Implementation Patterns for Microservices Architectures

KYLE BROWN, IBM Corporation
BOBBY WOOLF, IBM Corporation

Abstract In this paper we describe a set of implementation patterns for building applications using microservices. We discuss the application types and requirements that lead to the n eed for microservices, examine different types of microservices, and discuss patterns required for implementing data storage and devops in a microservices environment.

Categories and Subject Descriptors: • **Software and its engineering~Software design engineering**

General Terms: Software Architectures

Additional Key Words and Phrases: Microservices, Agile Development, Pattern Languages

ACM Reference Format:

Brown, K. and Woolf, B., 2016. Implementation Patterns of Microservices Architectures. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2016), 35 pages.

1. INTRODUCTION: PURPOSE OF THE PATTERN LANGUAGE

The microservices architecture is one of the most rapidly expanding architectural paradigms in commercial computing today. Since its introduction in a white paper by Martin Fowler and James Lewis [Fowler] it has since become a de-facto standard for developing large-scale commercial applications.

But despite the clear descriptions of the principles of this architecture in the Fowler whitepaper, and also in later works such as Newman's that elaborate on the architecture, many development teams still struggle with basic practical questions about how to implement systems using the microservices architecture. In particular, they struggle with questions of how the microservices are invoked from a client application, how different client application styles affect their microservices implementation, and, most important, how to build their microservices efficiently.

It is that last point that is perhaps the most troublesome in practical microservices implementations. A benefit and a drawback of Services Oriented Architectures (including the microservices approach) is that services are implemented as simple HTTP endpoints – but the simplicity and power of HTTP is sometimes outweighed by the latency inherent in process-to-process communication using HTTP. Overcoming that latency requires forethought and planning in your architecture in order to reduce unnecessary overhead.

Another issue with microservices architectures is that while they make it possible for developers to have more choices – for instance, by allowing for both Polyglot programming and Polyglot persistence -- in fact the set of choices facing developers is too large – with too few guidelines in place for helping navigate. The result is that developers find it hard to decide where to start, and find few good examples of template applications of different styles that use microservices effectively. In this paper, we will help address a few of those issues by introducing a set of simple patterns that can help developers understand how to apply microservices in a variety of different situations.

This paper is intended for use by Architects, Lead Developers or Senior Developers who are thinking about adopting the microservices architecture in their projects. It is intended to provide guidance on how microservices should be designed, how they fit into a larger architectural picture, and how they can be built to operate efficiently. While it specifically addresses issues of systems design, microservices design and microservices efficiency, it does not cover issues of security or development process. Those nonfunctional requirements will have to be covered in later papers.

1.1 What is a Microservice?

Microservices is an application architectural style in which an application is composed of many discrete, network-connected components, termed *microservices*. The microservices architectural style can be seen as an evolution of the SOA (Services Oriented Architecture) architectural style. The key differences between the two are that while applications built using SOA services tended to become focused on technical integration issues, and the level of the services implemented were often very fine-grained technical APIs, the microservices approach instead stay focused on implementing clear business capabilities through larger-grained business APIs. An applicable rule in this case is that any microservice API should be directly related to a business entity.

But aside from the service design questions, perhaps the biggest difference is one of deployment style. For many years, applications have been packaged in a monolithic fashion – that is a team of developers would construct one large application that does everything required for a business need. Once built, that application would then be deployed multiple times across a farm of application servers. By contrast, in the microservices architectural style, several smaller applications that each implement only part of the whole are built and packaged independently.

Microservices have applicability in both new application development and in refactoring. While many teams prefer to apply the microservices approach in new or "greenfield" development, the approach can be applied stepwise to existing applications as part of an application refactoring effort. In cases like this, approaches such as Fowler's "Strangler Application" [Fowler2] pattern apply, although that lies outside of the scope of this paper.

When you look at the implications of packaging services independently, you see that five simple rules drive the implementation of applications built using the microservices architecture. They are:

- **Deploy applications as sets of small, independent services** -- A single network-accessible service is the smallest deployable unit for a microservices application. Each service should run in its own process. This rule is sometimes stated as "one service per container", where "container" could mean a Docker container or any other lightweight deployment mechanism such as a Cloud Foundry runtime, or even a Tomcat or WebSphere Liberty server.
- **Optimize services for a single function** – In a traditional monolithic SOA approach a single application runtime would perform multiple business functions. In a microservices approach, there should be one and only one business function per service. This makes

each service smaller and simpler to write and maintain. Robert Martin [Martin] calls this the "Single Responsibility Principle".

- **Communicate via REST API and message brokers** – One of the drawbacks of the SOA approach was that there was an explosion of standards and options for implementing SOA services. The microservices approach instead tries to strictly limit the types of network connectivity that a service can implement to achieve maximum simplicity. Likewise, another rule for microservices is to avoid the tight coupling introduced by implicit communication through a database – all communication from service to service must be through the service API or at least must use an explicit communication pattern such as the *Claim Check* Pattern from [Hohpe].
- **Apply Per-service CI/CD** -- When building a large application comprised of many services, you soon realize that different services evolve at different rates. Letting each service have its own unique Continuous Integration/Continuous Delivery pipeline allows that evolution to proceed at is own natural pace – unlike in the monolithic approach where different aspects of the system were forced to all be released at the speed of the slowest-moving part of the system.
- **Apply Per-service HA/clustering decisions** – Another realization when building large systems is that when it comes to clustering, not one size fits all. The monolithic approach of scaling all the services in the monolith at the same level often led to overutilization of some servers and underutilization of others – or even worse, starvation of some services by others when they monopolized all of the available shared resources such as thread pools. The reality is that in a large system, not all services need to scale and can be deployed in a minimum number of servers to conserve resources. Others require scaling up to very large numbers.

The power of the combination of these points (each of which will be referenced in the patterns below) and the benefits obtained from following them is the primary reason why the microservices architecture has become so popular.

As an example of this approach, let's look at a very simplified view of an airline website. In traditional technologies, the only way to deploy an application would be as a monolith that is scaled to the maximum extent of any possible request – so if booking drove the most traffic to your website, the entire application must be deployed as many times as you need just to handle the booking traffic (see Figure 1: Traditional Monolithic deployment).

