

Scalable Web Systems

Integration Part 2

Overview

1. DRY - Don't Repeat Yourself
2. Versioning
3. User Interfaces
 - a. Backends for Frontends

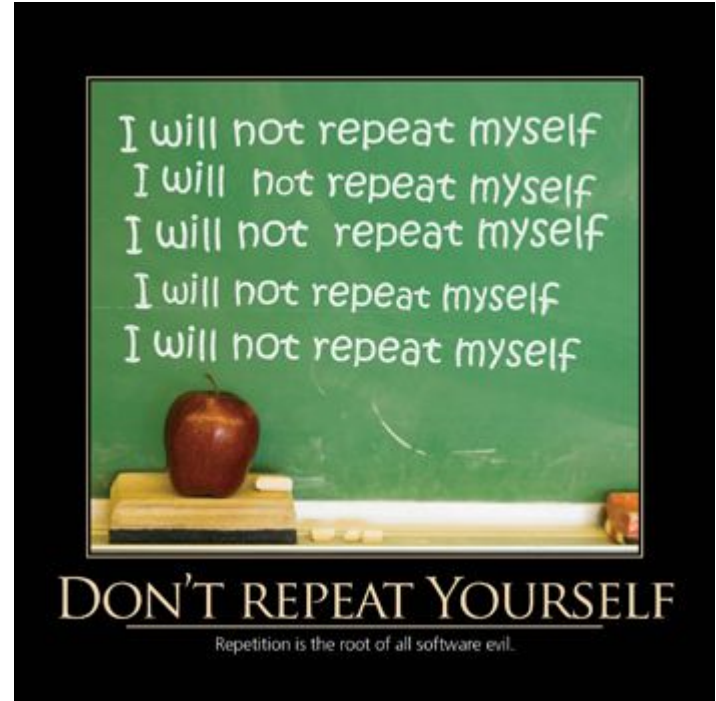
DRY and Microservices

Don't Repeat Yourself (DRY) is a concept that is ingrained in all programmers.

Code reuse is key - indeed, it is built in to every programming language with the idea of functions, modules, packages, and libraries!

However - you must be careful with DRY in a microservices world!

It can lead to tight coupling between services which does not scale well.



Client Library

What is a client library?

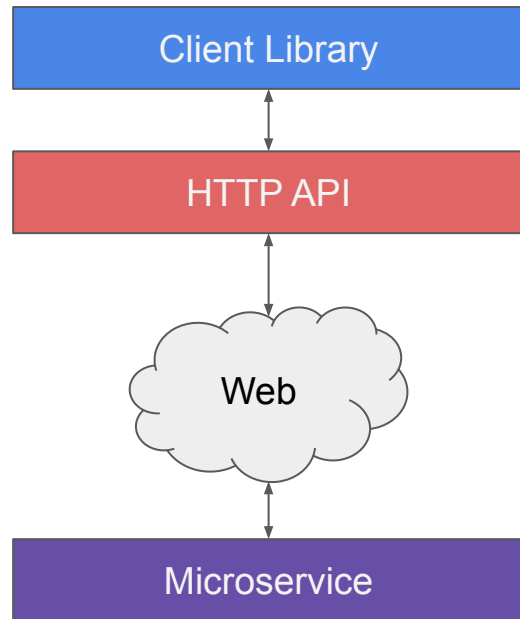
A client library is a library used by a consumer to access your service.

Typically supports multiple programming languages.

It can be used on top of the underlying API to handle service discovery, failure modes, logging, and other aspects that are not part of the service itself.

It can provide more robust client applications in that it is written by those who wrote the service.

Amazon AWS provides [client libraries](#) in a variety of languages and Netflix also supports client libraries.



Access by Reference

Another important aspect about integration is how “objects” or domain entities are referenced.

For example, if we have a customer service that provides functionality for all things about **Customers**.

When we retrieve a given **Customer** resource from the customer service, we get to see what that resource looked like *when we made the request*.

Often this is good enough, but what happens if the data is stale?



Access by Reference

If we need to most up-to-date information do we make a request to refresh? Do we only refresh part of the data or all of it?

