How I learned to develop software like a bureaucrat

...

Alex Bunardzic, May 28, 2024
Vancouver DevOps Meetup

Let’s start with silly meme

About the presenter

Started developing software professionally in 1990

Made all possible mistakes along the way

Worst mistake

What problem am I trying to solve here?

What do people complain about the most?

Pressure to deliver

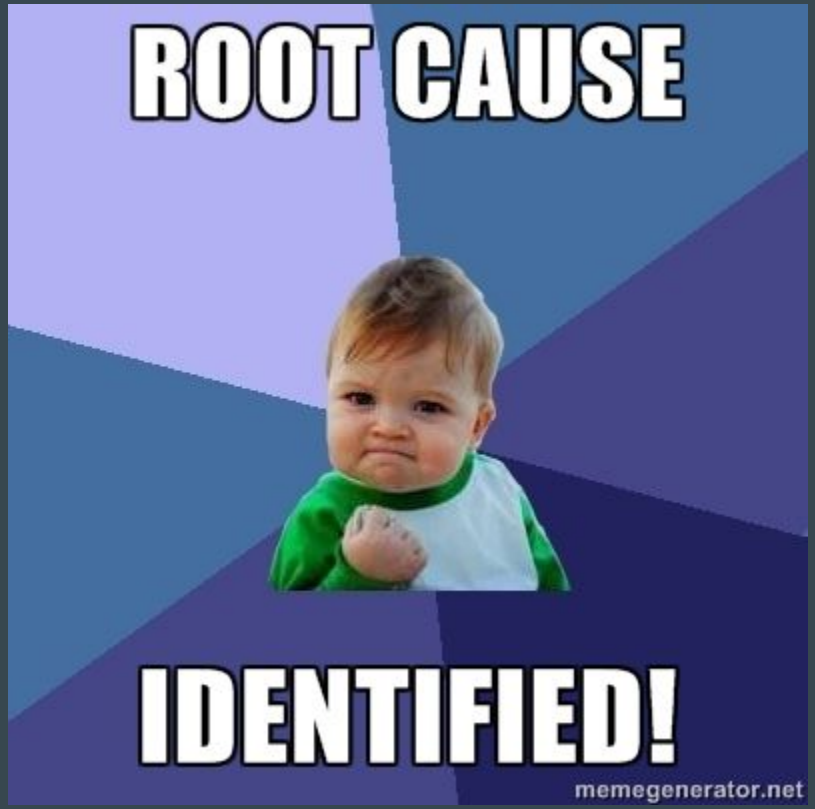
Lack of technical skills

Lack of domain knowledge

But those are symptoms, not the causes

Attenuating/removing the symptoms is not going to solve the problem

We need to find the root cause



History of human species is the history of technology

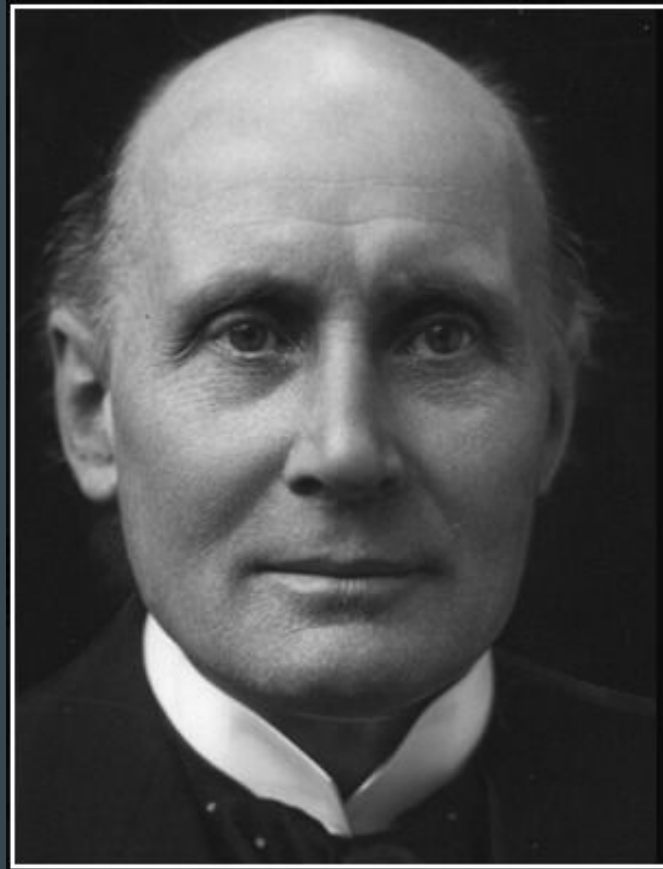
In the early, pre-technological days, every human generation lived pretty much the same way as all previous human generations

