# Stream-based, reactive programming and event systems

## The Problem

In a microservices architecture, there needs to be a common hub that centralises message processing in such a way that disparate systems are given the opportunity to respond to events of interest. A key factor of this sub-system is the programming model - keeping core business logic free of any boilerplate required to support the event system.

As Firstmac moves further and further away from its existing monolithic enterprise applications, the need for this common channel becomes more important. The microservices architecture works on a model of breaking apart core functions so that smaller pieces of an application can be scaled appropriately.

## Current Implementation

Some of the .NET implementations of services that we have, contain varying levels of "trigger" code. This trigger code is what exposes internal business functions to a database-configured trigger system, allowing a common observing library to execute code at particular points in the chain. An example of how this looks follows:

https://git.firstmac.com.au/apps/jobs/blob/bf298966ab54e6705592860f639d91c14d4597b3/Activities.Service/JobService.cs#L288

```
// process all of the "after complete" processes
await TriggerActionHelper.ProcessAllTasksAsync(job, TriggerActionPoints.BeforeReassign, messages, from);
// perform the update
await store.Collection.UpdateOneAsync(findDoc, updater);

Common.Log.WriteInfomation(id, "Job reassigned", from);

ThreadPool.QueueUserWorkItem(new WaitCallback(async f => {

// process all of the "after complete" processes
await TriggerActionHelper.ProcessAllTasksAsync(job, TriggerActionPoints.AfterReassign, messages, from);

}));
```

The developer needs to remember to actually call the **ProcessAllTasksAsync,** ensuring that the hard-wired parameters are correct. This can be error-prone.

The code must be entered in amongst the business logic which decreases code hygiene. The noise-to-code ratio moves away from a healthy level, because of the added boilerplate.

The application in charge of executing the business logic also needs to execute trigger invocation code. Any errors introduced by trigger code, now has an opportunity to push the business code over. This isn't ideal.

## Separation of concerns

Due to the nature of a microservices system, the measurement and observation of atomic pieces of functionality can be exploited in such a way that these now become **obersvable events**. This is achieved through a few linked pieces of architecture. The **restify** applications that we are producing use a library called **bunyan** to assist in the logging process. **restify** itself has a hook for auditing the response object as an action occurs, and as standard is hooked in all of our applications:

https://git.firstmac.com.au/apis/client-api/blob/ea228657dabe7559596f47eb55eea77750ddf7a1/server.js#L114

```
// setup the audit logger after each request
server.on('after', (req, res, route, error) => {

restifyPlugins.auditLogger({
log: logger,
event: 'after'
})(req, res, route, error);

  . . .
. . .

});
```

The audit logger plugin produces log items that look like the following:

```
{
"name":"reactor-api",
"hostname":"3ed7508c14c9",
"pid":1,
"audit":true,
"component":"after",
"level":30,
"remoteAddress":"::ffff:172.30.21.250",
"remotePort":41848,
"req_id":"7f8ac121-443e-4262-813c-9000bf111162",
"req": {
"query":{},
"method":"GET",
"url":"/ver",
"headers": {
"host":"api.firstmac.com.au",
"user-agent":"HTTP-Monitor/1.1",
"connection":"close"
},
"httpVersion":"1.0",
"trailers":{},
"version":"*",
"timers": {
"parseAccept":82,
"parseAuthorization":5,
"restifyCORSSimple":2,
"parseDate":1,
"parseQueryString":8,
"_jsonp":12,
"gzip":10,
"readBody":18,
"parseBody":1,
"handler-0":6,
"rateLimit":36,
"checkIfMatch":5,
"checkIfNoneMatch":1,
"checkIfModified":13,
"checkIfUnmodified":4,
"handler-1":171
}
},
"res": {
"statusCode":200,
"trailer":false,
"body": {
"name":"reactor-api",
"version":"0.2.3+build-20",
"description":"Trigger and web hook invocation endpoint"
}
},
"latency":0,
"_audit":true,
"event":"after",
"msg":"handled: 200",
"time":"2018-06-18T22:42:07.084Z",
"v":0
}
```


You can see that there are some interesting pieces of information on this audit packet that would allow us to define different events. We can see the **method**, the **url** as well as the **statusCode** and the **name** of the application that the record has come from.

