Microservices Architecture

Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

1. Common Characteristics

Following are the some of the common characteristics of microservice architecture. As with any definition that outlines common characteristics, not all microservice architectures have all the characteristics, but we do expect that most microservice architectures exhibit most characteristics.

1.1. Componentization via Services

There's been a desire to build systems by plugging together components, much in the way things are made in the physical world. During the last couple of decades there has been some considerable progress with large compendiums of common libraries that are part of most language platforms. Component can be defined as a unit of software that is independently replaceable and upgradeable. Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services. We define libraries as components that are linked into a program and called using in-memory function calls, while services are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call.

One main reason for using services as components (rather than libraries) is that services are independently deployable. If an application consists of multiple libraries in a single process, a change to any single component results in having to redeploy the entire application. But if that application is decomposed into multiple services, a change to single service require only that service to be redeployed. That's not an absolute, some changes will change service interfaces resulting in some coordination, but the aim of a good microservice architecture is to minimize these through cohesive service boundaries and evolution mechanisms in the service contracts.

Another consequence of using services as components is a more explicit component interface. Most languages do not have a good mechanism for defining an explicit published interface. Often, it's only documentation and discipline that prevents clients breaking a component's encapsulation, leading to overly-tight coupling between components. Services make it easier to avoid this by using explicit remote call mechanisms. Using services like this does have downsides. Remote calls are more expensive than in-process calls, and thus remote APIs need to be coarser-grained, which is often more awkward to use. If there is a need to change the allocation of responsibilities between components, such movements of behavior are harder to do when it is crossing service boundaries.

1.2. Organized around Business Capabilities

When looking to split a large application into parts, often management focuses on the technology layer, leading to UI teams, server-side logic teams, and database teams. When teams are separated along these lines, even simple changes can lead to a cross-team project taking time and budgetary approval. A smart team will optimize around this and plump for the lesser of two evils - just force the logic into whichever application they have access to. Logic everywhere in other words. This is an example of Conway's Law in action - "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure".