That's similar to how animals live; so long as some species does not get extinct, each generation lives the exact same lifestyle as any other generation of the same species

Humans are different

We are prone to inventing things that our ancestors never dreamed about

And we do it by focusing on technological advancements



Civilization advances by extending the number of important operations which we can perform without thinking of them.

— Alfred North Whitehead —

AZ QUOTES

How do we perform important operations without thinking of them?

We outsource those operations

There are two ways to outsource important operations: create organizations responsible for carrying those important operations, or automate those important operations by utilizing machines

Of the two, automation appears to be more cost effective (and hopefully more predictable & more convenient). It is for that reason that the progress of our civilization always seems to lean toward more automation

But automation means rigidity

To be able to automate some process, we need to make sure it is repeatable

And if some process is repeatable, that means it is a rigid process

When we automate such process and virtualize it, any future changes to the actual process will be ignored by its automated equivalent, which means our automation is not that useful in the long run

We all know that there are precious few things in this world that don't change; most of the time, what seemed like a solid, unchangeable thing quickly turns into a moving target

Automated systems in use today are expensive to maintain

Automated systems need to be constantly modified to accommodate the unstoppable onslaught of the changes that occur in our world

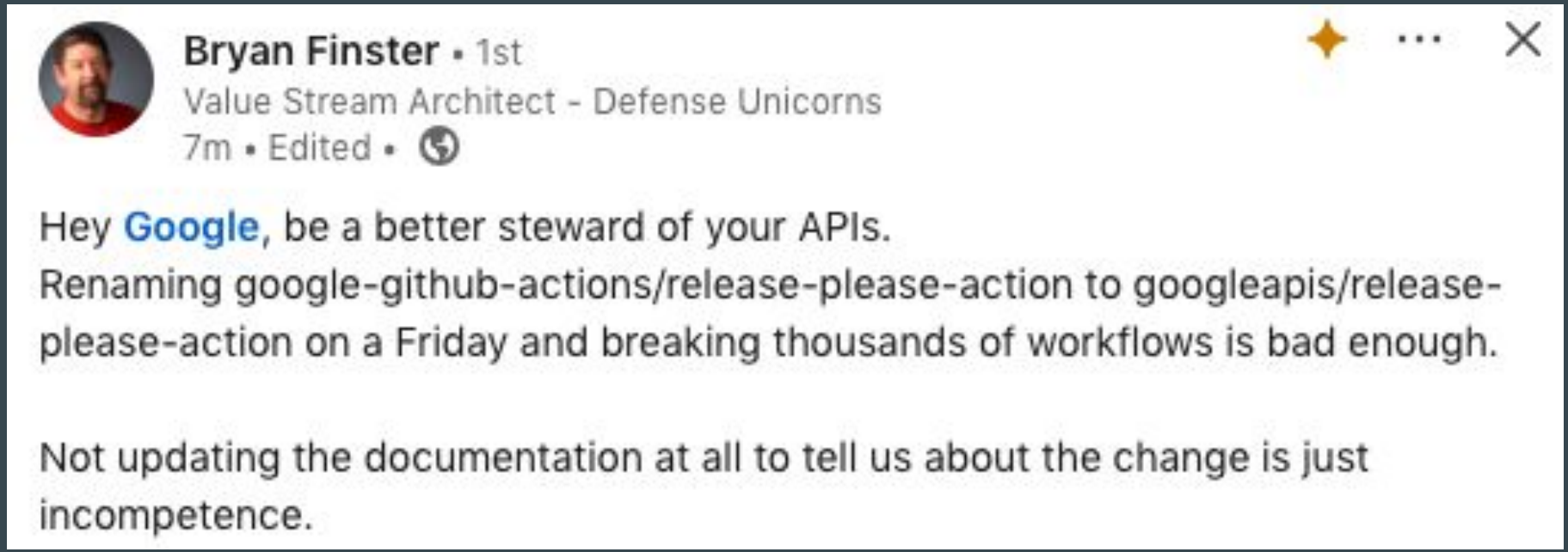
We see that automated systems are brittle (digital rot never sleeps)

What usually causes the brittleness are dependencies

Parts of the system depend on some other parts of the system, which in turn depend on some other parts, and so on. Such dependencies tend to harden into tightly coupled brittle surfaces. As soon as we start applying the pressures of unavoidable changes, the brittle surface begins to crack. More often than not, such cracks quickly turn into serious breakages

For example...

