

## Auditoria Live

Una auditoría que siendo lo más dinámicamente posible permite registrar campos automaticos en las tablas requeridas para tener un control de los usuarios que operan dichos registros, fecha, hora con el fin de mejorar el rendimiento y calidad de la misma

## Auditoría Live o facilidad de consulta en última transacción

La auditoría en RSI tal y como la conocemos nos permite registrar información sobre el usuario que activa una transacción, registrando principalmente información asociada a la Ip del usuario, nombre de usuario, sistema operativo, la conexión a la base de datos por medio de la cual se conecta a la misma, entre otros.

Estos Datos son suficientes para tener control de los datos modificados y poder realizar un seguimiento con el usuario y en caso tal de comportamiento anómalo poder identificar más fácilmente al implicado, sin embargo la auditoría no es algo únicamente asociado a los proyectos RSI, tiene una parte complementaria a nivel de base de datos, en Oracle utilizando la tecnología flashback se permite relacionar por medio de transacciones los datos registrados en la tabla de auditoría con las operaciones registradas en otras tablas bajo la correspondencia de una transacción, esto implicó [el desarrollo previamente documentado](#) donde se realiza tanto la auditoría como el proceso normal del programa en una única transacción para tener este seguimiento.

Sucede que esta tecnología es funcional pero muchas veces es costoso estar relacionándolo por medio de vistas al tiempo en el que esta tecnología está ejecutándose o está activa sobre la tabla en cuestión por esto se propone una auditoría live o complementaria que analice si las tablas a modificar hacen parte de auditoría y se inserte directamente valores automáticos en campos como **usuarioModifica**, **fechaModificacion** etc, que sea a nivel de rendimiento más optimo.

Aclarando las necesidades del proyecto actualmente se ofrecen a nivel de datos la siguiente información mínima con un **par de items**:

1. Una tabla que relaciona las tablas que deben tener este tipo de auditoría live.
2. Un mapeo a nivel de la librería que haga de intermediario para las columnas que se agregarán.

Cuando se realiza un guardado o una actualización podemos tener interceptores de estos eventos y registrar la información necesaria utilizando la tecnología Entity listener en las diferentes Fases como @Preinsert, @Preupdate etc, esto facilita el guardado de la información sin embargo aún queda un inconveniente y es como hacer frente al mapeo dinámico sobre las tablas a auditar, actualmente todas las tablas son auditadas a

nivel de backend, todas las transacciones se auditan pero a nivel de entidades ninguna cuenta con campos adicionales para esta auditoría live, lo más sencillo sería mapear las columnas a medida que se agreguen en base de datos pero esto complica la autonomía del proyecto y lo ideal debería ser que el software esté diseñado para soportar los cambios a nivel de base de datos sin problemas pues para esto el par de items anteriores maneja el comportamiento.

Utilizando técnicas de java reflection podemos lograr acceder a los cambios de las entidades y analizar si es necesario o no auditar una tabla, esto en conjunto con la tecnología entity listener permite cargar de manera automática los valores antes de ser insertados o actualizados.

En el caso dónde no se realice el mapeo, la auditoría complementaria esta preparada para soportarlo y realizar una actualización de manera nativa, esto causa un incremento en los tiempos de respuesta por la manera en la que se actualiza, pero se espera poder utilizar la tecnología asíncrona para este manejo de auditoría y tratar de que el aumento en el rendimiento sea imperceptible.

El código será plasmado en la librería de backend RSI de modo que los cambios en los proyectos no serán significativos, se da la opción al usuario de realizar la auditoría en 3 formas:

- ▶ **#1** - Ignorar la auditoría complementaria y optar por el insert nativo automático, con la penalización de ser menos optimo y aumentar el tiempo de respuesta, solución temporal mientras bloqueos o problemas para modificar el código a tiempo real con el ajuste en base de datos.
- ▶ **#2** - Mapear los campos de la tabla de base de datos en la entidad, automáticamente auditable.
- ▶ **#3 - Sugerida.** Mapear los campos de la tabla de base de datos en la entidad utilizando herencia, automáticamente auditable utilizando la clase proporcionada en la librería.

## Desarrollo o implementación

En la librería queda el código modificado dónde la auditoría se hace necesaria en dos momentos, un **@pre**(persist o update) y un **@Post**(persist o update), pues es un primer caso la unica auditoría posible es mediante mapeo directo en la entidad, ya sea por herencia o de manera manual para cada campo en la base de datos, y en la etapa post tenemos disponibles los Identificadores generados para manejar una auditoría automática sin mapeo o con mapeo.

