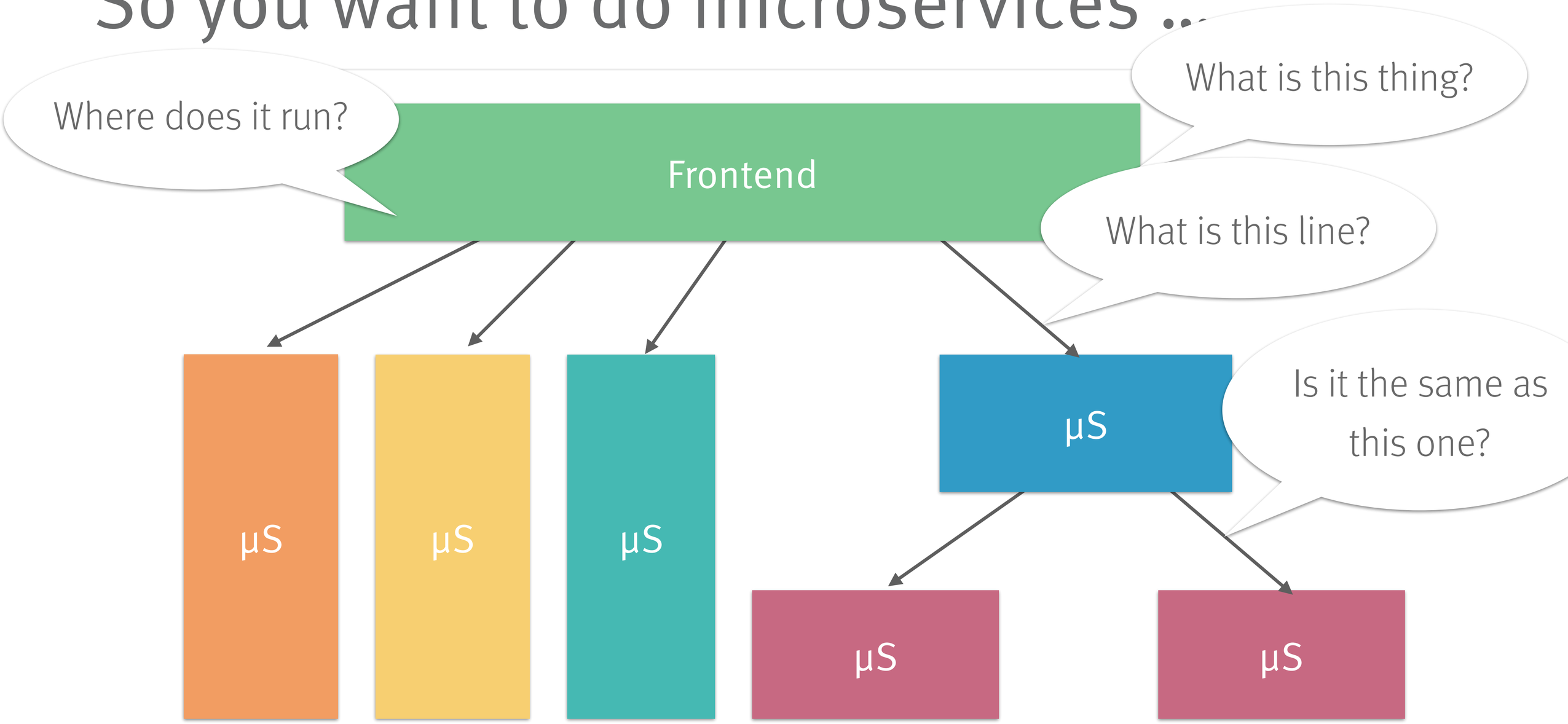
Wait, what? Our microservices have actual human users?

Stefan Tilkov, innoQ
@stilkov

innoQ

So you want to do microservices ...