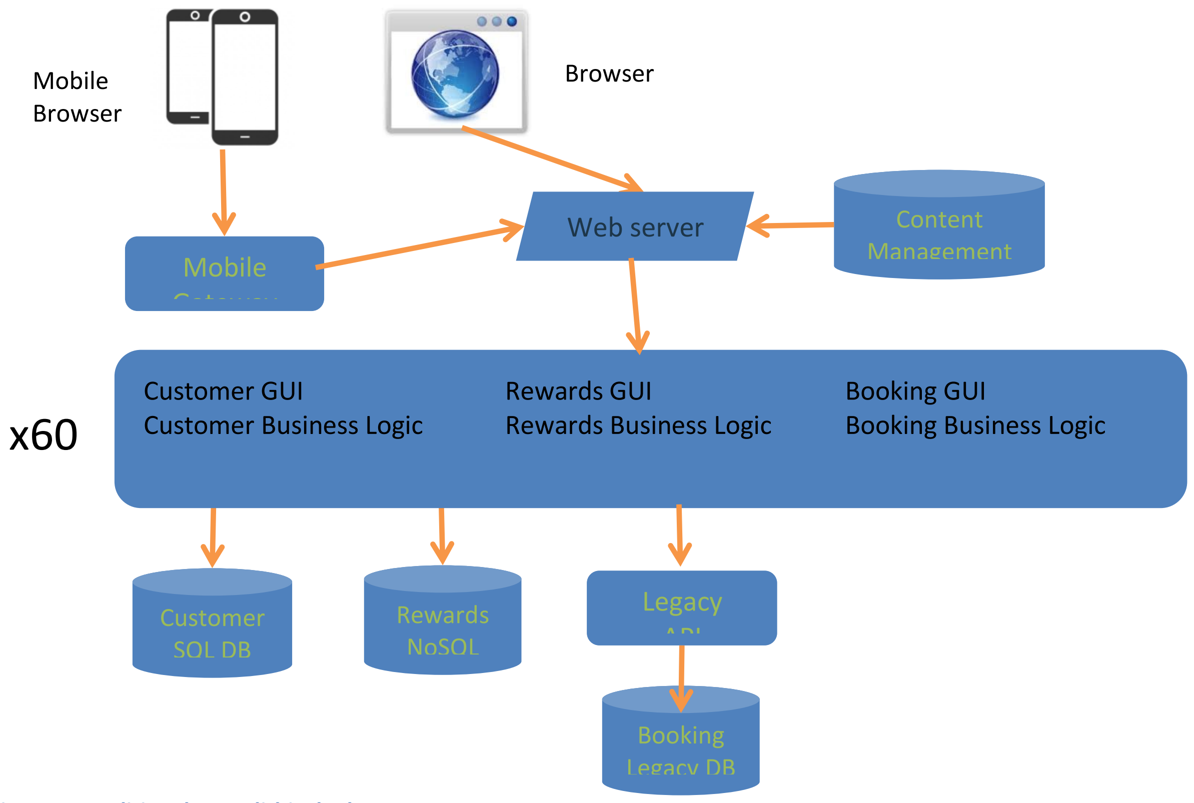

Figure 1: Traditional Monolithic deployment

However, in a microservices approach, each individual business purpose; Booking Flights, Customer management, the Rewards program, etc., is broken down into separate microservices that can be deployed and managed independently. What's more, you need only deploy as many copies of each service as is actually required to handle the traffic for that specific business area. So, if Loyalty is only used by 10% of your customers, you may be able to deploy a drastically smaller number of servers for that business function than you would for flight booking. An example of this kind of architecture is shown below (see Figure 2: Airline application as microservices).

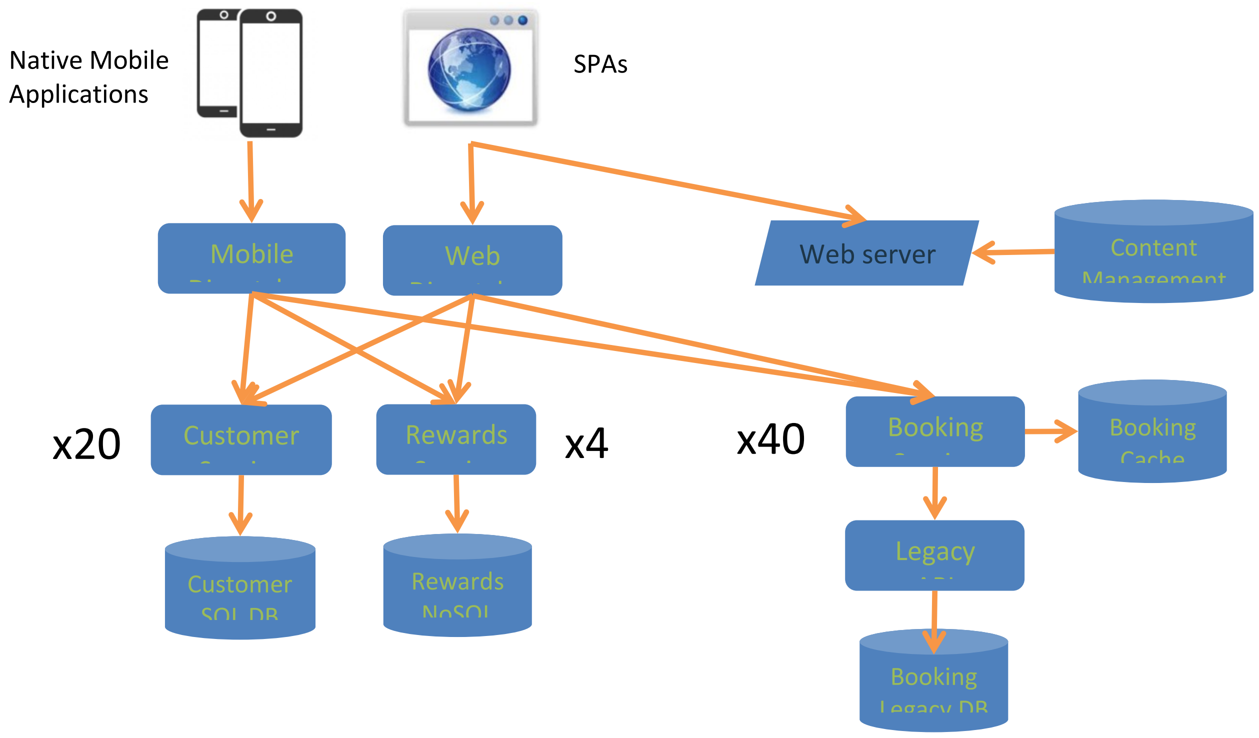

Figure 2: Airline application as microservices

1.2 How this pattern language is constructed

This pattern language is split into four parts; the first part, *Modern Web Architecture Patterns*, refers to those patterns that are helpful in understanding how the front end of an application built using a microservices Architecture can be effectively implemented. The second part, *Microservice Architecture Patterns*, refers to patterns specifically related to building Microservices and making them run efficiently. The next part, *Scalable Store Patterns*, relate to decisions about data storage that are necessary to build systems that are both responsive and highly available. The final part, *Microservices DevOps Patterns* discusses decisions around managing and debugging microservices applications.

The patterns are written in a simplified POSA[1] style with explicit sections for Context, Problem, Forces, Solution and Results.

One of the lessons that we learned when writing *Enterprise Integration Patterns* [Hohpe] is that when you are working in large problem domains there is often the need for a Root Pattern to introduce the overall problem domain. That allows you to set the larger context of your domain without directly jumping into the details. Since our pattern language has four separate sections that each describe a different aspect of the microservices approach, we need four root patterns. In our pattern language we introduce each section with one of these four root patterns: **Modern Web Architecture**, **Microservices Architecture**, **Scalable Store** and **Microservices DevOps**.

[1] e.g. following the pattern style used in *Pattern Oriented System Architecture* [Buschmann]

1.3 Pattern Map

The following pattern map (Figure 3: Pattern Map) shows the split of the patterns into the four sections described above, as well as the connections between related patterns.

