Políticas y Buenas Prácticas de Programación

- Políticas y Buenas Prácticas de Programación
- Página nuevaRevisión (Peer Review + Code Reviewer)
- Plantilla Base de Pruebas Funcionales
- Guía de Estilo y Nomenclatura para Desarrollo
- Estandarización de Servicios
- Directiva Técnica para Pruebas Controladas de Emisión de Comprobantes
- Directiva Técnica para Pruebas Controladas de Emisión de Comprobantes
- Directiva para el uso de rutinas
- Documento Técnico

Políticas y Buenas Prácticas de Programación

1. Objetivo

Establecer un conjunto de políticas, normas y buenas prácticas que permitan:

- Mejorar la calidad del código fuente.
- Reducir errores y observaciones por parte del equipo, encargados, QAs y jefes de área.
- Alinear al equipo con estándares y reglas establecidos en la empresa para lo que es temas referidos al código, el desarrollo de los sistemas y proyectos, y todo lo que abarca en funciones a temas de TI y sistemas.
- Facilitar el trabajo colaborativo y escalabilidad de los sistemas.
- Facilitar el mantenimiento, escalabilidad y reutilización del software.
- Establecer una base estandarizada entre todos los desarrolladores del equipo.
- Estandarizar el trabajo entre todos los desarrolladores del equipo.

2. Alcance

Estas políticas aplican a todos los miembros del equipo de desarrollo involucrados en la creación, mantenimiento o revisión de código en cualquier tecnología y proyecto.

3. Buenas Prácticas Generales

- Estandarización del Código
    - Nombrar clases, funciones y variables de forma clara, descriptiva y consistente.
    - Tabular adecuadamente el código, así como también las funciones, las llaves, el inicio y cierre de cada bloque de código, se recomienda usar la tabulación de 4 espacios de cada línea respectiva.
    - Se recomienda en mayor medida escribir funciones, métodos, variables y constantes en inglés para facilitar la escalabilidad y colaboración con equipos internacionales.
    - Usar CamelCase para funciones, métodos y variables, PascalCase para clases.
    - Guiarse de la documentación oficial de cada lenguaje de programación o frameworks para nombres de archivos, estructura de carpetas y convenciones de escritura.

- Principios de Diseño
  - Usar patrones de diseño cuando sea apropiado.
  - Dividir responsabilidades: separación de lógica de negocio, presentación y acceso a datos.
- Reutilización y Mantenimiento
  - Aplicar el principio DRY (Don't Repeat Yourself): evitar duplicación de lógica, código o constantes.
  - Promover el desarrollo de funciones y componentes reutilizables.
  - Refactorizar bloques de código duplicados o que podrían abstraerse.
  - Centralizar validaciones, utilidades y reglas de negocio comunes.
- Validaciones
  - Validar todos los datos entrantes desde formularios, APIs, bases de datos o usuarios.
  - Establecer reglas claras de validación y sanitización según el contexto.
  - Rechazar entradas inválidas y notificar de forma segura al usuario.
- Manejo de Errores y Excepciones
  - Capturar y manejar errores de forma controlada usando try-catch o equivalentes.
  - Generar mensajes de error útiles para el usuario sin exponer detalles técnicos.
  - Implementar logging estructurado para errores técnicos (Ej. con Monolog, Winston, Logback).
  - Clasificar errores por tipo: lógicos, de red, validaciones, etc.
- Legibilidad y Simplicidad
  - Priorizar la claridad del código sobre la complejidad.
  - Evitar comentarios innecesarios y en su lugar usar nombres descriptivos.
  - Evitar tener líneas y bloques de códigos comentados innecesariamente.
  - Se puede comentar el por qué se hace algo, no el qué (el qué debe ser obvio por el código).
  - Nombrar adecuadamente variables, métodos y clases para que el código sea autoexplicativo.
  - Comentar solo cuando sea necesario para explicar decisiones complejas o no

evidentes.

- Refactorización Continua
  - Eliminar código muerto, duplicado o innecesario.
  - Modularizar funciones o clases extensas.
  - Dividir responsabilidades que estén agrupadas incorrectamente.

4. Control de Calidad y QA

- Antes de subir código
  - Verificar que el código pase todos los tests existentes.
  - Verificar que no se introduzcan errores en funcionalidades existentes.
  - Ejecutar pruebas manuales básicas del flujo que se viera afectado.
  - Validar flujos críticos y posibles regresiones.
  - Revisar que el código siga los estándares establecidos.

- Para los Pull Requests
  - Usar ramas semánticas (feature/, bugfix/, refactor/, etc.).
  - Validar que el PR esté relacionado a un ticket o funcionalidad o historia de usuario.
  - Describir claramente el propósito del PR, cambios realizados y cómo probarlos.

- Flujo de Validación

Todo commit deberá seguir el siguiente flujo antes de ser fusionado a producción:

1. Pruebas y verificación por el equipo de QA.
2. Revisión técnica por un Code Reviewer.
3. Autorización final antes del merge.

