Conf42 2024 | Feb 29 2024 | Online

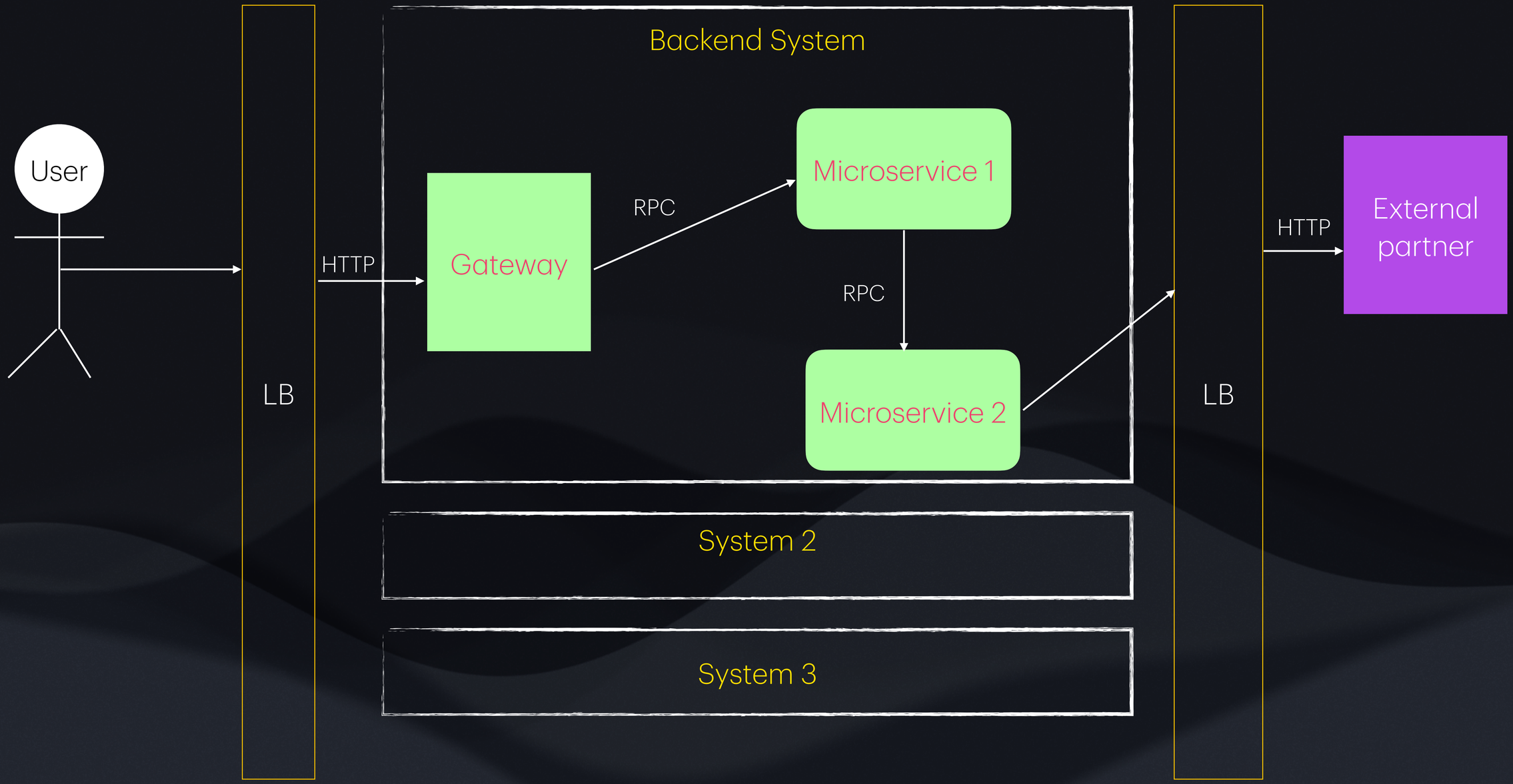
Advanced API Design for data-intensive distributed systems

