UNIVERSIDAD EL BOSQUE

BOGOTÁ D.C

FACULTAD DE INGENIERÍA

INGENIERÍA DE SISTEMAS

TALLER 1: ESCENARIOS DE CALIDAD - RENDIMIENTO

INGENIERÍA DE SOFTWARE 2

Profesor:

Ing. Andres Felipe Rey Chaves (areyc@unbosque.edu.co).

Estudiantes:

Andres Felipe Espitia Rodriguez(aespitiar@unbosque.edu.co)

Johan Sebastian Gomez Beltran (jsgomezb@unbosque.edu.co)

Kevin David Peña Bustos (kpenab@unbosque.edu.co)

David Esteban López Castillo (delopezc@unbosque.edu.co)

SEPTIEMBRE 2024

Tabla de contenido

Medición de la API

Se realizarán pruebas de concurrencia para probar el rendimiento de la api en su primera versión, esto se hará mediante la herramienta Apache JMeter 5.6.3.

Componentes principales

- Usuarios simulados: Se configuraron diferentes grupos de usuarios para realizar solicitudes HTTP.

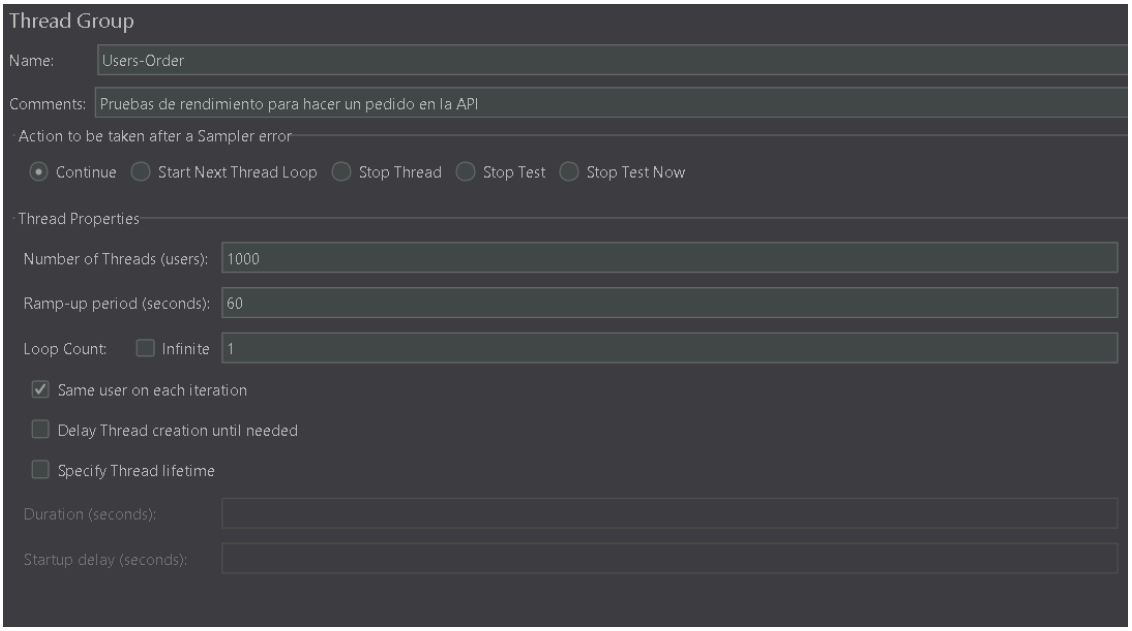


Figura 1. Usuarios simulados en la aplicación

- Solicitudes HTTP: Se prueba la operación procesar

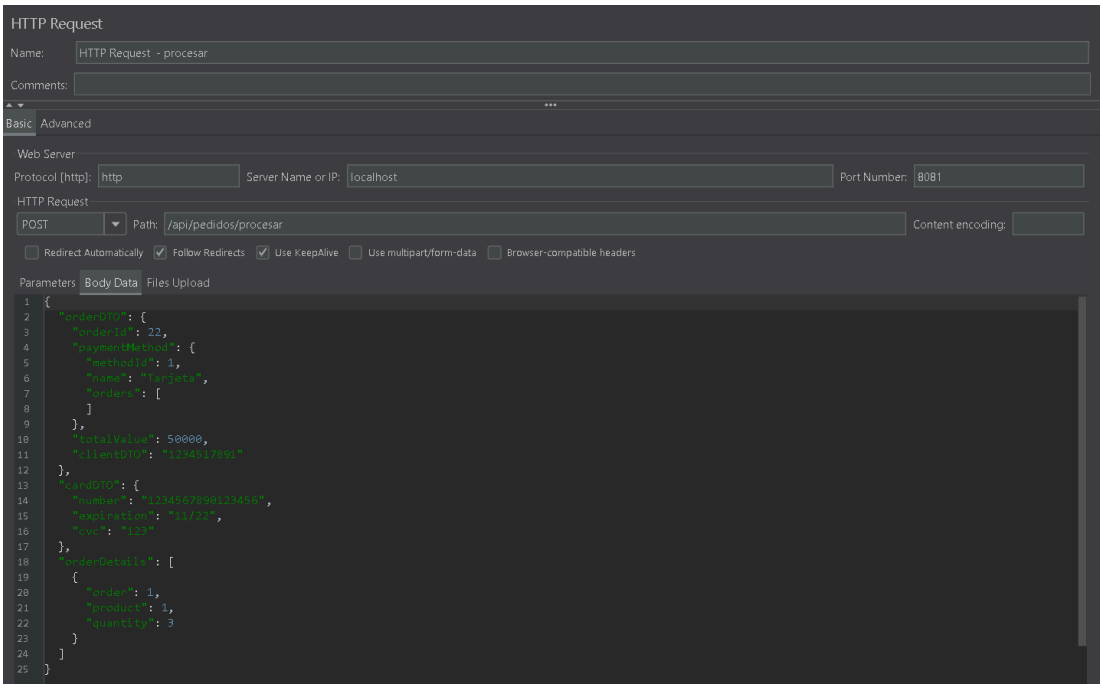
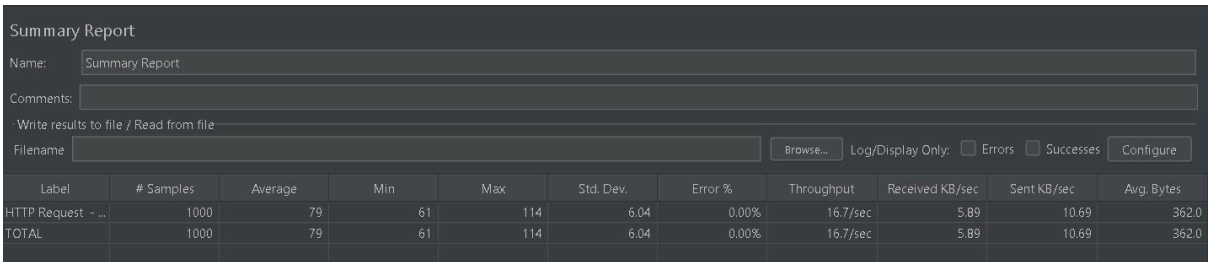


Figura 2. Solicitudes que simulará el aplicativo

- Listeners: Se emplearon listeners como el Summary Report y Response Time Graph para registrar y visualizar los resultados de las pruebas.

Figura 3. Listeners de JMeter

- Número de muestras: Se realizaron 1000 solicitudes en total a lo largo de 60 segundos.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request - ...	1000	79	61	114	6.04	0.00%	16.7/sec	5.89	10.69	362.0
TOTAL	1000	79	61	114	6.04	0.00%	16.7/sec	5.89	10.69	362.0

