Pattern: API Gateway / Backends for Frontends

Context

Let's imagine you are building an online store that uses the Microservice architecture pattern (microservices.html) and that you are implementing the product details page. You need to develop multiple versions of the product details user interface:

- HTML5/JavaScript-based UI for desktop and mobile browsers - HTML is generated by a server-side web application
- Native Android and iPhone clients - these clients interact with the server via REST APIs

In addition, the online store must expose product details via a REST API for use by 3rd party applications.

A product details UI can display a lot of information about a product. For example, the Amazon.com (http://Amazon.com) details page for POJOs in Action (http://www.amazon.com/POJOs-Action-Developing-Applications-Lightweight/dp/1932394583) displays:

- Basic information about the book such as title, author, price, etc.
- Your purchase history for the book
- Availability
- Buying options
- Other items that are frequently bought with this book
- Other items bought by customers who bought this book
- Customer reviews
- Sellers ranking
- …

Since the online store uses the Microservice architecture pattern the product details data is spread over multiple services. For example,

- Product Info Service - basic information about the product such as title, author
- Pricing Service - product price
- Order service - purchase history for product
- Inventory service - product availability
- Review service - customer reviews …

Consequently, the code that displays the product details needs to fetch information from all of these services.

Problem

How do the clients of a Microservices-based application access the individual services?

