



Documentación técnica para sistema de depuración CleanSystem

Sady Guzmán, Ethan Pimentel, Daniel Rojas

Índice

1	Proyecto	2
2	Función del proyecto	2
3	Glosario	2
4	Usuario administrador	2
5	Estructura del proyecto.	3
5.1	Contenedores	3
5.2	Compose y volúmenes	3
5.3	Directorios	4
6	Módulos, archivos, rutas Flask, y decoradores	5
6.1	Decoradores	5
6.2	Template dependiendo de sesión	5
6.3	Módulos y sus archivos	6
7	Función depuración.	14
7.1	Acerca del módulo	14
7.2	Reglas de depuración	14
7.3	Código: Llamadas a depuración por pasos.	16
7.4	Código: Función duplicados.	17
7.5	Parámetros: Función duplicados.	21
7.6	Código: Función registra salida.	22
7.7	Parámetros: Función registra salida.	24
7.8	Código: Función salidas saltadas.	25
7.9	Parámetros: Función salidas saltadas.	26
7.10	Código: Función marcaje opuesto.	27
7.11	Parámetros: Función marcaje opuesto.	29
7.12	Código: Fin depuración y retorno de marcaje.	30
8	Función de Validación	31
8.1	Acerca del módulo	31
8.2	Requisitos previos	31
8.3	Código: Conexión con frontend.	32
8.4	Código: Exportación.	33
8.5	Parámetros: Exportación	35
8.6	Código: Validación.	36
8.7	Parámetros: Validación.	40
9	Función de Historial.	41
9.1	Acerca del módulo.	41
9.2	Código: Llamada a función.	41
9.3	Código: Función historial.	42
9.4	Parámetros: Historial.	45

1. Proyecto

El proyecto se lleva a cabo como trabajo semestral para la asignatura 'Ingeniería de Software II' por estudiantes de la carrera Ingeniería en Computación de la Universidad de La Serena, en conjunto con el departamento de asistencias del Hospital San Pablo de Coquimbo.

Repositorio del proyecto es: github.com/Sady-Guzman/ISW2_Proyecto_Asistencias

2. Función del proyecto

El sistema actual de marcaje y registro de horas trabajadas presenta múltiples problemas, especialmente en cuanto a la precisión del registro de turnos y horas trabajadas a lo largo del mes. Estos errores se ven amplificados por el gran volumen de trabajadores, lo que complica aún más el proceso de depuración de los datos de asistencia. Actualmente, esta depuración se realiza manualmente por el personal del área de Gestión de Personas, lo que supone una significativa inversión de tiempo.

Se plantea una propuesta de software para optimizar el proceso de registro de asistencia que permita una integración eficiente con el Sistema de Información de Recursos Humanos (SIRH), reduciendo así errores y mejorando la eficiencia operativa. Detectando el mismo día del registro los marcajes duplicados, falta de marcaje de salida, entre otros.

3. Glosario

Template: Archivo tipo HTML que contiene elementos que se muestran al usuario para que interactúe con el sistema.

Output: Salida, la información que el sistema produce: Archivo de marcajes depurado y archivo con historial de cambios.

SIRH: Sistema de Información de Recursos Humanos del MINSAL

Depuración: En este sistema la depuración se entiende como la propuesta de cambios que se hace de forma automática después de que el usuario importe un archivo de marcaje aún sin corregir. Esta propuesta se visualiza y necesita la aprobación del usuario antes de llevarse a cabo. Más detalles en la sección de la función depuración.

4. Usuario administrador

Por defecto el sistema incluye un usuario administrador que tiene funciones para crear, editar, visualizar, y eliminar cuentas de usuarios de depuración. Las credenciales por defecto para este usuario son (Usuario: admin) (Clave: hospital)

5. Estructura del proyecto.

Se usa Python con el framework de Flask para backend, Postgres como base de datos, Bootstrap como framework frontend en conjunto con Jinja2 para generar HTML de forma dinamica. Todo esto es encapsulado en contenedores usando Docker, y Compose para orquestar y gestionar los contenedores y volúmenes.

5.1. Contenedores

Se usan 2 contenedores. El primer contenedor se usa para el servicio de base de datos usando PostgreSQL. El segundo contenedor se usa para la aplicación web usando Python.

En compose se define que se debe ejecutar completamente el contenedor de base de datos para poder proceder a la ejecución del otro contenedor.

5.2. Compose y volúmenes

flask_app

Este servicio representa la aplicación backend construida con Flask. se mapea el puerto 5000 del contenedor al puerto 5000 del host.

```
1 volumes:
2   - ./horario_mensual:/app/horario_mensual
```

Mapea el directorio horario_mensual del host a /app/horario_mensual en el contenedor, permitiendo persistir datos relacionados con los horarios mensuales que se suben cada mes por el equipo de asistencia. Asegurar asignar permisos adecuados al directorio en el host (chmod 777 ./horario_mensual) para evitar problemas de escritura.

db

usa imagen oficial de PostgreSQL. Las variables de entorno son

```
1 environment:
2   POSTGRES_USER: postgres
3   POSTGRES_PASSWORD: domino
4   POSTGRES_DB: postgres
```

Se mapea el puerto 5432 del host al puerto 5432 del contenedor.

```
1 volumes:
2   - postgres_data:/var/lib/postgresql/data
3   - ./init.sql:/docker-entrypoint-initdb.d/init.sql
```

postgres_data: Persisten los datos de las cuentas creadas.

init.sql: Mapea un archivo SQL (init.sql) desde el host al directorio de inicialización de PostgreSQL. Se usa para configurar la base de datos y crear el usuario de administrador la primera vez que se ejecuta.

5.3. Directorios

/static

Directorio que contiene imágenes, logos, iconos, y archivo de estilos css.

/templates

Directorio que contiene los archivos HTML de cada página.

/temp

Directorio donde se guardan los archivos temporales durante el funcionamiento de la depuración y la exportación de los marcajes.

/horario_mensual

Directorio donde se guarda el archivo de horarios de trabajadores que sube administrador con frecuencia mensual. Es permanente ya que se usa un volumen de docker para esta ruta.

6. Módulos, archivos, rutas Flask, y decoradores

6.1. Decoradores

Solo existen dos decoradores. `@user_login_required` y `@admin_login_required`. Se aseguran de que las rutas a las que se aplican sean sólo accesibles por sesiones que tengan su respectivo tipo de usuario validado. De esta manera las funciones de administrador no pueden ser usadas por usuarios de menor nivel.

6.2. Template dependiendo de sesión

Dependiendo del tipo de sesión con el que el usuario se valida (o si aun no se valida ningún tipo de sesión) los elementos que se cargan de `layout.html` cambian.

La principal diferencia está en el header que se carga en todo momento. Si aún no se valida ningún tipo de sesión en el header solo se muestra el logo del sistema y las opciones para acceder a las rutas de inicio de sesión para usuarios y para administrador.

Una vez iniciada la sesión de administrador se muestran las opciones para acceder a las rutas de 'Registro de usuario', 'Visualizar usuarios existentes', 'Administrar usuario', y 'Cerrar sesión'. Para la sesión de usuario se muestran las opciones de 'Cargar archivo' y 'Cerrar sesión'.

Luego en cada template HTML que se carga en cada ruta se hereda la estructura HTML de `layout.html` usando Jinja2.

6.3. Módulos y sus archivos

app.py

Define la configuración principal y el punto de entrada de una aplicación web construida con el framework Flask. Su estructura permite modularidad y escalabilidad al integrar varias funcionalidades mediante blueprints, cada blueprint correspondiendo a un módulo de la aplicación.

```
1 app.config['UPLOAD_FOLDER'] = os.path.join(os.getcwd(), 'temp')
```

Define UPLOAD_FOLDER, que apunta al directorio temp dentro del contenedor, donde se almacenan temporalmente archivos cargados por los usuarios.

def after_request(response): Garantiza que las respuestas no sean cacheadas, forzando al navegador a solicitar el contenido al servidor.

Ruta raíz: Solo muestra la página principal (index.html), no incluye lógica.

Blueprints: Se usan blueprints de Flask para modularizar y organizar el código.

- session_routes: Gestiona el login de usuario y administrador.
- manejo_cuentas: Incluye lógica de admin para crear, eliminar, y administrar cuentas de usuario.
- carga_archivo: Permite a los usuarios cargar archivos al servidor.
- visualizacion: Maneja la visualización y filtrado de marcajes depurados.
- subir_reglas: Funcionalidad para que los administradores carguen reglas de horario.

Carga de Archivo.

Este módulo se compone de la ruta `/carga`, el archivo `carga_archivo.py` y el template HTML `carga.html`.

El archivo `carga_archivo.py` implementa la lógica para recibir, validar, almacenar y procesar archivos cargados por el usuario.

la ruta `/cargar` maneja solicitudes `/GET` y `/POST` y está protegida por el decorador `@user_login_required`, asegurando que solo usuarios autenticados (que tengan una sesión iniciada) puedan acceder a la ruta. el método `/GET` muestra el template del archivo `carga.html`. El método `/POST` se accede cuando el usuario sube un archivo mediante el formulario del template en `carga.html`.

La lógica luego de acceder mediante el método `/POST` es:

Validación inicial: comprueba si se ha enviado un archivo y si tiene un nombre válido. Verifica que el archivo que sube el usuario tenga la extensión `.log`. Si no cumple, muestra un mensaje de error usando `'flash'` y recarga la página de `carga_archivo`.

Manejo del nombre del archivo: El nombre original del archivo se guarda en `'NOMBRE_ORIGINAL_ARCHIVO.txt'` para referencia futura. El archivo cargado se renombra a `'marcajes_original.csv'` para un manejo uniforme a lo largo de toda la lógica de depuración.