El código entonces queda de la siguiente manera.

```
1 | @Transactional
2 |     @PreUpdate
3 |     @PrePersist
4 |     @PreRemove
5 |     public void handleLifecycleEvent(Object entity) throws Exception {
6 |         if (!(entity instanceof AuditoriaBackend)) {
7 |             this.auditLive(entity);
8 |         }
9 |     ... audit classic
```

10 | }

Se hace necesaria entonces una exclusión para no revisar los cambios producidos en una entidad del tipo Auditoría backend por obvias razones pues es una auditoría en si misma la cual no requiere auditoría live.

Para el método `auditLive` decidimos primero revisar que solo entidades de **Oracle** sean auditadas, en el proyecto RSI y en particular en el microservicio de integración se cuentan con entidades asociadas al motor de base de datos de **Informix** y por tanto entidades compuestas las cuales no se planean auditar, para diferencia este comportamiento se hace con un chequeo de los campos `@ID` de las entidades **Oracle**, y luego se analiza si la tabla requiere ser auditada.

```
1 private void auditLive(Object entity) throws Exception {
2     if (!this.checkIdAnotation(entity)) return;
3     if (this.isEntityAuditable(entity)) this.updateEntityWithFullAudit(entity)
4 }
```

Para saber si la entidad es auditable utilizamos una clase auxiliar la cual contiene los campos que van a ser añadidos dinámicamente a las tablas elegidas, en esta primera versión se contemplan dos campos, un campo para el id del usuario y otro campo para la fecha de la transacción, dicha clase auxiliar queda de la siguiente manera.

```
1 package co.edu.uis.library.model.auditor;
2
3 import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
4 import lombok.AllArgsConstructor;
5 import lombok.Getter;
6 import lombok.NoArgsConstructor;
7 import lombok.Setter;
8
9 import javax.persistence.Column;
10 import javax.persistence.MappedSuperclass;
11 import javax.persistence.Temporal;
12 import javax.persistence.TemporalType;
13 import java.io.Serializable;
14 import java.util.Date;
15
16 @Setter
17 @Getter
18 @MappedSuperclass
19 @JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
20 @AllArgsConstructor
21 @NoArgsConstructor
```

```

22 public class AuditLiveFields implements Serializable {
23     private static final long serialVersionUID = 3815907516577675869L;
24
25     @Column(name = "ID_USUARIO_AUDITORIA")
26     private Long idUsuarioAuditoria;
27
28     @Column(name = "FECHA_TRANSACCION_AUDITORIA")
29     @Temporal(TemporalType.DATE)
30     private Date fechaTransaccionAuditoria;
31
32
33 }

```

Con base en los campos de esta clase y utilizando reflection se implementa una auditoría completa y una parcial, para la completa se analizan los campos y si contiene el total de los campos a auditar ya sea de manera manual o por herencia, entonces se guarda en una etapa pre persist invocando los setters y llenando dichos campos.

Este primer caso queda de la siguiente manera.

```

1 private void updateEntityWithFullAudit(Object entity) throws Exception {
2     //recorrer todos los campos de auditoría comparar el tipo y mapearlo a un
3     for (var field : FieldUtils.getFieldsListWithAnnotation(AuditLiveFields.class)) {
4         var splited = field.getName().split("[.]");
5         var capitalizedString = splited[splited.length - 1].substring(0, 1).toUpperCase();
6         var method = metodosAuditoriaLiveService.getClass().getDeclaredMethod("set" + capitalizedString, field.getType());
7         Object value = method.invoke(metodosAuditoriaLiveService, entity, field.getName());
8         var fieldName = field.getName();
9         new Statement(entity, "set" + fieldName.substring(0, 1).toUpperCase() + fieldName.substring(1), value);
10    }
11 }

```

El segundo caso es un poco más complejo y requiere un split para completar la etapa, primero se trabaja un **post persist** que genera las instrucciones en formato sql para ser ejecutadas posterior al insert o actualización general, esto porque no se puede ejecutar una instrucción nativa directamente sobre la base de datos sin que la base de datos tenga los datos comiteados, por lo tanto demos separar la logica en 2 partes, la primera parte queda de la siguiente manera.

```

1 @PostPersist @PostUpdate
2 public void auditLiveListener(Object entity) throws NoSuchFieldException, IllegalAccessException {
3     if (!this.checkIdAnotation(entity) || (entity instanceof AuditoriaBackendEntity)) {
4         var tableAnotation = entity.getClass().getAnnotation(Table.class);
5         var nombre = tableAnotation.name().toUpperCase(Locale.ROOT);