These audit packets are logged to the stdout stream, within the docker container that hosts them. The docker container itself is then configured to use **syslog** as its logging mechanism to get these logged packets out; these are then sent to a central logstash repository. See **Reactive** in the Logical Architecture article.

The **Logstash** application is then responsible for sending the log message to both our **ElasticStack** infrastructure which is visualised through http://kibana.firstmac.com.au/ as well as to the **Kinesis** stream called **content-stream**.

## Kinesis, Lambda, and Reactor

Once a log message is on its way down the **content-stream** we now have an opportunity to respond to it. A **lambda** function is created within AWS which is called **reactor**. The **reactor** function's responsibility is to observe all of the messages flowing through the **content-stream**. In **react or's** database we define different events (which are just criteria that we test against each audit record). When an event matches, we respond with a task. A task could be "send an email", "update a database table", etc ...

The administration interface for the **reactor** application is found in FAAC https://faac.firstmac.com.au/. Using the 

link, we're presented with a menu of **Events**, **Tasks**, and **Reactions**.

## Events

An event is simply and audit message filter. We use a javascript predicate to test an audit message; if it matches we call this an "Event". Here's in an example:

### Name

Version of client-api is requested

### Description

the version number is requested from client-api

Complete the sentence, "When . . .". e.g. "When **the report finishes"**

### Event Type

Javascript Predicate

### Predicate

```
x => x &&
     x.req &&
     x.name === 'client-api' &&
     x.req.method === 'GET' &&
     x.req.url === '/ver' &&
     x.req.headers &&
     x.req.headers['user-agent'] !== 'HTTP-Monitor/1.1' &&
     x.res.statusCode === 200
```

We are testing the audit message to be a **GET** request that hits the **/ver** endpoint of the **client-api** application. We also test that it was successful (a 200 status code) and we filter out traffic from the HTTP-Monitor application. Anytime reactor sees a packet matching this description, it'll treat is at the "Version of client-api is requested" event.

This now applies a concept on top of audit messages. We give special meaning to some audit messages by filtering them out with predicates. Messages that are caught by these filters, we now call "Events".

## Tasks

The job of the reactive stream is to not only allow applications to observe things as they occur, but also to be able to respond and react to them. The reactor system allows for a few different execution models; each of which make sense in different contexts.

A **Webhook** task bundles the event audit message in a POST request against a defined URL. This is the most common of any of the execution models as it allows programmers to bundle application-specific reaction code into the originating application; rather than trying to centralise it all into one massive application (like TraxTriggers).

A **Http** task is much like a **Webhook** task only the programmer is allowed to control more features of the HTTP pipeline.The Verb, Headers, and Body are all controllable in the defined Task.

A **Redirect** task takes the current audit message and simply puts it on another Kinesis stream. If your software solution has a number of Kinesis streams, all in-charge of their own workflow arm, this may be useful.

A **Lambda** task allows the programmer to invoke an AWS Lambda function from a task. When units of functionality don't belong in a API, don't make sense to access over the HTTP channel or multiple invocation channels are required for the same unit of functionality, hosting a function in AWS Lambda can quickly solve a lot of outstanding question.

In the following example, I have setup a task that invokes a Lambda function called "test-log".

# Task Details

Choose the type of task

○ Webhook ○ Http ○ Redirect ◉ Lambda

**Id**

5b1df021b3bd45000191dd5c

**Name**

Test log

**Description**

execute the test-log function

Complete the sentence, "Then . . ". e.g. "When the report finishes, then **send an email**"

**Function name**

test-log

**Object mapping**

```
{
  "req": "req",
  "res": "res"
}
```

A object-mapper compliant block of JSON that will transform the input context document into the object to be passed to this function.