Almacenamiento del archivo: El archivo renombrado se guarda en el directorio (`/app/temp/`). Si ocurre algún error durante el guardado, se notifica al usuario y se renderiza una plantilla de error `'apology.html'`.

Procesamiento del archivo:: Se llama a la función `depurar_archivo()` para realizar los pasos iniciales de depuración. La función de depuración no necesita ningún parámetro ya que usa la ruta predefinida `TEMP` para recuperar la información en el archivo que sube el usuario.

Redirección: Si el archivo se procesa correctamente, el usuario es redirigido a `/visualizacion` para ver los datos procesados. Si ocurre un error durante la depuración, se muestra un mensaje de error.

helpers.py.

Contiene código para decoradores de `login_required`, `user_login_required`, y `admin_login_required`. Cambien contiene función `'format_rut'` para mostrarlos de manera correcta en `/view_accounts`.

Inicio de sesión

Este módulo gestiona la autenticación de usuarios y administradores. Se compone de las rutas ***/adlogin:***, ***/login:***, y ***/logout:***. Se cargan los templates ***/adlogin.html:*** y ***/login.html:***. Cuenta con métodos `/GET` y `/POST`. `/GET` muestra el respectivo template, `/POST` maneja la lógica de validación de inicio de sesión.

/login y /adlogin: Valida los campos de entrada de nombre de usuario y contraseña usando `'check_password_hash'`. Al iniciar sesión se guarda el nombre de usuario de forma temporal en la ruta `'/app/temp/username.txt'` para su uso en la función de historial, de esta forma queda registro de que persona fue la que usó el sistema para depurar un archivo de marcajes.

/logout: limpia la sesión previa con `'session.clear()'` y redirige a la página principal.

Manejo de cuentas por administrador.

Proporciona funcionalidades esenciales para la administración de cuentas de usuario en el sistema, incluyendo la creación, edición, visualización y eliminación de usuarios. como también cargar al sistema los horarios de cada mes. Las funcionalidades que se implementan en este módulo solo son accesibles por una sesión de administrador.

Este módulo se compone de las rutas `/register`, `/change_password`, `/view_accounts`, `/delete_account`, templates HTML `/edit_account.html`, `/register`, `/subir_reglas`, `/view_accounts`, y el archivo `/manejo_cuentas.py`. Todas las rutas están protegidas por el decorador `@admin_login_required`, que garantiza que solo los administradores puedan acceder a ellas.

las rutas y sus funcionalidades son:

`/register`: Registra un nuevo usuario en el sistema. Se usa la información (RUT, nombre de usuario, y contraseña).

Para permitir la creación del nuevo usuario se valida que se haya ingresado un RUT con un formato correcto, Que el nombre de usuario no esté en uso, que el RUT no esté asociado a otro usuario, que los campos de contraseña y confirmación de contraseña coincidan. Si todo lo anterior es válido se aplica una función de hash a la contraseña y se guarda el nuevo usuario en la base de datos. En caso de error en alguna de los pasos de validación se muestra un error con la función flash y se recarga el template de registro para que el administrador vuelva a intentarlo.

`/change_password`: Permite al administrador cambiar la contraseña de un usuario. El administrador debe ingresar el nombre de usuario y la nueva contraseña (junto con su confirmación). Valida que el usuario exista en la base de datos y que la nueva contraseña coincida con la confirmación. Esta ruta carga el archivo `edit_account.html`.

`/view_accounts`: Muestra una lista de todos los usuarios registrados en el sistema. Cada cuenta incluye el RUT formateado en `helpers.py`, el nombre de usuario y el ID del usuario. Esta ruta carga el archivo `view_accounts.html`.

`/delete_accounts`: Permite al administrador eliminar una cuenta de usuario especificada por su ID. Esta ruta carga el archivo `view_accounts.html`.

Carga de reglas de horarios.

Este módulo gestiona la funcionalidad de subir, procesar y limpiar archivos de horarios mensuales, diseñada específicamente para ser utilizada por administradores en el sistema. Espera un formato **.csv**

Método `/GET` muestra el template `subir_reglas.html`.

método `/POST` maneja lógica de carga y depuración de horarios. Se especifica el proceso a continuación:

Validación del archivo: Verifica si se ha subido un archivo con el campo `file` en `request.files` y que no esté vacío, Si falla se dirige nuevamente a la ruta de `/subir_reglas`.

Guardado del archivo:: Se usa el directorio de almacenamiento permanente `'/app/horario_mensual'`. Establece el nombre del archivo como `'horarios_creados.csv'` y elimina el archivo anteriormente cargado por el administrador en caso de existir uno.

Gestión del archivo:: Después de guardar el archivo en el directorio especificado se llama a la función `'depurarReglas()'` para limpiar y procesar el contenido, dejando las columnas en el estado esperado para la depuración de marcajes.

depurarReglas(): Limpia y filtra datos en el archivo de reglas subido por administrador.

Pasos de Proceso

- Carga el archivo a un `df` de Pandas, se define `';`' como separador. También se renombran las columnas para facilitar la lógica en la función.
- Se filtran los datos excluyendo las filas donde la columna `'Codigo'` empieza con `"Fecha :"`.
- Aplica la función `dividir_hora_minuto` para extraer horas y minutos de las columnas `entrada` y `salida`.
- Maneja valores `NaN` y convierte las columnas resultantes a tipo entero.
- Utiliza valores predeterminados para filtrar registros correspondientes al año y mes deseados.
- Sobrescribe el archivo original con los datos filtrados y limpios.

dividir_hora_minuto(): Divide una cadena con formato `'hh : mm'` en dos columnas: hora y minuto. Valida si la entrada es una cadena válida que contiene `':'`. retorna dos valores separados.

Visualización.

Este módulo gestiona la funcionalidad de visualización y filtrado de datos procesados en el sistema. Fue diseñado para proporcionar a los usuarios autenticados (@user_login_required) una interfaz que permita visualizar, filtrar, y descargar datos depurados previamente, incluyendo también un historial de cambios hechos por el sistema.

Se compone de las rutas: */visualizacion*, *apply_filters*, *download_csv*, *download_historial*. Se carga el template *visualizacion.html*.

La lógica de las funciones son:

/visualizacion:. Usa la función 'def visualizar()'. Primero de define *file_path*. como 'app/temp/datos_procesados.csv'. Luego comprueba que el archivo en *fil_path* exista, en caso de existir lo carga a un dataframe de Pandas y filtra las columnas que son relevantes visualizar. Se guardan en las variables *table_columns*, *table_data*, *distinct_days* las columnas y tuplas del dataframe, Estas dos variables se entregan a la función *render_template* junto el template *visualizacion.html*.

Se guarda en la variable *distinct_days* los días existentes en el dataframe para luego usar su contenido en uno de los campos de filtrado.

/apply_filters:. La función *apply_filters* permite a los usuarios aplicar diversos filtros sobre un conjunto de datos cargado desde un archivo CSV, con el objetivo de personalizar y refinar la información visualizada en la interfaz. Los filtros incluyen criterios como rango de tiempo, RUT, tipo de marcaje, estado de los registros (Correcto, Duplicado, Saltado, Invertido), y días específicos.

Para saber cuales son los parámetros que el usuario quiere aplicar como filtros se recupera la información del formulario de *visualizacion.html* usando:

```
1 from flask import request
2
3 rut_filter = request.form.get('rut_filter')
4 from_hour = request.form.get('from_hour')
5 to_hour = request.form.get('to_hour')
6 tipo_marcaje = request.form.get('tipo_marcaje')
7 condicion = request.form.get('condicion')
8 codigo_filter = request.form.get('codigo_filter')
9 day_filter = request.form.get('day_filter')
```

Luego se usa un dataframe de Pandas para aplicar los filtros sobre las columnas.

/download_csv: Descarga el resultado de la depuración y validación de los marcajes a un archivo que se guarda en la carpeta de descarga del sistema operativo. Usa la información del dataframe de Pandas para generar un archivo .csv con todas las columnas necesarias, descartando otras creadas para facilitar la depuración, visualización, y filtrado como 'Hora'. Se usa extensión '.log' para el resultado, ya que es la extensión que se usa con anterioridad con la herramienta SIRH.

/download_historical: Descarga un historial de cambios hechos sobre el archivo de marcajes original. Incluye el nombre de usuario de la persona que usa el sistema para depurar los marcajes, la fecha en la que se hace la depuración, RUT de persona a la que se le efectúa un cambio en su marcaje, descripción del error encontrado y la solución que se usa para depurar.

Estructura y funcionalidad visualizacion.html:

Este archivo se compone de cinco partes: i) Barra de navegación superior, ii) tabla de visualización, iii) panel de filtros, iv) funciones de Javascript, v) estilos en CSS.

i) Barra de navegación: Está implementada directamente en este template HTML en vez de extenderla desde layout.html porque este panel necesita usar una porción más grande la pantalla para visualizar la información en la tabla más cómodamente.

ii) Tabla de visualización: Se construye en base a un dataframe de Pandas, del dataframe que se carga en *visualizacion.py* se obtienen dos variables, *table_columns* y *table_data*, estas corresponden a los headers y las tuplas con la información. El contenido de la tabla se construye dinámicamente gracias a la funcionalidad de ciclos de Jinja2, los ciclos para construir HTML se marcan con '*% ... %*.' También se agregan checkboxes al inicio de cada tupla.

```
1 <thead>
2   <tr>
3       {% for column in table_columns %}
4           <th>{{ column }}</th>
5       {% endfor %}
6   </tr>
7 </thead>
8 <tbody>
9     {% for idx, row in enumerate(table_data) %}
10        <tr>
11            <td class="checkbox-cell">
12                <input type="checkbox" class="row-checkbox" name="selected_rows"
13                    value='{{ row | tojson }}'>
14            </td>
15
16            {% for cell in row %}
17                <td>{{ cell }}</td>
18            {% endfor %}
19        </tr>
20    {% endfor %}
</tbody>
```

Las casillas tipo checkbox se agregan al inicio de cada tuplas ya que son usadas para dar la funcionalidad de validación que se implementa en *validacion.py*. Los códigos de [1: Entrada] [3: Salida] son basados en los códigos que ya usan los relojes de marcajes que están instalados en el hospital.

