TÉCNICAS PARA EL ASEGURAMIENTO DE LA CALIDAD

Aseguramiento de la calidad

- El aseguramiento de la calidad son todas aquellas actividades y los procesos que se realizan para asegurar que los productos y servicios de un proyecto posean el nivel de calidad requerido, está orientado al proceso y se centra en el desarrollo del producto o servicio.
- El aseguramiento de la calidad del software (SQA) es un conjunto de prácticas, metodologías y actividades adoptadas en el proceso de desarrollo con el objetivo de garantizar que el software producido cumpla con altos estándares de calidad. El QA en el contexto del desarrollo de software es esencial para reducir defectos, mejorar la confiabilidad del software, garantizar la seguridad y satisfacer al cliente. Sus principales características son:
  - Preventivo: es un enfoque preventivo para la gestión de la calidad. Se centra en la definición de procesos, estándares y procedimientos para prevenir defectos y problemas de calidad desde el inicio.
  - Actividades de planificación: implica la elaboración de planes de calidad que establecen objetivos, estrategias, recursos y procesos para el control de calidad durante todo el ciclo de vida del proyecto.
  - Orientado a procesos: el enfoque principal está en la calidad de los procesos utilizados para desarrollar el software, se centra en asegurar que los procesos estén bien definidos, bien gestionados y que se sigan.
  - Prevención de defectos: el objetivo principal es prevenir defectos antes de que se produzcan y mejorar constantemente los procesos para minimizar la probabilidad de errores en el software.
  - Involucración continua: está involucrado en todo el ciclo de vida del proyecto, desde la planificación hasta la entrega, para garantizar que se cumplan los estándares de calidad en cada fase.

Diseñar para probar

- Si bien la fase de pruebas es la última del ciclo de vida, las actividades del proceso de pruebas deben ser incorporadas desde la fase de especificación y tenerlas en cuenta en las etapas de análisis y diseño con facilitadores de las pruebas.
- Fundamentos de diseño
  - Programar para las interfaces, no para la herencia.
  - Favorecer la composición antes que la herencia.
- Patrones:
  - Delegación
  - Doble herencia
  - Inversión de Control e Inyección de Dependencias
  - Segregación IU/Código: Modelo Vista Controlador (MVC), Model View ViewModel (MVVM)
- Metodologías (Test-first development):
  - Desarrollo Guiado por Pruebas (TDD)
  - Desarrollo Dirigido por Comportamiento (BDD)
  - Desarrollo Dirigido por Tests de Aceptación (ATDD)

Simulación de objetos

- Las dependencias introducen ruido en las pruebas: ¿el fallo se está produciendo en el objeto de la prueba o en una de sus dependencias?
- Las dependencias con sistemas externos afectan a la complejidad de la estrategia de pruebas, ya que es necesario contar con sustitutos de estos servicios externos durante el desarrollo. Ejemplos típicos de estas dependencias son Servicios Web, Sistemas de envío de correo, Fuentes de Datos o simplemente dispositivos hardware.
- Estos sustitutos, muchas veces son exactamente iguales que el servicio original, pero en otro entorno o son simuladores que exponen el mismo interfaz pero realmente no realizan las mismas tareas que el sistema real, o las realizan contra un entorno controlado.
- Para poder emplear la técnica de simulación de objetos se debe diseñar el código a probar de forma que sea posible trabajar con los objetos reales o con los objetos simulados:
  - Doble herencia
  - IoC: Inversión de Control (Inversion Of Control) o
  - DI: Inyección de Dependencias (Dependency Injection)
  - Objetos Mock

Dobles de prueba

- La regla de oro de las pruebas unitarias, es que una unidad (unit) tiene que ser probada sin utilizar ninguna de sus dependencias.
- Siguiendo la misma regla de oro, las pruebas de integración y sistema deben estar aisladas de sus dependencias salvo cuando se estén probando dichas dependencias. Así mismo, el resultado de las pruebas debe ser previsible.
- Entre las ventajas de esta aproximación se encuentran:
  - Devuelven resultados determinísticos
  - Permiten crear o reproducir determinados estados (por ejemplo errores de conexión)
  - Obtienen resultados mucho mas rápidamente y a menor coste, incluso offline.
  - Permiten el inicio temprano de las pruebas incluso cuando las dependencias todavía no están disponibles.
  - Permiten incluir atributos o métodos exclusivamente para el testeo.

Dobles de prueba

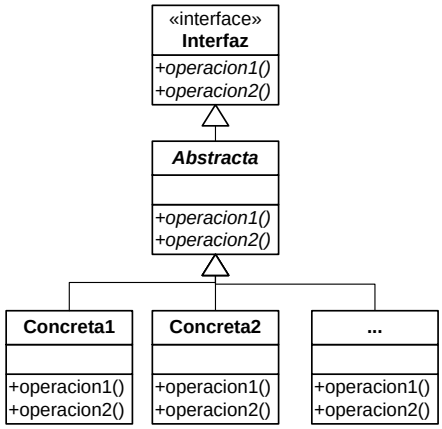
- **Fixture**: Es el término se utiliza para hablar de los datos de contexto de las pruebas, aquellos que se necesitan para construir el escenario que requiere la prueba.
- **Dummy**: Objeto que se pasa como argumento pero nunca se usa realmente. Normalmente, los objetos dummy se usan sólo para rellenar listas de parámetros.
- **Fake**: Objeto que tiene una implementación que realmente funciona pero, por lo general, usa una simplificación que le hace inapropiado para producción (como una base de datos en memoria por ejemplo).
- **Stub**: Objeto que proporciona respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. Los stubs pueden también grabar información sobre las llamadas (**spy**).
- **Mock**: Objeto pre programado con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. Son más complejos que los stubs aunque sus diferencias son sutiles.