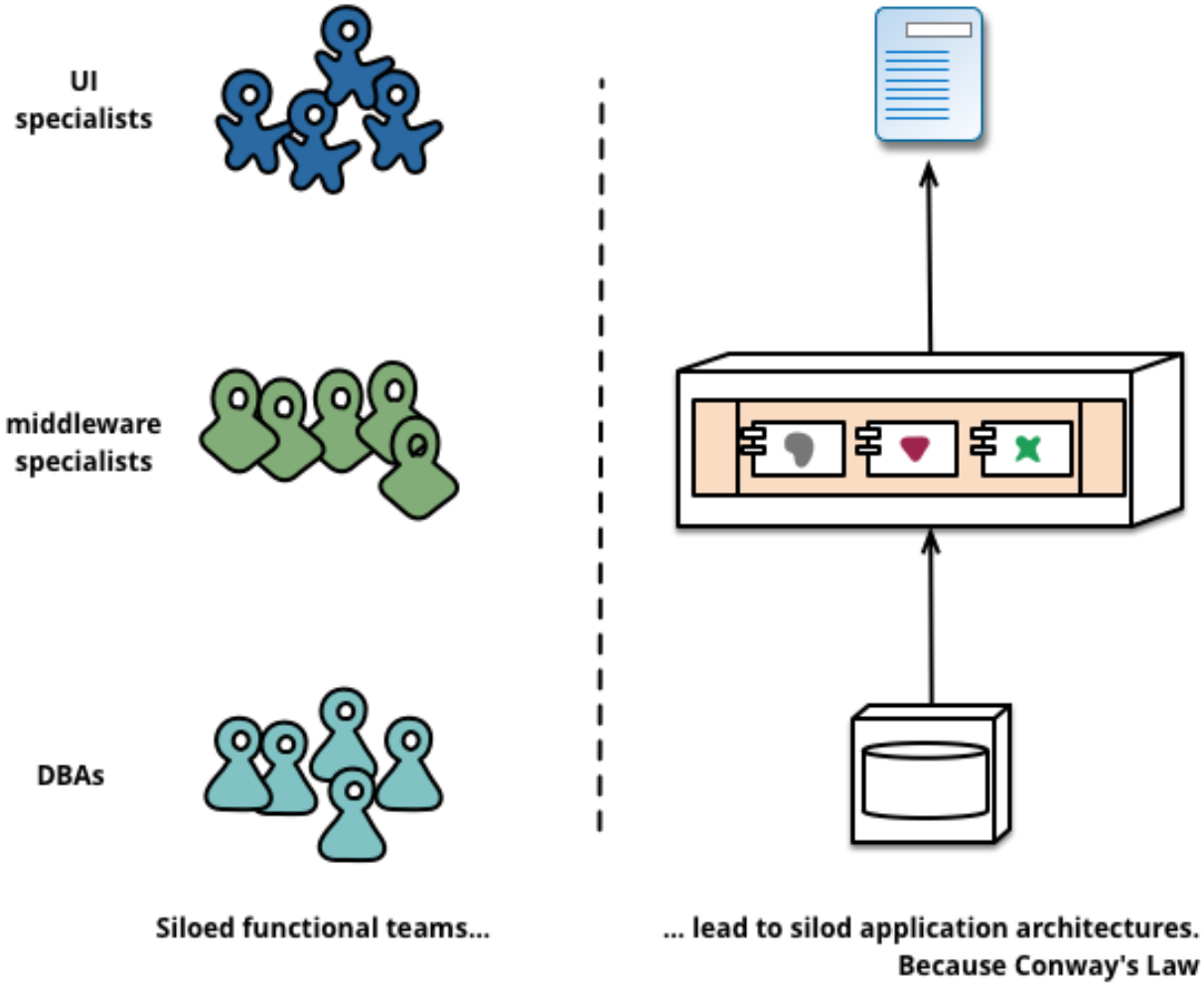

Figure 1: Organizing team based on technology layer

The microservice approach to division is different, splitting up into services organized around business capability. Such services take a broad-stack implementation of software for that business area, including user-interface, persistent storage, and any external collaborations. Consequently, the teams are cross-functional, including the full range of skills required for the development: user-experience, database, and project management.

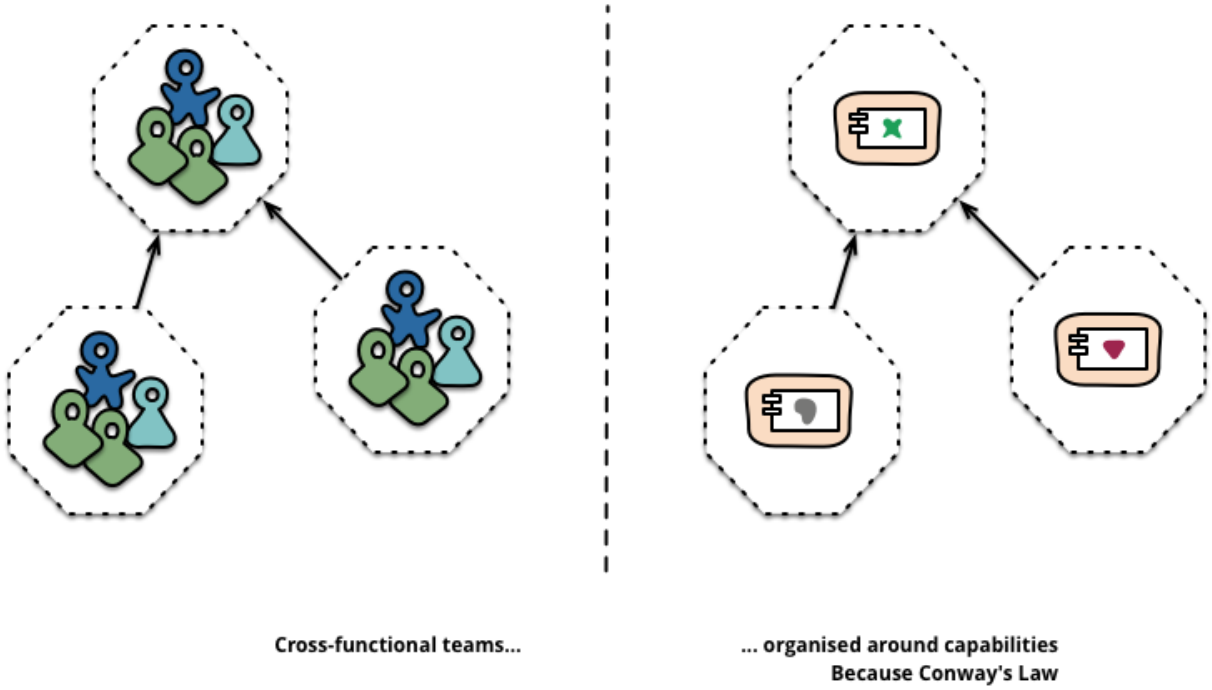

Figure 2: Organizing team based on business capability

1.3. Products not Projects

Most application development efforts that we see use a project model: where the aim is to deliver some piece of software which is then considered to be completed. On completion the software is handed over to a maintenance organization and the project team that built it is disbanded. Microservice proponents tend to avoid this model, preferring instead the notion that a team should own a product over its full lifetime. This brings developers into day-to-day contact with how their software behaves in production and increases contact with their users, as they have to take on at least some of the support burden.

The product mentality ties in with the linkage to business capabilities. Rather than looking at the software as a set of functionalities to be completed, there is an on-going relationship where the question is how software can assist its users to enhance the business capability. There's no reason why this same approach can't be taken with monolithic applications, but the smaller granularity of services can make it easier to create the personal relationships between service developers and their users.

1.4. Smart endpoints and dumb pipes

When building communication structures between different processes, we've seen many products and approaches that stress putting significant smarts into the communication mechanism itself. A good example of this is the Enterprise Service Bus (ESB), where ESB products often include sophisticated facilities for message routing, choreography, transformation, and applying business rules.

