




AsyncAPI Specs from Real-World API Data

Rob Galanakis
Cofounder, Principal, WebhookDB
 [@robgalanakis](https://twitter.com/robgalanakis)

Natalie Edson
Senior Engineer, WebhookDB



R: Hi, my name is Rob Galanakis,

N: my name is Natalie Edson,

R: we are the founder and the senior software engineer at WebhookDB. Today, we have something fun to present, well at least we hope it's fun, it's definitely a bit different than many of the other talks. We're going to build a complete and comprehensive AsyncAPI specification without writing a single line of it ourselves.

So let's set the stage. There are two things you should know. First, about our company. It's just us two right now. We'll talk about our product, WebhookDB, in a minute, but we also run a software consultancy, Lithic Technology. We're juggling clients, and customers, and support, and life. So there's always quite a lot going on.



N: The second thing you should know is that we're pretty lazy. "Lazy" of course meaning, we like to spend time doing things we enjoy, and avoid spending time doing things we don't enjoy. I am a published poet. Rob is the primary parent for his two kids. We also enjoy programming, thankfully, but when we program we like to really let our creativity flow. We are not "specification people."

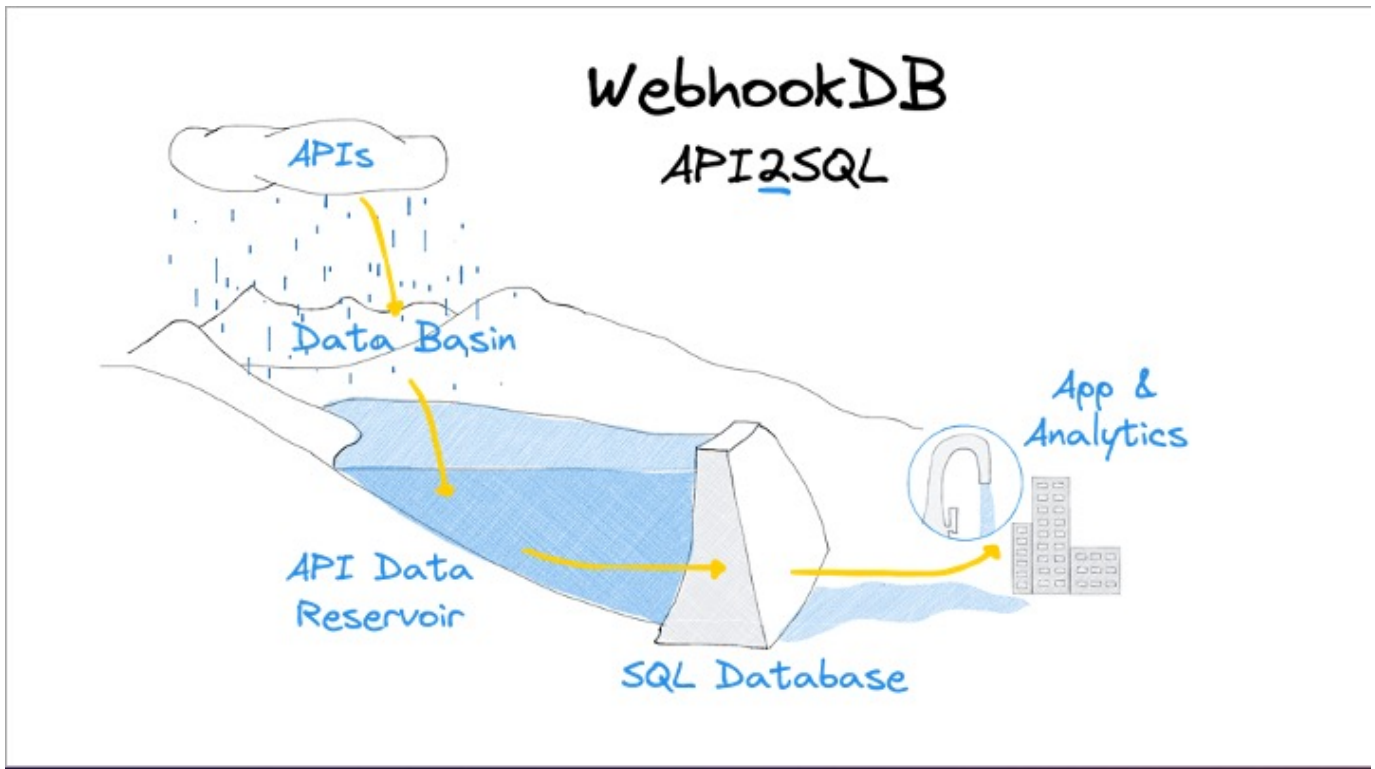
“

An application is any kind of computer program or a group of them. It **MUST** be a producer, a consumer or both. An application **MAY** be a microservice, IoT device (sensor), mainframe process, etc. An application **MAY** be written in any number of different programming languages as long as they support the selected protocol. An application **MUST** also use a protocol supported by the server in order to connect and exchange messages.

R: On top of that, I'll come out and say I'm not a huge fan of code generators. I'm also skeptical of any technical documentation artifacts I run across in version control. And my eyes run right over this sort of specification language that reads like someone's lecturing a misbehaving child.

