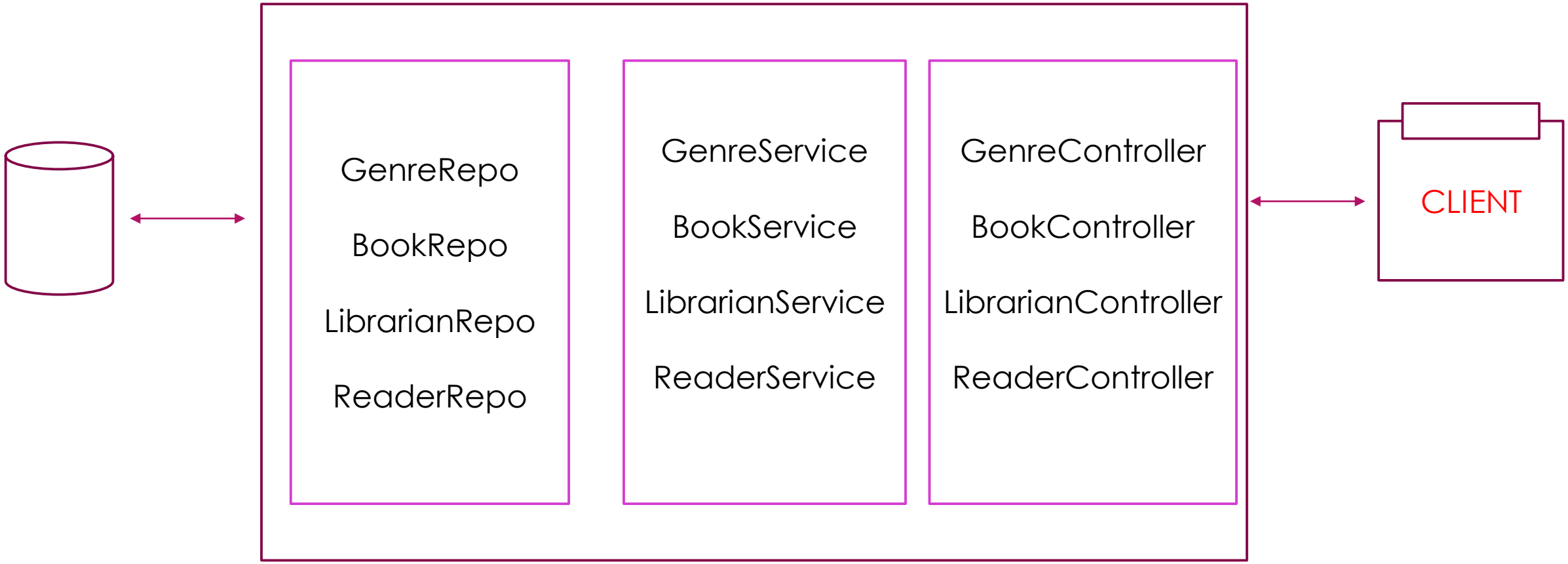
Micro Services Introduction

Objectives

- Monolithic Approach
- Modern Application Expectations and Demands
- Monolithic Demerits
- Micro Services – A high level view
- Micro Service – Design Patterns

Monolithic Approach

Monolithic Approach
a hypothetical case study



Modern Application Expectations and Demands

- Scalability
- Availability
- Rapid Development and Deployment
- Interoperability
- Adaptability of Latest Technologies
- Maintainability

Monolithic Approach Benefits

- Simple to develop.
- Simple to test.
- Simple to deploy.
- Simple to scale horizontally by running multiple copies behind a load balancer.