Además de tener un botón (descargar archivo) en la esquina inferior derecha de la página que llama la funcionalidad de descarga, esta lee el estado de la tabla en el momento que se hace click en el botón y hace la descarga pasando antes por el módulo de validación.

iii) Panel de filtros: En este panel existen tipos 6 filtros. Uno de ellos siendo dividido en dos, Teniendo en total 7 filtros. Estos filtros son: RUT, rango de tiempo (desde y hasta), tipo de marcaje (entrada/salida), día, código de horario, estado del marcaje (sin problemas, duplicado, saltado, invertido, con problemas (combina todos los anteriores)). Se toma la información del formulario y se retorna a *visualizacion.py* para ser procesados en la función *apply_filters()*, donde se aplican los filtros al dataframe para luego mostrar en la tabla el resultado.

El único filtro que tiene un script de Javascript dedicado es el filtrado por día ya que usa una variable llamada 'distinct_days' para saber qué días están presentes en el archivo de marcajes que se está depurando en la sesión, además de esconder la lista de días por defecto para tener un panel más organizado.

iv) Funciones Javascript: La primera función se usa para el filtro de día en el panel de filtros, muestra o esconde la lista de días existentes en el dataframe (archivo de marcajes) usando la variable 'distinct_days'.

La segunda función marca todas las casillas tipo checkbox de las tuplas de la tabla. Esto resulta en que el archivo no se depure y mantenga su estado original ya que el marcar la casilla a la izquierda de un marcaje hace que el módulo de validación ignore esa depuración antes de descargar los archivos resultantes.

La tercera y última funcionalidad de Javascript se usa para procesar cuales casillas el usuario depurador marcó antes de usar el botón 'descargar archivo' para dejar saber al módulo de validación cuales marcajes tienen que ser ignorados y por lo tanto descargarse en su estado original. Luego de eso descarga los archivos 'marcajes_depurados%FECHA ACTUAL%' e 'historial_cambios%FECHA ACTUAL%', La funcionalidad de obtener la fecha actual también está implementada en este script.

v) estilos: Se especifican dentro de este template HTML ya que solo se usan para esta página, aplican estilos para la tabla de visualización y su barra de scroll.

7. Función depuración.

7.1. Acerca del módulo

Este módulo define la capacidad del sistema para detectar errores de marcaje y proponer correcciones. El módulo consta principalmente de tres partes: la lectura del archivo cargado, la depuración basada en reglas definidas por el usuario, y finalmente, la creación de un archivo temporal para su posterior visualización y validación.

7.2. Reglas de depuración

La depuración funciona abordando los tres errores más comunes en el marcaje, y para cada uno se propone una corrección específica. A continuación, se define cada tipo de error y su respectiva corrección:

1. Duplicados

Esta categoría incluye todas las entradas de marcaje que aparecen más de una vez en el archivo, independientemente de su tipo. La corrección para este error consiste en crear una entrada correspondiente para cada salida y una salida para cada entrada repetida.

2. Omisiones

Un error de tipo *omisión* ocurre cuando una entrada de marcaje tiene una entrada pero no una salida. Estos errores se identifican por la presencia de un registro generado automáticamente por el sistema de marcaje con la hora 00:00. La corrección para este tipo de error consiste en reemplazar la hora 00:00 con el horario de salida establecido en el contrato de la persona.

3. Marcaje opuesto

Este error ocurre cuando una persona marca una entrada en lugar de una salida. La corrección implica analizar los márgenes de tiempo alrededor de los horarios esperados de salida para verificar si hay un comportamiento anómalo.

Código: Lectura de archivo y reglas.

Ya aclaradas las bases de la depuración se procederá con la explicación del código, y el funcionamiento del proceso.

```
1 import pandas as pd
2 def depurar_archivo(file_path):
3     try:
4         marcaje = pd.read_csv(file_path, header= None, sep=',',
5                               names=["Codigo", "a", "entrada/salida", "rut","b", "hora",
6                                     "minuto", "mes", "día", "año", "c", "d", "e", "f", "g",
7                                     "h", "i", "j", "k"],
8                               dtype={"Codigo": str,"a": str,"entrada/salida": str,"b":
9                                     str,"c": str,"d": str,"e": str,"f": str,"g": str,"h":
10                                    str,"i": str,"j": str,"k": str})
11        # Juntar hora y minuto en una sola columna
12        marcaje['Hora'] = marcaje['hora'].astype(str).str.zfill(2) + ':' +
13            marcaje['minuto'].astype(str).str.zfill(2)
14        try:
15            ruta_reglas = "/app/horario_mensual/horarios_creados.csv"
16            reglas = pd.read_csv(ruta_reglas, sep=',').dropna(axis='columns',
17                           how='all')
18        except Exception as e:
19            print(f"Error al cargar archivo de reglas {e}")
20            return None
21    except Exception as e:
22        print(f"Error DEPURACION - Crea DF con contenido de archivo subido:
23              {e}")
24        return None
```

Para empezar es sumamente importante aclarar que la función de depuración es una función llamada desde el módulo de carga, por ende recibe la ruta directa del archivo proporcionado por el usuario, archivo el cual es cargado en un dataframe llamado marcaje mediante pandas, en el que se da nombre a todas sus columnas, sin embargo las columnas realmente importantes son:

1. Código: El código hace referencia al código de horario, el cual es un identificador único para el tipo de turno que esta haciendo una persona.
2. entrada/salida: La columna de “entrada/salida” corresponde a una columna de tipo numérico que contiene números 01 y 03. Estos hacen referencia a las marcas de entradas y salidas de los trabajadores respectivamente.
3. rut: Es el rut de la persona y su identificador único.
4. hora: La hora en la que se marco la acción.
5. minuto: Minuto en la que se marco la acción.
6. día: Día en el que se marco la acción.
7. mes: Mes en el que se marco la acción.
8. año: año en el que se marco la acción.
9. Hora: Columna “Hora” con H mayúscula, es una columna creada en la que se guarda como un solo string la hora completa, es decir horas y minutos separados con “:”, en la que se marca la acción.

7.3. Código: Llamadas a depuración por pasos.

Se llama de manera secuencial a las funciones de depuración y se actualiza de manera reiterativa con las correcciones pertinentes al dataframe marcaje.

```
1  ''' DUPLICADOS '''
2  try:
3      marcaje = duplicados(marcaje)
4  except Exception as e:
5      print(f"Error DEPURACION - proceso DUPLICADOS: {e}")
6      return None
7
8  '''REVISION DE SALIDAS'''
9  try:
10     marcaje['cierre'] = "No tiene cierre"
11     marcaje = marcaje.sort_values(by=['rut', 'día',
12                                     'Hora']).reset_index(drop=True)
13
14     for indice in range(len(marcaje.index)):
15         if marcaje.at[indice, 'entrada/salida'] == 1: # Solo evalúa entradas
16             registraSalida(marcaje, indice)
17
18     marcaje = marcaje.sort_values(by=['día', 'Hora',
19                                     'rut']).reset_index(drop=True)
20 except Exception as e:
21     print(f"Error DEPURACION - proceso TIENE SALIDA: {e}")
22     return None
23
24 ''' FALTA SALIDA (Omisiones)'''
25 try:
26     marcaje = faltaSalida(marcaje, reglas)
27 except Exception as e:
28     print(f"Error DEPURACION - proceso FALTA SALIDA: {e}")
29     return None
30
31 ''' MARCA OPUESTO '''
32 try:
33     marcaje = marcaOpuesto(marcaje, reglas)
34 except Exception as e:
35     print(f"Error DEPURACION - proceso MARCA OPUESTO: {e}")
36     return None
```

7.4. Código: Función duplicados.

```
1 ''' DUPLICADOS '''
2 try:
3     marcaje = duplicados(marcaje)
4 except Exception as e:
5     print(f"Error DEPURACION - proceso DUPLICADOS: {e}")
6     return None

1 def duplicados(marcaje):
2     entrada = marcaje.copy() # Crear una copia para evitar modificar el original
3
4     # Ordenar por rut, día y hora
5     entrada = entrada.sort_values(by=['rut', 'día',
6     'Hora']).reset_index(drop=True)
7
8     # Columna de errores
9     entrada['Error'] = '0k'
10
11    # Verificar entradas duplicadas
12    for rut, group in entrada.groupby('rut'):
13
14        # Variable para almacenar la última acción
15        ultima_accion = None
16        for i in range(len(group) - 1):
17            fila_actual = group.iloc[i]
18            fila_siguiente = group.iloc[i + 1]
19
20            # Verificar si hay entradas duplicadas sin salida entre ellas y son
21            # el mismo día
22            if (fila_actual['entrada/salida'] == 1 and
23                fila_siguiente['entrada/salida'] == 1 and ultima_accion != 3 and
24                fila_actual['día'] == fila_siguiente['día']):
25                # Marcar como entrada duplicada
26                entrada.loc[group.index[i + 1], 'Error'] = 'Entrada duplicada'
27
28            # Verificar si hay salidas duplicadas sin entrada entre ellas y son
29            # el mismo día
30            elif (fila_actual['entrada/salida'] == 3 and
31                fila_siguiente['entrada/salida'] == 3 and ultima_accion != 1 and
32                fila_actual['día'] == fila_siguiente['día']):
33                # Marcar como salida duplicada
34                entrada.loc[group.index[i + 1], 'Error'] = 'Salida duplicada'
35
36            ultima_accion = fila_actual['entrada/salida']
37
38    entrada = entrada.sort_values(by=['día', 'Hora',
39    'rut']).reset_index(drop=True)
```

```
1 nuevoDf = []
2
3 for i, row in entrada.iterrows():
4     # Si no hay error, incluir la fila tal cual
5     if row['Error'] == 'Ok':
6         nuevoDf.append(row)
7
8     elif row['Error'] == 'Entrada duplicada':
9         # Agregar la fila actual
10        nuevoDf.append(row)
11
12        # Crear una fila de "salida creada por duplicado" con los mismos datos
13        salida_row = row.copy()
14        salida_row['entrada/salida'] = 3 # Cambiar a salida
15        salida_row['Error'] = 'Salida creada por duplicado'
16        nuevoDf.append(salida_row) # Agregar la nueva fila
17
18    elif row['Error'] == 'Salida duplicada':
19        # Crear una fila de "entrada creada por duplicado" con los mismos
20        # datos
21        entrada_row = row.copy()
22        entrada_row['entrada/salida'] = 1 # Cambiar a entrada
23        entrada_row['Error'] = 'Entrada creada por duplicado'
24        nuevoDf.append(entrada_row) # Agregar la nueva fila
25
26        # Agregar la fila actual
27        nuevoDf.append(row)
28
29    # Crear un nuevo DataFrame a partir de nuevoDf, que ahora incluye todas las
30    # filas
31    entrada = pd.DataFrame(nuevoDf)
32
33    return entrada
```