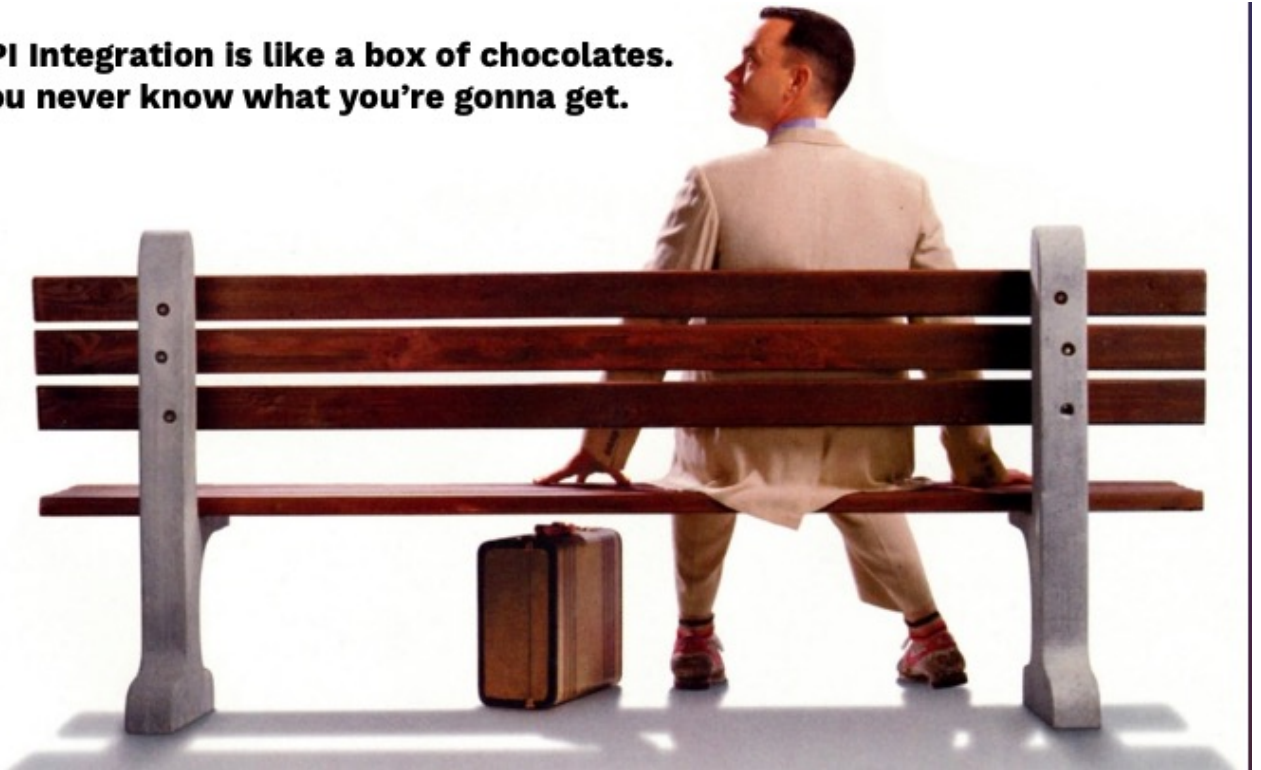
There's no question, though, that specifications are absolutely vital to the world of programming, and I'm glad there are people that enjoy building them. Just because I don't want to isn't a criticism of specifications as a pursuit.

But it does make us very strange speakers at a conference designed around a specification, then. I mean, how did we get involved with AsyncAPI in the first place?



R: To answer that, we need to tell you a little bit about WebhookDB, our software product. I will keep this short since this is not a sales pitch. WebhookDB is a self-hosted or fully managed data reservoir that instantly synchronizes, schematizes, and normalizes data from 3rd and 1st party APIs and can be queried using SQL. Or put another way, we take data from APIs, jam it into a Postgres database so it's normalized and structured, and then you query that database.

**API Integration is like a box of chocolates.
You never know what you're gonna get.**



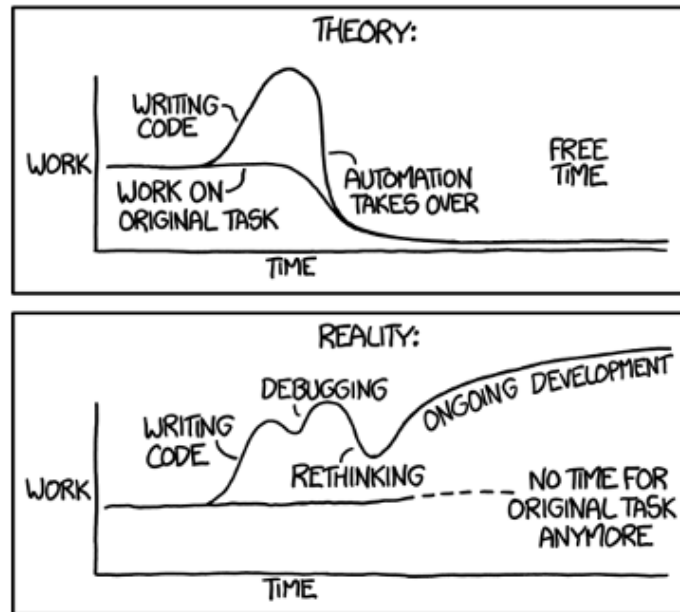
N: The biggest challenge of an integration product is the sheer, uh, variety of programmer expression, that exists. Every time we start a new integration, we have to be prepared for anything. No documentation, examples, or staging environments are ever complete. And many APIs lack them entirely!