Save

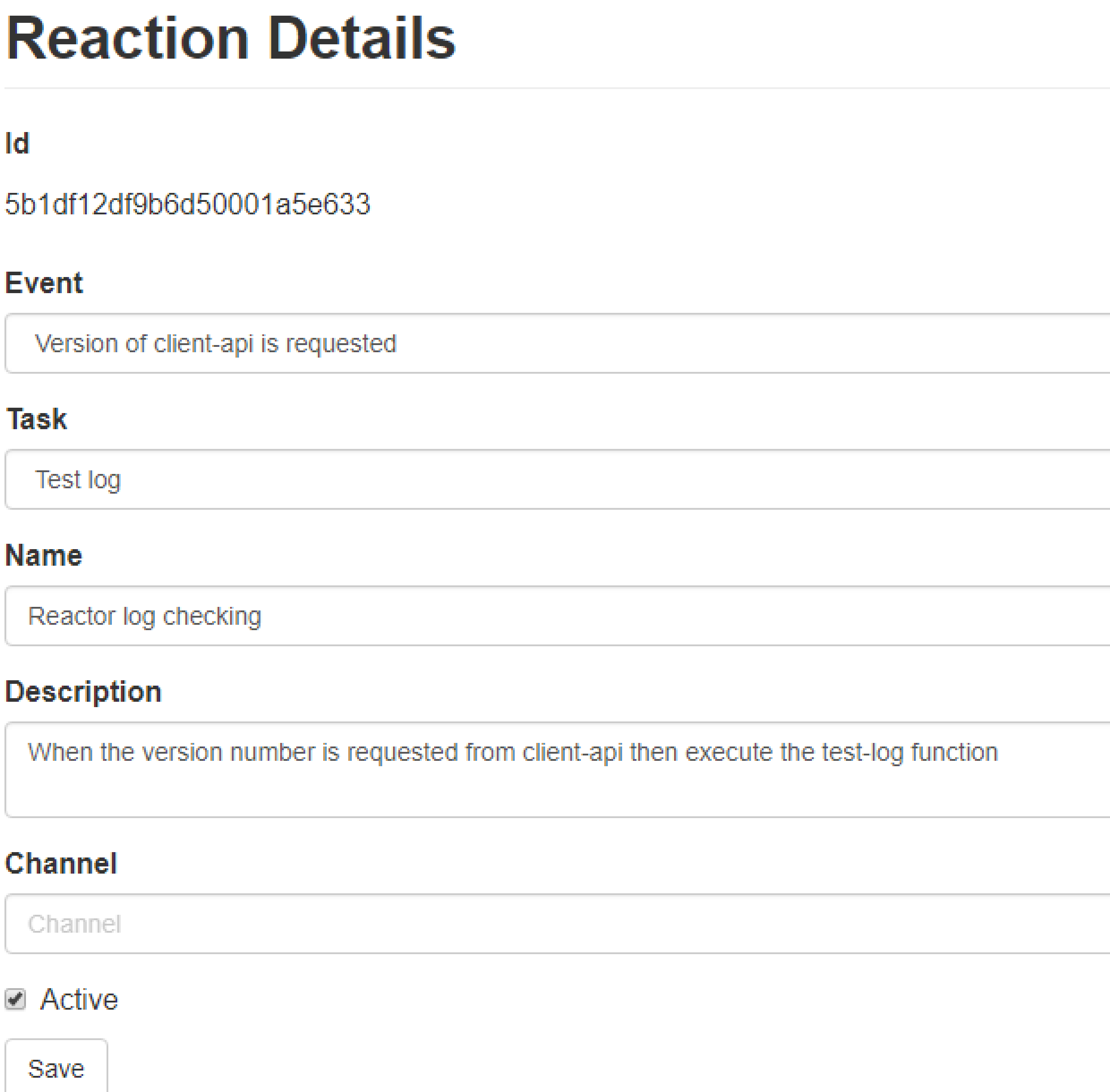
The resulting object that is supplied to the **context** of the Lambda function is dependent on the object mapping supplied. The actual code of this Lambda function in AWS looks as follows:

```
exports.handler = (event, context, callback) => {
console.log(context);
console.log(event);
return callback(null, 'Hello from Lambda');
};
```

