LTIMindtree

MICROSERVICES ARCHITECTURE FOR MODERN DIGITAL PLATFORMS

A white paper by Dr. Shailesh Kumar Shivakumar, LTIMindtree Limited

Microservices Architecture is becoming the mainstream services-based integration model and the de-facto standard for services development for enterprise applications. As enterprise applications tend to become complex, demanding on-the-fly scalability and high responsiveness, microservices play a crucial role in fulfilling these criteria. Enterprises can realize a strategic vision of an API-based, loosely-coupled, scalable and flexible platform architecture with containerized microservices.

In this white paper, we discuss the salient points of microservices-based solution architecture and shed light on the architecture patterns, real-world use cases and microservices best practices.

AN INTRODUCTION TO MICROSERVICES

Microservices are modular, autonomous and logical units of functionality that are independently deployable and scalable. Its architecture involves decomposing the complex business functionality into modular and granular microservices that can be quickly and continuously developed, deployed and maintained. Microservices architecture offer lightweight and stateless services architecture that comes handy in modern digital architecture.

The complex application can be built as a suite of indendently deployable microservices. Modern digital applications employ lightweight models at all tiers to enable high performance and scalability:

- **User Interface:** The user interface layer is built using JavaScript framework such as ReactJS or Angular to provide responsive and lightweight user interface.
- **Integration:** The integration is mainly done with REST invocations with JSON being the payload structure. This again provides lightweight integration method.
- **Business services:** Business services are implemented as microservices (for ground-up development) or as microservice wrapper (for legacy applications).
- **Security:** Lightweight token-based security is used for authentication and authorization.

Martin Fowler and James Lewis define microservices as ***"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 are 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."***

DRIVERS AND THE MOTIVATION FOR MICROSERVICES ARCHITECTURE

When enterprises embark on a digital transformation journey, the enterprise architecture team aims to design a flexible and extensible architecture. Given below are the key drivers for microservice architecture:

- Decouple systems of engagement (SoE) with systems of record (SOR) through well-defined API contracts. This provides the flexibility to change the tools and technologies without impacting the experience and business functionality
- Build large and complex systems with high fault tolerance that provides on-demand scalability and high availability
- Adopt a lean model to build the presentation and integration layer, so that responsive and high performing experiences can be built
- Use platform principles to develop modular components for high extensibility
- The platform should provide on-demand scalability and high availability.
- Adopt DevOps principles for continuous integration, continuous delivery with faster release time

KEY TENETS OF MICROSERVICES

- **Modularity:** Each microservice represents a logically cohesive, lightweight and independent business functionality with well-defined boundaries. By design, microservices are highly granular, and independently built and deployed.
- **Single functionality principle:** Each microservice encapsulates a single business functionality or use case.
- **Stateless:** Microservices provide stateless communication between the client and the server.
- **Independent deployment and independent scalability:** Each microservice can be independently deployable, and hence, they are independently scaled to respond to varying workloads and user demands.
- **Self-containment:** The microservice deployment unit is self-contained as it includes all dependent libraries, storage units, databases and such. Microservices uses decentralized data management wherein service-specific database is part of microservice deployment unit/container.
- **Resiliency:** Microservices architecture eliminates single point of failure through distribution of coherent functionality to various microservices. Even if a single service goes down, the application will continue to work. By leveraging the circuit breaker pattern, the fault tolerance can also be enhanced.
- **Loose coupling:** Microservices are designed to be loosely coupled with minimal dependency on other services and libraries.
- **Smart Endpoint and dumb pipes:** Microservices communicate with each other with well-defined APIs (smart endpoints) over simple protocols such as REST over HTTP (dumb pipes).

TWELVE-FACTOR APPS AND MICROSERVICES

Modern architecture aims to develop large and complex applications in software as service (SaaS) models. The twelve-factor methodology provides guidelines for developing SaaS-based applications. Microservices, containers and their ecosystem fit well into the twelve-factor methodology. We have given various solution components in the microservices architecture ecosystem that can be leveraged for building twelve-factor apps in table 1.

TABLE 1 SOLUTION COMPONENTS FOR 12 FACTOR APPS

FACTORS	BRIEF DETAILS *(Ref: https://www.12factor.net/)*	MICROSERVICE ECOSYSTEM COMPONENT
Codebase	One codebase tracked in revision control, many deploys	Source control systems such as gitlab or bitbucket can be leveraged for this. Source control systems provide in-built support for code revisions and version controls. Deployment can be done through build pipeline and CI/CD pipeline.
Dependencies	Explicitly declare and isolate dependencies	Build and packaging libraries such as Maven, Gradle and npm allow us to declare the dependent libraries along with the library version.
Config	Store config in the environment	Environment specific configurations (such as URLs, connection strings etc.) can be injected to the configuration files (such as application.properties in Spring application) in the build pipeline.
Backing services	Treat backing services as attached resources	Backing services such as storage, database or an external service should be accessible and managed through configuration files to ensure portability.
Build, release, run	Strictly separate build and run stages	Source code branches and automated CI/CD pipelines can be leveraged to manage environment specific releases.
Processes	Execute the app as one or more stateless processes	Stateless is the core tenets of microservices. Implementing token-based security helps us implement stateless authentication and authorization.