```

```

6      var schema = tableAnotation.schema().toUpperCase(Locale.ROOT);
7      if (!this.isEntityAuditable(entity) && this.isEntityAuditableBySql(nombre,
8          var query = auditorService.executeUpdateQuery(nombre, schema, entity)
9          if (!Objects.equals(query, "")) {
10              var idSplited = FieldUtils.getFieldsWithAnnotation(entity.getClass(), Id.class);
11              var idField = idSplited[idSplited.length - 1];
12              var getIdMethod = entity.getClass().getDeclaredMethod("get" + idField.getName());
13              MDC.put(entity.getClass().getName() + "auditable" + getIdMethod.getName(), query);
14          }
15      }
16  }

```

Utilizando un viejo conocido, el **MDC** permite guardar strings, bajo el concepto de clave valor a nivel de thread local y esto es util para tener las instrucciones sql para las actualizaciones en esta bolsa

temporal, por ultimo en la segunda etapa es a nivel de eventos, y utilizando un **evento transaccional** en una **fase final**, podemos invocar un servicio que procese todas las instrucciones de actualización y lo haga de manera automática, dicho evento queda de la siguiente manera.

```

1  @TransactionalEventListener(
2      phase = TransactionPhase.AFTER_COMPLETION
3  )
4  public void handleCompletion(Evento<Object> evento) {
5      Map<String, String> paiValues = MDC.getCopyOfContextMap();
6      Map<String, String> auditableMap = (Map) paiValues.entrySet().stream().filter(
7          p -> ((String) p.getKey()).contains("auditable"));
8      auditableMap.collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
9      auditorService.executeAuditNativeQueryFromList(auditableMap.values());
10 }

```

Ahora los métodos para la generación de las instrucciones sql para la actualización y para el guardado se detallan a continuación, dónde el código en formato sql se obtiene de la siguiente manera:

```

1  public String executeUpdateQuery(String tableName, String schemaName, Object entity) {
2      var columnsToAudit = FieldUtils.getFieldsListWithAnnotation(entity.getClass(), AuditLiveFields.class);
3      var valuesToUpdate = getColumnsAndValues(columnsToAudit, entity);
4
5      var idField = FieldUtils.getFieldsWithAnnotation(entity.getClass(), Id.class).get(0);
6      var getIdMethod = entity.getClass().getDeclaredMethod("get" + idField.getName());
7      Object fieldValue = getIdMethod.invoke(entity);
8
9      var lista = Arrays.asList(schemaName == null ? tableName : schemaName + "." + tableName);
10     var arreglo = new String[lista.size()];

```

```

11     var parametros = lista.toArray(arreglo);
12     var updateQuery = "UPDATE {0} SET {1} WHERE ID = " + fieldValue;
13     return MessageFormat.format(updateQuery, parametros);
14 }
15
16 private String getColumnsAndValues(List<Field> auditableColumns) {
17     var fields = new ArrayList<String>();
18     for (var field : auditableColumns) {
19         var s = field.getAnnotation(Column.class);
20         String nameSql;
21         var splited = field.getName().split("[.]");
22         if (!Objects.equals(s.name(), "")) {
23             nameSql = s.name();
24         } else {
25             nameSql = splited[splited.length - 1].toUpperCase(Locale.ROOT);
26         }
27         String responseSql;
28         if (field.getType().getTypeName().equals("java.util.Date")) {
29             responseSql = nameSql + " = " + getValuesFromColumn(splited[splited.length - 1]);
30         } else {
31             responseSql = nameSql + " = " + "'" + getValuesFromColumn(splited[splited.length - 1]);
32         }
33
34         fields.add(responseSql);
35     }
36     var delimiter = ",";
37     return String.join(delimiter, fields);
38 }

```

Para obtener los valores de la columna utilizamos el método `getValuesFromColumn`, el cual utilizando el api de reflection ejecuta los métodos que son nombrados en la clase [MetodosAuditoriaLiveService](#) y se nombran de la forma `getNombreCampo(...parametros)`.