This is a particular problem for WebhookDB, because we also support what we call the “API 99%,” which is the long tail of companies who offer APIs as an additional thing rather than their main product. Integrating your Stripes and Shopifys is not so difficult, but most APIs aren’t up to those standards.

```
{
  "currentUserHasAccessToDetails": true,
  "DeletedDate": null,
  "end_date": "06/10/2022 12:30",
  "start_date": "06/10/2022 11:30",
  "LastKeptAppointmentDateTimeMDY": "05/02/2022 10:00 AM",
  "Age": "15",
  "NextAppointmentDateTimeMDY": "-",
  "status": {
    "AbsenceReason": null,
    "Status": "Upcoming"
  },
  "statusBadge": "",
  "statusJson": "{\"Status\":\"Upcoming\",\"AbsenceReason\":null}",
  "teletherapySessionId": null,
  "teletherapyTitle": null,
  "type": 1
}
```

N: This for example is part of an actual payload from an API we support. There's a lot going on here, and it's trickier than it seems at first look. Like, you can't really tell what the 'start date' and 'end date' refer to in terms of AM and PM. You can only know for sure by seeing more data.

"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"



R: So, every time we have to integrate an API, we'd have to carefully go over documentation and examples, implement based on that, hope everything is correct, and, tell me if you've heard this one before, continually find out all the ways their production API differs from what's documented.

But, we're lazy, and a small company, I don't like fixing production issues at my kid's soccer practice, and, we like building silly but sometimes useful stuff.

So we stepped back and said, is there a better way? Yes there is.

Read docs to
integrate API.



R: Well, maybe there is.

What we realized is that, instead of integrating an API and *then* testing it live, we could capture a bunch of data from production, and use that to drive the integration.

From there, we realized it's actually really difficult to understand an API by looking at 1,000 different individual requests. We needed a way to classify them.

This is what led us to AsyncAPI. Instead of using a specification to drive and control the development of a service, we could use it to classify and describe a service that already exists. That is: we could feed individual requests into a program that generated an AsyncAPI spec, *even before* we started writing a real integration!

And then! We decided we could use the specification to create fixtured events, and publish them against WebhookDB, as a way to regression test every single one of our integrations.

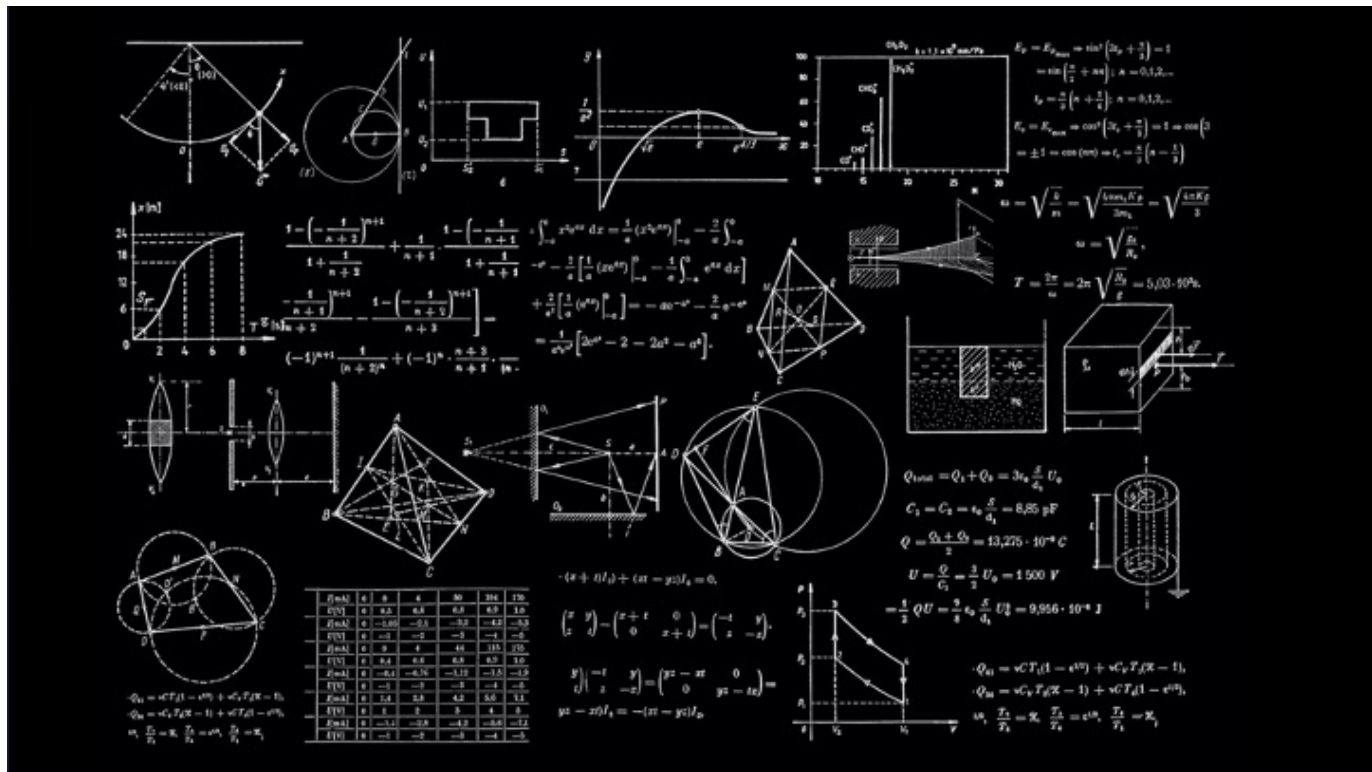


“

Instead of circumscribing development, specifications can be used to explore unfamiliar systems and navigate how we integrate with them.

