A Point of View

Microservices: The Building Blocks of an API Framework

Abstract

Traditionally, applications in the financial services domain are monolithic in nature. Many of these applications are built on legacy technologies, and as they grow in size and scope, they become difficult to manage. Also, legacy infrastructure lacks agility, and often increases the time to market new offerings. In order to retain their competitive position, it is essential for banks to quickly enable digital services by rendering necessary functionalities through smartphone apps. Application programming interface (API) tools help banks and financial institutions facilitate the development of applications.

This article takes a look at the challenges posed by monolithic applications in quickly introducing digital offerings. It also highlights how financial institutions can utilize a microservices framework for APIs, thus enabling the development of agile, scalable applications that guarantee superlative performance.

Introduction

Most banking applications are multi-tiered and modularized. With multiple teams working on such monolithic applications growing in size and scope, change management poses notable challenges. Agile development and continuous delivery are considerably difficult to achieve when components are tightly coupled. Changes to one section of an application could impact other areas, owing to dependencies. This necessitates complete application testing every time a change is made, thus resulting in a longer development lifecycle.

Moreover, the performance of the application could be affected if the performance of any of the components is below par. Scaling a monolithic application therefore requires the entire application to be rebuilt, which is inefficient, and entails additional resources. Further, such applications incur additional infrastructure costs even during off peak hours, as opposed to lean applications that help enterprises save infrastructure costs. Another disadvantage is that monolithic applications make new technology adoption unviable, both in terms of cost and time, which forces organizations to stick with obsolete technologies.

One way of tackling the problems posed by monolithic applications is to migrate to a microservices architecture. This involves breaking down an application into multiple smaller service units, where each unit or microservice is a mini application delivering specific functionalities of the main application. By splitting the application into manageable chunks, microservices result in a modular architecture, which facilitates faster development and easier maintenance. Dedicated teams can independently develop and deploy each service, and choose the technology best suited for the functionality to be delivered. Most importantly, this architecture allows a service to be scaled independently of other services.

Another key advantage of microservices is that they offer an API endpoint that can be accessed through the internet. APIs, in combination with microservices, help expose legacy functionality on distribution channels to enable developers to create innovative apps, or add new functionality to existing apps. Moreover, with different departments working on applications, chances of implementing similar or identical functionality are high, especially in the absence of proper governance at the enterprise level. This means additional resources being spent on existing functionality. Enterprises can have better control by creating a central repository of APIs covering all the functionalities and features.

Components of a Microservice

A microservice is defined based on the ubiquitous language, and the bounded context of the requirements. Each microservice does its own work, and multiple command or query services can be composed to deliver the complete functionality. The core components of a microservice are:

RESTful API component: This is a popular HTTP-based protocol that exposes the functionality to other services. Service contracts are established for the bounded context as per Rest API specifications, and the RESTful API component handles interaction with services.

Logic component: Each microservice functions as a disparate application with its own business logic relevant to the bounded context. The logic is identified as part of the initial requirements, or as part of moving the logic from the monolithic application.

Event handling component: Synchronous processing of information degrades the performance of the application. Generally, in an application, information can be processed in parallel through asynchronous processing. Asynchronous processing is achieved by adopting an event-driven architecture. Microservices publish events about the data change. Interested services perform the relevant functionality when they receive the event. Events are exchanged through the message broker, which decouples producers and consumers. If a consumer isn't available to process, then the message broker will queue the event until processed by the consumer.

Data component: This is where the data is stored for the microservice. Ideally, microservices do not share data as each has its own data store. The data is exposed through the service API of the microservice. Separating the data store this way ensures high performance.

Data storage involves centralized transaction management in the application of all the participating components. Microservices generally invoke multiple remote services, and transactions cross the service boundary. Ideally, transactions should not cross a service boundary, but this may not be the case in complex systems. If an application demands high consistency of information, a microservices architecture may be unsuitable, as it won't fulfill the application demand.

One approach for transaction management is for the orchestration layer to control transactions across multiple services, and roll back the components in case of failures. Another approach is to use the eventual consistency approach. In this approach, the system guarantees that details captured from the user will be persisted accurately even during system failures, so that users don't have to worry about losing data. Data users will be provided the right content, and the application prevents users from proceeding further in case of inconsistent data.