Función para detectar y corregir entradas duplicadas en un conjunto de registros de marcaje.

La función identifica registros de entrada y salida duplicados en los datos de 'marcaje', y realiza correcciones automáticas al agregar la entrada o salida correspondiente en caso de duplicación. Los registros corregidos incluyen una columna 'Error' que especifica el tipo de duplicado detectado y la corrección realizada.

```
1 # Ordenar por rut día y hora
2 entrada = entrada.sort_values(by=['rut', 'día',
3     'Hora']).reset_index(drop=True)
4
5 # Columna de errores
6 entrada['Error'] = 'Ok'
```

En este segmento se ordena el DataFrame entrada por 'rut', 'día' y 'Hora' para analizar secuencialmente los registros, además de esto se crea una nueva columna que luego será incorporada al dataframe original llamada Error, en la cual se comentarán los errores solucionados.

```

1      # Verificar entradas duplicadas
2      for rut, group in entrada.groupby('rut'):
3
4          # Variable para almacenar la última acción
5          ultima_accion = None
6          for i in range(len(group) - 1):
7              fila_actual = group.iloc[i]
8              fila_siguiete = group.iloc[i + 1]
9
10             # Verificar si hay entradas duplicadas sin salida entre ellas y son
               el mismo día
11             if (fila_actual['entrada/salida'] == "01" and
                  fila_siguiete['entrada/salida'] == "01" and ultima_accion != "03"
                  and
12                 fila_actual['día'] == fila_siguiete['día']):
13                 # Marcar como entrada duplicada
14                 entrada.loc[group.index[i + 1], 'Error'] = 'Entrada duplicada'
15
16             # Verificar si hay salidas duplicadas sin entrada entre ellas y son
               el mismo día
17             elif (fila_actual['entrada/salida'] == 3 and
                    fila_siguiete['entrada/salida'] == 3 and ultima_accion != "01" and
                    fila_actual['día'] == fila_siguiete['día']):
18                 # Marcar como salida duplicada
19                 entrada.loc[group.index[i + 1], 'Error'] = 'Salida duplicada'
20
21             ultima_accion = fila_actual['entrada/salida']
22
23         entrada = entrada.sort_values(by=['día', 'Hora',
24                                           'rut']).reset_index(drop=True)

```

En esta parte se agrupa los registros por 'rut' para procesar las entradas y salidas de cada usuario individualmente, para luego iterar sobre cada grupo de registros para detectar duplicados de entrada y salida, utilizando las siguientes reglas:

1. Si una entrada es seguida de otra entrada sin una salida intermedia, marca la segunda entrada como 'Entrada duplicada'
2. Si una salida es seguida de otra salida sin una entrada intermedia, marca la segunda salida como 'Salida duplicada'.

Una vez completado este proceso se procede a reordenar nuevamente el dataframe, pero esta vez por 'día' y 'Hora'

```
1 nuevoDf = []
2
3 for i, row in entrada.iterrows():
4     # Si no hay error, incluir la fila tal cual
5     if row['Error'] == 'Ok':
6         nuevoDf.append(row)
7
8     elif row['Error'] == 'Entrada duplicada':
9         # Agregar la fila actual
10        nuevoDf.append(row)
11
12        # Crear una fila de "salida creada por duplicado" con los mismos datos
13        salida_row = row.copy()
14        salida_row['entrada/salida'] = 3 # Cambiar a salida
15        salida_row['Error'] = 'Salida creada por duplicado'
16        nuevoDf.append(salida_row) # Agregar la nueva fila
17
18        elif row['Error'] == 'Salida duplicada':
19            # Crear una fila de "entrada creada por duplicado" con los mismos
20            datos
21            entrada_row = row.copy()
22            entrada_row['entrada/salida'] = 1 # Cambiar a entrada
23            entrada_row['Error'] = 'Entrada creada por duplicado'
24            nuevoDf.append(entrada_row) # Agregar la nueva fila
25
26            # Agregar la fila actual
27            nuevoDf.append(row)
28
29            # Crear un nuevo DataFrame a partir de nuevoDf, que ahora incluye todas las
30            filas
31            entrada = pd.DataFrame(nuevoDf)
32
33            return entrada
```

En esta última parte, se toma el DataFrame 'entrada' y se aplican las correcciones necesarias en un nuevo DataFrame, 'nuevoDf'.

1. Iteración por filas: La función itera sobre cada fila de 'entrada' usando `**entrada.iterrows()**, donde cada 'row' representa un registro de marcaje y 'i' es su índice.`
2. Detección de errores: Durante la iteración, se verifica si la fila actual contiene algún error marcado en la columna 'Error', con posibles valores: 'Ok' (caso en el que no se hace nada), 'Entrada duplicada' o 'Salida duplicada'.
3. Corrección de duplicados:
 - a) Para una 'Entrada duplicada': Si una fila tiene el error 'Entrada duplicada', se crea una nueva fila de salida con la misma hora, que sirve como una "Salida creada por duplicado". Esta nueva fila se agrega a 'nuevoDf' inmediatamente después de la fila de entrada duplicada.
 - b) Para una 'Salida duplicada': Si una fila tiene el error 'Salida duplicada', se crea una nueva fila de entrada con la misma hora, que sirve como una "Entrada creada por duplicado". Esta fila se agrega a 'nuevoDf' inmediatamente antes de la fila de salida duplicada.

4. Reconstrucción del DataFrame: Una vez finalizado el proceso de iteración, 'nuevoDf' contiene tanto los registros originales como las correcciones. Se convierte 'nuevoDf' nuevamente en 'entrada', aplicando así las correcciones al DataFrame original.
5. Resultado final: La función devuelve el DataFrame corregido 'entrada', que actualiza así los registros de 'marcaje'.

7.5. Parámetros: Función duplicados.

Nombre	Tipo/Descripción	Uso
marcaje	DataFrame	Input principal que contiene información de ruts, días, horas y acciones de entrada/salida.
entrada	DataFrame (copia de marcaje)	Almacenamiento temporal de los datos para evitar modificar el original.
Error	Columna de DataFrame	Marca filas con errores identificados, como Entrada duplicada o Salida duplicada.
rut	str	Identificador único del grupo al que pertenece una fila.
group	DataFrame	Subconjunto de entrada agrupado por rut .
ultima_accion	str o None	Registro de la última acción (01 o 03) para comparar con la acción actual.
i	int	Índice utilizado para recorrer filas de un group .
fila_actual	Series	Fila actual del group durante la iteración.
fila_siguiente	Series	Fila siguiente del group para comparar con fila_actual .
nuevoDf	list	Lista para almacenar las filas ajustadas con datos corregidos o errores marcados.
salida_row	Series	Fila generada para representar una "Salida creada por duplicado".
entrada_row	Series	Fila generada para representar una ". ^{En} trada creada por duplicado".

Cuadro 1: Descripción de los parámetros y variables utilizados en la función de depuración duplicados.

7.6. Código: Función registra salida.

```
1 '''REVISION DE SALIDAS'''
2 try:
3     marcaje['cierre'] = "No tiene cierre"
4     marcaje = marcaje.sort_values(by=['rut', 'día',
5                                     'Hora']).reset_index(drop=True)
6
7     for indice in range(len(marcaje.index)):
8         if marcaje.at[indice, 'entrada/salida'] == "01":
9             registraSalida(marcaje, indice)
10
11     marcaje = marcaje.sort_values(by=['día', 'Hora',
12                                     'rut']).reset_index(drop=True)
13
14 except Exception as e:
15     print(f"Error DEPURACION - proceso TIENE SALIDA: {e}")
16     return None
```

```
1 def registraSalida(marcaje, indice):
2     # Obtener la fila actual
3     fila = marcaje.iloc[indice]
4
5     # Buscar posibles salidas válidas después de esta entrada
6     posibles_salidas = marcaje[
7         (marcaje['rut'] == fila['rut']) &
8         (marcaje['día'] == fila['día']) &
9         (marcaje['entrada/salida'] == "03") &
10        (marcaje.index > indice)
11    ]
12
13    if not posibles_salidas.empty:
14        # Tomar la primera salida encontrada
15        salida_index = posibles_salidas.index[0]
16
17        # Marcar tanto la entrada como la salida
18        marcaje.at[indice, 'cierre'] = "Tiene cierre"
19        marcaje.at[salida_index, 'cierre'] = "Tiene cierre"
20
21        # Llamada recursiva para buscar un cierre adicional desde la fila de
22        # salida
23        registraSalida(marcaje, salida_index)
24
25    return
```

En esta función se realiza un proceso de verificación y registro de salidas en un DataFrame llamado `marcaje`, que contiene datos organizados en columnas como `rut`, `día`, `Hora`, y `entrada/salida`. Este proceso evalúa si cada entrada ("01") tiene una salida correspondiente ("03") y marca los registros con una etiqueta de cierre.

