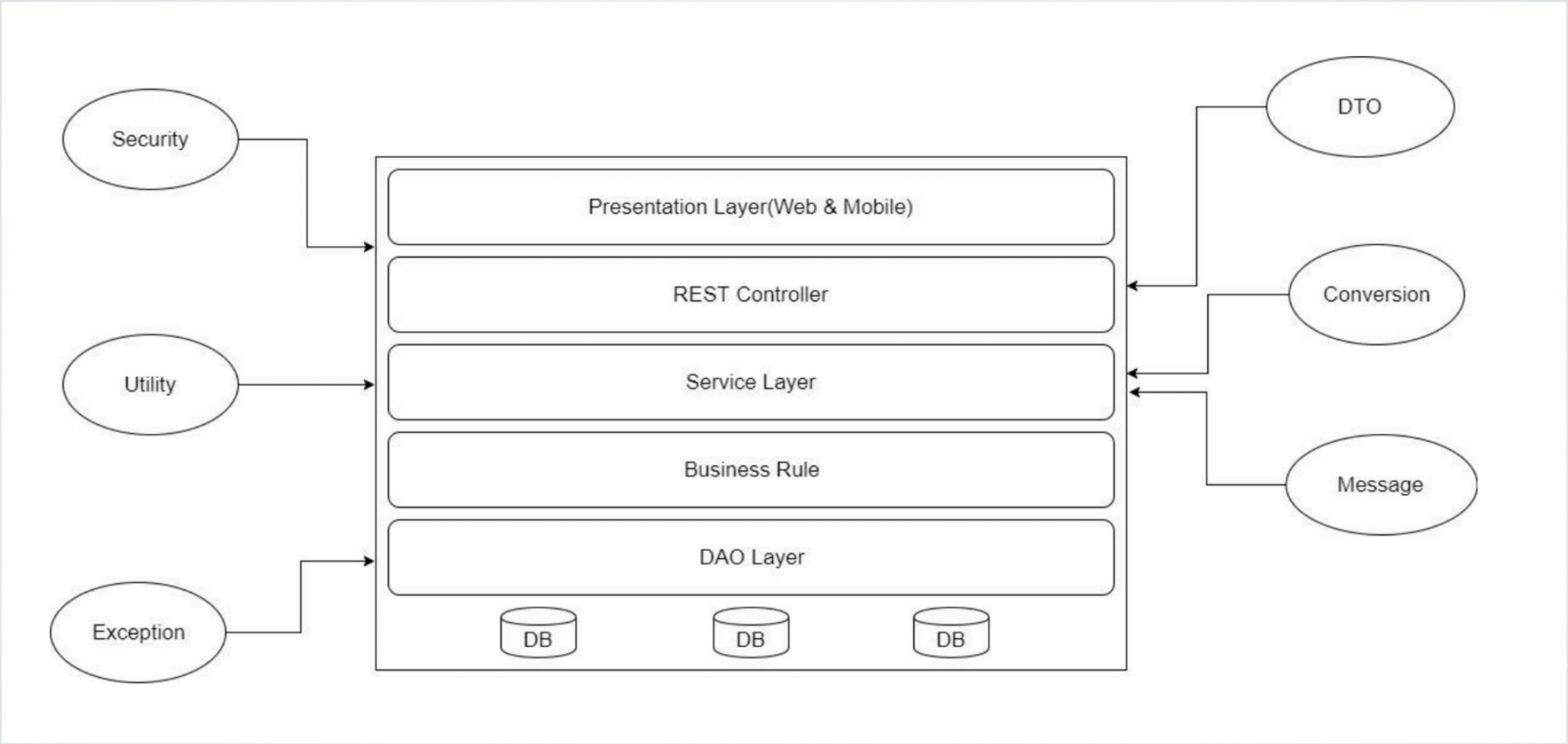
JAVA FULL STACK DEVELOPMENT PROGRAM

Session 26: Microservices and Spring Cloud

SCENARIO

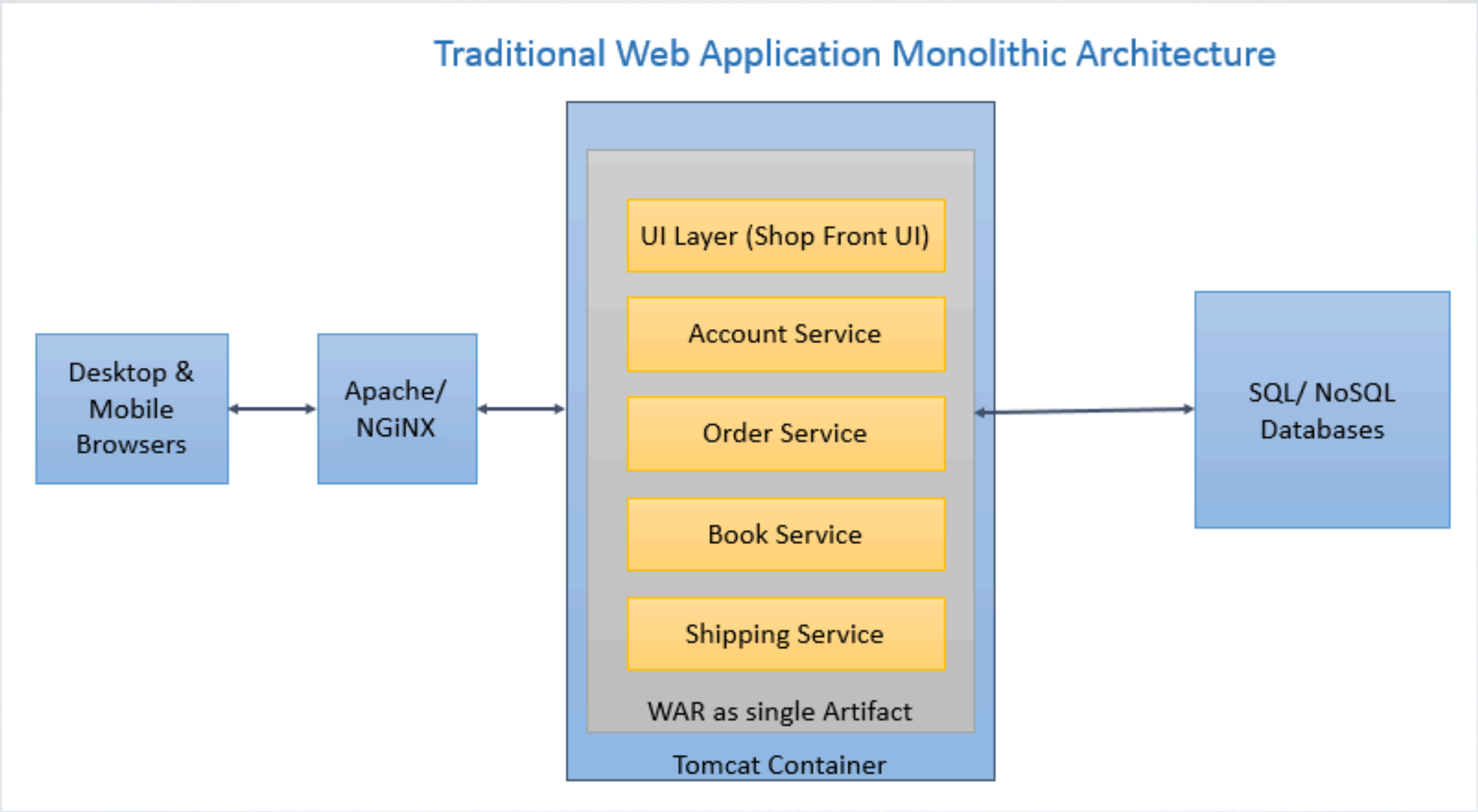
- Assume that we have following application structure



SCENARIO

- A server-side enterprise application:
    - It must support a variety of different clients including desktop browsers, mobile browsers and native mobile applications.
    - The application might also expose an API for 3rd parties to consume.
    - It might also integrate with other applications via either web services or a message broker.
    - The application handles requests (HTTP requests and messages) by executing business logic; accessing a database; exchanging messages with other systems; and returning a HTML/JSON/XML response.
    - There are logical components corresponding to different functional areas of the application.
- What would be potential problems?