TABLE 1 SOLUTION COMPONENTS FOR 12 FACTOR APPS

FACTORS	BRIEF DETAILS *(Ref: https://www.12factor.net/)*	MICROSERVICE ECOSYSTEM COMPONENT
Port binding	Export services via port binding	The services can be made visible through exposed ports. Container infrastructure provides configuration files (such as service.yaml in Docker) to bind the ports for services.
Concurrency	Scale out via the process model	By leveraging independent deployment feature of microservices, we can individually scale the most needed microservice by using on-demand scaling feature of containers.
Disposability	Maximize robustness with fast startup and graceful shutdown	Individual containers/pods can be started quickly. Container orchestrator can handle container shutdown gracefully.
Dev/prod parity	Keep development, staging, and production as similar as possible	We can achieve parity in environment dependencies, server dependencies, configurations through a container model.
Logs	Treat logs as event streams	Each microservice can log to standard output, which can be picked up by tools such as Kibana or Splunk to manage and visualize the logs centrally.
Admin processes	Run admin/management tasks as one-off processes	Container orchestration is managed by tools such as Kubernetes, while log management is carried out by tools such as Kibana or Splunk. Other application-specific administration can be deployed as a separate microservice.

ADVANTAGES OF MICROSERVICES

Microservice architecture is a preferred option for modern digital architecture as it is possible to design and develop extensible solutions. The key advantages of microservices have been provided below:

- **Agile delivery:** Decomposing the services into logically modular, independent microservices helps in Agile delivery, easily fits in the DevOps model (continuous integration, continuous deployment and continuous delivery) and faster time to market. If the application needs high deployment velocity agile delivery is the preferred option
- **Diversified technologies and distributed teams:** Development models support distributed teams developing the microservices in various languages. Each service can be built using the most appropriate language, tool and technology.
- **Asynchronous invocation:** Microservices are stateless by default, helping us to asynchronously invoke them to deliver high performance and high scalability.
- **Headless integration model:** The systems of record (SOR) such as CMS, DAM workflow and such can be integrated in headless mode through microservices so that the system of engagement will be completely decoupled from the system of record.
- **Token-based security:** We could implement lightweight token-based security with microservices to implement authentication and fine-grained authorization for stateless microservices.
- **Lightweight services:** The microservices provide a lightweight services model that can leverage JSON data contract.
- **Extensibility:** Microservices can be leveraged to create an extensible solution by quickly onboarding newer ones.
- **Independent scalability:** As each microservice can be independently deployed along with all the dependencies, they can also be independently scaled based on the load.
- **Multi-speed IT model:** We can build microservices layer on top of traditional SOA-based webservices in legacy platforms to implement a multi-speed IT model.

- **Decoupling and loose coupling:** Systems of engagement (SOE) and SOR will be fully decoupled so that we have flexibility to change the backend systems.
- **Cloud readiness:** The microservices design can be easily integrated with Cloud environments on Cloud native containers or over Cloud-based virtual machines.
- **Open standards:** Microservices can be built using open standards such as REST, JSON, OAuth and others.
- **High performance and high availability:** Containerized microservices can be leveraged for high performance and availability. The asynchronous nature of microservices invocation also improves the performance.
- **Enabler for large and complex applications:** Microservices architecture can be leveraged for developing large and complex applications that need multiple technologies and distributed teams with independent scalability.
- **Resiliency and fault tolerance:** Container eco-system offers features such as clustering, load balancing, circuit breaking and others to offer high resiliency and fault tolerance for the microservices. The design provides graceful degradation of functionality.
- **Independent development and deployment:** Each microservice can be independently developed and deployed by different teams. This helps us to incrementally add features and functionalities.
- **Responsive to changes:** We can easily change the existing features and extend the functionality with microservices architecture.

MICROSERVICES ARCHITECTURE

In this section, we have elaborated on the solution architecture for microservices.

MICROSERVICES REFERENCE ARCHITECTURE

The microservices reference architecture identifies the main layers and solution components involved in the solution. We have depicted Microservices reference architecture with key components in Figure 1.

