ISSN No:-2456-2165

REST API Applications Vs Monolithic Applications

Swanand Anand Kolekar
Master of Computer Applications
ASM IMCOST
Thane, India

Abstract:- This paper tackles the questions "Are REST API applications better than monolithic applications?" and "Is one better than the other?". This is answered via a comparative analysis of the architecture structure used within both the applications via which it is concluded that whenever a scalable application is to be implemented REST api applications are better than monolithic applications.

Keywords:- *REST; API;*

I. INTRODUCTION

There often comes a choice at the beginning of a new project where the project architecture is to be decided on which the scope of the project is dependent. The architecture style chosen to be implemented also affects how the application will behave when there is a need to scale it to meet the increasing business demands. In regards to this, this paper compares the architectural design of REST API applications to that of Monolithic applications to determine which one is suitable for the business needs that come along with the scope of the project in terms of scalability.

II. RESEARCH METHODOLOGY

A. Phase 1 : Concept Introductions

➢ *API :*

- API is an acronym for Application Program Interface.
- APIs are basically software programs or software intermediaries that enable the communication between two applications.
- An API is the middle-man which is used to retrieve information or perform an operation required by the consumer , this is known as an "API call" which when completed by the information or operation provider is delivered back to the consumer, this is known as a "response" to the API called by the consumer.
- An API also makes it so that the API call workings are not needed to be known by the consumer i.e. the resource retrieval and origin are hidden.
- API usage within an organization also ensures that the information and resources within the organization are shared while monitoring and controlling access to the same, which is an important aspect from a security standpoint.

➢ *REST :*

- REST is an acronym for Representational
- State Transfer.
- REST is a software architectural style created by computer scientist Roy Fielding.
- REST was developed to guide the behavior of the architecture of hypermedia systems which operate on an Internet-scale.

- For this REST has defined six architectural constraints namely:

✓ Client-server architecture.
✓ Statelessness.
✓ Cacheability.
✓ Layered system.
✓ Code on demand.
✓ Uniform interface.

The above mentioned six constraints when used on distributed hypermedia systems enhances their non-functional attributes, namely scalability, simplicity, modularity, visibility, and performance.

➢ *Microservices :*

- Microservices architecture is an architectural style used for developing software applications.
- The core essence of microservices architecture is the breaking down and separating a rather large scale application into smaller independent parts which create a plethora of opportunities.
- Microservices architecture is best suited for Rapid
- Application Development model also known as RAD model.
- Using microservices for software development makes it possible to develop deliver and deploy core functionalities that are independent of each other in a much more robust, efficient and at a faster pace as compared to the traditional monolithic applications.

➢ *Monolithic Architecture :*

- Monolith in regards to software means all in one piece. Monolith also means something that is way too big or unable to be changed which in the context of software translates to something that is tightly coupled.
- Monolithic architecture is the traditional way of developing a software in which all the components albeit independent or dependent by their core functionality are integrated in a way everything is connected and dependent.

ISSN No:-2456-2165

➢ *Tightly Coupled :*

- It is a system design and computing concept where all hardware and software components are linked together in such a manner that each component is dependent upon each other.
- An application based on tightly coupled architecture promotes interdependent code.
- Application functionalities based on this type of concept are dependent on each other regardless of the aim of the functionality.
- Monolithic applications are excellent examples of tightly coupled system design.

➢ *Loosely Coupled :*

- This is a software design and computing concept which is essentially the exact opposite of tightly coupled system design and computing concept.
- In a loosely coupled architectural concept the components of a system are connected in an independent manner and also each individual component in a loosely coupled architectural concept has a single function.

B. Phase 2: Comparative Analysis

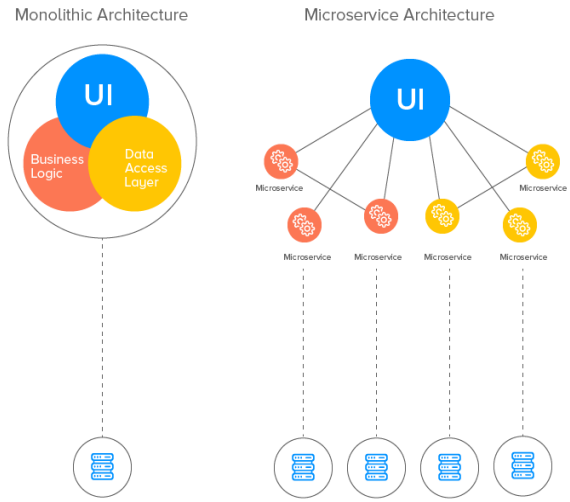

Fig1:- Architectural Layout Comparison

❖ *Monolithic Architecture:*

A typical single instance of monolithic architecture consists of four components :

1. A Database.
2. Database Access Layer.
3. Business Logic.
4. User Interface(UI).

- Database : This usually consists of many tables in a relational management system.
- Database Access Layer : This layer of the architecture is responsible for connecting the rest of the application to the database to perform CRUD operations.
- Business Logic : This layer holds the source code responsible for all the operations that the application is designed to perform and then gives results back to the User Interface.
- User Interface: UI is a representational layer by which a user interacts with the application and it also displays results of an operation or task completed by the application's business logic.

