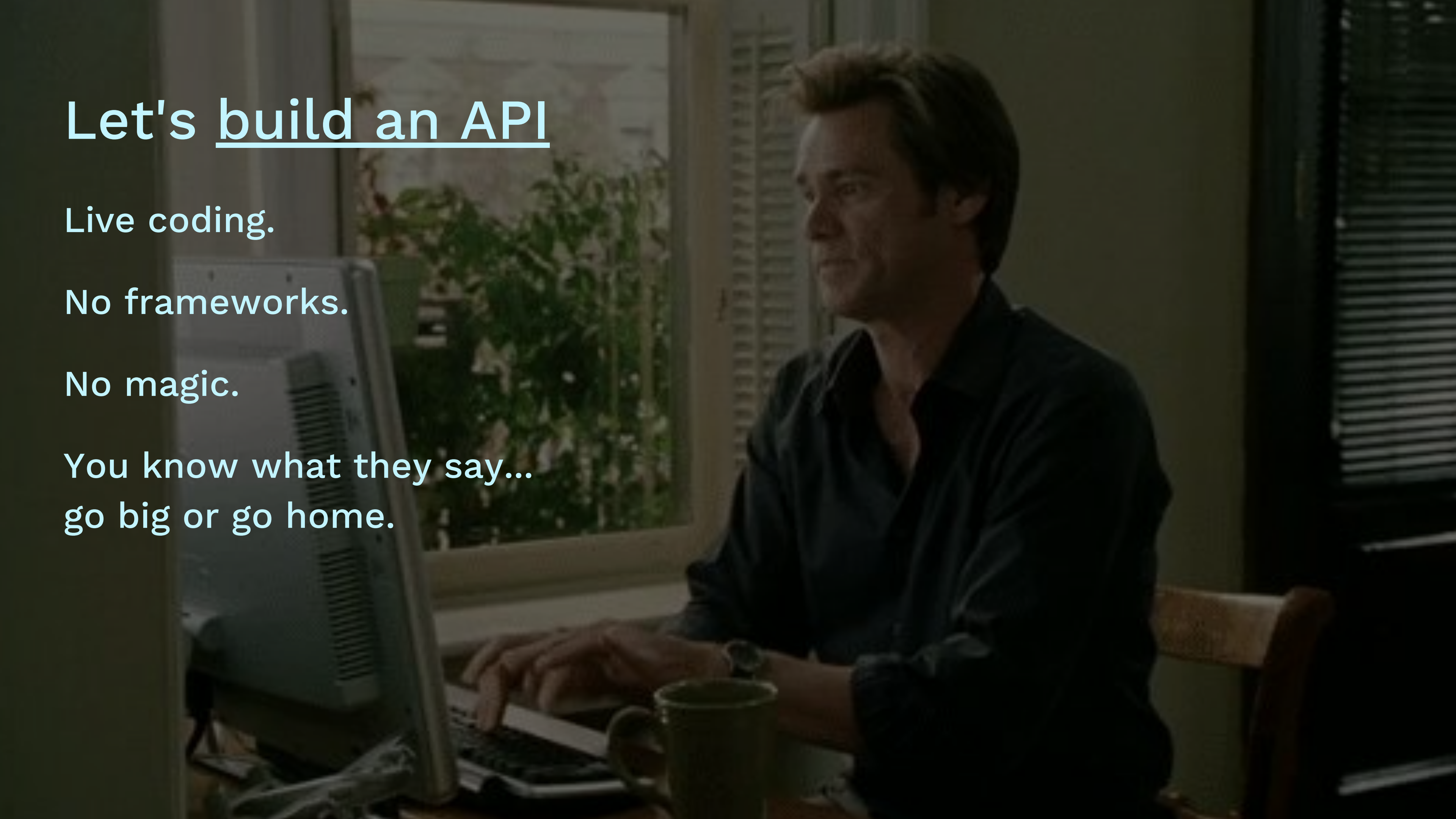
Brisbane APIs & Microservices

{ GET /bris/apis+microservices }

Alex Babkov   CTO@REX   @alxbabkov

Proudly sponsored by

APIs and making the unknown unknowns known



Let's build an API

Live coding.

No frameworks.

No magic.

You know what they say...
go big or go home.

We’ve built our API… <u>now what?</u>

Now what...

How do people know about it?
Do I need to publish it to a service or link from our website?