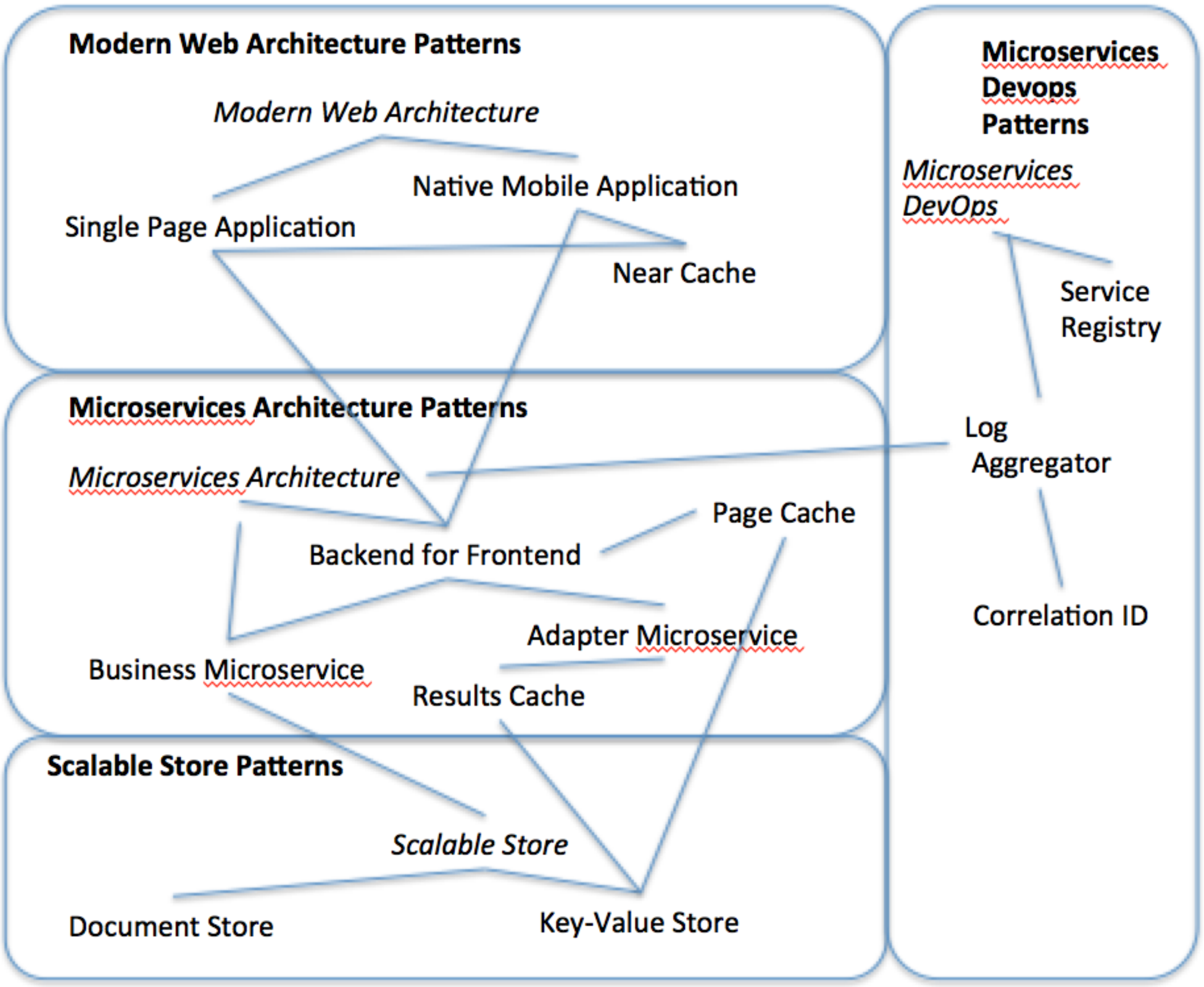

Figure 3: Pattern Map

2. MODERN WEB ARCHITECTURE PATTERNS

As described in the previous section, the Modern Web Architecture Patterns begin with the root pattern **Modern Web Architecture**. Adopting a Modern Web Architecture will lead you to different potential implementation choices such as a **Single Page Application** or a **Native Mobile Application**. A **Near Cache** is commonly implemented in Modern Web Applications in order to improve performance and to allow for some functioning when the client is disconnected from back-end systems.

2.1 Modern Web Architecture

Title: Modern Web Architecture

Context:

You are building a new application or refactoring an existing application that needs to provide a rich user interface that can be usable on many different devices such as smartphones, tablets, or laptops.

While the dynamic page approach is perhaps the most common application style on the Web, dating back to technologies like Active Server Pages (ASP), Java Server Pages (JSP), and PHP, the major drawback of this style is that the page-at-a-time semantics of these systems limit the type of interfaces that can be effectively developed with them. It is difficult if not impossible to write highly interactive user interfaces that take advantage of the user interface capabilities of mobile devices, tablets and touchscreens using only the capabilities of HTML.

Problem:

How do you design an application that meets the needs of your users, while still working within the capabilities of the different interface devices that now exist?

Forces:

- Web developers want to be able to take advantage of the capabilities of the JavaScript engines built into modern web browsers.
- Modern Devices have more user-interaction capabilities than previous generations of devices that only supported mice and keyboards
- Users expect more from their applications and websites today than they did when the web was first introduced.

Solution:

Adopt a *Modern Web Architecture* that combines intelligent front-end components with general-purpose back-end components. In contrast to the page-at-a-time dynamic page solutions that were common in the early days of Browser technology, with Modern Web Architecture there is no longer a need to have an explicit one-to-one map between a web page and a back-end server call.

Picture:

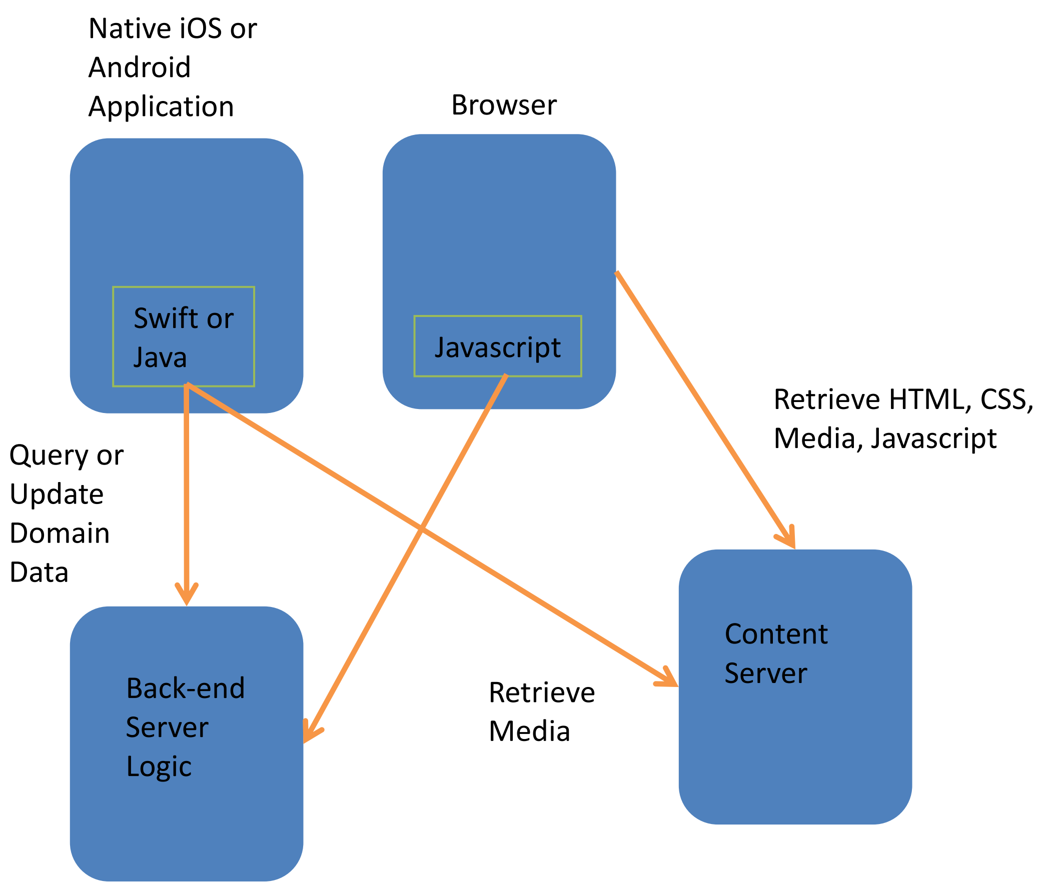
Native iOS or Android Application
Browser
Swift or Java
Javascript
Retrieve HTML, CSS, Media, Javascript
Query or Update Domain Data
Content Server
Back-end Server Logic
Retrieve Media

Figure 4: Modern Web Architecture Components

In the Modern Web Architecture (see Figure 4: Modern Web Architecture Components) you see that there is a division between the front-end application (written in Javascript for a Single Page Application, or written in Swift or Java for iOS or Android Native Mobile Applications) and the back-end Server logic. Essentially, this corresponds to a division between the "V" in a Model-View-Controller (MVC) Architecture, which can now remain entirely on the client side, and the "M" in the MVC, which can now remain mostly on the server side. Another aspect of the Modern Web Architecture shown here is that most application now rely on a separate content server to provide ephemeral aspects of the GUI (e.g. graphics, explanatory text, or even style sheets).

Result:

The Modern Web Architecture takes advantage of the fact that nearly all Browsers now support Javascript as a client programming language. This allows you to make multiple back-end server calls within the context of a single logical page. But the overall approach does not end at the Browser. You can build intelligent front-ends either as *Native Mobile Applications* or *Single Page Applications* that interface with more general-purpose back-end services representing the services within a business domain, e.g. *Business Microservices*. By assuming that you can execute programs at the client end then you can structure your architecture much more around serving data to an intelligent front-end application than around a specific user interface design. That allows the front-end and the back-end to evolve at different rates. To facilitate this evolution, you can use

Backends for Frontends where needed to adapt between the needs of specific Native Mobile Applications, Single Page Applications and Business microservices. *Near Caches* can improve the performance of your intelligent front-ends.