R: It was a real “aha” moment that instead of using a spec to circumscribe development, we could use it to explore unfamiliar systems and navigate how we integrate with them. This could bridge the gap between the highly structured, procedural world of specifications, and the free-er flowing, more artisanal style of programming that we like to do.

So we set out to build a tool that could accomplish this for us.



N: To do this, we depend on cutting edge techniques across the disciplines of machine learning, genetic programming, multivariate statistics, and formal proof modeling.

```
if t == jsontype.T_STRING {
    s := value.(string)
    if sniffEmail(s) {
        return F_EMAIL
    } else if sniffUrl(s) {
        return F_URI
    } else if sniffIPv4(s) {
        return F_IPV4
    } else if sniffIPv6(s) {
        return F_IPV6
    } else if sniffCountry(s) {
        return F_COUNTRY
    } else if sniffCurrency(s) {
        return F_CURRENCY
    } else if sniffUuid4(s) {
        return F_UUID4
    } else if sniffNumericalString(s) {
        return F_NUMERICAL
    } else if sniffDateTimeNoTZ(s) {
        return F_DATETIME_NOTZ
    } else if sniffDateTimeTZ(s) {
```

N: Just kidding, it's just a lot of if statements.

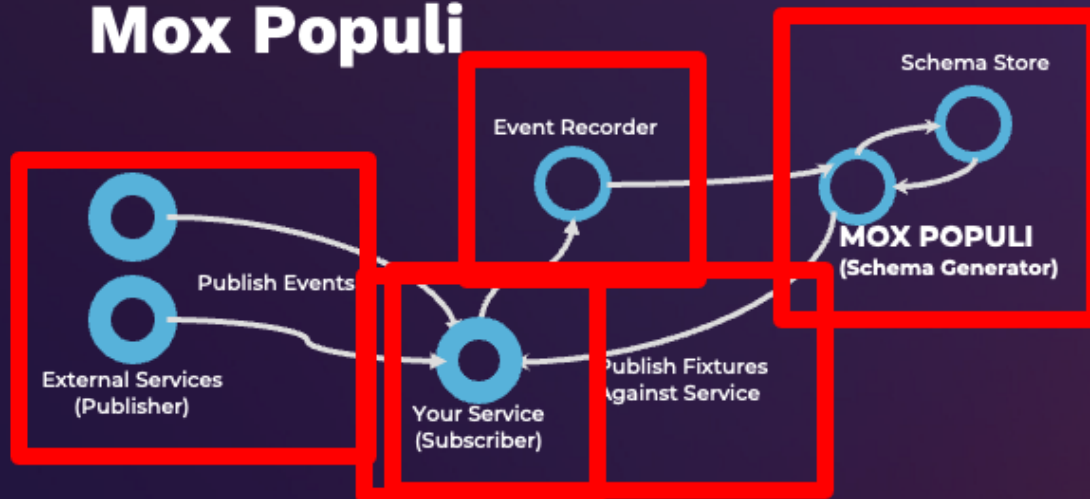
SPECIFICATIONS FOR THE PEOPLE

MOX POPULI

N: The tool we built is called “Mox Populi,” or “Mox Pop” for short. When we think about authoring specifications, there is normally a central group that makes decisions, and then it is up to other devs to ensure their services conform. Mox Pop flips that on its head. The implementation is what drives the specification. Mox Pop generates specs for how your service integrates with an external service. This perspective has been important for us while working on WebhookDB, because we’re working primarily with 3rd party APIs that don’t have formal specs, and cannot be modified.

SPECIFICATIONS FOR THE PEOPLE

Mox Populi



N: Here's how this all works conceptually. Events are published by External Services, on the left, to your service, in the center.

When you use Mox Populi to build a spec for your service, these events end up describing subscription messages.

As events from the external services are handled by your service, you sample and record them at full fidelity.

Those recorded events get fed into Mox Pop, which generates AsyncAPI specifications, and then incrementally improves them, using newly recorded events.

Finally, you can use those specs to generate fixtures that you can publish back to your service as part of integration tests, or to generate data for unit or fuzz testing.

SPECIFICATIONS FOR THE PEOPLE

Mox Populi

- 1. Recording Events
- 2. Generating JSONSchema for Message Payloads
- 3. Generating the rest of AsyncAPI Specification
- 4. Fixturing to test your integration

R: That was a lot, let's break it down into four steps.
The first step is recording events,
then we use the recorded events to generate JSONSchema for message
payloads,
we then use the recorded events to generate the rest of the AsyncAPI spec.
And finally, we turn generated specifications into fixtured events and publish
them to our service.

STEP ONE

Recording Events

Sample and record full-fidelity event data into a database.

Capture enough information so you can replay the event.

R: The first step is the most straightforward. Capture full-fidelity events in whatever format they are being published.

For example, if these are HTTP events, you will need to capture method, url, headers, and body, even if you only think you care about the body. If you've captured enough information so that you can replay the event, you did it right.

Also, you'll probably want to sample the events you capture. There's no reason you need to re-run Mox Pop for each event.

STEP TWO

Generating Payload Schemas

N: Step two. Generate JSONSchema for message payloads.

