Extracted from:

From Objects to Functions

Build Your Software Faster and Safer with Functional Programming and Kotlin

The Pragmatic Bookshelf

Dallas, Texas

The Pragmatic Programmers

From Objects to Functions

Build Your Software Faster and Safer with Functional Programming and Kotlin

Uberto Barbini

edited by Adaobi Obi Tulton

From Objects to Functions

Build Your Software Faster and Safer with Functional Programming and Kotlin

Uberto Barbini

The Pragmatic Bookshelf
Dallas, Texas

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Adaobi Obi Tulton
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-845-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2023

Accessing the Database with Monads

It’s now time to put the monads to work on the database library we started before. Using the ContextReader we wrote in the previous chapter, we can now combine the read and write operations in the same transaction:

```
  val listUpdater = readRow("myRowId")
    .transform { r-> r.copy(active = false) }
1   .bind { r -> writeRow(r) }

2 runInTransaction(listUpdater).expectSuccess()
```

1. With the bind method, we’re able to make the writeRow call compile and work correctly without using a non-local return.

2. Until we run the ContextReader in a transaction, ContextReader is only maintaining a record of the operations we intend to carry out on the database, without performing them. All the actual database calls happen in this line.

Now that we have the functions to access the database, and we have a way to read and write from it safely and in a functional-friendly way, we need to put all this together.

First, we have to finish implementing the readRow and writeRow functions, using a ContextReader working with the transaction:

```
1 typealias TxReader<T> = ContextReader<Transaction, T>

2 fun readRow(id: String): TxReader<ToDoListProjectionRow> = TxReader { tx ->
3     toDoListProjectionTable.selectWhere(tx, toDoListProjectionTable.id eq id)
          .map { it[toDoListProjectionTable.row_data] }
4         .single()
  }

5 fun writeRow(row: ToDoListProjectionRow): TxReader<Unit> = TxReader { tx ->
6     toDoListProjectionTable.insertInto(tx) { newRow ->
          newRow[id] = row.id.toRowId()
          newRow[row_data] = row
      }
  }
```

1. We start by defining a type alias for our convenience.

2. We wrap the result of readRow inside a transaction reader.

3. Then, we select all rows from the projection table where the row ID is equal to requested ID.

4. Finally, we check that the query must return a single row.

5. We wrap writeRow inside a reader in the same way.
6. We insert a new row on the projection table using the transaction from the reader.

To complete our task, we need to implement the ContextProvider for the transaction. The goal here is to control the way transactions are used by the database, so that we roll them back in case of errors, and we always close the database connection. We also want to decide the isolation level to use to run the reader case by case:

```
data class TransactionProvider(
            private val dataSource: DataSource,
            val isolationLevel: TransactionIsolationLevel,
            val maxAttempts: Int = 10): ContextProvider<Transaction> {

  override fun <T> tryRun(
        reader: ContextReader<Transaction, T>): Outcome<ContextError, T> =

1    inTopLevelTransaction(
2          db = Database.connect(dataSource),
          transactionIsolation = isolationLevel.jdbcLevel,
          repetitionAttempts = maxAttempts) {

3        addLogger(StdOutSqlLogger)

        try {
4            reader.runWith(this).asSuccess()
        } catch (t: Throwable) {
5            rollback()
            TransactionError("Transaction rolled back: ${t.message}", t)
                .asFailure()
        }
    }
}
```

1. inTopLevelTransaction from Exposed does exactly what we need here.
2. We pass the database connection, the isolation level, and the max attempts parameters from the constructor.
3. Exposed will log out all the SQL commands to the console.
4. We run our reader inside a try...catch block. Note that runWith is a field storing a function, not a method of the ContextReader.
5. In case of exceptions, we'll rollback the transaction.

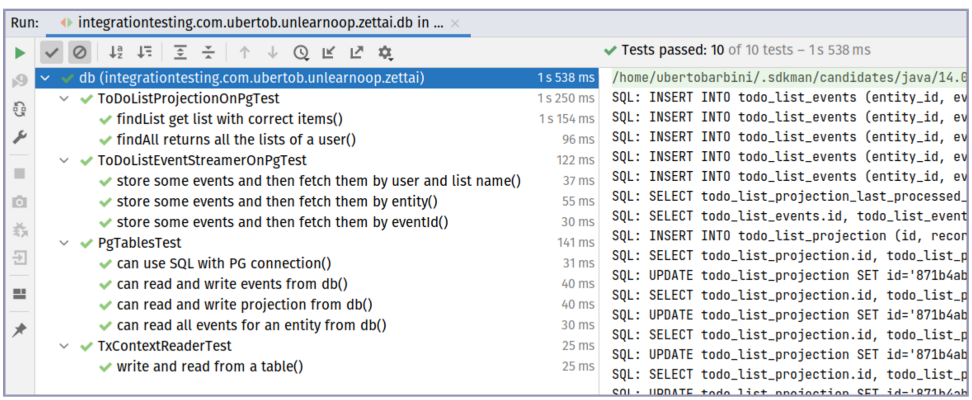
We can now successfully run the full test on the projection row with actual code that can run on the database:

```
class TxContextReaderTest {
```

```
    @Test
    fun `write and read from a table`() {

        val user = randomUser()
        val expectedList = randomToDoList()
        val listId = ToDoListId.mint()
        val row = ToDoListProjectionRow(listId, user, true, expectedList)

        val listReader: TxReader<ToDoList> =
            writeRow(row)
                .bind { readRow(listId.toRowId()) }
                .transform { row -> row.list }

        val list = transactionContextForTest().tryRun(listReader)
            .expectSuccess()

        expectThat(list).isEqualTo(expectedList)
    }

}
```

We need to start the PostgreSQL Docker container or another database instance before running the tests, otherwise, they'll fail.