The microservice community favors an alternative approach: smart endpoints and dumb pipes. Applications built from microservices aim to be as decoupled and as cohesive as possible - they own their own domain logic and act more as filters in the classical Unix sense - receiving a request, applying logic as appropriate and producing a response. These are choreographed using simple RESTish protocols rather than complex protocols such as WS-Choreography or BPEL or orchestration by a central tool.

The two protocols used most commonly are HTTP request-response with resource API's and lightweight messaging. Microservice teams use the principles and protocols that the world wide web is built on. Often used resources can be cached with very little effort on the part of developers or operations folk. The second approach in common use is messaging over a lightweight message bus. The infrastructure chosen is typically dumb (dumb as in acts as a message router only) - simple implementations such as RabbitMQ or ZeroMQ don't do much more than provide a reliable asynchronous fabric - the smarts still live in the end points that are producing and consuming messages in the services.

In a monolith, the components are executing in-process and communication between them is via either method invocation or function call. The biggest issue in changing a monolith into microservices lies in changing the communication pattern. A naive conversion from in-memory method calls to RPC leads to chatty communications which don't perform well. Instead you need to replace the fine-grained communication with a coarser -grained approach.

1.5. Decentralized Governance

One of the consequences of centralized governance is the tendency to standardize on single technology platforms. Experience shows that this approach is constricting - not every problem is a nail and not every solution a hammer. Right tool should be used for the job; while monolithic applications can take advantage of different languages to a certain extent, it isn't that common. Splitting the monolith's components out into services gives choice when building each of them. Node.js can be used to standup a simple reports page, C++ for a particularly gnarly near-real-time component, etc.

Teams building microservices prefer a different approach to standards too. Rather than use a set of defined standards written down somewhere on paper they prefer the idea of producing useful tools that other developers can use to solve similar problems to the ones they are facing. These tools are usually harvested from implementations and shared with a wider group, sometimes, but not exclusively using an internal open source model. Now that git and GitHub have become the de facto version control system of choice, open source practices are becoming more and more common in-house.

1.6. Decentralized Data Management

Decentralization of data management presents in a number of different ways. At the most abstract level, it means that the conceptual model of the world will differ between systems. This is a common issue when integrating across a large enterprise, the sales view of a customer will differ from the support view. Some things that are called customers in the sales view may not appear at all in the support view. Those that do may have different attributes and (worse) common attributes with subtly different semantics.

This issue is common between applications, but can also occur within applications, particular when that application is divided into separate components. A useful way of thinking about this is the Domain-Driven Design notion of Bounded Context. DDD divides a complex domain up into multiple bounded contexts and maps out the relationships between them. This process is useful for both monolithic and microservice architectures, but there is a natural correlation between service and context boundaries that helps clarify and reinforce the separations.

As well as decentralizing decisions about conceptual models, microservices also decentralize data storage decisions. While monolithic applications prefer a single logical database for persistent data, enterprises often prefer a single database across a range of applications - many of these decisions driven through vendor's commercial models around licensing. Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems - an approach called Polyglot Persistence. Polyglot persistence can be used in a monolith, but it appears more frequently with microservices.

Decentralizing responsibility for data across microservices has implications for managing updates. The common approach to dealing with updates has been to use transactions to guarantee consistency when updating multiple resources. This approach is often used within monoliths. Using transactions like this helps with consistency, but imposes significant temporal coupling, which is problematic across multiple services. Distributed transactions are notoriously difficult to implement and as a consequence microservice architectures emphasize transactionless coordination between services, with explicit recognition that consistency may only be eventual consistency and problems are dealt with by compensating operations.