2.2 Single Page Application

Title: Single Page Application

Context:

You are developing a new web application or you are refactoring a section of a web application to make it more modern.

Problem: How do you design the front end of your application to take advantage of the programmability of modern browsers and provide the best mix of client responsiveness and server optimization?

Forces:

- You want to provide the user with a very responsive application that they can interact with easily despite network lags.
- You want the user to be able to interact with the application naturally, without arbitrary restrictions in user interface design.
- You want to enable your designers to provide the cleanest, most attractive user interface possible, and to be able to make design modifications without requiring them to work through your development team.
- You want to allow your application developers to focus on application development issues and not have to inordinately concern themselves with the minutiae of user interface component layout and design.

Solution: Design your application as a *Single Page Application* using HTML5, CSS3 and JavaScript, which are natively implemented in modern browsers. Store page state within the client and connect to the backend through REST services. This approach is called the "Single Page Application" because all of the HTML, CSS and JavaScript code necessary for a complex set of business functions, which may represent multiple logical screens or pages, is retrieved as a single page request. The JavaScript that manipulates the logical screens will handle all of the manipulation of the HTML DOM, the page navigation and the access to back end data.

Picture:

The following picture (Figure 5: SPA overview) shows the interaction between the different components of the Javascript within an SPA and the back-end systems that implement it communicates with.

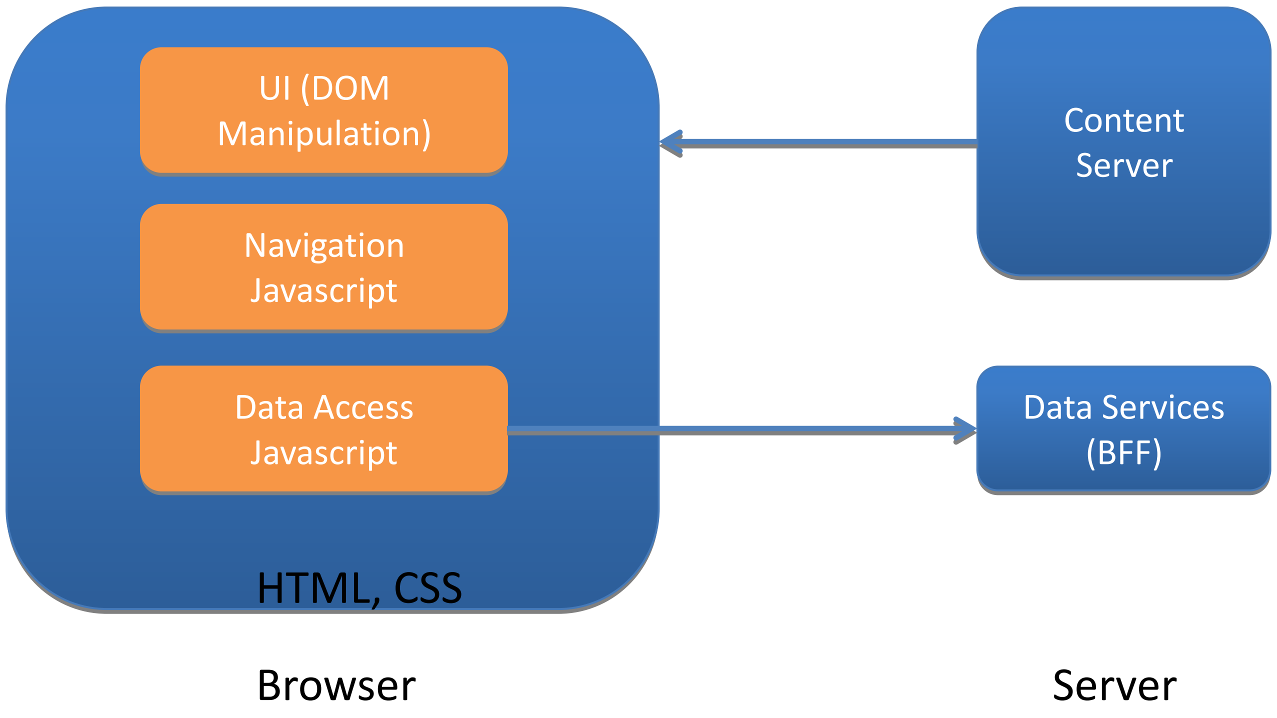

Figure 5: SPA overview

Result: The main benefit of the Single Page Application approach is that it allows much more responsive and intuitive application designs than the dynamic page approach. JavaScript code that executes within a web page can control not only the look and feel of the application by generating and manipulating the client-side DOM in any way it chooses, but can request information from the server at any point based on user interactions – resulting in more responsive user interfaces.

Single Page Applications often are written to take advantage of Responsive Design principles (see [Brown]) to optimize user experience for screen layout and size. CSS Media queries are often used to include specific blocks that only apply for specific screen types. This technique allows a different set of CSS rules to be specified for tablets, mobile phones or laptops, resulting in screens that are laid out and configured specifically for those devices.

In a complex business application you may implement several Single Page Application instances. Each one represents a single logical set of screen interactions that perform a business function. This approach maps extremely well into the microservices approach, as you can match an SPA to the capabilities of one or more *Business Microservices*. However, you may need to perform some translation or conversion of the results of a *Business Microservice* in order to match the unique user interface requirements of your SPA.

2.3 Native Mobile Application

Title: Native Mobile Application

Context: You are building applications that need to support on a variety of platforms. At the same time, most of your customers are going to use a Mobile device as their primary means of accessing your application.

Problem: How do you provide the most optimized user experience on a mobile device and take advantage of the features that make mobile computing unique?

Forces:

- Mobile device capabilities change and evolve quickly
- Applications that are built to emulate a mobile device look and feel often seem outdated when the native user interface libraries of the Mobile OS changes
- Mobile devices, while rapidly improving, often do not have the same processing capability that larger desktop or tablet devices do.

Solution: Write a *Native Mobile Application* for each of the two major platforms (iOS and Android).

While *Single Page Applications* provide a good user interface that can be adapted to different screen sizes and orientations using Responsive Design no browser-based application can take advantage of all the features and capabilities of a mobile device. What's more, even though advances have been made in the speed and performance of Javascript in many browsers, the performance of browser-based applications still noticeably falls behind those of native applications. Another drawback of browser-based applications is that the user must remember to bookmark the webpage of the application – it is not present on their device home screen in the same way that native applications are present.

For these (and other) reasons, developers have found that constantly used, highly interactive applications, should be implemented as native applications using the tools and capabilities provide by the native development tool suite. This allows the developer to take maximum advantage of the platform's capabilities, and at the same time allows users to easily locate and download the application through the platforms application store.

That is not to say, however, that the two patterns can't be used together. There are certain situations in which it may not be absolutely required that you take advantage of the performance of a Native Mobile application in all parts of a large mobile application. In those cases, a hybrid approach in which the most performance-intensive parts of the application are implemented using native mobile components, while the less performance-intensive pieces are implemented as a *Single Page Application* that runs within an embedded mobile browser, may be an effective tradeoff of development time for runtime performance.

Result: Just as with *Single Page Applications,* microservices are a good match to *Native mobile applications* since the business-oriented capabilities of a *Business Microservice* map cleanly to the complex screen flow and interaction capabilities of a Native mobile application.

However, *Native Mobile Applications* have their own drawbacks that necessitate additional architectural decisions – most notably the limited screen real estate of an application and the potentially poor Internet connectivity over a mobile network may necessitate the use of other patterns such as a *Near Cache* or *Backend for Frontend*.

Native Mobile Applications often are paired with *Backend for Frontends* that can filter and translate results to data format that are specific to the Mobile platform. Likewise, *Native mobile applications* may benefit from a *Near Cache* when Internet connectivity is slow or unreliable.