Figura 4. Número de solicitudes simuladas

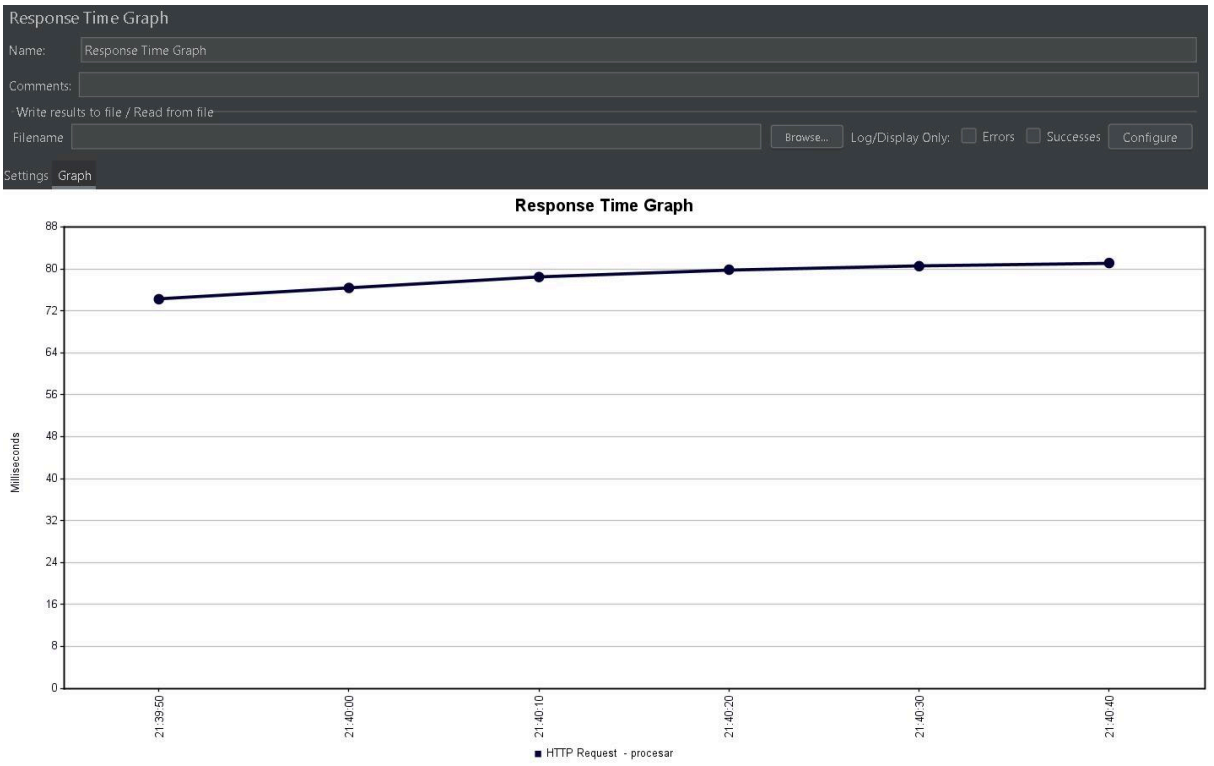


Figura 5. Resultado obtenido

Análisis de Resultados

- **Rendimiento General:** El tiempo de respuesta promedio de 79 ms se considera aceptable, lo que sugiere que el servidor responde bien bajo la carga de 1000 solicitudes. El tiempo máximo fue de 114 ms no es considerablemente más alto que el promedio.
- **Consistencia:** La desviación estándar de 6.04 ms indica cierta variabilidad en los tiempos de respuesta. El servidor manejó la mayoría de las solicitudes de manera eficiente y no hay tanta diferencia entre el tiempo mínimo y máximo.

Mejoras para futuras versiones

Optimización de rendimiento: Aunque el tiempo promedio es aceptable para cumplir con el atributo de calidad de rendimiento propuesto en el taller. Esto puede implicar la optimización del código, ajustes en la base de datos y/o mejoras en la infraestructura del servidor para reducir los picos de latencia. También hay que decir que no había tiempo de espera en el proceso de aceptación del pago lo que sí habrá en futuras versiones.

Experimentación: Análisis de diseño y estrategias de optimización

Estrategias seleccionadas e implementación, junto a la justificación de la misma

Estrategia 1 - Optimización de las estructuras de datos usadas:

En nuestro caso la optimización de estructuras de datos se presentó al momento de finalizar la versión 1, por lo que no contamos con un antes y un después, no obstante sí significó una mejora significativa dentro del rendimiento y la construcción de la aplicación, ya que evitaremos redundancias y en muchos casos errores de entendimiento a la hora de probar con JSONs.
En el caso de la primer instancia de redundancias, optamos por usar Sets en vez de List a la hora de hablar de relaciones de uno a muchos, donde un objeto puede tener muchos de otros objetos dentro o asociados a él, como lo es un cliente que puede tener muchos pedidos, pero estos no se pueden repetir.

```
68          /**
69           * Conjunto de órdenes realizadas por el cliente.
70           *
71           * Relación uno a muchos con la entidad @linkOrder.
72           */
73          @Schema(description = "Conjunto de órdenes realizadas por el cliente")
74          @OneToMany(fetch = FetchType.LAZY, mappedBy = "client", cascade = CascadeType.ALL)
75          private Set<Order> orders;
```