Programar con interfaces

- La herencia de clase define la implementación de una clase a partir de otra (excepto métodos abstractos)
- La implementación de interfaz define como se llamara el método o propiedad, pudiendo escribir distinto código en clases no relacionadas.
- Reutilizar la implementación de la clase base es la mitad de la historia.
- Ventajas:
  - Reducción de dependencias.
  - El cliente desconoce la implementación.
  - La vinculación se realiza en tiempo de ejecución.
  - Da consistencia (contrato).
- Desventaja:
  - Indireccionamiento.

Doble Herencia

- Problema:
  - Organizar clases que tienen un comportamiento parecido para que sean intercambiables y consistentes (polimorfismo).
  - Mantener las clases que implementan el interfaz como internas del proyecto (internal o Friend), pero la interfaz pública.
- Solución:
  - Crear un interfaz paralelo a la clase base
  - Clase base es abstracta y sus herederos implementaciones concretas.
  - La clase base puede heredar de mas de una interfaz.
  - Cualquier clase puede implementar el interfaz independientemente de su jerarquía.



Construcción vs Uso

- Los sistemas de software deben separar el proceso de inicio, en el que se instancian los objetos de la aplicación y se conectan las dependencias, de la lógica de ejecución que utilizan las instancias. La separación de conceptos es una de las técnicas de diseño más antiguas e importantes.
- El mecanismo mas potente es la Inyección de Dependencias, la aplicación de la Inversión de Control a la gestión de dependencias. Delega la instanciación en un mecanismo alternativo, que permite la personalización, responsable de devolver instancias plenamente formadas con todas las dependencias establecidas. Permite la creación de instancias bajo demanda de forma transparente al consumidor.
- En lugar de crear una dependencia en sí misma, una parte de la aplicación simplemente declara la dependencia. La tediosa tarea de crear y proporcionar la dependencia se delega a un inyector que se encuentra en la parte superior.
- Esta división del trabajo desacopla una parte de la aplicación de sus dependencias: una parte que no necesita saber cómo configurar una dependencia, y mucho menos las dependencias de la dependencia, etc.

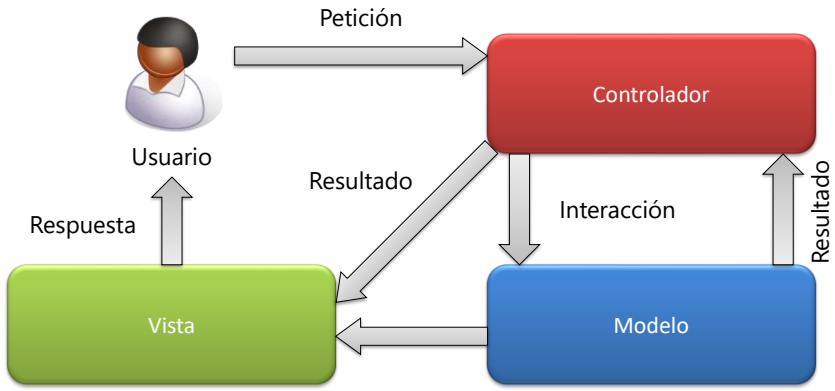
Inversión de Control

- Inversión de control (Inversion of Control en inglés, IoC) es un concepto junto con unas técnicas de programación:
  - en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales,
  - en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones.
- Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.
- En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir. Técnicas de implementación:
  - Service Locator: es un componente (contenedor) que contiene referencias a los servicios y encapsula la lógica que los localiza dichos servicios.
  - Inyección de dependencias.

Inyección de Dependencias

- Las dependencias son expresadas en términos de interfaces en lugar de clases concretas y se resuelven dinámicamente en tiempo de ejecución.
- La Inyección de Dependencias (en inglés Dependency Injection, DI) es un patrón de arquitectura orientado a objetos, en el que se inyectan objetos a una clase en lugar de ser la propia clase quien cree el objeto, básicamente recomienda que las dependencias de una clase no sean creadas desde el propio objeto, sino que sean configuradas desde fuera de la clase. La inyección de dependencias (DI) procede del patrón de diseño más general que es la Inversión de Control (IoC).
- Al aplicar este patrón se consigue que las clases sean independientes unas de otras e incrementando la reutilización y la extensibilidad de la aplicación, además de facilitar las pruebas unitarias de las mismas.
- Desde el punto de vista de Java o .NET, un diseño basado en DI puede implementarse mediante el lenguaje estándar, dado que una clase puede leer las dependencias de otra clase por medio del Reflection y crear una instancia de dicha clase inyectándole sus dependencias.

El patrón MVC



El patrón MVC

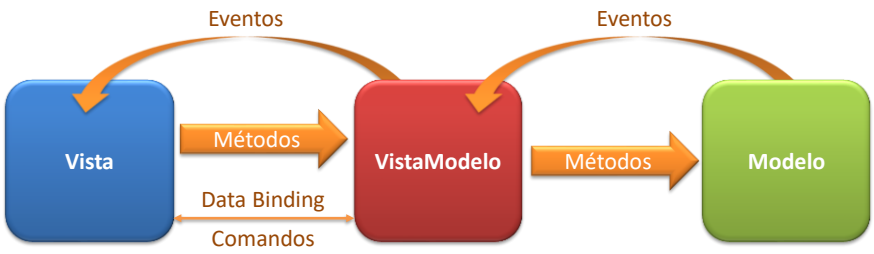
- Representación de los **datos del dominio**
- Lógica de **negocio**
- Mecanismos de **persistencia**

- **Interfaz** de usuario
- Incluye elementos de **interacción**

- **Intermediario** entre Modelo y Vista
- **Mapea acciones** de usuario → acciones del Modelo
- **Selecciona** las vistas y les **suministra** información

Model View ViewModel (MVVM)

- El **Modelo** es la entidad que representa el concepto de negocio.
- La **Vista** es la representación gráfica del control o un conjunto de controles que muestran el Modelo de datos en pantalla.
- La **VistaModelo** es la que une todo. Contiene la lógica del interfaz de usuario, los comandos, los eventos y una referencia al Modelo.