In monoliths the database is connected to a data access layer which is connected to business logic which is then finally connected to the user interface.

Since there is the above-mentioned connection flow, alternate layers do not communicate with each other independently, as they are always dependent on the layer between them. i.e. The User Interface can not connect with the data access layer without not first connecting to the business logic layer.

➢ ***Advantages:***

- **Simplicity of development:** The monolithic approach is the traditional way of building applications. All source code is located in one place which can be quickly understood in early stages.
- **Offline Application**s can be developed using monolithic architecture.
- **Simplicity of debugging:** The debugging process is simple because all code is located in one place. You can easily follow the flow of a request and find an issue.
- **Hassle free deployment:** Only one deployment unit (e.g. jar file) should be deployed. There are no external dependencies that are required to be plugged. In cases when UI is packaged with backend code you do not have any breaking changes. Everything exists and changes in one place.
- **Simplicity in onboarding new team members:** As all the source code is located in one place. New team members can easily get familiar with the application.
- **Low cost in the early stages of the application.** All source code is located in one place, packaged in a single deployment unit, and deployed. There is no overhead neither in infrastructure cost nor development cost.

Even though monoliths present to be a simple and effective software architectural decision, some serious drawbacks/disadvantages which are listed below present themselves as the monolith grows in size.

➢ ***Disadvantages:***

- **Slow speed of development:** For a monolith that contains a lot of components. If each component in this monolith is covered with tests that are executed for each code pull request. Even for a small change in a source code, one could be waiting a long time (e.g. a few hours) for the request to succeed. If for some reason the request fails one would have to wait all over again for it to succeed.

- **Spaghetti code:** As monoliths increase in size it is observed that more often than not there will be some spaghetti code in at least a few places within the project. As a result, the system becomes harder to understand especially for new team members.
- **Code ownership cannot be practiced:** As the system is growing. The next logical step is to split responsibilities between several teams. However, since monoliths have tightly coupled code there are no boundaries between those services. One team can affect another.
- **Code reusability:** A block of code once created in monoliths is tailored to fit the business requirements for that application which makes having reusable code highly unlikely.
- **Lengthy Testing:** Even a small change can negatively affect the system. As a result, the regression for full monolithic service is required.
- **Performance issues:** As a single database is used for all the services. Performance issues are quite evident even after various types optimization (e.g. database query optimization).The size of the application can slow down the start-up time.
- **Inevitable infrastructure costs:** To make sure the performance issues are mitigated to the maximum the obvious approach is to scale the whole infrastructure to support the heavy work-loads and operations, to achieve this upscaling of infrastructure additional costs will be incurred.
- **Legacy technologies:** Upgrading and migrating the technologies originally used in development is difficult and can often prove just outright impossible due to the tightly coupled nature of the codebase of used technologies. Not being able to upgrade or migrate to newer versions or technologies imposes a huge risk factor on longevity of the monoliths.
- **Rigidity:** As monoliths grow and age, newer technologies/tools that might be more efficient cannot be adopted due to the tightly bound codebase and nature of monoliths.
- **Problems with deployment:** Even a minor change requires the whole redeployment of the monolith which causes application downtimes which then in turn causes revenue loss as well as data loss in the sense that no new data is being added as the application is down for redeployment.

Due to these factors continuous deployment of the monoliths is not viable.

❖ *REST API Architecture:*

APIs based on web services that follow the REST architectural constraints are called RESTful APIs.

RESTful APIs transfer the representation of the state of the resource to an endpoint when a request is made by a client. This representative information also referred to as resource/data can be delivered by many formats via HTTP, namely JSON (javascript object notation), HTML, PHP, Python or just simple plain text.

Currently JSON is the most preferred way of delivering data as it is readable by both humans and machines.

RESTful APIs that are based on HTTP have the following characteristics :

- An URI, such as http://example.restapi.com/
- Standard HTTP methods(GET, POST, PUT and DELETE).
- A media type (e.g. application/json, application/xml, application/vnd.github.v3+json, etc.).
- An endpoint at the end of URI, such as /api/books.