Santosh Nikhil Kumar Adireddy
Engineering Lead, ByteDance

Chapter 1

REST for HTTP
And
Thrift for RPC

High level architecture of a modern distributed system



REST (for HTTP)

- Architectural style or design pattern for creating web services, implemented over HTTP communication protocol
- RESTful APIs use HTTP methods GET, POST, PUT, DELETE
- Stateless communication
- Resource-oriented design (through URIs)
- Flexibility to add authentication, authorization, rate limiting, caching, logging, and monitoring

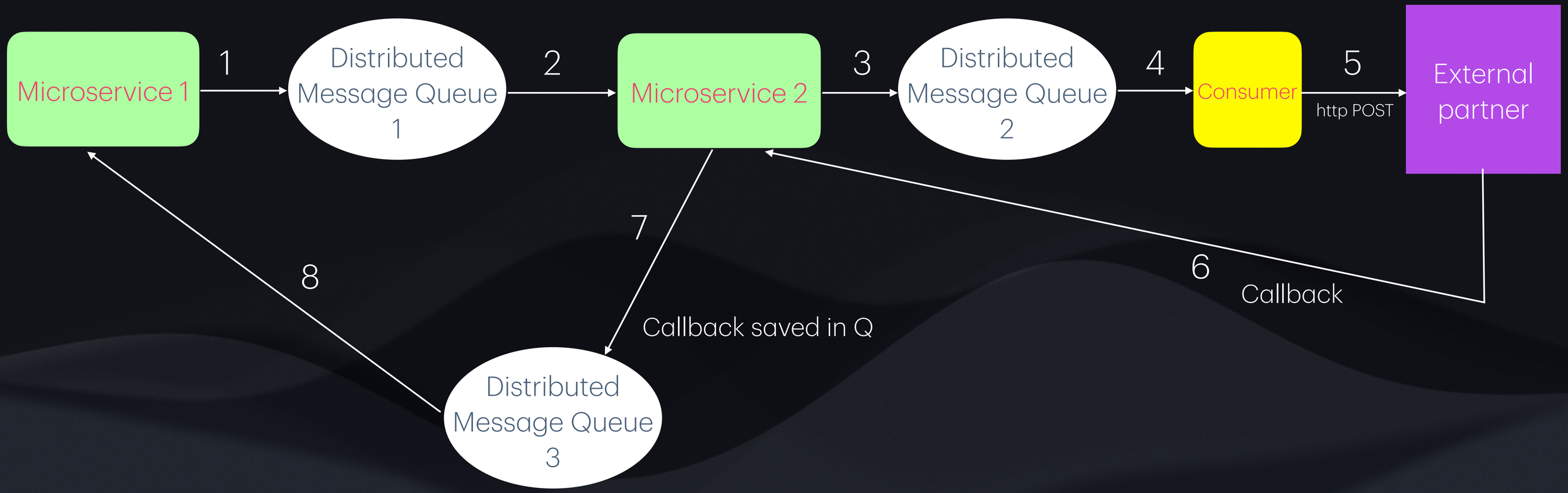
Thrift (for RPC)

- Software framework used for building services communicating through RPC protocol
- Code generation
- Cross-language support for inter-operability
- Efficient binary protocol for data serialization over network.
- Scalability - async comm, concurrent requests

Chapter 2

Design Asynchronous API calls with Callback

Async communication with callbacks



Example API design with callback

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

@SpringBootApplication
@RestController
public class AsyncCallbackAPI {

    private final ExecutorService executorService = Executors.newSingleThreadExecutor();

    public static void main(String[] args) {
        SpringApplication.run(AsyncCallbackAPI.class, args);
    }

    @PostMapping("/process")
    public ResponseEntity<String> processAsync(@RequestBody RequestData requestData) {
        // Simulate asynchronous processing
        executorService.submit(() -> {
            // Perform asynchronous processing
            // For demonstration, simply sleep for 5 seconds
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }

