Spring Kafka Advanced Features: Complete Developer Guide

A comprehensive guide covering advanced Spring Kafka features including ReplyingKafkaTemplate for request/reply patterns, Kafka Streams integration, Kafka Connect setup, and multi-tenancy architectures with extensive Java examples and production patterns.

Table of Contents

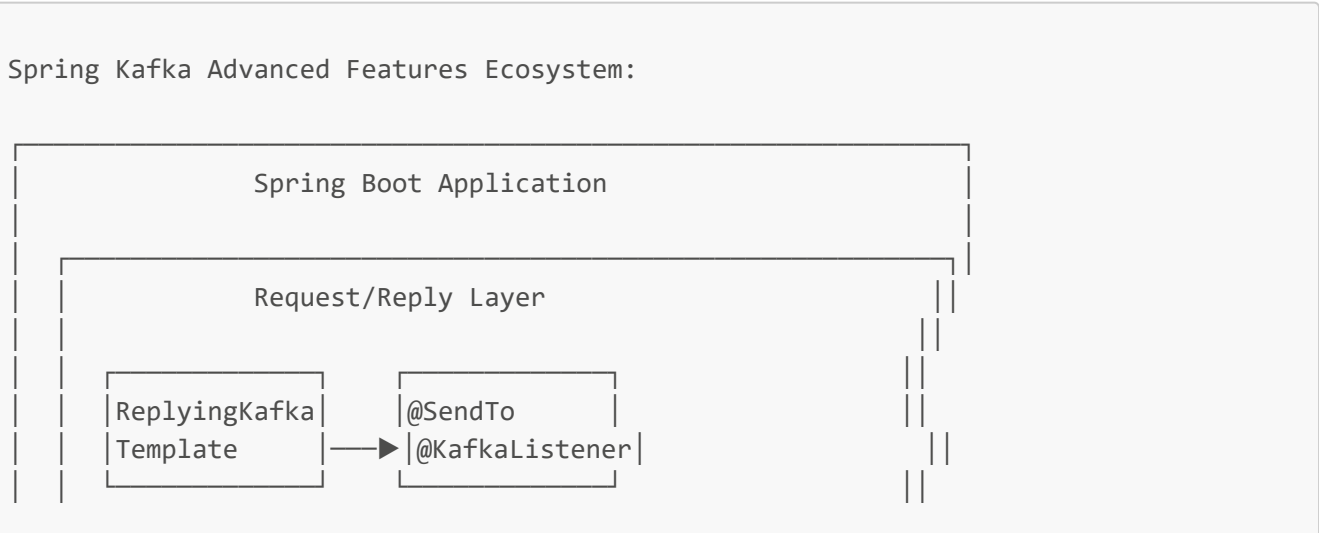
What are Spring Kafka Advanced Features?