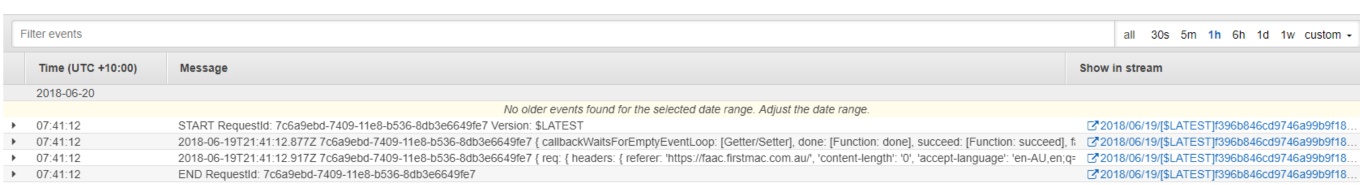
It's only very basic and acts as a proof of concept to the system.

## Reactions

When we want to start ***making tasks execute*** as a result of ***observing events in the system***, we'll use a ***reaction***. A reaction pairs an event with a task. In the following example, we'll take the "version requested" event for the client-api application and make the test-log lambda function execute, every time that it's seen:




An example of how this function looks when it's logging information out (in response to a client-api version retrieve) looks like this:



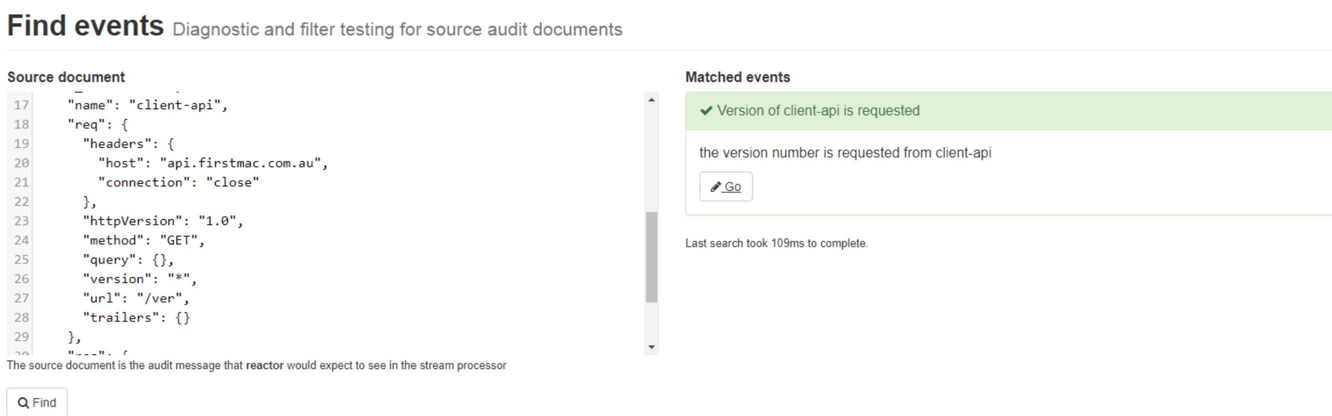

The function starts; it logs the context; it logs the event, the function finishes without error.

## Testing

Due to the distributed nature of this system, it becomes difficult to test end to end. Isolating pieces of the chain is quite simple though.

## Find Events

The "Find Events" function inside of the reactor admin allows you to specify a source audit message and run it against all of the filters supported. This user interface will then tell you what event filters have fired off. Here's an example of this user interface looking at a version request packet for client-api:




This utility helps us understand how well our predicates are working, and if other events are firing based on our packets.

## Invoke Task

The "Invoke Task" function inside of the reactor admin allows you to isolate the execution of your Task. You can give it any message that you'd like; see what the result of the execution was.


To make sure, you can also take a look in the CloudFront logs to see that your event has fired as you need.


# Key notes

- Don't litter your application code with side-effects; let the reactive stream handle it.
- Try to make your tasks as **composable** as possible; this will promote re-use.
- Design your workflow on paper (or visio) before trying to implement it; understand the audit messages that you want to respond to.
- Use the "Find Events" utility witihn reactor to understand if your particular event has already been defined.
- At nominal operating speeds, reactor will fire your tasks within two (2) seconds of observing it.
- IteratorAge is important for the reactor lambda as well as for the kinesis stream; when these fall back, so does the perceived time of the system.
- CloudWatch logs sometimes don't keep up in realtime, and backfill in.