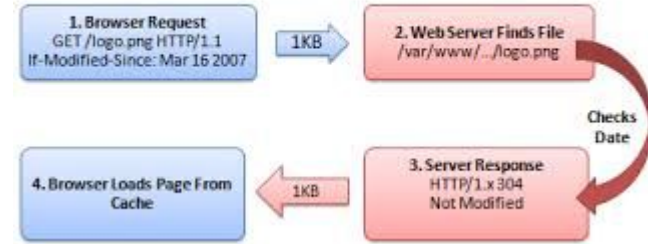
How do we even know when something is stale?

A refresh will lead to “chatter”, i.e., lots of additional HTTP traffic. This could clog up your services.

Can HTTP caching help us? We will look at this at a later point.

There isn't a hard-and-fast rule here, but be very wary of passing around data in requests when you don't know its freshness.

HTTP Cache: Last-Modified



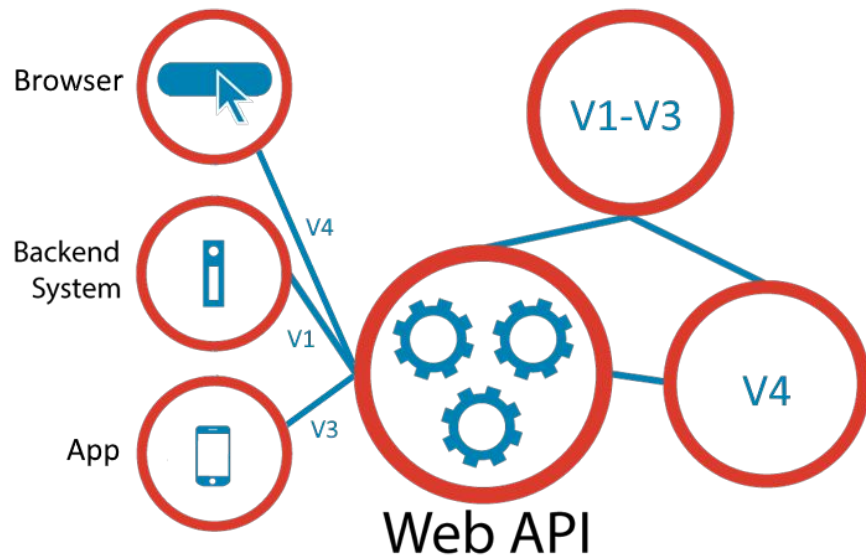
Versioning

Versioning an important part of integration.

If the API to a service changes we must be able to track those changes to publicize those changes to consumers and to provide a clear path for upgrade.

There are many ways to version a service or even individual API end-points in a service.

We may even want to support multiple versions of a service at the same time.



Defer It for as Long as Possible

The best way to reduce the impact of making breaking changes is to avoid making them in the first place.

Pick the right integration technology.

Database integration is a great example of technology that can make it very hard to avoid breaking changes.

REST, on the other hand, helps because changes to internal implementation detail are less likely to result in a change to the service interface.

```
<customer>
  <firstname>Sam</firstname>
  <lastname>Newman</lastname>
  <email>sam@magpiebrain.com</email>
  <telephoneNumber>555-1234-5678</telephoneNumber>
</customer>
```

V1

Did this break things?

```
<customer>
  <naming>
    <firstname>Sam</firstname>
    <lastname>Newman</lastname>
    <nickname>Magpiebrain</nickname>
    <fullname>Sam "Magpiebrain" Newman</fullname>
  </naming>
  <email>sam@magpiebrain.com</email>
</customer>
```