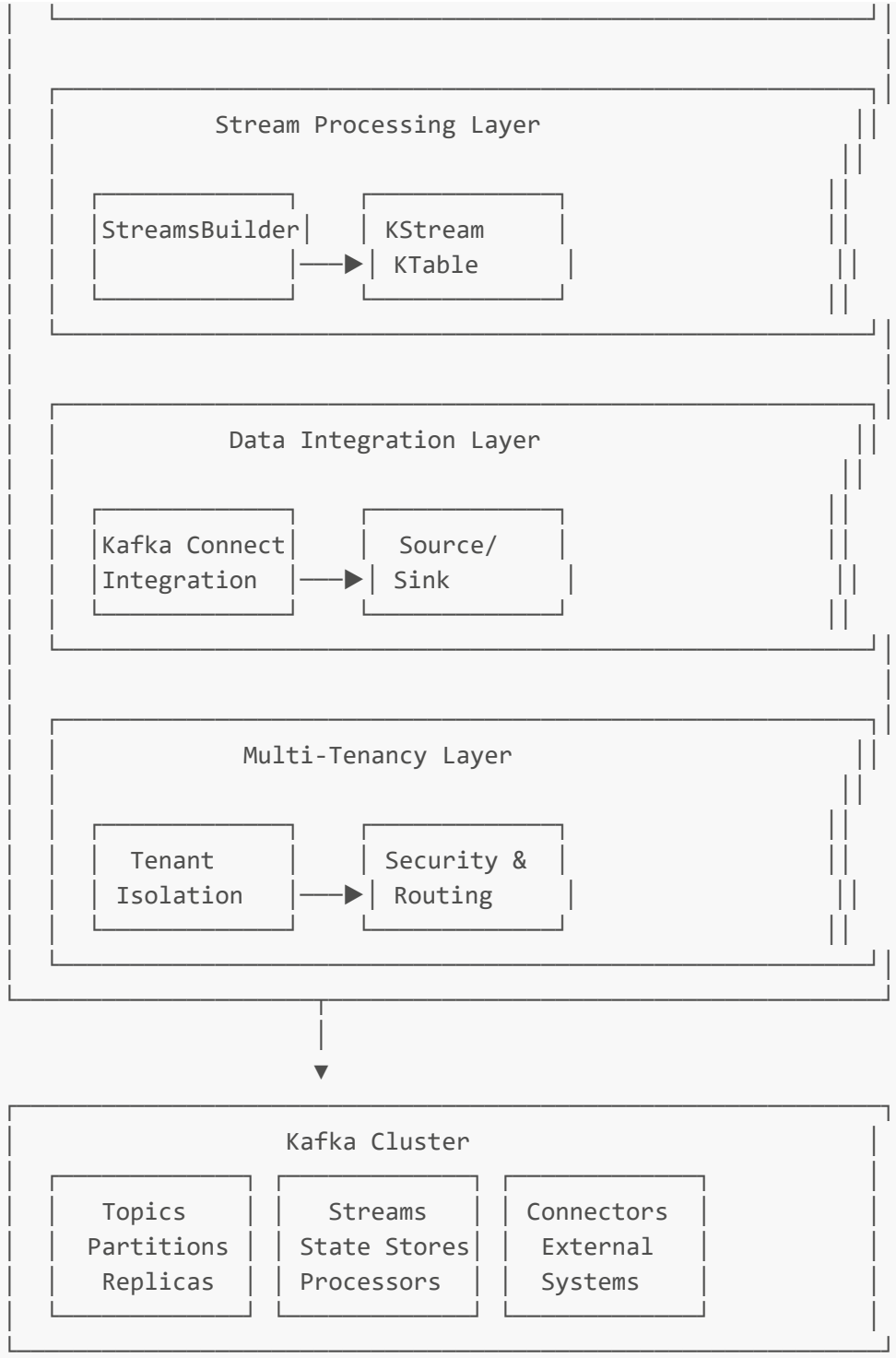
Simple Explanation: Spring Kafka Advanced Features extend beyond basic producer/consumer patterns to support complex enterprise architectures. These include synchronous request/reply communication, stream processing with Kafka Streams, data pipeline integration through Kafka Connect, and multi-tenant configurations for enterprise applications.

Why Advanced Features Exist:

- **Synchronous Communication**: Enable request/reply patterns over Kafka's async messaging
- **Stream Processing**: Real-time data transformation and analysis capabilities
- **Data Integration**: Seamless connection to external systems and databases
- **Enterprise Architecture**: Support for multi-tenant, scalable, and secure applications
- **Operational Excellence**: Production-ready patterns for complex scenarios

Advanced Features Architecture:





🔁 ReplyingKafkaTemplate (Request/Reply)

