Frontend based on Functional Reactive Programming

Introduction

Our programs execute code that fires thousands of events, which need to be prepared to handle them effectively. This creates two important challenges: scalability and latency. For example, getting data from a server or running any deferred computation, or getting data from multiple remote locations.

The solution to manage these situations effectively is to choose the paradigm which suits these types of problems best.

To solve these problems, there are two emerging paradigms: functional programming (FP) and reactive programming (RP). This exhilarating composition is what gives rise to Functional Reactive Programming (FRP). One of the libraries that encodes that paradigm is RxJS, which deals with asynchronous and event based data sources effectively.

Limitations of the current solutions

Synchronous versus asynchronous computing

The main factor that separates the runtime of synchronous and asynchronous code is latency, also known as “wait time.” It’s much easier to reason about solutions when the execution occurs synchronously in the same order as you’re writing it, but the world of software does not grant that.

The traditional approach of having idle applications waiting for an operation is simply not acceptable; we need to take advantage of asynchronous execution to get our application always responsive.

blocking code

Synchronous execution occurs when each block of code must wait for the previous block to complete before running.

Writing code this way is much easier to maintain and debug; unfortunately, in long running tasks like a remote HTTP call, synchronous execution is an awfully bad design choice, because it causes the entire application is waiting for the data to be loaded and wasting computing cycles that could easily be executing any other code.

User interfaces could never be implemented this way. Not only does it create a terrible user experience, but also browsers may deem your scripts unresponsive after a certain period of inactivity and terminate them.

Non-blocking code with callback functions

Using functions as callbacks have been the key of JavaScript development for years.

Callback functions were created to tackle the problem of blocking for long running operations, providing a handler function which will be invoked once the data it’s ready for use. In the meantime, your application can continue carrying out any other task.

Callbacks creates an inversion of control that permits your application to continue executing the next lines of code. Inversion of control in this sense refers to the way in which certain parts of your code receive the flow of control back from the runtime system.

Understanding execution types

Asynchronous functions allow us to stay responsive, but they come at a price: where synchronous programs allow us to reason directly about the state of the application, asynchronous code forces us to reason about its future state.

When tasks complete at different times, it’s very difficult to guarantee how they’ll behave together. Functions that terminate at unpredictable times are typically harder to deal with. In this case, the application state would be determined by the order in which the tasks are called. These conditions are known as side effects. Functions with side effects can perform very unreliably when run in any arbitrary order. In functional programming, we try to limit the presence of side effects in our code by using pure functions.

When we execute asynchronous code, our tasks have a temporal dependency, we need to implement mechanisms to synchronize the responses of these tasks.

Can callbacks solve the temporal dependency?

Consider a scenario where we need three http calls and merge their responses to present the information. A possible solution for this, which requires the use of three nested callback functions. As a result, our model management starts to grow horizontally. That solution is known in the JavaScript world as "callback hell," and will become an unmaintainable code which can cause some really unpredictable and hard to diagnose bugs.

Event Emitters

Event emitters are very popular mechanism for asynchronous event-based architectures.

Subscribing to an event emitter is done through the addListener() method, which allows to provide the callback that will be called when an event of interest is fired. Unfortunately, with event emitters we run the risk of losing events if we don't start listening at the right time, and the callbacks to handle that emitted data, presents all of the same problems described above.

Promises

JavaScript ES6 introduces the Promise class to represent any asynchronous computation that is expected to complete in the future. With promises we can chain together a set of actions with future values to form a continuation.

A Promise is a data type that wraps an asynchronous or long- running operation, a future value, with the ability to subscribe to its result, or its error, respectively. Because we can't alter the value of a promise once it's been executed, it's actually an immutable type, which is a functional quality we seek for in our programs.

The drawback of using promises, is that they're unable to handle data sources that produce more than one value, like mouse movements. Promises are also missing important features like debouncing, throttling, and the ability to retry from failure. The most important downside is that promises can't be cancelled.

Promises and event-emitters are solving what are essentially the same problems in slightly different ways. The result of this is that in many scenarios a developer must use both in order to accomplish their goal, which can often lead to disjointed and confusing code.

RxJS and FRP

RxJS can resolve all problems described above, because abstracts out the notion of latency away from our code while allowing us to model our solutions using a linear sequence of steps through which data can flow over time.

RxJS treats all possible data sources in exactly the same way, as data streams: containers or wrappers of data very similar to an infinite Array.

These streams produce various forms of data to be consumed by an application. And exist data consumers, the entities that subscribe to these events and will do something with data that they receive.

A stream remains dormant and nothing has actually happened, until there's a subscriber that listens for it. This is very different from promises, that execute their operations as soon as are created. Streams are lazy data types.

RxJS takes inspiration from Publish-Subscribe methodology, but adds a few extras like signals that indicate when a stream has completed, lazy initialization, cancellation, resource management, and disposal.

RxJS encourages a style of development known as data-driven programming. The data-driven approach is a way of creating applications such that we can separate the behavior of an application from the data that is passing through it.

In the object-oriented approach, we place more emphasis on the supporting structures rather than the data itself.

Separating data from the behavior of the system is at the heart of data-driven/centric design. Similarly, loosely coupling functions from the objects that contain data is a design principle of functional programming and, by extension, reactive programming.