This is where Mox Pop enters the picture, and things start to get interesting.

STEP TWO

Generating Payload Schemas

```
{
  "x": 0,
  "z": 1664818615
}
```

```
{
  "x": {
    "type": "integer"
  },
  "z": {
    "type": "integer"
  }
}
```

N: In this step, we're not actually focused on AsyncAPI, we're just focused on JSONSchema.

We take a JSON object, and generate the JSONSchema that describes it. Now, *without* using Mox Pop, here is the JSONSchema we would expect.

This schema is accurate, but not very precise.

Precision vs. Accuracy



`{type: string}`



`{minLength: 10,
maxLength: 11}`



`{format: date}`



`{???`

N: To generate schemas that we can use to generate fixtures, we need to be both precise, **and** accurate. These targets, from left to right, show low precision and low accuracy, like a JSONSchema with just a type. It doesn't provide much detail about anything.

high precision but low accuracy, like if we generate constraints that are wrong.

low precision but high accuracy, like if we use the 'format' property. Knowing a string is in a "date" format is useful, but it doesn't tell us anything about the actual date being represented.

and finally, high precision and high accuracy, whatever that means. We need to achieve high precision and high accuracy. That's what Mox Pop's schema generation is all about.

STEP TWO - GENERATING SCHEMA PAYLOADS

High Precision, Low Accuracy

```
{
  "x": 0,
  "z": 1664818615
}
```

```
// moxpopuli schemagen -p=_ -pa='{"x":0,"z":1664818615}'
{
  "properties": {
    "x": {
      "format": "int32",
      "type": "integer",
      "x-seenMaximum": 0,
      "x-seenMinimum": 0
    },
    "z": {
      "format": "timestamp",
      "type": "integer",
      "x-seenMaximum": 1664818615,
      "x-seenMinimum": 1664818615
    }
  },
  "type": "object",
  "x-samples": 1
}
```

R: This is where we start to get creative. We can use AsyncAPI “extension properties” to provide a new layer of precision to the data we’ve seen. Using Mox Pop, this is the JSONSchema we would end up with for the previous example.

You can see here that we have an extremely high level of precision. But, we’ve made a lot of assumptions. These assumptions may drive down accuracy. We assume x will always be a 32 bit integer, and that z will be an integer and a timestamp. We can’t know these things for sure, though. It’s very likely we’re wrong about one or all of the assumptions we’ve made.

STEP TWO - GENERATING SCHEMA PAYLOADS

Increasing Accuracy

Commutative: changing the order of operands does not change the result.

Schema is accurate for all data every processed.

```
// noxpopuli schemegen -pfile:///testdata/simplepayloads.jsonl
{
  "properties": {
    "x": {
      "format": "float",
      "type": "number",
      "x-samples": 4,
      "x-seenMaximum": 12433403232320.5,
      "x-seenMinimum": 0
    },
    "a": {
      "oneOf": [
        {
          "format": "timestamp",
          "type": "integer",
          "x-samples": 3,
          "x-seenMaximum": 1664010617,
          "x-seenMinimum": 1664010615
        },
        {
          "type": "string",
          "x-samples": 1,
          "x-seenMaxLength": 1,
          "x-seenMinLength": 1
        }
      ]
    }
  }
}
```

R: So, in order to improve accuracy, we can generate schemas for many payloads and merge them together. The resulting schema will have a decreased precision, but increased accuracy. For example, what we thought was an integer may actually be a floating point number, or could be a string in some cases.

We keep “fitting” the schema to the data we see. This fitting process is commutative, so we can keep doing this incrementally, as we record new events. The resulting schema will always be accurate for all events ever processed.

STEP TWO - GENERATING SCHEMA PAYLOADS

Regaining Precision

```
// moxpopuli schemagen -p=_ -pa="{\"x\": \"abc123\"}"
{
  "properties": {
    "x": {
      "type": "string",
      "x-seenMaxLength": 6,
      "x-seenMinLength": 6,
      "x-seenStrings": ["abc123"]
    }
  }, "type": "object", "x-samples": 1}
```

```
// moxpopuli schemagen -p=file:///testdata/ids.jsonl
{
  "properties": {
    "x": {
      "type": "string",
      "x-identifier": true,
      "x-samples": 6,
      "x-seenMaxLength": 28,
      "x-seenMinLength": 28,
      "x-sensitive": true
    }
  }, "type": "object", "x-samples": 6}
```

N: However there are a few cases where we can't be precise at the beginning. For example, if we get a random string of ASCII letters and numbers, we can't tell what it is right away. An identifier? A freeform text entry?

In these cases, Mox Pop will store and accumulate the values we observe in each payload, as we see on the left.

Eventually, after we've sampled enough data, we can make a decision about the schema. For example, if every value we've seen has the same length, and all seem like identifiers, we can assume this is an id field.

STEP TWO - GENERATING SCHEMA PAYLOADS

Regaining Precision

```
// moxpopuli schemagen -p=file:///testdata/enums.jsonl
{
  "properties": {
    "x": {
      "enum": [
        "VALUE1",
        "VALUE2",
        "VALUE3",
        "VALUE4"
      ],
      "type": "string",
      "x-samples": 34,
      "x-seenMaxLength": 6,
      "x-seenMinLength": 6
    }
  }, "type": "object", "x-samples": 34}
```

N: We do the same thing for enums. At some point, we've seen enough samples, and seen few-enough distinct, enum-like values, that we can assume the field is an enum. These enum values get put into the 'enum' JSONSchema property.