Some service we depend on suddenly changes:



Bryan Finster • 1st
Value Stream Architect - Defense Unicorns
7m • Edited •

Hey Google, be a better steward of your APIs.
Renaming google-github-actions/release-please-action to googleapis/release-please-action on a Friday and breaking thousands of workflows is bad enough.

Not updating the documentation at all to tell us about the change is just incompetence.

The goal of good engineering is (should be?) to avoid situations like the one with the Google API

Things keep changing not only at a brisk pace, but also without us being notified of the change

We cannot build robust reliable systems if our solutions keep breaking because previously agreed-upon ways get modified without us being aware of the change

How can we make future-proof automation?

A future-proof automated system would be a system that will not stop working as expected once the rules governing that system change

Is there such system in existence today?

Passport renewal process

Passport renewal process

Steps for passport renewal process

1. Search online for the nearest passport office
2. Travel to the office
3. Upon arrival enquire about the passport renewal process with the concierge
4. Pick the passport renewal brochure available on display
5. Examine the brochure to learn what are the requirements for the renewal
6. Follow the instructions, procure necessary photos/documents, fill out the form, bring it back it, pay the free, and wait

Takeaways

To begin the process, we need to obtain the entry point – the address of the government service for renewing passports

Once we make it to the entry point, we need to pick up the instructions found inside the office

Follow the instruction printed in the passport renewal brochure and provide all necessary artifacts

Once all the instructions have been fulfilled, we must submit the request, pay the fee, and wait for the response

Why is this process not brittle?

All the information necessary for successfully completing the transaction is available at the government office

That information is applicable right now

By making an effort to go straight to the source (i.e., the government office), I avoided the risk of acting based upon some stale information and in the process wasting mine, and everyone else's time

Had I relied on some prior knowledge of the passport renewal process, I might have missed on the latest changes in the government regulations

Piecemeal discovery

It is possible to design an automated system in such ways that when the conditions in one part of the system change, the dependent parts of system doesn’t have to be changed

How do we do that?

Emulate the bureaucratic procedure

Producers and consumers

The government office is the producer of passports, and the citizens are the consumers of passports

In-between the producer and the consumers lie the rules

The rules are prone to change

The biggest challenge is how do we protect the consumers from having to learn about the rules, and then when acting on the learned rules, finding out that, oops! the rules have changed!

How do we avoid making our automated system brittle?

Piecemeal discovery online

1. I would have to somehow obtain the online address of the government passport renewal service (I could do it the similar way I obtained the government physical address — by searching online)
2. Once I get the online address (the URL), I would click on the hyperlink, which will take me to the online office
3. Once in the online office, I would look for the online brochure
4. When I find the online brochure, I would have to dedicate some time to study the requirements for passport renewal
5. I would then obtain all necessary artifacts (i.e., two headshots and anything else specified in the instruction manual)
6. I would then fill out the online form, attach the required artifacts, click on “I accept terms and conditions” checkbox, and submit my passport renewal application
7. Lastly, I would have to process the required payment before I could expect the government officials to spring into action and start working on renewing my passport

In-band process

All the information necessary for successfully renewing my passport was supplied by the system I was interacting with

There wasn't any need, at any point during the above 7 steps, for me to go and obtain additional information from some other resource

In-band process means that the information, the knowledge of what needs to be done is self-contained, self-explanatory

There is no need to seek counselling from any third party in order to be able to successfully reach the goal

Piecemeal discovery is a resilient approach

By relying on information that travels with the response, the clients do not have to prematurely bond with any prior information/knowledge

The set of instructions necessary to be able to process the response is contained in the response itself

The set of instructions necessary to process the response is obtained at the last responsible moment

That means that when the knowledge needed to process a response changes, the client does not break

Technically, this is called Late Binding

No one has a crystal ball

How to deal with brittle automated systems?