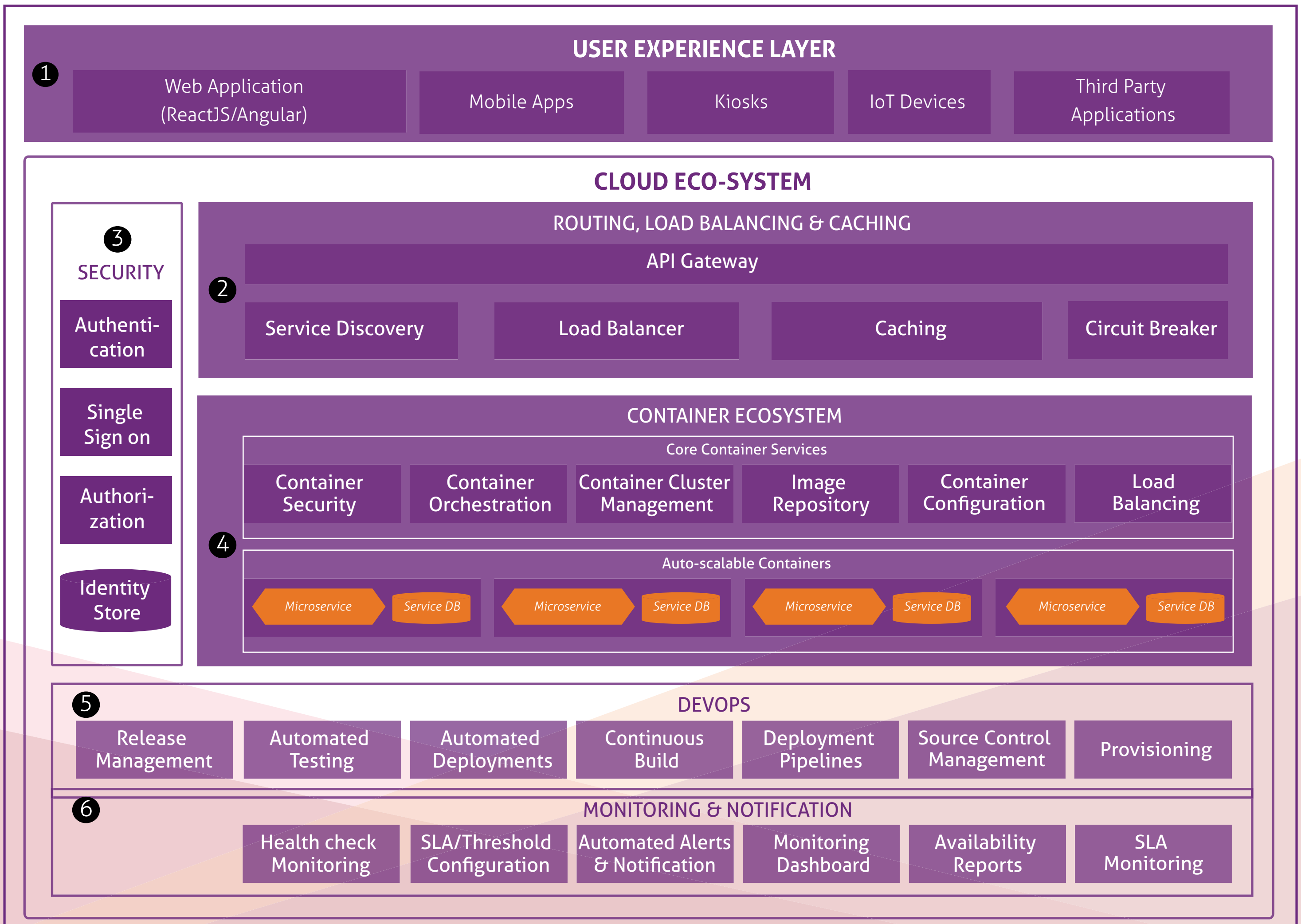

Figure 1: Microservices Reference Architecture

USER EXPERIENCE LAYER

The user experience layer consists of consumers such as mobile apps, web applications, IoT devices, third party services, kiosks and others. The consumers of microservices invoke stateless microservices with parameters.

ROUTING, LOAD BALANCING AND CACHING

This layer mainly has components to route requests to specific consumers, load balance them and cache the response. The details of these components have bveen provided below:

- **API Gateway:** Gateway/ proxy for the client to access microservices. All cross-cutting functionalities like security, loaded balancing, governance, protocol transformation, analytics, performance management, payload transformation etc. are implemented here.
- **Service discovery:** It works as a directory service for all microservices in a domain. API gateway consults with the service directory to route all client requests. Theinter service communication also leverages service directory.
- **Load balancer:** The load balancer is responsible for routing the requests to the microservice instance.
- **Caching:** Static data (such as images, text files etc.) and service response will be cached for optimal performance. Systems such as AWS CloudFront provide edge-side caching and systems such as Redis provide in-memory caching.
- **Circuit breaker:** This component is used to detect service failure and provide fallback.

CONTAINER ECOSYSTEM

Container images are the preferred deployment units of microservices. The container eco-system mainly provides a container orchestrator (such as Kubernetes) which manages the lifecycle of containers. Below are the core services provided by the container eco-system:

- **Container security:** Ensures the security of container images, container access management, container security testing, infrastructure security, container pipeline security and others.
- **Container orchestration:** The container orchestrator is responsible for managing the lifecycle of containers, monitoring the container's health, configuring the service ports and others.
- **Container cluster management:** The cluster management module is responsible for managing various container clusters such as blue/ green clusters.
- **Image repository:** Users can reuse and publish the images on the image repository.
- **Container configuration:** We can specify container configuration elements such as public IP, deployment strategy, namespace, inter-container dependency, custom configuration values, storage volumes, container labels etc.
- **Load balancing:** Based on the container's availability, traffic volume and health status, the load balancer evenly distributes the load to various containers.

SECURITY

The cloud infrastructure provides various security related managed services such as user directory service, authentication service, authorization service, SSO service and others. In the OAuth 2.0 model, we will have time-based tokens for secured resource access.

DEVOPS

The DevOps module includes components for release management, automated deployments, continuous build, deployment pipeline, source control management and provisioning.

MONITORING AND NOTIFICATION

The monitoring and notification infrastructure includes components for configuring SLA thresholds (CPU, memory, response times etc.), health check monitoring (monitoring system and service availability, performance), threshold-based alerting and notification, monitoring dashboard, availability reports and others.

