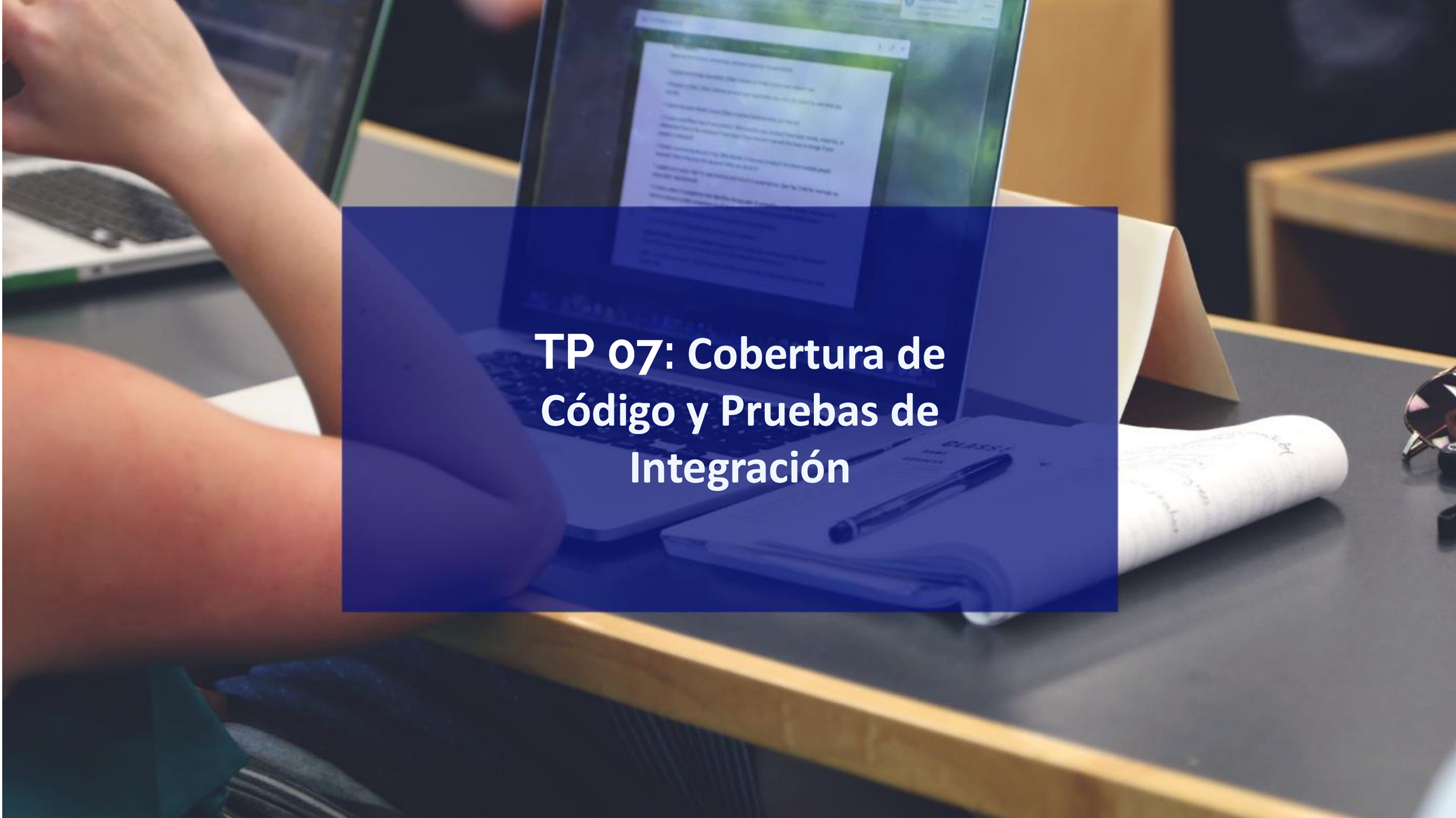


TP 07: Cobertura de Código y Pruebas de Integración

Cobertura de Código y Pruebas de Integración

Ing. Ariel Schwindt

https://www.linkedin.com/in/arielschwindt/

MS Certified DevOps Engineer Expert
MS Certified Azure Developer Associate
MS Certified Azure AI Engineer Associate

Objetivo de la Sesión

- Analizar la cobertura de pruebas (Code Coverage).
- Comprender el análisis estático de código.
- Aplicar pruebas de integración (Integration Tests)
- Integrar análisis estático, cobertura de pruebas y pruebas de integración en un pipeline CI/CD.