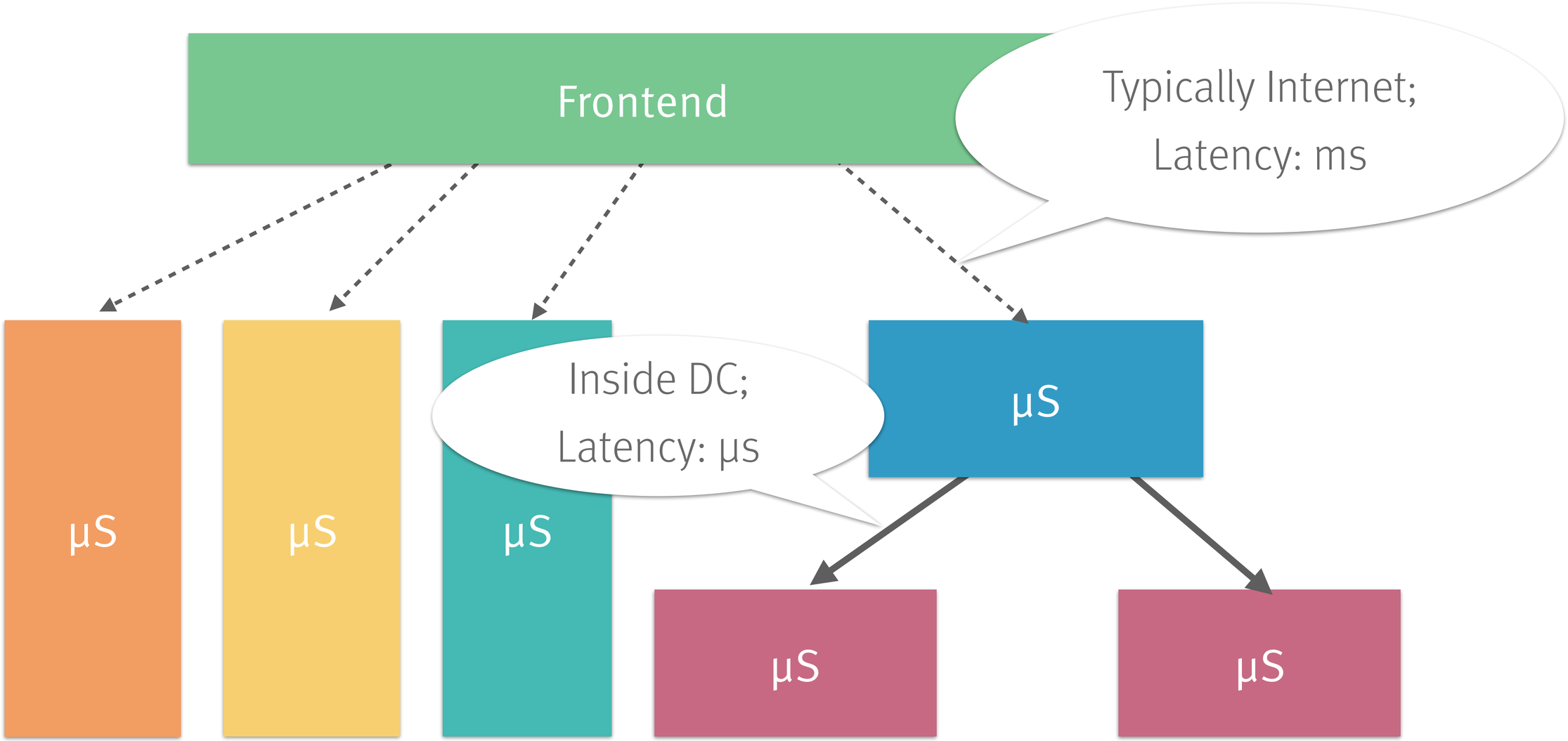


Challenging Assumptions

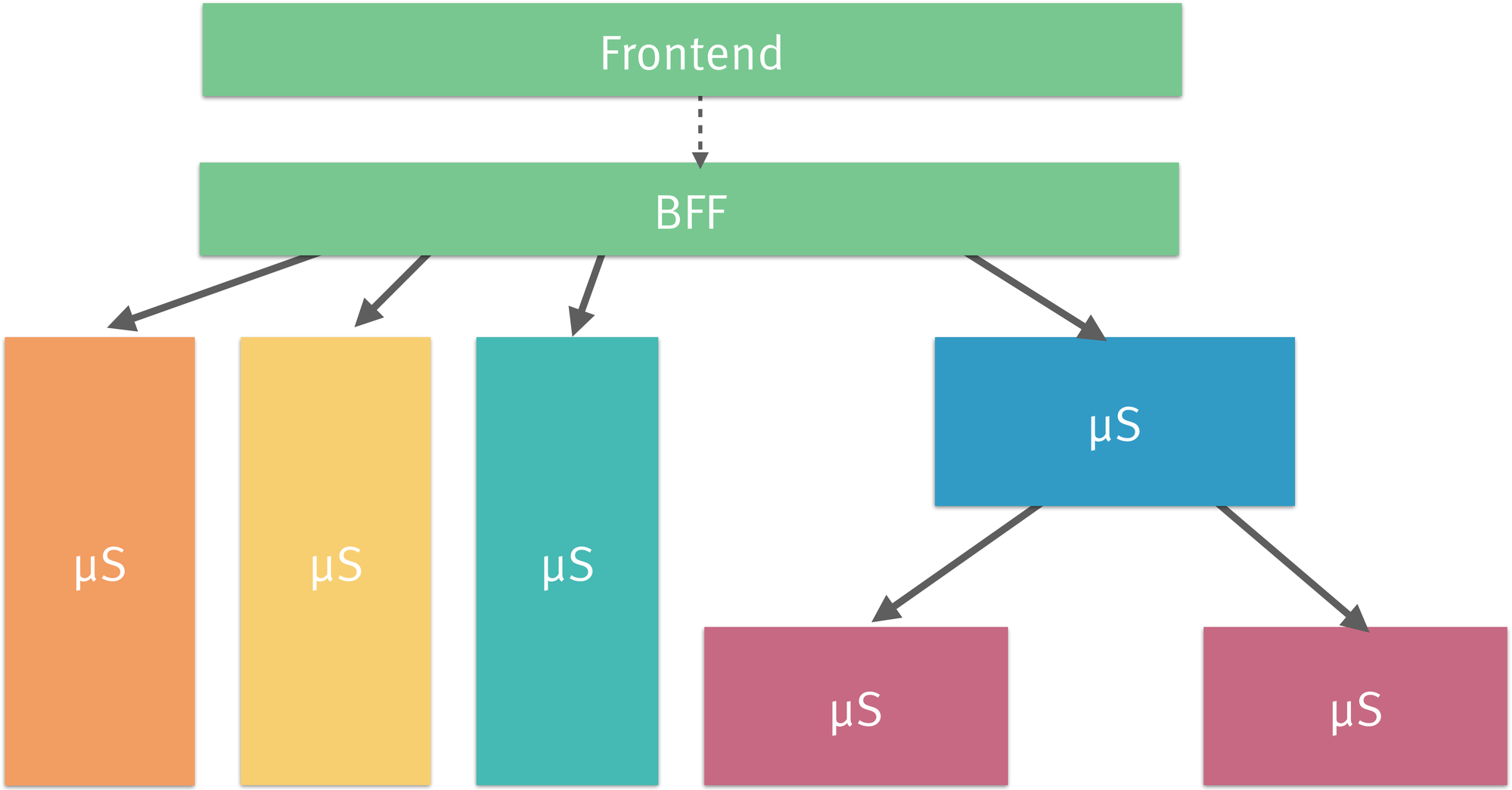
Assumption:
Orchestration is cheap

No.

“Really remote” vs. “almost local”



“Backend for Frontend”



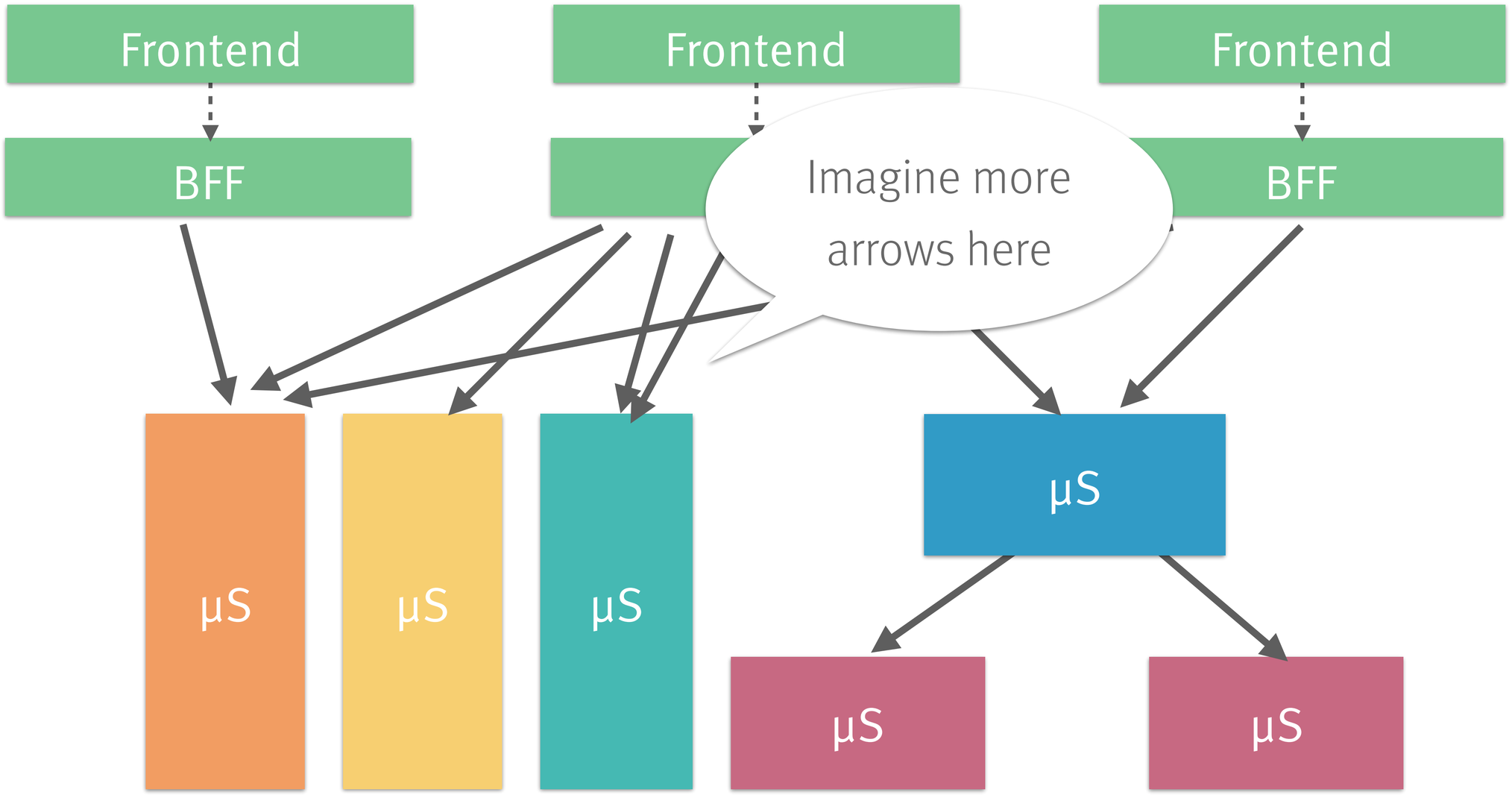
http://samnewman.io/patterns/architectural/bff/

Assumption:

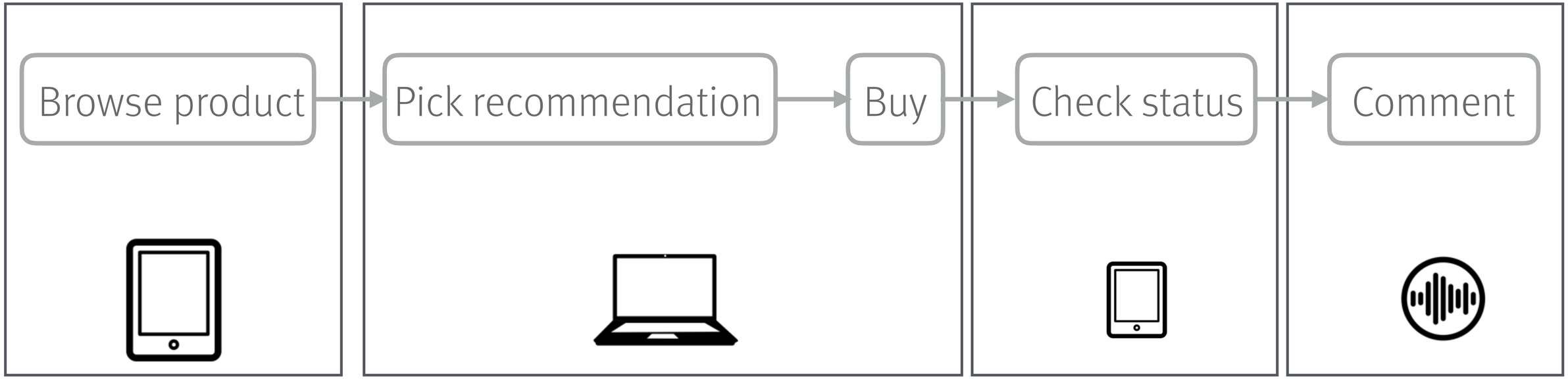
Channels matter

Not as much as you think

Multiple BFFs for different clients

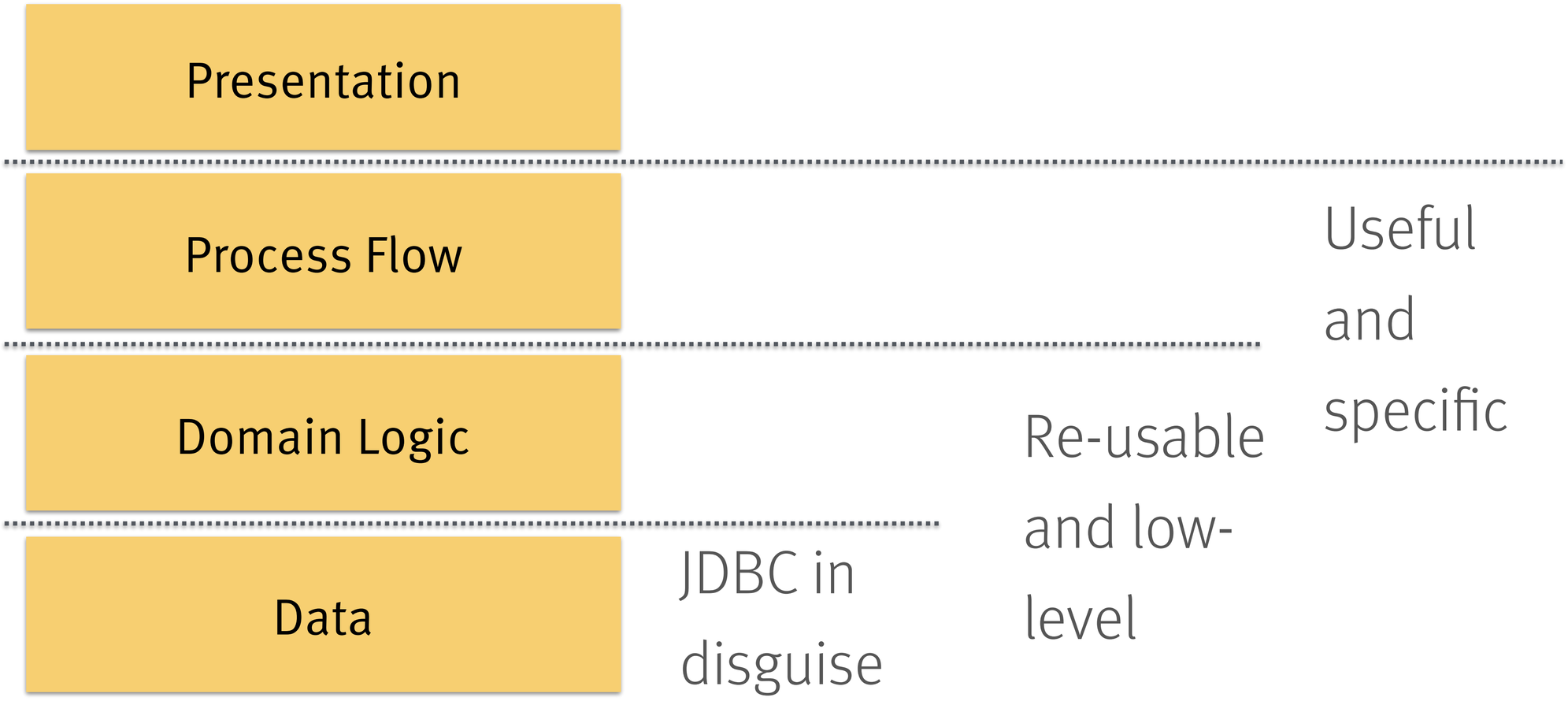


Multiple channels – facing *every* user



Users expect a seamless experience across channels – everything accessible, everywhere.

Build services that actually do something



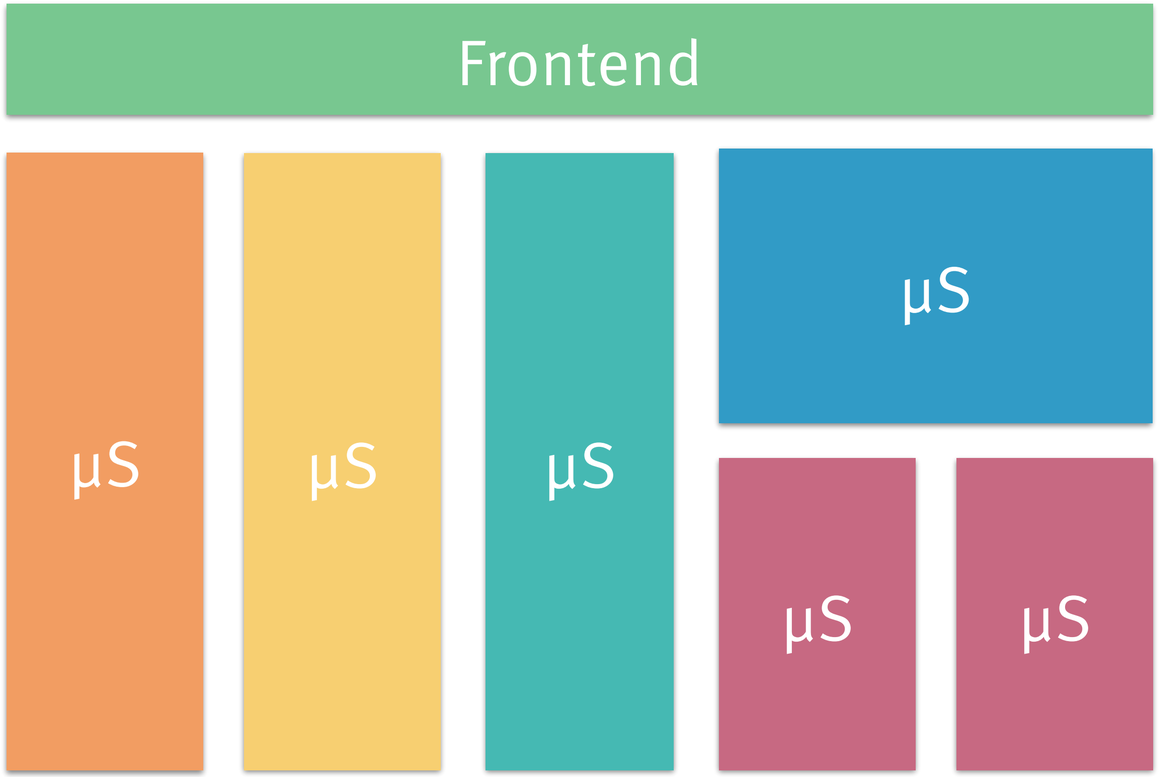
Assumption:

Services matter most

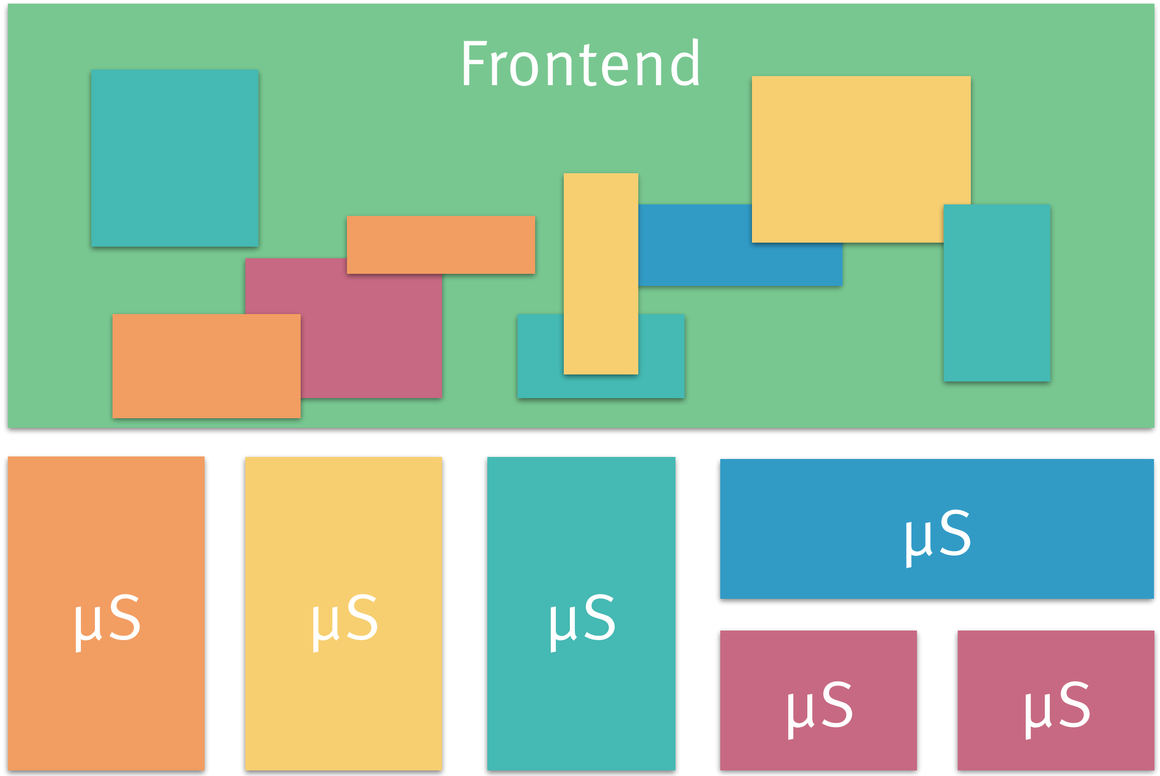
(a.k.a. “SOAs Original Sin”)

UIs matter more.

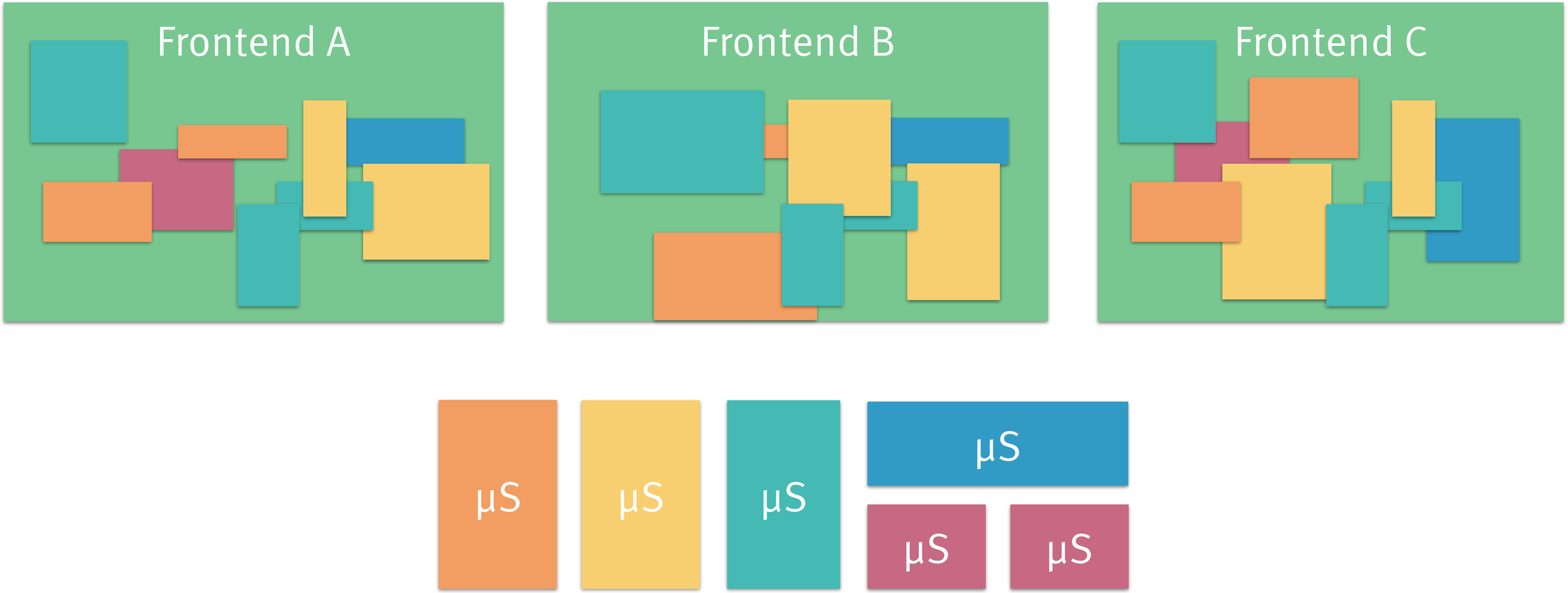
Frontend + services in a backend architect's mind