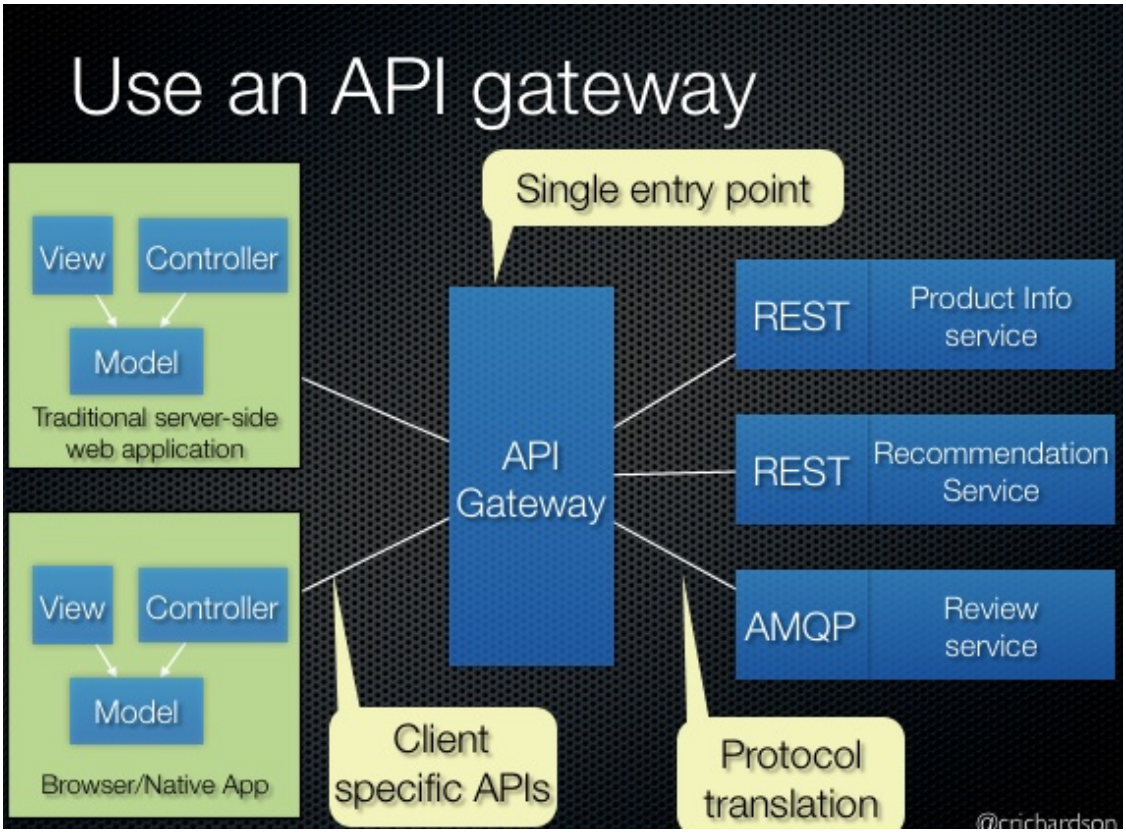
Forces

- The granularity of APIs provided by microservices is often different than what a client needs. Microservices typically provide fine-grained APIs, which means that clients need to interact with multiple services. For example, as described above, a client needing the details for a product needs to fetch data from numerous services.
- Different clients need different data. For example, the desktop browser version of a product details page desktop is typically more elaborate then the mobile version.

- Network performance is different for different types of clients. For example, a mobile network is typically much slower and has much higher latency than a non-mobile network. And, of course, any WAN is much slower than a LAN. This means that a native mobile client uses a network that has very difference performance characteristics than a LAN used by a server-side web application. The server-side web application can make multiple requests to backend services without impacting the user experience where as a mobile client can only make a few.
- The number of service instances and their locations (host+port) changes dynamically
- Partitioning into services can change over time and should be hidden from clients
- Services might use a diverse set of protocols, some of which might not be web friendly

Solution

Implement an API gateway that is the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services.