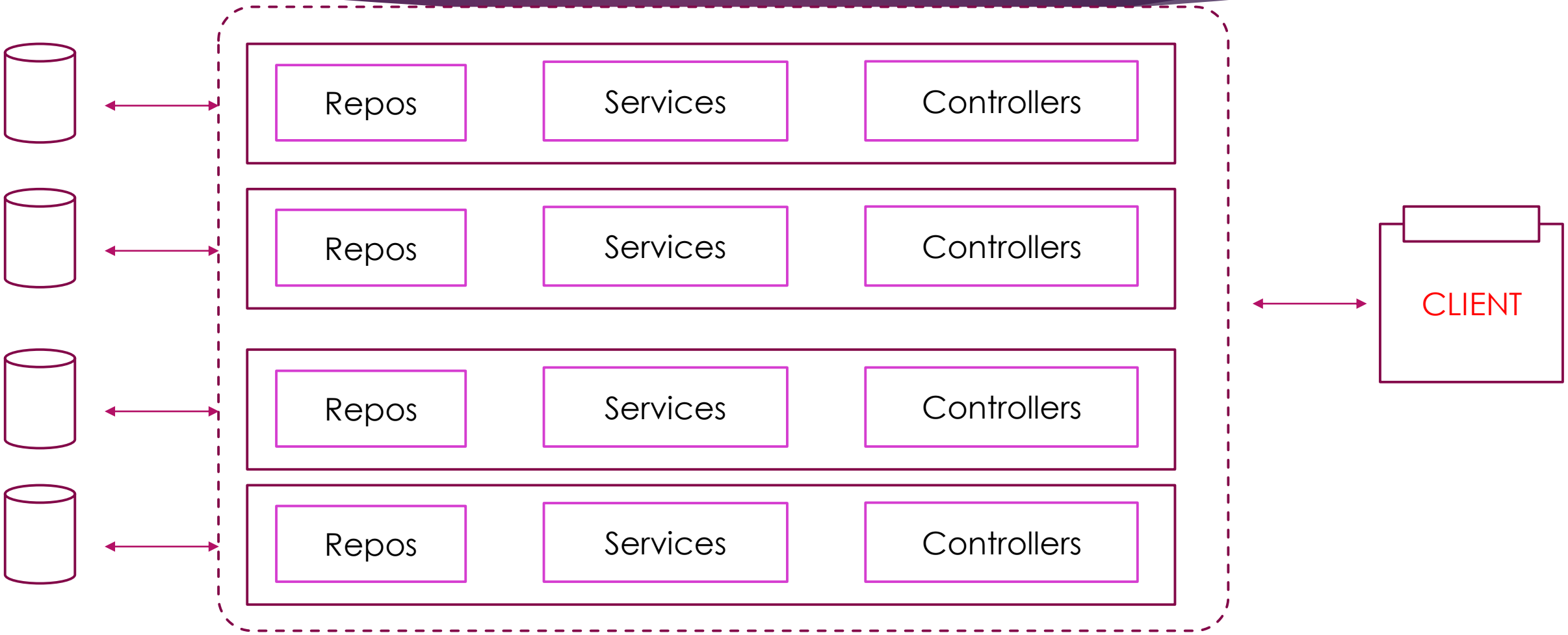
Monolithic Approach Demerits

- This simple approach has a limitation in size and complexity.
- Application is too large and complex to fully understand and made changes fast and correctly.
- The size of the application can slow down the start-up time.
- You must redeploy the entire application on each update.
- Impact of a change is usually not very well understood which leads to do extensive manual testing.
- Continuous deployment is difficult.