Proceso General

1. Función principal (bloque try):

- Inicializa la columna `cierre` con el valor "No tiene cierre" para todos los registros.
- Ordena el `DataFrame` por las columnas `rut`, `día`, y `Hora` para facilitar la identificación de pares entrada/salida.
- Itera sobre cada fila del `DataFrame`:
 - Si encuentra una entrada ("01"), llama a la función `registraSalida`.

2. Función `registraSalida`:

- Verifica si el registro actual tiene una salida válida asociada:
 - La salida debe tener un valor "03" en la columna `entrada/salida`.
 - El `rut` de ambos registros debe coincidir.
 - El registro de salida debe ocurrir después del registro de entrada.
- Si encuentra una salida válida, actualiza ambos registros con el valor "Tiene cierre" en la columna `cierre`.
- Si no encuentra una salida inmediata, busca la siguiente salida válida más adelante en el `DataFrame`.

3. Salida final:

- Al finalizar, el `DataFrame` `marcaje` contiene las marcas correspondientes en la columna `cierre`, indicando cuáles entradas tienen salidas válidas asociadas.

7.7. Parámetros: Función registra salida.

Nombre	Tipo/Descripción	Uso
<code>marcaje</code>	<code>DataFrame</code>	Input principal que contiene los datos de entradas y salidas.
<code>indice</code>	<code>int</code>	Índice actual de la fila que se está evaluando en el <code>DataFrame</code> .
<code>i</code>	<code>int</code>	Índice auxiliar usado por <code>registraSalida</code> para buscar la salida correspondiente a una entrada.
<code>entrada/salida</code>	<code>str</code>	Columna que contiene los valores "01" (entrada) o "03" (salida).
<code>rut</code>	<code>str</code>	Identificador único para agrupar las entradas y salidas de una persona o entidad.
<code>día</code>	<code>str</code> o <code>datetime</code>	Columna que contiene la fecha del registro.
<code>Hora</code>	<code>str</code> o <code>datetime</code>	Columna que contiene la hora del registro.
<code>cierre</code>	<code>str</code> (inicialmente "No tiene cierre", actualizado a "Tiene cierre")	Columna utilizada para marcar si un registro de entrada tiene una salida asociada.
<code>Exception e</code>	<code>Exception</code>	Variable que captura cualquier error durante la ejecución del bloque <code>try</code> , facilitando el diagnóstico en caso de problemas.
<code>try/except</code>	Control de errores	Maneja posibles errores durante el procesamiento del <code>DataFrame</code> , asegurando que el programa no falle por datos inconsistentes o mal estructurados.

Cuadro 2: Descripción de los parámetros y variables utilizados en la función de registro de salidas.

7.8. Código: Función salidas saltadas.

```
1 ''' FALTA SALIDA'''
2 try:
3     marcaje = faltaSalida(marcaje, reglas)
4     marcaje = marcaje.sort_values(by=['día', 'Hora',
5         'rut']).reset_index(drop=True)
6 except Exception as e:
7     print(f"Error DEPURACION - proceso FALTA SALIDA: {e}")
8     return None

1 def faltaSalida(marcaje, reglas):
2     salida = marcaje.copy()
3
4     for i, row in salida.iterrows():
5
6         error = 'Salida automatica corregida'
7
8         # Si el registro es una entrada y no tiene salida
9         if row['entrada/salida'] == "01" and row['cierre'] == "No tiene cierre":
10
11             codigoHorario = int(row['Codigo'])
12
13             # Buscar el horario correspondiente en reglas
14             regla = reglas[reglas['Codigo'] == codigoHorario]
15             if not regla.empty:
16                 HorarioSalida = regla.iloc[0]['salida']
17                 horaSalida = regla.iloc[0]['horaSal']
18                 minutoSalida = regla.iloc[0]['minutoSal']
19
20                 # Si la hora de salida por regla es mayor que la hora actual de
21                 # la fila (sale el mismo día)
22                 if horaSalida > row['hora'] or (horaSalida == row['hora'] and
23                     minutoSalida > row['minuto']):
24                     # Actualizar fila que no tiene cierre
25                     salida.at[i, 'cierre'] = "Tiene cierre"
26
27                     # Crear nueva fila y añadirla al DataFrame
28                     nueva_filas = row.copy()
29                     nueva_filas['entrada/salida'] = "03"
30                     nueva_filas['hora'] = horaSalida
31                     nueva_filas['minuto'] = minutoSalida
32                     nueva_filas['Hora'] = HorarioSalida
33                     nueva_filas['Error'] = error
34                     nueva_filas['cierre'] = "Tiene cierre"
35
36                     salida = pd.concat([salida, pd.DataFrame([nueva_filas])],
37                         ignore_index=True)
38
39             # Ordenar el DataFrame por día y hora
40             salida = salida.sort_values(by=['día', 'Hora', 'rut']).reset_index(drop=True)
41
42     return salida
```

La función `faltaSalida` tiene como propósito corregir automáticamente los registros de entrada sin salida, añadiendo una salida correspondiente según las reglas definidas en el DataFrame `reglas`. El proceso comienza creando una copia del DataFrame `marcaje`, que se almacena en la variable `salida`. Luego, recorre cada fila del DataFrame `salida` para verificar si existe alguna entrada sin cierre.

Si se encuentra una entrada sin cierre ("No tiene cierre"), la función busca en el DataFrame `reglas` el horario de salida correspondiente a la entrada mediante el código de horario (`Codigo`). Si se encuentra una regla válida, se compara la hora de salida establecida en las reglas con la hora actual de la entrada. Si la hora de salida de la regla es posterior a la hora de la entrada, se actualiza el valor de `cierre` en el registro de entrada a "Tiene cierre".

A continuación, se crea una nueva fila que representa la salida correspondiente, con los mismos valores que la entrada, pero con los campos de hora y minutos modificados según la regla de salida. Esta nueva fila se agrega al DataFrame `salida`. Finalmente, el DataFrame `salida` se ordena por día, hora y rut, y se devuelve como resultado la tabla con las entradas y las salidas corregidas.

7.9. Parámetros: Función salidas saltadas.

Nombre	Tipo/Descripción	Uso
<code>marcaje</code>	DataFrame	Contiene los registros de entrada y salida, junto con la columna <code>cierre</code> que marca si una entrada tiene salida asociada. Es el input principal de la función.
<code>reglas</code>	DataFrame	Contiene las reglas para determinar las horas y minutos de salida de cada código de horario. Se utiliza para determinar las salidas faltantes.
<code>salida</code>	DataFrame	Copia del DataFrame <code>marcaje</code> , utilizada para realizar las correcciones sin modificar el original.
<code>error</code>	str	Mensaje que indica que la salida ha sido corregida automáticamente.
<code>codigoHorario</code>	int	Código de horario extraído de la columna <code>Codigo</code> de <code>marcaje</code> , utilizado para buscar las reglas de salida en <code>reglas</code> .
<code>HorarioSalida</code>	str	Nombre del horario de salida asociado al código de horario.
<code>horaSalida</code>	int	Hora de salida extraída de la regla correspondiente.
<code>minutoSalida</code>	int	Minuto de salida extraído de la regla correspondiente.
<code>nueva_fila</code>	Series	Fila creada para representar la salida correspondiente, basada en la entrada sin salida.

Cuadro 3: Descripción de los parámetros y variables utilizados en la función `faltaSalida`.

7.10. Código: Función marcaje opuesto.

```

1  ''' MARCA OPUESTO '''
2  try:
3      marcaje = marcaOpuesto(marcaje, reglas)
4  except Exception as e:
5      print(f"Error DEPURACION - proceso MARCA OPUESTO: {e}")
6      return None

1  def marcaOpuesto(marcaje, reglas):
2      df = marcaje.copy()
3      for i, row in df.iterrows():
4          codigoHorario = int(row['Codigo'])
5          rut = row['rut']
6          for j, row2 in reglas.iterrows():
7              if (codigoHorario == row2['Codigo']):
8                  horaEntrada = row2['horaEn']
9                  minutoEntrada = row2['minutoEn']
10                 horaSalida = row2['horaSal']
11                 minutoSalida = row2['minutoSal']
12                 break
13
14             # Buscar salida con una ventana de 10 minutos en donde se marca entrada y
15             # corregir
16             if (row['hora'] == horaSalida and (minutoSalida - 10) <= row['minuto']
17                 and row['minuto'] <= (minutoSalida + 10) and rut == row['rut']
18                 and row['entrada/salida'] == "01" and row['Error'] == "0k" and
19                 row['cierre'] != "Tiene cierre"):
20                 df.at[i, 'entrada/salida'] = "03"
21                 if (row['Error'] == '0k'):
22                     df.at[i, 'Error'] = "Entrada invertida a salida"
23                 else:
24                     df.at[i, 'Error'] += ", Entrada invertida a salida"
25
26             return df

```

Esta ultima función detecta y corrige casos en los que una entrada se marca incorrectamente como salida y viceversa, utilizando reglas horarias y ventanas de tiempo específicas.

```

1      for i, row in df.iterrows():
2
3          codigoHorario = row['Codigo']
4          rut = row['rut']
5
6          for j, row2 in reglas.iterrows():
7              if (codigoHorario == row2['Codigo']):
8                  horaEntrada = row2['horaEn']
9                  minutoEntrada = row2['minutoEn']
10                 horaSalida = row2['horaSal']
11                 minutoSalida = row2['minutoSal']
12                 break

```

La función primero busca el horario esperado para el codigoHorario de cada fila de df, asignando las horas y minutos de entrada y salida del DataFrame reglas.

```
1      # Buscar salida con una ventana de 10 minutos en donde se marca entrada y
      corregir
2      if (row['hora'] == horaSalida and (minutoSalida - 10) <= row['minuto']
      and row['minuto'] <= (minutoSalida + 10) and rut == row['rut']
3      and row['entrada/salida'] == "01" and row['Error'] == "Ok" and
      row['cierre'] != "Tiene cierre"):
4
5      df.at[i, 'entrada/salida'] = "03"
6
7      if (row['Error'] == 'Ok'):
8          df.at[i, 'Error'] = "Entrada invertida a salida"
9      else:
10         df.at[i, 'Error'] += ", Entrada invertida a salida"
```

Este bloque busca salidas mal registradas como entradas en una ventana de 10 minutos alrededor de la hora de salida. Si encuentra un registro de 'entrada/salida' de 1 (entrada) en el horario de salida, lo cambia a 3 (salida) y anota el error.