If everything has been set up correctly, we can now successfully run our test and see the generated SQL commands in the console, as in this figure:



EventStreamer with ContextReader

With all the necessary components in position, we're now able to run our event store in a database instead of relying on in-memory maps. What's more, our persistence framework has been designed in a manner that allows for its operation in memory, with a database, or with alternative persistence solutions, provided they can be incorporated within a ContextReader.

In other words, we defined an algebra of data and functions to manage the persistence of our system using functional effects.

Let's briefly recap where we are. Every domain operation that needs some kind of persistence should return a ContextReader. We can then combine them, and once we assemble enough pieces to complete a task—something that should atomically work or fail—we can run it in a ContextProvider to obtain the final result or a detailed error.

All the domain logic should ignore the actual context that will be used, because it will be something injected from an outside adapter (see Separating the Domain from the Infrastructure, on page ?).

Let's look again at our EventStreamer interface. Its role is to read and write events to the repository. It shouldn't know about entity and the rest of the model:

```
interface EventStreamer<E : EntityEvent, NK: Any> {
    fun fetchByEntity(entityId: EntityId): List<E>?
    fun fetchAfter(eventSeq: EventSeq): Sequence<StoredEvent<E>>
    fun retrieveIdFromNaturalKey(key: NK): EntityId?
    fun store(newEvents: Iterable<E>): List<StoredEvent<E>>
}
```

Typically, each entity has a *natural key* that should be unique, like the combination user and list name in Zettai. A good practical consideration is to add a method to retrieve events using the natural key from our database. We could use projections for this, but it's faster and safer to directly query the events.

Joe asks:

Why Is Using a Projection Not Safe?

The problem is that the projections are created by observing the events created by the command handler. So, there is always a risk that they aren't completely up-to-date.

This is called eventual consistence, and it's usually not a problem for the read model, but it can be problematic for the event store. Depending on the domain, the risk can be quite small or not of much consequence. But in general, it's better to avoid having the write model depend on projections.

It's now time we put into practice what we learned about monads! We need to change the interface to return ContextReader, also making the EventStreamer generic over the context:

```
interface EventStreamer<CTX, E : EntityEvent, NK : Any> {
  fun fetchByEntity(entityId: EntityId): ContextReader<CTX, List<E>>
  fun fetchAfter(eventSeq: EventSeq): ContextReader<CTX, List<StoredEvent<E>>>
  fun retrieveIdFromNaturalKey(key: NK): ContextReader<CTX, EntityId?>
```

```
  fun store(newEvents: Iterable<E>): ContextReader<CTX, List<StoredEvent<E>>>
}
```

Rewrite the In-Memory Event Streamer

Rather than writing a new database event streamer for the database from scratch, it's preferable to split the work into two parts: first, we convert the current streamer to use the ContextReader operating with a list of events in memory, and second, we can migrate it to an external database.

By proceeding in this way, we can validate each step separately and minimize the potential for errors.

For this we need to move the event lists from the in-memory event streamer to the ContextProvider for in-memory events:

```
typealias ToDoListInMemoryRef = AtomicReference<List<ToDoListStoredEvent>>
1 typealias InMemoryEventsReader<T> = ContextReader<ToDoListInMemoryRef, T>

class InMemoryEventsProvider() : ContextProvider<ToDoListInMemoryRef> {
2     val events = AtomicReference<List<ToDoListStoredEvent>>(listOf())
    override fun <T> tryRun(reader: InMemoryEventsReader<T>) =
        try {
3             reader.runWith(events).asSuccess()
        } catch (e: Exception) {
            ToDoListEventsError("Operation failed: ${e.message}", e)
4                 .asFailure()
        }
}
```

1. First, we define the alias for the in-memory events reader.
2. The list for events is now in the in-memory provider; it will be shared to all the readers.
3. Here, we run the reader inside a try...catch block as inside the transaction provider. In this way, we're sure that no exception can leak outside.
4. In case of exception, we return a failure with the exception details.

We also need to adapt the EventStreamerInMemory using the list from the context instead of the private field. Let's just look at the store method since the rest are quite similar:

```
class EventStreamerInMemory : ToDoListEventStreamer<ToDoListInMemoryRef> {

    override fun store(newEvents: Iterable<ToDoListEvent>) =
        InMemoryEventsReader { events ->
            newEvents.toSavedEvents(events.get().size.toLong())
```

```
                .also { ne -> events.updateAndGet { it + ne } }
        }
//... similar changes to rest of the methods

}
```