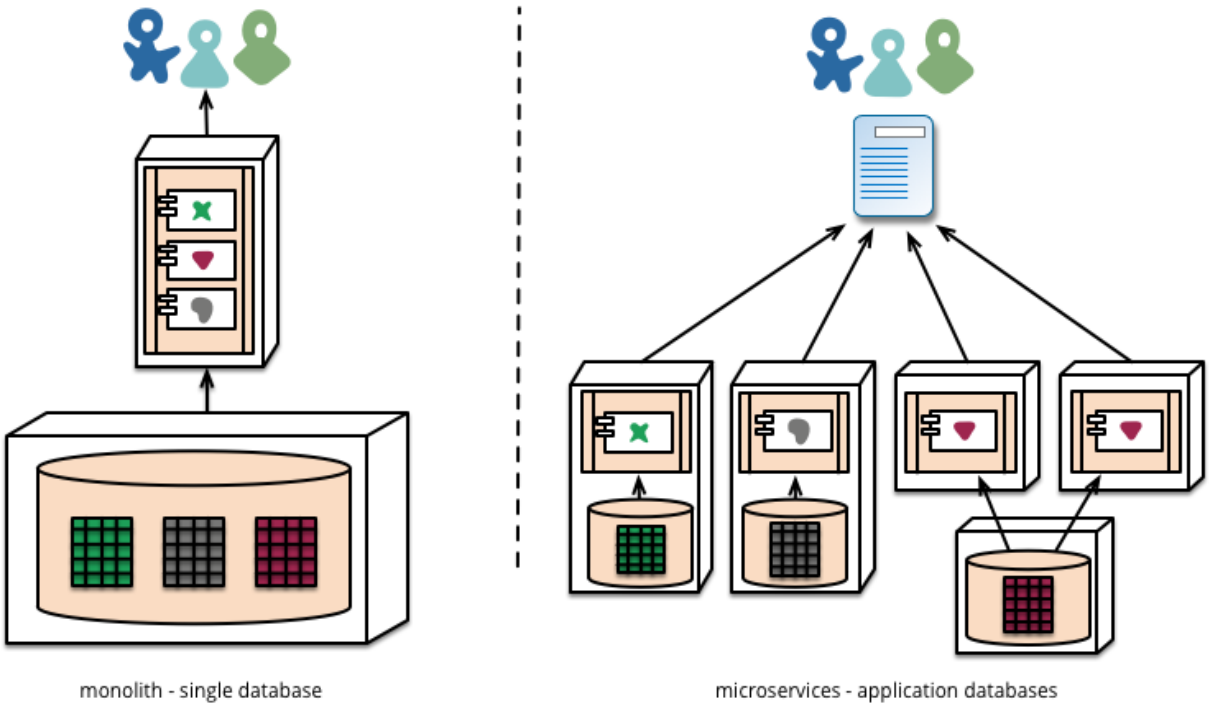

Figure 3: Polyglot Persistence

Choosing to manage inconsistencies in this way is a new challenge for many development teams, but it is one that often matches business practice. Often businesses handle a degree of inconsistency in order to respond quickly to demand, while having some kind of reversal process to deal with mistakes. The trade-off is worth it as long as the cost of fixing mistakes is less than the cost of lost business under greater consistency.

1.7. Infrastructure Automation

Infrastructure automation techniques have evolved enormously - the evolution of the cloud and AWS in particular has reduced the operational complexity of building, deploying and operating microservices. Many of the products or systems being built with microservices are being built by teams with extensive experience of Continuous Delivery and its precursor, Continuous Integration. Teams building software this way make extensive use of infrastructure automation techniques. This is illustrated in the build pipeline shown below. Promotion of working software 'up' the pipeline means automated deployment to each new environment.

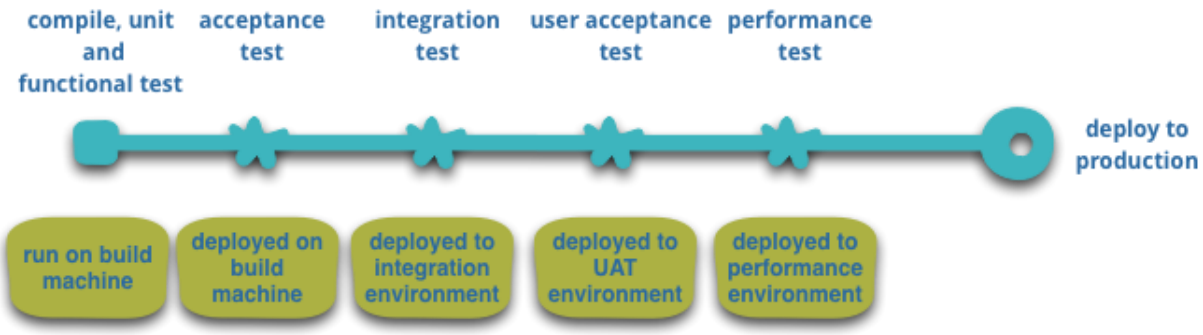

Figure 4: Basic build pipeline

A monolithic application will be built, tested and pushed through these environments quite happily. It turns out that once you have invested in automating the path to production for a monolith, then deploying more applications doesn't seem so scary any more. Remember, one of the aims of CD is to make deployment boring, so whether it's one or three applications, as long as it's still boring it doesn't matter. Another area where teams are using extensive infrastructure automation is when managing microservices in production. Operational landscape is huge in microservice architecture compared with monolithic architecture.