A sample microsevice interaction is given in Figure 2.

CLOUD NATIVE MICROSERVICE FRAMEWORK – SOLUTION COMPONENTS

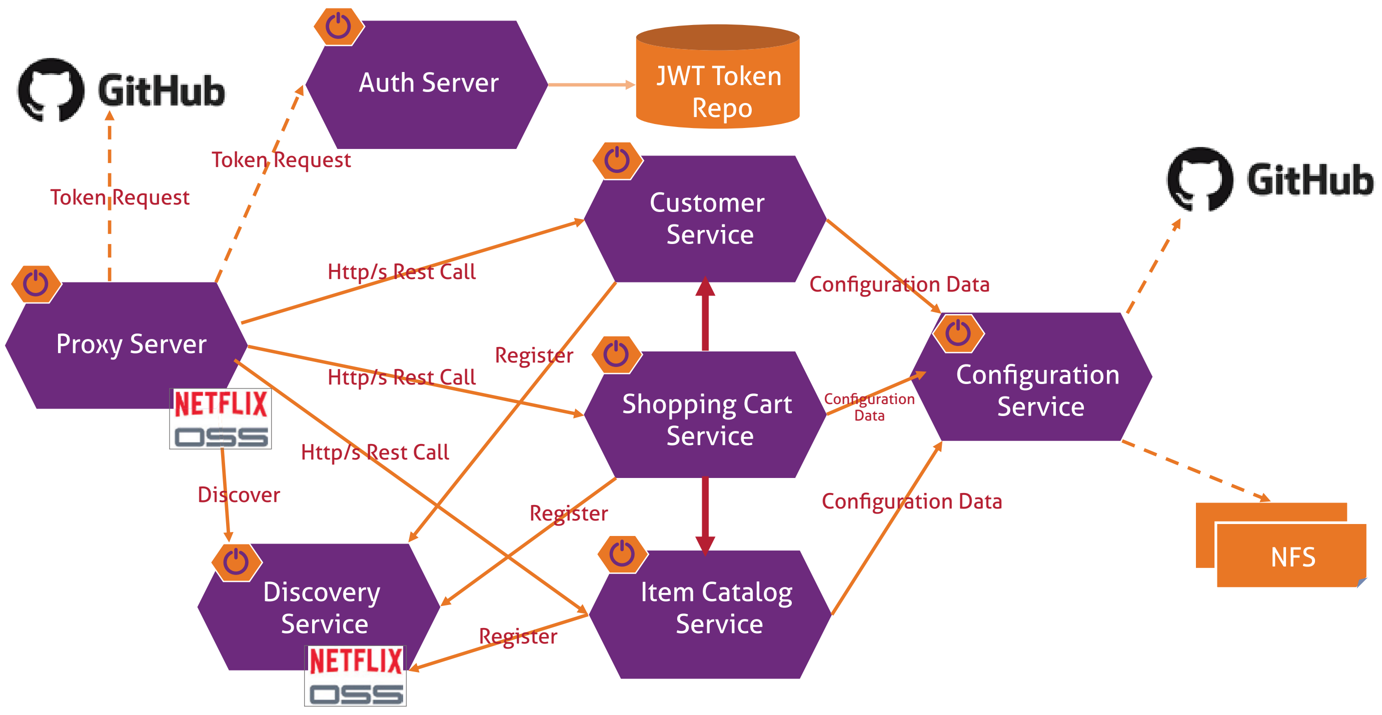

Figure 2: Sample Microservice Interaction Diagram

These microservices are supported by cross-cutting infrastructure services e.g. Authorization, Discovery, Proxy, Configuration etc.

TABLE 2 SERVICE DETAILS

INFRA SERVICE	DETAILS
Auth Server	Custom Authorization server implementation which provides a configurable (in-memory, JWT) identity token. The token will be used to verify a user's authenticity every time the client tries to access the above business service
Proxy Sever	We can leverage the Netflix OSS- Zuul service.
Service Discovery	We can leverage the Netflix OSS- Eureka service
Configuration Service	Custom cloud configuration service implementation. It provides configurable in-file or Git storage for service configuration.

We have given the sample solution components and product stack in table 3.

TABLE 3 SAMPLE MICROSERVICES PRODUCT STACK

ARCHITECTURE COMPONENT	SAMPLE PRODUCT STACK
Api Gateway	Netflix OSS- Zuul
Authentication Service	Spring – Security Oauth2, OpenID Connect
Service Discovery	Netflix OSS- Eureka, Apache Zookeeper
Configuration Service	Spring – Cloud Config Server
Microservice	Spring – Boot, Vert.x, Dropwizard
Monitoring	Netflix OSS- Turbine, Prometheus, Splunk, ELK (Elasticsearch, Logstash, Kibana), CAdvisor Visualization – Grafana, Kibana
Circuit Breaker	Netflix OSS- Hystrix
Microservices Testing	Wiremock
Container Ecosystem	Docker – Container technology Docker Swarm, Kubernetes – Container orchesrator

CLOUD DEPLOYMENT ARCHITECTURE

Microservices can be easily deployed in popular Cloud platforms. We have discussed the deployment details of microservices on two popular Cloud platforms.