How do people know what they can do with it?
Maybe I need to document it?

People probably don't want to read my raw API specifications
I guess I should make them pretty...

I want people to be able to be able to add more details about books than just their names...

I want to make sure people provide me with a book name and author name...
I guess that means I need validation?

I probably don't want this API to be fully public...
Do I lock it down?

Are all my API users equal?

Jason, a developer has complained to us that his language doesn't support JSON decoding / encoding - he loves and wants xml (#JasonHatesJSON)
Do I need to support XML?

Now what... (cont.)

Hmm, my books are parts of libraries...
I guess i'll add a new endpoint

How do I search across books?

I need to request both books and libraries to build up my UI - I have to make two seperate requests. Two is a lot of requests David.

Why am I even building this API?

In Summary

- Developer portals and user provisioning
- API type (RPC, SOAP, REST, GraphQL)
- Documentation standards (GraphQL, Open API (formerly swagger), API Blueprint, Siren, WSDL, PDF/Word Doc)
- Introspection for certain aspects of documentation
- Authentication and authorisation
- Standards - do I use standards for things or not, which do I pick, when do I roll my own?

Summary (cont.)

- Caching and HTTP verbs
- File Uploads!
- Eventual consistency
- HTTP codes
- Response and error formats
- Transmission formats (xml, json, protobuff)
- Language support (JSON decode / stringify, response sizes - partial parsing etc.)
- Testing (Unit tests, Integration tests, Snapshot tests, Documentation tests)
- Versioning (Trust)
- Security, scalability (rate limiting, token security - how do you stop leaks esp in frontend land)
- Consolidated endpoints vs not consolidated endpoints. Business logic and complexity of relationships vs request protocol overhead (less an issue now with HTTP 2.0)

Summary (cont. 2)

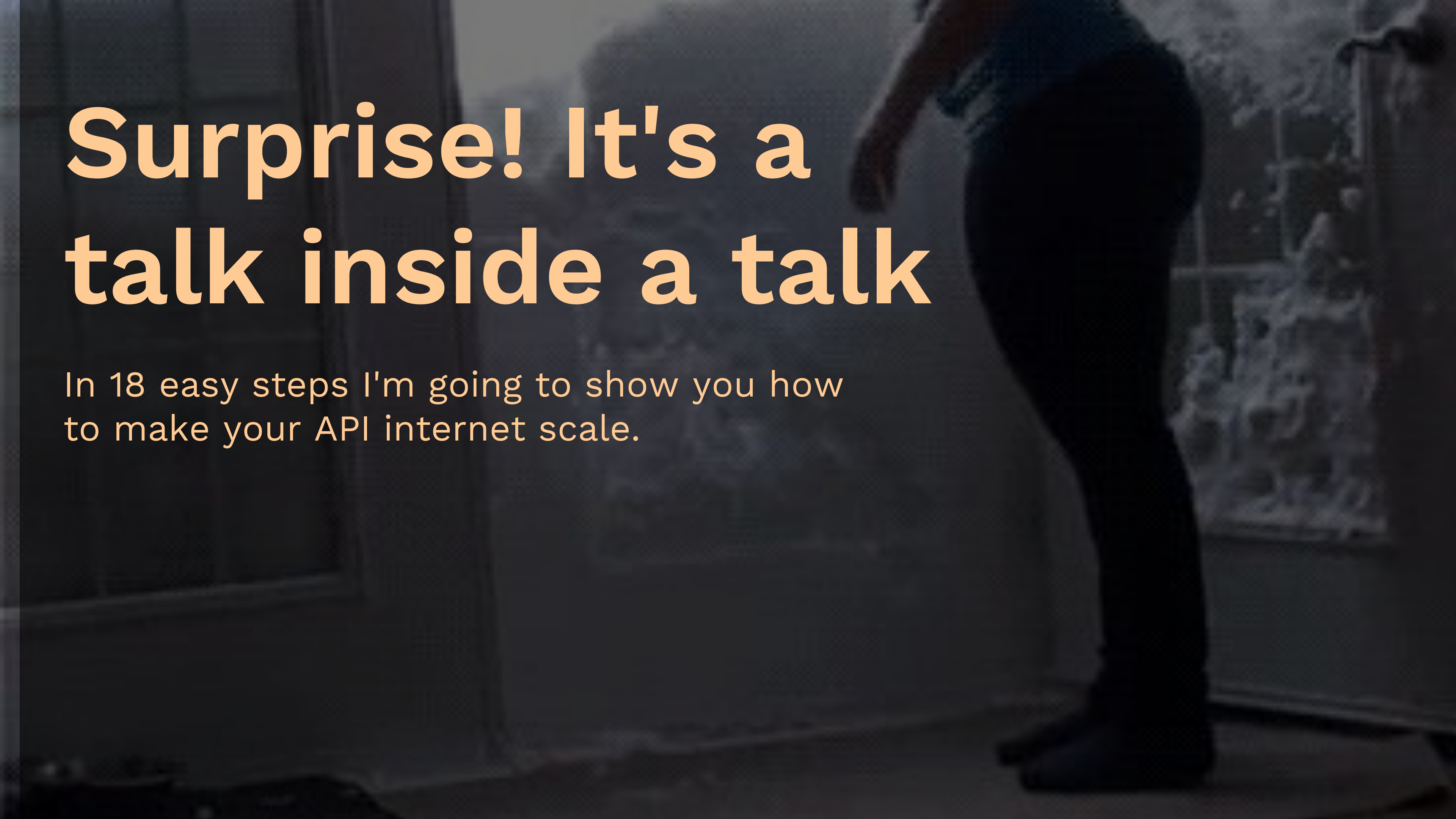
- Sparse resources - response versioning, Backend for frontend, customizable requests?
- Monitoring - so many things
- How do I deploy any of this
- How are microservices relevant?

