eventful Documentation

Release

David Reaver

Sep 03, 2017

Contents

`eventful` is a set of Haskell packages that are building blocks to event sourced applications. You should use `eventful` if you want to explore event sourcing, or if you already have experience with event sourcing and you want pre-built event storage and other convenient abstractions for your application.

CHAPTER 1

Introduction

`eventful` is a Haskell library for building event sourced applications.

What is Event Sourcing?

At its core, event sourcing is the following: instead of simply storing the current state of your application, you store a sequence of state-changing events. Events become the atomic unit of storage in your application, and the current state is computed from this sequence of events.

General Event Sourcing Resources

We could explain event sourcing in detail here, but if you've got far enough to google "haskell event sourcing" and you found this library, chances are you have a decent idea of what event sourcing is. In lieu of a more complete explanation, here are some great introduction materials:

- This first article isn't specifically about event sourcing per se, but it is a very compelling and thorough introduction to storing your data in a log and using logs as a communication mechanism between multiple consumers. I highly recommend starting here.
- Introductory talk by Greg Young.
- Great, but slightly dated overview by Martin Fowler.
- Article by Martin Kleppmann about stream processing in general, but also specifically about event sourcing.

Features

The goal of `eventful` is not to be a framework that imposes design decisions on application developers. It is meant to be a toolbox from which you can choose specific features to construct your application. Some features include:

- Robust event stream storage using our `EventStore` type. There are multiple backends implemented, including in-memory, SQLite, PostgreSQL, and AWS DynamoDB.
- Simple `EventStore` API so you can easily construct new event store backends.
- Convenience layer with common ES/CQRS abstractions, including `Projection`, `Aggregate`, and `ProcessManager`. All of these integrate with `EventStore` so you get transparent integration with your underlying event storage.
- Extremely flexible serialization system. All `eventful` components that do serialization use a type parameter called `serialized`. You provide the `serialized` type and functions to serialize/deserialize your events, and we handle storage to one of the available backends.
- The `EventStore` type exposes the monad it operates in as a type parameter; we don't force the use of any particular monad stack on you. If your event store's monad supports transactions, like `SqlPersistT` or `STM` does, then you get transactional semantics for free.
- `eventful` aims to use the most vanilla Haskell possible. We prefer value types over type classes, avoid any type-level computations, etc. This makes the core API extensible and easy to understand. If you want, you can easily add more advanced type system features to your application while still maintaining the ability to use the core `eventful` constructs.

CHAPTER 2

Tutorial

This tutorial will introduce new users to the concepts behind `eventful`. Additionally, once you read this tutorial you will know how to build a simple event sourced application using `eventful`.

Manipulating state with events

The core concept behind event sourcing is your current state should be derived from past events. To illustrate this, we will use an extremely simple example of an integer "counter". The user can increment, decrement, or reset the counter to zero. The purpose of this example is not to present a compelling business use case for event sourcing. It is here to simply introduce event sourcing.

Counter State

The state for our counter is laughably simple:

```haskell
module Counter where

import Eventful

newtype Counter = Counter { unCounter :: Int }
  deriving (Show, Eq)
```

Our `Counter` is just a simple newtype wrapper around an `Int`.

In a non event sourced world, interacting with the `Counter` would probably involve a few functions on the `Counter`:

```haskell
incrementCounter :: Counter -> Int -> Counter
incrementCounter (Counter count) amount = Counter (count + amount)

decrementCounter :: Counter -> Int -> Counter
decrementCounter (Counter count) amount = Counter (count - amount)
```

```haskell
resetCounter :: Counter -> Counter
resetCounter _ = Counter 0
```

You could imagine these functions being wrapped in some sort of CLI or a REST API. Note how the functions are written in an imperative tone, and they directly modify the state. You could imagine the integer representing the count being stored directly as in integer in some database.

Moving to events

Now what if your boss comes up to you one day and says "hey, we think users often make mistakes when incrementing the counter. We want to know how often an increment is followed by a decrement of a smaller amount." With our current model, this is simply not possible without some detective work (hello log analysis! That is, if you have logs...).

If we had stored our state changes as events, we could easily give our boss what he/she wants! What would some events look like in our case? How about this:

```haskell
data CounterEvent
  = CounterIncremented Int
  | CounterDecremented Int
  | CounterReset
  deriving (Show, Eq)
```

Note the parallels with our previous state modifying function. In this case, we use the *past* tense to describe events. That is, an event is something that has already occurred, and we are simply storing that fact.

Using events to replay state

So we have some events, how do we use them? If events are records of what happened in the past, then we want out internal state to be a function of these facts. Let's write a function that can handle each event:

```haskell
handleCounterEvent :: Counter -> CounterEvent -> Counter
handleCounterEvent (Counter count) (CounterIncremented amount) = Counter (count +␣
↪amount)
handleCounterEvent (Counter count) (CounterDecremented amount) = Counter (count -␣
↪amount)
handleCounterEvent _ (CounterReset) = Counter 0
```

Easy right?