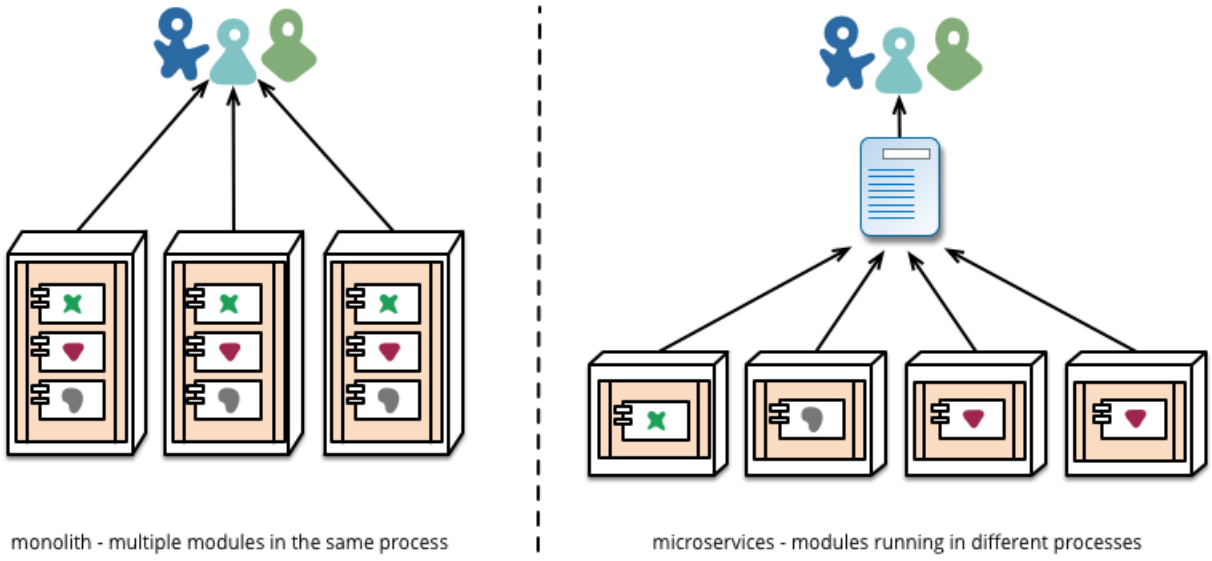

Figure 5: Deployment difference

1.8. Design for failure

A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services. Any service call could fail due to unavailability of the supplier, the client has to respond to this as gracefully as possible. This is a disadvantage compared to a monolithic design as it introduces additional complexity to handle it. The consequence is that microservice teams constantly reflect on how service failures affect the user experience. Project often induces failures of services and even datacenters during the working day to test both the application's resilience and monitoring.

Since services can fail at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore service. Microservice applications put a lot of emphasis on real-time monitoring of the application, checking both architectural elements (how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received). Semantic monitoring can provide an early warning system of something going wrong that triggers development teams to follow up and investigate.

Microservice teams would expect to see sophisticated monitoring and logging setups for each individual service such as dashboards showing up/down status and a variety of operational and business relevant metrics. Details on circuit breaker status, current throughput and latency are other examples we often encounter in the wild.

1.9. Evolutionary Design

Microservice practitioners, usually have come from an evolutionary design background and see service decomposition as a further tool to enable application developers to control changes in their application without slowing down change. Change control doesn't necessarily mean change reduction - with the right attitudes and tools you can make frequent, fast, and well-controlled changes to software.

Whenever you try to break a software system into components, you're faced with the decision of how to divide up the pieces - what are the principles on which we decide to slice up our application? The key property of a component is the notion of independent replacement and upgradeability - which implies we look for points where we can imagine rewriting a component without affecting its collaborators. Indeed, many microservice groups take this further by explicitly expecting many services to be scrapped rather than evolved in the longer term.

Putting components into services adds an opportunity for more granular release planning. With a monolith any changes require a full build and deployment of the entire application. With microservices, however, you only need to redeploy the service(s) you modified. This can simplify and speed up the release process. The downside is that you have to worry about changes to one service breaking its consumers. The traditional integration approach is to try to deal with this problem using versioning, but the preference in the microservice world is to only use versioning as a last resort. We can avoid a lot of versioning by designing services to be as tolerant as possible to changes in their suppliers.

2. Topologies

While there are literally dozens of ways to implement a microservices architecture pattern, three main topologies stand out as the most common and popular: the API REST-based topology, application REST-based topology, and the centralized messaging topology.