2.4 Near Cache

Title: Near Cache

Context: You are writing either a *Single Page Application* or a *Native Mobile Application*. The application must be able to operate efficiently even when Internet connectivity is not available at the highest speeds.

Problem: How do you reduce the total number of calls to back-end microservices (particularly *Backend for Frontend*'s) for repeated information?

Forces:

- You don't want to cross the network any more than necessary, especially when network bandwidth is at a premium in a mobile device.
- You don't want to make the user wait any more than is absolutely necessary.

Solution: Use a Near Cache located within the client implementation. Cache the results of calling the Backend for Frontend services so as to reduce unnecessary round trips to the server.

The simplest type of near cache is a globally scoped variable containing a hashtable data structure - something that is easily supported by JavaScript for Single Page Application or Java or Swift for Native Mobile Applications. However that may not always be the best solution -- for instance in a Single Page Application, the application can also use HTML5 Local Storage for storage of cached information.

Near caches are easily implemented in Native Mobile Applications – for instance in iOS, Core Data or Property Lists can easily be used to store cached data locally. SQLite can be used on both iOS and Android as a fast, local structured data store.

Result: The benefit of a *Near Cache* is that it reduces the total number of times you must call a *Backend for Frontend* to retrieve repeated information. The drawback is that you must now manage the lifetime of the information in the cache to avoid it becoming stale, which can add complexity to your application code.

3. MICROSERVICES ARCHITECTURE PATTERNS

Microservices Architecture Patterns deal with the nuts and bolts of identifying and implementing Microservices. **Microservices Architecture** is the basic root pattern that you begin your design journey by following. That will lead you to implement multiple **Business Microservices**. You may have to implement **Adapter Microservices** if you are required to communicate with existing back-end systems. The **Backend for Frontends** patterns is a key component for implementing a Microservices Architecture in the context of a Modern Web Architecture in connecting different client types to your Business Microservices. Finally, **Page Caches** and **Results Caches** are commonly used approaches to improve Microservice performance.

3.1 Microservices Architecture

Title: Microservices Architecture

Context: You're designing a server-side, multi-user application. You want your application to be modular, and you want the modules to be independent. Your application modules need to be capable of composition, scalability, and continuous deployment.

Problem: How do you architect an application as a set of independent modules?

Forces:

- Modular code doesn't tend to stay modular when it's being developed by one big team and runs in one monolithic process.
- You don't want to have to wait until development of all modules is complete in a large system before releasing some of the functionality if it is independently useful to the business.
- When a single business function is divided into separate modules, those modules must evolve in lockstep.
- Separate business functions can evolve at different rates.
- For a component to be modular, not only must its code be modular, but its data must be isolated and logically cohesive as well.

Solution: Design the application with a *Microservices Architecture*, which is a set of modules where each is an independent business function with its own data, developed by a separate team and deployed in a separate process. For instance, a large retailing website may have separate microservices for different functions, such as catalog item search, cart checkout, and customer service. These can be developed and released on different timelines.

By developing a large application as a set of independent microservices, the individual services can be released and likewise evolve at their own pace. You don't have to wait for the entire system to be complete before the business can begin to realize value from the system. There are different styles of microservices for different needs, for instance adapting to existing systems vs.

implementing new business logic, but all share the same traits of independent scaling and independent development.

In fact, the microservices approach gives you better control over scalability than a traditional approach can give you. By only scaling each microservice to the level at which they are required in order to handle the requests made for that specific service, you will be able to more efficiently use your computing resources than in the traditional model of scaling the entire monolith to the largest number of instances needed to handle requests for the most commonly used functions.

Result: Splitting up business functions into multiple small *Business Microservices*, allows each microservice to be developed independently by a small team.   Where you need to adapt an existing interface such as a SOA service, you can write an *Adapter Microservice*. *Backends for Frontends* (BFFs) are used to mediate between different client types and Business Microservices. Microservices can be deployed independently with zero downtime for continuous deployment. Microservices can scale and fail independently. Microservices developed and deployed independently will tend to become and remain separate modules.

One monolith is easier to monitor and manage than multiple microservices, so you'd better get good at *Microservices DevOps*.

Picture: The following diagram (Figure 6: Typical Microservices Architecture) shows a typical style of connection between a number of microservices comprising both a *Single Page Application* for Web clients and a *Native Mobile Application*.

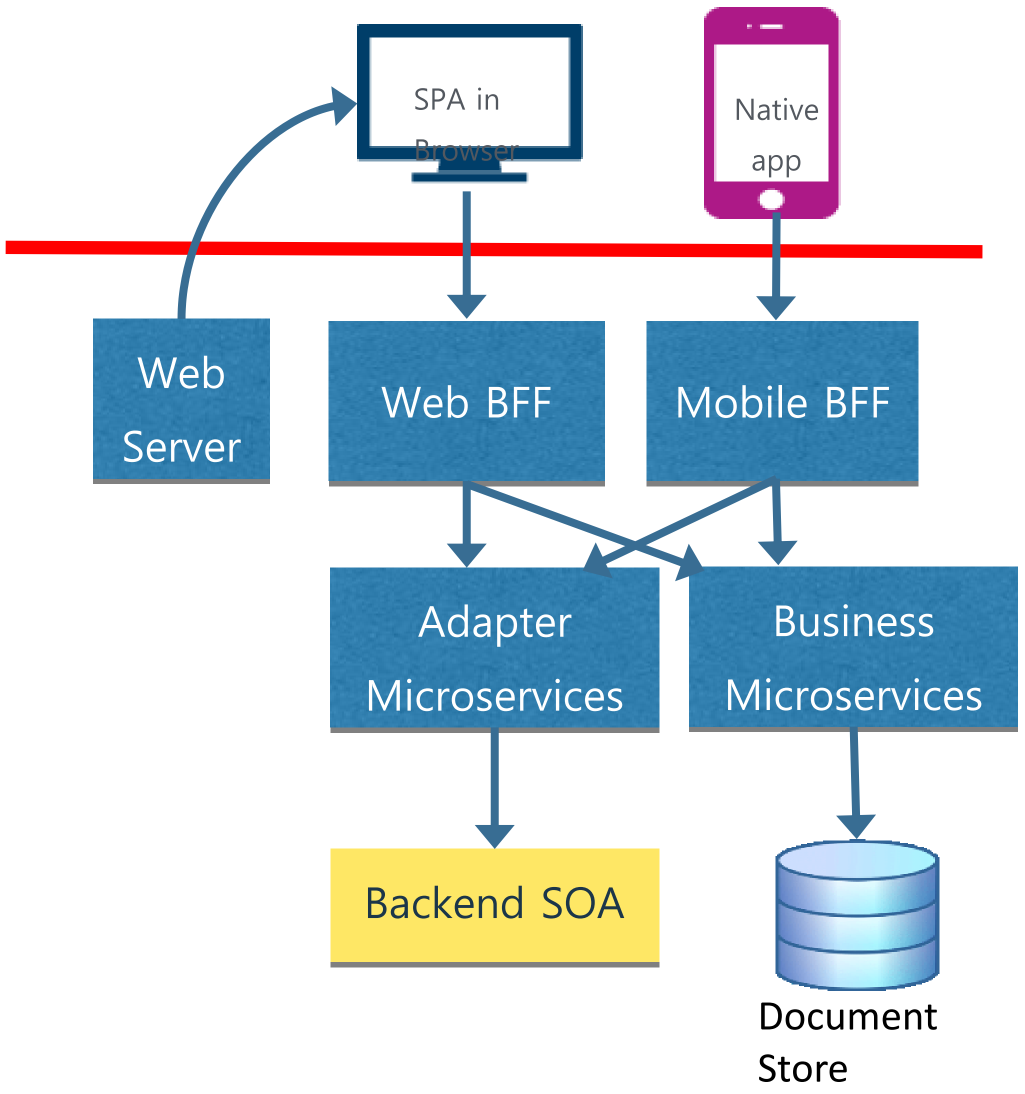

Figure 6: Typical Microservices Architecture

3.2 Business Microservice

Title: Business Microservice

Context: You are implementing an application using the *Microservices Architecture* style or refactoring an application to use that architectural style. The application may run in either the cloud or on-premise.