Figura 6. Ejemplo de un set usado en la entidad de Cliente

Por otro lado la optimización adicional es justamente sobre los objetos dentro de objetos, estos en los DTO, ya que debíamos ser lo más claros posibles a la hora de hacer un request, por lo que en los DTO no debíamos incluir en un request datos completos como por ejemplo el cliente al que está asociado un pedido, sin embargo la entidad si tiene estos datos completos, mientras que los DTO se pueden asociar directamente a Id de esa clase, sin tener que tener un atributo del objeto completo dentro del mismo

```java
public class Order {

    /**
     * Identificador único del pedido.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_pedido")
    @Schema(description = "Identificador único del pedido", example = "1")
    private Integer orderId;

    /**
     * Cliente asociado al pedido.
     */
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "id_cliente", referencedColumnName = "id_cliente")
    @Schema(description = "Cliente que realizó el pedido", example = "12345678")
    private Client client;

    /**
     * Método de pago utilizado en el pedido.
     */
    @JsonBackReference
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "id_met_pago")
    @Schema(description = "Método de pago utilizado en el pedido", example = "Tarjeta de crédito")
    private PaymentMethod paymentMethod;

    /**
     * Valor total del pedido.
     */
    @Column(name = "valor_total")
    @Schema(description = "Valor total del pedido", example = "150000")
    private Long totalValue;
```

Figura 7. Entidad del pedido con otro objeto cliente dentro de él

```java
public class OrderDTO {

        /** El identificador del pedido */
        @Schema(description = "Identificador único del pedido", example = "1001", required = true)
        @NotNull(message = "El pedido no tiene identificador")
        private Integer orderId;

        /** El nombre del cliente asociado al pedido */
        @Schema(description = "Nombre del cliente asociado al pedido", example = "Juan Perez", required = true)
        @NotBlank(message = "El cliente no está asociado")
        private String client;

        /** El método de pago utilizado en el pedido */
        @Schema(description = "Método de pago utilizado en el pedido", required = true)
        @JsonBackReference
        @NotNull(message = "Se debe asociar un método de pago")
        private PaymentMethod paymentMethod;

        /** El valor total del pedido */
        @Schema(description = "Valor total del pedido", example = "50000", required = true, minimum = "0")
        @Range(min = 0, message = "El valor de la compra no debe ser 0")
        private Long totalValue;
```

Figura 8. DTO de pedido, con la llave primaria en lugar del objeto

Estrategia 2 - Optimización de diseño:

Para esta estrategia tendríamos que hablar del diseño del programa, ya que teníamos muchos servicios inutilizados, lo que nos hacía peso de líneas de código que no se iban a utilizar, por lo que nos resultaban haciendo estorbo, de tal forma que los servicios, por ejemplo, pasaron de llevar una cantidad de clases extensa, a una cantidad más moderada de los mismos. Además esto va ligado con la siguiente estrategia que presentaremos en este informe.