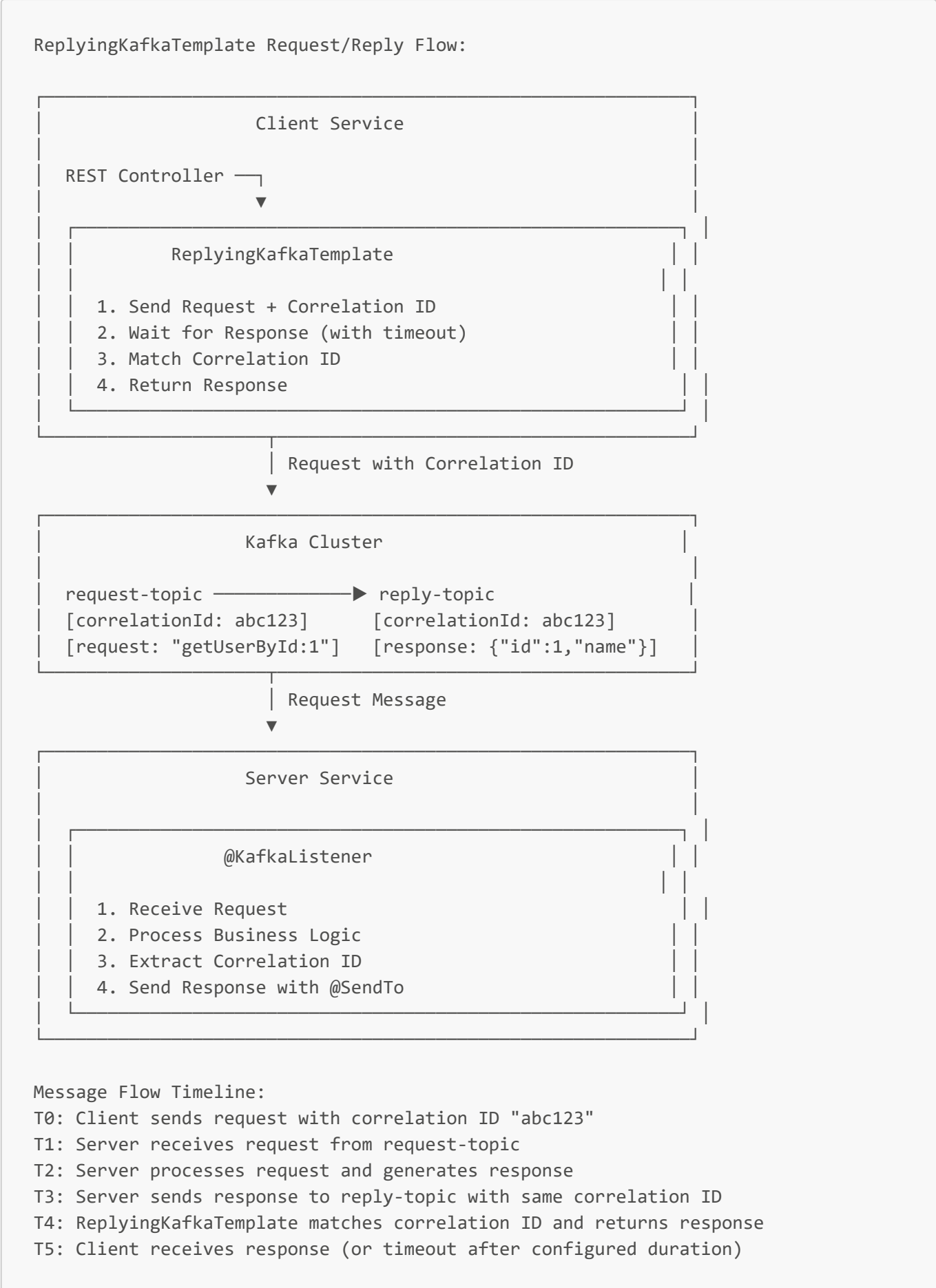
Simple Explanation: ReplyingKafkaTemplate extends KafkaTemplate to provide synchronous request/reply messaging patterns over Kafka. It sends a message to a request topic and waits for a response on a reply topic, using correlation IDs to match requests with responses.