Who are your stakeholders?

- Third Parties
- Management
- Internal desktop app developers (fe)
- Internal mobile app developers (fe)
- Internal backend developers (microservices)
- Developers writing the API

A talk inside a talk - Babushka Doll Rant:

My API is going to go viral, I want my application to be internet scale OR how to make your API internet scale in 18 steps.



Surprise! It's a talk inside a talk

In 18 easy steps I'm going to show you how to make your API internet scale.

Step 1

Stop letting people hit your API so much (rate limiting).

How should I rate limit - 60 requests per minute, 60 requests per minute

Step 2

Switch from storing your data in JSON files to using SQL lite.

SQL let's you query your data easier, SQL lite is good.

Step 3

Drop PHP and replace it with Node. You can use your frontend developers to write backend code.

Node will make your backend great again.

Step 4

Tell the user the resource hasn't changed (Cache headers)

Step 5

Drop node - node didn't make your backend great again. Node only has one package manager and Go has many package managers.

Go can also output hello world at least 2.783x FASTER than node in real world tests.

Step 6

You discover all languages are slow.
Add more servers.

Step 7

Ooops, SQL lite doesn't work across multiple severs.

Switch from SQLite to MySQL - MySQL is more scaleable than SQL lite.

Step 8

You discover any trip to your language and databases that can avoided is the best kind of trip. Cache requests at a CDN level.

Step 9

Your database requests are still slow. Upgrade your database server - more RAM solves everything.

Step 10

Your database reads are slow. Add more read replicas or shard it or something.

Problem solved.

Step 11

Your database writes are slow.
Throw more money at it, shard all
of things (maybe).

Step 12

Cache requests at web server level
- CDN cache expires sometimes.

Step 13

Decide that MySQL is owned by oracle and is not scaleable because oracle hides all the good code.

Switch to Maria DB - MariaDB is #MoarOpenSauce than MySQL

Step 14

Cache objects at application level to filesystem to prevent hitting database.

Step 15

Discover the file system is slow and starts choking on disk i/o. Come to the conclusion that caching is dumb and it's the databases fault.

Switch to postgres - postgres is better than MariaDB. MariaDB is #oldschool.

Step 14

Postgres is still slow, filesystem is slow.... Cache objects in Memcache or something then decide Redis is better.

Step 15

Switch to a document store like
AWS DynamoDB.
#SqlIsDeadLongLiveNoSQL

Step 16

Realise DynamoDB isn't fast enough. You need web 2.0 scale! Switch to our one true saviour mongo db, mongo db is internet scale

Step 17

Start piping all your requests to `/dev/null` after you realise that's how mongo achieves its speed (#shots-fired).

Joking, mongo is probably less bad than it used to be.

Step 18

Internet scale achieved.