Test-First Development (TFD)

- La prueba gana peso hasta convertirse en "ciudadano de primera clase", es un enfoque que consiste empezar por la prueba para una característica determinada, para luego diseñarla y desarrollarla en un solo paso.
- Al escribir la prueba primero, estas pueden centrarse en los resultados esperados en lugar de en los detalles de implementación, facilitan el diseño de una solución clara y eficaz y, una vez que se completa el desarrollo, se ejecutan para ver si pasan o no.
- Como evolución del Test-first development (XP) las principales metodologías son:
  - Desarrollo Guiado por Pruebas (TDD)
  - Desarrollo Dirigido por Comportamiento (BDD)
  - Desarrollo Dirigido por Tests de Aceptación (ATDD)

Desarrollo Guiado por Pruebas (TDD)

- El Desarrollo Guiado por Pruebas, es una técnica de programación (definida por KentBeck); consistente en desarrollar primero el código que pruebe una característica o funcionalidad deseada antes que el código que implementa dicha funcionalidad y refactorizar después de implementar dicha funcionalidad.
- El objetivo a lograr es que no exista ninguna funcionalidad que no esté avalada por una prueba.
- Lo primero que hay que aprender de TDD son sus 3 leyes:
  - No escribirás código de producción sin antes escribir un test que falle.
  - No escribirás más de un test unitario suficiente para fallar (y no compilar es fallar).
  - No escribirás más código del necesario para hacer pasar el test.

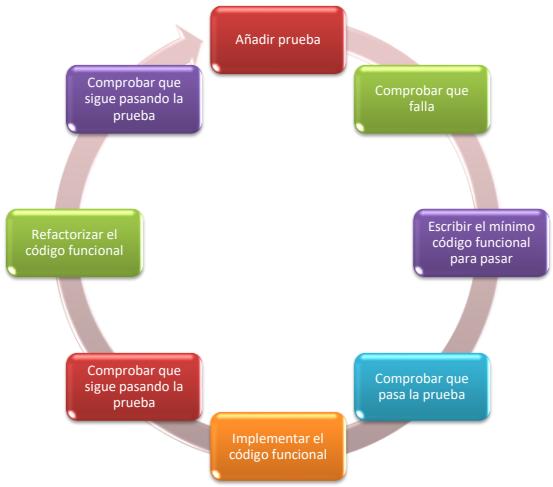
Escribir la prueba

- La escritura de las pruebas sigue varios patrones identificados y detallados por Beck. En particular se destaca que las pruebas:
  - Son escritas por el propio desarrollador en el mismo lenguaje de programación que la aplicación, y su ejecución se automatiza. Esto último es primordial para que obtenga un retorno inmediato, indicándose que el ritmo de cambio entre prueba-código-prueba esta pautado por intervalos de no más de diez minutos. La automatización también permite aplicar, en todo momento, la batería de pruebas, implicando pruebas de regresión inmediatas y continuas.
  - Las pruebas se deben escribir pensando en primer lugar en probar las operaciones que se deben implementar.
  - Deben ser aisladas, de forma que puedan correrse independientemente de otras, no importando el orden. Este aislamiento determinará que las soluciones de código cohesivas y bajo acoplamiento, con componentes ortogonales.
  - Deben de escribirse antes que el código que se desea probar.

Ritmo TDD

- TDD invita a seguir una serie de tareas ordenadas, que a menudo se denomina ritmo TDD, y que se basa en los siguientes pasos:
  1. Escoger un requisito.
  2. Escribir una prueba que demuestre la necesidad de escribir código (assert first).
  3. Escribir el mínimo código para que el código de pruebas compile (probar la prueba)
  4. Fijar la prueba
  5. Implementar exclusivamente la funcionalidad demandada por las pruebas
  6. Ejecutar todas las prueba.
  7. Mejorar el código (refactorización) sin añadir funcionalidad
  8. Actualizar la lista de requisitos (añadir los nuevos requisitos que hayan aparecido)
  9. Volver al primer paso
- Este ritmo permite formalizar las tareas que se han de realizar para conseguir un código fácil de mantener, bien diseñado y probado automáticamente.

Ritmo TDD



Refactorizar el código en pruebas

- Una refactorización es un cambio que está pensado para que el código se ejecute mejor o para que sea más fácil de comprender.
- No está pensado para alterar el comportamiento del código y, por tanto, no se cambian las pruebas.
- Se recomienda realizar los pasos de refactorización independientemente de los pasos que amplían la funcionalidad.
- Mantener las pruebas sin cambios aporta la confianza de no haber introducido errores accidentalmente durante la refactorización.

Estrategia RED – GREEN – REFACTOR

- Se recomienda una estrategia de test unitarios conocida como RED (fallo) – GREEN (éxito), es especialmente útil en equipos de desarrollo ágil.
- Una vez que entendamos la lógica y la intención de un test unitario, hay que seguir estos pasos:
  - Escribe el código del test (**Stub**) para que compile (pase de **RED** a **GEEN)**
    - Inicialmente la compilación fallará **RED** debido a que falta código
    - Implementa sólo el código necesario para que compile **GREEN** (aún no hay implementación real).
  - Escribe el código fuente para que se **ejecute** (pase de **RED** a **GREEN)**
    - Inicialmente el test fallará **RED** ya que no existe funcionalidad.
    - Implementa la funcionalidad que va a probar el test hasta que se ejecute adecuadamente **GREEN.**
  - **Refactoriza** el test y el código una vez que este todo **GREEN** y la solución vaya evolucionando.