What Problem It Solves:

- **Synchronous Communication**: Enables sync patterns over async Kafka messaging
- **Service Integration**: Facilitates RPC-like communication between microservices
- **Legacy System Integration**: Bridges sync systems with event-driven architectures

- **API Gateway Patterns**: Supports API gateways that need sync responses

Request/Reply Architecture:



Complete ReplyingKafkaTemplate Implementation

```
import org.springframework.kafka.requestreply.ReplyingKafkaTemplate;
import org.springframework.kafka.requestreply.RequestReplyFuture;
import org.springframework.kafka.support.SendResult;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.listener.ContainerProperties;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;

/**
 * Comprehensive ReplyingKafkaTemplate configuration
 */
@Configuration
@EnableKafka
@lombok.extern.slf4j.Slf4j
public class RequestReplyKafkaConfiguration {

    @Value("${kafka.request.topic:request-topic}")
    private String requestTopic;

    @Value("${kafka.reply.topic:reply-topic}")
    private String replyTopic;

    @Value("${kafka.request.timeout-ms:30000}")
    private Long requestTimeoutMs;

    /**
     * Producer factory for request messages
     */
    @Bean
    public ProducerFactory<String, Object> requestProducerFactory() {

        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaBootstrapServers);
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // Optimized settings for request/reply
        configProps.put(ProducerConfig.ACKS_CONFIG, "all");
        configProps.put(ProducerConfig.RETRIES_CONFIG, 3);
        configProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
        configProps.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 1);

        // Lower latency settings for sync communication
        configProps.put(ProducerConfig.LINGER_MS_CONFIG, 0);
        configProps.put(ProducerConfig.BATCH_SIZE_CONFIG, 1024);

        return new DefaultKafkaProducerFactory<>(configProps);
    }
```

```
    /**
     * Consumer factory for reply messages
     */
    @Bean
    public ConsumerFactory<String, Object> replyConsumerFactory() {

        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaBootstrapServers);
        configProps.put(ConsumerConfig.GROUP_ID_CONFIG, "reply-consumer-group");
        configProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        configProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // Optimized settings for reply consumption
        configProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        configProps.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        configProps.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, 1000);

        // Lower latency settings
        configProps.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1);
        configProps.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 100);

        return new DefaultKafkaConsumerFactory<>(configProps);
    }

    /**
     * Reply listener container for ReplyingKafkaTemplate
     */
    @Bean
    public ConcurrentMessageListenerContainer<String, Object>
replyListenerContainer() {

        ContainerProperties containerProperties = new
ContainerProperties(replyTopic);
        containerProperties.setGroupId("reply-consumer-group-" +
UUID.randomUUID());
        containerProperties.setMissingTopicsFatal(false);
        containerProperties.setAckMode(ContainerProperties.AckMode.RECORD);

        ConcurrentMessageListenerContainer<String, Object> container =
            new ConcurrentMessageListenerContainer<>(replyConsumerFactory(),
containerProperties);

        container.setConcurrency(3);
        container.setAutoStartup(true);

        return container;
    }

    /**
     * ReplyingKafkaTemplate bean configuration
```

```
     */
    @Bean
    public ReplyingKafkaTemplate<String, Object, Object> replyingKafkaTemplate() {

        ReplyingKafkaTemplate<String, Object, Object> template =
            new ReplyingKafkaTemplate<>(requestProducerFactory(),
replyListenerContainer());

        // Set default request timeout
        template.setDefaultReplyTimeout(Duration.ofMillis(requestTimeoutMs));

        // Set default topic
        template.setDefaultTopic(requestTopic);

        // Set message converter for JSON handling
        template.setMessageConverter(new StringJsonMessageConverter());

        return template;
    }

    /**
     * Regular KafkaTemplate for reply messages
     */
    @Bean
    public KafkaTemplate<String, Object> replyKafkaTemplate() {