AMAZON WEB SERVICES

We have depicted sample AWS deployment architecture for microservices in Figure 3. The main solution components are as follows:

- **Cloudfront CDN:** Cloudfront native CDN allows to effectively deliver the content for the consumer from different geographies.
- **S3 Bucket:** ReactJS/Angular-based web applications will be deployed on S3 bucket & it is integrated with AWS Cloud front. The frontend is secured with WAF (Web Application Firewall). Deploying frontend static content on S3 makes it highly scalable & cost effective.
- **Custom Services Layer:** Backend microservices are developed using NodeJS/ Spring Boot and are deployed on containers using AWS Cloud-native container orchestration services ECS. These services are accessed through AWS API Gateway.

AWS-BASED MICROSERVICES DEPLOYMENT ARCHITECTURE

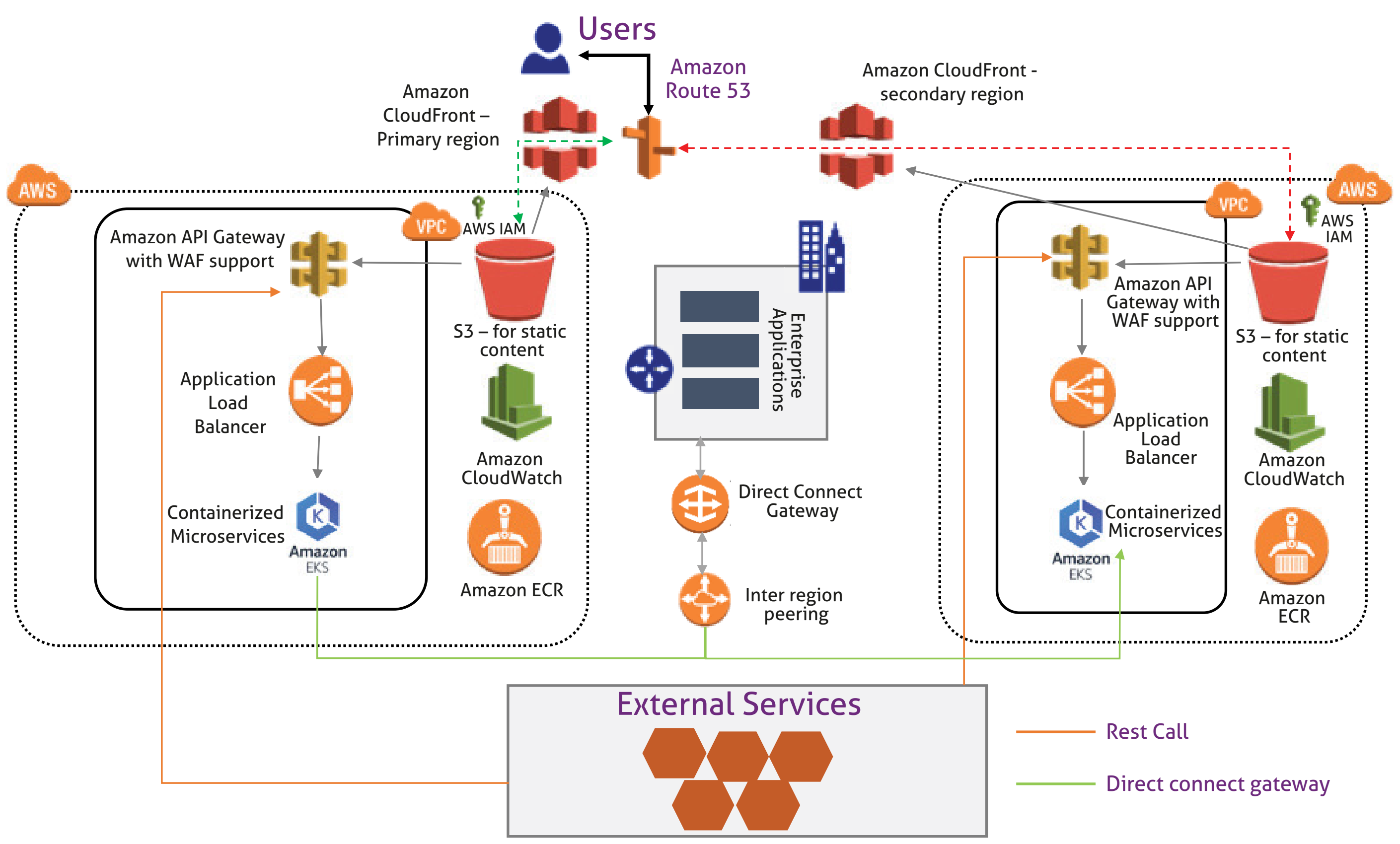

Figure 3: Sample AWS-based Microservices Deployment Architecture

- **API Gateway:** API integration with the internal system via API gateway through direct connect internal network.

Given below are the key architectural considerations:

- Cloud-native PaaS solutions like API Gateway, Cloudfront, and S3 to achieve the low cost of cloud operations and scalability
- Containerized backend services on Docker containers orchestrated through Cloud-native ECS which helps meet the required availability and scale at need basis
- Integration with internal applications through high speed secured direct connect connection to on-premise data center or Cloud.

MICROSOFT AZURE

We have depicted sample Azure deployment architecture for microservices in Figure 4. The main solution components are as follows.