2.1. API REST-based topology

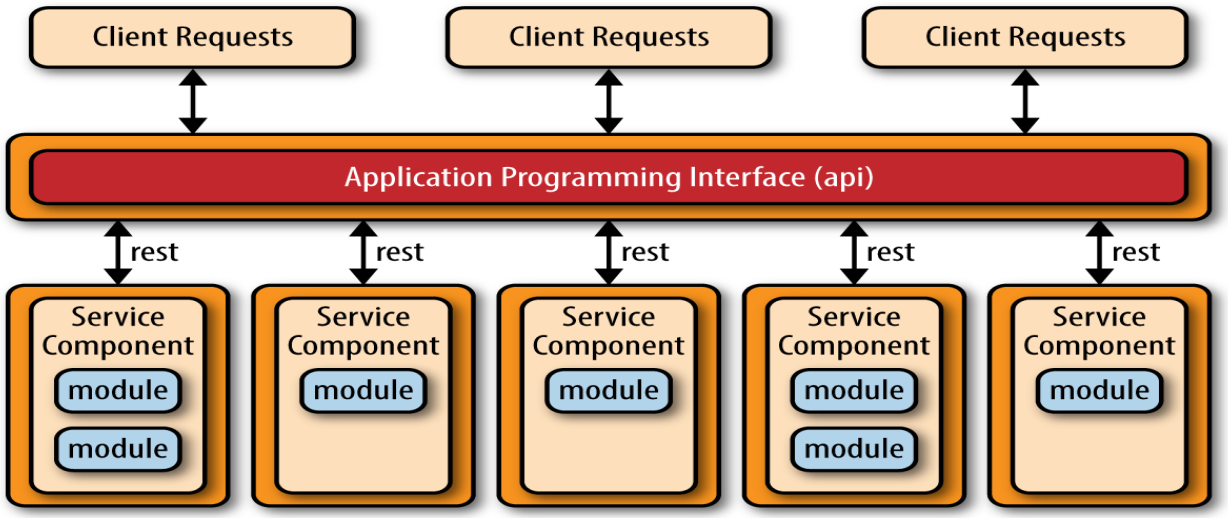

Figure 6: API REST-based topology

The API REST-based topology is useful for websites that expose small, self-contained individual services through some sort of API. This topology, which is illustrated in above figure (*Figure 6: API REST-based topology*), consists of very fine-grained service components (hence the

name microservices) that contain one or two modules that perform specific business functions independent from the rest of the services. In this topology, these fine-grained service components are typically accessed using a REST-based interface implemented through a separately deployed web-based API layer. Examples of this topology include some of the common single-purpose cloud-based RESTful web services found by Yahoo, Google, and Amazon.

2.2. Application REST-based topology

The application REST-based topology differs from the API REST-based approach in that client requests are received through traditional web-based or fat-client business application screens rather than through a simple API layer. As illustrated in below figure (*Figure 7: Application REST-based topology*), the user-interface layer of the application is deployed as a separate web application that remotely accesses separately deployed service components (business functionality) through simple REST-based interfaces. The service components in this topology differ from those in the API-REST-based topology in that these service components tend to be larger, more coarse-grained, and represent a small portion of the overall business application rather than fine-grained, single-action services. This topology is common for small to medium-sized business applications that have a relatively low degree of complexity.

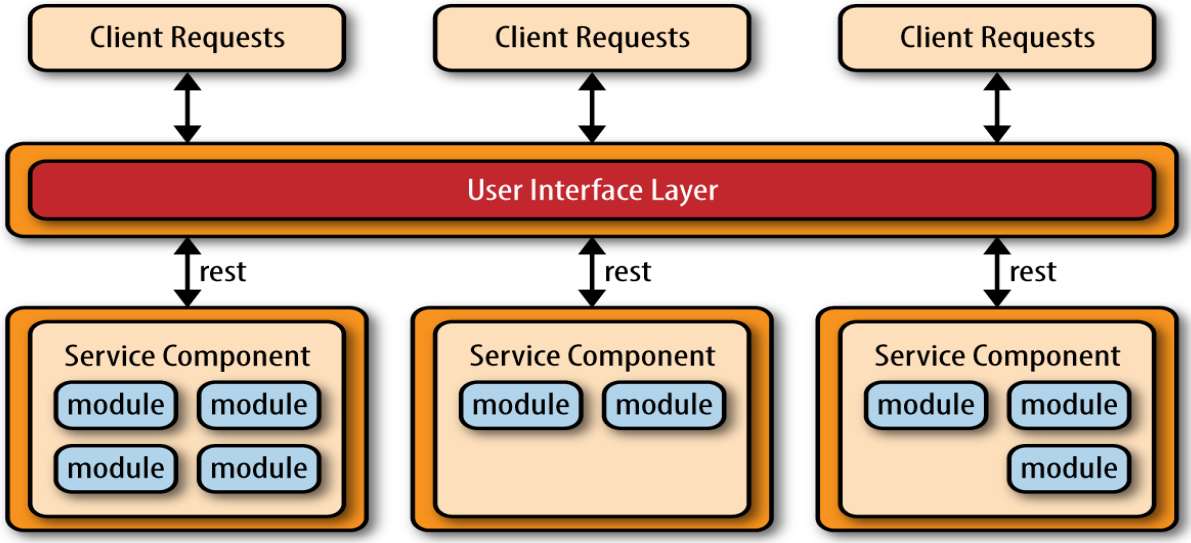

Figure 7: Application REST-based topology

2.3. Centralized messaging topology

Another common approach within the microservices architecture pattern is the centralized messaging topology. This topology is similar to the previous application REST-based topology except that instead of using REST for remote access, this topology uses a lightweight centralized message broker (e.g., ActiveMQ, HornetQ, etc.). It is vitally important when looking at this topology not to confuse it with the service-oriented architecture pattern or consider it “SOA-Lite." The lightweight message broker found in this topology does not perform any orchestration,

transformation, or complex routing; rather, it is just a lightweight transport to access remote service components.