Problem: Ho do you isolate and implement business logic in an application built using a Microservices architecture?

Forces:

- Attempting to implement more than one business function in a component results in compromises in scaling (you may scale functions to levels greater than they are required) and introduces unnecessary coupling between functions.
- Component API's should be business oriented rather than technically oriented. A business-oriented component API can be more readily understood by

Solution: Develop each business function as a *Business Microservice* that encapsulates the business logic and makes it composable. Each Business Microservice should solve one and only one business problem. That business problem should be addressed in single deployable component that has a business-oriented API. In this way, you can not only allow for evolution of your implementation of each business function (as long as the API remains the same or upwardly compatible, the implementation is immaterial) but you allow for the evolution of the system as a whole as new *Business Microservices* can be developed to meet new business needs or the system can recombine existing Business Microservices in new ways.

As an example, let's return to the Retailing website described in the *Microservices Architecture* pattern. Let's say that today we implement the three microservices described there. The catalog item search microservices may be implemented initially using a technology like Apache SOLR. If you provide a business-oriented API on that microservice that does not expose internal technical details then you can entirely replace the implementation of catalog item search with a new version that implements the same API but uses a newer technology like ElasticSearch easily without having to retest the entire system. Since you have isolated the technology and data dependencies within the microservice you can allow the team building that microservice to evolve it independent from other parts of the system. Note that this only works if each microservice is only related to other microservices through its published interface – if there is hidden coupling (such as through a shared database) then independent evolution and scaling is not possible.

Result: The business functionality of an entire application is composed of individual business functions implemented as business microservices. Deciding to implement each business entity as a microservice is not the end of your design problems, however. You must also think about how you would implement the microservice and how that microservice relates to the other services in your overall business application.

One of the microservices principles described by [Fowler] and referenced above is that you should avoid indirect communication between microservices through a database. One of the ramifications of that principle is that each individual microservice will usually store its own data in a separate database. In order to allow those databases to scale at the same rate as the microservices, the decision is often made to use a *Scalable Store* (e.g. a NoSQL Database of some sort) instead of a relational database. This is usually because the native data representation that the microservice presents to the outside world in its REST interface (JSON or possibly XML) is more easily stored and retrieved in that format.

Earlier in this paper we referred to the development benefits that are accrued from following the microservices principles of Per Service CI/CD and Per-Service HA and clustering. Unfortunately these two benefits come with a shared drawback - adopting an approach of building multiple *Business Microservices* means that you will have more operational complexity in your application than you might have if you followed a more traditional monolithic approach.

In particular, the increase in the number of application servers running your microservices means that there are more application logs to gather and correlate than you might have in a monolithic architecture. Likewise, this same increase means that monitoring the status of each individual microservice and debugging problems that may occur also becomes more complex. Thus, when building *Business Microservices*, you may need to employ a *Log Aggregator* together with an approach like *Correlation ID's*.

3.3 Backend for Frontend

Title: Backend for Frontend

Context: You are designing an application using a *Microservices Architecture.* You realize that the *Business Microservices* that encapsulate individual business functions do not map cleanly to the channel-specific needs of your client applications. You have multiple different channels (e.g. front-end application types, such as mobile applications and web applications) that you must support.

Problem: How do you represent a channel-specific service interface that is consistent with a microservices architecture but allows enough uniqueness that it can be adapted to the needs of a specific client type?

Forces:

- Microservices break one set of functionality into multiple APIs.
- The easiest way for a client to interface with a server is through a single API.
- Different types of clients—browser, mobile, CLI—want different APIs for the same functionality.
- Business logic is rarely specific to a single client type. It should be implemented such that all client types can share it.

Solution: Build a "Backend for Frontend" (e.g. a *BFF* or Dispatcher) that acts as a single API for a client. Implement different BFFs for different types of clients, each with an API that is customized to what that client type needs. Each Backend for Frontend will then interface

Results: A *BFF* can perform the following actions:

- **Orchestration** – It can orchestrate several calls to business microservices that result from a single client action.
- **Translation** – It can translate the results of a microservice into a channel-specific representation that more cleanly maps to the needs of the user experience of that client.
- **Filtering** – It can filter results from a business microservice that are not needed by a particular client type.

A BFF should not contain any business logic. Because it's specific to a single client type, any business logic implemented in BFFs won't be shared across all client types. This is the single biggest potential pitfall of the BFF pattern – there is a risk that business logic becomes embedded in the BFF instead of within the underlying business microservices where it should be implemented, making that logic inaccessible to other channels.

This pattern was introduced in [Newman] and is a key part of building microservices architectures that have dynamic front-ends such as *Native Mobile Applications* or *Single Page Applications*. Both of these patterns introduce unique translation or filtering requirements that often necessitate the use of a Backend for Frontend. For instance, in a native application you may need to filter long data sets in order to fit on the limited screen real estate of the mobile application. Likewise a Single Page Application may require orchestration of several calls to individual *Business Microservices* to return a particular set of information needed to represent a screen flow.

An important point about implementing the Backend for Frontend pattern is that in most cases, it is the same team that is responsible for building both the client application and the Backend for Frontend. This leads to some particular development efficiencies; the first is that it is often convenient to use the same programming language for both. Thus, many times JavaScript developers building a Single Page Application will also want to develop their Backend for Frontend services using JavaScript and Node on the server side. Likewise, Java developers building an Android native application may want to develop their Backend for Frontend services with Java.

Backend for Frontend's often use *Page Caches* to store long results obtained from *Business Microservices* so that the results can be obtained a page at a time from a client. A *Service Registry* may help the client code that makes up the bulk of a Backend for Frontend to be resilient in the face of changes to the physical address of the *Business Microservices* that it depends on.

3.4 Adapter Microservice

Title: Adapter Microservice

Context: You are designing an application using a Microservices Architecture. You want it to incorporate existing services (e.g. SOAP, JMS, or mainframe-based services) but their APIs do not use the RESTful or queue-based approach that is consistent with the microservices architecture.

Problem: How do you handle translation of existing service implementations into good microservices APIs?

Forces:

- If you have existing services that run well enough as is, you don't want to rewrite them, you want to reuse them instead.
- Changing the APIs of existing services can break existing clients.
- If an existing service doesn't have a good microservice API and that's inconvenient to change, it's also inconvenient for each client microservice to use that existing service via its legacy API. The components in a microservices architecture should integrate via good microservice APIs.

Solution: Build simple *Adapter Microservice* that converts the existing service's non-microservice API to an API that client microservies will expect. An Adapter Microservice follows the Adapter pattern from [Gamma] in that it converts one API to another.

Results: In many cases, it's a straightforward exercise to convert a function-based interface (for instance one built using SOAP) into a business-concept-based interface. In many ways this can be thought of as moving from a verb-based (functional) approach to a noun-based (entity) approach. Often, the functions exposed in a SOAP endpoint correspond one-to-one to CRUD (create, read, update, delete) operations on a single business object type, and therefore can map easily to a REST interface where the URL represents the type of object, and the GET, POST, PUT and DELETE methods correspond to read, create, update and delete for that type respectively. These operations would then simply send the corresponding SOAP messages to the existing SOAP endpoint and then translate the XML datatypes from the corresponding SOAP operations to JSON datatypes for the new REST interface.

However, there are cases where this mapping falls down that must also be handled. An example of this case is a bank transfer - which is an operation between two bank accounts that does not correspond to a CRUD operation on either account. In this case, the simplest solution is often the same one taken in relational databases for the corresponding problem - you create a new type that represents the transfer itself - analogous to creating a relationship table in a relational database.

So in this case you would build a new REST interface whose URL represents the Transfer type, with a POST operation corresponding to performing a transfer, a GET operation that could return information about a transfer, and a DELETE operation that is a (hopefully well-controlled) undo.

Adapter Microservices are a special type of *Business Microservice* - they are often transient solutions, which only last until the underlying existing services can be replaced by a natively implemented microservice.