STEP TWO - GENERATING SCHEMA PAYLOADS

Sensitive Information

```
// moxpopuli schemagen -p=_ -pa='{ "x": "94flkfw03qpf89" }'  
{  
  "properties": {  
    "x": {  
      "type": "string",  
      "x-seenMaxLength": 14,  
      "x-seenMinLength": 14,  
      "x-seenStrings": [  
        "LlLWDYu9TqVohj"  
      ],  
      "x-sensitive": true  
    }  
  },  
  "type": "object",  
  "x-samples": 1  
}
```

R: Recording information in the x-seenValues property is tricky in that there may be sensitive values we don't want to include. Sometimes access tokens or IDs come through in an event. Whenever we get a string value that looks like gibberish, we generate a new string of the same size, and use that, so that we never write out IDs, tokens, or anything else sensitive.

STEP TWO - GENERATING SCHEMA PAYLOADS

Recording Examples

```

// printf '{"id":"3bfa"}\n{"id":5}' | moxpopuli schemagen -p=- --examples=1
{
  "examples": [
    {"id": "3bfa"},
    {"id": 5}],
  "properties": {
    "id": {
      "oneOf": [
        {
          "type": "string",
          "x-seenMaxLength": 4,
          "x-seenMinLength": 4,
          "x-seenStrings": ["3bfa"]
        },
        {

```

N: One last thing Mox Pop does is record copies of payloads into the 'examples' field. This allows you to see actual event payloads in the spec. We try to be clever about when we collect them, like when a property format changes. Being selective about examples bridges the gap between reviewing an API in the abstract *only from the specification*, and reviewing it *only through live events*. You get the best of both worlds- the API is condensed into a spec, and every example shows you some unexpected change vector.

STEP TWO - GENERATING SCHEMA PAYLOADS

Accurate and Precise

```
// moxpopuli fixturegen --lines --count=10 | moxpopuli schemagen -p-
{
  "properties": {
    "date-time": {
      "format": "date-time",
      "type": "string",
      "x-seenMaximum": "2032-02-01T10:04:39-08:00",
      "x-seenMinimum": "2018-02-23T21:20:42-08:00"
    },
    "date-time-notz": {
      "format": "date-time-notz",
      "type": "string",
      "x-seenMaximum": "2032-05-18T21:21:30",
      "x-seenMinimum": "2013-06-25T18:49:31"
    },
    "duration": {
      "format": "duration",
      "type": "string",
      "x-seenMaximum": "P4Y9M20DT17H13M45S",
      "x-seenMinimum": "P2M12DT16H7M24S"
    },
    "iso-country": {
      "format": "iso-country",
      "type": "string"
    }
  }
}
```

```
"numerical": {
  "format": "numerical",
  "type": "string",
  "x-seenMaximum": "742129038987",
  "x-seenMinimum": "-935726130713"
},
"timestamp": {
  "format": "timestamp",
  "type": "integer",
  "x-seenMaximum": 1925840647,
  "x-seenMinimum": 310039324
},
"timestamp-ms": {
  "format": "timestamp-ms",
  "type": "integer",
  "x-seenMaximum": 1847512946466,
  "x-seenMinimum": 93315572188
},
"uuid": {
  "format": "uuid",
  "maxLength": 36,
  "minLength": 36,
  "type": "string",
  "x-sensitive": true
},
"zero-one": {
  "enum": [0, 1],
  "format": "zero-one",
  "type": "integer"
}
```

R: Putting it all together, here are some of the formats and properties Mox Pop handles. You can see full examples in the [GitHub repo](#). The version here uses a superset of AsyncAPI and JSONSchema formats, plus some formats describing data we've seen in the wild, like numerical strings and integer timestamps. These additional formats, coupled with the constraints and the seen values, give us an accurate and precise description of the data we see from an API.

STEP THREE

Generating an AsyncAPI Spec

N: So the previous step was all about generating JSONSchema for payloads. We're gonna take the same idea and apply it to the rest of the AsyncAPI specification.

STEP THREE - REST OF THE SPEC

What can we generate?

- | Servers
- | Channels and Channel Items
- | Subscribe Operations
- | Messages

N: We can actually generate most of the spec from the full-fidelity events we record. We can't handle sections like "info" that require out-of-band information, but we can do quite a lot.



N: I'll warn you here that the examples in this section are HTTP-specific. Because WebhookDB is focused on webhooks, most of our usage is HTTP-based. All of these examples should work for other protocols, but we wanted warn you as we'll be using HTTP-specific terms ahead.

STEP THREE - REST OF THE SPEC

Anatomy of an HTTP Event

- Path ⇒ `channels[request path]`
- Method ⇒ `channel.bindings.http.method`
- Body ⇒ `message.payload`
- Headers ⇒ Special, Protocol, Application

N: We can take each component of the HTTP event and figure out what parts of the spec the component applies to. Path, method, and body map directly to channels, channel bindings, and message payload, but request headers are more interesting.

The HTTP request headers map to multiple parts of the spec. We categorize headers into Special, Protocol, and Application headers.

STEP THREE - REST OF THE SPEC

Special Headers

- "Host", "Accept" ⇒ `api.webhookdb.com`, `http 1.1`
- "Content-Type" ⇒ `message.contentType`
- "Request-Id" ⇒ `message.correlationId`

R: Special headers are those with a special purpose. For example, Host and Accept can be used in the 'server' object, describing the server URL and protocol version. The content type header sets the message content type. And we can figure out the message correlation ID by guessing which header, if any, looks like it's the correlation id.