Al finalizar todos estos pasos se devuelve 'df' y se finalizan los pasos de depuración del marcaje.

7.11. Parámetros: Función `marcaje opuesto`.

Nombre	Tipo/Descripción	Uso
<code>marcaje</code>	<code>DataFrame</code>	Contiene los registros de entrada y salida. Es el input principal de la función.
<code>reglas</code>	<code>DataFrame</code>	Contiene las reglas para determinar las horas de entrada y salida según el código de horario. Se utiliza para buscar los horarios correspondientes para cada entrada y salida.
<code>df</code>	<code>DataFrame</code>	Copia del <code>DataFrame</code> <code>marcaje</code> , utilizado para realizar las correcciones sin modificar el original.
<code>codigoHorario</code>	<code>int</code>	Código de horario, utilizado para buscar las reglas de entrada y salida correspondientes en <code>reglas</code> .
<code>rut</code>	<code>str</code>	Identificador único del grupo, extraído de la columna <code>rut</code> de <code>marcaje</code> , utilizado para filtrar las reglas.
<code>horaEntrada</code>	<code>int</code>	Hora de entrada extraída de las reglas correspondientes al <code>codigoHorario</code> .
<code>minutoEntrada</code>	<code>int</code>	Minuto de entrada extraído de las reglas correspondientes al <code>codigoHorario</code> .
<code>horaSalida</code>	<code>int</code>	Hora de salida extraída de las reglas correspondientes al <code>codigoHorario</code> .
<code>minutoSalida</code>	<code>int</code>	Minuto de salida extraído de las reglas correspondientes al <code>codigoHorario</code> .
<code>row</code>	<code>Series</code>	Fila actual del <code>DataFrame</code> <code>df</code> en la iteración, que contiene los datos del registro de entrada o salida.

Cuadro 4: Descripción de los parámetros y variables utilizados en la función `marcaOpuesto`.

7.12. Código: Fin depuración y retorno de marcaje.

```
1  try:
2      # Se termina la depuración y se eliminan las columnas que no sirven
3
4      data = marcaje
5
6      # Save the DataFrame to a CSV file
7      path_temp = '/app/temp/datos_procesados.csv'
8      data.to_csv(path_temp, index=False, encoding='utf-8')
9      print(f"Se guarda archivo procesado en {path_temp}. [Mod Dep]")
10
11
12  except Exception as e:
13      print(f"Error DEPURACION - proceso GUARDADO ARCHIVO DEPURADO: {e}")
14      return None
```

La parte final del proceso toma el dataframe 'marcaje' depurado, y se guarda en un nuevo dataframe llamado 'data' el cual se ocupa para crear un nuevo archivo CSV llamado 'datos_procesados.csv'.

Finalmente la ruta de este archivo se genera en '/app/temp/datos_procesados.csv'.

8. Función de Validación

8.1. Acerca del módulo

Este módulo es responsable de realizar las validaciones definidas por el usuario en el módulo de visualización y exportación de archivos. Se divide en dos partes:

- Validación de las correcciones realizadas por el usuario.
- Exportación del archivo después de realizar las correcciones.

8.2. Requisitos previos

La validación comienza cuando el usuario selecciona las correcciones que desea revertir. Para ello, debe marcar las casillas de verificación (checkbox) correspondientes a las filas que no deben ser corregidas por el sistema. Una vez seleccionadas todas las filas a revertir, el usuario debe presionar el botón "Descargar archivo". Esto inicia el proceso de validación y exportación de los datos.

Clean System

Cargar archivo marcajes

Cerrar Cuenta

Contenido de marcajes

Filtros

RUT

Ingrese RUT

Hora Desde

--:--

Hora Hasta

--:--

Entrada/Salida

Cualquiera

Condición

Cualquiera

Días (click para mostrar)

Código horario

Ingrese código

Aplicar Filtros

<input type="checkbox"/>	006	01	1e1b5fe1b1c27ba878afa4ee9a3bcd0a144e82e2ff54a34b6cc9c60fa8277c0	7	57	8	1	24	Ok
<input type="checkbox"/>	006	01	34beed08901c2d0b845543ebce462f3a2064defe4401eab2bf4f6a3cef081c	7	57	8	1	24	Ok
<input type="checkbox"/>	007	01	7b7917a0898926c2c0e23df3100eb8fbc52f313af18458aab418905f471d88	7	57	8	1	24	Ok
<input type="checkbox"/>	006	01	d8e8164ede82834858130b3110cb02f5100329574e27d7e9a400f740df1192	7	57	8	1	24	Ok
<input type="checkbox"/>	003	01	e1b0248d941d5635e250dfb23a1e68bba5ba7e29cf6430ccb28ca30a9bde8a3	7	57	8	1	24	Ok
<input type="checkbox"/>	001	01	f2fe5e4fcec0e681d3b635cdf548133de6fb5ba703c61a915c15055be1e2a813	7	57	8	1	24	Ok
<input checked="" type="checkbox"/>	007	01	1a0da39251583e767526b7001f26186bb824a3f99bc870e10a32a11ce994727ae	7	58	8	1	24	Entrada duplicada
<input checked="" type="checkbox"/>	007	03	1a0da39251583e767526b7001f26186bb824a3f99bc870e10a32a11ce994727ae	7	58	8	1	24	Salida creada por duplicado
<input type="checkbox"/>	006	01	2a3b4b95fcd0b6d0b7025355f115656e7d5f2c301d8011c459227777dfc33eb	7	58	8	1	24	Ok
<input type="checkbox"/>	006	01	39486abcc3f57a57449f93df4a5b9fac5f74ab677a9e1b29a213df821508ef	7	58	8	1	24	Ok
<input type="checkbox"/>	007	01	a15166fc89a669b9355cae635fec4127267df893ab6fd7ecdb280735d9b635a	7	58	8	1	24	Ok
<input type="checkbox"/>	002	01	d88c8b8f13ca4a6e03bae690a7b4265d2a050ad07a46f67f35d8b1e511c41b3	7	58	8	1	24	Ok
<input type="checkbox"/>	006	01	db1b73f5a0bda004558726ce57775509d9c1715871adb5b359d8fd2f2a3358d1	7	58	8	1	24	Ok
<input type="checkbox"/>	004	01	03e6a1d3c48350e9c0da40478183e12abb4c5015935ace7b189f910021624317	7	59	8	1	24	Ok
<input type="checkbox"/>	003	01	123c2068c51270225f7d424474ac14b7b40c5e202b78aa40ed3f3c6c2b66fd	7	59	8	1	24	Ok
<input type="checkbox"/>	006	01	77c8fde926bfe8f485d2f4c45d5c205c882b80f54770eb318efca4f9b18c488	7	59	8	1	24	Ok
<input type="checkbox"/>	004	01	db32b6c9f8e2a2b62dc937e515b4a89883b5aa20ba52c6cfcba8bbd1ebfcd3	7	59	8	1	24	Ok
<input type="checkbox"/>	003	01	073126b7e824f0d0a6d54240645c33f3d6eac23b257a12df01243a5f32f65a	8	0	8	1	24	Ok
<input type="checkbox"/>	006	01	5540503c193ed5c6d781ee263510182acb5449bebd9c0215effe432d219f8ed	8	0	8	1	24	Ok

Código: [1: Entrada] [3: Salida]

Descargar Archivo

Figura 1: Marcar las casillas de verificación

8.3. Código: Conexión con frontend.

Como ya fue mencionado antes, el módulo de validación toma en cuenta todas las filas seleccionadas por el usuario para ser revertidas en el módulo de visualización, para llevar a cabo este proceso se recuperan todas las filas que fueron marcadas, esto se hace mediante la implementación de una función que está dentro del módulo de exportación que es llamada desde el form “process_selected_rows”.

```
1      <form id="process_selected_rows" action="/download_csv" method="POST">
2      {% for idx, row in enumerate(table_data) %}
3          <tr>
4              <td class="checkbox-cell">
5                  <input type="checkbox" class="row-checkbox"
6                      name="selected_rows" value="{{ idx }}">
7              </td>
8              {% for cell in row %}
9                  <td>{{ cell }}</td>
10             {% endfor %}
11         </tr>
12     {% endfor %}
13 </form>
14
15
16 <div class="mt-3 text-end">
17 <button type="submit" form="process_selected_rows" class="btn
18     btn-primary">Descargar Archivo</button>
19 </div>
```