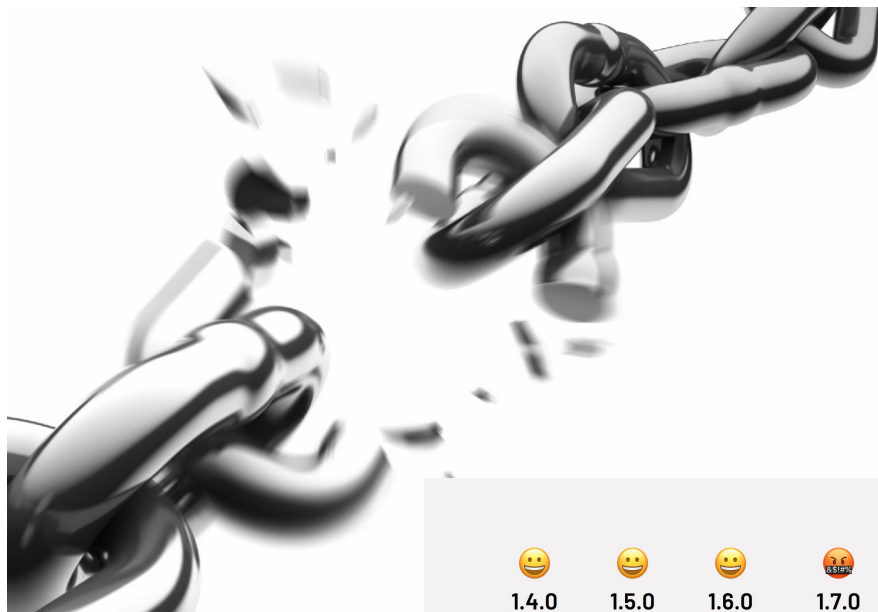
V2

Catch Breaking Changes Early

It's crucial to make sure we pick up changes that will break consumers as soon as possible, because even if we choose the best possible technology, breaks can still happen.

If you're supporting multiple different client libraries, running tests using each library you support against the latest service is another technique that can help.

Once you realize you are going to break a consumer, you have the choice to either try to avoid the break altogether or else embrace it and start having the right conversations with the people looking after the consuming services.



1.4.0



1.5.0



1.6.0



1.7.0



Use Semantic Versioning

1.3.5

X.Y.Z


Use Semantic Versioning

1.3.5

X.Y.Z

Major Version

Incremented whenever
major changes are
made like architectural
change.



Use Semantic Versioning

1.3.5

X.Y.Z

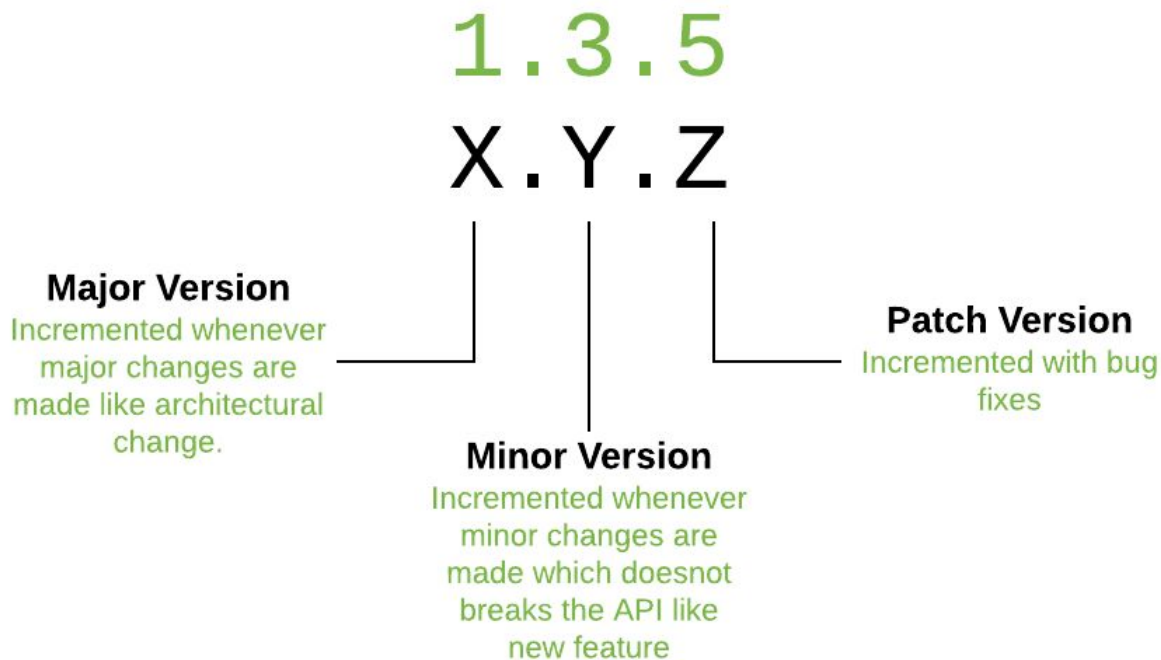
Major Version

Incremented whenever
major changes are
made like architectural
change.