STEP THREE - REST OF THE SPEC

Protocol Headers

```
{
  "message": {
    "bindings": {
      "http": {
        "Accept-Encoding": {
          "type": "string",
          "x-lastValue": "gzip;q=1.0,deflate;q=0.6,identity;q=0.3"
        },
        "Host": {
          "type": "string",
          "x-lastValue": "localhost:18001"
        },
        "User-Agent": {
          "type": "string",
          "x-lastValue": "WebhookDB/v1 webhookdb.com 2022-10-01T00:00:00Z"
        },
        "Version": {
          "type": "string",
          "x-lastValue": "HTTP/1.1"
        }
      }
    }
  }
}
```

R: For protocol headers, we keep a list based on HTTP standards. Any protocol headers are set on the message http binding headers, as in the example. We include the actual header value in the JSONSchema so the spec is easier to understand. Protocol headers, as you may guess, can't easily be fixtured, since the value is meaningful for the protocol. So in these cases, we end up leaving them out or use the x-lastValue verbatim.

STEP THREE - REST OF THE SPEC

Application Headers

```
{
  "channels": {
    "/v1/service_integrations/svi_81f5em7skqagk7pstse7b4j1r": {
      "subscribe": {
        "message": {
          "headers": {
            "properties": {
              "Whdb-Secret": {
                "type": "string",
                "x-identifier": true,
                "x-samples": 7,
                "x-seenMaxLength": 25,
                "x-seenMinLength": 25,
                "x-sensitive": true
              }
            }
          }
        }
      }
    }
  }
}
```

N: Finally, anything we don't identify as a protocol or special header, we treat as an application header. For application headers, we use standard Mox Pop schema deriving and fitting, so they get as much fidelity as the message payloads.

STEP THREE - REST OF THE SPEC

Example Spec

```
{
  "asyncapi": "2.4.0",
  "info": {
    "contact": {
      "email": "hello@webhookdb.com",
      "name": "Hello"
    },
    "description": "These are the WebhookDB endpoints av",
    "termsOfService": "https://webhookdb.com/terms/",
    "title": "WebhookDB Integrations for Demo Org",
    "version": "1.0.0"
  },
  "servers": {
    "localhost:18001": {
      "protocol": "http",
      "protocolVersion": "1.1",
      "url": "localhost:18001"
    }
  },
  "channels": {
    "/v1/service_integrations/svi_81f5em7skgagk7pstse7b4": {
      "subscribe": {
        "bindings": {
          "http": {
            "method": "POST",
            "type": "request"
          }
        },
        "message": {
          "bindings": {
            "http": {
              "headers": {
                "properties": {
                  "Accept": {
                    "type": "string",
                    "x-lastValue": "**/*"
                  }
                },
                "type": "object"
              }
            }
          }
        }
      }
    }
  }
}
```

N: When you put this all together, we can take some recorded events and generate a totally usable and valid AsyncAPI specification. You can see the full example from the GitHub repo, which is linked to at the end of the talk.

STEP FOUR

Mox Populi Fixturing

R: Alright, we're finally at the reason we built Mox Populi in the first place. Which is, to use the generated AsyncAPI specs to fixture and publish a huge number of events against a service.

STEP FOUR - FIXTURING

Generating From JSONSchema

```
// moxpopuli datagen --l=file:///testdata/fixturedemo.schema.json
{
  "SessoainIP": "30.34.254.115",
  "array-of-ids": [
    "14f35dd7-ddbd-5238-dc14-1fb7ec03b5bc",
    "b4fa9ab3-4c59-629c-9019-f87272617e90"
  ],
  "arrayofobjects": [{"myid": "7f968714-d8af-55fd-9e44-c9e710bb7c8f"}],
  "base64bytes": "d3c06f1859fd1a9e82d5c8",
  "currency": "XBB",
  "databaseid": "937",
  "email": "xiMWdMU@luaDPwc.com",
  "ended_at": "2024-08-08T02:00:00-07:00",
  "homepage": "https://PQdXgBt.info/xuZlXoA",
  "started_on": "2021-09-30"
}
```

R: There are two types of fixturing going on. Here's the first one. Mox Pop can create fixtured objects for the JSONSchema we saw it generate previously. It uses all the extension properties to make sure the fixtured value looks pretty close to real. There's obviously a lot of fake data here, but from a technical standpoint, this looks legit, and probably exercises the same code and database paths that real data would.

The detail in the fixtures comes mostly from the number of custom formats we support, which we saw earlier, but also, the extension property fields act as constraints that narrow down the fixtured values.

For example, if you have a "created" field that is an "integer timestamp", all of the fixtured values will be between the minimum and maximum that has been seen in any recorded event. You won't get 'created' values from 20 years ago or 20 years in the future.