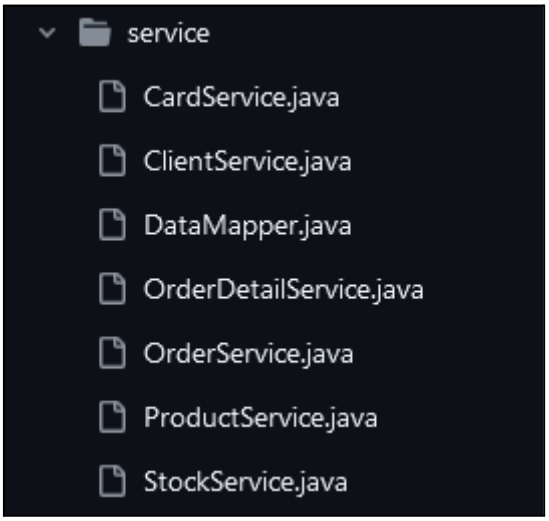


Figura 9. Paquete de servicios en la versión 1

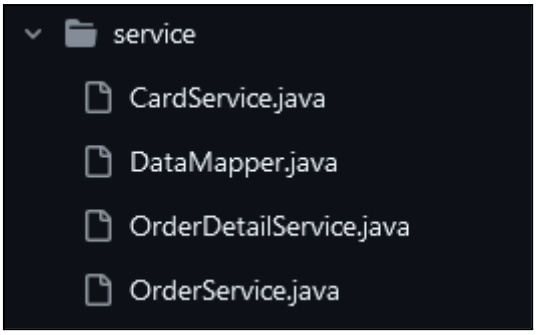


Figura 10. Paquete de servicios en la versión 2

Estrategia 3 - Optimización de código:

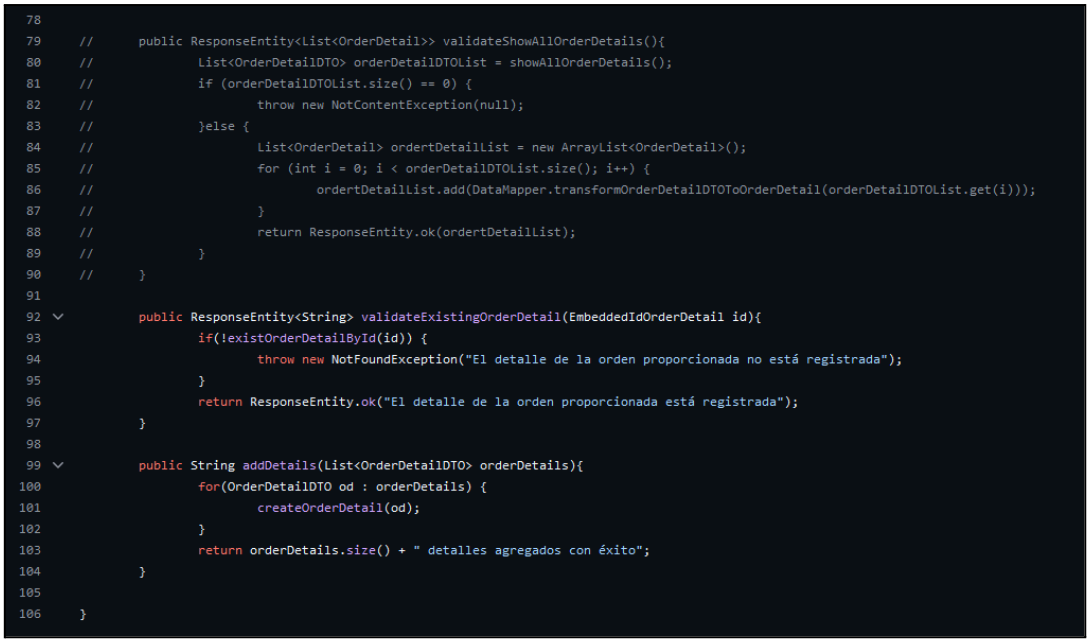
Ahora ya que tenemos optimizada esta parte del diseño, haciendo que los servicios no se sobrepusieron, por lo que el siguiente paso era sustraer el uso de esos servicios por otros, y hacer que los servicios fueran autosuficientes usando los repositorios de estos servicios, ya que de cierta forma esto es lo más óptimo, ya que en vez de usar un servicio que exclusivamente llama un método del repositorio, utilizar directamente este método desde el repositorio que voy a usar.

```
61      //      public ResponseEntity<List<Order>> validateShowAllOrders(){
62      //              List<OrderDTO> orderDTOList = showAllOrders();
63      //              if (orderDTOList.size() == 0) {
64      //                      throw new NotContentException(null);
65      //              }else {
66      //                      List<Order> ordertList = new ArrayList<Order>();
67      //                      for (int i = 0; i < orderDTOList.size(); i++) {
68      //                              ordertList.add(DataMapper.transformOrderDTOToOrder(orderDTOList.get(i)));
69      //                      }
70      //                      return ResponseEntity.ok(ordertList);           }
71      //      }
72
73          public ResponseEntity<String> validateExistingOrder(Integer id){
74                  if(!existOrderById(id)) {
75                          throw new NotFoundException("La orden proporcionada no está registrada");
76                  }
77                  return ResponseEntity.ok("La orden proporcionada está registrada");
78          }
79  }
```