Resultados Esperados

Al final de la sesión, los participantes deben ser capaces de:

- **Medir y analizar la cobertura de código** en proyectos de front-end y back-end
- **Configurar y ejecutar análisis estático** con SonarCloud, interpretando los resultados para mejorar la calidad del código.
- **Implementar pruebas de integración** utilizando Cypress para verificar la interacción entre componentes de una aplicación web.
- **Integrar la cobertura de pruebas, análisis estático y pruebas de integración** en un pipeline de CI/CD, asegurando que el software cumple con altos estándares de calidad.
- **Publicar resultados** de pruebas y cobertura en el pipeline, facilitando su interpretación y seguimiento continuo.

Cobertura de Pruebas (Code Coverage)

Permite medir el porcentaje de código que ha sido ejecutado por pruebas unitarias automatizadas. Es una métrica que indica qué parte del código está siendo validada por pruebas, ayudando a detectar áreas no cubiertas.

¿Para qué sirve?

- **Asegura la robustez del software**: Una alta cobertura reduce la probabilidad de errores en producción al garantizar que la mayoría del código haya sido probado.
- **Identifica áreas no probadas**:
- Detecta secciones del código que no han sido alcanzadas por las pruebas, permitiendo enfocar los esfuerzos en esos puntos.
- **Ayuda a mejorar la calidad del software**: Al saber qué partes del código no están probadas, se puede priorizar la creación de pruebas adicionales para mejorar la confiabilidad.

¿Cómo se interpretan?

- **Porcentaje de cobertura:** Un alto porcentaje de cobertura (por ejemplo, 80-90%) indica que una gran parte del código ha sido validada por pruebas. Sin embargo, no garantiza la ausencia de errores, ya que la cobertura no asegura que todas las posibles combinaciones y escenarios han sido probados.
- **Baja cobertura:** Puede indicar que hay partes críticas del código no probadas, lo que representa un riesgo para la estabilidad y el rendimiento del software.

Beneficios

- **Mejor cobertura, menor riesgo:** Una cobertura de código alta disminuye el riesgo de errores no detectados**.**
- **Optimización del esfuerzo de pruebas:** Facilita la priorización de pruebas en áreas críticas.
- **Mantenimiento continuo:** Permite detectar la disminución de la cobertura cuando se añaden nuevas funcionalidades o se modifican partes del código.

Cobertura de Pruebas (Code Coverage)

Estándares de Cobertura en la industria del software

70-80% de cobertura: Esto es considerado un buen nivel en la mayoría de los proyectos. Indica que una gran parte del código está cubierta por pruebas, pero deja margen para ciertas áreas que pueden no necesitar pruebas exhaustivas, como código de infraestructura o utilidades simples.

80-90% de cobertura: Proyectos críticos o con alto requerimiento de calidad suelen buscar una cobertura en este rango.

90-100% de cobertura: Rara vez se alcanza el 100% de cobertura, pero es el objetivo en proyectos de misión crítica (como sistemas financieros o de salud) donde la confiabilidad es esencial.

Factores que influyen en el estándar

Tipo de proyecto:

Proyectos críticos: Sistemas como los de seguridad, salud, finanzas, o infraestructuras críticas exigen altos niveles de cobertura. Esto se debe a que cualquier fallo puede tener consecuencias graves, desde pérdidas económicas hasta riesgos para la seguridad o la vida de las personas. En estos proyectos, se busca una cobertura de pruebas que cubra al menos un 80-90% del código, asegurando que los componentes esenciales están bien probados.

Proyectos no críticos o con bajo riesgo: En proyectos que no manejan información sensible o que no afectan directamente la seguridad, como aplicaciones menos complejas o herramientas de uso interno, puede ser aceptable una cobertura más baja (60-80%). El enfoque aquí es asegurar la funcionalidad principal, pero sin gastar recursos innecesarios en probar cada aspecto del sistema. Ejemplos incluyen sitios web informativos, aplicaciones con un bajo impacto económico o herramientas internas de automatización.

Compromiso entre cobertura y esfuerzo (Trade-off):

Cobertura del 100% no siempre es eficiente: Alcanzar una cobertura total implica probar cada línea de código y cada posible escenario. Esto puede requerir un gran esfuerzo, especialmente en áreas donde los riesgos son bajos, como configuraciones, código que rara vez se ejecuta o validaciones triviales. Por lo tanto, en muchos proyectos, se busca una cobertura que sea alta en las áreas más críticas, pero más relajada en otras.

Optimización del esfuerzo: En lugar de enfocarse en probar el 100% del código, los equipos a menudo priorizan pruebas en las **áreas críticas**, donde los errores pueden tener un impacto mayor. Esto permite que el equipo enfoque sus recursos en las partes del sistema más susceptibles a fallos o más complejas, sin gastar tiempo innecesario en probar detalles con bajo riesgo de error.

