Slide 1

Add download link here. Write it down on the whiteboard as well. Quick aside, I learned recently that early computer programs were sometimes called machines. I've written a tiny support library for us called micromachines, borrowing and modifying some real-world code.

Slide 2

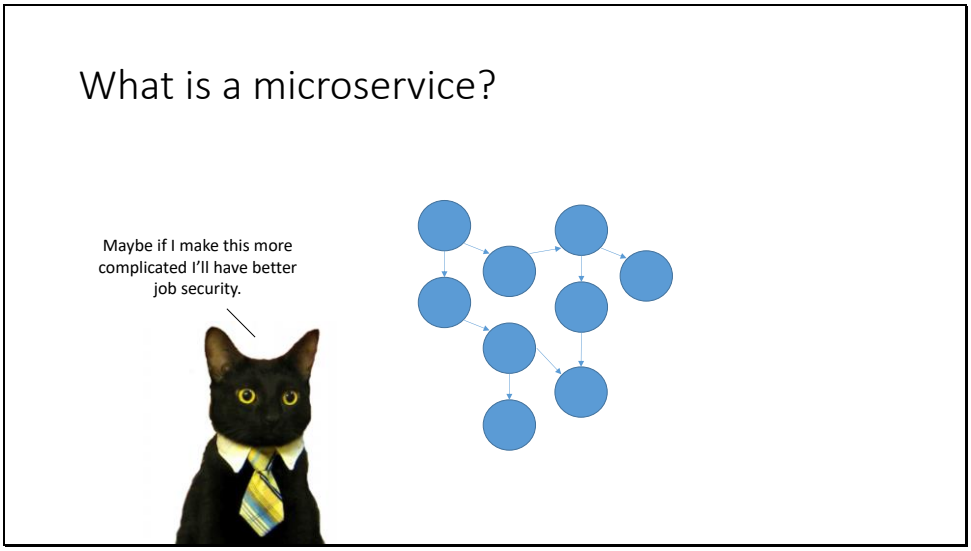
About Us

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.

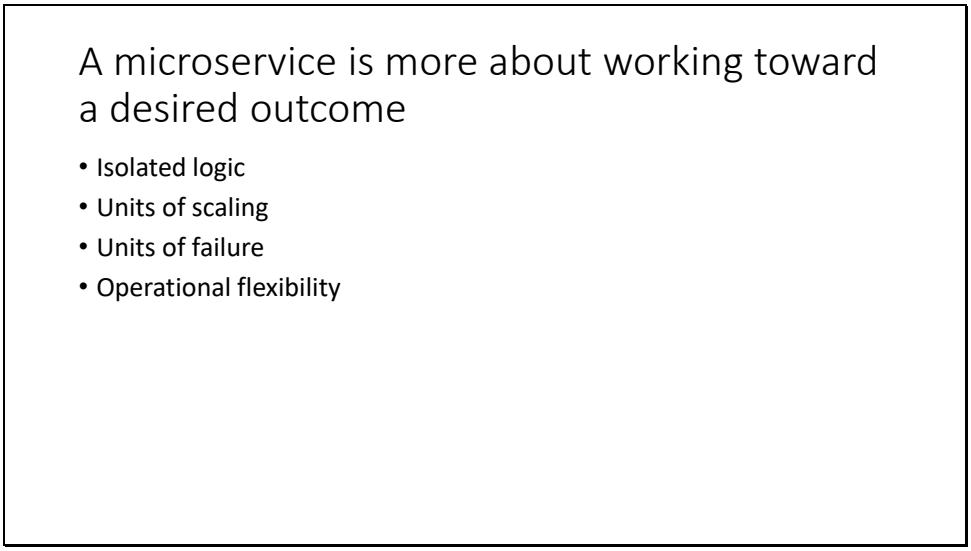
Introduce ourselves. Talk about our backgrounds and what we did before learning F#. Ask where the audience is coming from.