- Formato de commits
  - Estilo de mensaje:
    - (Descripción breve). task: url-erp-de-la-tarea
  - Ejemplo:

- “Corrección creación de usuario. task: https://erp.overskull.com/app/task/TASK-2025-XXXXX”
- módulo de entrega task: tareas
- módulo de entrega fix: correcciones o incidencias
- módulo de entrega hotfix: prioridad de observación

5. Testing

- Incluir pruebas unitarias, de integración y de extremo a extremo cuando sea necesario.
- Usar herramientas de pruebas según tecnología (PHPUnit, Jest, PyTest, etc.).
- Mantener una cobertura de código significativa (>70%), sin caer en pruebas innecesarias.
- Validar casos de uso normales, límites y errores.

6. Seguridad

- Validar y sanitizar todas las entradas del usuario.
- Evitar inyecciones SQL, XSS, CSRF y otros ataques comunes.
- No exponer información sensible en errores, logs o interfaces.
- No hardcodear credenciales o secretos: usar archivos .env o gestores como Vault.
- Aplicar el principio de menor privilegio en base de datos, APIs y servicios.

7. Documentación

- Mantener actualizada la documentación técnica (README, Diagramas).
- Documentar estructuras complejas, decisiones técnicas y configuraciones.
- Usar anotaciones o herramientas automáticas cuando sea posible (PHPDoc, JSDoc, Swagger).

8. Control de Versiones

- Usar Git correctamente: commits atómicos, ramas claras, y mensajes estándar.
- Seguir la convención de mensajes de commit: (Descripción breve). task: url-tarea
- Validar que el código pase por QA y Code Review antes de hacer merge.
- Evitar subir archivos temporales, .env, dependencias o archivos generados.

9. Responsabilidad del Desarrollador

Cada desarrollador debe:

- Entregar código probado, funcional y conforme a los estándares.
- Participar activamente en la revisión de código de sus compañeros.
- Aprender de las observaciones de QA o revisores técnicos.
- Refactorizar su propio código cuando detecte oportunidades de mejora.
- Compartir soluciones reutilizables y colaborar con otros miembros del equipo.

10. Seguimiento y Mejora Continua

- Los líderes técnicos y Code Reviewers pueden proponer cambios.
- Se comunicarán las actualizaciones a todo el equipo, y se incluirán sesiones de formación si es necesario.
- Se realizará capacitaciones al personal implicado y ligado en todo este proceso de desarrollo de los proyectos, así como reuniones cada que se requiera y fuese necesario o para explicar también algunas actualizaciones o puntos que no hayan quedado claro.

Página nuevaRevisión (Peer Review + Code Reviewer)

Revisión (Peer Review + Code Reviewer)

Asignación de Pares (base)

- Jordan → Fiorella, Flavio (6)
- Marco → Gabriel, Pierito (2)
- Edson → Daisy, Elian (2)
- José → Jordy, Marcelo (2)

Rotación semanal

Revisión Cruzada entre Seniors

- Jordan ↔ Edson / APP
- Marco ↔ José

Reglas Anti?Cuello de Botella

- Tiempo de revisión: máx. 15 min (peer) / 15 min (checklist code reviewer).
- Toda revisión se responde en ≤ 2 horas hábiles. Si no, escalar al “revisor del día”.
- Automatización obligatoria. (PENDIENTE)

Checklist Peer Review (rápido) / PROGRAMADOR

1. Nombres claros, funciones cortas, sin duplicaciones.
2. Validaciones de entrada y mensajes de error adecuados.
3. Evitar queries dentro de bucles; revisar índices/joins.
4. Los archivos coinciden con el alcance.

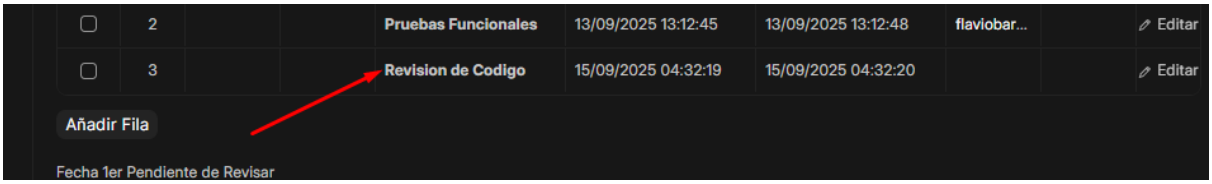
Checklist Code Reviewer (crítico)

1. Seguridad: sanitización, queries parametrizadas.
2. Performance: consultas pesadas detectadas, uso de caché cuando aplica.
3. Pruebas/CI: linters ok, tests mínimos verdes, build sin warnings críticos.(PENDIENTE)
4. Higiene del MR: sin cambios ajenos, commits limpios, descripción con rutas/archivos.

Flujo Operativo (resumen)

1. Dev (misma rama dev) → el programador documenta archivos/rutas tocadas en la tarea.(TRABAJANDO)

2. Peer Review → el par asignado revisa código + funcionalidad mínima. (INSERT DE REVISIÓN CÓDIGO)



3. QA → prueba funcional completa. (ÁREA DE QA)
4. Commit/MR por tarea a pre-prod → Code Reviewer aplica checklist crítico. (PENDIENTE DE PULL)
5. Merge a pre-prod → validación en entorno estable.