Minor Version

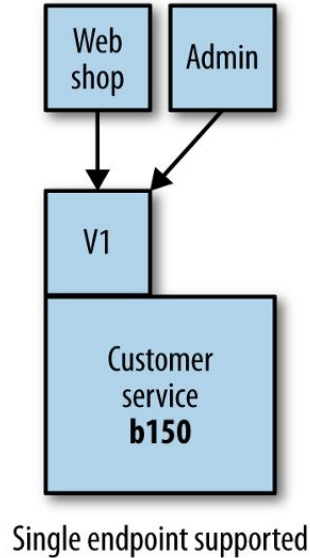
Incremented whenever
minor changes are
made which doesnot
breaks the API like
new feature

Use Semantic Versioning



Coexist Different Endpoints

At some point
versions will
change....

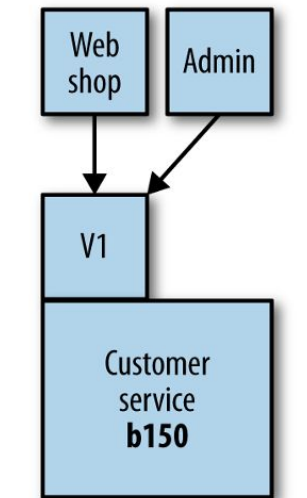


Coexist Different Endpoints

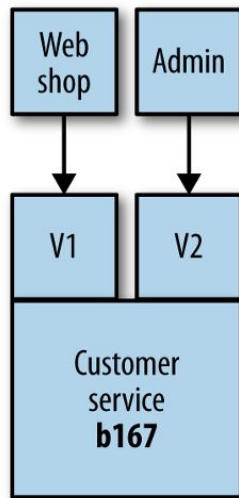
At some point versions will change....

Avoid forcing consumers to upgrade in lock-step with.

One way to do this is to co-exist both the old and new interfaces in the same running service.



Single endpoint supported



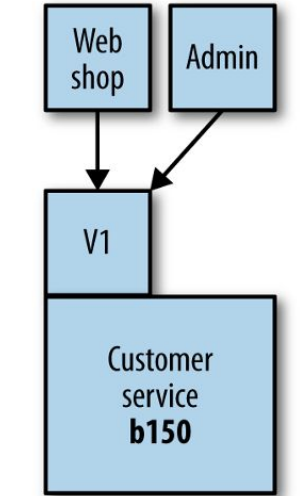
New release exposes and additional endpoint

Coexist Different Endpoints

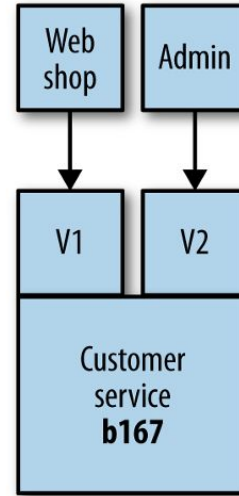
At some point versions will change....

Avoid forcing consumers to upgrade in lock-step with.

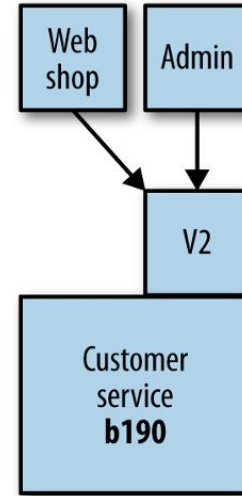
One way to do this is to co-exist both the old and new interfaces in the same running service.