STEP FOUR - FIXTURING

From an AsyncAPI Spec

```
// moxpopuli vox -l=file:///testdata/vhdbspec.json --count=100 --print

REQUEST /v1/service_integrations/avi_ct14kxb4ngg3auyrysjwzjlk5-0
POST /v1/service_integrations/avi_ct14kxb4ngg3auyrysjwzjlk5 HTTP/1.1
Host: localhost:18001
User-Agent: my awesome app
Content-Length: 284
Accept: */*
Accept-Encoding: gzip
Connection: close
Content-Type: application/json
Trace-Id: d274eb7f-d4a2-elf8-17f7-cdd7c97587f7
Version: HTTP/1.1
Whdb-Secret: c13cc9e79e6fb8347c7aba14

{"created_at": "2022-09-14T20:14:39-07:00", "email": "oOQtOKV8YsW8HXP.ru", "id": 20, "name": "55e3a8498bc5f8d6e961", "note": "", "opaque_id": ""}

RESPONSE /v1/service_integrations/avi_ct14kxb4ngg3auyrysjwzjlk5-0
HTTP/1.0 200 OK
Connection: close
Content-Type: application/json
Vary: Origin

{"o": "k"}
```

N: Finally, here is what happens when you run a regression test using Mox Pop “vox” and your service’s AsyncAPI spec. It publishes the configured number of events against the server. We use this fixturing to both load test and integration test our backend.

What you see here is the request and response dump of a fixtured HTTP request sent to the server, and the actual server response. Again, you can see we’re getting pretty nice fixtured data, and the server accepted it happily.

CONCLUSION


Takeaways

N: So what have we learned building Mox Populi? Three big lessons.



**AsyncAPI can describe any
kind of API.**

R: The first is that AsyncAPI truly can describe any kind of API. This was our first time using it and we were super impressed. I've done pretty extensive work with OpenAPI; AsyncAPI has some subtle and significant improvements, and I think all of the are great. I know I'm preaching to the choir but I did want to mention it, and affirm the amazing work so many folks here have done. As outsiders to the world of specifications, this work gets a huge thumbs up.

A close-up photograph of a bulldog lying down, its head resting on a dark, textured surface. The dog's eyes are closed, and it appears to be asleep. The lighting is soft, highlighting the dog's features.

**Don't build it yourself if you
want to be lazy.**

R: Our second takeaway is probably what you started thinking a couple minutes into this presentation. Building Mox Pop was a lot more work than *not* building it. While it's been useful, I wouldn't say it's been essential for us at WebhookDB, at least not yet. We expect this to change as it's developed further, though, and we get to lean on it for more and more integrations. We may have been able to solve some of our problems with other tools, or with a little manual labor.

However - and this is a big however - the third takeaway is



R: Whether it was worth it or not, building Mox Pop was a lot of *fun*. Software development at scale is frustrating and stressful. Two of the best ways we have found to make it more fun are to build better automated testing and tooling, and to work on tightly scoped projects that allow us to be creative within that scope.

In the case of Mox Pop, when your project is a tightly scoped testing tool, it's the best of all worlds. Mox Pop helped fill out integration testing around WebhookDB, *and* it was a ton of fun to work on. This is largely because AsyncAPI is so well done, from the spec, to the tooling, to the documentation, and the community, I can't say enough good things.

WEBHOOKDB & MOX POPULI

Thank You!



www.webhookdb.com



github.com/lithictech/moxpopuli



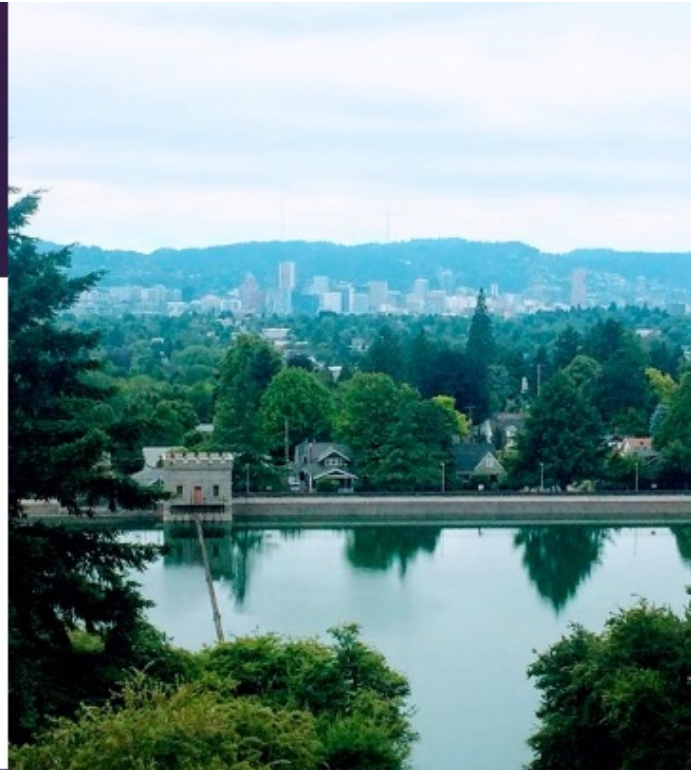
twitter.com/robgalanakis



rob@webhookdb.com
natalie@webhookdb.com



Portland, Oregon, USA



N: Thanks so much for watching us today! You can check out Mox Populi on GitHub. It's a Go project you can use as a CLI or library, but even better, there's a link in the README that allows you to try this whole thing out yourself using cURL. Really, you should go grab some events from your own services, run them through this endpoint, and see what sort of AsyncAPI spec you end up with. We hope it's as useful to your teams as it's been to ours.

You can also reach Rob and me via email if you want to follow up, we're happy to answer any questions or just talk shop.

Finally, if you are dealing with any 3rd party API integrations, you should check out WebhookDB.

And if you're ever in Portland, Oregon, please let us know, we'd love to buy you a coffee or tea and chat. Thanks again!