

Micromachine

F# <3 Events

About Us



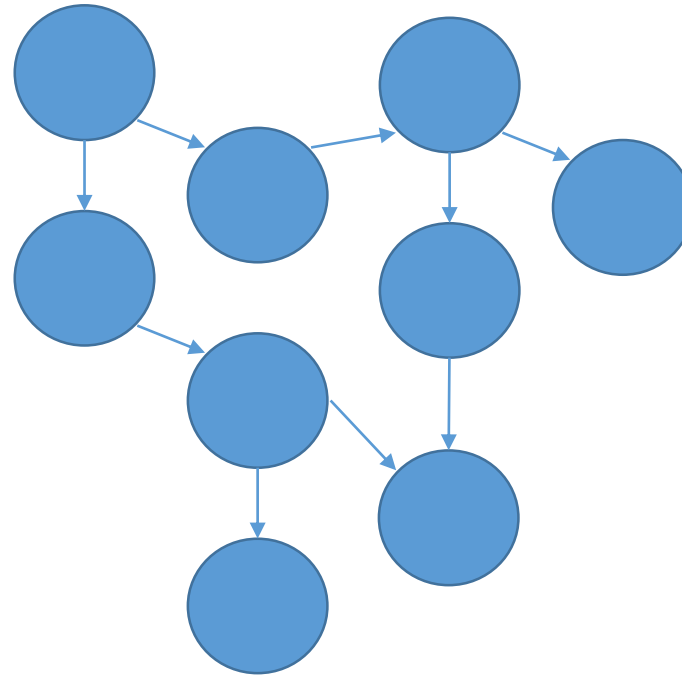
Gina Maini is a functional programmer from unorthodox backgrounds of web development and unikernel systems development in OCaml. At Jet.com, Gina works on core-infrastructure and spends her daytime hours thinking about logs.



Brian Mitchell likes to work with databases and distributed systems. He's a polyglot at heart and thinks there should be a programming language for everything. He's working on event sourced infrastructure at Jet.com.

What is a microservice?

Maybe if I make this more complicated I'll have better job security.



A microservice is more about working toward a desired outcome

- Isolated logic
- Units of scaling
- Units of failure
- Operational flexibility

Microservices are just as much
about architectural intuitions as
they are about rules

Back to F#:

Functional Programming in the Large

How do we go from simple and small functions to a working system?
We need something to help us structure larger functional programs.

“Life was simple before World War II. After that, we had systems.”

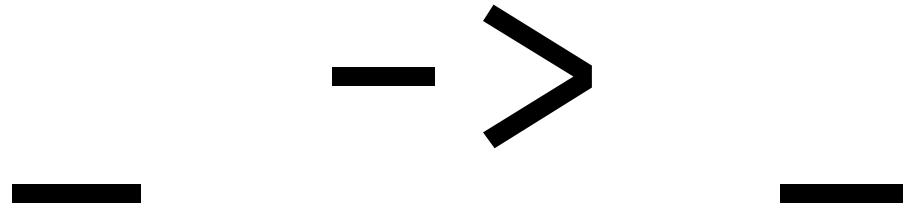
- Rear Admiral Grace Hopper

Events & Event Sourcing

- Modeling change in an immutable environment is natural with events
- Event sourcing is a functional modeling tool
- Events are compositional and allow a domain to be enriched in a modular way

Classifying Event Driven Microservices

We can classify common “arrows”:



Interactive Services

This kind of service is usually represented by some kind of message or request handling API. Users or other systems trigger effects or send commands.

$$\text{Context} \rightarrow \text{Command} \rightarrow \text{Event}[]$$
$$\text{AsyncSeq} < \text{Context} * \text{Command} > \rightarrow \text{AsyncSeq} < \text{Event}[] >$$

AsyncSeq Playground

Open the **playground/asyncseq.fsx** script and read through the examples and try adjusting them.

Note the **async {...}** expressions wrapping our code. Why do you think this matters? What is a useful side effect of asynchronous flow?

Okay. Let's build a
microservice already!

Things we'll need:

1. A model of our events for our domain
2. A way to store and retrieve these events
3. A way to drive our interactions

I. Modeling events for a survey application

We'll want to create and edit our surveys as well as publish them and close them. We'll model the state of the survey with events.

Maybe lets draw this. What might be some invalid states? Let's uncomment some tests and see that we can interpret our events safely.

What about the contents of our survey? Let's build up a working structure which contains the proper state. Implement the suggested fields in our survey type, or if you want, some fields of your choosing.

II. Review

We can fold or “reduce” over our sequence to build up our states and at each step we can decide if an event may or may not be valid in our domain.

We can always go back to prior revisions of our state. This historical record is an effective way to lift immutability into our database as well as our code.

Records and discriminated unions are useful types to help describe our domain model and arrows (functions) are useful to understand transitions and transformations between them.

II. Storage and Retrieval

There are many ways to do this but we'll pick something that's likely already familiar. We'll use JSON to serialize and deserialize our events.

How do we map JSON into these rich F# datatypes? Let's check our tests and see what we have and see if we can get the tests passing. Advanced users are welcome to use their preferred JSON library.

Tip: If you get stuck with a confusing error, try adding explicit type annotations to your code to force the compiler to be more specific.

Installation: EventStore

For the next part, we'll want to get EventStore installed. We can download it from:

<https://geteventstore.com/> and run `EventStore.ClusterNode.exe`

(mono users should remember to run it as **mono EventStore.ClusterNode.exe**)

OR if you have docker installed:

```
docker run -p 2113:2113 -p 1113:1113 eventstore/eventstore
```

Once installed we can test it by visiting: <http://localhost:2113/>

II. Review

- We can use EventStore as our database of immutable events
- Each entity we want to describe with events gets a unique stream identifier
- All events are specifically ordered within a stream

III. Lets Build Some Surveys

We'll use interactive F# to drive our code. Go ahead and check the service directory now and run the example service.

Check out the data in the user interface and verify that you're seeing the events properly stored and retrieved.

III. Review

We have seen that we can run code against our API quite easily. We don't necessarily need to shove an HTTP interface in front of everything right away.

This makes it easy to interact and revise code. It also helps avoid coupling with things like HTTP semantics. (Avoid RPC when message passing will do).

Storing durable copies of events is a viable asynchronous communication/signaling mechanism and we'll see more of this later.

Aside: Building and Deploying

Why not just use scripts? We can easily split and recombine service & process boundaries.

But you say it's just an .fsx file and my server only runs compiled artifacts!

It's fine. We can use the compiler service and fake. Check the build script out. This gives us tremendous flexibility and helps avoid the inherit complexities of large msbuild solutions. It's also quicker for development since we can simply run our scripts and build libraries as needed.

Stream Enrichment and Effects

Discussion: Survey Results

We can consider stream enrichment works by adding some data into other streams which we may want to compute with and store.

We'll allow survey responses to be taken and then each response gets compiled into a survey result and stored. This enrichment process is a common scenario when moving between in bounded contexts.

Since this is a bit more involved, we'll keep this section open format for discussion.

Future

Other problems to consider and solve:

- Tracing microservices with events
- Evolving your schema
- Refining the publishing and subscription abstractions
- Caching and mechanical sympathy
- More testing, perhaps with properties
- Beautiful interfaces by propagating events between the client and server
- Ephemeral events vs durable events
- Inter-stream ordering and clocks