Plantilla Base de Pruebas Funcionales

Plantilla Base de Pruebas Funcionales

Este documento sirve como formato estándar para documentar las pruebas funcionales que cada programador debe realizar antes de enviar su desarrollo a QA. Debe completarse para cualquier tipo de tarea (roles, incidencias, métricas, validaciones, creación de doctypes, etc.).

1) Recursos

- Enlace/s del módulo, apartado, sistema, doctype, etc.
- Enlace/s de apk para pruebas

2) Precondiciones

- Usuarios/Roles necesarios:
- Datos iniciales cargados:
- Estados previos del sistema:
- Estados previos del sistema:
- Permisos o configuraciones activas:
- Integraciones requeridas (API, ERP, App, etc.)
- Flujo de la tarea
- Flujo de la tarea
- Flujo de la tarea
- Condiciones del flujjo

3) Casos de Prueba Funcionales

Regla: Para cada tarea documentar al menos 3 casos válidos y 1 caso no válido.

- Descripción
- Pasos
- Datos / Condiciones
- Resultado esperado
- Evidencia (Screenshot)

Ejemplos de flujos válidos: creación de registro, consulta, modificación.
Ejemplo de flujo no válido: intento con datos inválidos o permisos insuficientes.

4) Casos Negativos / Seguridad

- Acceso sin permisos → debe bloquear con mensaje.

- Datos inválidos o campos vacíos → debe mostrar validaciones.
- Condiciones fuera de rango/tiempo → debe manejarse con error controlado.

5) Consideraciones (opcional)

6) Rutas / Funciones / Opciones Tocadas

- Rutas frontend involucradas:
- Endpoints backend modificados:
- Archivos de configuración cambiados:
- Botones, menús u opciones del sistema probadas
- Campo:
- 

7) Evidencias

Incluir capturas de pantalla de cada caso probado

8) Checklist del Programador

[ ] Evidencias adjuntadas
[ ] Precondiciones cumplidas
[ ] Casos válidos y no válidos documentados
[ ] Rutas/funciones tocadas listadas / DETALLE DE PULL

[ ] Consideraciones (OPCIONAL)

9) Resultado del Peer Review

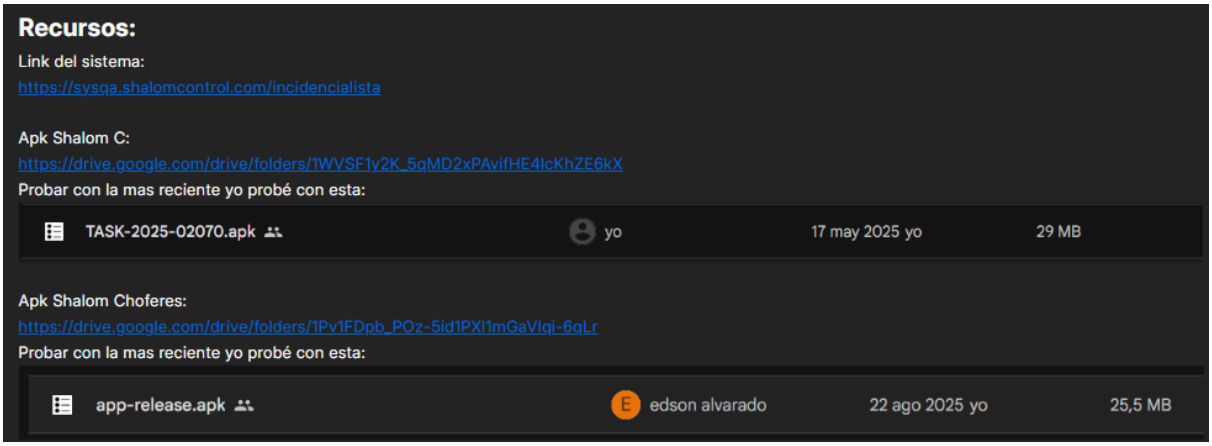
• Revisor:
• Fecha:
• Observaciones:
• Decisión: Aprobado / Requiere cambios