Figura 11. Parte final del servicio de pedidos, con código comentado y 79 líneas de código

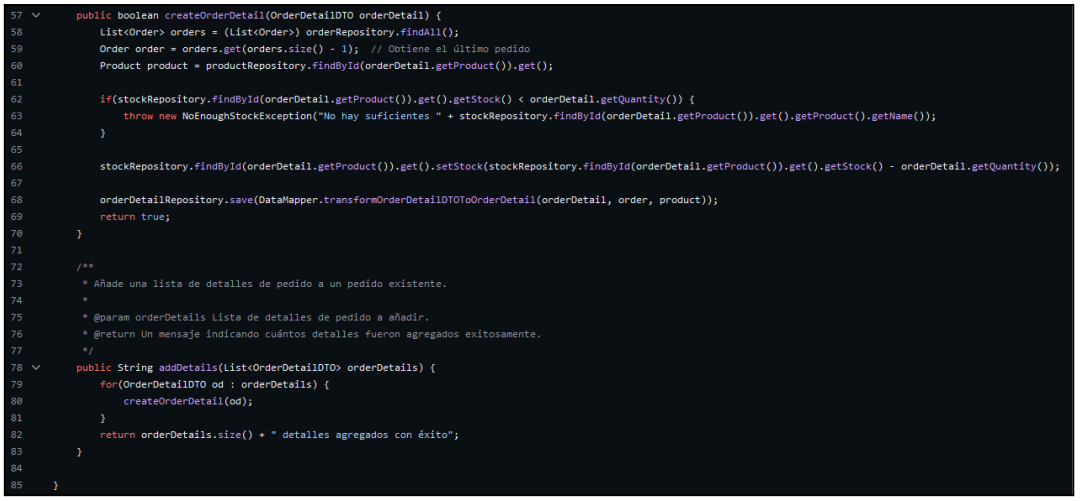
```
35          /**
36           * Crea un nuevo pedido basado en un objeto DTO de pedido.
37           *
38           * Busca el cliente correspondiente en el repositorio y luego guarda el pedido
39           * en el repositorio de pedidos.
40           *
41           * @param order Objeto @linkOrderDTO que contiene la información del pedido a
42           *              crear.
43           * @return @codetrue si el pedido se crea y guarda correctamente.
44           */
45          public boolean createOrder(OrderDTO order) {
46                  Client client = clientRepository.findById(order.getClientDTO()).get();
47                  orderRepository.save(DataMapper.transformOrderDTOToOrder(order, client, order.getPaymentMethod()));
48                  return true;
49          }
50  }
```

Figura 12. Nueva versión del servicio de pedidos, con menos líneas de código incluso con documentación



```
78
79      //      public ResponseEntity<List<OrderDetail>> validateShowAllOrderDetails(){
80      //              List<OrderDetailDTO> orderDetailDTOList = showAllOrderDetails();
81      //              if (orderDetailDTOList.size() == 0) {
82      //                      throw new NotContentException(null);
83      //              }else {
84      //                      List<OrderDetail> ordertDetailList = new ArrayList<OrderDetail>();
85      //                      for (int i = 0; i < orderDetailDTOList.size(); i++) {
86      //                              ordertDetailList.add(DataMapper.transformOrderDetailDTOToOrderDetail(orderDetailDTOList.get(i)));
87      //                      }
88      //                      return ResponseEntity.ok(ordertDetailList);
89      //              }
90      //      }
91
92          public ResponseEntity<String> validateExistingOrderDetail(EmbeddedIdOrderDetail id){
93                  if(!existOrderDetailById(id)) {
94                          throw new NotFoundException("El detalle de la orden proporcionada no está registrada");
95                  }
96                  return ResponseEntity.ok("El detalle de la orden proporcionada está registrada");
97          }
98
99          public String addDetails(List<OrderDetailDTO> orderDetails){
100                 for(OrderDetailDTO od : orderDetails) {
101                         createOrderDetail(od);
102                 }
103                 return orderDetails.size() + " detalles agregados con éxito";
104         }
105
106 }
```