We pretty much know that if we automate a system based on today's rules, it's inevitable that in the future some of those rules will change

When that happens, the automated system will break

The way we tend to deal with unavoidable breakages is to make an effort to design our automation in such ways that will enable swift and painless and hopefully risk-free future changes

We are accustomed to rolling with the punches

Why don’t we automate a piecemeal approach?

So far, we’ve seen that piecemeal approach works well with human users

But can machines be designed to take advantage of the piecemeal approach?

Machines are rigid

Unlike humans, machines lack agency and are therefore devoid of goals

It is for that reason that we need to preload the knowledge describing what to do into machines

Without that prior knowledge, machines would not be in the position to make the decision what to do next

Machines lack sentience

Can we overcome the machine rigidity?

Seems like a daunting, even impossible challenge to let machines make decisions at the last responsible moment

But if we do not enable that, we will forever be stuck in the world of rigid, brittle, easily breakable automated systems

There is, however, one way that I am aware of that could enable breaking out of the rigidity prison

REST

REpresentational State Transfer

State transfer implies that a resource representation (say, a passport) changes its state from ‘expired’ to ‘renewed’

What is representational in there?

A passport represents the citizen status with regards to them being able to travel abroad

If resource representation of someone’s ability to travel abroad status is ‘expired’, then that status is invalid (i.e., the citizen is not fit for traveling abroad)

When resource representation transfers its state to ‘renewed’, the underlying resource (i.e., passport of a citizen who is cleared to travel abroad) is updated

How is REST implemented?

REST is implemented via 6 constraints:

1. Client-server architecture
2. Stateless
3. Allows caching
4. Must have uniform interface
5. Is a layered system
6. Allows for code-on-demand

Client-server constraint

REST is an architectural style that is only applicable to systems that are running on a network

REST describes both the client and the server, as they communicate on the network

The stateless constraint

Every request should encapsulate all information necessary to respond to that request, with no side state or context stored on either the client or the server, nor elsewhere (no out-of-band knowledge needed)

This constraint ensures simplicity, which in turn ensures robustness and resilience

The caching constraint

Any future requests for the same resource representation should be cache-able

This means that there must be explicit information on the cache-ability of responses for future requests of the same resource representation

The uniform interface constraint

The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components... In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state

Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures

Identification of resources

Universal Resource Locator (URL)

Simplicity – removal of any possible ambiguity

Manipulation of resources through representations

A single resource can be represented in many (innumerable?) ways

It is possible to use media type to signal to the server what kind of resource representation is acceptable

Self-descriptive messages

Everything necessary for deciding what is possible to do must be contained in the message itself, in the form of hypermedia controls (in-band knowledge)

Client does not have to know anything ahead of time (i.e., no out-of-band knowledge is required)

Hypermedia As The Engine of Application State (HATEOAS)

Made possible by the self-descriptive messages

A self-descriptive message enables the calling client to discover what capabilities are available on the server

The calling client can then decide what action to take

It is the uniform interface of the hypermedia that provides the ability to control the state of the application

That capability is the engine that drives the interaction between the client and the server

HATEOAS is not Remote Procedure Call (RPC)

Software developers at large are trained to implement the functionality of the app by calling various procedures (either locally, or remotely)

To call a procedure, we must know beforehand not only the location of the code that implements the procedure, but also the signature of the procedure (i.e., the API) as well as the type of the returned values

And since APIs are subject to inevitable change, the brittleness is controlled by imposing the versioning of the functionality/API

HATEOAS does not function like that

There is no versioning of the functionality in HATEOAS

RPC is Early Binding (Tight Coupling)

Before a client code can prepare the context for calling a remote procedure, the authors of the code must know everything there is to know about the signature of the remote procedure, as well as the types of any returned values

That situation is what we call Early Binding

The client and the server are tightly coupled, which is making the arrangement brittle

If the procedure signature or the type of the returned values change, the calling client breaks

How can we make sure the calling client doesn't break?

There is no way for the calling client code to detect that the signature/type of return values of the called procedure has changed

The onus is on the server to play nice and not let the breakage happen

It is customary (and often contractually obliging) to maintain backwards compatibility by versioning the API

That arrangement, of course, increases and bloats the accidental complexity

Accidental complexity is by far the worst problem that's plaguing software development!