MONOLITHIC


Traditional Web Application Monolithic Architecture
Desktop & Mobile Browsers
Apache/ NGiNX
UI Layer (Shop Front UI)
Account Service
Order Service
Book Service
Shipping Service
WAR as single Artifact
Tomcat Container
SQL/ NoSQL Databases

MONOLITHIC

- This architecture has a number of benefits:
    - Simple to develop - the goal of current development tools and IDEs is to support the development of monolithic applications
    - Simple to deploy - we simply need to deploy the WAR file (or directory hierarchy) on the appropriate runtime
    - Simple to scale - we can scale the application by running multiple copies of the application behind a load balancer
- However, once the application becomes large and the team grows in size
    - The large monolithic code base intimidates developers, especially ones who are new to the team
    - Overloaded IDE and web container
    - Scaling the application can be difficult (We can only scale in one direction — we can run multiple copies of same instance to handle large volume of transaction, however we can't scale with an increasing data volume since each instance will access all data)
    - Requires a long-term commitment to a technology stack (Hard to adopt new technologies)

SOLUTION

- What if we can decouple/split the entire application into different parts?
    - Account Application
    - User Application
    - Billing Application
    - Calling Plan Change Application
    - Add or Remove Feature Application
    - Order Application
- Each of those application holds their own databases.
- How can we achieve the goal?