        KafkaTemplate<String, Object> template = new KafkaTemplate<>
(requestProducerFactory());
        template.setDefaultTopic(replyTopic);
        template.setMessageConverter(new StringJsonMessageConverter());

        return template;
    }

    /**
     * Topics configuration
     */
    @Bean
    public NewTopic requestTopic() {
        return TopicBuilder.name(requestTopic)
            .partitions(6)
            .replicas(3)
            .config(TopicConfig.RETENTION_MS_CONFIG, "3600000") // 1 hour
retention
            .build();
    }

    @Bean
    public NewTopic replyTopic() {
        return TopicBuilder.name(replyTopic)
            .partitions(6)
            .replicas(3)
            .config(TopicConfig.RETENTION_MS_CONFIG, "3600000") // 1 hour
retention
```

```
                .build();
    }
}

/**
 * Request/Reply service implementation (Client side)
 */
@Service
@lombok.extern.slf4j.Slf4j
public class RequestReplyClientService {

    @Autowired
    private ReplyingKafkaTemplate<String, Object, Object> replyingKafkaTemplate;

    @Value("${kafka.request.topic:request-topic}")
    private String requestTopic;

    /**
     * Synchronous request/reply method
     */
    public <T> T sendRequestAndWaitForReply(String key, Object request, Class<T>
responseType)
            throws InterruptedException, ExecutionException, TimeoutException {

        log.info("Sending request: key={}, request={}", key, request);

        // Create producer record
        ProducerRecord<String, Object> record = new ProducerRecord<>(requestTopic,
key, request);

        // Add custom headers
        record.headers().add("request-type",
request.getClass().getSimpleName().getBytes());
        record.headers().add("timestamp",
String.valueOf(System.currentTimeMillis()).getBytes());
        record.headers().add("client-id", "request-reply-client".getBytes());

        try {
            // Send and wait for reply
            RequestReplyFuture<String, Object, Object> future =
                replyingKafkaTemplate.sendAndReceive(record);

            // Get the send result (for monitoring)
            SendResult<String, Object> sendResult = future.getSendFuture().get(5,
TimeUnit.SECONDS);
            log.debug("Request sent successfully: partition={}, offset={}",
                sendResult.getRecordMetadata().partition(),
                sendResult.getRecordMetadata().offset());

            // Wait for reply with timeout
            ConsumerRecord<String, Object> replyRecord = future.get(30,
TimeUnit.SECONDS);

            log.info("Received reply: key={}, value={}, partition={}, offset={}",
```

```
                replyRecord.key(), replyRecord.value(),
                replyRecord.partition(), replyRecord.offset());

            // Convert response
            Object response = replyRecord.value();
            if (responseType.isInstance(response)) {
                return responseType.cast(response);
            } else {
                // Use ObjectMapper for conversion if needed
                ObjectMapper mapper = new ObjectMapper();
                return mapper.convertValue(response, responseType);
            }

        } catch (TimeoutException e) {
            log.error("Request timed out: key={}, request={}", key, request);
            throw new TimeoutException("Request/reply timed out after 30
seconds");
        } catch (ExecutionException e) {
            log.error("Request execution failed: key={}, request={}", key,
request, e);
            throw new ExecutionException("Request execution failed",
e.getCause());
        }
    }

    /**
     * Asynchronous request/reply with callback
     */
    public void sendRequestWithCallback(String key, Object request,
                                        Consumer<Object> onSuccess,
                                        Consumer<Throwable> onFailure) {

        log.info("Sending async request: key={}, request={}", key, request);

        ProducerRecord<String, Object> record = new ProducerRecord<>(requestTopic,
key, request);
        record.headers().add("async", "true".getBytes());

        RequestReplyFuture<String, Object, Object> future =
            replyingKafkaTemplate.sendAndReceive(record);

        // Handle send result
        future.getSendFuture().addCallback(
            sendResult -> log.debug("Async request sent: offset={}",
                sendResult.getRecordMetadata().offset()),
            failure -> log.error("Failed to send async request", failure)
        );

        // Handle reply asynchronously
        CompletableFuture.supplyAsync(() -> {
            try {