Figura 13. Primera versión del servicio de detalles con 106 líneas de código



```
57      public boolean createOrderDetail(OrderDetailDTO orderDetail) {
58          List<Order> orders = (List<Order>) orderRepository.findAll();
59          Order order = orders.get(orders.size() - 1);  // Obtiene el último pedido
60          Product product = productRepository.findById(orderDetail.getProduct()).get();
61
62          if(stockRepository.findById(orderDetail.getProduct()).get().getStock() < orderDetail.getQuantity()) {
63              throw new NoEnoughStockException("No hay suficientes " + stockRepository.findById(orderDetail.getProduct()).get().getProduct().getName());
64          }
65
66          stockRepository.findById(orderDetail.getProduct()).get().setStock(stockRepository.findById(orderDetail.getProduct()).get().getStock() - orderDetail.getQuantity());
67
68          orderDetailRepository.save(DataMapper.transformOrderDetailDTOToOrderDetail(orderDetail, order, product));
69          return true;
70      }
71
72      /**
73       * Añade una lista de detalles de pedido a un pedido existente.
74       *
75       * @param orderDetails Lista de detalles de pedido a añadir.
76       * @return Un mensaje indicando cuántos detalles fueron agregados exitosamente.
77       */
78      public String addDetails(List<OrderDetailDTO> orderDetails) {
79          for(OrderDetailDTO od : orderDetails) {
80              createOrderDetail(od);
81          }
82          return orderDetails.size() + " detalles agregados con éxito";
83      }
84
85  }
```

Figura 14. Segunda versión del servicio de detalles, con 85 líneas de código incluyendo documentación

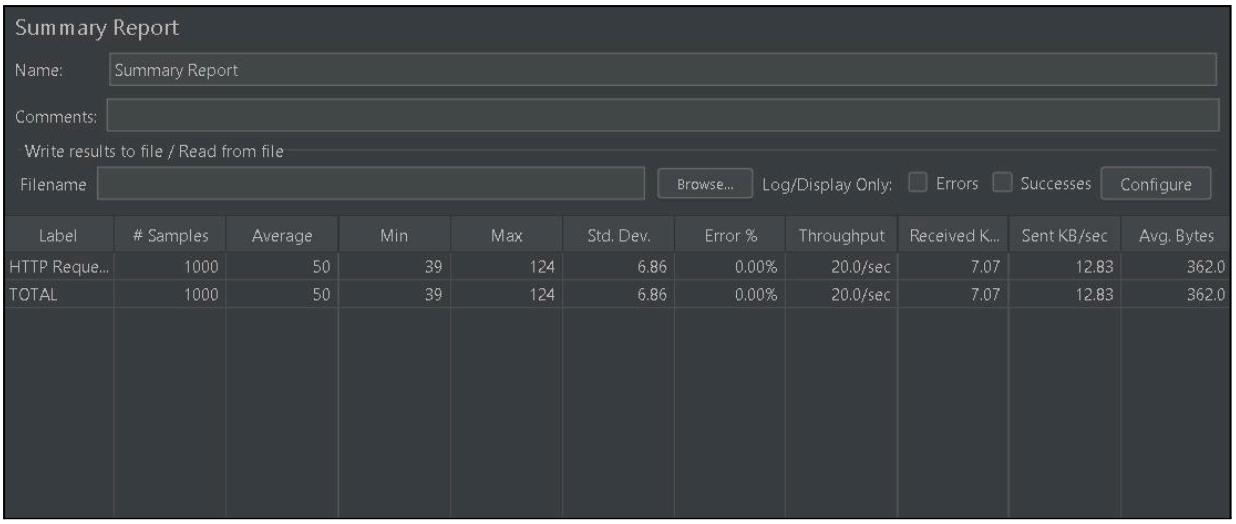
Ahora si bien esto no redujo en mucho el tiempo de respuesta, si bajo el peso y el uso de memoria del aplicativo, haciéndolo mucho más sencillo y puntual, cumpliendo con ser conciso y por supuesto liviano, ahora que ya se hicieron los cambios podemos partir con la medición se la segunda versión.

Medición de la API version 2

Se realizaron nuevamente las pruebas de concurrencia para probar el rendimiento de la api pero ahora para su segunda versión, esto se hará mediante la herramienta Apache JMeter 5.6.3.