10) Checklist del Code Reviewer

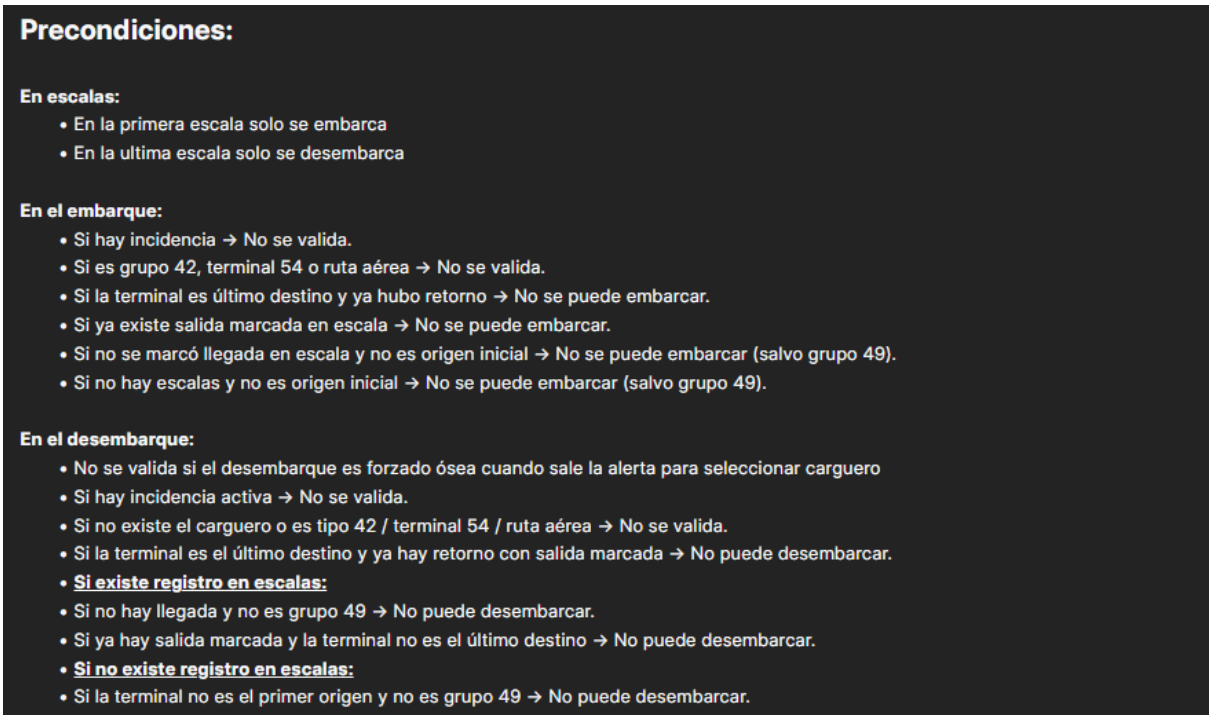
[ ] Seguridad (inputs sanitizados, queries parametrizadas, CSRF/XSS)
[ ] Performance (consultas optimizadas, uso de caché)
[ ] Arquitectura (respeta MVC, modularidad)
[ ] Higiene del MR (commits limpios, cambios aislados)
[ ] Evidencias revisadas y conformes

Ejemplo: https://erp.overskull.com/app/task/TASK-2025-04449

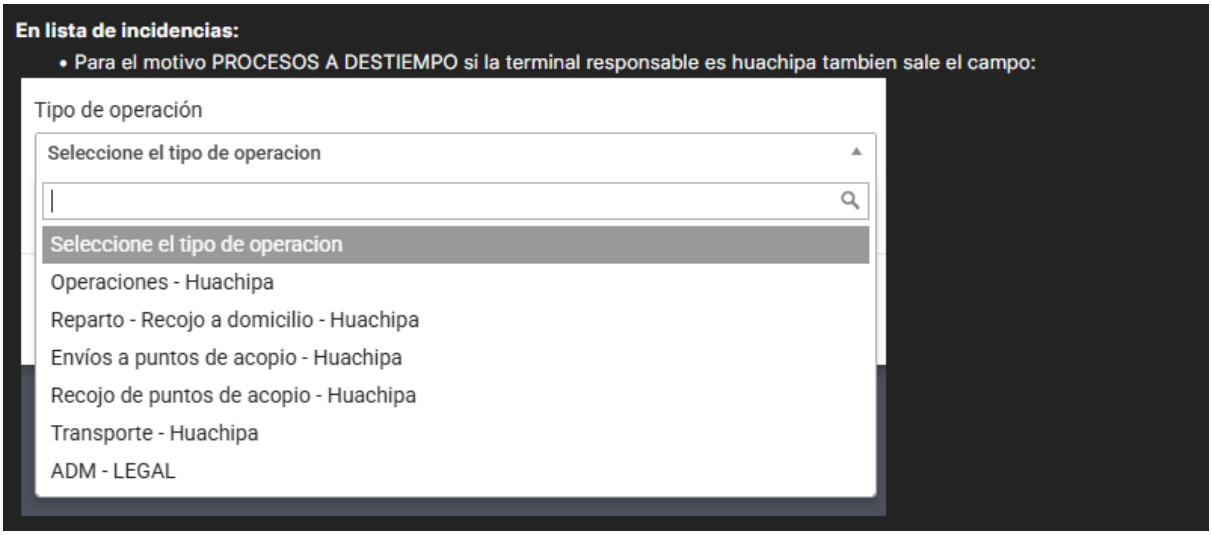
Recursos:



Precondiciones:



Consideraciones:



Guía de Estilo y Nomenclatura para Desarrollo

Guía de Estilo y Nomenclatura para Desarrollo

1. Uso de Variables

- Se deben utilizar nombres de variables descriptivos y significativos.
- Se debe emplear camelCase para nombrar variables en todos los lenguajes.
- El idioma de las variables y funciones debe ser inglés para estandarización.
- Evitar abreviaciones innecesarias y nombres genéricos como $temp, $data, $var.