AZURE-BASED MICROSERVICES DEPLOYMENT ARCHITECTURE

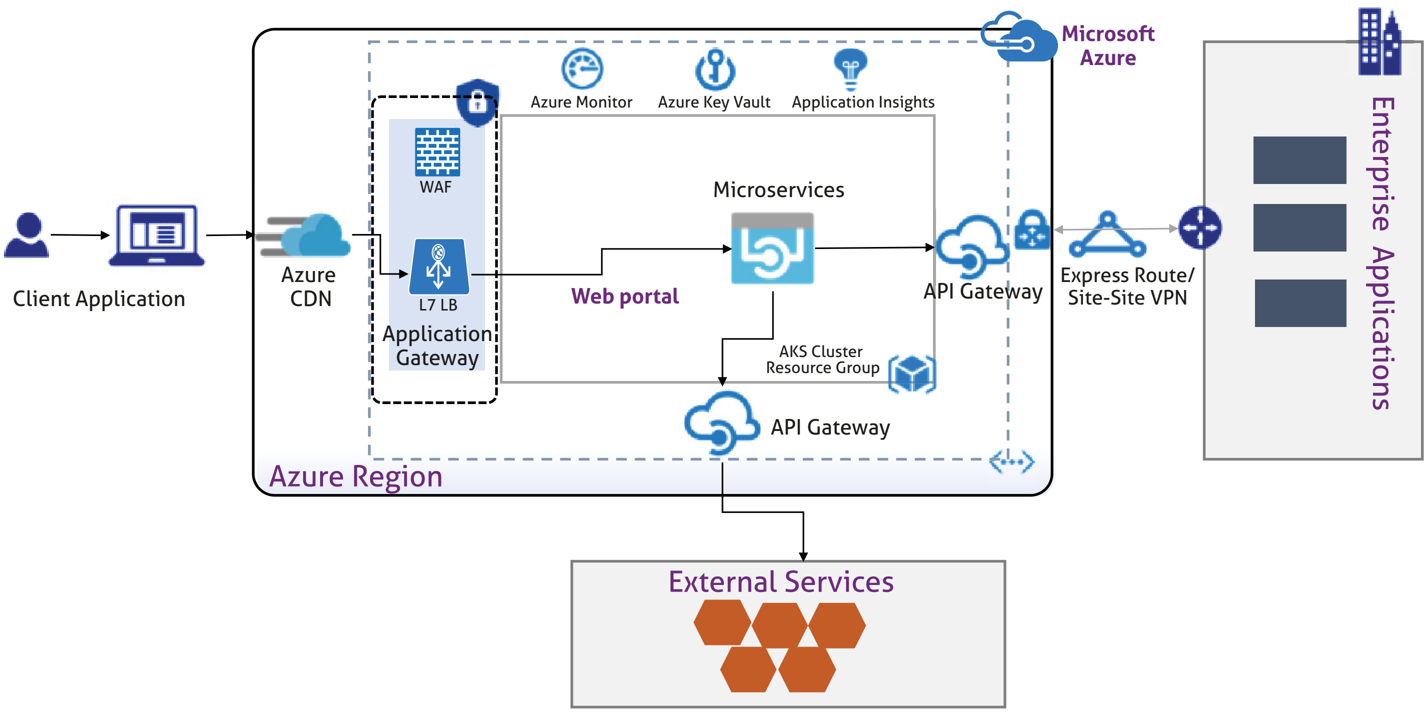

Figure 4: Sample Azure-based Microservices Deployment Architecture

- **Azure CDN:** Azure CDN allows the effective delivery of content for the consumer from different geographies. This helps the content faster and closer to consumer regions.
- **Web-App Services:** ReactJS/ Angular frontend will be deployed on Azure Web App services, front ended by the application gateway with WAF security layer.
- **API – App Services:** Custom backend microservices are deployed on API app services. App services are fully managed Azure-native PaaS offerings.
- **Integration:** Third party services are consumed through API Gateway. On-premise application are integrated to Cloud through express route, while the communication goes through the API gateway.

The primary considerations are given below:

- **Highly scalable infrastructure –** Being a purely PaaS offering from Azure, it can scale on a need basis.
- **Highly performant –** CDN helps deliver the content closer to consumers across the geographies.
- **Reduced maintenance cost –** Cloud-native solutions like App Services, API Gateway do not require any maintenance.
- **IaaC –** Infrastructure as a Code template to consistently create the right environment.

MICROSERVICES PATTERNS

By leveraging microservices patterns, we can solve some of the common problems and apply the best practices. In this section, we discuss the main patterns used in Microservices architecture.

DECOMPOSITION PATTERNS

Microservices are loosely coupled with a high functionality coherence. Optimum granularity of microservices is required for high performance and high scalability. We can decompose the services in following ways:

- **Decomposition based on business capability:** Create microservices based on business capabilities. For instance, in an e-commerce solution, the main business capabilities are order management, product promotions, service management and others. We can create microservices based on these.
- **Decomposition based on sub-domain:** We can identify the sub-domains of the core domain (business) and create microservices based on that. For instance, the order management domain has sub-domains such as product catalog, inventory management and others.
- **Decomposition based on transaction:** Identify the main transactions of the application and develop microservices for them. For instance, the main transactions of an e-commerce application are login, checkout, search and such; We can create microservices for these transactions.
- **Decomposition based on resources:** We can create microservices based on nouns or resources and define the operations. For instance, in an e-commerce solution, 'products' is a resource and we can define the list all products (GET /products), query particular product (GET /product/{1}), insert product (PUT /product/{}).