Análisis Estático de Código

Permite detectar Errores y Vulnerabilidades en etapas tempranas del proceso de desarrollo

¿Para qué sirve?

- **Prevenir errores antes de la ejecución:** Detecta errores de programación que podrían causar fallos en la ejecución del sistema, como referencias nulas, variables sin inicializar o bucles infinitos.
- **Identificar vulnerabilidades de seguridad**: Ayuda a descubrir riesgos como inyecciones SQL, Cross-Site Scripting (XSS) o fugas de información sensible, que podrían ser explotados por atacantes.
- **Mejorar la calidad y mantenibilidad del código:** Encuentra malas prácticas como duplicación de código, métodos o funciones excesivamente largas, o el uso innecesario de recursos, lo que dificulta el mantenimiento y la comprensión del sistema.
- **Cumplimiento de estándares:** Garantiza que el código cumpla con normativas y estándares de calidad establecidos, como OWASP (seguridad web) o reglas internas del equipo de desarrollo.

¿Cómo se interpreta?

- **Métricas clave:** Herramientas como SonarCloud generan informes con métricas como la deuda técnica (tiempo necesario para corregir problemas), bugs detectados, vulnerabilidades de seguridad, code smells (malas prácticas), duplicación de código, y cobertura de pruebas.
- **Clasificación de problemas:** Los errores se categorizan por severidad (críticos, mayores, menores), lo que permite priorizar las correcciones más importantes.
- **Toma de decisiones:** Los equipos pueden usar los resultados del análisis para decidir qué áreas necesitan refactorización, qué partes del código presentan riesgos de seguridad, o qué secciones requieren más pruebas.

Beneficios

- **Detección temprana de problemas**: Permite identificar errores y vulnerabilidades en las primeras fases del desarrollo, reduciendo costos y esfuerzo en fases posteriores del ciclo de vida del software.
- **Aumento de la seguridad:** Al detectar vulnerabilidades en el código antes de que sean explotadas, el análisis estático ayuda a proteger el software contra ataques.
- **Mejora continua**: Proporciona informes y métricas que permiten rastrear la evolución de la calidad del código a lo largo del tiempo, fomentando una cultura de mejora continua.
- **Automatización en pipelines CI/CD:** Integrar el análisis estático en pipelines de integración continua asegura que cada nueva versión del código se revise automáticamente, manteniendo un estándar de calidad constante.

Pruebas de Integración (Integration Tests)

Verifican si los diferentes componentes de una aplicación funcionan correctamente al interactuar entre sí.

¿Para qué sirve?

- **Validar la interacción entre módulos:** Aseguran que los componentes del sistema, como el front-end y el back-end o diferentes microservicios, se comuniquen correctamente y produzcan el comportamiento esperado.
- **Simular escenarios reales:** Reproducen flujos de trabajo del usuario o procesos completos, probando cómo los módulos interactúan en situaciones que reflejan el uso real del sistema.
- **Detectar errores en la integración:** Identifican problemas como incompatibilidades en la comunicación, errores en el manejo de datos, o malinterpretaciones de contratos entre servicios (como API).
- **Asegurar la estabilidad del sistema:** Verifican que las dependencias entre componentes se mantengan estables, evitando que cambios en un módulo afecten negativamente a otros.

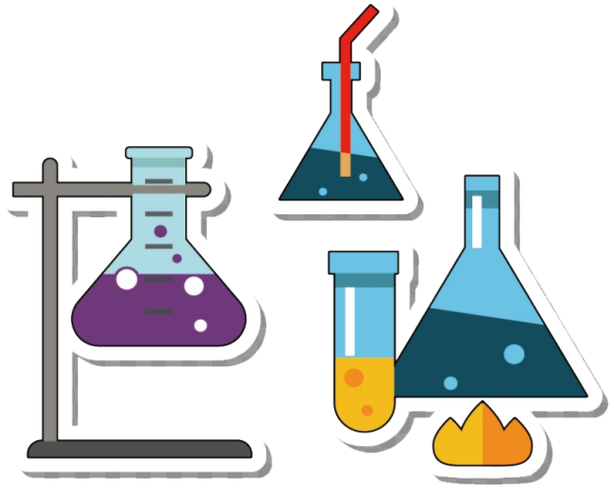
¿Cómo se interpretan?

- **Éxito o fallo de los flujos de trabajo:** Un resultado exitoso indica que los componentes se integraron correctamente y que el sistema produjo el comportamiento esperado bajo condiciones reales. Un fallo, en cambio, sugiere un problema en la interacción entre módulos, como una mala configuración o error en el intercambio de datos.
- **Rendimiento de la interacción:** Además de verificar si los componentes se comunican correctamente, también se puede evaluar el rendimiento de estas interacciones, como la latencia en las solicitudes de red entre módulos.
- **Cobertura de casos de uso:** Es importante analizar si las pruebas de integración cubren todos los casos de uso críticos, especialmente aquellos que involucran varias partes del sistema que deben interactuar.