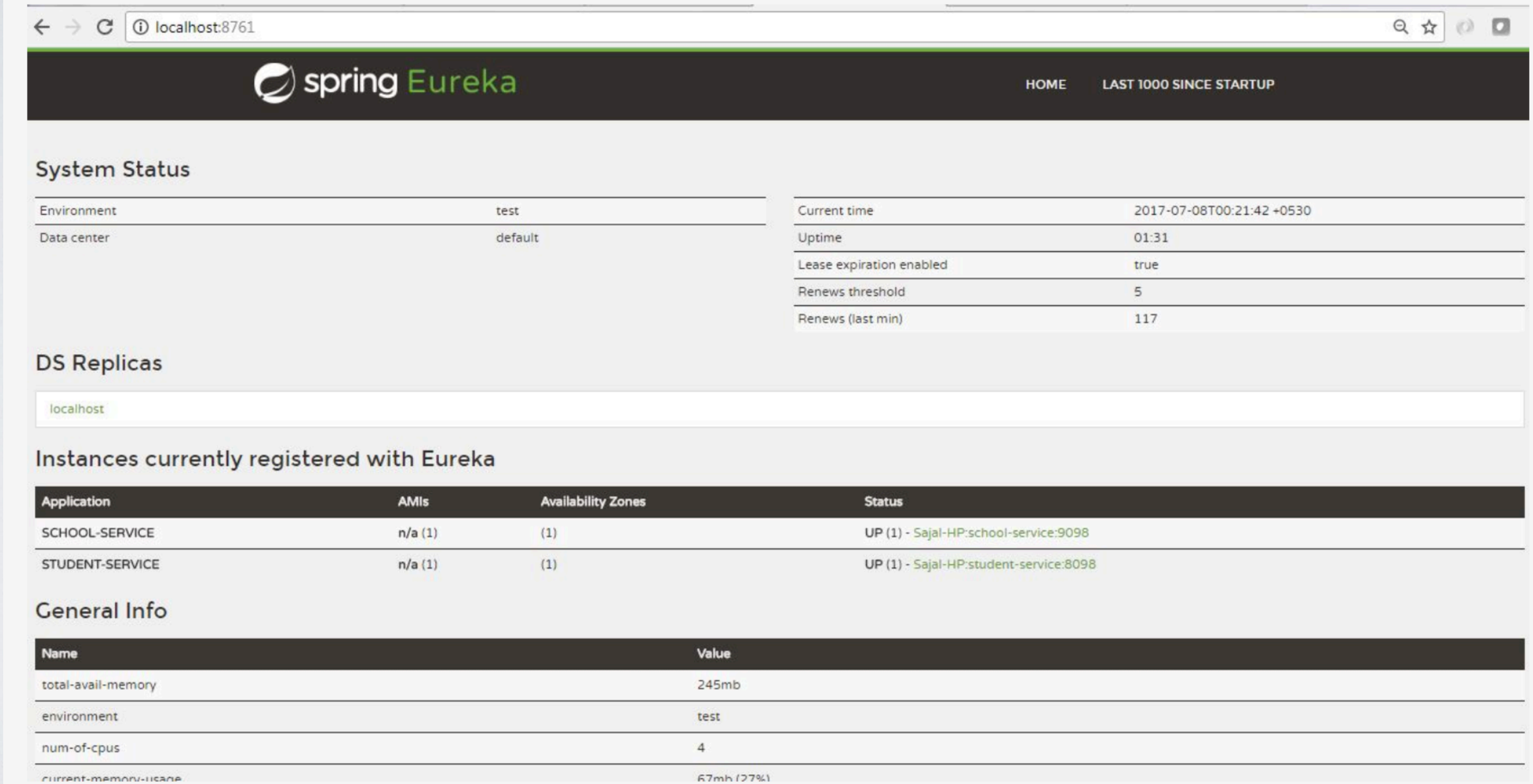
SCENARIO

- Now we have a lot of independent services/application in the company.
- Each service will have its own
    - Host name, Port
    - Application context
    - Entry points
    - Database connection
    - Technology stack
- What if we have 10 independent services?
- What if we have 50 independent services?
- What if we have 100 independent services?

SERVICE DISCOVERY

- Each service/application should have one way to know where are other services/ applications.
- *Client-side service discovery* allows services to find and communicate with each other without hard-coding hostname and port
- With Netflix Eureka each client can simultaneously act as a server, to replicate its status to a connected peer.
    - In other words, a client retrieves a list of all connected peers of a *service registry* and makes all further requests to any other services through a load-balancing algorithm.
    - To be informed about the presence of a client, they have to send a heartbeat signal to the registry

SERVICE DISCOVERY



SERVICE DISCOVERY

- Creating an Eureka Server is very easy
  - Adding *spring-cloud-starter-netflix-eureka-server* to the dependencies
  - Enable the Eureka Server in a *@SpringBootApplication* by annotating it with *@EnableEurekaServer*
  - Configure some properties (YAML is an alternative for .properties)

```
server:
  port: 8761
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

  - registerWithEureka and fetchRegistry — we set it to false, since this is our Eureka Server

EUREKA CLIENT

- In order to register our services in the Eureka Server, we have to
  - Add spring-cloud-starter-netflix-eureka-starter to our dependencies
  - Use *@EnableDiscoveryClient* or *@EnableEurekaClient* on configuration or starter
  - Configure some properties

```
spring:
  application:
    name: spring-cloud-eureka-client
server:
  port: 0
eureka:
  client:
    serviceUrl:
      defaultZone: ${EUREKA_URI:http://localhost:8761/eureka}
  instance:
    preferIpAddress: true
```

- registerWithEureka and fetchRegistry are default to
- preferIpAddress: the Eureka will register the application by hostname by default. However it is not applicable if we run our application on some Virtual Machines where they don't register themselves on DNS

FEIGN CLIENT

- In order to communicate with each others, we need a way to call other services
- Feign — a declarative HTTP client developed by Netflix
- *Feign* aims at simplifying *HTTP API* clients.
    - Simply put, the developer needs only to declare and annotate an interface while the actual implementation will be provisioned at runtime.
- Think of Feign as discovery-aware *Spring RestTemplate* using interfaces to communicate with endpoints. This interfaces will be automatically implemented at runtime and instead of service-urls, it is using service-names
- Without *Feign* we would have to autowire an instance of *EurekaClient* into our controller with which we could receive a service-information by *service-name* as an *Application* object

FEIGN CLIENT

- Without Eureka
  - We will use the *@RequestLine* annotation to specify the *HTTP verb* and a path part as argument, and the parameters will be modeled using the *@Param* annotation