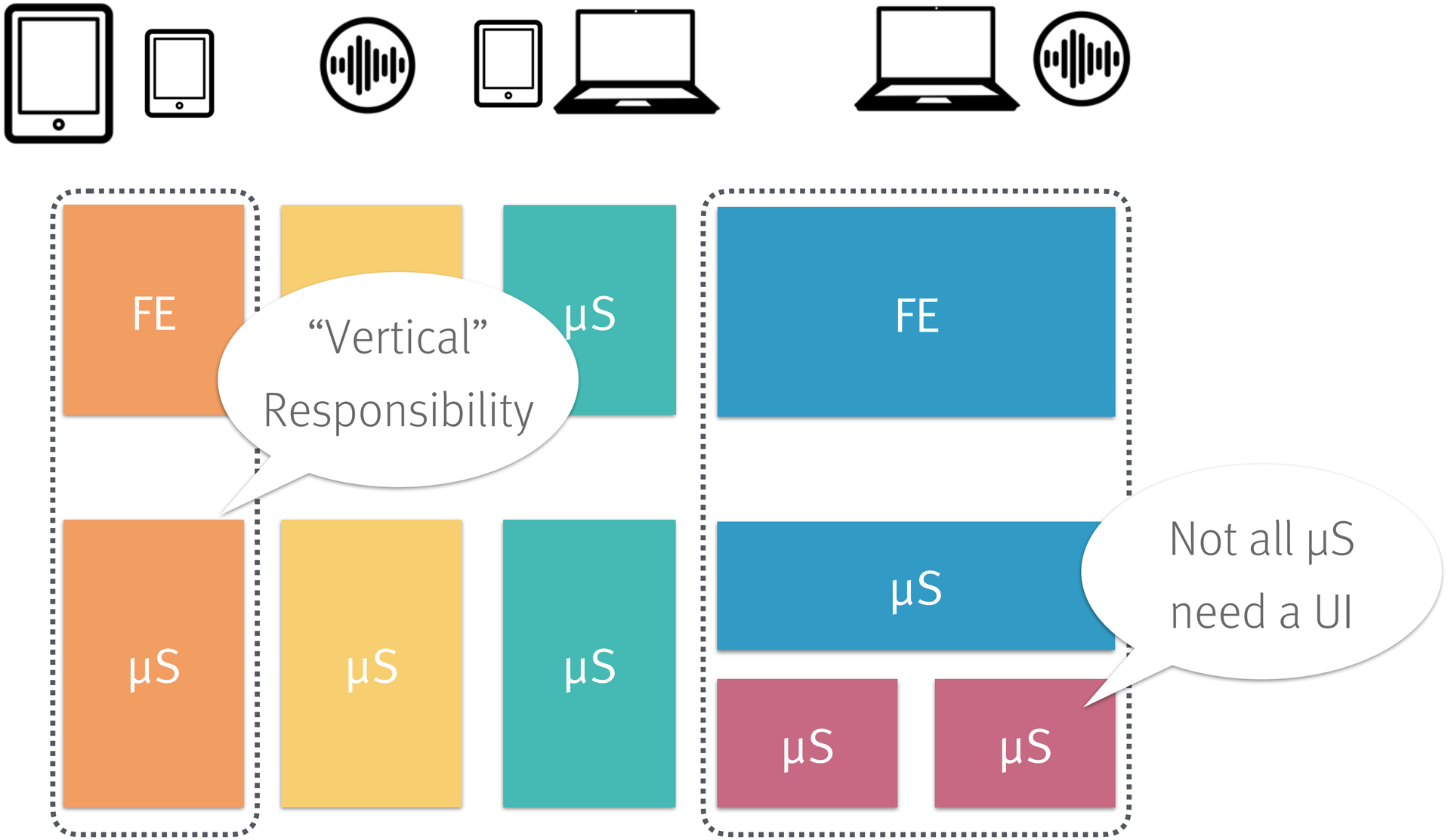
Frontend + services in the real world



Redundancy with Multiple BFFs



Ideal world UI componentization



SCS: Self-contained Systems

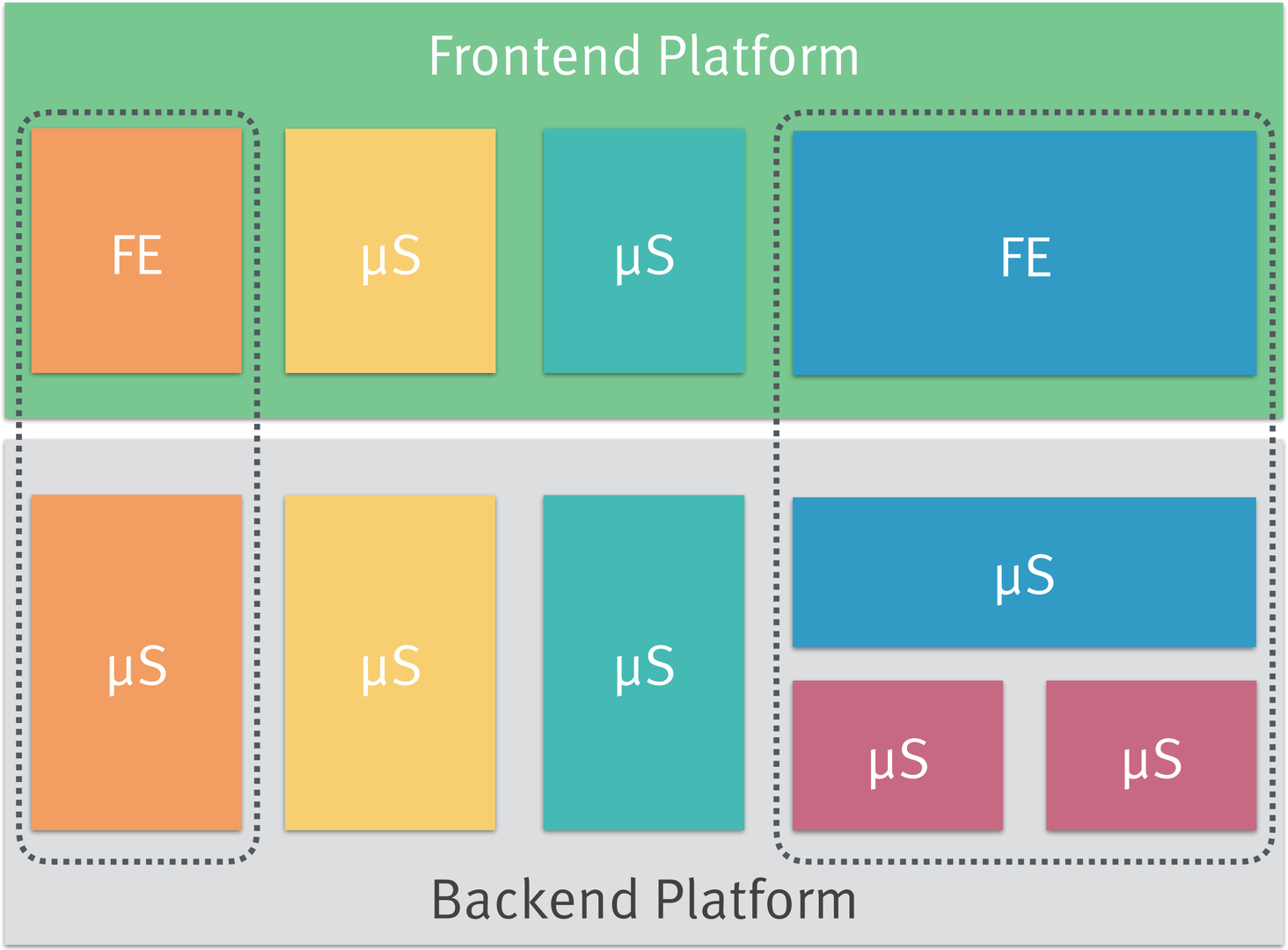
http://scs-architecture.org

Assumption:

Frontend technology is an implementation detail

Not at all.

More than one platform



Microservices backend platform goals

Backend Platform

- As few assumptions as possible
- No implementation dependencies
- Small interface surface
- Based on standards
- Parallel development
- Independent deployment
- Autonomous operations

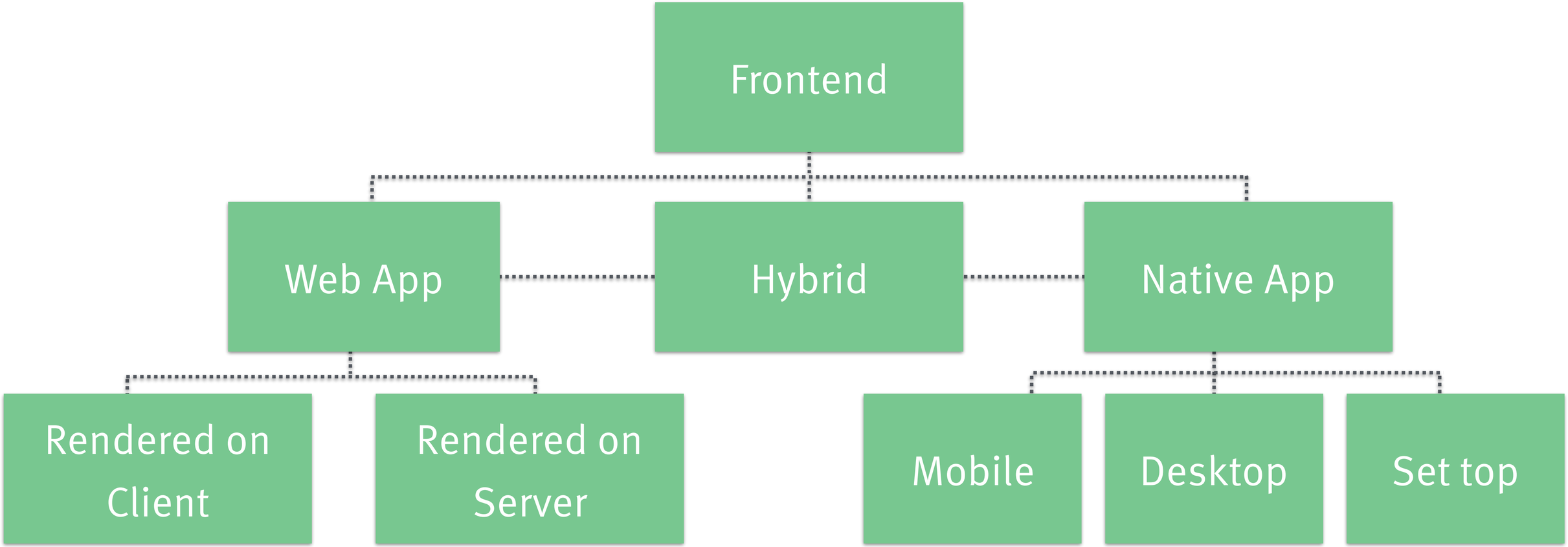
What’s the frontend platform analogy?

