Spain

Blog Post

Server-sent events with fanout.io

SWORD
UPGRADE YOUR BUSINESS

At Sword we build regular stateful business applications with an event driven architecture. At the core of this sit Event Sourcing and CQRS. The UI is always a reactive web application, which sends commands to aggregates. A command may have an effect on an aggregate instance. A published event reflects the change. Any other component can listen and do something meaningful with it.

We wanted to extend this one-way event driven data flow to the UI. This is how we ended up using Server-Sent Events. When the UI sends a command to an aggregate instance it doesn't wait for a response. The service stores the command in a durable way. It then replies with the status code 202 (Accepted). The effect of a command goes back to the UI in the form of an event. The UI reacts to that in the same way as it would for the internal actions. The data store would get updated, and some part of the UI would be re-rendered.

ISSUES WITH SERVER-SENT EVENTS

We run our software in the cloud on Amazon Web Services and that makes using SSE not straightforward. The UI needs a long running connection with some endpoint. Our API service provides it. But this service isn't exposed to the Internet. Requests come in via CloudFront. They are then directed to a load balancer, which then contacts the API service. You have no control over the timeout behaviour of these services. In fact, at any moment a connection interrupt may occur. This problem is not specific to the cloud. Many on premises set-ups have a similar structure.

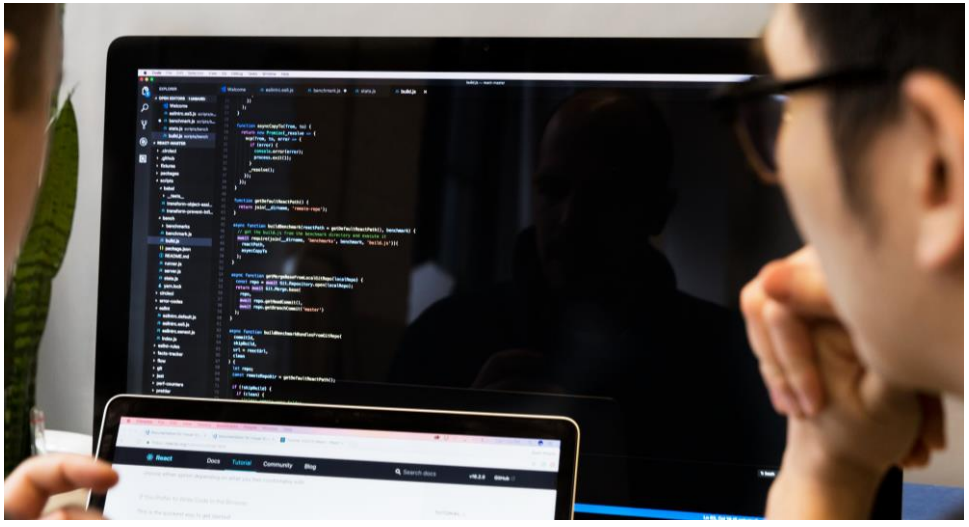
Providing many concurrent long running connections is a technical challenge. A standard Linux box could probably handle the scale of our applications well. But for very large scale operations this is a very specialized topic. The blog by Michai Rotaru shows that it is not trivial.

USING FANOUT.IO

Because of those issues we looked for a specialized player in the market. We ended up with fanout.io rather quickly. With this cloud service we could create a set-up in no time that works in all situations. You only need to provide two endpoints. First, you have the general SSE endpoint in your API service. You don't provide the SSE connection there. Instead, you redirect the browser to the fanout URL that comes with your account. In that account you specify your second endpoint, which should set up the fanout channel. Fanout will use it before completing the SSE connection with the browser. Any service in your system can now send an event to the client through the proper fanout channel.

We use the username to set up the channels. The SSE endpoint requires authentication. In the redirect to fanout we set the encrypted username as a URL parameter. Fanout adds this URL parameter when it calls the SSE set-up endpoint. In there we decrypt the username and create a fanout channel with the same name.