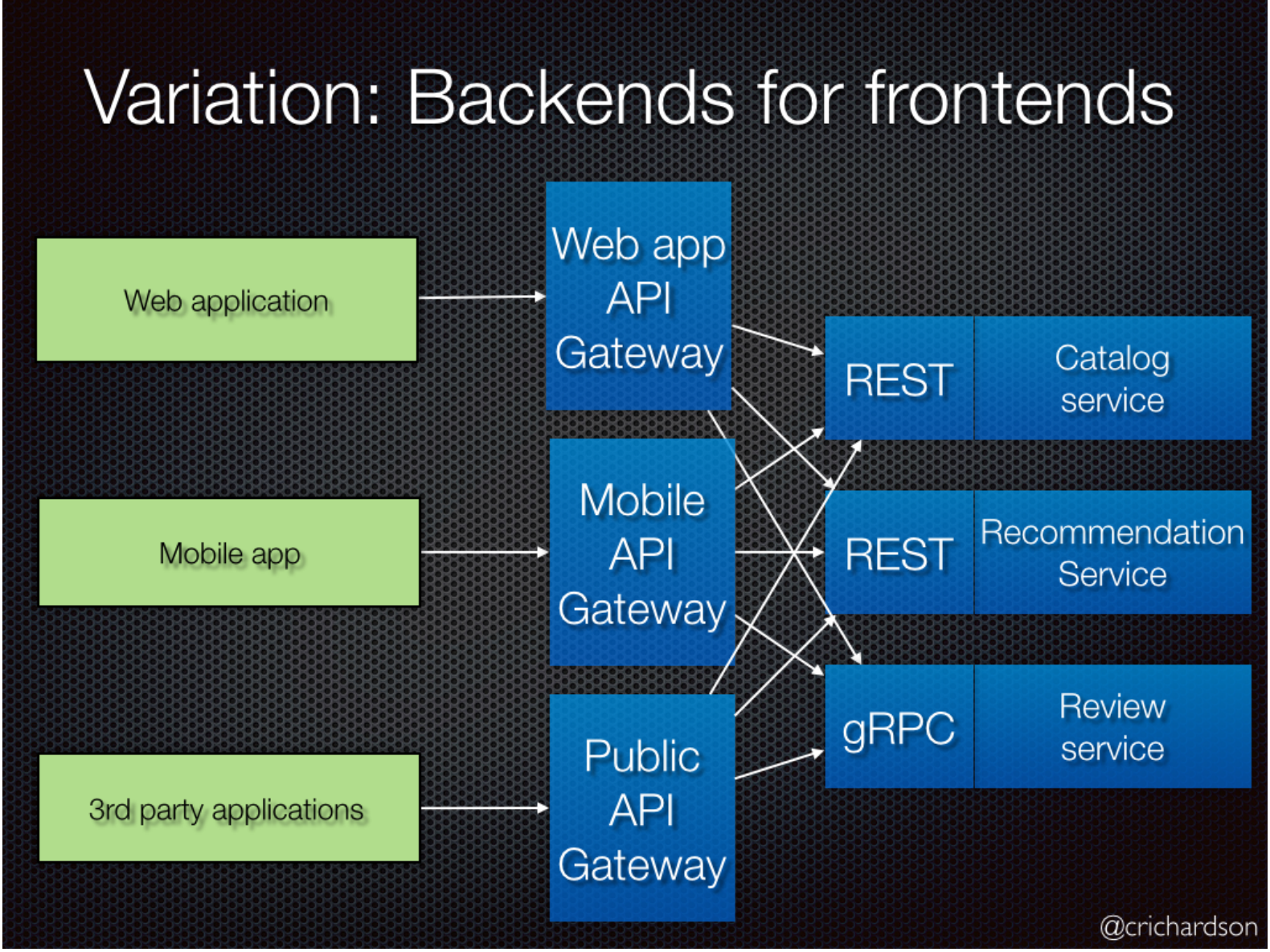


Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client. For example, the Netflix API (http://techblog.netflix.com/2012/07/embracing-differences-inside-netflix.html) gateway runs client-specific adapter code that provides each client with an API that's best suited to its requirements.

The API gateway might also implement security, e.g. verify that the client is authorized to perform the request

Variation: Backends for frontends

A variation of this pattern is the Backends for frontends pattern. It defines a separate API gateway for each kind of client.



In this example, there are three kinds of clients: web application, mobile application, and external 3rd party application. There are three different API gateways. Each one is provides an API for its client.

Examples

- Netflix API gateway (http://techblog.netflix.com/2013/01/optimizing-netflix-api.html)
- A simple Java/Spring API gateway (https://github.com/cer/event-sourcing-examples/tree/master/java-spring/api-gateway-service) from the Money Transfer example application (https://github.com/cer/event-sourcing-examples).

Resulting context

Using an API gateway has the following benefits:

- Insulates the clients from how the application is partitioned into microservices
- Insulates the clients from the problem of determining the locations of service instances
- Provides the optimal API for each client
- Reduces the number of requests/roundtrips. For example, the API gateway enables clients to retrieve data from multiple services with a single round-trip. Fewer requests also means less overhead and improves the user experience. An API gateway is essential for mobile applications.
- Simplifies the client by moving logic for calling multiple services from the client to API gateway
- Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally

The API gateway pattern has some drawbacks:

- Increased complexity - the API gateway is yet another moving part that must be developed, deployed and managed
- Increased response time due to the additional network hop through the API gateway - however, for most applications the cost of an extra roundtrip is insignificant.

Issues:

- How implement the API gateway? An event-driven/reactive approach is best if it must scale to scale to handle high loads. On the JVM, NIO-based libraries such as Netty, Spring Reactor, etc. make sense. NodeJS is another option.

Related patterns

- The Microservice architecture pattern (microservices.html) creates the need for this pattern.
- The API gateway must use either the Client-side Discovery pattern (client-side-discovery.html) or Server-side Discovery pattern (server-side-discovery.html) to route requests to available service instances.
- The API Gateway may authenticate the user and pass an Access Token (security/access-token.html) containing information about the user to the services
- An API Gateway will use a Circuit Breaker (reliability/circuit-breaker.html) to invoke services
- An API gateway often implements the API Composition pattern (/patterns/data/api-composition.html)

Known uses

- Netflix API gateway (http://techblog.netflix.com/2012/07/embracing-differences-inside-netflix.html)

Example application

See the API Gateway that part of my Microservices pattern's example application (https://github.com/microservice-patterns/ftgo-application). It's implemented using Spring Cloud Gateway.