Configuration component: Configuration details required for microservices are externalized to allow changes to the configuration without having to rebuild and deploy. Microservices refresh the details whenever required.

The supporting components are:

Deployment component: Microservices are packaged as container images and deployed to the destination environment. Containers allow spawning and stopping of instances based on various conditions. A container encapsulates the details of the technology used to build the service, and enables effective monitoring and maintenance of microservices.

Documentation component: The Swagger API specification is used to document the REST APIs that are exposed by microservices. It specifies the format (URL, method, and representation) to describe REST web services. Swagger provides tools to generate API documentation and client SDKs, and allows testing of the API.

Service discovery component: Microservices generally have multiple instances running at any point of time and they are dynamic in terms of the number of instances. Clients need a static URL to avoid changing the client as and when services come up or go down. Service discovery component helps provide a single URL to the client and allows the load balancer to provide the service by connecting to the correct service.

Proposed Approach for Migrating to Microservices

Migration to a microservices architecture is a lengthy process. Rewriting monolithic applications with complex business logic is a time consuming activity. Effecting normal functional changes is difficult till the migration is completed. However, to make the migration process smoother, banks can start by migrating the frequently used functionalities of the application. Once the team is comfortable, and the framework is finalized, the various modules can be moved to the microservices architecture according to a defined priority. We present a step-by-step approach of migrating to a microservices architecture:

Finalize the scope of each microservice: Functional decomposition of the application into bounded contexts is the key to breaking down the complexity of an application. It results in loose coupling between the functional components, and a high degree of cohesion. By splitting the monolithic application into multiple smaller business boundaries called subdomains, multiple services can be composed to achieve a specific functionality. Each small unit forms the bounded context. To invoke another bounded context, messages are exchanged with other bounded contexts using the API.

Assign functions and features to the microservice: Adding a code to a monolithic application to implement new functionality will make the application bigger. Any new functionality should therefore be implemented using the microservices architecture. Enterprises can start with frequently changing or heavily loaded modules that need to be scaled. Use the concept of bounded context to extract modules into microservices. As microservices replace modules in the monolithic application, integrating them with the application becomes essential.

Finalize interactions between microservices: Since a microservice-based application runs on several machines, it is essential to implement a communication mechanism to enable interaction between services. Such interactions may be between microservices of the same application, or with the services of the internal or external ecosystem.

Decomposing an Application

The process of decomposing a monolithic application into microservices is complicated, therefore several aspects need to be considered to ensure successful partitioning.

Interfaces

User interface: The user interface needs to access the database for displaying the details. The data could be static data. Caching mechanism can be implemented to render static data across services. For functionality related information display, the 'query' part of the service can be used and for persisting, and the 'command' part can be used.

Upstream and downstream systems: Integration with upstream systems can be achieved through rule-based routing technologies like Apache Camel or AKKA, which enable the implementation of enterprise integration patterns. These tools use domain specific language to configure routing rules to convert messages from one format to another. Integration with downstream systems can be achieved in the same manner as for upstream systems.

Synchronous or asynchronous communication: Communication between services can be synchronous or asynchronous. For synchronous messaging, both client and server must be simultaneously available, and failure of either will cause system failures. For asynchronous messaging, Advanced Message Queuing Protocol-based message broker is used, wherein messages queue up when the system is not available. This decouples message producers and consumers. The message broker stores the messages until the system is available to process it.

Business logic: Each microservice executes the business logic for a particular functionality. The business logic should align with the requirements, and the ubiquitous language defined for the bounded context.

Business rules: Business changes necessitate modifications to business rules. Externalizing business rules from the application helps banks make changes to the rules easily. Utilizing a rules engine helps externalize business rules from the application.

Process orchestration: While splitting a monolithic application into microservices provides enterprises the necessary degree of agility and flexibility, the business goal is to package microservices into an application, and deploy it to deliver the required functionality. Where multiple services need to be invoked to deliver the functionality, the aggregator microservice design pattern is used. The entry point would be the aggregator, which invokes other services, aggregates, and returns the result.

Service or library

Sharing code between services could lead to tight coupling between components. If a functionality is related to a requirement needing its own persistence, and its own database, it needs to be exposed as a service. However, general functionalities like logging, email messaging, and caching can be exposed as a library, which is part of each service.