Beneficios de TDD

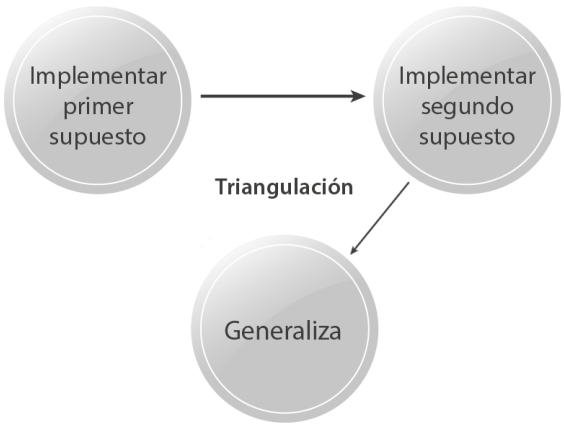
- Facilita desarrollar ciñéndose a los requisitos.
  - Ayuda a encontrar inconsistencias en los requisitos
  - Reduce el número de errores y bugs ya que éstos se detectan antes incluso de crearlos.
- Facilita el mantenimiento del código:
  - Protege ante cambios, los errores que surgen al aplicar un cambio se detectan (y corrigen) antes de subir ese cambio.
  - Protegen ante errores de regresión (rollbacks a versiones anteriores).
  - Dan confianza.
- Las pruebas facilitan entender el código y que, eligiendo una buena nomenclatura, sirven de documentación.
- Ayudan a especificar comportamientos
- Ayudan a refactorizar para mejorar la calidad del código (Clean code)
- A medio/largo plazo aumenta (y mucho) la productividad.

Inside-Out vs Outside-In

- ¿La mejor manera de empezar es por los detalles de lo que está construyendo, y dejar que la arquitectura vaya emergiendo con un enfoque de adentro hacia afuera (Inside Out)? ¿O bien, empezar por algo grande y dejar que los detalles vayan revelando a medida que se van usando desde afuera hacia adentro (Outside In)?.
- La escuela clásica (Inside Out/Bottom Up/Chicago School/Aproximación Clásica) toma su nombre por representar el concepto original definido en libros como «Test-driven Development By Example» de Kent Beck, y que se distingue por enfatizar la práctica de la triangulación (Inside Out).
- La escuela de Londres (Outside In TDD/Top Down/Mockist Approach) toma su nombre debido a que tiene su base, principalmente, en el libro «Growing Object Oriented Software Guide By Test» de Stephen Freeman y Nat Pryce, enfatizando los roles, responsabilidades e interacciones (Outside In).

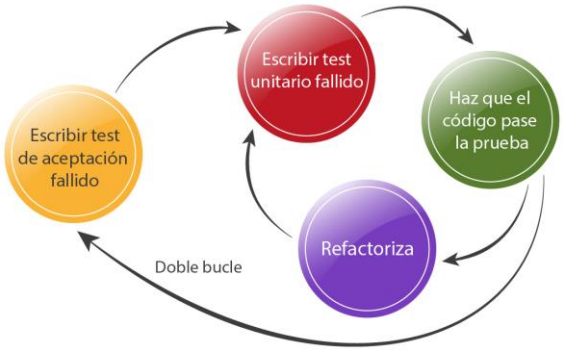
Técnica de la triangulación

- Una vez que tenemos el test fallando, tenemos tres estrategias posibles para pasar a verde:
  - Fake It: crear un fake que devuelva una constante.
  - Obvious Implementation: implementamos la solución más simple posible si es obvia.
  - Triangulation
- Triangular, o la técnica de la triangulación, es el paso natural que sigue a la técnica de la implementación falsa:
  - Escoger el caso más simple que debe resolver el algoritmo.
  - Aplicar Red-Green-Refactor.
  - Repetir los pasos anteriores cubriendo las diferentes casuísticas.
- El concepto se basa en obtener dos o más casos similares antes de proceder con la implementación de una solución genérica.



Técnica del doble bucle

- La escuela de Londres toma un enfoque distinto, centrándose en verificar que el comportamiento de los objetos es el esperado. Dado que este es el objetivo final, verificar las correctas interacciones entre objetos, y no el estado en sí mismo de los objetos, ahorrándonos todo el trabajo con los objetos reales (creación y mantenimiento) sustituyéndolos por dobles de prueba. Podremos empezar por la funcionalidad final que se necesita, implementando poco a poco toda la estructura que dé soporte a dicha funcionalidad, Outside-in, desde el exterior hacia dentro.
- Mediante el uso de la técnica del doble bucle:
  - Comenzaremos por un test de aceptación que falle, lo que nos guiará al bucle interno.
  - En el bucle interno, que representa la metodología de trabajo TDD («Red-Green-Refactor»), implementaremos la lógica de nuestra solución.
  - Realizaremos las iteraciones necesarias de este bucle interno hasta conseguir pasar el test de aceptación.



Aproximaciones

Escuela Clásica

La aproximación de TDD De adentro hacia afuera (Inside Out) favorece a los desarrolladores a los que les gusta ir construyendo pieza a pieza. Puede ser más sencillo para empezar, siguiendo un enfoque metódico para identificar las entidades, y trabajando duro para llevar a cabo el comportamiento interno.