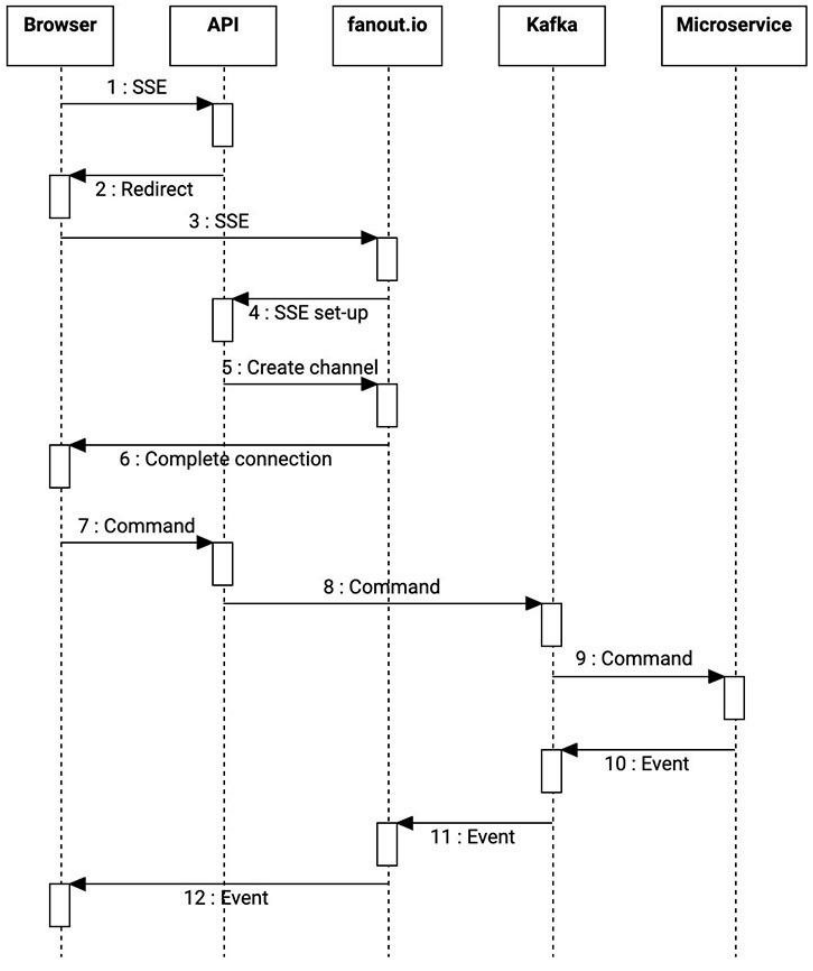
The commands and events carry the JSON Web Token of the original request. The standard "sub" field always contains the username. Our microservices communicate only through Kafka, which is a distributed publish/subscribe system. They receive commands via specific topics. They emit events via yet other specific ones. Certain Kafka topics have a fanout connector attached to them. So any microservice involved in the flow can notify the user. It simply places a message on the right Kafka topic.



THE COMPLETE FLOW

The whole thing is set up in the following steps, which are also shown in the sequence diagram below:

1. The browser connects to the SSE endpoint of the API.
2. The API redirects the browser.
3. The browser connects to the fanout SSE endpoint, which is part of your fanout account.
4. fanout contacts your SSE set-up endpoint. It adds the encrypted username from the redirect URL. This endpoint is also part of your fanout account.
5. The API creates a channel at fanout and gives it the name of the user.
6. fanout completes the connection from step 3.
7. The browser sends some command to the API.
8. The API relays the command to a Kafka command topic.
9. A microservice listens to that Kafka topic and executes the command.
10. This may have an effect on the aggregate instance. The microservice publishes the effect as an event on a Kafka reply topic.
11. The fanout Kafka connector sends the event to the proper channel using the username in the event.
12. fanout sends the event to the browser over the SSE connection.



SSE sequence diagram