Slide 3



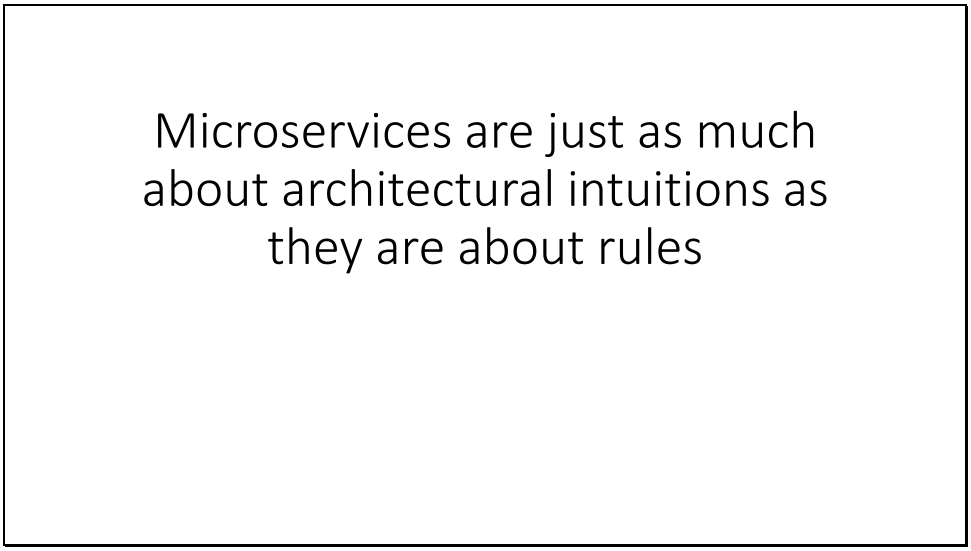
Microservices are not necessarily about having a giant graph of interacting components. Trying to immediately create a system of tiny microservices may be harmful over-engineering– it's best to think of your domain and business logic and tailor your structure accordingly.

Slide 4



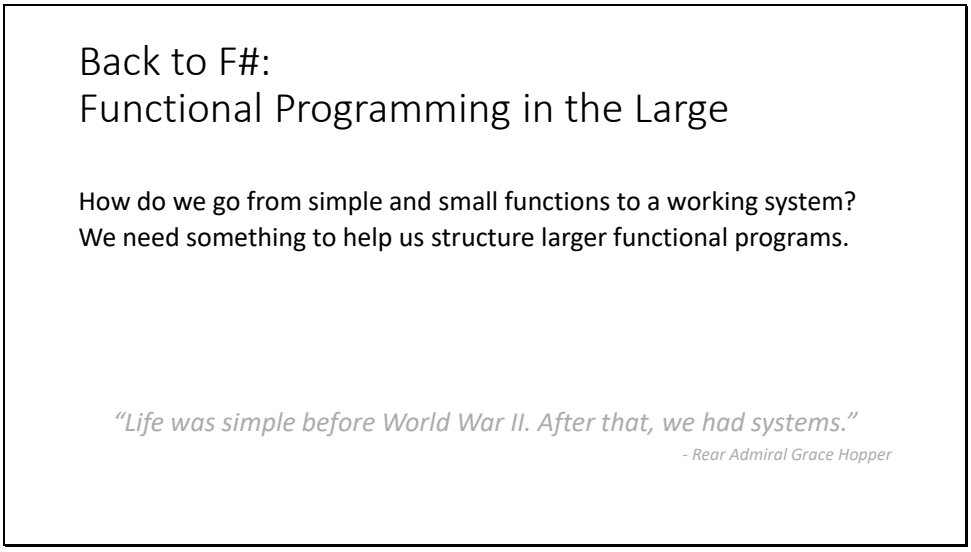
Step through each of these benefits and talk about the trade-offs. For example, with operational flexibility, we might ask if it's required that code runs on separate machines/vms/processes. The result of these is that your service design should mirror organizational trade-offs (one team might need different characteristics than another).

Slide 5



We'll focus on building code that allows us to apply many strategies w/o picking only one specific setup. No more monolith vs microservice discussions since it's really only relevant when it's painted as an all or nothing trade. This aspect of software is about bringing the technology to the human and not the human to the technology. Be sympathetic to your limited brain! Breaking the system down helps us fit this into our mind.

