GUIA #2 – CALIDAD DE SOFTWARE

En el siguiente documento se plantea la actividad correspondiente a la aplicación de pruebas sobre el software desarrollado por cada grupo a la fecha, perteneciente al segundo corte en el que se incluyen los siguientes temas.

- Definición de endpoint.
- Aplicación de pruebas por objetivo y cumplimiento de aspectos de calidad.
- Aspectos a tener en cuenta por cada tipo de prueba.
- Registro de hallazgos.
- Control y seguimiento mediante el uso de stándares.

Se aplicarán los siguientes tipos de prueba sobre la parte representativa del backend o programación asociada a modelo de negocio, quien será la que manipule directamente los datos sobre la base de datos escogida por el grupo de desarrollo.

- Pruebas de caja blanca.
- Pruebas de estrés de servicio.
- Pruebas funcionales.

Para cada una de estas pruebas se debe tener en cuenta los siguientes campos (construir formato estándar para las pruebas que sea aplicado independiente del tipo desarrollado).

Campo	**Contenido Sugerido**
ID Prueba	Identificador único (CN-MOD-001, CB-MOD-002, etc) – caja negra – módulo – 1
Tipo de prueba	Según el tipo desarrollado, relacionado en la lista anterior.
Objetivo	Qué es lo que se desea verificar. Se debe detallar las funciones revisadas.
Precondiciones	Datos o estado que la aplicación debe cumplir para iniciar la prueba.
Datos de entrada	Descripción y tipos de datos que se deben suministrar.
Procedimiento	Paso a paso del ingreso, registro o resultado del proceso observado.
Resultado esperado	Considerando el flujo normal de la aplicación que se espera obtener.
Resultado real.	Evidencia del resultado arrojado por la aplicación.
Comentarios / Observaciones	Hallazgos realizados durante la prueba, si hay a lugar.
Prioridad / severidad	Según el product backlog, la prioridad del proceso y el impacto que se calcula se aplique sobre el equipo que desarrolla la tarea.
Frecuencia / repetición	Si la prueba debe repetirse varias veces, configuración de la prueba y variación entre pruebas.
Dependencias	Servicios externos involucrados, condiciones de funcionamiento y vigilancia de intercambio de datos.

A continuación, se detallan los datos que deben ser observados según el tipo de prueba.

1. Pruebas funcionales (Functional Testing): generalmente caja negra (no se requiere ver el código), centrándose en que las APIs cumplan los requisitos funcionales.

Checklist funcional:

- Validar endpoints principales (CRUD: GET, POST, PUT/PATCH, DELETE) Por cada uno se deberían hacer dos pruebas de uso sin mediar con frontend sino directamente con el backend, para la siguiente prueba.
- Verificar que se responde correctamente con los códigos HTTP esperados (200, 201, 204, etc.). Mostrar códigos y cabeceras recibidos en la prueba anterior para verificar consistencia.
- Si los códigos arrojados por la aplicación son de error, verificar respuesta al usuario desde Vista y consola del navegador, revelación de datos innecesarios – producción.
- Validar estructura del cuerpo (JSON/XML/etc.) según contrato / especificación o cabecera usada para la petición. Determinar si es la correcta acorde a la función ejecutada.
- Verificar valores, campos obligatorios y opcionales.
- Verificar validaciones de entrada: formatos (numérico, texto), rangos (fechas, números, mayor que o menor que), longitud, tipos de datos, existencia en tablas relacionadas, etc.
- Validar lógica de negocio específica (por ejemplo, si el cliente no tiene permiso no puede realizar acción; reglas de negocio, cálculos, transformaciones, etc.)
- Validar paginación / filtros / ordenamiento / búsqueda si están disponibles. Determinar el formato usado por el backend para mostrar la lista de registros de una tabla y sus propiedades útiles para paginación.
- Verificar manejo de errores: qué pasa si request mal formado, parámetros faltantes, etc.
- Verificar condiciones límite (“edge cases”) — por ejemplo strings vacíos, valores nulos, valores máximos/mínimos, etc.
- Verificar APIs bajo distintos roles/permisos (autenticado, no autenticado, distintos roles)

2. Pruebas de caja blanca (White-box testing) se mira el código, la estructura, lógica interna, cobertura, etc.

Checklist caja blanca:

- *Cobertura de código: líneas, ramas, condiciones, caminos posibles (statement, branch, path coverage)*. Establecimiento y análisis de todas las posibles rutas de ejecución que pueden seguirse a través del código, considerando combinaciones de decisiones y bucles. En conjunto, estos indicadores permiten evaluar la efectividad de las pruebas, asegurando que los distintos escenarios lógicos del programa sean validados y reduciendo el riesgo de errores ocultos en partes no ejecutadas del sistema.
- *Verificar lógica interna: bucles, condiciones, manejo de errores internos, excepciones.* analizar el flujo de ejecución dentro de **bucles**, comprobando que se repitan el número de veces esperado y que se detengan en las condiciones adecuadas; revisar las **condiciones lógicas** (`if`, `else`, `switch`) para confirmar que todas las posibles rutas del código producen los resultados correctos; y evaluar el **manejo de errores internos**, garantizando que las situaciones inesperadas se detecten y se gestionen sin provocar fallos del sistema. Asimismo, se verifica que las **excepciones** se lancen y capturen apropiadamente, evitando fugas de información o bloqueos en la ejecución. En conjunto, este proceso busca asegurar la solidez y fiabilidad del comportamiento interno del software bajo distintas circunstancias.
- *Verificación de optimización interna: algoritmos, complejidad, uso de recursos.* Examen de los algoritmos implementados para validar si resuelven los problemas de la manera más adecuada posible, evitando operaciones redundantes o ineficientes. También se estudia la complejidad temporal y espacial — cuánto tiempo y memoria requieren las funciones— para identificar posibles puntos de mejora en el rendimiento. Además, se supervisa el consumo de recursos como CPU, memoria, disco o red, con el fin de detectar sobrecargas innecesarias o fugas que

puedan afectar la estabilidad del sistema. En conjunto, esta verificación busca asegurar que el backend funcione de forma óptima, aprovechando eficientemente los recursos disponibles y manteniendo un equilibrio entre desempeño, escalabilidad y sostenibilidad del software.

- *Pruebas de integración / unitarias profundas: mocks/stubs para dependencias internas, verificaciones internas del estado.* verifican que cada componente o función individual del código funcione correctamente de forma aislada, utilizando herramientas como *mocks* o *stubs* para simular dependencias externas —por ejemplo, bases de datos, servicios o módulos que no se desean ejecutar directamente durante la prueba—. Por otro lado, las **pruebas de integración** examinan cómo interactúan entre sí los distintos componentes, asegurando que el intercambio de datos, las llamadas entre servicios y los flujos de ejecución conjuntos se realicen sin errores. En ambos casos, se incluyen **verificaciones internas del estado**, observando cómo cambian las variables, estructuras o registros internos del sistema tras ejecutar las operaciones. Este enfoque permite detectar fallos tanto en la lógica local de cada módulo como en la comunicación global del backend, garantizando coherencia, fiabilidad y correcto acoplamiento entre las partes del software.
- Validación de invariantes del sistema, propiedades internas que deberían mantenerse. Verificación de **propiedades internas fundamentales** del software en busca de que se mantengan siempre verdaderas, sin importar las operaciones o condiciones a las que se someta el sistema. Estas invariantes representan reglas o estados que deben conservarse para garantizar la coherencia y estabilidad del programa, como por ejemplo que un contador nunca sea negativo, que una transacción se complete por completo o se revierta, o que la suma de ciertos valores se mantenga constante tras diversas operaciones. Durante las pruebas, se observa cómo el sistema modifica sus datos y estructuras internas, verificando que dichas propiedades no se violen incluso ante errores, entradas extremas o comportamientos simultáneos.

3. Pruebas de rendimiento / estrés / carga (Performance / Stress Testing) se prueba hasta dónde aguanta el sistema, cómo se degrada ante carga alta, etc.

Checklist de estrés / rendimiento:

- Determinar cargas esperadas: usuarios concurrentes, peticiones por segundo, volumen de datos (bits o bytes). Documentar los medios a través de los que se determina la carga esperada.
- Pruebas de estrés (Stress): llevar sistema más allá de los límites esperados para ver comportamiento, fallos, degradaciones, recuperación. Documentar elementos observados y resultados obtenidos con evidencia.
- Pruebas de pico (Spike): ráfagas de tráfico repentinas.
- Pruebas de resistencia/duración (Endurance / Soak test): durante largo tiempo para ver fugas de memoria, acumulación de errores, degradación de performance con el tiempo, reducción en niveles de atención.
- Medición de tiempos de respuesta (latencia), percentiles (p.ej. 95-percentil, 99-percentil)
- Monitorización de recursos: CPU, memoria, uso de la red/disco, conexiones de BD, etc.
- Verificar escalabilidad: con aumento de usuarios, instancias, etc.
- Evaluar comportamiento ante servicios externos lentos o caídos.
- Verificar límites de timeout / retry / fallback.

4. Herramientas sugeridas para desarrollo de pruebas.

Objetivo	**Herramienta sugerida**
Pruebas rápidas locales	JMeter o k6
Integración con CI/CD	k6, Artillery
Simulación a gran escala	Gatling, BlazeMeter
Escenarios personalizados en Python	Locust
Pruebas distribuidas simples desde la nube	Loader.io
Scripts minimalistas en CLI	Vegeta

Estructura sugerida para informe de pruebas:

1. Portada y control de versión. Incluye información general del documento:
   - Nombre del proyecto y versión del backend.
   - Fecha de elaboración.
   - Autor(es) o equipo de QA.
   - Versión del informe (control de cambios).
   - Entorno de pruebas (dev, staging, preproducción).