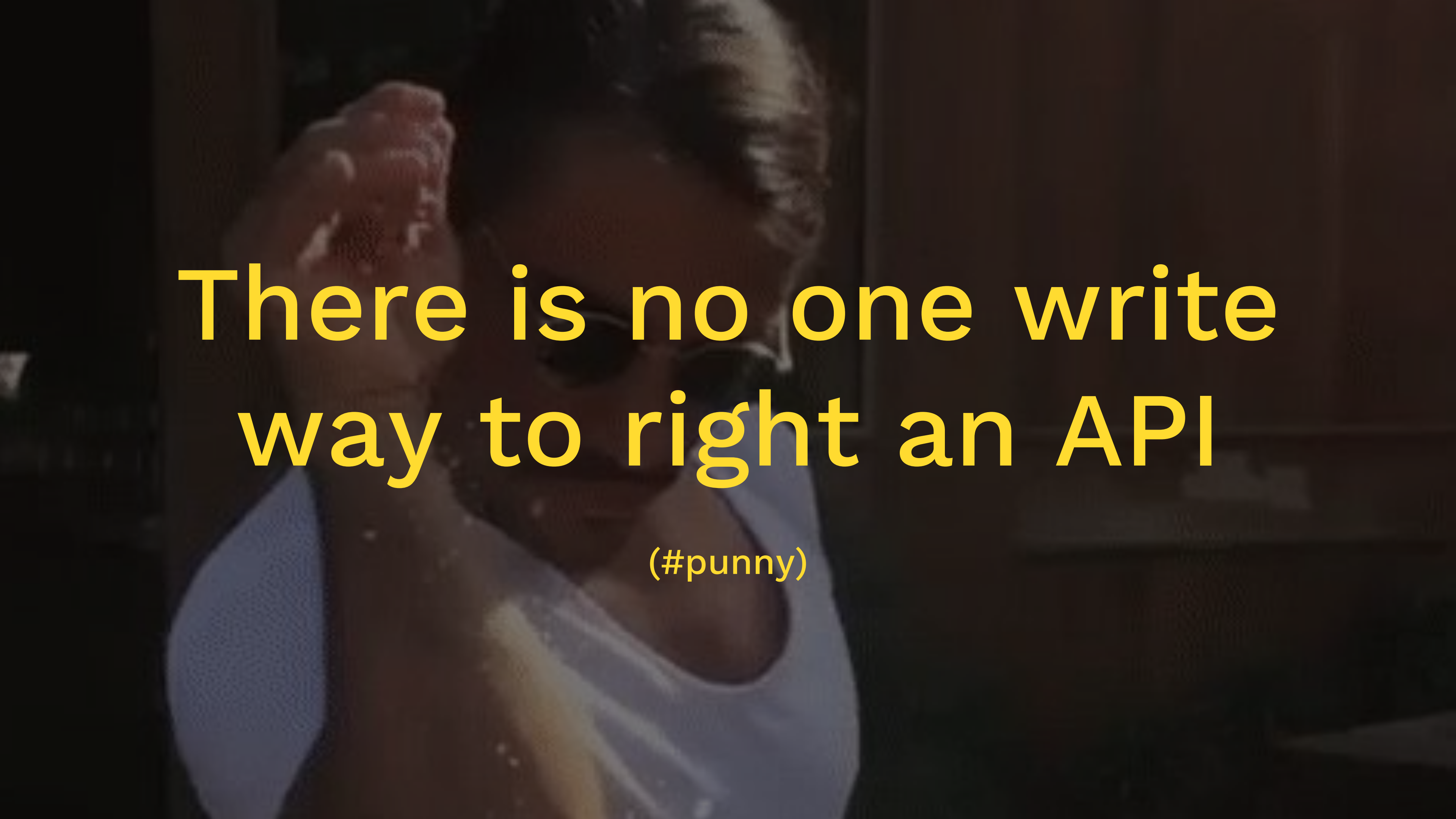
...obviously this is a joke - highlight cache steps only. You don't often have a choice of data store

With so many concerns, don't cargo cult

Something on the internet defines Cargo cult programming as "a style of computer programming characterized by the ritual inclusion of code or program structures that serve no real purpose".

Don’t cargo cult

- If you don't have the problems netflix has with it's 10,000 engineers, don't blindly use the solutions that netflix comes up with.
- If your organisation writes all of its APIs internally as badly documented SOAP api's, don't write your public facing API that you want millennials to use as a SOAP API.
- If your organisation has bad API practices, don't just keep rolling bad API practices.
- If an old or boring technology works for you, don't switch away from it just because someone has told you to or someone else is using something else.