- Cuando se quiera verificar el estado de los objetos.
- Cuando las colaboraciones entre objetos son sencillas.
- Si no se quiere acoplar la implementación a las pruebas.
- Si se prefiere no pensar en la implementación mientras se escriben las pruebas.

Escuela de Londres

La aproximación de TDD De fuera hacia Adentro (Outside In) el desarrollador empieza a construir todo el sistema, y lo descompone en componentes más pequeños cuando se presentan oportunidades de refactorización. Esta ruta es más exploratorio, y es ideal en las situaciones donde hay una idea general del objetivo, pero los detalles finales de la implementación están menos claros.

- Cuando se quiera verificar el comportamiento de los objetos.
- Cuando las colaboraciones entre objetos sean complejas.
- Si no importa acoplar la implementación a las pruebas.
- Si se prefiere pensar en la implementación mientras se escriben las pruebas.

Desarrollo Dirigido por Comportamiento (BDD)

- El Desarrollo Dirigido por Comportamiento (Behaviour Driver Development) es una evolución de TDD (Test Driven Development o Desarrollo Dirigido por Pruebas), el concepto de BDD fue inicialmente introducido por Dan North como respuesta a los problemas que surgían al enseñar TDD.
- En BDD también vamos a escribir las pruebas antes de escribir el código fuente, pero en lugar de pruebas unitarias, lo que haremos será escribir pruebas que verifiquen que el comportamiento del código es correcto desde el punto de vista de negocio. Tras escribir las pruebas escribimos el código fuente de la funcionalidad que haga que estas pruebas pasen correctamente. Después refactorizamos el código fuente.
- Partiremos de historias de usuario, siguiendo el modelo “Como [rol] quiero [característica] para [los beneficios]”. A partir de aquí, en lugar de describir en 'lenguaje natural' lo que tiene que hacer esa nueva funcionalidad, vamos a usar un lenguaje ubicuo (un lenguaje semiformal que es compartido tanto por desarrolladores como personal no técnico) que nos va a permitir describir todas nuestras funcionalidades de una única forma.

BDD

- Para empezar a hacer BDD sólo nos hace falta conocer 5 palabras, con las que construiremos sentencias con las que vamos a describir las funcionalidades:
  - Feature (característica): Indica el nombre de la funcionalidad que vamos a probar. Debe ser un título claro y explícito. Incluimos aquí una descripción en forma de historia de usuario: “Como [rol] quiero [característica] para [los beneficios]”. Sobre esta descripción empezaremos a construir nuestros escenarios de prueba.
  - Scenario: Describe cada escenario que vamos a probar.
  - Given (dado): Provee el contexto para el escenario en que se va a ejecutar el test, tales como el punto donde se ejecuta el test, o prerequisitos en los datos. Incluye los pasos necesarios para poner al sistema en el estado que se desea probar.
  - When (cuando): Especifica el conjunto de acciones que lanzan el test. La interacción del usuario que acciona la funcionalidad que deseamos testear.
  - Then (entonces): Especifica el resultado esperado en el test. Observamos los cambios en el sistema y vemos si son los deseados.

Ejemplo

```
# language: es
Característica: Suma de dos números
  Como matemático novato
  Yo quiero obtener la suma de dos cifras
  Para aprender a sumar

  Escenario: Sumar dos números positivos
  Dado que estoy en la aplicación
  Cuando pongo los números 1 y 3
  Y solicito el resultado del cálculo
  Entonces el resultado debe ser 4

  Escenario: Sumar dos números negativos
  Dado que estoy en la aplicación
  Cuando pongo los números -1 y -3
  Y solicito el resultado del cálculo
  Entonces el resultado debe ser -4

  Escenario: Sumar un numero positivo y uno negativo
  Dado que estoy en la aplicación
  Cuando pongo los números -2 y 3
  Y solicito el resultado del cálculo
  Entonces el resultado debe ser 1
```

```
public class CalculadoraStepDefinitions {
    Calculadora calc;
    int op1, op2, rslt;

    @Dado("que estoy en la aplicación")
    public void que_estoy_en_la_aplicación() {
        calc = new Calculadora();
    }
    @Cuando("pongo los números {int} y {int}")
    public void pongo_los_números_y(Integer a, Integer b) {
        op1 = a;
        op2 = b;
    }
    @Cuando("solicito el resultado del cálculo")
    public void solicito_el_resultado_del_cálculo() {
        rslt = calc.suma(op1, op2);
    }
    @Entonces("el resultado debe ser {int}")
    public void el_resultado_debe_ser(Integer r) {
        assertEquals(r, rslt);
    }
}
```


Desarrollo Dirigido por Tests de Aceptación (ATDD)

- El Desarrollo Dirigido por Test de Aceptación (ATDD), técnica conocida también como Story Test-Driven Development (STDD), es una variación del TDD pero a un nivel diferente.
- Las pruebas de aceptación o de cliente son el criterio escrito de que un sistema cumple con el funcionamiento esperado y los requisitos de negocio que el cliente demanda. Son ejemplos escritos por los dueños de producto. Es el punto de partida del desarrollo en cada iteración.
- ATDD/STDD es una forma de afrontar la implementación de una manera totalmente distinta a las metodologías tradicionales. Cambia el punto de partida, la forma de recoger y formalizar las especificaciones, sustituye los requisitos escritos en lenguaje natural (nuestro idioma) por historias de usuario con ejemplos concretos ejecutables de como el usuario utilizara el sistema, que en realidad son casos de prueba. Los ejemplos ejecutables surgen del consenso entre los distintos miembros del equipo y el usuario final.
- La lista de ejemplos (pruebas) de cada historia, se escribe en una reunión que incluye a dueños de producto, usuarios finales, desarrolladores y responsables de calidad. Todo el equipo debe entender qué es lo que hay que hacer y por qué, para concretar el modo en que se certifica que el sotfware lo hace.