INTEGRATION PATTERNS

As microservices are mainly used for integration, let's look at optimal ways to invoke multiple microservices, microservice invocation sequence, data and resource security, data transformation and responses for different clients and others.

- **API gateway pattern:** An API gateway provides a centralized access point for invoking a microservice. The API gateway handles security (such as authentication, authorization), governance (such as logging service, monitoring service), request routing, protocol transformation, data transformation and the aggregation of responses from multiple services.
- **Aggregation pattern:** When a single microservice needs responses from multiple microservices, a composite service can take the responsibility of aggregating the response.
- **UI composition pattern:** The end user interface layer is laid out into various sections, which individually invokes the corresponding microservice asynchronously. Modern user interfaces use single page application (SPA) built by Angular or ReactJS frameworks.
- **Backend for frontend:** Instead of creating a general-purpose microservice, we can design a microservice and its response specifically for the client agents (such as desktop browsers, mobile devices etc.). This tight coupling of client agents with the corresponding backend service helps us to efficiently create response data.

DATA-RELATED PATTERNS

As microservices are self-contained and designed for independent scalability, we end up having service-specific databases (such as database server per service, database schema per service and service-specific tables). Due to this design, we face challenges such as:

- A single service that reads or updates data from multiple databases
- A single business transaction spanning multiple services and databases
- Replication of data in the databases

The common patterns used for data scenarios have been listed below:

- **Shared database:** Though not a recommended approach, when we are decomposing a monolith application to microservices, the approach can comprise a single being shared by multiple microservices. Once the transformation is complete, each service should gets its own database.
- **Command Query Responsibility Segregation (CQRS):** Database handling is split into two categories, the command part for handling data creation, update, deletion and the query part that uses materialized views to retrieve data. The materialized view is updated by subscribing to data change events. Event sourcing pattern is used along with CQRS to create immutable events.
- **Saga pattern:** When a business transaction needs to manage data consistency that is spread across multiple databases, we could use Saga pattern. As a part of the Saga pattern, each transaction is orchestrated locally or centrally to execute it entirely and handle the failure/rollback scenario. For instance, if a business transaction needs to handle data related to order and customer service, each of these services produces and listens to each other to handle the transaction.

OBSERVABILITY PATTERN

In this section, we list patterns that help in real-time data aggregation and notification.

- **Log aggregation:** Since microservices are deployed into individual containers, the logs generated by each of the containers (a.k.a pods) need to be aggregated to create a centralized log repository. Microservices can log to standard output or to a log file. The log management systems such as Splunk or Kibana can aggregate the log stream in real time to a centralized log repository and we can query the real-time logs.
- **Performance monitoring:** Performance monitoring services such as Prometheus, AppDynamics and NewRelic can be used to monitor the performance metrics of microservices. The performance metrics are depicted visually and we can configure the performance thresholds and notification triggers.
- **Distributed tracing:** When the request flows across various layers and microservices, it is necessary to trace the request end-to-end for error handling and for performance troubleshooting scenarios. In distributed tracking, we create a unique request ID (such as x-request-id) that is passed across all layers and microservices and logged for troubleshoot ing purposes.
- **Health check pattern:** In order to properly distribute the load and route the traffic accordingly, each microservice has to publish health check endpoint (such as /health) that provides the status of the overall health of the service. The health check service should check the status of dependent systems (such as databases, storage systems) and host connectivity to provide the overall health status of the service.

CROSS-CUTTING CONCERN PATTERNS

In the microservices eco-system, we need to handle many common, cross-cutting concerns such as security, configuration management, deployment and others. The patterns related to these concerns have been discussed in this section:

- **External configuration:** All environment-specific configurations such as connection strings, application properties and URLs should be loaded from an external configuration file. The CI/CD pipeline can inject the environment-specific configuration values during the build.
- **Service discovery:** A centralized service discovery module should handle the responsibilities such as service registra tion/ de-registration and request routing, based on service health. For client side service discovery service registry is used for load balancing and for server side service discovery, server side load balancing is used.
- **Circuit breaker:** When one of the services in the request-processing pipeline fails, the circuit breaker is responsible in terms of handling the failure and preventing the cascading of the error. The circuit breaker can monitor the error from a dependent service and fallback to a default handler in case of error. Netflix Hysterix is an example of a circuit breaker.

- **Blue green deployment pattern:** In order to seamlessly deploy the newer version of microservices with minimal downtime, we can maintain two identical production instances (blue instance and green instance), one of which will be live, serving the requests at any time. During production deployment, we can update the non-live instance and route traffic to it.
- **Access Token:** Due to the stateless nature of microservices, each request should securely pass the user identity. Access tokens such as JSON Web Token (JWT) encapsulates the claim details in microservices architecture.
- **Auditing:** Log the user actions such as authentication, password changes in logs that are centralized, immutable, and secure for auditing purposes.
- **Exception Logging:** Logs all service exceptions in a central location and provides notification feature.
- **Microservice chassis:** Reuse an existing microservices framework such as Spring Boot to leverage in-built features such as configuration handling, logging, request filtering etc.