 Ejemplo correcto:

```
$isActiveUser = true;

$attemptCount = 3;
```

 Ejemplo incorrecto:

```
$usuarioActivo = true;

$numeroDeIntentos = 3;
```

---

2. Uso de Funciones y Métodos

- Las funciones deben tener nombres descriptivos.
- Usar camelCase para nombrarlas.
- La función debe indicar claramente su propósito.
- El idioma de las funciones debe ser inglés.

 Ejemplo correcto:

```
function getUserById($id) {

    // Código...

}
```

□□ Ejemplo incorrecto:

```
function obtenerUsuarioPorId($id) {

    // Código...

}
```

---

3. Nomenclatura de Archivos

- Los nombres de archivos deben ser claros y descriptivos.
- Se debe usar PascalCase para los nombres de archivos.
- Para archivos de clases y modelos, incluir "Model" o "Controller" en el nombre.
- Los nombres de archivos deben estar en inglés.

□ Ejemplo correcto:

```
UserMaintenanceModal.php

UserController.php
```

□□ Ejemplo incorrecto:

```
MantenimientoModal.php

UsuarioController.php
```

---

4. Indentación y Formato

- Se debe utilizar 4 espacios por nivel de indentación (no tabs).
- El código debe estar correctamente indentado para mejorar la legibilidad.

 Ejemplo correcto:

```
function validateUser($user) {

    if ($user->isActive) {

        return true;

    }

    return false;

}
```

 Ejemplo incorrecto:

```
function validarUsuario($usuario) {

if ($usuario->activo) {

return true;

}

return false;

}
```

---

5. Uso de Comentarios

- Los comentarios deben ser claros y aportar valor.
- Se deben utilizar comentarios solo cuando sea necesario.
- Se prefiere el uso de PHPDoc en funciones y clases.

 Ejemplo correcto:

```
/**

 * Calculates the total with applied discount.

 *

 * @param float $subtotal

 * @param float $discount Discount percentage (e.g., 0.15 for 15%)

 * @return float

 */

function calculateTotal($subtotal, $discount) {

    return $subtotal - ($subtotal * $discount);

}
```

 Ejemplo incorrecto:

```
// Esta función calcula el total

function calcularTotal($subtotal, $descuento) {

    return $subtotal - ($subtotal * $descuento);

}
```

---

6. Uso de Constantes

- Las constantes deben escribirse en mayúsculas y con guiones bajos.
- Deben estar en inglés.

 Ejemplo correcto:

```
define('TIME_LIMIT', 300);


const MAX_ATTEMPTS = 5;
```

 Ejemplo incorrecto:

```
define('TiempoLimite', 300);


const max_intentos = 5;
```

---

7. Manejo de Espacios en Blanco

- Dejar una línea en blanco entre bloques lógicos de código.
- No dejar espacios en blanco innecesarios dentro de líneas.

 Ejemplo correcto:

```
if ($isActive) {

    processUser($user);

}
```

```
logAction($user);
```

❌ Ejemplo incorrecto:

```
if ($activo) { procesarUsuario($usuario); }

registrarAccion($usuario);
```

---

8. Longitud Máxima de Funciones

- Una función no debe exceder las 50 líneas de código.
- Idealmente, debe contener entre 5 y 20 líneas.
- Si una función supera las 50 líneas, se debe dividir en funciones más pequeñas.

✅ Ejemplo correcto:

```
function processOrder($order) {

    $user = validateUser($order->userId);

    $discount = calculateDiscount($user->type);

    $total = calculateTotal($order->items, $discount);

    handlePaymentAndStock($order, $total);

    return "Order processed successfully";

}
```

❌ Ejemplo incorrecto:

```
function procesarOrden($orden) {

    $usuario = getUserById($orden->userId);

    if (!$usuario) { throw new Exception("Usuario no encontrado"); }

    $descuento = ($usuario->tipo == "VIP") ? 0.15 : 0.10;

    $subtotal = 0;

    foreach ($orden->items as $item) { $subtotal += $item->precio * $item->cantidad; }

    $total = $subtotal - ($subtotal * $descuento);

    if ($total > 1000) { aplicarPromocion($orden); }

    registrarPago($orden, $total);

    actualizarStock($orden->items);

    enviarConfirmacion($usuario, $orden);

    return "Orden procesada con éxito";

}
```

Estandarización de Servicios

Estandarización de Servicios

1. Estructura de Respuesta

Todos los servicios deben seguir una estructura de respuesta estándar en formato JSON:

```
{

    "success": false,

    "message": "Mensaje de servicio",

    "data": []

}
```

Reglas para la Respuesta:

- success: Indica si la operación fue exitosa (true o false).
- message: Mensaje claro y descriptivo sobre el resultado de la operación.
- data: Contiene la información devuelta por el servicio. Si no hay datos, debe ser un array vacío ([]).