            // Invoke callback URL provided by the client
            invokeCallback(requestData.getCallbackUrl(), "Processing completed");
        });

        return ResponseEntity.accepted().body("Request accepted for processing");
    }

    private void invokeCallback(String callbackUrl, String message) {
        // Make an HTTP POST request to the callback URL
        // In a real-world scenario, you would use a HTTP client library like Apache HttpClient or Spring WebClient
        // Here, we're just printing the callback URL and message for demonstration purposes
        System.out.println("Invoking callback URL: " + callbackUrl);
        System.out.println("Callback message: " + message);
    }

    static class RequestData {
        private String callbackUrl;

        public String getCallbackUrl() {
            return callbackUrl;
        }

        public void setCallbackUrl(String callbackUrl) {
            this.callbackUrl = callbackUrl;
        }
    }
}
```

- The processAsync in the API endpoint handler*(step 5 in previous slide)*. It accepts a POST request with a JSON body containing a callbackUrl field.
- Upon receiving the request, the server starts asynchronous processing (simulated by sleeping for 5 seconds).
- After the processing is complete, the server invokes the callback URL provided by the client*(step 6 in previous slide)*.
- The invokeCallback method simulates making an HTTP POST request to the callback URL.

Callback

- Initiated by requests
- Inbound communication
- Push model
- Synchronous or Asynchronous
- Use case: API integrations in Payment systems

WebHook

- Initiated by events
- Outbound communication
- Pull model
- Asynchronous
- Use case: Event-driven architectures like sending GitHub update notifications to Slack

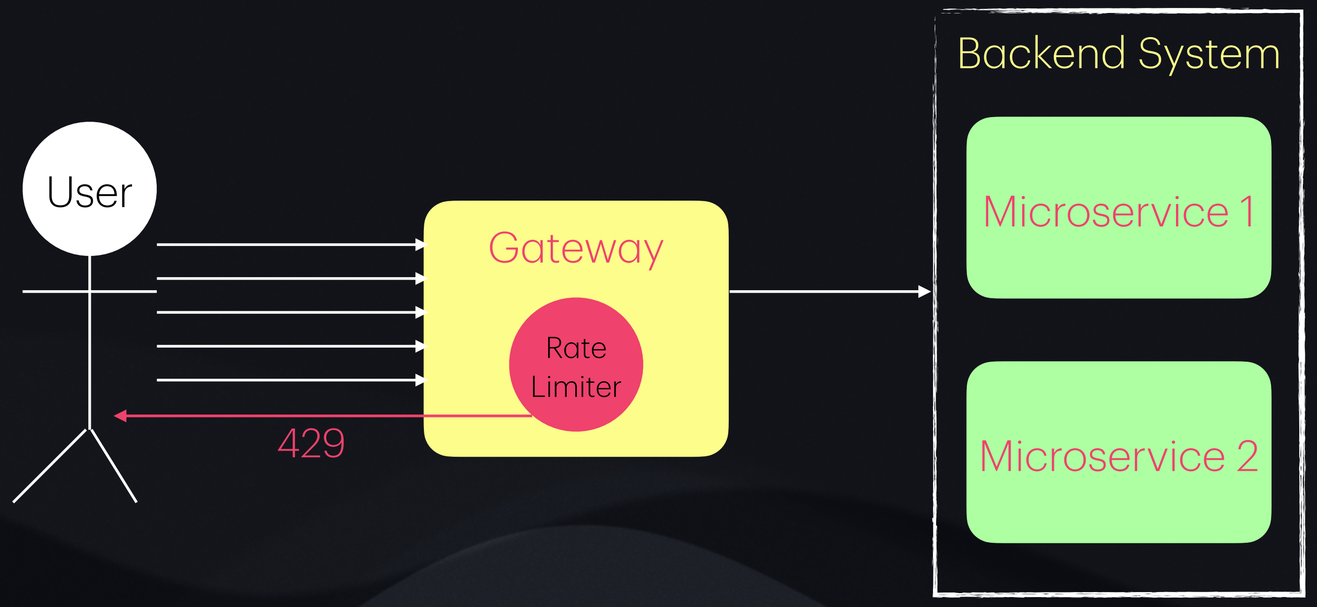
Chapter 3

Design Rate Limiting for APIs

What and Why Rate limiting

- What is it ?
  - Rate limiting restricts the number of requests a client can make within a specified time period.

- Why is it needed ?
  - Rate limiting prevents abuse, ensures fair resource usage, and protects the API from being overwhelmed by excessive requests.
  - It promotes stability, reliability of the distributed system and fair access to resources.



Example for API with Rate limiting