Inventory is waste

Inventory may be an asset in some industries, but it is always a waste in software

Maintaining multiple versions of the same app is insanely wasteful

And yet we all seem to be doing it!

We're doing it because we want to maintain backwards compatibility

Which is meaningless – time marches on, and business operations grow and mature; why bother with backwards compatibility?

Why needlessly hang on to something that is not true anymore?

The only reason: immature technology

HATEOAS is Late Binding

In HATEOAS the calling client knows nothing about the existing capabilities of the service provider (nor does the calling client assume anything about the capabilities of the service provider)

The calling client discovers the capabilities AFTER it makes the call

That approach is diametrically opposite from the Remote Procedure Call approach, where the calling client must know the capabilities of the service provider BEFORE making the call

It is for that reason that in HATEOAS the inventory does not bloat and the accidental complexity is kept at bay

HATEOAS implements Loose Coupling

The calling client and the server are only initially coupled, but that coupling does not need to be tight

All that the client needs to know is the location of the capability it intends to utilize

The knowledge of the location of the subsequent capabilities is not coupled between the client and the server

That knowledge gets revealed to the client once the server responds to the client's call

It is not possible to implement looser coupling than that

HATEOAS mimics bureaucratic systems

Same as with the procedure to renew a passport, all that a client needs to know is the entry point – the first point of contact between the client and the service provider

From that moment on, the client is not expected to know anything nor is it expected to memorize anything

The client remains stateless

Statelessness is desirable because less inventory

Actually, in HATEOAS both the server and the clients are stateless

This is similar to how real life bureaucracies function – they don't know anything about their client's details beforehand, and the clients don't know anything about the service provider's details

They learn/uncover as they go

There is an elephant in the room

How can machines deal with HATEOAS?

We've seen that hypermedia, with self-descriptive messages and uniform interface, enables human clients to piecemeal discover the capabilities of the app

That discovery seems innate to all literate humans – it's intuitive

But how can machines intuit that discovery?

It seems like hypermedia is useless to machines

Therefore, it's back to the brittle RPC world

Meet Carson Gross, creator of HTMX

HYPERMEDIA SYSTEMS WITH CARSON

BUY HIS BOOK ⟶

Carson is a thought leader in the field of software architecture

His creation https://htmx.org/ is taking the world of web development by the storm

One reason is that https://htmx.org/ can drastically minimize accidental complexity

Carson's mission is to enhance the original hypermedia, as implemented via HTML tech

Carson argues that HATEOAS is only applicable to human users

HATEOAS is for Humans

8 May 2016

##TLDR##

- HATEOAS is actually a simple concept: a server sends both data and the network operations on that data to the client, which has no special knowledge about the data.
- This simple but powerful idea has been lost because it is typically examined in terms of machines and APIs, rather than in terms of humans and HTML.
- Humans are uniquely positioned to take advantage of the power of HATEOAS in a way that machines are not (yet) because they have agency.