Frontend Platform

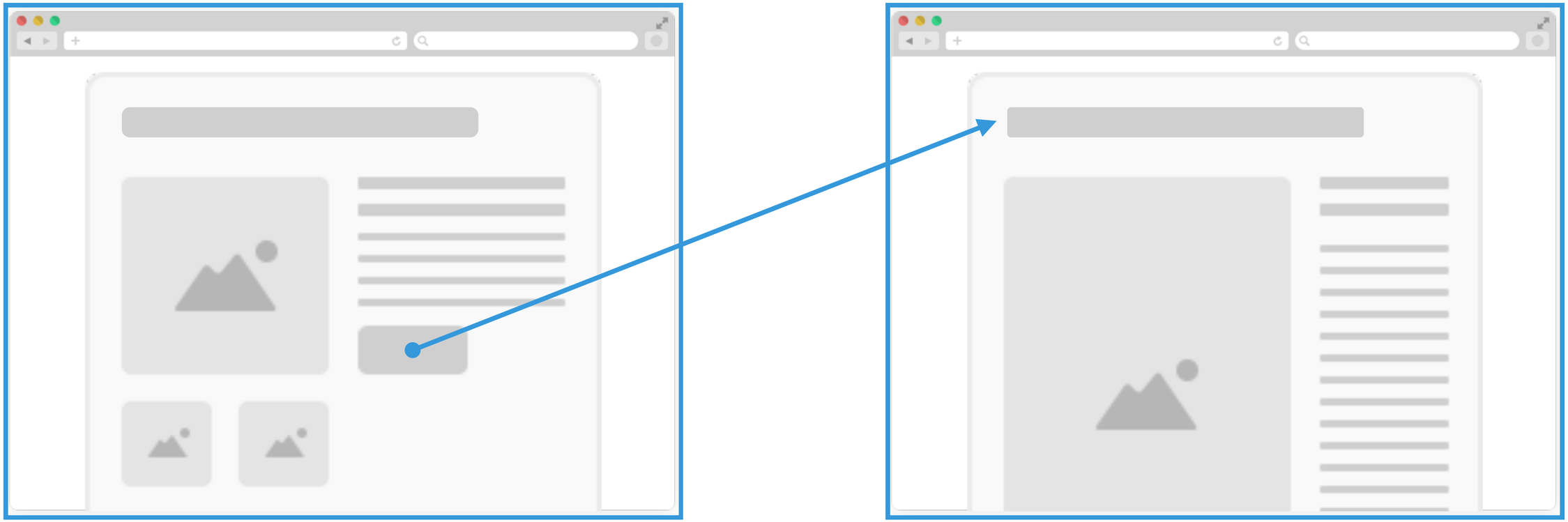
Backend Platform

- As few assumptions as possible
- No implementation dependencies
- Small interface surface
- Based on standards
- Parallel development
- Independent deployment
- Autonomous operations