```
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@SpringBootApplication
@RestController
@RequestMapping("/api")
public class RateLimitedAPI {

    @GetMapping("/resource")
    @RateLimit(limit = 5, duration = 60) // 5 requests per 60
seconds
    public String getResource() {
        return "This is your resource.";
    }

    public static void main(String[] args) {
        SpringApplication.run(RateLimitedAPI.class, args);
    }
}
```

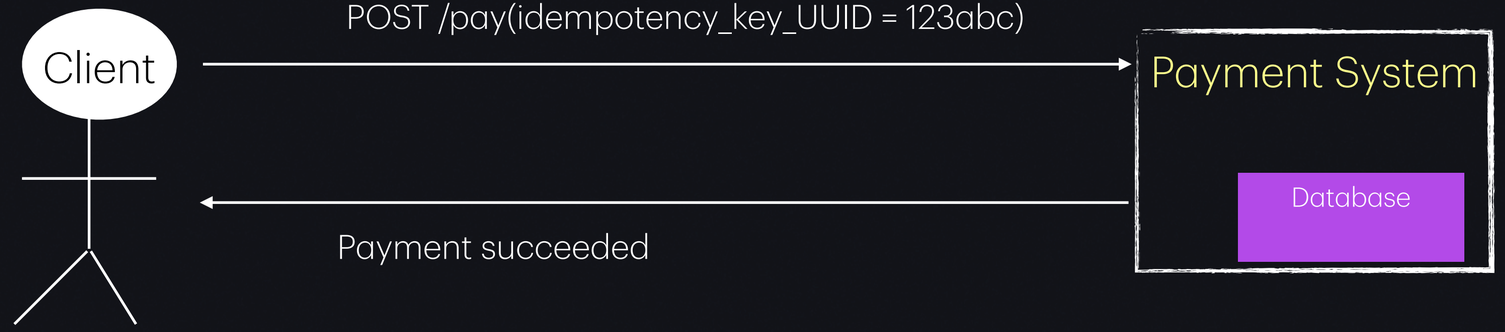
- Define a controller class RateLimitedAPI and specify the base path /api.
- Inside the controller, define a method getResource() which represents the API endpoint.
- Annotate the getResource() method with @RateLimit to apply rate limiting. We specify the limit (5 requests) and duration (60 seconds).

Chapter 4

Design Idempotency for APIs

Idempotency in Payment System

First payment attempt



Payment retry

Example API design with idempotency

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.HashMap;
import java.util.Map;

@RestController
@RequestMapping("/users")
public class UserController {

    private Map<String, String> userMap = dal.getDBTable();

    // Endpoint for updating user information (idempotent)
    @PutMapping("/{userId}")
    public ResponseEntity<String> updateUser(@PathVariable String userId, @RequestBody
String newName) {
        if (userMap.containsKey(userId) && userMap.get(userId).equals(newName)) {
            // If the user information is already up to date, return success
            return ResponseEntity.ok("User information already up to date: " + userMap);
        } else {
            // Perform the update operation
            userMap.put(userId, newName);
            return ResponseEntity.ok("User information updated successfully: " + userMap);
        }
    }
}
```

- The updateUser API handler is annotated with @PutMapping to handle HTTP PUT API requests to the /users/{userId} endpoint.
- The userId is extracted from the path variable, and the new name is obtained from the request body.
- The method checks if the userMap contains the userId, and if the new name matches the existing name. If so, it returns a success response indicating that the user information is already up to date.
- If the user information needs to be updated, it performs the update operation by putting the userId and newName into the userMap and returns a success response.