Application development using MicroServices

Karanbir Gill [1], Praneeth Ramesh [2], and Jinan Fiaidhi [2]

[1]Lakehead University.
[2]Affiliation not available

October 30, 2023

Abstract

Application development has been in a trend for a while in which all small services can be scaled, deployed and tested independently. In this paper we have discussed the micro service architecture and moreover how to implement a simple set of micro services and integrating it with a memory database.

Application Development using Microservices

Karanbir Singh Gill Department of Computer Science*Lakehead University*Thunder Bay, Canada kgill7@lakeheadu.ca

Praneeth Ramesh Department of Computer Science*Lakehead University*Thunder Bay, Canada pramesh3@lakeheadu.ca

Jinan Fiaidhi Department of Computer Science Lakehead Univeristy Thunder Bay, Canada jfiaidhi@lakeheadu.ca

Abstract — **Applications developed using microservices has been trending in recent years as it provides an architecture where each service is small and can be deployed and scaled independent of each other. Further, it is easy to convert monolithic legacy software into flexible microservices as they can be done for each service independently and have it tested by integrating it into the monolithic software. Microservices can provide scalable, flexible, agile and reliable software that can be useful for many applications and business ventures. In this research paper, we have discussed the microservice architecture, how to implement a simple set of microservices and integrating it with a in memory database.**

Index Terms—**Microservices, Database, Scalable, Services, Monolithic Architecture.**

Introduction

Application development has been the backbone of the computer revolution. Information systems have traditionally been developed with the help of horizontally scalable transactional database management system built using monolithic software. The main intention of the microservice architecture is to overcome this deficit in the monolithic architecture and make the system more scalable. In a microservice architecture, many collaborating microservices work together to form the application. These collaborating microservices have no centralized control and based on the business requirements a full-stack implementation of any language and hardware can be used to implement the business logic.

Microservice architecture can be defined as an approach of developing a suite of small services working on a single application. These services can communicate with each other using simple light-weight protocols such as HTTP resource API and each service runs independently in its own process.

Monolithic architecture on the other hand, is a single codebase that implements all of the services required for the application and communicated with the external resources using or consumers via interfaces such as Web Services, HTML pages or REST API. The applications developed using monolithic architecture can include many different services that are tightly coupled in a monolithic codebase which may cause difficulty in working together as a team as code conflict is on the main issues that developers face while working as a team on a single monolithic software using version control.

The microservice architecture relies on technology heterogeneity, where each service can use different technology while still being able to be seamlessly integrated to fulfill business requirements. If a microservice fails the entire system does not fail only the particular service that the microservice provides will fail and the rest of the application will still be up and running whereas, in monolithic architecture if a service fails the entire system bugs out and unusable. The biggest and most useful advantage of the microservice architecture is that of its scalable nature. Only the services that need to be scaled can be scaled whereas in a monolithic architecture the entire application needs to be scaled which can cause more hardware usage. Microservice architecture is adapted highly in the software world due to its ease of deployment as it can be integrated with any deployment service such as cloudfoundry or docker without affecting the performance of other services. The microservice architecture can be formed in such a way that multiple people can work on different parts of the application without affecting the security policy as the number of people working on a service can be contained.

In this paper, we focus mainly on developing a simple application using the microservice architecture while dwelling on the details of the service discovery mechanism, eureka server and integrating with an in-memory database called H2.

background

Monolithic Architecture (MA) was the standard approach to software development, used by major corporations such as Amazon and eBay in the past. For MA the functions are embedded in a single application. Monolith applications have their own advantages if not complicated; ease of development, testing and deployment. But as the application starts to get more complex, the monolith structure increases in complexity, becoming a massive piece of software that is hard to handle and scale up.