8.4. Código: Exportación.

```
1 @visualizacion.route('/download_csv', methods=['GET', 'POST'])
2 @user_login_required
3 def download_csv():
4
5     # Recuperar filas seleccionadas como JSON
6     selected_rows = request.form.getlist('selected_rows')
7
8     file_path = '/app/temp/datos_procesados.csv'
9     df = pd.read_csv(file_path, dtype={"Codigo": str, "a": str, "entrada/salida":
10         str, "b": str, "c": str, "d": str, "e": str, "f": str, "g": str, "h": str, "i":
11         str, "j": str, "k": str})
12
13     if not selected_rows:
14
15         try:
16             df_final = df.copy()
17             df_final.drop(columns=['Hora', 'Error', 'cierre'], inplace=True)
18             df_final['hora'] = df_final['hora'].apply(lambda x: f"{x:02}")
19             df_final['minuto'] = df_final['minuto'].apply(lambda x: f"{x:02}")
20             df_final['mes'] = df_final['mes'].apply(lambda x: f"{x:02}")
21             df_final['día'] = df_final['día'].apply(lambda x: f"{x:02}")
22             df_final['año'] = df_final['año'].apply(lambda x: f"{x:02}")
23
24             crearHistorial(df, None)
25
26             # Guardar dataframe en csv
27             df_final.to_csv(file_path, index=False, header=False)
28
29             # Enviar archivo para descargar
30             return send_file(file_path,
31                             as_attachment=True,
32                             download_name="filtered_data.csv",
33                             mimetype='text/csv')
34
35         except Exception as e:
36             print(f"Error al generar el archivo CSV: {e}")
37             flash("Error al generar el archivo CSV.", "error")
38             return redirect('/visualizacion')
```

```
1     else:
2         try:
3             # Convertir las filas seleccionadas en DataFrame
4             columnas = ["Codigo", "entrada/salida", "rut", "hora", "minuto",
5                         "mes", "día", "año", "Error"]
6
7             try:
8                 selected_rows = [json.loads(row) for row in selected_rows]
9
10            except Exception as e:
11                print("Error al cargar Filas: ", e)
12
13            df_selected = pd.DataFrame(selected_rows, columns=columnas)
14
15            # Convertir la columna 'día' a tipo entero
16            df_selected["día"] = df_selected["día"].astype(int)
17
18            df_final = validar(df, df_selected)
19
20            df_final = df_final.iloc[:, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
21                                         12, 13, 14, 15, 16, 17, 18]]
22            print("YA SE GUARDO LAS FILAS ELEGIDAS")
23
24            # Guardar dataframe en csv
25            df_final.to_csv(file_path, index=False, header=False)
26
27            # Enviar archivo para descargar
28            return send_file(file_path,
29                             as_attachment=True,
30                             download_name="filtered_data.csv",
31                             mimetype='text/csv')
32
33        except Exception as e:
34            print(f"Error al generar el archivo CSV: {e}")
35            flash("Error al generar el archivo CSV.", "error")
36            return redirect('/visualizacion')
```

Una vez dentro de la función de exportación se obtienen las filas seleccionadas mediante la función `request.form.getlist('selected_rows')` y se guardan en `'selected_rows'` para posterior uso, también se recupera el csv del marcaje corregido por el sistema y se guarda en `'df'`.

Al finalizar se verifica si existen filas seleccionadas, en caso de que no existan se procede a exportar el archivo con `'df_final'`. Para ello, primero se seleccionan las filas importantes para el usuario y después se importa.

Si se detecta que el usuario marcó filas que requieren ser revertidas, se entra al proceso de validación.

8.5. Parámetros: Exportación

Nombre	Tipo/Descripción	Uso
<code>selected_rows</code>	<code>list</code>	Lista de filas seleccionadas enviadas desde el formulario como JSON. Contiene los datos a procesar si se seleccionaron registros específicos.
<code>file_path</code>	<code>str</code>	Ruta al archivo CSV temporal donde se guardan los datos procesados antes de enviarlos al usuario.
<code>df</code>	<code>DataFrame</code>	<code>DataFrame</code> cargado desde el archivo <code>file_path</code> , que contiene los datos originales a procesar.
<code>df_final</code>	<code>DataFrame</code>	<code>DataFrame</code> que contiene los datos finales procesados, filtrados y preparados para ser guardados en el CSV.
<code>columnas</code>	<code>list</code>	Lista de nombres de columnas utilizadas para convertir las filas seleccionadas (<code>selected_rows</code>) en un <code>DataFrame</code> .
<code>df_selected</code>	<code>DataFrame</code>	<code>DataFrame</code> generado a partir de las filas seleccionadas (<code>selected_rows</code>), que se valida y procesa antes de generar el CSV.
<code>json</code>	módulo	Módulo utilizado para cargar y decodificar las filas seleccionadas (<code>selected_rows</code>) desde formato JSON.
<code>validar</code>	función	Función auxiliar que valida las filas seleccionadas (<code>df_selected</code>) en el contexto de los datos originales (<code>df</code>).
<code>crearHistorial</code>	función	Función auxiliar que registra un historial de las operaciones realizadas sobre el <code>DataFrame df</code> .
<code>flash</code>	función	Función utilizada para mostrar mensajes de error al usuario en caso de problemas durante el procesamiento.
<code>send_file</code>	función	Función utilizada para enviar el archivo CSV procesado al usuario como respuesta de descarga.

Cuadro 5: Descripción de los parámetros y variables utilizados en la función `download_csv`.

8.6. Código: Validación.

Cuando se reconoce que el usuario seleccionó filas estas son almacenadas, para luego llamar a la función validar, la que retornará un df para df_final que tendrá todos los cambios solicitados por el usuario.

```
1      try:
2          # Convertir las filas seleccionadas en DataFrame
3          columnas = ["Codigo", "entrada/salida", "rut", "hora", "minuto",
4                      "mes", "día", "año", "Error"]
5
6          try:
7              selected_rows = [json.loads(row) for row in selected_rows]
8
9          except Exception as e:
10             print("Error al cargar Filas: ", e)
11
12             df_selected = pd.DataFrame(selected_rows, columns=columnas)
13
14             # Convertir la columna 'día' a tipo entero
15             df_selected["día"] = df_selected["día"].astype(int)
16
17             df_final = validar(df, df_selected)
18
19             df_final = df_final.iloc[:, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
20                                         12, 13, 14, 15, 16, 17, 18]]
21             print("YA SE GUARDO LAS FILAS ELEGIDAS")
22
23             # Guardar dataframe en csv
24             df_final.to_csv(file_path, index=False, header=False)
25
26             # Enviar archivo para descargar
27             return send_file(file_path,
28                             as_attachment=True,
29                             download_name="filtered_data.csv",
30                             mimetype='text/csv')
```

```
1 import pandas as pd
2 from historial import crearHistorial
3
4 def validar(df_corregido, selected_rows):
5
6     # Combina las columnas que quieres usar como identificadores
7     cols_identificadores = ["Codigo", "entrada/salida", "rut", "hora", "minuto",
8                             "mes", "día", "año", "Error"]
9
10    # Filtra el DataFrame para obtener las filas que coincidan
11    indices = []
12    for _, selected_row in selected_rows.iterrows():
13        mask = (df_corregido[cols_identificadores] ==
14                selected_row[cols_identificadores]).all(axis=1)
15        indices.extend(df_corregido[mask].index.tolist())
16
17    try:
18
19        df = df_corregido.copy()
20        df['hora'] = df['hora'].apply(lambda x: f"{x:02}")
21        df['minuto'] = df['minuto'].apply(lambda x: f"{x:02}")
22        df['mes'] = df['mes'].apply(lambda x: f"{x:02}")
23        df['día'] = df['día'].apply(lambda x: f"{x:02}")
24        df['año'] = df['año'].apply(lambda x: f"{x:02}")
25
26        for i in indices:
27
28            # Acceder a los valores actuales en la columna 'Error'
29            errores = df.loc[i, 'Error']
30
31            if (errores != "Ok"):
32                lista_errores = errores.split(", ")
33
34                for error in lista_errores:
35                    if error == "Entrada duplicada":
36                        print("Se revierte ENTRADA DUPLICADA")
37
38                    # Verificar si existe la fila siguiente y eliminarla
39                    if i + 1 in df.index and df.at[i + 1, 'Error'] == "Salida
40                        creada por duplicado":
41                        print(f"Eliminando fila {i + 1} (Salida que sigue a
42                            la Entrada duplicada)")
43                        df.drop(index=i + 1, inplace=True)
44
45                    # Si la fila siguiente está en 'indices', eliminarla
46                    # de la lista
47                    if i + 1 in indices:
48                        indices.remove(i + 1)
```

```
1         elif error == "Salida duplicada":
2             print("Se revierte SALIDA DUPLICADA")
3
4             # Verificar si existe la fila anterior y eliminarla
5             if i - 1 in df.index and df.at[i - 1, 'Error'] == "Salida
6                 creada por duplicado":
7                 print(f"Eliminando fila {i - 1} (Salida que sigue a
8                     la Entrada duplicada)")
9                 df.drop(index=i - 1, inplace=True)
10
11             # Si la fila anterior está en 'indices', eliminarla de la
12             lista
13             if i - 1 in indices:
14                 indices.remove(i - 1)
15
16         elif error == "Salida automatica corregida":
17             print("Se revierte correccion SALIDA AUTOMATICA")
18
19             # Eliminar fila
20             df.drop(index = i, inplace=True)
21
22         elif error == "Entrada invertida a salida":
23             print("Se revierte ENTRADA INVERTIDA A SALIDA")
24
25             df.at[i, 'entrada/salida'] = "01"
26
27         elif error == "Entrada creada por duplicado":
28             print("Se eliminara ENTRADA CREADA POR DUPLICADO")
29
30             # Si la fila siguiente no está en 'indices', agregarla a
31             la lista
32             if i + 1 not in indices:
33                 indices.append(i + 1)
34
35         elif error == "Salida creada por duplicado":
36             print("Se eliminara SALIDA CREADA POR DUPLICADO")
37
38             # Si la fila anterior no está en 'indices', agregarla a
39             la lista
40             if i - 1 not in indices:
41                 indices.append(i - 1)
42
43         df.at[i, 'Error'] = "Correcciones revertidas"
44
45         crearHistorial(df_corregido, indices)
46
47         return df
48
49     except Exception as e:
50         print(f"Error en el proceso de correccion: {e}")
```