Due to their special nature of converting one interface to another, Adapter Microservices often use *Results Caches* in order to reduce the number of times they have to invoke the underlying SOA service that they convert.

3.5 Results Cache

Title: Results Cache

Context: You are building clients for services, particularly in a Microservices Architecture. Making network calls to remote databases or services are expensive. Latency can be added by distance, layers or simply by long processing in a service.

Problem: How do you improve the performance of your application or microservice when it makes many repeated calls to services?

Forces:

- Some services, when called repeatedly, return the same data every time and don't cause any side effects. Example: A read-only database query.
- Some services produce side effects. Example: A query that writes data or increments counters.
- Caches can become very complex – especially if you try to anticipate what may need to be cached ahead of time.
- Caching is optional. It should improve the performance of a system, but shouldn't change behavior.

Solution: Use a *Results Cach*e that shortcuts the need to make repeated calls to the same service.

Results: A Results Cache can be as simple as a *Key-Value Store* (either in-memory or in a *Scalable Store*). For this to work, the cache client must be able to derive the key from the parameters of the database query or services call. Then the value is the last result returned from the service.

Determining what to cache is often as difficult as setting up a cache itself. A common approach that is sometimes taken is to try to reproduce the data in a database locally to the application that is making requests of that data. The logic here is that if the problem is that database calls are expensive and slow, then by making those calls locally (perhaps in-memory) then you can reduce that overhead.

The problem is that if you simply reproduce the same data structures you have in a database locally to your program that you will have to also be able to reproduce the same type of operations on that data that your database is capable of making. The downside of that is that it's not easy or obvious to determine how to do the equivalent of a multi-way SQL JOIN statement together with database query optimization either in a language like Java or Node, or in a simple *Key-Value Store* like Redis!

So instead, use a *Results Cach*e. A Results cache can be as simple as a *Key-Value store* (either in-memory or in a *Scalable Store*) in which the key is easily derived from the parameters of a Database query or services call and the value is the last result returned from the service.

Probably the hardest issue to resolve in implementing a *Results Cache* is how to keep it from becoming stale[2]. In many cases, the solution is simply to have a relatively low timeout value - in most interactive business applications it is relatively rare to have any cached data need a lifetime beyond that of a user session, which is usually measured in minutes -- so a short cache lifetime of a few minutes or less can significantly improve performance while still maintaining a minimal risk of the cache growing stale.

[2] For help with that problem, see [Fernandes]

3.6 Page Cache

Title: Page Cache

Context: You are building a web or mobile application using a *Microservices Architecture*. You are applying the *Backend for Frontends* Pattern to build dispatchers for a web or mobile application.

Problem: A single microservice that represents a business entity or concept may return more information that can be easily displayed, particularly on a mobile application, without a great deal of scrolling.

Forces:

- You don't want to re-fetch all of the data in a long list each time the list needs to be paged
- You don't want to spend unnecessary network bandwidth transmitting data that may never be viewed
- You don't want the user to have to wait a long time for data to be fetched before it is displayed.

Solution: Use a *Page Cache* with your *Backend for Frontend*. The Backend for Frontend will present an interface that allows for the client application to request a limited subset of a much larger set of data. The information can be indexed by page number based on the total size of the dataset and the number of elements per page that can be displayed on the device.

Results: For very long datasets, a Page Cache is preferable to fetching all the data at once and storing it in a *Near Cache* since you can render the first set of information more quickly – and in many cases, the user will never scroll past the first page of information. However, when you do use a page cache in conjunction with a Near Cache it makes it possible to scroll backwards through the list more quickly – otherwise you would need to re-fetch the page data from the Backend for Frontend on both page down and page up requests.

4. SCALABLE STORE PATTERNS

A key part of the microservices architecture is that wherever possible each microservice should control or “own” its own data. However, microservices are also expected to be scalable and stateless. This combination of requirements leads to the need for **Scalable Stores,** which is the root pattern of this section. A Scalable Store can come in many different types, such as a **Key-Value Store** or a **Document Store**. The choice for which you need depends upon the type of data you are storing.

4.1 Scalable Store

Title: Scalable Store

Context: You are developing an application using *Business Microservices*. However, your application will need persistent state to represent current and previous user interaction.

Problem: How do you represent persistent state in an application?

Forces:

- Storing application data inside the memory of an application runtime leads you to require routing solutions like sticky sessions that make building highly available systems difficult.

- You don’t want to limit the scaling of your application by forcing the application to store state inside a data store that cannot scale linearly.

Solution: Put all state in a *Scalable Store* where it can be available to and shared by any number of application runtimes.

The key here is that the database must be naturally distributed and able both to scale horizontally and to survive the failure of a database node. For the last several years, that has driven developers to the concept of “NoSQL” databases as described in [Sadalage]. Examples of this type of database include Redis, Cloudant, Memcached.

However, quite recently a number of new distributed databases based on the relational model and collectively called “NewSQL” have also become available. These include options like Clustrix, MemSQL, or even MySQL cluster. NoSQL databases are typically less efficient at SQL-like queries because of differences in approaches to query optimization. So, if your application depends on SQL-centric complex query capability then a solution such as a NewSQL database or a distributed in-memory SQL database may be more efficient.

In the end it does not matter which particular database you choose – so long as you choose the right database structure and retrieval mechanism for the job you are trying to perform. If the driving forces in your application are scalability and redundancy either a NoSQL or NewSQL database would suffice, but other application requirements will determine the particular type of database you must choose.

Results:

Result Caches and *Page Caches* are usually implemented with a Scalable Store.

4.2 Key Value Store

Title: Key Value Store

Context: You are building an application with a Microservices architecture.

Problem: How do you store data that is naturally accessed through a simple key lookup such as cache entries?

Forces:

- You want to use the simplest tool for the job at hand.
- You want to make your accesses (Cache lookups or other accesses) as fast as possible.

Solution: Store your data in a scalable Key-Value Store. The principal advantage of a key value store over other types of Distributed Store is its simplicity. Most Key-Value Stores act, in principle, like a hash map. For example, Redis has simple GET and SET commands to store and retrieve string values. What's more, for simple key lookup operations, Redis offers O(1) performance.

If, on the other hand, you used a more complex store type such as a Document store for storing cache entries, then you would find that the performance of such solutions is often not as good. That is because other distributed store types optimize for more complex cases such as searching by the contents of the documents stored.

Results:

There is no magic in using a Key-Value store. In many cases such as EhCache, they are in-memory stores, and as such only can provide Consistency and Availability but not Partition tolerance (as per Brewer's CAP Theorem). This limits the size of the cache. The mechanism used by many others (e.g. Redis with clusters) is to map keys to specific distributed nodes, and to route requests to the appropriate server, which then stores that corresponding set of values in memory. In order to maintain Availability, this means that you must have copies (slaves) of each node. This means that by the CAP theorem you can gain Partition tolerance, but at the potential loss of Consistency while the master synchronizes with the slave.

4.3 Document Store

Title: Document Store

Context: You are building applications with a Microservices Architecture. You are using RESTful API's and representing the contents of your HTTP requests and responses as JSON (Javascript Object Notation) documents.

Problem: How do you most efficiently store and retrieve the data corresponding to your HTTP responses?

Forces:

- You don't want to have to translate the on-the-wire representation of an entity into another form just to store it in a database.
- You want to build your solution quickly, and don't want to have to add lots of additional libraries or code to manipulate a database.

Solution: Store your JSON documents in a data store that is designed to quickly and efficiently retrieve and store JSON documents – a *Document Store*.

Results:

In a microservice, the requests and responses that for the values accepted and returned by the microservice are most commonly presented as JSON values. JSON has emerged as the lingua-franca of REST, quickly eclipsing and replacing XML for most service schemas. When a resource is thus represented naturally by a JSON document developers want to use the most efficient database they can for storing and retrieving that information. Increasingly, this choice is a Document Store such as MongoDB or CouchDB.

For instance, in MongoDB the basic construct is a collection of JSON documents – and adding a JSON document to that collection is as simple as obtaining a reference to the collection and calling an add method. This type of simplicity is what makes this approach attractive to developers building microservices in Javascript.