Single endpoint supported



New release exposes and additional endpoint



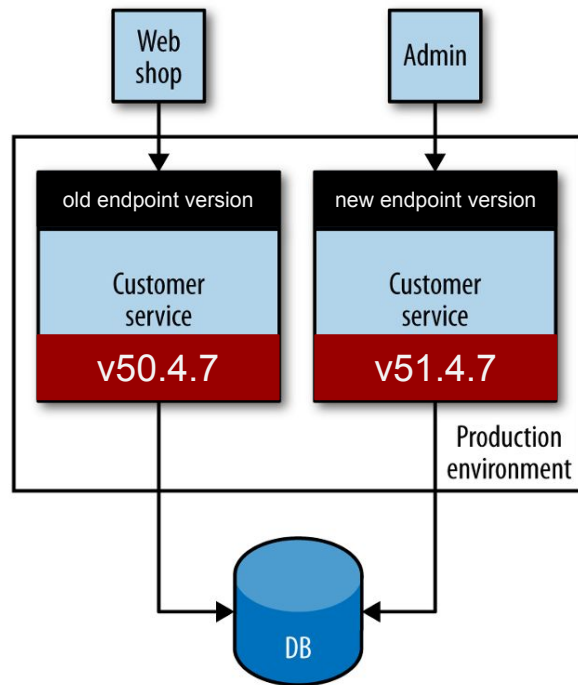
Once the old endpoint is no longer used, a new release of the service can remove it

Use Multiple Concurrent Service Versions

Another versioning solution often cited is to have different versions of the service live at once, and for older consumers to route their traffic to the older version, with newer versions seeing the new one.

Problems:

1. If there is a bug, we need to fix it in multiple services.
2. Need to build the logic to direct consumers to the right service.
3. Shared database is only asking for trouble.



User Interfaces

“A few of you out there might just be providing a cold, hard, clinical API to your customers, but many of us find ourselves wanting to create beautiful, functional user interfaces that will delight our customers.”

But we really do need to think about them in the context of integration.

The user interface, after all, is where we'll be pulling all these microservices together into something that makes sense to customers.

Over the last couple of years, organizations have started to move away from thinking that web or mobile should be treated differently.

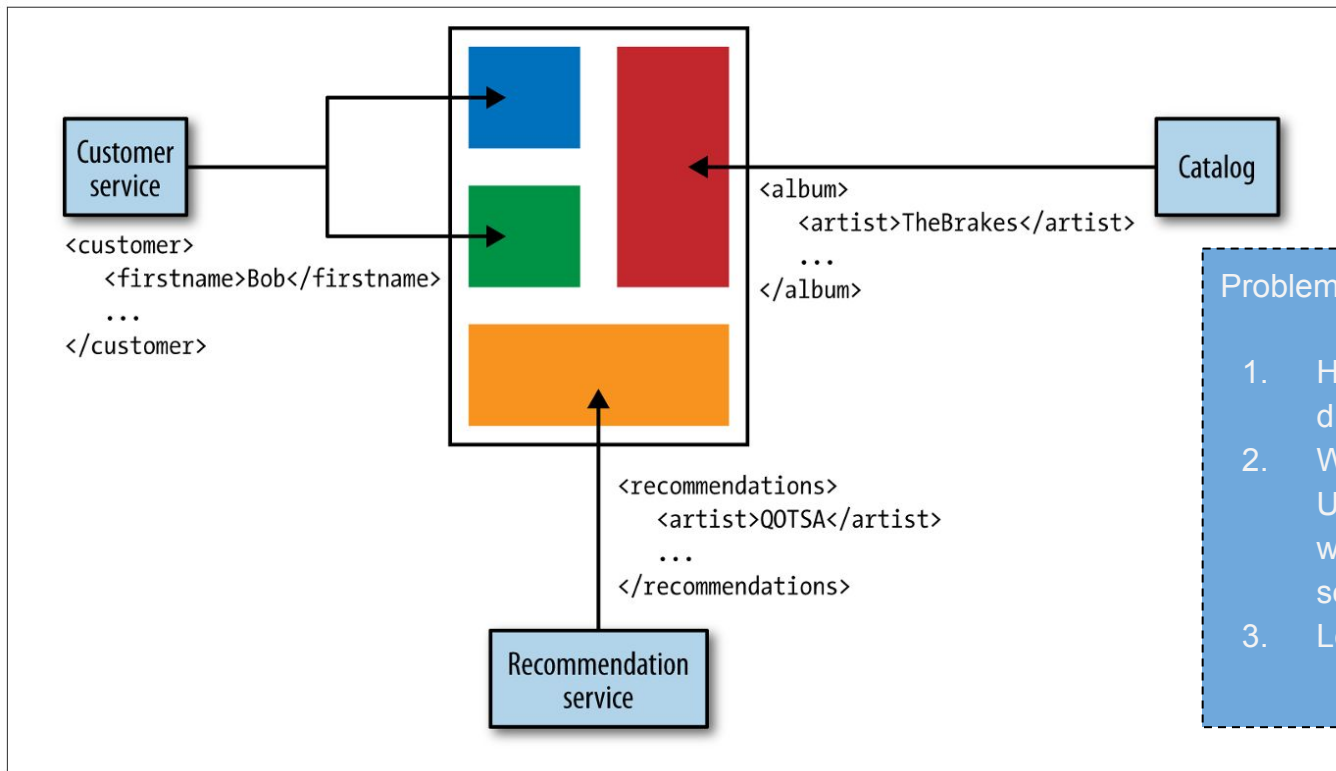
They are instead thinking about digital more holistically.