Microservices could be seen as a technique for the development of software applications that, by inheriting Service-Oriented Architecture (SOA)-style principles and concepts, enable the structure of a service-based application as a set of very small, loosely coupled software services. Using the composition of small services, microservices architecture can be seen as a new approach for programming applications, each running its own processes and communicating through some lightweight mechanism. Microservices are very small (micro) in terms of their contribution to the application not because of their code lines. Boundary contexts, versatility and modularity are core features of microservices applications. In this regard, microservices is a popular trend in software architecture, emphasizing the creation and design of highly maintainable and scalable applications.

Related work

In [1], Hasselbring et al. discusses the vertical decomposition into self-contained systems and appropriate microservice granularity as well as coupling, integration, scalability and microservice monitoring at otto.de.

While growing agility to more than 500 live deployments per week, high reliability is achieved through continuous integration and delivery, through automated quality assurance.

In [2], Al-Debagy et al. compares the two architectures; monolithic and microservice, where monolithic architecture demonstrated better throughput efficiency by 6 percent compared to architecture of microservices. The scenario of load testing posed no major difference between the two architectures. Lastly, microservices applications developed with a variety of service discovery technologies, such as Consul and Eureka, showed that Consul applications produced better performance results.

In [3], De Lauretis, introduces a strategy that help in migration of monolithic architecture to a microservice architecture. Through this migration approach, the newborn framework can take advantage of a range of benefits provided by architecture of microservices, such as scalability and maintenance.

In [4], Fetzer describes a microservice-based system that builds trustworthy networks in addition to legacy hardware and software modules, ensuring the integrity, confidentiality and proper execution of microservices using secure enclaves.

proposed architecture

The Service Discovery Mechanism is an important part of the microservices architecture, all microservices need to be able to be registered to be used. There are two types of service discovery mechanisms, Client-side discovery and Server-side discovery.

In Client-Side discovery, the client queries the service registry to determine the available services using load balancing mechanism. It works with service registry to load balance requests across the available service instances. The disadvantage of client-side discovery is tight coupling of Client with Service Discovery as the service discovery logic needs to be written at the client side.

Fig 1. Client-Side Discovery

In Server-side discovery, the client queries the discovery service via an abstraction layer, that queries the service registry and routes the HTTP request to an available service instance. The advantage of server-side discovery is that, the client does not need to write the discovery logic, which leads to an ideal loose coupling of client with the Service Discovery.

Fig 2. Server-Side Discovery

A key part of the microservice architecture is the service registry, it is a database of available service instances. It assures that its highly available and up to date with the network locations of the service instances. It must keep refreshing the network locations since, the network locations keep changing as the cache data becomes out of date.

In this project we are using Netflix Eureka for Service Registry. It provides a REST-API to register (POST) and query (GET), service instances. Netflix achieves high availability by running one or more Eureka servers in each available EC2 availability zones. It has an elastic IP address. DNS text records are used to store the Eureka cluster configurations. Netflix – Ribbon is an IPC (inter process communication) client that works with Eureka to load balance requests across available service instances. The @LoadBalanced Annotation configures the RestTemplate to use Ribbon, which has been configured to use the Eureka client to query service discovery and fetch available service instances.

Fig 3. Eureka Server.

To Enable the Eureka Server, the eureka server dependency should be added in the main configuration file (pom.xml).

The @EnableEurekaServer annotation is used to make the spring boot application act as a Eureka Server. To Register services with the Eureka Server, add dependency to the build configuration file (pom.xml) of the microservice that needs to be registered. The @EnableDiscoveryClient annotation is used in the main Spring Boot Application class file, to make the application act as a Eureka Client. We also need to add the URL of the eureka server in the application.yml file.

We then create the Producer Microservice, Register the producer microservice with the Discovery Service and use the @EnableDiscoveryClient annotation to activate the Netflix eureka server. Producer microservice contains the DTO, Controller Class, Application Class and Profile Repository.

DTO – Defines the objects that the microservice should return

Controller Class – Contains the @RequestMapping for the URL's.

Application Class – Starts the SpringApplication in a main method

Profile Repository – Acts as the interface to the database.