```
public interface BookClient {
    @RequestLine("GET /{isbn}")
    BookResource findByIsbn(@Param("isbn") String isbn);

    @RequestLine("GET")
    List<BookResource> findAll();

    @RequestLine("POST")
    @Headers("Content-Type: application/json")
    void create(Book book);
}
```

  - Now we'll use the *Feign.builder()* to configure our interface-based client. The actual implementation will be provisioned at runtime

```
BookClient bookClient = Feign.builder()
  .client(new OkHttpClient())
  .encoder(new GsonEncoder())
  .decoder(new GsonDecoder())
  .logger(new Slf4jLogger(BookClient.class))
  .logLevel(Logger.Level.FULL)
  .target(BookClient.class, "http://localhost:8081/api/books");
```

FEIGN CLIENT

- With Eureka
  - We can specify the services/application name with *@FeignClient* and use *@RequestMapping* to specify the URL

```java
@FeignClient("spring-cloud-eureka-client")
public interface GreetingClient {
    @RequestMapping("/greeting")
    String greeting();
}
```

- We also need to provide some properties so that the Eureka Server can find where it is

```yaml
spring:
  application:
    name: spring-cloud-eureka-feign-client
server:
  port: 8080
eureka:
  client:
    serviceUrl:
      defaultZone: ${EUREKA_URI:http://localhost:8761/eureka}
```

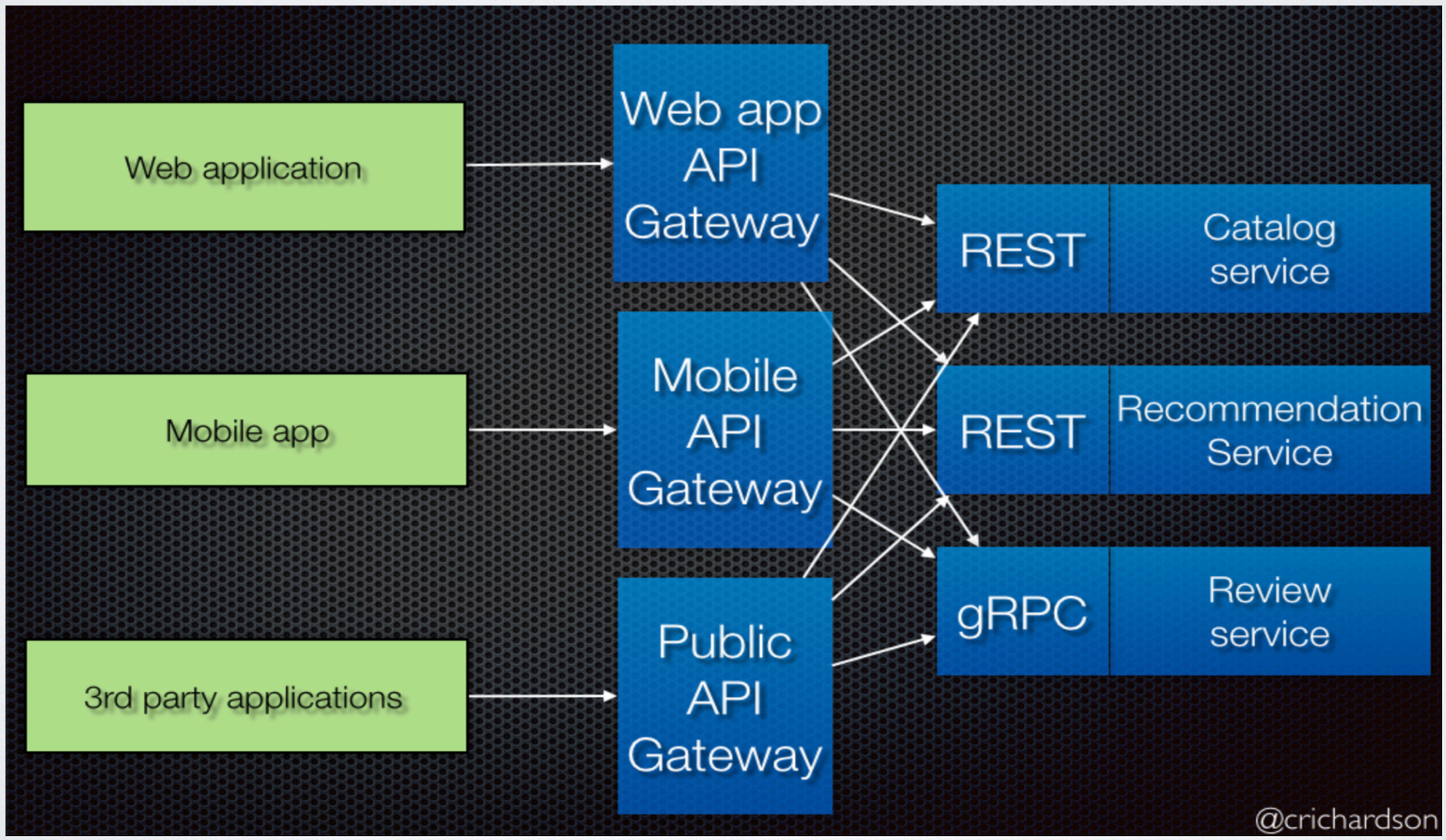
SCENARIO

- Assuming we have 10 services and each service provides 10 entry points, so we will have 100 different URL.

- As a client/consumer, how will I know which URL mapped to which service/application?

GATEWAY

- API gateway is the single entry point for all clients.
- The API gateway handles requests in one of two ways.
    - Some requests are simply proxied/routed to the appropriate service.
    - It handles other requests by fanning out to multiple services