**Cada campo nuevo en auditoría debe ser añadido en la entidad [AuditLiveFields](#) y en la clase [MetodosAuditoriaLiveService](#) debe estar el método que obtenga dicho valor.**

```

1 private String getValuesFromColumn(String auditableColumn) {
2     try {
3         var capitalizedInput = auditableColumn.substring(0, 1).toUpperCase(Locale.ROOT);
4         var method = this.metodosAuditoriaLiveService.getClass().getDeclaredMethod("get" + capitalizedInput);
5         Object response = method.invoke(this.metodosAuditoriaLiveService);
6         if (response.getClass().getTypeName().equals("java.util.Date")) {

```

```

7         return castDateToString(response);
8     }
9     return (response.toString());
10
11     } catch (NoSuchMethodException e) {
12         throw new RSIValidacionException();
13     } catch (InvocationTargetException | IllegalAccessException e) {
14         throw new RSIDataNotFoundException();
15     }
16 }

```

tanto la interfaz y clase que implementa dichos métodos queda de la siguiente manera.

```

1  @Service
2  public interface IMetodosAuditoriaLiveService {
3
4      Long getIdUsuarioAuditoria();
5
6      Date getFechaTransaccionAuditoria() throws ParseException;
7  }
8
9  @Service
10 public class MetodosAuditoriaLiveServiceImpl implements IMetodosAuditoriaLiveSe
11
12     private HttpServletRequest httpRequest;
13
14     @Override
15     public Long getIdUsuarioAuditoria() {
16         return httpRequest.getHeader(IConstantes.ID_USUARIO_HEADER) == null ?
17             0L : Long.parseLong(httpRequest.getHeader(IConstantes.ID_USUARIO_HEADER));
18     }
19
20     @Override
21     public Date getFechaTransaccionAuditoria() {
22         return new Date();
23     }
24
25     @Autowired
26     public void setHttpRequest(HttpServletRequest httpRequest) {
27         this.httpRequest = httpRequest;
28     }
29 }

```

Ahora bien, la segunda parte que consiste en el guardado nativo de los datos en las entidades modificadas, en una etapa post queda de la siguiente manera.

```
1  @Override
2  @Async
3  @Transactional(propagation = Propagation.REQUIRES_NEW)
4  public void executeAuditNativeQueryFromList(Collection<String> values) {
5      var batchSize = 100;
6      try (Connection connection = hikariDataSource.getConnection();
7          var statement = connection.createStatement()
8      ) {
9          int counter = 0;
10         for (var query: values) {
11             statement.addBatch(query);
12             if ((counter + 1) % batchSize == 0 || (counter + 1) == values.size()) {
13                 statement.executeBatch();
14                 statement.clearBatch();
15             }
16             counter++;
17         }
18     } catch (Exception e) {
19         e.printStackTrace();
20     }
21 }
```

Como punto importante, hacemos un método asíncrono y utilizando la estrategia de inserciones masivas mediante el uso de batch para ejecutar una instrucción SQL estática de manera eficiente y disminuir el tiempo de respuesta final lo máximo posible.



**Conclusión:** De esta manera podemos lograr la auditoría dinámica sin verse afectado tanto el rendimiento, sin embargo como se mencionó anteriormente es mas recomendado que los campos sean mapeados ya sea con herencia o directamente en la entidad para optimizar tiempos.

## Ejemplo de Aplicación

### Escenario # 1 - Ignorando y manejando auditoría totalmente automática.

Tomando como ejemplo la Entidad **Clase situación administrativa**, a nivel de base de datos esta cuenta con los siguientes campos, podemos ver como los últimos dos campos hacen referencia a la auditoría automática,



de acuerdo al desarrollo, con solo añadir estos campos y añadir la tabla asociada en base de datos, en este caso `RECHUMANO.RH_CLASES_SITU_ADMI` La auditoría automática se activa.

	ID	DESCRIPCION	SENAL_GRUPAL	IDENTIFICADOR	ID_USUARIO_AUDITORIA	FECHA_TRANSACCION_AUDITORIA
1	1	LICENCIAS	1	LICEN	<null>	<null>
2	12	COMISIONES	1	COMIS	<null>	<null>
3	24	SITUACIONES CATEDRA	1	SITCAT	<null>	<null>
4	34	INCAPACIDADES	1	INCAP	<null>	<null>
5	35	PERMISOS	1	PERMS	<null>	<null>
6	39	VACACIONES	1	VACA	<null>	<null>
7	38	AÑO SABATICO	1	ANOSA	<null>	<null>
8	40	AUSENCIAS	0	AUS	<null>	<null>
9	7140196	aduitlilas	0	audtli1 diferente	<null>	<null>
10	7140194	aduitlilas	0	audtli1 diferente	<null>	<null>

A continuación un ejemplo de una inserción en dicha clase y luego una actualización de un registro, para ver como se comporta en ambos casos, además del tiempo obtenido en este nuevo escenario.

```

1 | Post: http://localhost:8080/training/api/auditor
2 | body:{
3 |     "id":null,
4 |     "descripcion":"wiki1",
5 |     "senalGrupal":false,
6 |     "identificador":"wiki1"
7 | }
```

Obtenemos lo siguiente, los registros se insertan de manera dinámica y el tiempo de la petición es de alrededor de **450ms**.