                return future.get(30, TimeUnit.SECONDS);
            } catch (Exception e) {
                throw new RuntimeException(e);
```

```
            }
        }).whenComplete((replyRecord, throwable) -> {
            if (throwable != null) {
                onFailure.accept(throwable);
            } else {
                onSuccess.accept(replyRecord.value());
            }
        });
    }

    /**
     * Batch request/reply operations
     */
    public List<CompletableFuture<Object>> sendBatchRequests(List<KeyValueRequest>
requests) {

        log.info("Sending batch requests: count={}", requests.size());

        return requests.stream()
            .map(req -> CompletableFuture.supplyAsync(() -> {
                try {
                    return sendRequestAndWaitForReply(req.getKey(),
req.getValue(), Object.class);
                } catch (Exception e) {
                    log.error("Batch request failed: key={}", req.getKey(), e);
                    throw new RuntimeException(e);
                }
            }))
            .collect(Collectors.toList());
    }
}

/**
 * Request/Reply server implementation (Server side)
 */
@Service
@lombok.extern.slf4j.Slf4j
public class RequestReplyServerService {

    /**
     * Handle user service requests
     */
    @KafkaListener(topics = "${kafka.request.topic:request-topic}",
                   groupId = "user-service-group")
    @SendTo // Default reply topic from ReplyingKafkaTemplate
    public UserResponse handleUserRequest(@Payload UserRequest userRequest,
                                          @Header(KafkaHeaders.RECEIVED_TOPIC)
String topic,
                                          @Header(KafkaHeaders.RECEIVED_PARTITION)
int partition,
                                          @Header(KafkaHeaders.OFFSET) long offset,
                                          @Header(name = "request-type", required =
false) String requestType) {
```

```
        log.info("Processing user request: userId={}, operation={}, topic={},
offset={}",
            userRequest.getUserId(), userRequest.getOperation(), topic, offset);

        try {
            UserResponse response = processUserRequest(userRequest);

            log.info("User request processed successfully: userId={},
responseType={}",
                userRequest.getUserId(), response.getClass().getSimpleName());

            return response;

        } catch (Exception e) {
            log.error("Error processing user request: userId={}, operation={}",
                userRequest.getUserId(), userRequest.getOperation(), e);

            return UserResponse.builder()
                .userId(userRequest.getUserId())
                .status("ERROR")
                .message("Request processing failed: " + e.getMessage())
                .timestamp(Instant.now())
                .build();
        }
    }

    /**
     * Handle order service requests
     */
    @KafkaListener(topics = "${kafka.request.topic:request-topic}",
                   groupId = "order-service-group")
    @SendTo("${kafka.reply.topic:reply-topic}")
    public OrderResponse handleOrderRequest(@Payload OrderRequest orderRequest,
                                            @Header(KafkaHeaders.CORRELATION_ID)
byte[] correlationId) {

        String correlationIdStr = new String(correlationId);
        log.info("Processing order request: orderId={}, correlationId={}",
            orderRequest.getOrderId(), correlationIdStr);

        try {
            OrderResponse response = processOrderRequest(orderRequest);
            response.setCorrelationId(correlationIdStr);

            return response;

        } catch (Exception e) {
            log.error("Error processing order request: orderId={}",
orderRequest.getOrderId(), e);

            return OrderResponse.builder()
                .orderId(orderRequest.getOrderId())
                .status("FAILED")
                .errorMessage(e.getMessage())
```

```
                .correlationId(correlationIdStr)
                .timestamp(Instant.now())
                .build();
        }
    }

    /**
     * Generic request handler with dynamic routing
     */
    @KafkaListener(topics = "${kafka.request.topic:request-topic}",
                   groupId = "generic-service-group")
    @SendTo
    public Object handleGenericRequest(@Payload Map<String, Object> request,
                                       @Header(KafkaHeaders.RECEIVED_TOPIC) String
topic,
                                       @Header(name = "request-type", required =
false) String requestType) {