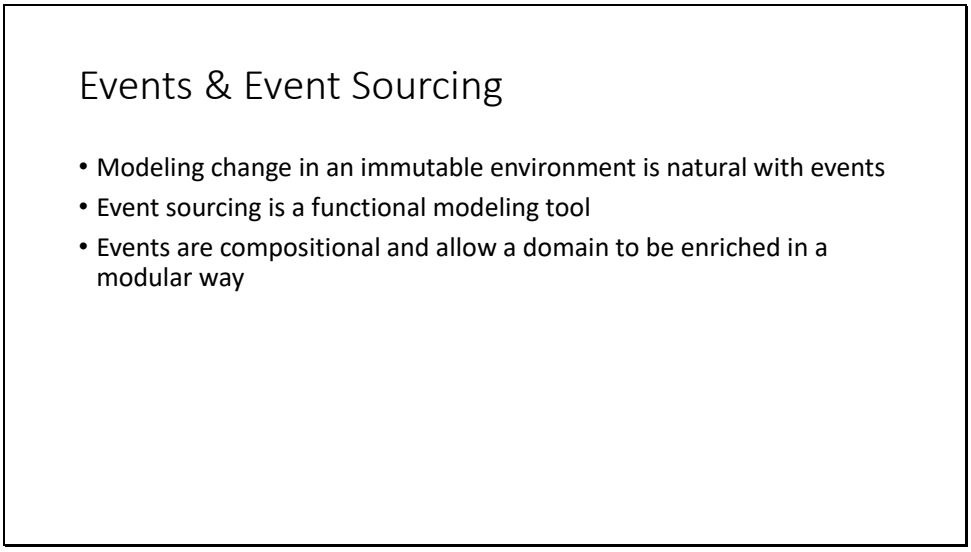
Slide 6



Thought Experiments:

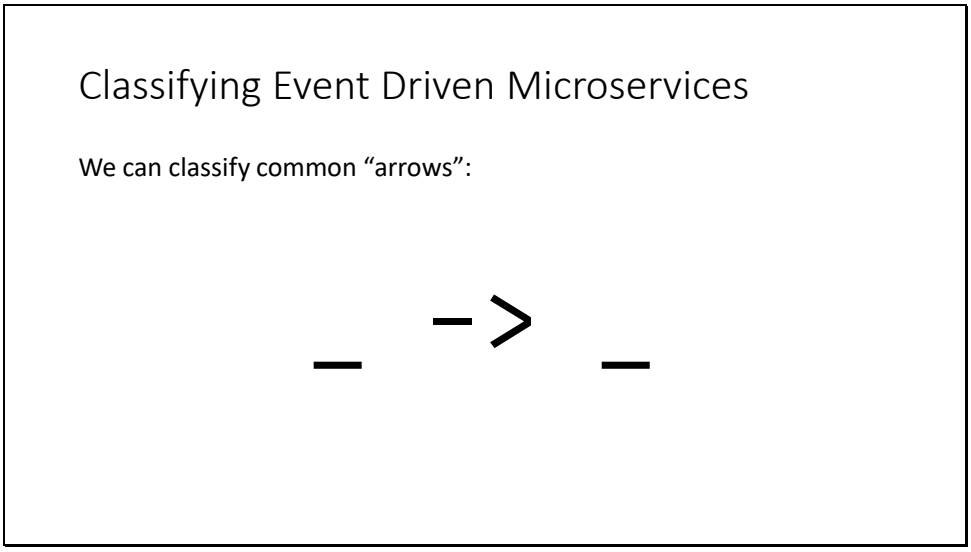
How do you suggest building large scale, functional codebases?
What sorts of techniques do people use to deal with the growth and complex interaction of functions and types in their system?

Slide 7



Mention the different between durable and ephemeral events. We gain determinism by using a stream concept rather than partially ordered protocols.

Slide 8



If you are not used to thinking in terms of arrows, it’s a good idea to take a chance and consider the most basic functional abstraction.

Thought Experiments:

What happens if the left side is an event?
What sorts of functions operate on an event and what might an output be?
Can we think of what might happen if we have more than one event as input?

Slide 9

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.

```
Context -> Command -> Event[]
```

```
AsyncSeq<Context * Command> -> AsyncSeq<Event[]>
```

Explain interactive services to the audience. Talk about what context a command is evaluated in. These are the most common in the wild and the callback form above is the most common view. The alternative is the one sequence transformation.

Slide 10

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?

Explain AsyncSeq with audience. This should be a quick aside. Mention why AsyncSeq<_> is different from seq<_>.

Slide 11

Introduce the survey concept.