Componentes principales

- Usuarios simulados: Se configuraron diferentes grupos de usuarios para realizar solicitudes HTTP.
- Solicitudes HTTP: Se prueba la operación procesar
- Listeners: Se emplearon listeners como el Summary Report y Response Time Graph para registrar y visualizar los resultados de las pruebas.
- Número de muestras: Se realizaron 1000 solicitudes en total como solicita el ejercicio a lo largo de 60 segundos.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec	Avg. Bytes
HTTP Reque...	1000	50	39	124	6.86	0.00%	20.0/sec	7.07	12.83	362.0
TOTAL	1000	50	39	124	6.86	0.00%	20.0/sec	7.07	12.83	362.0

Figura 15. Nueva parametrización para las pruebas

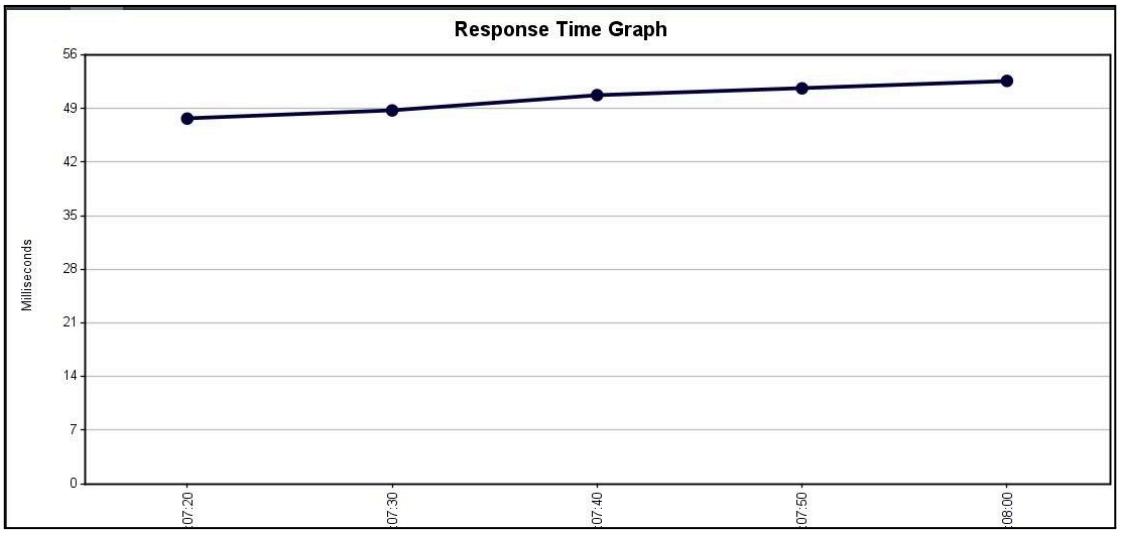


Figura 16. Resultados obtenidos

Análisis de los Resultados

- **Tiempo de respuesta promedio:** Se disminuyó el tiempo de respuesta promedio a 50 ms gracias a las optimizaciones realizadas. Este valor es más que aceptable para la aplicación web en términos de experiencia de usuario.
- **Mejora en el tiempo máximo:** El tiempo máximo de 124 ms es un poco peor que los 114 ms registrados en la primera prueba lo que quiere decir que las optimizaciones introducidas en la segunda versión del proyecto no han aumentado en gran medida los picos de latencia más elevados.
- **Menor variabilidad en tiempos de respuesta:** La desviación estándar de 6.86 ms es muestra de consistencia en los tiempos de respuesta. Manteniéndose de la primera versión.

Conclusiones

Mejora en el rendimiento: La segunda versión del sistema muestra una mejora en términos de consistencia de tiempos de respuesta, especialmente teniendo en cuenta que una la desviación estándar y el tiempo máximo de respuesta no cambiaron mucho y en el caso del tiempo promedio mejoró teniendo en cuenta que no se tenía un delay por procesamiento de pago simulado. Esto quiere decir que las mejoras fueron lo suficientemente adecuadas para que se cumpla con el requerimiento de estar por debajo de los 200 ms.

Tabla comparativa de resultados entre las pruebas

#muestras	promedio	Min response time	Max response time	Desviación estándar	%Error	Rendimiento
1000	79	61	114	6.04	0	16.7/sec
1000	50	39	124	6.86	0	20.0/sec