 Ejemplo de Respuesta Exitosa:

```
{

    "success": true,

    "message": "User retrieved successfully",

    "data": {

        "id": 1,
```

```
        "name": "John Doe",
        "email": "johndoe@example.com"
    }
}
```

 Ejemplo de Respuesta con Error:

```
{
    "success": false,
    "message": "User not found",
    "data": []
}
```

---

2. Generación de Documentación (Swagger)

- Todo servicio debe contar con su respectiva documentación en Swagger.
- La documentación debe incluir:
  - Descripción clara del servicio.
  - Parámetros requeridos y opcionales.
  - Ejemplos de respuestas esperadas.
  - Códigos de estado HTTP posibles.

 Ejemplo de Documentación Swagger en OpenAPI 3.0 (YAML):

paths:

```
  /users/{id}:
    get:
      summary: Get user by ID
```

```
description: Retrieves a user using their unique ID.

parameters:

  - name: id

    in: path

    required: true

    schema:

      type: integer

responses:

  "200":

    description: User retrieved successfully

    content:

      application/json:

        schema:

          type: object

          properties:

            success:

              type: boolean

            message:

              type: string

            data:
```

```
            type: object
            properties:
              id:
                type: integer
              name:
                type: string
              email:
                type: string
"404":
  description: User not found
  content:
    application/json:
      schema:
        type: object
        properties:
          success:
            type: boolean
          message:
            type: string
          data:
```

```
                    type: array

                    items: {}
```

---

3. Códigos de Estado HTTP Estándar

- 200 OK - Solicitud exitosa.
- 500 Internal Server Error - Error interno del servidor.

---

4. Seguridad y Buenas Prácticas

- Validar siempre los datos de entrada antes de procesarlos.
- Evitar exponer información sensible en los mensajes de error.

---

5. Uso de Estructuras de Control

Para mejorar la consistencia y calidad del código, se deben seguir estas reglas al manejar estructuras de control:

Condicionales (if/else)

- Evitar múltiples anidaciones innecesarias.
- Utilizar early return para mejorar la legibilidad.

 Ejemplo Correcto:

```
if not user:

    return {"success": false, "message": "User not found", "data": []}

return {"success": true, "message": "User retrieved", "data": user}
```

 Ejemplo Incorrecto:

```
if user:

    return {"success": true, "message": "User retrieved", "data": user}

else:

    return {"success": false, "message": "User not found", "data": []}
```

Bucles (for, while)

- Utilizar comprensiones de listas cuando sea posible.
- Evitar bucles anidados innecesarios.

 Ejemplo Correcto:

```
even_numbers = [x for x in numbers if x % 2 == 0]
```

 Ejemplo Incorrecto:

```
even_numbers = []

for x in numbers:

    if x % 2 == 0:

        even_numbers.append(x)
```

Directiva Técnica para Pruebas Controladas de Emisión de Comprobantes

Fecha: 24 / 11 / 2025
Dirigido a: Equipos de Desarrollo, QA, Soporte y Gestión
Emitido por: Área de Sistemas – Jefatura de Desarrollo

---

1. **Objetivo**
   Establecer los lineamientos obligatorios para la ejecución de pruebas relacionadas con la emisión de comprobantes electrónicos dentro de los entornos de QA / PRE-Producción / Producción Controlada, con el fin de evitar impactos contables, tributarios y operativos.

   ---

2. **Alcance**
   La presente directiva aplica de manera estricta a todas las áreas internas y proveedores externos que realicen pruebas funcionales o técnicas dentro de los sistemas corporativos que generan comprobantes electrónicos (boletas o facturas).

   ---

3. **Lineamientos Obligatorios**
   1. **DNIs autorizados para pruebas de emisión de Boleta**
      A partir de la fecha, únicamente se permite emitir **BOLETAS** de prueba usando los siguientes DNIs:
      - **75013406**
      - **70503353**
      - **71422696**
      - **75527393**
      - **71423703**

      ---

2. **RUC autorizados para pruebas de emisión de Factura**
   Solo se permite emitir **FACTURAS** de prueba utilizando los siguientes RUC:
   - **10750134063**
   - **10705033531**
   - **10714226962**

---

3. **Obligación posterior a la emisión**
   Todo comprobante emitido durante pruebas **debe ser ANULADO**, mediante:
   - Nota de crédito correspondiente, o
   - Anulación del comprobante según procedimiento interno.

   No se aceptará bajo ninguna circunstancia dejar comprobantes de prueba activos.

---

4. **Faltas y Sanciones**
   El incumplimiento de esta directiva constituye falta grave.
   Se aplicarán las siguientes medidas:
   - **Primera falta:** Amonestación formal escrita.
   - **Reiteración o incumplimiento crítico:** Evaluación para suspensión o despido
     inmediato.

---

5. **Vigencia**
   Esta directiva entra en vigencia inmediata a partir de su comunicación.

Directiva Técnica para Pruebas Controladas de Emisión de Comprobantes

Directiva Técnica para Pruebas Controladas de Emisión de Comprobantes

Fecha: 24 / 11 / 2025
Dirigido a: Equipos de Desarrollo, QA, Soporte y Gestión
Emitido por: Área de Sistemas – Jefatura de Desarrollo

---

1. Objetivo

Establecer los lineamientos obligatorios para la ejecución de pruebas relacionadas con la emisión de comprobantes electrónicos dentro de los entornos de QA / PRE-Producción / Producción Controlada, con el fin de evitar impactos contables, tributarios y operativos.