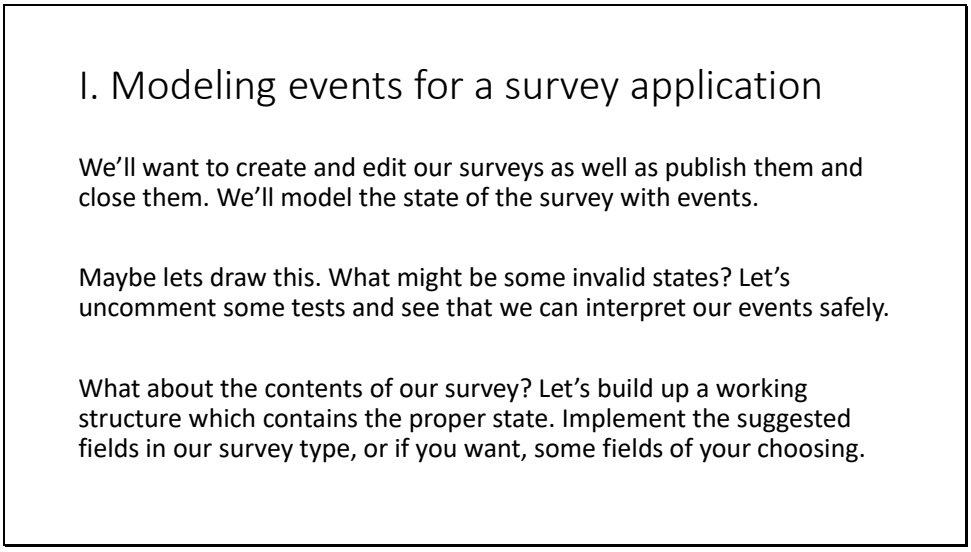
Slide 12

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

Let's make sure we know what we're getting into.

Slide 13



This will teach us how to fold a sequence of events. We'll use the prewritten test cases to write up some validation and state building functions.

Slide 14

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.

Slide 15

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 serialized 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.

JSON can be tricky to deal with. Let's get some practice by using the partially complete stubs.

Slide 16

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/

Slide 17

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

Note the similarity to a key-value store. The stream-events work in a similar way when given a function.

Thought Experiments:
What might happen if we have multiple ways to interpret events?
Why might this be good? Why might it be bad?

Advanced side note: The total ordering of events within a stream allows stronger guarantees to be made which allow more kinds of functions to be used with our events.
What might happen if we allowed events to be reordered? Would it make some functions break?

Slide 18

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.

We'll use a simple interactive stream API from our EventStore code.

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.

Slide 20

Aside: Building and Deploying

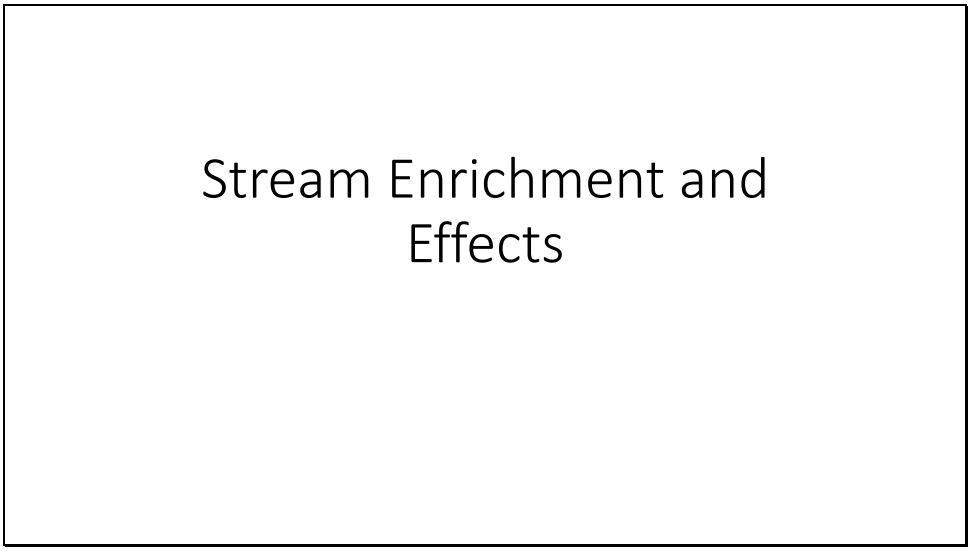
Why not just use scripts? We can easily split and recombine service & process boundaries.

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.

We'll not dwell on this but it's good to note that it can be as easy as zipping up the build folder and shipping it off somewhere to be run. It's also fine to use traditional console projects if you prefer. Pick what you like! That's one of the nice things about having all of these options with F# tooling.

Slide 21



Discuss with audience what stream enrichment might look like in this example. The term “stream enrichment” comes from the Kafka Streams project, but is generally applicable.

Slide 22

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.

Slide 23

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