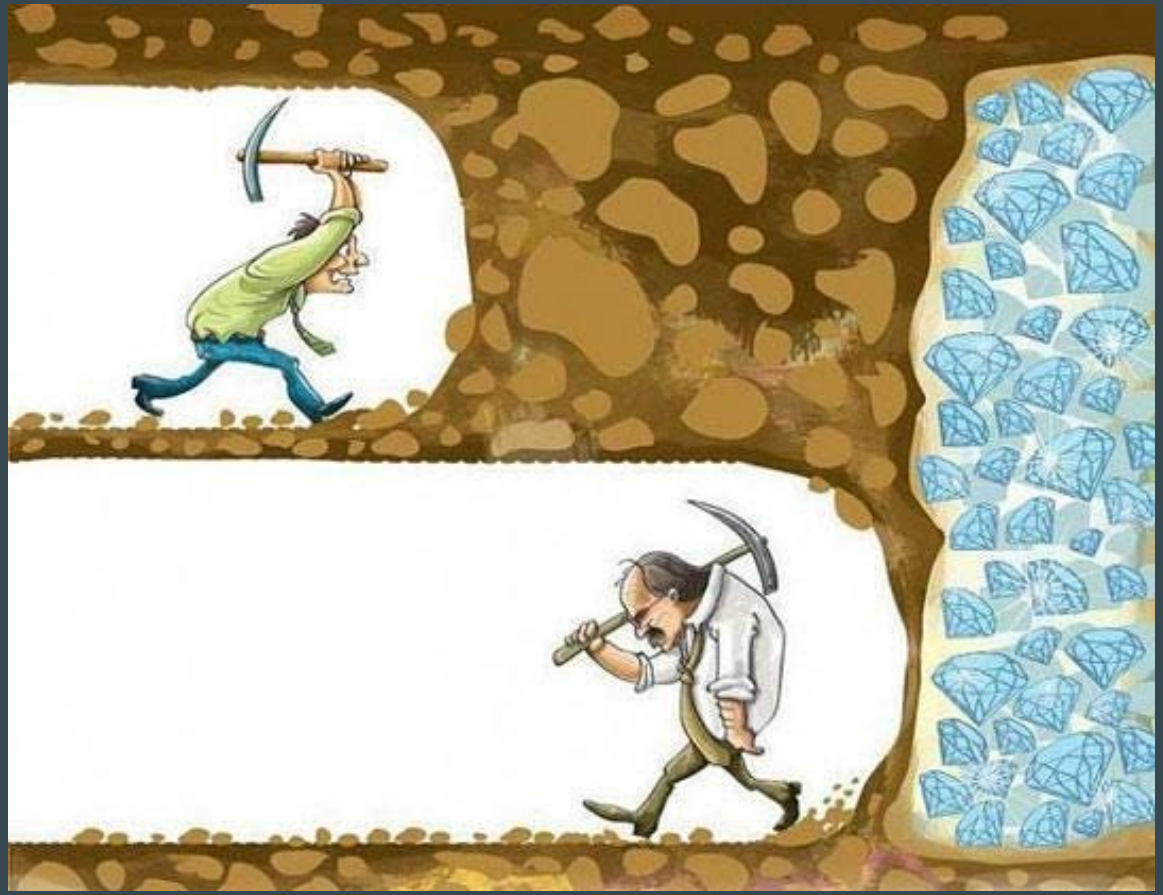
HATEOAS is wasted on machines?

In this essay (https://intercoolerjs.org/2016/05/08/hatoeas-is-for-humans.html) Carson argues the following:

> But what I am convinced of, and what I hope to convince you of, is that HATEOAS is largely wasted on machines.
>
> HATEOAS is for humans.

What a missed opportunity!

Should we give up?



Can machines process hypermedia?

Without the ability to parse and process hypermedia, any calling client code is only safe if it's tightly coupled to the already tried and tested version of the app

If that's the case, our attempts to avoid brittleness and bloated inventory are defeated

We cannot help but resign to those flakey, brittle Remote Procedure Calls

That, however, is not necessarily true

There is no reason why machines could not parse hypermedia and detect in the self-descriptive messages what is the next available capability (or capabilities) of the app

How to process self-descriptive messages?

The only way to programmatically process self-descriptive messages is by severely constraining the domain of the client code

Server apps are never general purpose systems; they're always narrowly focused on some business domain

A client wishing to interact with such services needs to be properly equipped with the knowledge about the domain

However, clients need not be equipped with the knowledge that describes HOW is the business domain functionality implemented

Unlike general purpose hypermedia clients, business domain clients have narrow focus

General purpose hypermedia clients (basically, web browsers) possess no domain knowledge

Web browsers are not opinionated

Web browsers know how to render any resource representation they receive from hypermedia servers

Web browsers have no idea (nor do they care about) the meaning of the resource representation

Web browsers indiscriminately interact with any hypermedia server, regardless of the domain that resources represent

Clients implementing HATEOAS functionality are narrow purpose

If a client wishes to interact with a hypermedia server, the client needs to limit that interaction only to a clearly defined business domain

We need a concrete example to illustrate that

Example – Client posts HTTP GET Request

A user wants to book a flight

The user obtains out-of-band information about the online location of the airline booking service

The user sends HTTP GET request:

```
GET /flights/book HTTP/1.1
Host: airline.example.com
```

The server sends an HTTP Response

The response contains hypermedia controls that prompt for more information (i.e., HTML FORM that enables the input of necessary details, such as the departure and return dates, the departure and arrival airports, etc.)