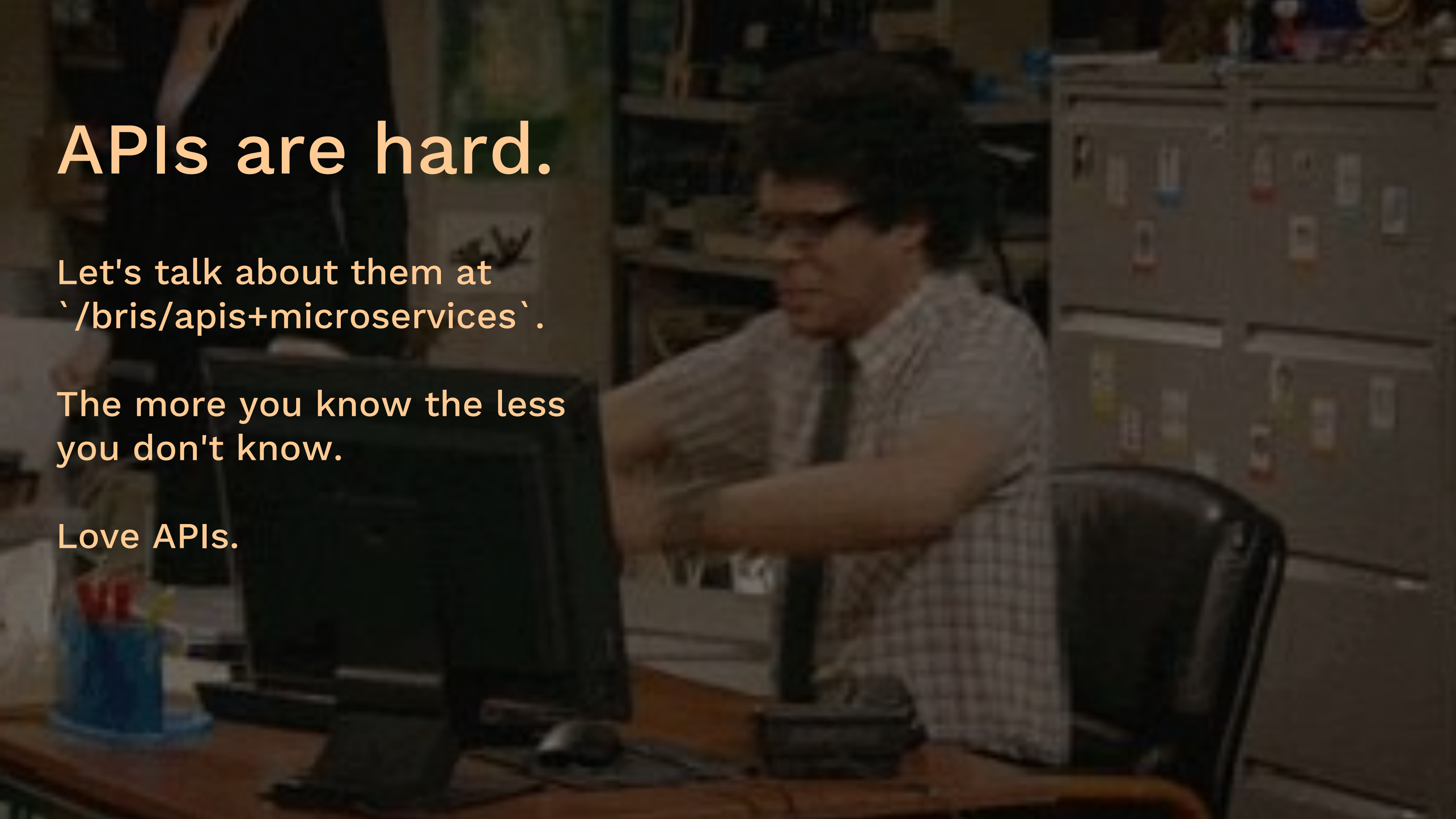
There is no one write way to right an API

(#punny)

A lot depends on your situation.

Answers should be different based on variables:

- What problem are you trying to solve
- Who are your stakeholders
- How big is your team
- How much can you rely on other people's experiences
- How much you trust what people on the internet say



APIs are hard.

Let's talk about them at `` `/bris/apis+microservices` ``.

The more you know the less you don't know.

Love APIs.