GATEWAY

- When it comes to choosing an API gateway for our services/application, there are a variety of options:
    - *Zuul* from netflix, *Kong*, *Nginx*, *HAProxy*, *Traefik*
    - Cloud vendor's solutions such as Amazon API Gateway or Google Cloud Endpoints
    - Spring Cloud gateway
- I will introduce Spring Cloud gateway here (All of these gateways works in the same way with different configurations)

GATEWAY

- Spring Cloud Gateway has three main building blocks
  - *Route* – the primary API of the gateway. It is defined by a given identification (ID), a destination (URI) and set of predicates and filters
  - *Predicate* – a Java 8's *Predicate* – which is used for matching HTTP requests using headers, methods or parameters
  - *Filter* – a standard Spring's *WebFilter*
- Assume we have one service called "movie-store"

```
spring:
  cloud:
    gateway:
      routes:
      - id: movie-store
        uri: lb://movie-store
        predicates:
        - Path=/api/movies/**
```

  - This configuration tells Spring Cloud Gateway to route all requests made to the gateway at /api/movies/ to the movie-store service

SCENARIO

- In monolithic application, it is easy to implement the security module, like session.

- But when it comes to a distributed system, it is not possible to pass session around different servers.

- How to solve the problem?

MICROSERVICES

- An architectural style that structures an application as a collection of services that are
  - Highly maintainable and testable
  - Loosely coupled
  - Independently deployable
  - Organized around business capabilities
- Services communicate using either synchronous protocols such as HTTP/REST or asynchronous protocols such as AMQP

LET'S TAKE A LOOK AT AN EXAMPLE