RESTful APIs use microservice architecture.

In REST API whenever there is a function or task that needs to be executed which is done by a user interacting with the user interface. An API call is triggered consisting of an URI with an endpoint which is then redirected to the specific microservice.

❖ *Microservice Architecture:*

A microservice architecture consists of many small (micro)services; each of these services can have their own independent databases if the application structure demands it.

In microservice architecture each microservice is responsible for a single function and if one of these functions is dependent on another the dependent microservices communicate with each other to perform the required task while maintaining independent state from other non-related microservices.

➢ ***Advantages :***

- **Loosely coupled code:** The biggest advantage of microservices is that each service is independent except for communication points. As a result, each component can be developed by a different team.
- **Limited service scope:** Since services are usually small, they are usually easier to understand.
- **Simple Debugging:** debugging process is simple in the scope of a single service.
- **Flexibility:** A service can operate on its own database which can be RDBMS, NoSQL, both or none of them. Also, services can be written in different programming languages.
- **Code reusability:** A created service has very little business logic constraints so the service can always be used in other applications with just a little bit of tweaking which makes development of related applications much faster.
- **Scalability:** This is one of the major advantages of this architecture as components can be scaled upwards or downwards depending on the load and optimized independently. Performance characteristics do not have limits. For example, an application can have few instances of one service and many instances of other service.
- **Cost-effective:** A RDBMS database can be replaced by a single purpose NoSQL database for a particular service. Because of that, microservices can also be cost-effective for large applications.

ISSN No:-2456-2165

- **Modularity:** Microservices are small. Migration to a new platform version can be very simple. For instance, migration from a lower Java version to a higher version.
- **High speed of development:** A lot of frameworks are optimized to be used as microservices (e.g. Spring Boot). Services are small, consequently, the execution is fast.
- **Efficient and fast deployments:** Bugs and features can be fixed and deployed in a single service. The deployment time is small due to the size of the services. Since each service is designed to perform a specific function, rapid deployment of features and changes is possible.

➢ ***Disadvantages :***

- **Debugging:** While working in the scope of a single component, everything is simple. However, the architecture as a whole is complex. A single request can operate on multiple services, tracking this request during debugging can sometimes be difficult.
- **Testing:** Testing is challenging because of the distributed components.
- **Deployment:** Services can be deployed separately. However, they usually communicate somehow. Interaction contracts between services have to be strictly defined. A change in one service should not break other services.
- **Cross-cutting concerns:** Security, logging, monitoring, configuration, etc. should be implemented in each service.

C. Phase 3 : Discussion

Whenever monolith applications grow in size and complexity :

- There often arises a situation where no single developer (or group of developers) understands the entirety or working of the application.
- Limited to non-existent component reusability is observed.
- Scaling monolithic applications is costly and challenging.
- Repeated deployment of monolithic applications causes revenue and data loss.
- By definition, monolithic applications are implemented using a single development stack of technologies, this hampers the integration of using a tool that will be better suited for the job which is outside the development stack.

When REST API application is considered for scalable and complex applications instead of monolithic application following points are evident:

- Achieving scalability is easier in REST API applications.
- Relative costs of scaling are also lower.
- Constant deployments of components and updates is only possible here without sacrificing major revenue and data loss as services not being updated are not affected.
- Understanding services involved is much simpler for new developers as each service(or small group of services) is designated with a single function.

Some Companies that migrated from monoliths to microservices realizing that monoliths are not scalable are Amazon, Netflix, Ebay, Uber.

III. CONCLUSION

In this paper REST API application is compared with the traditional monolithic application via a comparative analysis of their architectural styles to determine if one is better than the other. To which it can be concluded that it is viable to adopt a REST API based application if at all an application is to be complex and/or is required to have scalability to integrate future developments and features, whereas if an application is supposed to be simple and lightweight, developed in regards to a very specific subject domain which will have minimal to low scalability requirements in the future only then the monolithic application seems a viable choice.

REFERENCES

[1]. REST. [Online]: https://en.wikipedia.org/wiki/Representational_state_transfer
[2]. Microservice vs Monoliths. [Online]: https://www.mulesoft.com/resources/api/microservices-vs-monolithic
[3]. Anton Kharenko. Monolithic vs. Microservices Architecture. [Online]: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59
[4]. Oleksii Dushenin. How To Understand Microservices Architecture. [Online]: https://datamify.medium.com
[5]. Carlos. Migrated from monoliths to microservices. [Online]: https://www.kambu.pl/blog/companies-that-migrated-from-monolith-to-microservices/