	ID	DESCRIPCION	SENAL_GRUPAL	IDENTIFICADOR	ID_USUARIO_AUDITORIA	FECHA_TRANSACCION_AUDITORIA
1	7140209	wiki1	0	wiki1 diferente	888	2023-03-21 15:20:41
2	7140208	wiki1	0	wiki1	888	2023-03-21 15:20:41

## Escenario # 2 mapeo directo en la entidad

“

Es importante que los campos queden en estilo camelCase puesto que de esta manera son nombrados los métodos que resuelven dichos campos.

La entidad queda de la siguiente manera.

```

1 | @Entity
2 | @Setter
3 | @Getter
```

```

4  @NoArgsConstructor
5  @Table(name = "RH_CLASES_SITU_ADMI", schema = "RECHUMANO")
6  public class ClaseSituacionAdministrativa implements Serializable {
7
8      ... other fields
9
10     @Column(name = "ID_USUARIO_AUDITORIA")
11     private Long idUsuarioAuditoria;
12
13     @Column(name = "FECHA_TRANSACCION_AUDITORIA")
14     @Temporal(TemporalType.DATE)
15     private Date fechaTransaccionAuditoria;
16
17
18
19 }

```

A continuación un ejemplo de una inserción en dicha clase y luego una actualización de un registro, para ver como se comporta en ambos casos, además del tiempo obtenido en este nuevo escenario.

```

1  Post: http://localhost:8080/training/api/auditor
2  body:{
3      "id":null,
4      "descripcion":"wiki2",
5      "senalGrupal":false,
6      "identificador":"wiki2"
7  }

```

Obtenemos lo siguiente , los registros se insertan de manera dinámica y el tiempo de la petición es de alrededor de **340ms**.

	ID	DESCRIPCION	SENAL_GRUPAL	IDENTIFICADOR	ID_USUARIO_AUDITORIA	FECHA_TRANSACCION_AUDITORIA
1	7140267	wiki2	0	wiki2 diferente	888	2023-03-21 15:31:28
2	7140266	wiki2	0	wiki2	888	2023-03-21 15:31:28

## Escenario #3 mapeo directo en la entidad

La entidad queda de la siguiente manera.

```

1  @Entity
2  @Setter
3  @Getter

```

```
4  @NoArgsConstructor
5  @Table(name = "RH_CLASES_SITU_ADMI", schema = "RECHUMANO")
6  public class ClaseSituacionAdministrativa extends AuditLiveFields implements Se
7
8
9      ... other fields
10
11
12 }
```

A continuación un ejemplo de una inserción en dicha clase y luego una actualización de un registro, para ver como se comporta en ambos casos, además del tiempo obtenido en este nuevo escenario.

```
1  Post: http://localhost:8080/training/api/auditor
2  body:{
3      "id":null,
4      "descripcion":"wiki3",
5      "senalGrupal":false,
6      "identificador":"wiki3"
7  }
```

Obtenemos lo siguiente , los registros se insertan de manera dinámica y el tiempo de la petición es de alrededor de **350ms**.

	ID	DESCRIPCION	SENAL_GRUPAL	IDENTIFICADOR	ID_USUARIO_AUDITORIA	FECHA_TRANSACCION_AUDITORIA
1	7140289	wiki3	0	wiki3 diferente	888	2023-03-21 15:43:07
2	7140288	wiki3	0	wiki3	888	2023-03-21 15:43:07

## Resultados y comparación de alternativas para la optimización ( cachable vs stateful bean )

# Audit performance

size	10	50	100
<i>Sin Auditoria live</i>	0,6 s	2,6 s	5,1 s
<i>Cachable. Mapeo directo</i>	0,68 s	2,8 s	5,3 s
<i>Cachable. Mapeo herencia</i>	0,7 s	2,70 s	5,20 s
<i>Cachable. automatica</i>	1,5 s	6,45 s	12,2 s
<i>stateful bean. Mapeo directo</i>	0,72s	2,75 s	5,40 s
<i>stateful bean. Mapeo herencia</i>	0,71 s	2,85s	5,50 s
<i>stateful bean. automatica</i>	0,95 s	4,10 s	8,20 s