Beneficios

- **Detección temprana de problemas en la integración:** Al probar cómo interactúan los módulos, se pueden identificar problemas antes de que lleguen a producción, reduciendo la probabilidad de fallos críticos.
- **Prevención de errores en producción:** Garantizan que el sistema funcione de manera coherente en entornos de producción, ya que simulan escenarios reales con múltiples componentes trabajando juntos.
- **Aumenta la confianza en el sistema:** Al cubrir interacciones clave, las pruebas de integración aseguran que las piezas principales del sistema están bien coordinadas, lo que permite realizar cambios y adiciones de nuevas funcionalidades sin romper la funcionalidad existente.
- **Ahorro de tiempo y costos:** Detectar y corregir errores de integración temprano en el ciclo de vida del desarrollo ahorra tiempo y esfuerzo en comparación con la corrección de errores en producción.

Presentación del TP



Consignas, Desarrollo y Desafíos del Trabajo Práctico

Integrar en nuestro pipeline características de Code Coverage, Análisis Estático con SonarCloud, Pruebas de Integración

1) Agregar Code Coverage a nuestras pruebas unitarias de backend y front-end e integrarlas junto con sus resultados en nuestro pipeline de build.

- **Objetivo**: Aprender a configurar code coverage en nuestros proyectos de back y front.
- **Desarrollo**:
  - Agregar paquetes necesarios en ambos proyectos
  - Configurar karma para incluir reportes de cobertura
  - Modificar pipeline de build para que se genere y publique el resultado de la cobertura de código
- **Resultado esperado**: Un pipeline que al ejecutarse nos indique el % de code coverage en la interfaz de Azure Devops

2) Agregar Análisis Estático de Código con SonarCloud

- **Objetivo**: Aprender a configurar la integración de Pipelines con herramientas externas de análisis de código
- **Desarrollo**:
  - Crear cuenta en SonarCloud
  - Modificar nuestro pipeline para SonarCloud pueda realizar el análisis estático
  - Analizar la información generada en el proyecto de SonarCloud
- **Resultado esperado**: Un pipeline que al ejecutarse ejecute el análisis estático en SonarCloud en el back **y en el front** y detallar en la entrega del TP los puntos más relevantes del informe de SonarCloud, qué significan y para qué sirven.

Consignas, Desarrollo y Desafíos del Trabajo Práctico

Integrar en nuestro pipeline características de Code Coverage, Análisis Estático con SonarCloud, Pruebas de Integración

3) Implementar Pruebas de Integración

- **Objetivo**: Aprender a escribir pruebas de integración e integrarlas en nuestro pipeline de deploy

- **Desarrollo**:
  - Agregar paquetes necesarios en proyecto de front-end
  - Inicializar Cypress y crear el primer test de manera manual
  - Analizar resultados de éxito y fallas
  - Crear test grabando la interacción del usuario con nuestro front.
  - Modificar pipeline de build para que se genere y publique el resultado de la cobertura de código

- **Resultado esperado**: **Un pipeline que al ejecutarse implemente nuestras pruebas de Cypress incluyendo las pruebas unitarias del front del TP06**

Consignas, Desarrollo y Desafíos del Trabajo Práctico

Integrar en nuestro pipeline características de Code Coverage, Análisis Estático con SonarCloud, Pruebas de Integración

4) Incorporar al pipeline de Deploy la ejecución de las pruebas de integración y la visualización de sus resultados.

- **Objetivo**: Aprender a configurar un pipeline completo de build y deploy con pruebas unitarias, code coverage, análisis estático de código, pruebas de integración y reportes de todas ellas.
- **Resultados esperados**:
  - **Un Pipeline en YAML que incluya a) Build de QA y Front con ejecución y resultado de pruebas de code coverage, pruebas unitarias y análisis de Sonar Cloud y b) Deploy a WebApp(s) de QA y Front que incluya ejecución y resultado de pruebas de integración.**
  - **El pipeline debe contar con dos Stages: Una para Build, Test Unitarios, Code Coverage y SonarCloud y otra para el Deploy a QA con Tests de Integración.**
  - **En la pestaña Test, poder visualizar los Test Unitarios de Front y Back y los Test de Integracion:**
  - **En la pestaña Code Coverage, visualizar la cobertura de las pruebas unitarias de Back y de Front:**
  - **En la pestaña Extensions, ver el análisis de SonarCloud en verde**
  - **Un documento de una carilla explicando qué información pudieron sacar del análisis de Sonar Cloud y de las pruebas de cobertura.**

Espacio para preguntas, dudas y consultas