Information

Data persistence: Database selection affects the performance of an application. Depending on the requirement, and in cases where modules do not fit well with the relational database, the polyglot persistent choices should be considered.

Data caching: Reference data is continuously referred to but not updated frequently. Storing reference data as a table impacts the performance of the application. To improve the performance, caching mechanisms like Ehcache, Redis, and MemCached can be used as they enable quick retrieval of data from in-memory caches.

Session caching: Session data should be cached so that the application is able to access it at a later time. Using distributed session management tools across all the services will enable efficient retention of session data.

Statelessness: Services should be stateless to allow scalability of the application. The state management occurs on the client site, and the client should send the required information on each request so that the server can fulfill the request.

Security

Authentication and authorization: Traditional modes of identify and access management cannot be used for microservices, therefore an authentication service should be used. The authentication service should identify the user when the service is accessed for the first time, and then cache the details. For each subsequent access to the API, the service must check with the authentication service and continue only after establishing the authenticity of the user. Authorization should be handled by creating an authorization service that verifies whether the user has the right to access the requested resource.

Information security: Decomposing applications into microservices has some adverse security implications. Microservices expose APIs over the network that can be accessed by anyone if the URL is known. Certificate-based authentication should be adopted to make the services secure. An API management solution should be considered as it comes with features that assist in enhancing security.

Conclusion

APIs are crucial to digital transformation initiatives, and the microservices architecture is well positioned to support the move. The traditional banking model is inadequate to meet consumer demand for simplicity, convenience, and superior experience; while ensuring regulatory compliance and meeting the requirement for an open banking ecosystem. Financial services players should look upon these changes as an opportunity to rearchitect their IT infrastructure into a microservices framework with well-defined APIs.

A comprehensive API framework will drive innovative business models and facilitate the shift toward an open banking ecosystem, which will promote growth in the industry through increased collaboration between incumbent banks and new players. The success of the migration cannot be guaranteed because of the magnitude of the work involved. Therefore, it is advisable to start small. Once the teams are comfortable with the processes detailed above, a small number of services can be identified for migration for every release, so as to not disrupt the main release.

About the Author

Sanjay Anandkumar

Sanjay Anandkumar is a Solution Architect with the Banking and Financial Services business unit at Tata Consultancy Services (TCS). He has over 20 years of experience, and possesses vast technical experience in architecting, re-engineering, designing, and developing applications on various J2EE technologies; mobile applications; and various database technologies for banking and financial services firms. Sanjay has a Bachelor's degree in Computer Science from Bangalore University.

TATA CONSULTANCY SERVICES

Experience certainty.

About TCS' Banking and Financial Services Business Unit

With over four decades of experience in partnering with the world's leading banks and financial institutions, TCS offers a comprehensive portfolio of domain-focused processes, frameworks, and solutions that empower organizations to respond to market changes quickly, manage customer relationships profitably, and stay ahead of competition. Our offerings combine customizable solution accelerators with expertise gained from engaging with global banks, regulatory and development institutions, and diversified and specialty financial institutions. TCS helps leading organizations achieve key operational and strategic objectives across retail and corporate banking, capital markets, market infrastructure, cards, risk management, and treasury.

TCS has been ranked No. 1 in the 2016 FinTech Rankings Top 100 of global technology providers to the financial services industry, by both FinTech Forward (a collaboration of American Banker and BAI) and IDC Financial Insights. TCS has also been recognized as a 'Leader' in Everest Group's 2016 PEAK Matrix™ report for Capital Markets Application Outsourcing and Banking Application Outsourcing.

Contact
Visit TCS' Banking and Financial Services unit page for more information
Email: bfs.marketing@tcs.com
Blog: Drive Governance

About Tata Consultancy Services Ltd (TCS)

Tata Consultancy Services is an IT services, consulting and business solutions organization that delivers real results to global business, ensuring a level of certainty no other firm can match. TCS offers a consulting-led, integrated portfolio of IT and IT-enabled infrastructure, engineering and assurance services. This is delivered through its unique Global Network Delivery Model™, recognized as the benchmark of excellence in software development. A part of the Tata Group, India's largest industrial conglomerate, TCS has a global footprint and is listed on the National Stock Exchange and Bombay Stock Exchange in India.

For more information, visit us at **www.tcs.com**

IT Services
Business Solutions
Consulting

TCS Design Services | M | 10 | 16