ATDD

- El algoritmo o ritmo es el mismo de tres pasos que en el TDD practicado exclusivamente por desarrolladores pero a un nivel superior.
- En ATDD hay dos prácticas claves:
  - Antes de implementar (fundamental lo de antes de implementar) una necesidad, requisito, historia de usuario, etc., los miembros del equipo colaboran para crear escenarios, ejemplos, de cómo se comportará dicha necesidad.
  - Después, el equipo convierte esos escenarios en pruebas de aceptación automatizadas. Estas pruebas de aceptación típicamente se automatizan usando Selenium o similares, “frameworks” como Cucumber, etc.

Data Driven Testing (DDT)

- Se basa en la creación de tests para ejecutarse en simultáneo con sus conjuntos de datos relacionados en un framework. El framework provee una lógica de test reusable para reducir el mantenimiento y mejorar la cobertura de test. La entrada y salida (del criterio de test) pueden ser resguardados en uno o más lugares del almacenamiento central o bases de datos, el formato real y la organización de los datos serán específicos para cada caso.
- Todo lo que tiene potencial de cambiar (también llamado "variabilidad," e incluye elementos como el entorno, puntos de salida, datos de test, ubicaciones, etc) está separado de la lógica del test (scripts) y movido a un 'recurso externo'. Esto puede ser configuración o conjunto de datos de test. La lógica ejecutada en el script está dictada por los valores.
- Los datos incluyen variables usadas tanto para la entrada como la verificación de la salida. En casos avanzados (y maduros) los entornos de automatización pueden ser obtenidos desde algún sistema usando los datos reales o un "sniffer", el framework DDT por lo tanto ejecuta pruebas sobre la base de lo obtenido produciendo una herramienta de test automáticos para regresión.

Keyword Driven Development (KDD)

- Keyword Driven Development o KDD es un proceso de desarrollo software basado en KDT, que se impulsa por la palabra clave almacenada en una base de datos y en la automatización a la que hace referencia.
- Keyword Driven Testing o KDT es un framework de automatización de pruebas, donde se utiliza un formato de tabla para definir palabras claves o palabras de acción para cada función que nos gustaría utilizar. La ISO/IEC/IEEE 29119-5 define KDT como una forma de describir los casos de prueba mediante el uso de un conjunto predefinido de palabras clave, en un lenguaje de uso común. Estas palabras clave son nombres que están asociados con un conjunto de acciones que se requieren para realizar los pasos específico en un caso de prueba. Mediante el uso de palabras clave para describir las acciones de prueba en lugar de lenguaje natural, los casos de prueba pueden ser más fáciles de entender, mantener y automatizar.
- La definición de las keywords se produce de una forma independiente a las acciones asociadas, lo que provoca la separación del diseño de la prueba respecto a su ejecución. La reusabilidad de las keywords constituye otra de las potencialidades.

Componentes de Keyword Driven Testing

- Almacén de palabras clave (Keywords): Las palabras que creamos las guardaremos en un determinado formato para que posteriormente sean accedidas por los scripts.
- Almacén de datos: Contiene los datos necesarios para las pruebas y se almacenan en un determinado formato similar al seleccionado para las palabras clave.
- Repositorio de objetos: Repositorio de las referencias necesarias a los objetos UI sobre los que se actuará. Pueden estar referenciadas por id, XPath, etc. En este punto es importante señalar que durante la creación del frontal de la aplicación es conveniente que los desarrolladores codifiquen acorde a los estándares e identifiquen los objetos UI de forma adecuada.
- Biblioteca de funciones: Contiene la codificación de los flujos de negocio para los distintos elementos UI.
- Scripts de pruebas: Son el conjunto de funciones que se llaman desde la biblioteca de funciones para ejecutar el código asociado a una palabra clave.

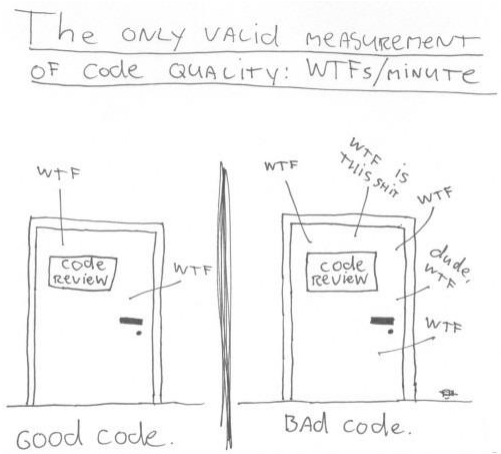
Clean Code y Calidad de software

- Robert C. Martin propone seguir una serie de guías y buenas prácticas a la hora de escribir el código que se enmarcar en la creación de código de calidad.
- En concreto se centra en el más bajo de los niveles: la escritura del código.
- Trata aspectos básicos como el formato, estilo, nomenclaturas, convenciones, comentarios, …
- Así mismo trata aspectos estructurales de como definir funciones y clases correctamente, qué uso hay que dar a los objetos y a las estructuras de datos, la relación entre excepciones y la lógica de negocio, cómo elegir y utilizar el código de terceros.
- Adicionalmente, trata la importancia de las pruebas unitarias y qué reglas deben cumplir, cómo construir sistemas basados en POJO´s y sostenidos por tests, definición de reglas que nos ayudarán en nuestro diseño, observaciones al diseño de sistemas concurrentes.