        log.info("Processing generic request: type={}, keys={}",
            requestType, request.keySet());

        String operation = (String) request.get("operation");

        try {
            switch (operation) {
                case "HEALTH_CHECK":
                    return Map.of(
                        "status", "UP",
                        "timestamp", Instant.now().toString(),
                        "service", "generic-request-handler"
                    );

                case "ECHO":
                    Map<String, Object> echoResponse = new HashMap<>(request);
                    echoResponse.put("echoed", true);
                    echoResponse.put("timestamp", Instant.now().toString());
                    return echoResponse;

                case "CALCULATE":
                    Double a = (Double) request.get("a");
                    Double b = (Double) request.get("b");
                    String operator = (String) request.get("operator");

                    return Map.of(
                        "result", performCalculation(a, b, operator),
                        "operation", String.format("%.2f %s %.2f", a, operator,
b),
                        "timestamp", Instant.now().toString()
                    );

                default:
                    return Map.of(
                        "error", "Unknown operation: " + operation,
                        "supportedOperations", Arrays.asList("HEALTH_CHECK",
```

```
"ECHO", "CALCULATE"),
                        "timestamp", Instant.now().toString()
                    );
            }

        } catch (Exception e) {
            log.error("Error processing generic request: operation={}", operation,
e);

            return Map.of(
                "error", "Request processing failed",
                "message", e.getMessage(),
                "timestamp", Instant.now().toString()
            );
        }
    }

    // Helper methods
    private UserResponse processUserRequest(UserRequest request) {
        // Simulate processing time
        try {
            Thread.sleep(100 + (long) (Math.random() * 200)); // 100-300ms
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        return UserResponse.builder()
            .userId(request.getUserId())
            .status("SUCCESS")
            .data(Map.of(
                "operation", request.getOperation(),
                "processed", true,
                "processingTime", "200ms"
            ))
            .timestamp(Instant.now())
            .build();
    }

    private OrderResponse processOrderRequest(OrderRequest request) {
        // Simulate processing time
        try {
            Thread.sleep(150 + (long) (Math.random() * 300)); // 150-450ms
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        return OrderResponse.builder()
            .orderId(request.getOrderId())
            .status("PROCESSED")
            .totalAmount(request.getTotalAmount())
            .timestamp(Instant.now())
            .build();
    }
```

```
    private Double performCalculation(Double a, Double b, String operator) {
        switch (operator) {
            case "+": return a + b;
            case "-": return a - b;
            case "*": return a * b;
            case "/": return b != 0 ? a / b : Double.NaN;
            default: throw new IllegalArgumentException("Unsupported operator: " +
operator);
        }
    }
}

/**
 * REST Controller demonstrating request/reply usage
 */
@RestController
@RequestMapping("/api/request-reply")
@lombok.extern.slf4j.Slf4j
public class RequestReplyController {

    @Autowired
    private RequestReplyClientService requestReplyService;

    /**
     * Synchronous user request endpoint
     */
    @PostMapping("/users")
    public ResponseEntity<UserResponse> processUserRequest(@RequestBody
UserRequest userRequest) {

        try {
            log.info("REST: Processing user request: userId={}",
userRequest.getUserId());

            UserResponse response =
requestReplyService.sendRequestAndWaitForReply(
                userRequest.getUserId(), userRequest, UserResponse.class);

            return ResponseEntity.ok(response);

        } catch (TimeoutException e) {
            log.error("Request timeout: userId={}", userRequest.getUserId());

            UserResponse errorResponse = UserResponse.builder()
                .userId(userRequest.getUserId())
                .status("TIMEOUT")
                .message("Request timed out")
                .timestamp(Instant.now())
                .build();

            return
ResponseEntity.status(HttpStatus.REQUEST_TIMEOUT).body(errorResponse);