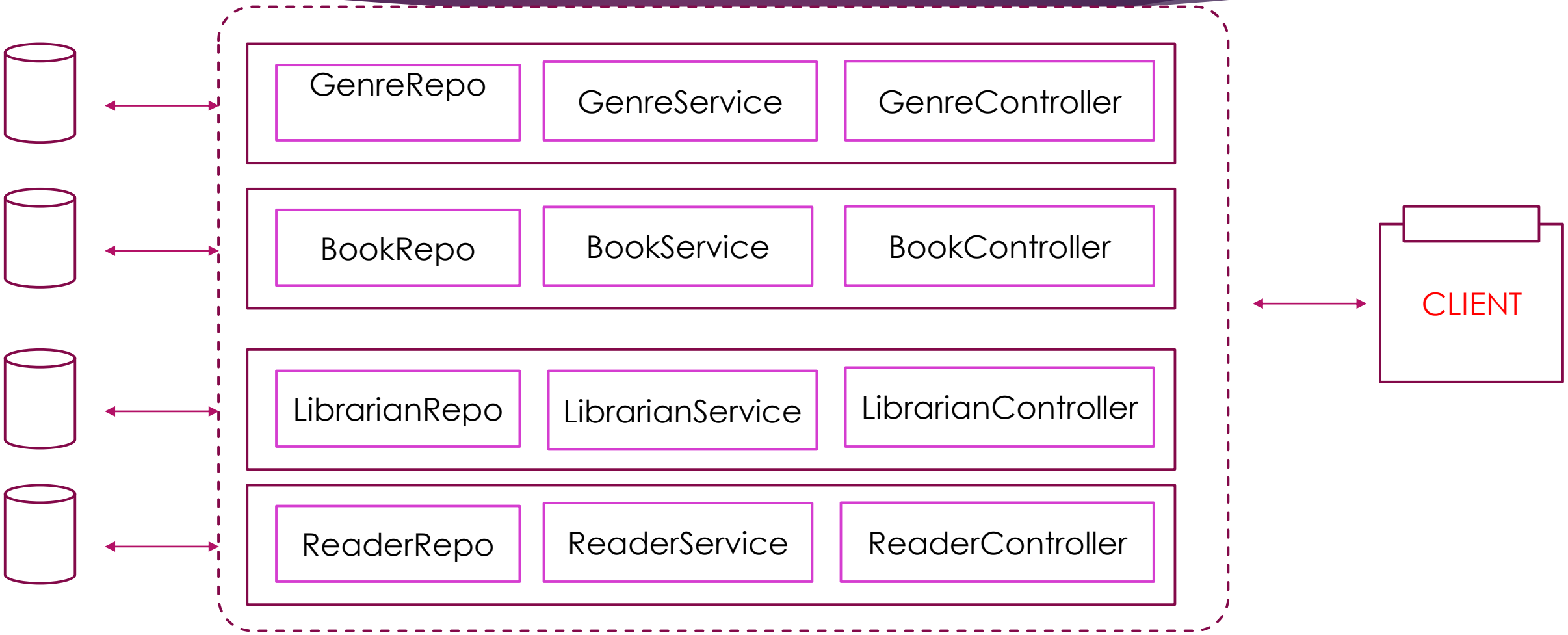
Monolithic Approach Demerits

- Monolithic applications can also be difficult to scale.
- Another problem with monolithic applications is reliability. Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the **availability** of the entire application.
- Monolithic applications has a barrier to adopting new technologies.

Micro Service A high level view



Micro Services
– a hypothetical case study



Micro Services Benefits

- It tackles the problem of complexity by decomposing application into a set of manageable services which are much faster to develop, and much easier to understand and maintain.
- It enables each service to be developed independently by a team that is focused on that service.
- It reduces barrier of adopting new technologies.
- Microservice architecture enables each microservice to be deployed independently, enabling continuous deployment.
- Microservice architecture enables each service to be scaled independently.

Micro Service Challenges

- Decomposition hiccups.
- Inter Service Communication
- Distributed Transaction and Data Consistency
- Testing (especially integration or system testing)
- Dynamic Scaling and addressing.
- Distributed Tracing.

Micro Service – Design Patterns

- Decomposition Patterns
- Integration Patterns
- Database Patterns
- Observability Patterns
- Cross Cutting Concern Patterns

Micro Service – Decomposition Patterns

- Decompose by Business Capability
- Decompose by Subdomain
- Strangler Pattern

Decompose by Business Capability

How do we decompose an application into small services?

Microservices is all about making services loosely coupled, applying the single responsibility principle. However, breaking an application into smaller pieces has to be done logically.

- One strategy is to decompose by business capability.
- A business capability is something that a business does in order to generate value.
- The set of capabilities for a given business depend on the type of business.
- For example, the capabilities of an insurance company typically include
  - sales,
  - marketing,
  - underwriting,
  - claims processing,
  - billing, compliance, etc.
- Each business capability can be thought of as a service, except it's business-oriented rather than technical.

Decompose by Subdomain

How do we decompose God Classes?

Decomposing an application using business capabilities might be a good start.

But you will come across so-called "God Classes" which will not be easy to decompose.

These classes will be common among multiple services. For example, the Order class will be used in Order Management, Order Taking, Order Delivery, etc.

- For the "God Classes" issue, DDD (Domain-Driven Design) comes to the rescue.
- It uses subdomains and bounded context concepts to solve this problem.
- DDD breaks the whole domain model created for the enterprise into subdomains.
- Each subdomain will have a model, and the scope of that model will be called the bounded context.
- Each microservice will be developed around the bounded context

Strangler Pattern

How do we keep a migration (brownfield) project from monolithic to Microservices conflict free?

The Strangler pattern comes to the rescue.

- The idea is to do it one domain at a time.
- This creates two separate applications that live side by side in the same URI space.
- Eventually, the newly refactored application “strangles” or replaces the original application until finally you can shut off the monolithic application.

Micro Service – Integration Patterns

- API Gateway Pattern
- Aggregator Pattern
- Client-Side UI Composition Pattern

Micro Service – API Gateway Pattern

- **How to call multiple microservices abstracting producer information?**
- **How to handle different type of Protocols some of which might not be supported by producer microservice?**

An API Gateway helps to address many concerns:

- the single point of entry for any microservice call
- proxy service to route a request, abstracting producer details.
- fan out a request to multiple services and aggregate the results
- offload the authentication/authorization responsibility of the microservice.

Micro Service – Aggregator Patterns

how to collaborate the data returned by each service?

This responsibility cannot be left with the consumer, as then it might need to understand the internal implementation of the producer application.

