User] {
          _.value match {
            case None =>
              OpResult.Success(event)
            case Some(x) if u.id == x.id =>
              OpResult.Success(event)


          }
        }
      saveAPI.save((u.id.company, u.username), operation)
    }
}
```

Saving a user record

```
def eventSourcedDataAccess(stream: UserAccountEventStream)
  (saveAPI: stream.SaveAPI[CompanyUsername, User]): DataAccess =
  new DataAccess {
    def saveUser(u: User): F[SaveResult[Long, User]] = {
      val event = InsertUser(u.id, u.name, u.username)
      val operation = Operation[Long, User] {
          _.value match {
            case None =>
              OpResult.Success(event)
            case Some(x) if u.id == x.id =>
              OpResult.Success(event)
            case _ =>
              OpResult.Reject(List(Reason("Duplicate username")))
          }
        }
      saveAPI.save((u.id.company, u.username), operation)
    }
}
```

Our event sourcing library so far…


Events
e.g. Add User
User Details
Group Details
Group Members
Operation
Query API
Query API
Query API
SaveAPI
Query API
EventStream
EventStorage

Step 4. Bring it all together

```
// 1. Instantiate a stream with an EventStorage
val eventStore = new DynamoEventStorage(...)
val stream = new UserAccountEventStream[Task](eventStore)
```

```
// 1. Instantiate a stream with an EventStorage
val eventStore = new DynamoEventStorage(...)
val stream = new UserAccountEventStream[Task](eventStore)

// 2. Create QueryAPIs defined for stream
val userById = new stream.UserById       // QueryAPI[CompanyUserId, User]
val userByName = new stream.UserByName   // QueryAPI[CompanyUsername, User]
```

```
// 1. Instantiate a stream with an EventStorage
val eventStore = new DynamoEventStorage(...)
val stream = new UserAccountEventStream[Task](eventStore)

// 2. Create QueryAPIs defined for stream
val userById = new stream.UserById       // QueryAPI[CompanyUserId, User]
val userByName = new stream.UserByName  // QueryAPI[CompanyUsername, User]

// 3. Create SaveAPIs defined for stream
val saveAPI = new stream.SaveAPI(userByName)
// Create DataAccess with saveUser with Operation logic
val dataLayer = new DataAccess(stream)(saveAPI)
```

```
// 1. Instantiate a stream with an EventStorage
val eventStore = new DynamoEventStorage(...)
val stream = new UserAccountEventStream[Task](eventStore)

// 2. Create QueryAPIs defined for stream
val userById = new stream.UserById       // QueryAPI[CompanyUserId, User]
val userByName = new stream.UserByName  // QueryAPI[CompanyUsername, User]

// 3. Create SaveAPIs defined for stream
val saveAPI = new stream.SaveAPI(userByName)
// Create DataAccess with saveUser with Operation logic
val dataLayer = new DataAccess(stream)(saveAPI)

val saveAndGetUser: Task[Option[User]] =
  for {
    _     <- dataLayer.saveUser(User(...))  // Task[SaveResult[User]]
    saved <- userById.query(...)
  } yield saved

request.run  // Run it!
```

Observations and next steps

Safety?

Type-safe append-only storage

Ability to query for historical values

Flexibility?

Add `QueryAPIs` for new views
of existing data

Accumulator model supports
incremental calculations e.g. hashing

Pluggable event storage

Code versus SQL...

Performance?

We’re still tuning...

with a few tricks up our sleeves:

- Snapshot caching
- Sharded snapshots
- Delivering events to appropriate tools e.g. ElasticSearch

event sourcing lib:

bitbucket.org/atlassianlabs/eventsrc