Now, let's introduce the concept of a `Projection` in eventful. First we'll create one for a `Counter` and then we can discuss details:

```haskell
counterProjection :: Projection Counter CounterEvent
counterProjection =
  Projection
  { projectionSeed = Counter 0
  , projectionEventHandler = handleCounterEvent
  }
```

A `Projection` is a pair of a "seed" and an event handler. A seed is simply the default value for the projection; we always have to know what to start with when we don't have events. The event handler tells the projection how to apply events to state. Note that the projection has two type parameters for the state and event types.

Convenience functions for `Projection`

`eventful` comes with some convenience functions to rebuilt the current state for a `Projection` from a list of events, and to show all `Projection` states.

```
myEvents :: [CounterEvent]
myEvents =
  [ CounterIncremented 3
  , CounterDecremented 1
  , CounterReset
  ]

myLatestCounter :: Counter
myLatestCounter = latestProjection counterProjection myEvents
-- Counter {unCounter = 0}

allMyCounters :: [Counter]
allMyCounters = allProjections counterProjection myEvents
-- [ Counter {unCounter = 0}
-- , Counter {unCounter = 3}
-- , Counter {unCounter = 2}
-- , Counter {unCounter = 0}
-- ]
```

Event Stores

Using events to change state is no good unless we can actually persist the events somewhere. In `eventful`, we do that using an EventStore. Before diving into the API, lets discuss some concepts related to event streams.

Streams of events

Events don't exist in a vacuum; in most real-world scenarios the events we receive have some natural association with other events. An example unrelated to event sourcing is a stream of pricing data from a particular stock in the stock market. For example, we could have a stream of bid quotes from Google's stock (GOOG):

```
{
  "price": 34.5,
  "time": "2017-05-17T12:00:00",
  "instrument": "GOOG"
}
{
  "price": 34.7,
  "time": "2017-05-17T13:00:00",
  "instrument": "GOOG"
}
{
  "price": 34.9,
  "time": "2017-05-17T14:00:00",
  "instrument": "GOOG"
}
```

There are a couple notable properties from this stream of events:

- The stream has an identity. In this case, it is "bid quotes for GOOG". If we were to store this stream in a database, a natural primary key would be the string `"GOOG"`.

- There is a natural ordering among the events; they can be ordered by `"time"`.

Event sourced streams of events

In event sourcing, it's natural to think of the events for a particular piece of state (a `Projection`) as a stream. Following the lead of the example above, we can give the stream an identity and also a natural ordering:

- It is common to use a UUID to identify event sourced state streams.
- For each stream, we can order the events by a strictly increasing sequence of integers. In `eventful`, this is represented by the EventVersion type.

Here's an example of a possible event stream for our Counter:

```
{
  "uuid": "123e4567-e89b-12d3-a456-426655440000",
  "type": "CounterIncremented",
  "amount": 3,
  "eventVersion": 0
}
{
  "uuid": "123e4567-e89b-12d3-a456-426655440000",
  "type": "CounterDecremented",
  "amount": 1,
  "eventVersion": 1
}
{
  "uuid": "123e4567-e89b-12d3-a456-426655440000",
  "type": "CounterReset",
  "eventVersion": 2
}
```

Basic `EventStore` usage

The `EventStore` interface in `eventful` has two primary functions:

- `storeEvents`: Store a list of events to a given stream specified by the `UUID`
- `getEvents`: Retrieve events from the given `UUID` stream

Simple right? There are multiple event store backends included in `eventful`. In the following example we are going to use the in-memory store from `eventful-memory`.

The event store type `EventStore serialized m` has two type parameters:

- `serialized` is the serialization type. In our case, we don't really need to serialize so we can just use `CounterEvent`.
- `m` is the monad the event store operates in. For the in-memory store, that is the `STM` monad.

```
{-# LANGUAGE ScopedTypeVariables #-}

module EventStore where

import Control.Concurrent.STM
import Eventful
import Eventful.Store.Memory

import Counter
```

```
counterStoreExample :: IO ()
counterStoreExample = do
  -- First we need to create our in-memory event store.
  tvar <- eventMapTVar
  let
    writer = tvarEventStoreWriter tvar
    reader = tvarEventStoreReader tvar

  -- Lets store some events. Note that the 'atomically' functions is how we
  -- execute STM actions.
  let
    uuid = read "123e4567-e89b-12d3-a456-426655440000"
    events =
      [ CounterIncremented 3
      , CounterDecremented 1
      , CounterReset
      ]
  _ <- atomically $ storeEvents writer AnyVersion uuid events

  -- Now read the events back and print
  events' <- atomically $ getEvents reader (allEvents uuid)
  print events'
```

Output:

```
[ StreamEvent
  { streamEventProjectionId = 123e4567-e89b-12d3-a456-426655440000
  , streamEventVersion = EventVersion {unEventVersion = 0}
  , streamEventEvent = CounterIncremented 3
  }
, StreamEvent
  { streamEventProjectionId = 123e4567-e89b-12d3-a456-426655440000
  , streamEventVersion = EventVersion {unEventVersion = 1}
  , streamEventEvent = CounterDecremented 1
  }
, StreamEvent
  { streamEventProjectionId = 123e4567-e89b-12d3-a456-426655440000
  , streamEventVersion = EventVersion {unEventVersion = 2}
  , streamEventEvent = CounterReset
  }
]
```

This section of the tutorial obviously glossed over many details of the `EventStore`. The main part of the documentation will cover those details.