¿Cómo detectar un código mal oliente? (Code Smells)

- Estos son los síntomas que podemos encontrar en el código fuente de un sistema que indican que muy probablemente existan problemas más profundos de calidad de código, de diseño o de ambos.
  - Rigidez es la tendencia que posee el software a ser difícil de cambiar, incluso en formas sencillas o cambios mínimos.
  - Fragilidad es la tendencia que posee un programa para romperse en muchos lugares cuando un simple cambio es realizado.
  - Inamovilidad es la dificultad de separar el sistema en componentes que pueden ser reutilizados en otros sistemas.
  - Viscosidad se presenta cuando hacer las cosas incorrectamente es más fácil que hacerlas del modo correcto.
  - Complejidad innecesaria es cuando hay muchos elementos que actualmente no son útiles.
  - Ambiente de desarrollo lento e ineficiente.
    - Tiempos muy largos de compilación
    - Subir el código toma horas
    - Hacer el despliegue toma varios minutos
  - Repetición innecesaria es cuando el código posee estructuras repetidas que pueden ser unificadas bajo una sola abstracción.
  - Opacidad es la tendencia que posee un módulo a ser difícil de leer y comprender.

Métrica WTF



WTF: Acrónimo de What The Fuck (Vulgarismo)

WTF

```
static int STRTXG(bool b, int i0, int i1, int i2, int i3, int i4) {
    if (b == true)
        return 0;
    else {
        switch (i0) {
            case -1:
                return STXTGH(i2);
            case 920:
                return SFFMAB(i1, i3);
            case 1222:
                return SFFMAB(i1, i3);
            case 824:
                return SGXECB(i4);
            //TODO: Uncomment Later
            //case 3:
            //    return STXGMB(i1) + STXKSB(i1) + STXNRB(i1);
            default:
                return 42;
        }
    }
    return 0;
}
```

WTF

```
static int STRTXG(bool b, int i0, int i1, int i2, int i3, int i4) {
    if (b == true)
        return 0;
    else {
        switch (i0) {
            case -1:
                return STXTGH(i2);
            case 920:
                return SFFMAB(i1, i3);
            case 1222:
                return SFFMAB(i1, i3);
            case 824:
                return SGXECB(i3);
            //TODO: Uncomment Later
            //case 3:
            //    return STXGMB(i1) + STXKSB(i1) + STXNRB(i1);
            default:
                return 42;
        }
    }
    return 0;
}
```

WCF? WCF WCF WCF? WCF? WCF? WCF? WCF? WCF i4?

WTF

- ¿Qué puerta representa tu código?
- ¿Qué puerta representa tu equipo o tu empresa?
- ¿Por qué estamos en esa habitación?
- ¿Es solo una revisión de código normal o hemos encontrado una serie de problemas horribles poco después del lanzamiento?
- ¿Estamos depurando en pánico, analizando el código que pensamos que funcionaba?
- ¿Los clientes huyen despavoridos y los gerentes nos respiran en el cogote?
- ¿Cómo asegurarnos de terminar detrás de la puerta correcta cuando las cosas se ponen difíciles?
- La respuesta es: la maestría.
- Hay dos factores para conseguir la maestría: conocimiento y trabajo. Debes obtener el conocimiento de los principios, patrones, prácticas y heurísticas que conoce un maestro; también debes machacar ese conocimiento, con tus dedos, ojos e intestino, trabajando duro y practicando.

WTF

Fundamentos

- “Escribir código que entienda la computadora es una técnica; escribir código que entienda un ser humano es un Arte” – Robert Martin
- El campo @author de un Javadoc nos dice quiénes somos. Somos autores. Y una cosa sobre los autores es que tienen lectores. De hecho, los autores son responsables de comunicarse bien con sus lectores. La próxima vez que escriba una línea de código, recuerde que es un autor, escribiendo para lectores que juzgarán su esfuerzo.
- La proporción de tiempo dedicado a leer frente a escribir es bastante superior a 10: 1. Estamos constantemente leyendo el código antiguo como parte del esfuerzo para escribir código nuevo. Debido a que esta relación es tan alta, queremos que la lectura del código sea fácil, incluso si dificulta la escritura. Por supuesto, no hay forma de escribir código sin leerlo, por lo que facilitar la lectura facilita la escritura.
- No hay escapatoria de esta lógica. No puede escribir código si no puede leer el código circundante. El código que intenta escribir hoy será difícil o fácil de escribir, dependiendo de lo difícil o fácil que sea leer el código circundante. Entonces, si quiere ir rápido, si quiere terminar rápidamente, si desea que su código sea fácil de escribir, hágalo fácil de leer.

Fundamentos

- Cada sistema está construido a partir de un lenguaje específico de dominio diseñado por los programadores para describir dicho sistema. Las clases son los sujetos y los métodos los predicados.
- El arte de la programación es, y siempre ha sido, el arte del diseño del lenguaje.
- Los programadores experimentados piensan en los sistemas como historias para ser contadas en lugar de programas para ser escritos. Utilizan las prestaciones del lenguaje de programación elegido para crear un lenguaje mucho más rico y expresivo que pueda usarse para contar dicha historia.
- Parte de ese lenguaje específico de dominio es la jerarquía de funciones que describen todas las acciones que tienen lugar dentro de ese sistema. En un acto artístico de recurrencia, esas acciones se escriben para usar el lenguaje específico del dominio que definen para contar su propia pequeña parte de la historia.