The centralized messaging topology as illustrated in below figure (*Figure 8: Centralized messaging topology*), is typically found in larger business applications or applications requiring more sophisticated control over the transport layer between the user interface and the service components. The benefits of this topology over the simple REST-based topology discussed previously are advanced queuing mechanisms, asynchronous messaging, monitoring, error handling, and better overall load balancing and scalability. The single point of failure and architectural bottleneck issues usually associated with a centralized broker are addressed through broker clustering and broker federation (splitting a single broker instance into multiple broker instances to divide the message throughput load based on functional areas of the system).

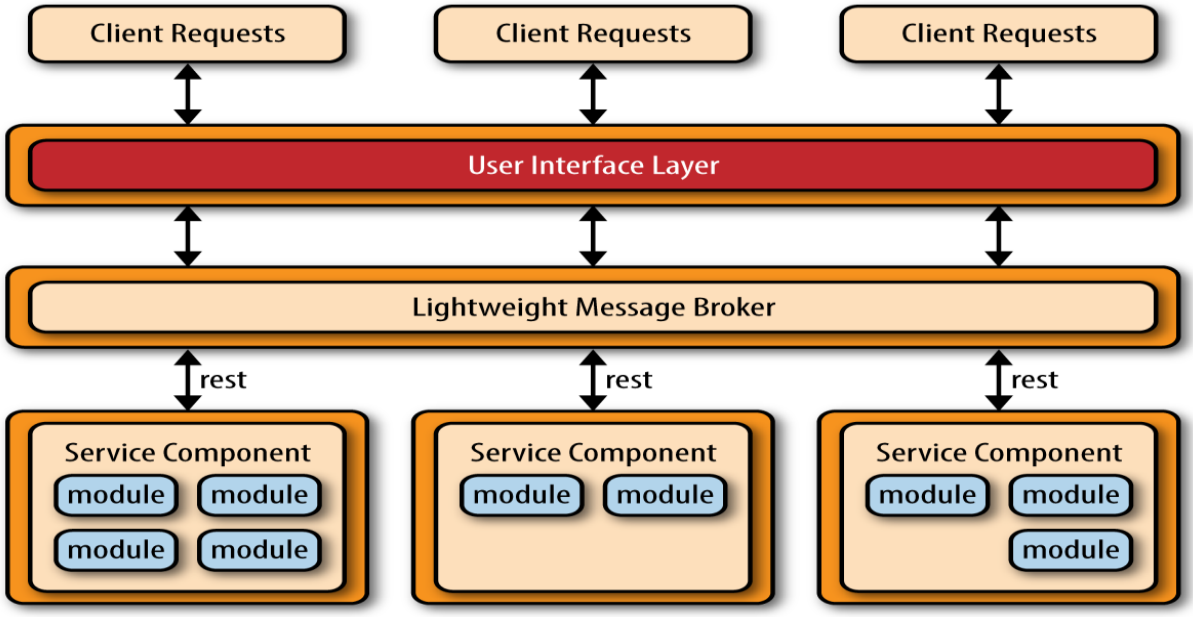

Figure 8: Centralized messaging topology

3. Considerations

3.1. Level of granularity

One of the main challenges of the microservices architecture pattern is determining the correct level of granularity for the service components. If service components are too coarse-grained you may not realize the benefits that come with this architecture pattern (deployment, scalability, testability, and loose coupling). However, service components that are too fine-grained will lead to service orchestration requirements, which will quickly turn your lean microservices architecture into a heavyweight service-oriented architecture, complete with all the complexity, confusion, expense, and fluff typically found with SOA-based applications.

If you find you need to orchestrate your service components from within the user interface or API layer of the application, then chances are your service components are too fine-grained. Similarly, if

you find you need to perform inter-service communication between service components to process a single request, chances are your service components are either too fine-grained or they are not partitioned correctly from a business functionality standpoint.

Inter-service communication, which could force undesired couplings between components, can be handled instead through a shared database. For example, if a service component handing Internet orders needs customer information, it can go to the database to retrieve the necessary data as opposed to invoking functionality within the customer-service component.

If you find that regardless of the level of service component granularity you still cannot avoid service-component orchestration, then it's a good sign that this might not be the right architecture pattern for your application.

3.2. Transaction

Because of the distributed nature of this pattern, it is very difficult to maintain a single transactional unit of work across (and between) service components. Such a practice would require some sort of transaction compensation framework for rolling back transactions, which adds significant complexity to this relatively simple and elegant architecture pattern.

3.3. Security

Compared to a monolithic application microservices pose enormous security challenges due to the increases in inter-service communication over the network. All of these interactions create an opportunity to outside entities to gain access to the system.

3.4. Others

One final consideration to take into account is that since the microservices architecture pattern is a distributed architecture, it shares some of the same complex issues found in the event-driven architecture pattern, including contract creation, maintenance, and government, remote system availability, and remote access authentication and authorization.

4. Advantages and Disadvantages

4.1. Advantages

Scalability: Individual service can scale as per need, there is now no need to scale all services together.