The client submits HTTP POST Request

The POST request contains the payload (i.e., departure and return dates, the departure and arrival airports, etc.)

That payload gets processed by the server...

The server responds with available flights (showing here only 1 flight due to limited screen space)

```
HTTP/1.1 200 OK
"flights": {
   "flight": {
        "flight_number": 12345,
        "departure-date-and-airport": {
            "date": "2024-06-01",
            "airport": "YVR"
        },
        "return-date-and-airport": {
            "date": "2024-06-21",
            "airport": "CDG"
        },
        "return-ticket": {
            "price": 1850,
            "currency": "CAD"
        },
        "links": {
            <link rel="book" href="http://airline.example.com/flight/12345/book"/>
            <link rel="details" href="http://airline.example.com/flight/12345/details"/>
            <link rel="upgrade" href="http://airline.example.com/flight/12345/upgrade"/>
        }
    }
}
```

What's the role of the 'rel' attribute?

The rel attribute (found in a self-descriptive message) specifies the relationship between the current resource representation and the linked resource capabilities

In the example, current resource representation is the flight 12345

There are three linked resource capabilities:
1. Book the flight
2. Flight details
3. Upgrade the flight

From this self-descriptive message, the client learns what is the current resource capable of

The client books the flight 12345

The client code activates the link

```
<link rel="book" href="http://airline.example.com/flight/12345/book"  />
```

(we'll skip the processing of the flight booking for brevity sake)

Suppose the client then sends HTTP GET Request

```
GET /flights/12345 HTTP/1.1
Host: airline.example.com
```

The server sends HTTP Response with resource representation

```
HTTP/1.1 200 OK
"flights": {
   "flight": {
       "flight_number": 12345,
       "departure-date-and-airport": {
           "date": "2024-06-01",
           "airport": "YVR"
       },
       "return-date-and-airport": {
           "date": "2024-06-21",
           "airport": "CDG"
       },
       "return-ticket": {
           "price": 1850,
           "currency": "CAD"
       },
       "links": {
           <link rel="cancel-booking" href="http://airline.example.com/flight/12345/cancel-booking" />
           <link rel="details" href="http://airline.example.com/flight/12345/details" />
           <link rel="upgrade" href="http://airline.example.com/flight/12345/upgrade" />
       }
   }
}
```

The interaction between the client and the server is dynamic

Clients need not know in advance that the server has the capability to cancel booked flights

That capability is only exposed to the clients if the flight has already been booked, and if the client inquires again about the capabilities of the resource

Similarly, many other changes to the server functionality get exposed to the clients at the last responsible moment – lazy loading/late binding/loose coupling

Can narrow business domains produce agency?

Unlike humans, machines possess no agency (zero sentience)

Narrowing the focus increases the chances of enabling machines to make correct decisions regarding what is the next appropriate action, given the business context

LLMs and machine learning can contribute quite a lot toward that goal

Specialized GPTs are showing lots of promise

Still early days, but the field is dynamic and is definitely not showing signs of slowing down


In summary,
this is what I
hope you learned.
imgflip.com

What makes bureaucratic systems resilient and why should we emulate them when developing software?

There are two characteristics of bureaucratic systems that make them robust and resilient:

1. Statelessness
2. Apathy

Designing automated systems to implement statelessness and apathy (i.e., lack of eagerness for details), is the only way to achieve true loose coupling

True loose coupling is absolutely necessary for making our systems future proof


THANK YOU
FOR LISTENING


I HAVE QUESTIONS
LOTS OF QUESTIONS

I'm available for consulting engagements

Email: alexbunardzic@gmail.com

Medium: https://medium.com/@alexbunardzic

Twitter: https://x.com/alexbunardzic

LinkedIn: https://www.linkedin.com/in/alexbunardzic/

Substack: https://substack.com/@alexbunardzic

Coding interviews with Alex
https://www.youtube.com/playlist?list=PLpQwCufTgsb21EuS262M1dn1dhgqInndD

Software breakthroughs for the 21st century
https://www.youtube.com/playlist?list=PLpQwCufTgsb2z_FW8FFbOzr7uBU-kuydQ

Diary of a perplexed software developer
https://www.youtube.com/playlist?list=PLpQwCufTgsb1jE8l6JuxkHAJnAbLzb46v