MICROSERVICES USE CASES

Microservices can be used for various enterprise solution scenarios. The common use cases for microservices have been discussed in this section:

Mobile App Services

- **Context:** Android and iOS mobile apps use services to get the information and transaction data needed for a healthcare app. Security features like registration, login and authorization are implemented through services. Mobile apps were also integrated with third party services such as voice search, analytics and others.
- **Microservice Solution Architecture:** Microservices architecture was used to develop the light weight services layer:
  - Integration middleware is used for centralized management of service invocation.
  - Post successful authentication, the security microservice creates a JWT token encapsulating the logged-in user's information.
  - Microservices are designed based on mobile app screens. Most mobile app screens invoked one microservice to get the response. The dashboard screen invoked 3 microservices.

Single Page Application (SPA) for B2C Application

- **Context:** A B2C SPA web application was developed using Angular framework. The web application provided many features such as product configuration, admin module, dashboard etc.
- **Microservice Solution Architecture:** Microservices architecture was implemented as part of services layer:
  - API Gateway was used for governance and orchestration of microservices.
  - X-request-id was used to trace the transaction end to end.
  - Microservices were built with Docker images and deployed as Kubernetes pods. The pods were configured for auto-scaling based on the metrics (CPU and memory utilization)
  - Security service creates a valid JWT token after successful authentication.
  - Microservices logged to standard output and Kibana was used for log aggregation.
  - Cloud services were used for real time performance monitoring
  - Health check end point was implemented for all microservices that depicted the overall health of the service.
  - External configuration management was done using application.properties of Spring Boot framework. The build pipeline injected the environment specific values (such as base URL, environment value, DB connection string) to the application.properties during the build.

Multi-speed IT for legacy applications

- **Context:** A financial application platform was developed on legacy portal platform. The application users faced multiple challenges such as performance issues, scalability and the time out of many services. Heavy weight SOAP-based legacy web services contributed to the overall performance. The user experience was not responsive or contemporary.

- **Microservice Solution Architecture:** Microservices architecture was used to develop the light weight services layer:
  - Microservices architecture was leveraged to develop a multi-speed IT model. Legacy web-services were decommissioned in a phased manner. In the first release, microservices were developed on top of legacy web services.
  - Redis caching layer was used to cache the costly resource calls. Event sourcing pattern was used to flush the stale cache entries when data is updated.
  - AppDynamic monitoring software was used for real time application and service monitoring.

Microservices best practices

Given below are the key best practices in microservices architecture:

- **Naming Conventions:** The microservices' URL is usually a noun that represents a resource. We will use the schema to perform the appropriate actions. For instance
  - GET api/v1/accounts will list all accounts
  - PUT api/v1/account/1234 adds/updates the account id 1234
  - DELETE api/v1/account/1234 deletes the account id 1234
- **Versioning:** The microservice releases are managed through versions, which are a part of the microservices endpoint.
- **Logging:** Logging must be enabled to capture the errors using centralized dashboard tools such as Kibana, Splunk to monitor the logs and errors. We should design and log unique request ids that can help us to trace a user transaction end to end.
- **Monitoring and alerting:** Monitoring tools should be used to observe the performance and availability of the microservices. Additional monitoring services can be configured to monitor the disk space, CPU utilization and others. Appropriate thresholds should be configured to alert the operations teams in case of SLA violation.
- **DevOps setup:** We need to setup the DevOps ecosystem to include the build and deployment pipeline.
- **Design for failure:** We need to implement features such as the following to handle failures:
  - Auto-scaling: Leverage the auto-scaling feature of container ecosystem to automatically scale based on the user load.
  - Circuit Breaker: Design the circuit breaker pattern to handle the service exception and fallback to default service response.
- **Design philosophy:** The granularity of microservices should be based on following design principles:
  - Business functionality: Each microservice should be designed to depict one business functionality.
  - Independence: Each microservice should be independently upgradeable without impacting the service consumers. The database, storage systems should be managed by microservices.
  - Coupling and Cohesion: Coupling between microservices should be avoided and each microservice should have high functionality cohesion.
- **Governance:** We should define define standards for development, deployment and functionality/performance validation.
- **Distributed Design:** As the system is composed of multiple microservices, we need to have optimal service decomposition, clean interfaces for services and appropriate database for each service.
- **Automation:** To reduce the operational complexity of microservices architecture, we need to automate operational tasks such as build, deployment, error reporting, alerting, monitoring, auto-scaling and others.

ABOUT THE AUTHOR

Dr. Shailesh Kumar Shivakumar has 17+ years of experience in a wide spectrum of digital technologies including, enterprise portals, content management systems, lean portals and microservices. Shailesh holds a PhD degree in computer science and has authored eight technical books published by the world's top academic publishers such as Elsevier Science, Taylor and Franscis, Wiley/IEEE Press and Apress. Shailesh has authored more than 10 technical white papers, five blogs, 12 research papers, eight textbook chapters and multiple articles.Shailesh holds two granted US patents, apart from five patent applications. He has successfully led severage large scale digital engagements for Fortune 500 clients. Shailesh can be reached at ***Shaileshkumar.Shivakumarasetty@ltimindtree.com.***