Agility: Changes in a particular micro service can be done very rapidly and can be deployed very quickly which makes it highly suitable architecture for ever changing business requirements meaning highly agile environment.

Availability: Even if one service fails, other micro services are highly available, and the failed micro service can be rectified very quickly with as minimal downtime as well.

Fault Tolerance: Even if one micro service has some faults with regards to say Database connection pool getting exhausted, there is a very clear boundary defined with regards to any fault

and unlike the monolithic way, other services operate smoothly and hence only a small part of the application is impacted instead of the entire application bogging down
Polyglot Persistence: Each Micro service can choose its own type of database based on the Use Case requirement. So, in general, application stack is not tied to a particular database.

Maintainability: For each business Service, a separate micro service is created. Thus, business code in a micro service is very easy to understand since it caters one business functionality. Also, since micro service caters to single business functionality, amount of code base is also reduced a lot, and this makes it highly maintainable.

Software Stack agnostic: Since a bigger application is decomposed into n number of smaller micro services, application is not tied to a single software stack and thus different software stacks can be used for different micro services.

Clear Separation of Business Concerns: Each Micro Service caters to a unique business functionality and thus there is a very clear separation of business concern between each one of them and thus each micro service can be built in a very robust way.

4.2. Disadvantages

Increased application complexity: One of the biggest downsides of moving from a single application package to many is a significant increase in application complexity. You now have to worry about many small applications instead of one. Documentation will also have to cover the new services as well as integration points. Test plans and code will have to cover both the smaller services as well as a full integration test plan every time you manipulate your services. Technical debt will also increase significantly, and intentional efforts must be made to make sure all team resources understand the smaller services that power your application.

Performance: As there will be lot of inter-service calls, performance may go down.

Developing distributed systems can be complex: Since everything is now an independent service, you have to carefully handle requests traveling between your modules. In one such scenario, developers may be forced to write extra code to avoid disruption. Over time, complications will arise when remote calls experience latency.

Deploying microservices can be complex: They may need coordination among multiple services, which may not be as straightforward as deploying a WAR in a container.

Testing: Testing a microservices-based application can be cumbersome. In a monolithic approach, we would just need to launch our WAR on an application server and ensure its connectivity with the underlying database. With microservices, each dependent service needs to be confirmed before testing can occur.

Persisting data across services: Multiple databases and transaction management can be painful. Communicating across services becomes more difficult when your databases are separated. While some great software packages exist to make inter-service communication possible you still have the same race condition issues and transactional processes become very difficult since the databases are specific to the services.

Health monitoring: With microservices, health of several servers needs to be monitored. No longer rely on a heartbeat monitor on web server to make sure application is still up and running.

Cross-cutting concerns: In a microservice architecture, either we need to incur the overhead of separate modules for each cross-cutting concern or encapsulate cross-cutting concerns in another service layer that all traffic gets routed through.

5. Pattern Analysis

The following table contains a rating and analysis of the common architecture characteristics for the microservices architecture pattern. The rating for each characteristic is based on the natural tendency for that characteristic as a capability based on a typical implementation of the pattern, as well as what the pattern is generally known for.

2.4. Overall agility

Rating: High
Analysis: Overall agility is the ability to respond quickly to a constantly changing environment. Due to the notion of separately deployed units, change is generally isolated to individual service components, which allows for fast and easy deployment. Also, applications build using this pattern tend to be very loosely coupled, which also helps facilitate change.

2.5. Ease of deployment

Rating: High
Analysis: Overall this pattern is relatively easy to deploy due to the decoupled nature of the event-processor components. The broker topology tends to be easier to deploy than the mediator topology, primarily because the event-mediator component is somewhat tightly coupled to the event processors: a change in an event processor component might also require a change in the event mediator, requiring both to be deployed for any given change.

2.6. Testability

Rating: High
Analysis: Due to the separation and isolation of business functionality into independent applications, testing can be scoped, allowing for more targeted testing efforts. Regression testing for a particular service component is much easier and more feasible than regression testing for an entire monolithic application. Also, since the service components in this pattern are loosely coupled, there is much less of a chance from a development perspective of making a change that breaks another part of the application, easing the testing burden of having to test the entire application for one small change.

2.7. Performance

Rating: Low
Analysis: While you can create applications implemented from this pattern that perform very well, overall this pattern does not naturally lend itself to high-performance applications due to the distributed nature of the microservices architecture pattern.

2.8. Scalability

Rating: High

Analysis: Because the application is split into separately deployed units, each service component can be individually scaled, allowing for fine-tuned scaling of the application. For example, the admin area of a stock-trading application may not need to scale due to the low user volumes for that functionality, but the trade-placement service component may need to scale due to the high throughput needed by most trading applications for this functionality.

2.9. Ease of development

Rating: High

Analysis: Because functionality is isolated into separate and distinct service components, development becomes easier due to the smaller and isolated scope. There is much less chance a developer will make a change in one service component that would affect other service components, thereby reducing the coordination needed among developers or development teams.