- A composite microservice will make calls to all the required microservices, consolidate the data, and transform the data before sending back.
- An API Gateway can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.

Micro Service – Client-Side UI Composition Pattern

When services are developed by decomposing business capabilities/subdomains, the services responsible for user experience have to pull data from several microservices

- UI has to be designed as a skeleton with multiple sections/regions of the screen/page. Each section will make a call to an individual backend microservice to pull the data
- These screens are known as Single Page Applications (SPA).

Micro Service – Database Patterns

- Database per Service
- Shared Database per Service
- Command Query Responsibility Segregation (CQRS)
- Saga Pattern

Micro Service – Database per Service

- One database per microservice must be designed.
- it must be private to that service only.
- It should be accessed by the microservice API only.
- One can use private-tables-per-service, schema-per-service, or database-server-per-service.
- Each microservice should have a separate database id so that separate access can be given to put up a barrier and prevent it from using other service tables.

Micro Service – Shared Database per Service

What is the suitable architecture in the case of Brown Field projects?

- A shared database per service is not ideal, but that is the working solution for the above scenario.
- This is considered as an anti-pattern for microservices, but for brownfield applications, this is a good start
- This should not be applied for greenfield applications.

Command Query Responsibility Segregation (CQRS)

How do we implement queries in microservice architecture when in database-per-service?

- CQRS suggests splitting the application into two parts — the command side and the query side.
- The command side handles the Create, Update, and Delete requests.
- The query side handles the query part by using the materialized views.
- The event sourcing pattern is generally used along with it to create events for any data change. Materialized views are kept updated by subscribing to the stream of events.

Micro Service – Saga Pattern

How do we ensure data consistency across services?

- A Saga represents a high-level business process that consists of several sub requests, which each update data within a single service. Each request has a compensating request that is executed when the request fails. It can be implemented in two ways:
- Choreography — When there is no central coordination, each service produces and listens to another service's events and decides if an action should be taken or not.
- Orchestration — An orchestrator (object) takes responsibility for a saga's decision making and sequencing business logic.

Micro Service – Observability Patterns

- Log Aggregation
- Performance Metrics
- Distributed Tracing
- Health Check

Log Aggregation

How can we understand the application behavior through logs for a particular request, when a request is processed by a line of service-instances running on a isolated servers?

- Centralized logging service that aggregates logs from each service instance.
- PCF (Pivotal Cloud Foundry)- Logaggregator, .AWS Cloud Watch ..etc

Performance Metrics

How should we collect metrics to monitor application performance?

- A metrics service is required to gather statistics about individual operations..
- There are two models for aggregating metrics:
  - Push — the service pushes metrics to the metrics service e.g. NewRelic, AppDynamics
  - Pull — the metrics services pulls metrics from the service e.g. Prometheus

Distributed Tracing

How do we trace a request end-to-end to troubleshoot the problem, where requests often span multiple services.?

We need a service which

- Assigns each external request a unique external request id.
- Passes the external request id to all services.
- Includes the external request id in all log messages.

Spring Cloud Slueth along with Zipkin server, is a common implementation.

Health Check

How do we ensure that a service is up and ready to serve our request before a request can be forwarded to it?

- Each service needs to have an endpoint which can be used to check the health of the application, such as /health.
- Spring Boot Actuator does implement a /health endpoint.

Micro Service – Cross Cutting Concern Patterns

- External Configuration
- Service Discovery Pattern
- Circuit Breaker Pattern

External Configuration

How do we avoid code modification for configuration changes as services need different config properties for different environments like dev, QA ..etc?

- Externalize all the configuration. The application should load them either at startup or on the fly.
- **Spring Cloud config** server provides the option to externalize the properties to GitHub and load them as environment properties.
- These can be accessed by the application on startup or can be refreshed without a server restart.

Service Discovery Pattern

How does the consumer or router know all the available service instances and locations, in the scenario where the IPs and ports are dynamically assigned to service-instances?