What is the best way for customers to use the services offered?

What does that do to our system architecture?

The understanding that we cannot predict exactly how a customer might end up interacting with a company has driven adoption of **more granular** APIs, *like those delivered by microservices*.

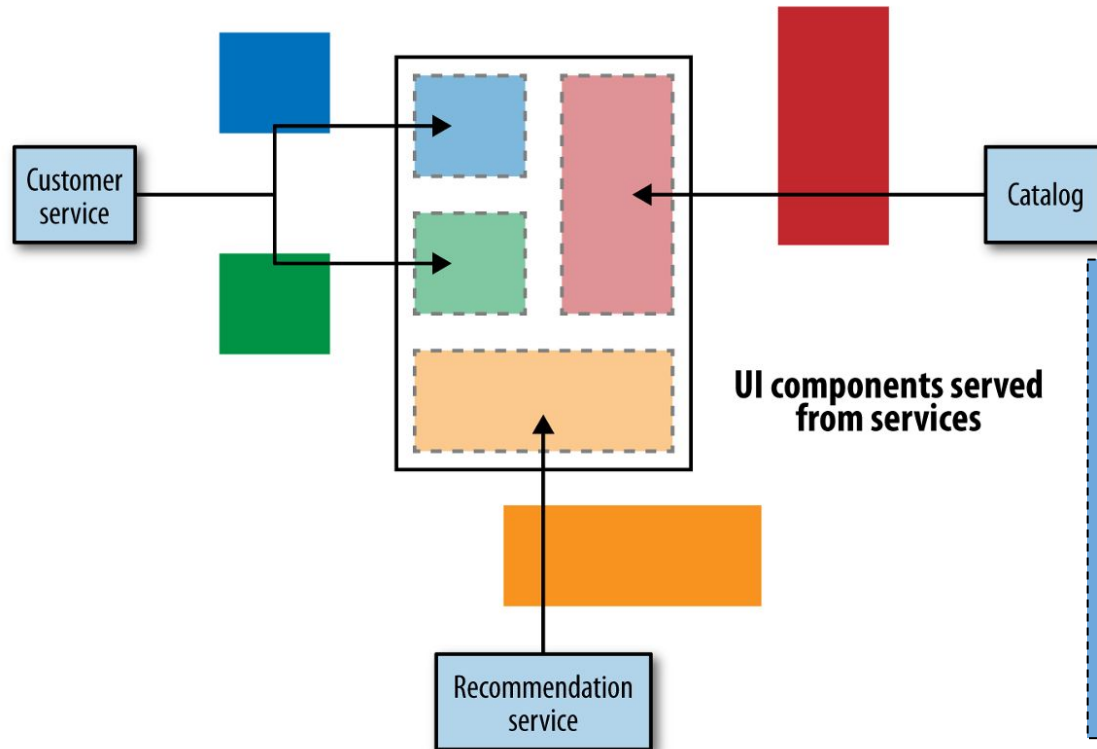
Using Multiple APIs to Compose an Interface



Problems?