2. Alcance

La presente directiva aplica de manera estricta a todas las áreas internas y proveedores externos que realicen pruebas funcionales o técnicas dentro de los sistemas corporativos que generan comprobantes electrónicos (boletas o facturas).

---

3. Lineamientos Obligatorios

3.1 DNIs autorizados para pruebas de emisión de Boleta

A partir de la fecha, únicamente se permite emitir BOLETAS de prueba usando los siguientes DNIs:

- **75013406 (DESARROLLO)**

- **70503353 (GESTIÓN)**
- **71422696 (QA)**
- **75527393 (QA)**
- **71423703 (QA)**

3.2 RUC autorizados para pruebas de emisión de Factura

Solo se permite emitir FACTURAS de prueba utilizando los siguientes RUC:

- **10750134063 (DESARROLLO)**
- **10705033531 (GESTIÓN)**
- **10714226962 (QA)**

3.3 Obligación posterior a la emisión

Todo comprobante emitido durante pruebas debe ser ANULADO, mediante:

- Nota de crédito correspondiente, o
- Anulación del comprobante según procedimiento interno.

No se aceptará bajo ninguna circunstancia dejar comprobantes de prueba activos.

3.4 Rango Máximo de Monto Autorizado

Todo comprobante emitido durante pruebas **deberá respetar el rango máximo de monto permitido,** comprendido entre:

S/ 1.00 (un sol) como mínimo y S/ 50.00 (cincuenta soles) como máximo.

En caso sea estrictamente necesario exceder dicho monto, el responsable de la prueba **deberá informar previamente y solicitar autorización expresa al Jefe de Programación.**

---

4. Faltas y Sanciones

El incumplimiento de esta directiva constituye falta grave.

Como medida disciplinaria aplicará directamente:

- **Suspensión**, y
- **Evaluación de despido inmediato**, según el nivel de afectación ocasionada.

---

5. Vigencia

Esta directiva entra en vigencia inmediata a partir de su comunicación.

Directiva para el uso de rutinas

Directiva Técnica para Control de Ejecución de CRONs mediante Token de Seguridad

Fecha: 01 / 12 / 2025
Dirigido a: Equipos de Desarrollo, QA, Soporte, Gestión y Operaciones
Emitido por: Área de Sistemas – Jefatura de Desarrollo

---

1. Objetivo

Establecer los lineamientos obligatorios para la ejecución de tareas programadas (**CRONs**) dentro de los sistemas corporativos, mediante el uso de un **token de seguridad** asociado a cada ruta. Esta medida tiene como finalidad prevenir la ejecución no autorizada, accesos indebidos, alteraciones en procesos operativos y riesgos de seguridad informática.

---

2. Alcance

La presente directiva aplica a todos los sistemas internos, servicios web, microservicios, módulos y proveedores externos que operen o interactúen con rutas HTTP utilizadas para la ejecución de CRONs en los entornos de **QA, PRE-Producción y Producción.**

---

3. Lineamientos Obligatorios

3.1 Uso obligatorio de Token

Toda ruta de CRON deberá utilizar el parámetro **cron_token** como mecanismo de autenticación obligatoria.
Ejemplo:

https://qawebservices.shalomcontrol.com/api/v1/cron/restriccion-clave-clientes?cron_token=.qaOverskull.

Ningún CRON podrá ejecutarse sin el token correspondiente.

3.2 Gestión del Token

El token será:

- Generado exclusivamente por el Área de Programación
- Único por entorno (QA / PRE / PROD)
- De carácter confidencial y restringido
- No podrá ser publicado en repositorios, compartido por mensajería abierta ni almacenado en lugares no seguros

Cualquier divulgación indebida constituirá falta grave.

3.3 Validación del Token

Todo endpoint deberá validar el token antes de ejecutar la rutina.
En caso de token inválido o inexistente, el sistema deberá devolver:

HTTP 403 – Acceso Denegado

Queda prohibido exponer rutas sin protección o saltar la validación definida.

3.4 Solicitudes de Ejecución y Cambios

Toda solicitud relacionada con:

- Activación de CRON
- Ejecución manual
- Modificación de una ruta existente
- Migración a otro entorno

deberá ser comunicada previamente al **Área de Programación**, quienes autorizarán o rechazarán la intervención según los lineamientos internos.

No se permiten ejecuciones directas sin aprobación previa.

---

4. Faltas y Sanciones

El incumplimiento de esta directiva constituye **falta grave**.

Dependiendo del impacto técnico, de seguridad u operativo ocasionado, podrán aplicarse:

- Suspensión temporal del acceso a los entornos
- Evaluación de medidas disciplinarias de mayor severidad, incluyendo terminación del vínculo laboral o contractual

---

5. Vigencia

La presente directiva entra en vigencia **de manera inmediata** a partir de su comunicación y es de **cumplimiento obligatorio** para todas las áreas involucradas.

Documento Técnico

1.Introducción

Actualmente, el área de desarrollo recibe solicitudes de cambios, errores o mejoras a través de diversos canales sin una estructura estandarizada. Esto puede generar ambigüedades, retrasos en la planificación, errores en la implementación y retrabajo.