        } catch (Exception e) {
```

```
            log.error("Request failed: userId={}", userRequest.getUserId(), e);

            UserResponse errorResponse = UserResponse.builder()
                .userId(userRequest.getUserId())
                .status("ERROR")
                .message("Request processing failed")
                .timestamp(Instant.now())
                .build();

            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(errorResponse);
        }
    }

    /**
     * Asynchronous request endpoint
     */
    @PostMapping("/users/async")
    public ResponseEntity<Map<String, String>>
processUserRequestAsync(@RequestBody UserRequest userRequest) {

        String requestId = UUID.randomUUID().toString();

        log.info("REST: Processing async user request: userId={}, requestId={}",
            userRequest.getUserId(), requestId);

        requestReplyService.sendRequestWithCallback(
            userRequest.getUserId(),
            userRequest,
            response -> log.info("Async response received: userId={}, requestId=
{}",
                userRequest.getUserId(), requestId),
            failure -> log.error("Async request failed: userId={}, requestId={}",
                userRequest.getUserId(), requestId, failure)
        );

        return ResponseEntity.accepted().body(Map.of(
            "requestId", requestId,
            "status", "ACCEPTED",
            "message", "Request is being processed asynchronously"
        ));
    }

    /**
     * Generic calculation endpoint
     */
    @PostMapping("/calculate")
    public ResponseEntity<Map<String, Object>> calculate(@RequestBody Map<String,
Object> calculationRequest) {

        try {
            log.info("REST: Processing calculation request");

            @SuppressWarnings("unchecked")
```

```
            Map<String, Object> response = (Map<String, Object>)
requestReplyService.sendRequestAndWaitForReply(
                "calculation", calculationRequest, Map.class);

            return ResponseEntity.ok(response);

        } catch (Exception e) {
            log.error("Calculation request failed", e);

            Map<String, Object> errorResponse = Map.of(
                "error", "Calculation failed",
                "message", e.getMessage(),
                "timestamp", Instant.now().toString()
            );

            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(errorResponse);
        }
    }

    /**
     * Health check endpoint
     */
    @GetMapping("/health")
    public ResponseEntity<Map<String, Object>> healthCheck() {

        try {
            Map<String, Object> healthRequest = Map.of("operation",
"HEALTH_CHECK");

            @SuppressWarnings("unchecked")
            Map<String, Object> response = (Map<String, Object>)
requestReplyService.sendRequestAndWaitForReply(
                "health", healthRequest, Map.class);

            return ResponseEntity.ok(response);

        } catch (Exception e) {
            log.error("Health check failed", e);

            Map<String, Object> errorResponse = Map.of(
                "status", "DOWN",
                "error", e.getMessage(),
                "timestamp", Instant.now().toString()
            );

            return
ResponseEntity.status(HttpStatus.SERVICE_UNAVAILABLE).body(errorResponse);
        }
    }
}

// Supporting data classes
@lombok.Data
```

```
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class UserRequest {
    private String userId;
    private String operation;
    private Map<String, Object> parameters;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class UserResponse {
    private String userId;
    private String status;
    private String message;
    private Map<String, Object> data;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderRequest {
    private String orderId;
    private String customerId;
    private BigDecimal totalAmount;
    private List<String> items;
    private Instant timestamp;
}

@lombok.Data
@lombok.Builder
@lombok.NoArgsConstructor
@lombok.AllArgsConstructor
class OrderResponse {
    private String orderId;
    private String status;
    private String errorMessage;
    private BigDecimal totalAmount;
    private String correlationId;
    private Instant timestamp;
}

@lombok.Data
@lombok.AllArgsConstructor
@lombok.NoArgsConstructor
class KeyValueRequest {
    private String key;
    private Object value;
}
```

This completes Part 1 covering ReplyingKafkaTemplate. The guide continues with Kafka Streams Integration, Kafka Connect, and Multi-tenancy in subsequent parts.