1. **Identificación de filas afectadas:** Se utilizan columnas clave como identificadores para determinar las filas específicas que deben ser procesadas.
2. **Formateo de datos:** Antes de procesar las filas, los campos de fecha y hora (**hora**, **minuto**, **mes**, **día**, **año**) se formatean para garantizar que tengan un formato consistente.
3. **Corrección basada en errores:** Según el tipo de error identificado en cada fila (almacenado en la columna **Error**), se aplican acciones específicas:
 - Revertir entradas o salidas duplicadas.
 - Eliminar filas creadas por errores automáticos.
 - Ajustar salidas invertidas.
 - Actualizar el estado del error después de la corrección.
4. **Registro de historial:** Una vez procesados los datos, se utiliza una función auxiliar para registrar las operaciones realizadas sobre los registros seleccionados.
5. **Devolución de resultados:** La función devuelve un nuevo **DataFrame** que refleja los datos corregidos.

8.7. Parámetros: Validación.

Nombre	Tipo/Descripción	Uso
<code>df_corregido</code>	DataFrame	DataFrame con los datos corregidos iniciales que serán validados y procesados para revertir o ajustar registros según las reglas definidas.
<code>selected_rows</code>	DataFrame	DataFrame con las filas seleccionadas por el usuario para ser procesadas en la validación. Contiene columnas como <code>Codigo</code> , <code>rut</code> , <code>entrada/salida</code> , entre otras.
<code>cols_identificadores</code>	list	Lista de nombres de columnas usadas como identificadores únicos para encontrar coincidencias entre <code>df_corregido</code> y <code>selected_rows</code> .
<code>indices</code>	list	Lista de índices en <code>df_corregido</code> que coinciden con las filas seleccionadas en <code>selected_rows</code> . Estos índices se procesan durante la validación.
<code>errores</code>	str	Cadena de texto en la columna <code>Error</code> de <code>df_corregido</code> , que almacena las anomalías asociadas a cada registro.
<code>lista_errores</code>	list	Lista de errores individuales obtenidos al dividir la cadena <code>errores</code> por comas.
<code>crearHistorial</code>	función	Función auxiliar importada desde <code>historial.py</code> , utilizada para registrar un historial de las operaciones realizadas sobre <code>df_corregido</code> .
<code>mask</code>	Series booleano	Máscara booleana que identifica si una fila de <code>df_corregido</code> coincide con una fila de <code>selected_rows</code> en las columnas identificadoras.
<code>pd</code>	módulo	Módulo de pandas, utilizado para manipular y procesar los DataFrame.

Cuadro 6: Descripción de los parámetros y variables utilizadas en el archivo `validacion.py`.

9. Función de Historial.

9.1. Acerca del módulo.

El módulo del historial es el encargado de generar un documento csv en el que se guardan las correcciones generadas por el módulo de depuración, excluyendo todos los cambios que el usuario quiso corregir y fueron corregidos en el módulo de validación.

9.2. Código: Llamada a función.

```
1      ...
2          df.at[i, 'Error'] = "Correcciones revertidas"
3
4      crearHistorial(df_corregido, indices)
5
6      return df
7      ...
```

La llamada a la función del historial se hace en la función validación, los parámetros que son pasados es el dataframe corregido por la función de validación que contiene los registros revertidos según lo explicado en el módulo de validación, y la lista de los índices de los registros seleccionados.

9.3. Código: Función historial.

```
1 import pandas as pd
2 from datetime import date
3
4 def crearHistorial(df, indices):
5     try:
6         # Si indices es None, asigna una lista vacía
7         if indices is None:
8             indices = []
9
10        # Inicializar el DataFrame con columnas
11        historial = pd.DataFrame(columns=['usuario', 'rut', 'fecha', 'error',
12                                         'cambio'])
13
14        # Recuperar usuario de archivo
15        temp_path = "/app/temp/username.txt"
16        with open(temp_path, "r") as file:
17            usuario = file.read()
18
19        fecha_actual = date.today().strftime("%d/%m/%Y") # día/mes/año
20
21        for index, row in df.iterrows():
22            errores = row['Error']
23            correcciones = []
24            if ((errores != "Ok" and index not in indices) and \
25                (errores != "Salida creada por duplicado") and (errores !=
26                    "Entrada creada por duplicado")):
27
28                lista_errores = errores.split(", ")
29
30                for error in lista_errores:
31                    if error == "Entrada duplicada":
32                        print("Se reconoce en historial ENTRADA DUPLICADA")
33                        correcciones.append("Se crea una salida para entrada
34                                            duplicada")
35                    elif error == "Salida duplicada":
36                        print("Se reconoce en historial SALIDA DUPLICADA")
37                        correcciones.append("Se crea una entrada para salida
38                                            duplicada")
39                    elif error == "Salida automatica corregida":
40                        print("Se reconoce en historial SALIDA AUTOMATICA")
41
42                        correcciones.append(f"Se crea hora de salida según reglas
43                                            ({row['Hora']})")
```

```
1         elif error == "Entrada invertida a salida":
2             print("Se reconoce en historial ENTRADA INVERTIDA A
3                 SALIDA")
4             correcciones.append("Se invierte marcaje de tipo entrada a
5                 marcaje de tipo salida")
6
7         cambios = f"Se hacen los siguientes cambios: {'',
8             '.join(correcciones)}"
9         rut = row['rut'] # Acceso correcto a la columna 'rut'
10
11         # Crear una nueva fila como DataFrame
12         nueva_fila = pd.DataFrame([
13             'usuario': usuario,
14             'rut': rut,
15             'fecha': fecha_actual,
16             'error': errores,
17             'cambio': cambios
18         ])
19
20         # Agregar la nueva fila al historial
21         historial = pd.concat([historial, nueva_fila], ignore_index=True)
22
23     elif (errores == "Salida creada por duplicado" and index - 1 not in
24         indices) or (errores == "Entrada creada por duplicado" and index +
25         1 not in indices):
26         if errores == "Entrada creada por duplicado":
27             print("Se reconoce en historial ENTRADA CREADA POR
28                 DUPLICADO")
29             correcciones.append("Entrada creada para corregir salida
30                 duplicada")
31         elif errores == "Salida creada por duplicado":
32             print("Se reconoce en historial SALIDA CREADA POR
33                 DUPLICADO")
34             correcciones.append("Salida creada para corregir entrada
35                 duplicada")
36
37         cambios = f"Se hacen los siguientes cambios: {'',
38             '.join(correcciones)}"
39         rut = row['rut'] # Acceso correcto a la columna 'rut'
40
41         # Crear una nueva fila como DataFrame
42         nueva_fila = pd.DataFrame([
43             'usuario': usuario,
44             'rut': rut,
45             'fecha': fecha_actual,
46             'error': errores,
47             'cambio': cambios
48         ])
49
50         historial = pd.concat([historial, nueva_fila],
51             ignore_index=True)
```

```
1      # Guardar el historial en el archivo CSV
2      file_path = '/app/temp/historial.csv'
3
4      #historial.to_csv(file_path, index=False, header=False)
5      historial.to_csv(file_path, index=False, header=['usuario', 'rut',
6                  'fecha', 'error', 'cambio'])
7
8  except Exception as e:
9      print("Error al generar el historial: ", e)
```

Funcionamiento

1. **Inicialización:** Se inicializa un `DataFrame` vacío con columnas: `usuario`, `rut`, `fecha`, `error`, y `cambio`.
2. **Usuario y fecha:** El nombre del usuario se recupera desde un archivo de texto (`username.txt`), y la fecha actual se obtiene en formato `día/mes/año`.
3. **Iteración sobre los datos:**
 - Para cada fila en `df`, se verifica si contiene errores que deban ser registrados en el historial.
 - Se omiten las filas cuyos índices están en la lista `indices`.
4. **Registro de correcciones:**
 - Dependiendo del error identificado en la columna `Error`, se determinan las correcciones necesarias y se registra el cambio correspondiente.
 - Se agregan filas al `DataFrame historial` con detalles sobre el usuario, el error identificado, las correcciones realizadas y el `rut` asociado.
5. **Guardado del historial:** El `DataFrame historial` se guarda como un archivo CSV en `/app/temp/historial.csv`.

Casos de errores reconocidos

Los siguientes tipos de errores son procesados y registrados:

- **Entrada duplicada:** Se crea una salida para corregir una entrada duplicada.
- **Salida duplicada:** Se crea una entrada para corregir una salida duplicada.
- **Salida automática corregida:** Se registra la corrección automática de una salida.
- **Entrada invertida a salida:** Se cambia una entrada a tipo salida.

9.4. Parámetros: Historial.

Parámetro	Tipo	Descripción
<code>df</code>	<code>DataFrame</code>	Contiene los datos procesados donde se identifican los errores y se aplican correcciones.
<code>indices</code>	<code>list</code> o <code>None</code>	Lista de índices de filas que deben omitirse en el historial. Si es <code>None</code> , se asigna una lista vacía.

Cuadro 7: Parámetros de entrada de la función `crearHistorial`.