2. Resumen ejecutivo. Una síntesis clara de los resultados:

   - Objetivo general de las pruebas.
   - Alcance cubierto (tipos de pruebas realizadas).
   - Principales hallazgos: rendimiento, estabilidad, errores críticos, cumplimiento funcional.
   - Conclusión general del estado de calidad del backend.

3. Objetivos y alcance. Define con precisión qué se evaluó:

   - Módulos, endpoints o servicios incluidos.
   - Límites del alcance (qué no se probó y por qué).
   - Tipos de pruebas aplicadas:
   - Caja blanca (código interno, lógica).
   - Rendimiento / estrés / carga.
   - Integración y unitarias.

4. Entorno de pruebas. Describe las condiciones bajo las cuales se realizaron las pruebas:

   - Hardware y sistema operativo.
   - Configuración del servidor backend (CPU, RAM, DB, conexiones).
   - Versiones de software (lenguaje, framework, base de datos, etc.).
   - Herramientas utilizadas: JMeter, k6, Postman, Jest, Locust, etc.
   - Variables de entorno o configuraciones especiales.

5. Estrategia y metodología. Explica el enfoque seguido:

   - Criterios de diseño de casos de prueba (basados en requerimientos, equivalencia, límites, etc.).
   - Metodología de caja negra y blanca.

- Uso de mocks, stubs o entornos simulados.
- Plan de ejecución: orden, priorización, iteraciones, regresión.
- Criterios de aceptación o éxito de cada prueba.

6. Diseño y ejecución de casos de prueba. Presenta los casos de prueba detallados en tablas o anexos. Cada caso debe contener:

Campo	Descripción
ID de prueba	Ejemplo: BE-FN-001
Tipo de prueba	Funcional / Caja blanca / Estrés / Integración
Descripción / objetivo	Qué se pretende verificar
Precondiciones	Estado o configuración previa necesaria
Entradas / datos de prueba	Parámetros, payloads, credenciales
Procedimiento	Pasos para ejecutar la prueba
Resultado esperado	Qué debería ocurrir
Resultado obtenido	Qué ocurrió realmente
Estatus	Aprobado / Fallido / En observación
Comentarios	Logs, excepciones, métricas, evidencias

7. Cobertura de pruebas. Evalúa el grado de completitud:

- Cobertura funcional: porcentaje de endpoints o funcionalidades probadas.
- Cobertura de código: líneas, ramas y caminos ejecutados (reportes de herramientas como Jest, PyTest, Coverage.py, Istanbul, etc.).
- Cobertura de escenarios de rendimiento: número de usuarios concurrentes, volumen de datos, duración de las pruebas.
- Cobertura de integraciones: módulos o servicios externos validados.

8. Resultados de pruebas. Organiza los resultados según el tipo de prueba:

- Pruebas de caja blanca
    - Resultados de cobertura.
    - Lógica interna validada (bucles, condiciones, manejo de errores).
    - Verificación de invariantes del sistema.
- Pruebas de integración / unitarias profundas.
    - Interacción entre módulos.
    - Comportamiento con mocks/stubs.
    - Verificaciones internas del estado.
- Pruebas de rendimiento / estrés / carga
    - Métricas clave =>
        - Tiempos de respuesta promedio y percentiles (p95, p99).
        - Throughput (peticiones por segundo).
        - Errores bajo carga (5xx, timeouts).
        - Uso de CPU, memoria y red.
        - Gráficos o reportes generados por JMeter, k6, Gatling, etc.
        - Límites alcanzados y puntos de degradación.

9. Análisis de resultados y hallazgos: Aquí se interpretan los datos, no solo se listan:

- Análisis comparativo entre resultados esperados y obtenidos.
- Identificación de cuellos de botella o puntos críticos.
- Evaluación del cumplimiento de métricas de calidad (rendimiento, disponibilidad, fiabilidad).
- Impacto potencial de los defectos encontrados.

10. Conclusiones. Síntesis del estado de calidad del backend:

- Cumplimiento general de los objetivos de prueba.
- Nivel de madurez del sistema (apto para despliegue, requiere correcciones, etc.).
- Fortalezas detectadas (robustez, escalabilidad, consistencia).
- Debilidades o riesgos pendientes.

11. Recomendaciones / Hallazgos / Mitigación. Propuestas de mejora técnica o de proceso:

- Optimización de algoritmos o consultas a base de datos.
- Ajuste de configuración del servidor o caché.
- Refactorización de módulos críticos.
- Ampliación de cobertura de pruebas automáticas.
- Refuerzo de mecanismos de seguridad o resiliencia.

12. Anexos. Material complementario:

- Logs, capturas de pantalla, reportes de herramientas.
- Scripts o configuraciones usadas (JMeter .jmx, k6 .js, etc.).
- Resultados de cobertura generados por herramientas automáticas.
- Evidencias de errores o tickets levantados.