We then create the Consumer Microservice, Register the consumer microservice with the Discovery Service and use the @EnableDiscoveryClient annotation to activate the Netflix eureka server. Requests for Discovery Client instance of Producer microservice using a smart RestTemplate.

Consumer microservice contains the DTO, Controller Class, Application Class and Profile Repository. The Consumer Microservice also contains the Views of the Project. Views are JSP (Java Server Pages) pages that are used to visualize the data that has been accessed from the microservices.

The views of the project have been enhanced with the help of CSS (Cascading Style Sheets) to improve the aesthetic appeal of the Web Services. CSS has been added for the three views in the project namely the index, user profile and user details.

Database Connectivity

For Database connectivity between the microservices we are using H2 Database. H2 is a relational database management system written in Java. It can be embedded in Java applications or run in client-server mode. The software is available as open source software Mozilla Public License 2.0 or the original Eclipse Public License.

The H2 database has a very fast database engine, is open source, java based, supports standard SQL & JDBC API. It can also be implemented in either an embedded or server mode while also providing support for clustering. The H2 database provides strong security features. The POST gre SQL ODBC driver can also be used with the help of H2 database.

The H2 database can be a disk based in memory database, can provide read only capabilities and temporary tables. Multiple connections and table level locking can be implemented using H2. The H2 also includes a cost-based optimizer, using a genetic algorithm for complex queries, zero – administration. The result set obtained using H2 can be scrollable, updateable, large with external result sorting. H2 also provides encryption using AES, SHA-256 encryption for password encryption and SSL.

The H2 database supports MySQL features like multiple schemas, information schema, referential integrity, foreign key constraints, inline views, trigger, stored procedures, wide range of data types, sequence and auto increment columns, computed columns and is compatible with IBM DB2, Apache Derby, HSQLDB, MS SQL Server, MySQL, Oracle, and PostgreSQL.

The security features of H2 include a solution for the SQL Injection problem while user password authentication uses SHA-256 with salt for added security. For server mode connections, user passwords are never transmitted in plain text over the network. The remote JDBC driver supports TCP/IP connections over TLS. The built-in web server supports connections over TLS.

H2 can be implemented in either of three connection modes, Embedded Mode (local connection using JDBC), Server Mode (Remote Connections using JDBC or ODBC over TCP/IP, or Mixed Mode (local and remote connections at the same time).

In our research, we have used H2 in an embedded mode meaning the local connections are made using JDBC. In embedded mode, an application opens a database from within the same JVM using JDBC. This is the fastest and easiest connection mode. The disadvantage is that a database may only be open in one virtual machine (and class loader) at any time. As in all modes, both persistent and in-memory databases are supported. There is no limit on the number of databases open concurrently, or on the number of open connections.

In embedded mode I/O operations can be performed by application's threads that execute a SQL command. The application may not interrupt these threads, it can lead to database corruption, because JVM closes I/O handle during thread interruption. Consider other ways to control execution of your application. When interrupts are possible the async: file system can be used as a workaround, but full safety is not guaranteed. It's recommended to use the client-server model instead as the client side may interrupt own threads.

For H2 we need to initialize database with some fixed schema (DDL) and insert default data (DML) into tables before the application is ready is run business usecases. We can achieve this by putting sql files into resources folder (/src/main/resources/). We add the Schema.sql and Data.sql files to create the database entries namely table and data.

By default, the console view of H2 database is disabled. We must enable it to view and access it in browser. We can customize the URL of H2 console which, by default, is '/h2'.

References

1. W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* , Gothenburg, 2017, pp. 243-246.
2. O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)* , Budapest, Hungary, 2018, pp. 000149-000154
3. L. De Lauretis, "From Monolithic Architecture to Microservices Architecture," *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* , Berlin, Germany, 2019, pp. 93-96.
4. C. Fetzer, "Building Critical Applications Using Microservices," in *IEEE Security & Privacy* , vol. 14, no. 6, pp. 86-89, Nov.-Dec. 2016.