1. Hard to tailor the response to different devices.
2. Who is responsible for creating the UI? Is it the responsibility of those who write the service or is there a separate UI team?
3. Lots of HTTP traffic

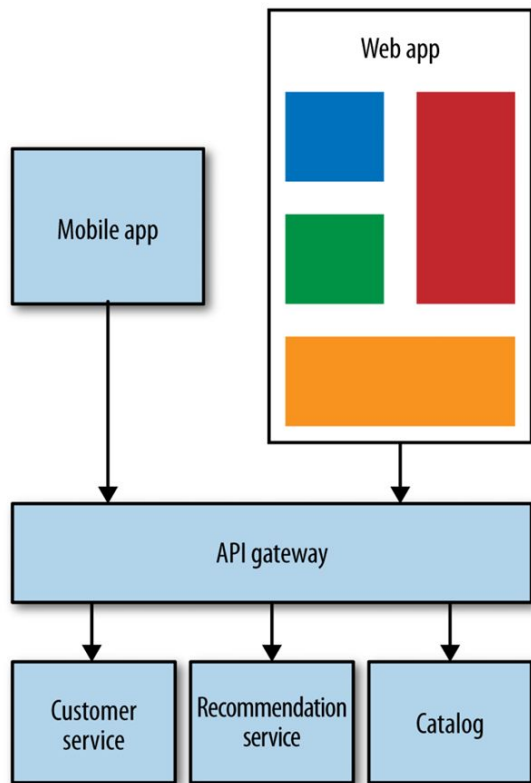
UI Fragment Composition



Problems?

1. Consistency in user experience.
2. How do native applications work with this?
3. What if we need dynamic data (search list) - we still need to fall back to using API calls, so why bother at all?

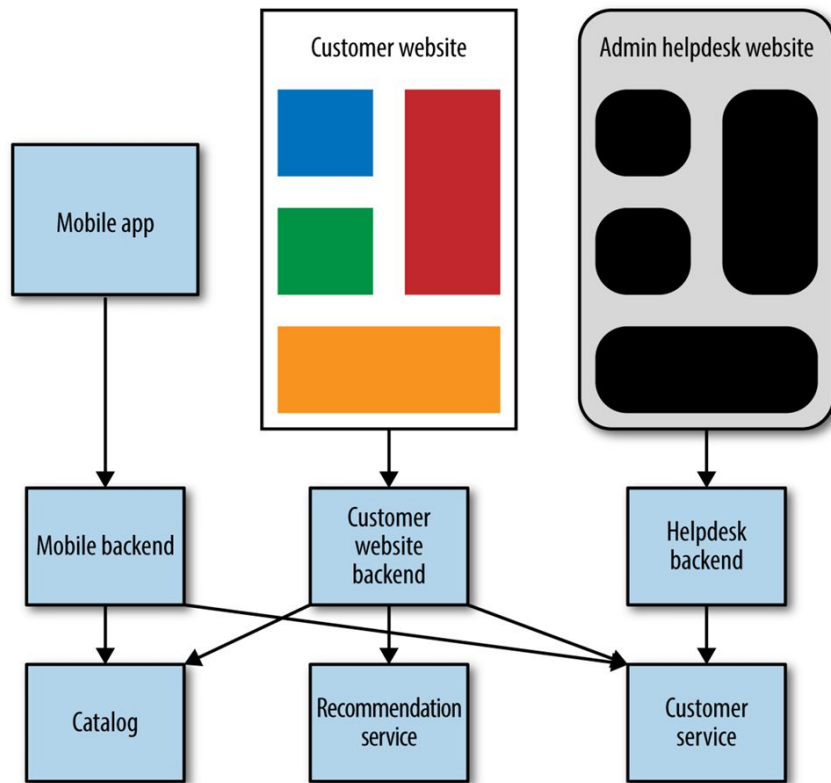
UI API Gateway



A common solution to the problem of chatty interfaces with backend services, or the need to vary content for different types of devices, is to have a server-side aggregation endpoint, or API gateway.

This can marshal multiple backend calls, vary and aggregate content if needed for different devices, and serve it up.

Backends For Frontends (BFF)



Restrict the use of these backends to one specific user interface or application.

This pattern is sometimes referred to as backends for frontends (BFFs).

It allows the team focusing on any given UI to also handle its own server-side components.

You can see these backends as parts of the user interface that happen to be embedded in the server.

Some types of UI may need a minimal server-side footprint, while others may need a lot more.