Con el objetivo de mejorar la eficiencia, trazabilidad y calidad del desarrollo, se propone formalizar el uso de tickets técnicos bajo una estructura determinada.

2.Objetivos

- Estandarizar la recepción y redacción de tickets técnicos.
- Facilitar la comprensión entre las áreas funcionales y el equipo de desarrollo.
- Asegurar que cada ticket sea implementado, probado y validado adecuadamente.
- Aumentar la velocidad de entrega y reducir errores funcionales.

3.Propuesta de Estructura de Ticket Técnico

Cada ticket deberá contener los siguientes puntos:

3.1.- Rutas

- Detallar los módulos, pantallas o endpoints que intervienen, desde donde inicia el cambio hasta donde finaliza.
- Se pondrá dominios de QA
- Se colocará título o nombre del apartado y su ruta

3.2.- Contexto

- Explicar claramente el motivo del cambio: necesidad de negocio, cumplimiento normativo, mejora funcional, corrección de errores, etc.

3.3.- Descripción

- Resumen funcional del requerimiento. Puede incluir el formato “Como… quiero… para que…” si aplica.
- Relación de orden entre proceso y cambios a realizar
    - El orden debe de ser de manera numérica
    - Contener imágenes de referencia si es necesario
    - Para tareas con nuevos módulos o vistas, generar mockups

3.4.- Principales Casos de Prueba

- Escenario 1: Empleado con jornada diurna → no se asigna bono.
- Escenario 2: Empleado con jornada nocturna → bono de 50 soles agregado automáticamente.
- Escenario 3: Empleado con cambio de jornada en el mes → validar comportamiento esperado según políticas.

3.5.- Anexos y referencias

- Ejemplo: Podría ser tareas que son importantes a considerar, referencias de interfaces para tareas con app,etc.
- Para tareas complejidad 4 y 5 agregar un diagrama (curso bizagi)

4.Encabezados

□□ RUTAS

□□ CONTEXTO

□□ DESCRIPCIÓN

⚙ PRINCIPALES CASOS DE PRUEBA

□□ ANEXOS Y REFERENCIAS

Ejemplo: https://erp.overskull.com/app/task/TASK-2025-04209

5.Prioridad

5.1.- Urgente ?

- Entrega: Inmediata (hoy o máximo 24h).
- Impacto: Afecta procesos críticos, el comité o la operación de Shalom.
- Flexibilidad: Cero, debe resolverse ya.

5.2.- Alta ??

- Entrega: En la fecha programada.
- Impacto: Relevante para la operación o para otros equipos (si no se entrega, bloquea a otros).
- Flexibilidad: No se puede postergar.

5.3.- Media ?

- Entrega: En la fecha asignada, pero no es prioridad del área.
- Impacto: Aporta al trabajo, pero no detiene operaciones si se retrasa.
- Flexibilidad: Puede ajustarse el orden dentro de la semana, pero debe cumplirse en la fecha acordada.
- Ejemplo: Ajuste estético en un sistema, reporte interno no crítico, documentación de apoyo.

5.4.-Baja ??

- Entrega: Con fecha programada, pero se puede reprogramar si no se completa a tiempo.
- Impacto: El área puede seguir trabajando sin la tarea, aunque es necesaria más adelante.
- Flexibilidad: Alta, se mueve en la agenda según prioridades mayores.
- Ejemplo: Optimización de procesos, capacitaciones, mejoras preventivas, generación de manuales.

TAREAS CAPACITACIÓN

Mes	julio		agosto		septiembre		Total	
nombre_solicitante	Cant	Comp	Cant	Comp	Cant	Comp	**Cant**	**Comp**
Stephania Sistemas	32	69	29	72	6	24	**67**	**165**
Giancarlo Sistemas	27	89	29	88	5	16	**61**	**193**
Gheraldy Aliaga Fajardo	20	56	33	108	1	12	**54**	**176**
JAIR ADHEMAR CALIXTO RENGIFO	10	28	14	42	3	19	**27**	**89**
Total	**89**	**242**	**105**	**310**	**15**	**71**	**209**	**623**

TAREAS OVERSKULL

programador_nombre	julio	agosto	septiembre	Total
stephania	99	76	38	**213**
Gheraldy	55	91	20	**166**
giancarlo	52	72	27	**151**
Jair C	27	36	14	**77**
Total	**233**	**275**	**99**	**607**

6.Complejidad

NIVEL	TIEMPO ESTIMADO	TIPO DE TAREAS
COMPLEJIDAD 1	0 - 30 m	Restricciones básicas
COMPLEJIDAD 2	30 m - 1 hr y 30 m	Restricción complejas (tarifas), Apis básicas
COMPLEJIDAD 3	1 hr y 30 m - 3 hr	Barridos, Apis complejos
COMPLEJIDAD 4	3 hr - 5 hr	Reportes, Optimizaciones, Nuevos módulos complejos
COMPLEJIDAD 5	> 5 hr	Integraciones

- Creación de tareas