5. MICROSERVICES DEVOPS PATTERNS

While not absolutely required in order to implement a **Microservices Architectur**e, teams soon find that the greatly increased number of runtimes needed in a microservices architecture lead to new challenges in managing that proliferation. This leads to the need for **Microservices DevOps,** which is the root pattern of this section. Microservices Devops will lead you to implement patterns such as a **Log Aggregator** and **Service Registry**. Debugging long chains of Microservices may lead you to implement **Correlation ID's**.

5.1 Microservices DevOps

Title: Microservices DevOps

Context: You are building a complex application with a *Microservices Architecture.* You need to be able to put that system into production.

Problem: How do you balance the needs of developers (which want to work on small, easy-to-modify artifacts) and operations teams, which need to minimize the impact of changes on critical systems in order to maintain availability?

Forces:

- If you deploy microservices in the same way you deployed a monolithic application, you will not obtain the benefits of a microservices architecture.
- If you let each team go off and follow their own approach without setting guidelines on how microservices are deployed, modified and managed, it will be impossible to debug the resulting system.

Solution:

Follow a **Microservices DevOps** approach that allows you to isolate each Microservice as much as possible, while being able to easily and quickly identify and resolve issues with the Microservices. Applying unique CI/CD pipelines to each Microservice will allow you to build and deploy each microservice individually. However, having a common approach allows you to set guidelines on how microservices interrelate and interact with each other.

Results:

Since changes to a microservice can occur at any time, other microservices need to be isolated from the results of that change through a *Services Registry*.

What's more, microservices cannot function completely independently – since each microservice will depend on other microservices or other downstream services (such as a DocumentStore) you need to apply techniques such as a *Log Aggregator* and *Correlation ID's* to understand the interaction between different microservices and debug issues.

5.2 Log Aggregator

Title: Log Aggregator

Context: You are building applications using a **Microservices Architecture** and need to be able to debug problems that cross the different components of that architecture.

Problem: How can you effectively view and search all of the different log files that emerge from all of the different distributed runtimes that comprise your collection of microservices?

Forces:

- A *Microservices Architecture* can span dozens or even hundreds of individual application server processes. This results in hundreds of individual log files.
- Microservices are often (but not always) implemented using cloud solutions that limit the lifetimes of the individual application server processes. When you use a solution like Cloud Foundry or docker, when an individual server container dies, any state within that container is lost.

Solution: Use a *Log Aggregator* that pulls all of the files together into a single, searchable database. The Log Aggregator will "listen for" or tail each individual log file and forward the log entries to an aggregated collection point as they are made.

Examples of Log Aggregators include: Splunk, Logstash and ElasticSearch, and the Cloud Foundry Loggregator.

Results:

Once you have collected log entries together, then the database of the entries can be searched in order to make debugging more meaningful. For instance, you can look for occurrences of the same log entry, perhaps a particular error message across multiple log files from multiple servers in order to see if a problem is sporadic or widespread.

But the most helpful debugging approach is to take entries that are connected by a single thread of control that spans multiple microservices and correlate them together to find problems that cross a network of microservices in a particular call graph. That problem can be solved through the combination of a *Log Aggregator* and *Correlation ID's*.

5.3 Correlation ID

Title: Correlation ID

Context: You are building an application in a *Microservices Architecture*. Your application may use *Backends for Frontends* and multiple *Business Microservies* and *Adapter Microservices*, and may have complex call graphs as a result.

Problem: How do you debug a complex call graph when you do not know where in the set of microservices along that call graph the problem may have been introduced?

Forces:

- You don't want to overly burden your developers with complex requirements for logging and monitoring
- You need to be able to trace a call regardless of how complex or simple that call is.

Solution:

The simplest and most effective debugging tool for a complex microservices web is consistent use of correlation id's. A correlation ID is a simple identifier (usually just a number) that is passed in to every service request and passed along on every succeeding request – when any service logs something for any reason, the correlation id is printed in the log entry. This allows you to match or correlate specific requests to one service to other service requests in the same call chain. Implementing correlation id's correctly requires four consistent actions:

1. Create correlation ID if none exists and attach it as a header to every outgoing Service Request
2. Capture the incoming correlation ID on every incoming request and log it immediately
3. Attach the correlation ID to the processing of the request (through a threadlocal variable…) and make sure that the same correlation ID is passed on to any downstream requests
4. Log the correlation id and timestamp of *any* messages that are connected to that thread of processing

Results:

Once you have implemented a *Correlation ID* you can then use a *Log Aggregator* to gather together all of the logs across all of the dependent systems in your *Microservices Architecture* and can perform a query for the *Correlation ID* against the aggregated log. When placed into timestamp order the results of this query will show the call graph of the solution and allow you to put errors into the appropriate context in terms of which microservice instance and thread handled the calls, and what the parameters to the calls were at the time that the problem was encountered.

5.4 Service Registry

Title: Service Registry

Context: You are building applications using a *Microservices Architecture.* You have many different *Business Microservices* comprising your application.

Problem: How do you decouple the physical address (IP address) of a microservice instance from its clients so that the client code will not have to change if the address of the service changes?

Forces:

- You don't want to have to change client code each time you update a microservice, for instance, if you update the version of the microservice, which may change the URL

- You don't want to have to hard-code the physical addresses of a microservice into client code – that would make it difficult, for instance, to move code between development and production when the physical addresses of the systems change.

Solution: Use a *Service Registry* to map between a unique identifier and the current address of a service instance in order to decouple the physical address of a service from the identifier. The Service Registry pattern was first introduced in [Daigneau] and remains useful in a microservices implementation – perhaps even more so given the number of individual services in a *Microservices Architecture*.

At its simplest, a Service Registry is just a repository for information about services. However, many implementations add other metadata about the service along with the physical address, such as a generic server name (useful when you have multiple instances of a service) and health information such as status or uptime.

Examples include IBM Bluemix's Service Discovery, Netflix Eureka and the Amalgam8 Registry.

Results: Not all microservices projects will require a Services Registry – if you have only a handful of services in your application then the setup and management of the registry infrastructure is often more trouble than it's worth. But if you have more than a half a dozen services then it may become useful when alternative ways of managing service location (such as configuration files) become cumbersome.

REFERENCES

[BUSCHMANN] BUSCHMANN, FRANK, ET. AL., *PATTERN ORIENTED SYSTEM ARCHITECTURE: A SYSTEM OF PATTERNS*, JOHN WILEY AND SONS, 1996

[BROWN], BROWN ., *MODERN WEB APPLICATION DEVELOPMENT WITH IBM WEBSPHERE*, IBM PRESS, UPPER SADDLE RIVER, NJ, 2014

[Daigneau] Daigneau, Rob, *Service Design Patterns*, Addison-Wesley, Nov 2011

[Fernandes] Fernandes, Benedict, *Architectural Patterns for Dealing with Stale Data*, IBM DeveloperWorks,, http://www.ibm.com/developerworks/websphere/techjournal/1506_fernandes/1506_fernandes-trs.html, dated June 2015, retrieved 14 June, 2016

[Fowler] Fowler, Martin and Lewis, James, *Microservices: A Definition of a new Architectural Term*, http://martinfowler.com/articles/microservices.html, originally dated 25 March 2014, retrieved 18 May 2016.

[Fowler2] Fowler, Martin, *The Strangler Application,* https://www.martinfowler.com/bliki/StranglerApplication.html, originally dated 30 June 2014, retrieved 18 February 2017

[Gamma] Gamma, Erich, et. al., *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1994

[Hohpe] Gregor Hohpe, et. al., *Enterprise Integration Patterns*, Addison-Wesley Professional, Upper Saddle River, NJ, 2003

[Martin] Martin, Robert, *Principles of Object Oriented Design*, http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, retrieved 18 February 2017

[Newman] Newman, Sam, *Building Microservices*, O'Reilly Media, February 2015

[Sadalage] Sadalage, Pramod, and Fowler, Martin, *NoSQL Distilled*, Addison-Wesley Professional, August 18, 2012