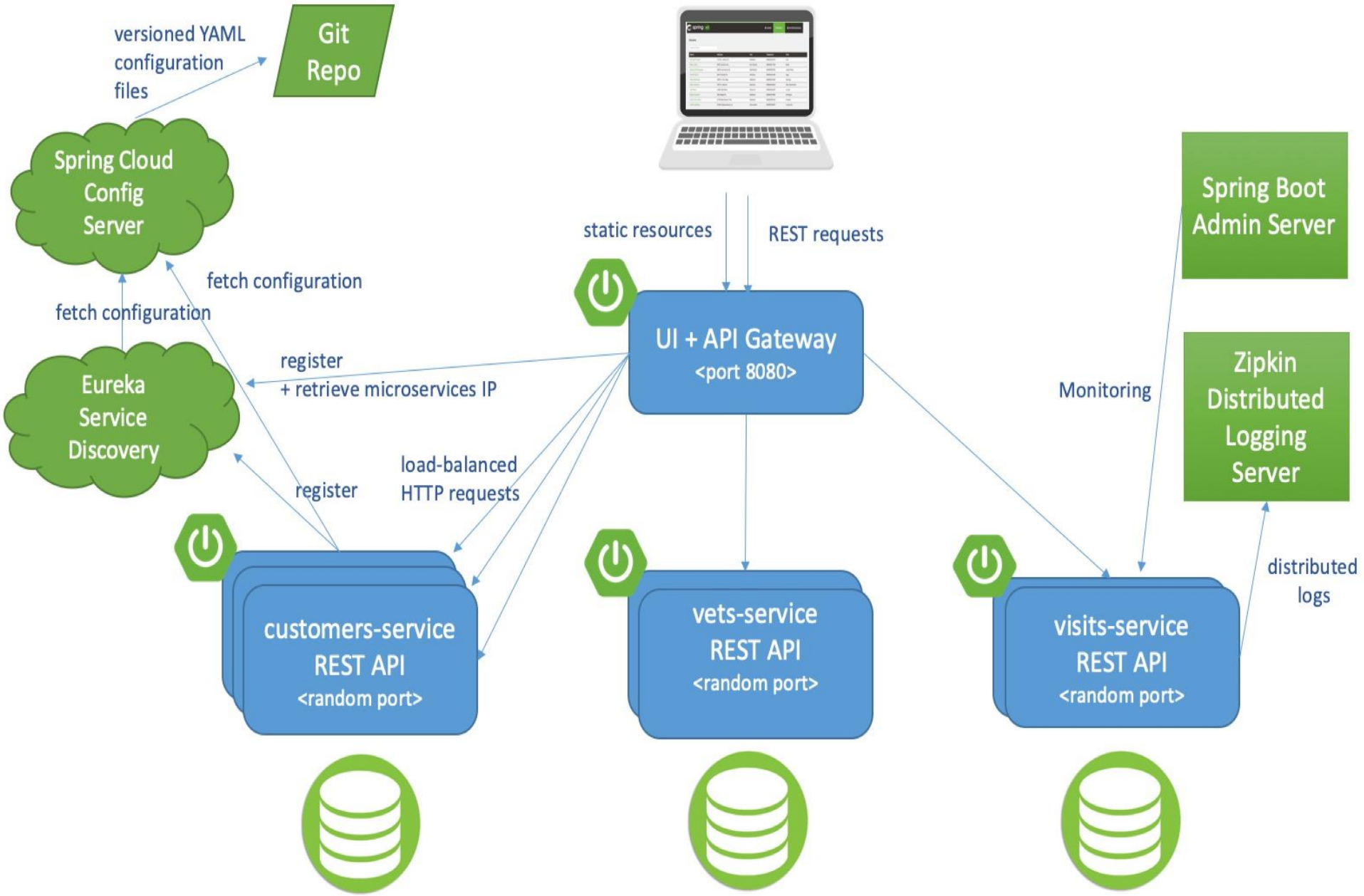
A service registry needs to be created which will keep the metadata of each producer service.

- A service instance should register to the registry when starting and should de-register when shutting down.
- The consumer or router should query the registry and find out the location of the service.
- The registry also needs to do a health check of the producer service to ensure that only working instances of the services are available to be consumed through it.
- There are two types of service discovery: client-side like Netflix Eureka and server-side like AWS ALB.

Circuit Breaker Pattern

How do we avoid cascading service failures and handle failures gracefully incase, the downstream service are down?

- A downstream service is one on which another service may rely to fetch data.
- A circuit breaker observes the attempts made to a downstream service and after a threshold of failures, the request is handled via a fallback mechanisim.
- After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. Otherwise, if there is a failure, the timeout period begins again.
- Netflix Hystrix is a good implementation of the circuit breaker pattern.


versioned YAML configuration files
Git Repo
Spring Cloud Config Server
fetch configuration
fetch configuration
Eureka Service Discovery
register
+ retrieve microservices IP
register
load-balanced HTTP requests
static resources
REST requests
UI + API Gateway
<port 8080>
Spring Boot Admin Server
Zipkin Distributed Logging Server
Monitoring
distributed logs
customers-service REST API
<random port>
vets-service REST API
<random port>
visits-service REST API
<random port>