Frontend, we've got frontends



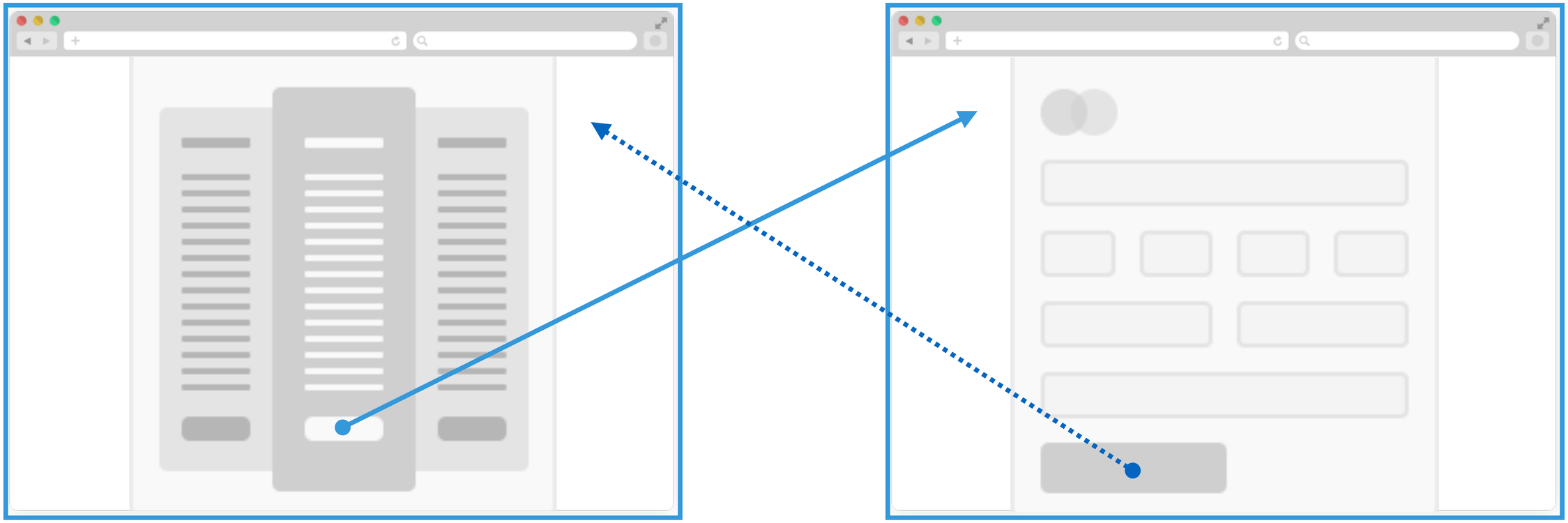
Web UI Integration: Links



System 1

System 2

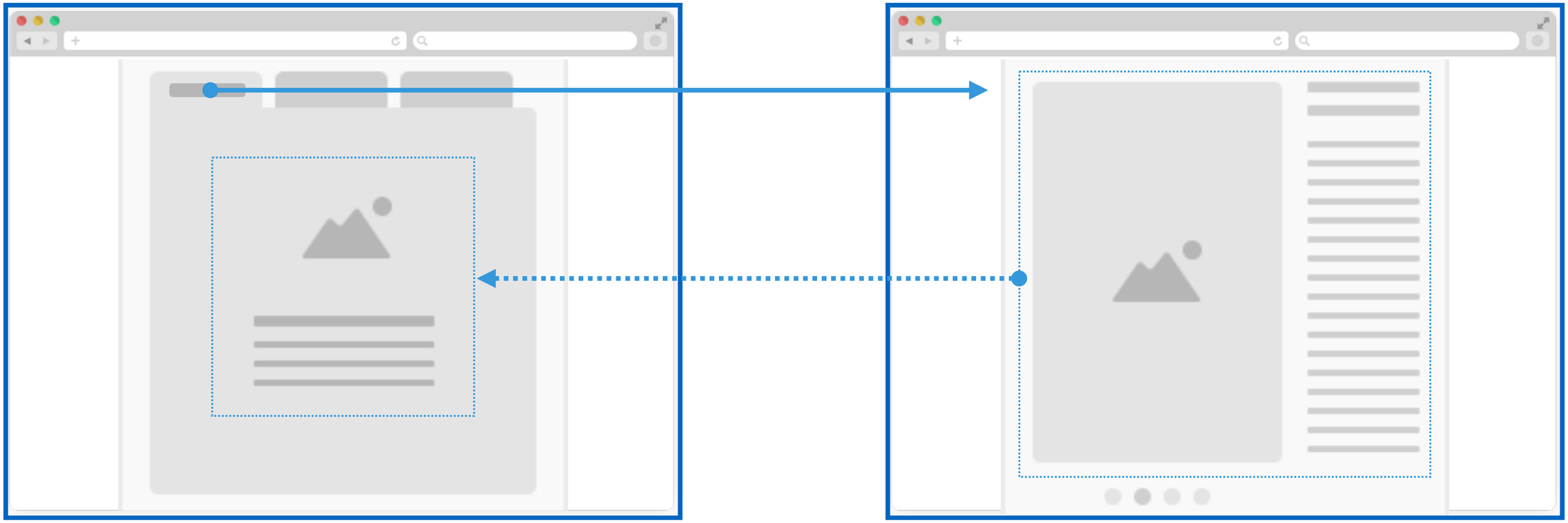
Web UI Integration: Redirection



System 1

System 2

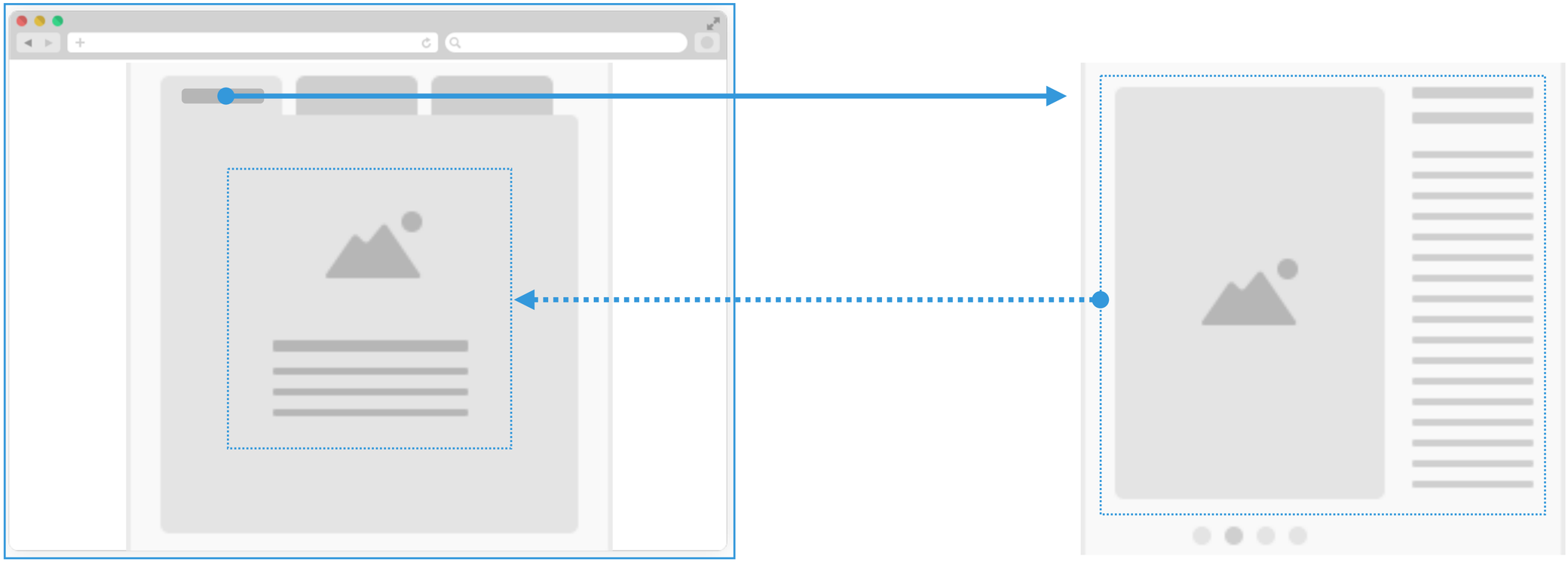
Web UI Integration: Transclusion



System 1

System 2

Web UI Integration: Web Components?



System 1

Component

The browser as a platform

Frontend Platform

Backend Platform

- Independent applications
- Loosely coupled
- Separately deployable
- Based on standard platform
- Updated on the fly
- Any device

How to get away with “just” the Web

- Mobile first
- Responsive design
- Progressive enhancement

- Shared assets
- Pull vs. push
- Sacrifice (some) efficiency

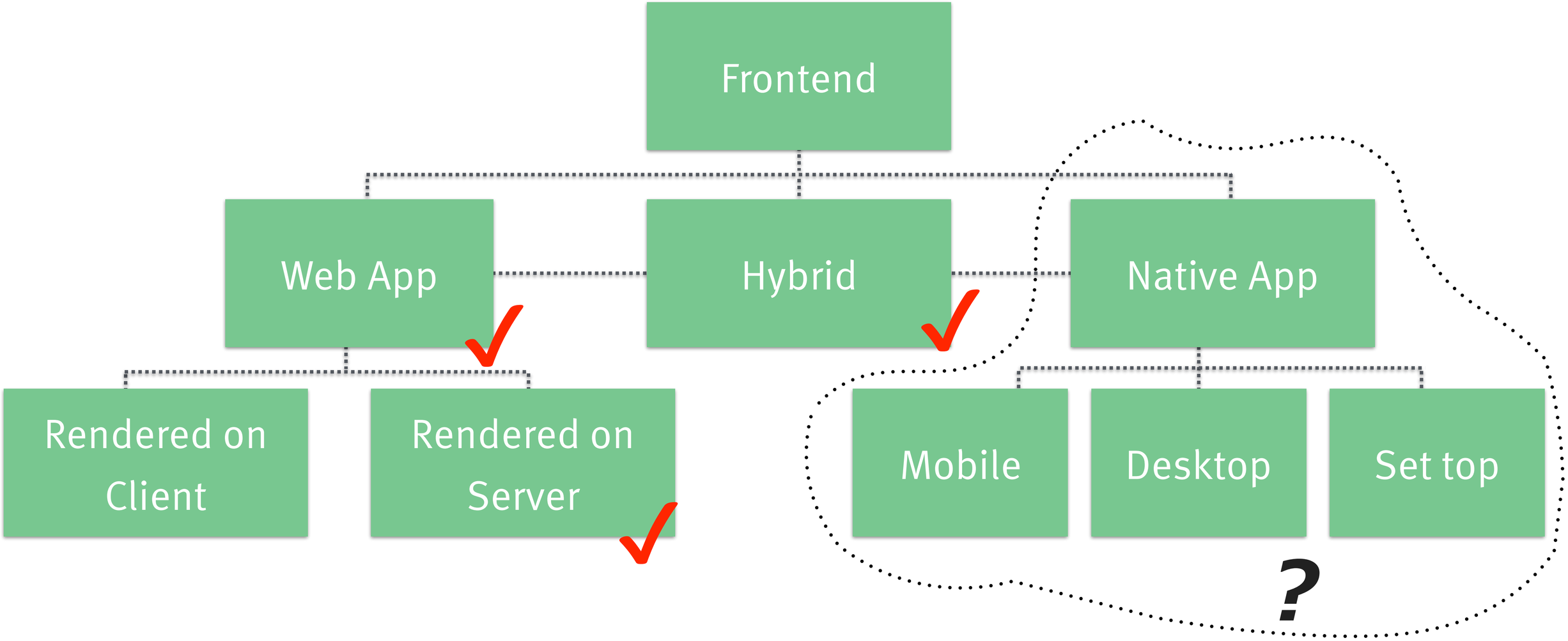
Small frontends, loosely coupled

Simple two-step secret to improving the performance of any website, according to Maciej Ceglowski (@baconmeteor):

"1. Make sure that the most important elements of the page download and render first.
2. Stop there."

http://idlewords.com/talks/website_obesity.htm

What about other approaches?



Assumption:

Frontend monoliths are OK

Sometimes.

Native frontends resemble server monoliths

Goals:

- As few assumptions as possible
- No implementation dependencies
- Small interface surface
- Based on standards
- Parallel development
- Independent deployment
- Autonomous operations

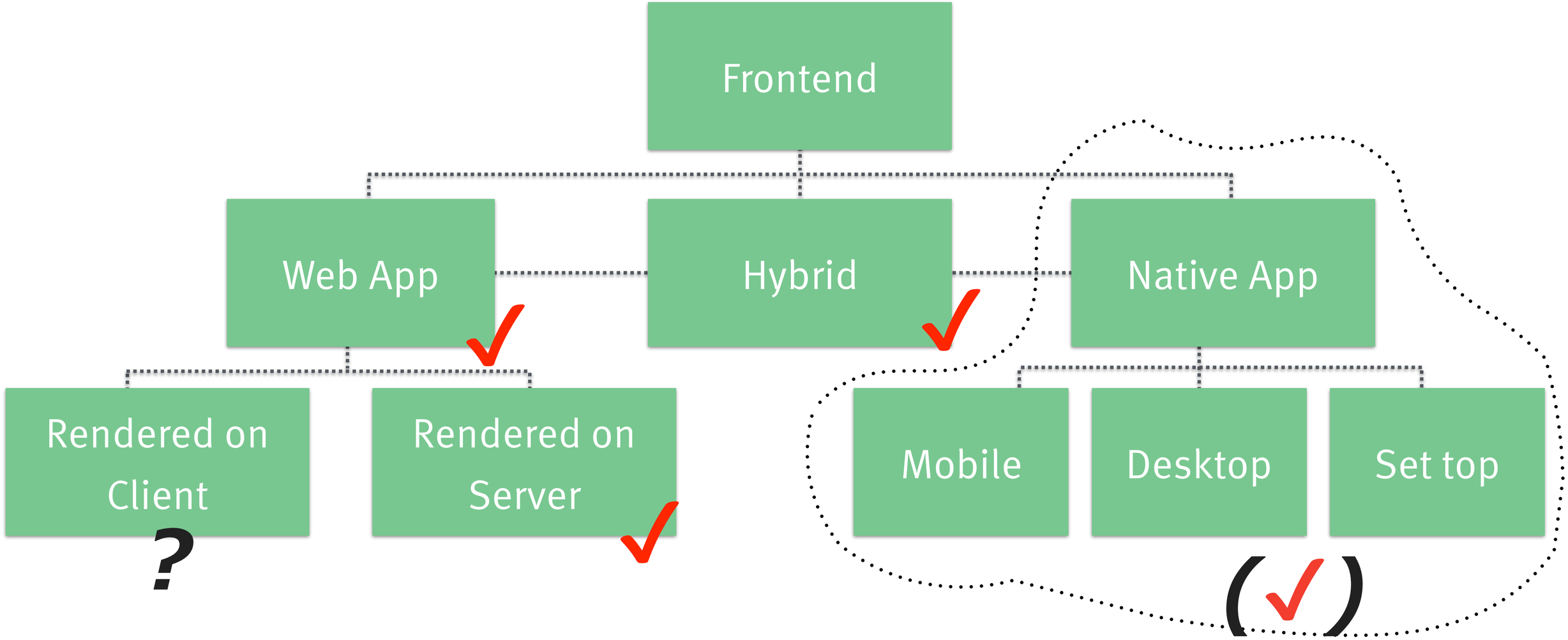
Constraint:

- Only internal modularization

Solution (sort of):

- Organizational structure
- Platform interfaces
- Release trains
- Discipline

What about other “modern” web apps?



Assumption:

JS-centric web apps can be *as good as* native apps

They shouldn't be as bad!

“Web service”[1]

- Use HTTP as transport
- Ignore verbs
- Ignores URIs
- Expose single “endpoint”
- Fails to embrace the Web

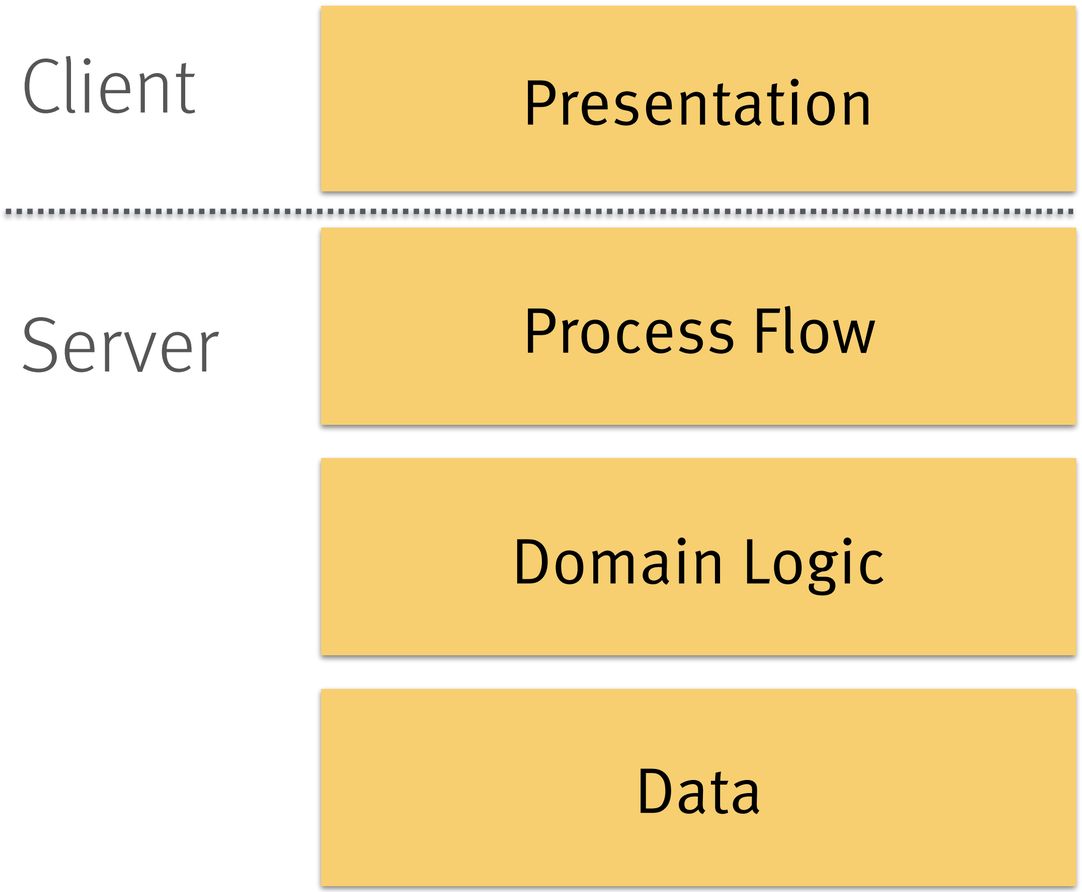
“Web app”[2]

- Uses browser as runtime
- Ignores forward, back, refresh
- Does not support linking
- Exposes monolithic “app”
- Fails to embrace the browser

[1] in the SOAP/WSDL sense

[2] built as a careless SPA

The web-native way of distributing logic

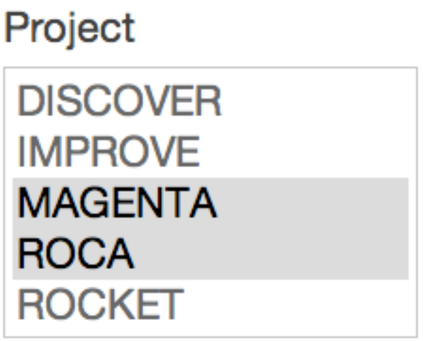


- Rendering, layout, styling on an *unknown* client
- Logic & state machine on server
- Client user-agent extensible via code on demand

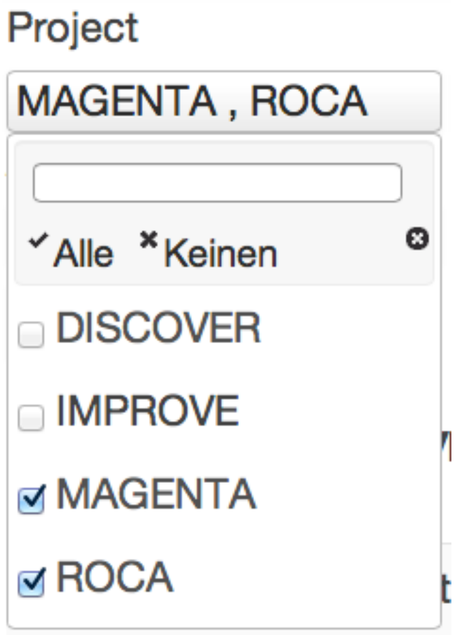
HTML & Hypermedia

- In REST, servers expose a hypermedia format
  - Option 1: Just use HTML
  - Option 2: Just invent your own JSON-based, incomplete clone
- Clients need to be RESTful, too
  - Option 1: Use the browser
  - Option 2: Invent your own, JS-based, buggy, incomplete implementation

```html
<div class="filter-column">
   <label for="project">Project</label>
   <select class="multiselect" id="project"
           name="project" size="5" multiple>
     <option>DISCOVER</option>
     <option>IMPROVE</option>
     <option >MAGENTA</option>
     <option>ROCA</option>
     <option>ROCKET</option>
   </select>
</div>
```




```javascript
$('.multiselect', context).each(function() {
        $(this).multiselect({
                selectedList: 2,
                checkAllText: "Alle",
                uncheckAllText: "Keinen"
        }).multiselectfilter({label:"",
                                width:"200px"});
});
```



Why choose a monolith if you don’t have to?

Any sufficiently complicated JavaScript client application contains an ad hoc, informally-specified, bug-ridden, slow implementation of half a browser.

(Me, with apologies to Phillip Greenspun)

ROCA: Resource-oriented Client Architecture

http://roca-style.org

If you're a fan of single page apps, at least build more than one

- Don't reinvent browser integration features
- Accept some inefficiency
- Trade-off for framework independence
- Avoid modularity à la Java EE, OSGi etc.

Disclaimer: Not a fan of SPAs (see https://medium.com/p/f08bb4ff9134

Summary

Few organizations are in the business of delivering APIs – UIs matter

Frontend monoliths are just as good, or bad, as backend monoliths

Nothing beats the browser with regards to modular frontend delivery

Thank you.

Questions?

Comments?

@stilkov

Stefan Tilkov
stefan.tilkov@innoq.com
Phone: +49 170 471 2625

Image credit: The Noun Project
Marek Polakovic, Arthur Shlain, Karthick Nagarajan