Fundamentos

- Con el tiempo, la automatización de las pruebas, y en especial las unitarias, se han convertido en una parte fundamental y formal del proceso de desarrollo de software.
- Antiguamente, los fragmentos de código se probaban de forma manual e informal. Hoy en día, se escribiría una prueba que se asegurara de que cada rincón y grieta de ese código funcionara como se esperaba. Se aislaría el código de las dependencias y las burlaría para tener un control absoluto sobre el.
- Una vez que pasaran un conjunto de pruebas, nos aseguraríamos de que esas pruebas fueran convenientes para cualquier otra persona que necesitara trabajar con el código. Nos aseguraríamos de que las pruebas y el código se registraran juntos en el mismo paquete fuente.
- Los movimientos Agile y TDD han animado a muchos programadores a escribir pruebas unitarias automatizadas.
- Muchas de las estrategias propuestas en clean code están destinadas a facilitar la creación de las pruebas, sin olvidar que su automatización requiere código que debe cumplir como el resto del código.

Aspectos tratados en Clean Code

- Nombres con sentido: Legible antes que conciso, constantes y expresiones condicionales nominadas
- Funciones: Responsabilidad única: Consultas vs Comandos, Tamaño, Parámetros y argumentos, Regla descendente
- Procesar errores: Tratamiento excepciones, control estricto de nulos
- Comentarios: El único comentario bueno es aquel que encuentras la manera de no escribirlo.
- Formato: Formateo vertical, Formateo horizontal, Conceptos antes que detalles
- Pruebas unitarias
- Clases: Principios S.O.L.I.D. Objetos vs estructuras de datos
- Límites, Sistemas, Emergentes, Concurrencia, Arquitectura, …

La regla KISS

- El principio KISS (del inglés Keep It Simple, Stupid!) «¡Mantenlo sencillo, estúpido!» es un acrónimo usado como principio de diseño.
- El principio KISS establece que la mayoría de sistemas funcionan mejor si se mantienen simples que si se hacen complejos; por ello, la simplicidad debe ser mantenida como un objetivo clave del diseño, y cualquier complejidad innecesaria debe ser evitada.
- Paradójicamente la filosofía KISS no es fácil de lograr porque estamos acostumbrados a creer que cualquier proceso está más trabajado cuanto más complicado parece. Sin embargo, la experiencia nos demuestra que la clave del éxito está en la sencillez y que todo funciona mejor si es simple. ¿Por qué?, porque todo lo que es simple es fácil de cambiar, de recordar, de adaptar y de mantener, y porque la complejidad está ligada al desorden y a las contradicciones lógicas, lo que nos genera incertidumbre y, casi siempre, desinterés y abandono.

La regla DRY

- Las duplicidades son una fuente inagotable de problemas.
- El principio DRY (Don't Repeat Yourself) es una concreción del principio KISS y viene a recordarnos este hecho: No te repitas.
- El código duplicado hace que nuestro código sea más difícil de mantener y comprender además de generar posibles inconsistencias. Hay que evitar el código duplicado siempre.
- Para ello, dependiendo del caso concreto, la refactorización, la abstracción o el uso de patrones de diseño pueden ser nuestros mejores aliados.
- Lo contrario de DRY es WET (we enjoy typing o "nos lo pasamos bien tecleando"), parece que te pagan por palabras: El código es WET cuando contiene duplicaciones innecesarias.

La regla del Boy Scout

- No es suficiente escribir bien el código. El código debe mantenerse limpio con el tiempo. Todos hemos visto que el código se pudre y degrada a medida que pasa el tiempo. Por lo tanto, debemos desempeñar un papel activo en la prevención de esta degradación.
- La regla del Boy Scout dice:
  - «deja el campamento más limpio de como te lo encontraste»
- Ampliándola a otros ámbitos sería algo así como:
  - «deja las cosas mejor de como te las encontraste»
- Sin olvidar:
  - «no hagas lo que no te gusta que te hagan»
- Muchas veces, revisando código, nos encontramos con que el nombre de una variable no es demasiado intuitivo o con un fragmento de código duplicado. Resolviendo este tipo de matices (en vez de mirar hacia otro lado y pasar de largo), estaremos aplicando la regla del Boy Scout.
- Con las pruebas automatizadas adecuadas, es menos probable que la refactorización rompa accidentalmente el código.

Refinamiento sucesivo

- La escritura de software es como cualquier otro tipo de escritura creativa. Nadie suele ser capaz de escribir las versión definitiva a la primera: es habitual que el primer borrador sea desorganizado y torpe por lo que se debe retocar y reestructurar hasta que se lea adecuadamente. En el primer esbozo, centrado en encontrar una solución que funcione, las funciones salen largas y complicadas: tienen mucha sangrías, bucles anidados, largas listas de argumentos, los nombres son arbitrarios, hay código duplicado …
- Mediante un proceso de refinamiento sucesivo, se va mejorando el código dividiendo en funciones, eliminando duplicaciones, cambiando nombres, reordenando …
- Si se dispone del conjunto de pruebas unitarias, este proceso de mejora, también conocido como refactorización, se puede realizar con garantías.
- Con la experiencia, el primer esbozo ira siendo cada vez mas fino, pero no evita el refinamiento, lo aligera.

Refactorización

- La refactorización (del inglés refactoring) es una técnica de ingeniería de software para reescribir un código fuente, alterando su estructura interna sin cambiar su comportamiento externo.
- La refactorización es un proceso sistemático de mejora del código sin crear nuevas funcionalidades. La refactorización transforma el desorden en código limpio y diseño simple.
- Al final de la refactorización, cualquier cambio en el comportamiento es claramente un bug y debe ser arreglado de manera separada a la depuración de la nueva funcionalidad.
- La refactorización siempre tiene el sencillo y claro propósito de mejorar el código. Con un código más efectivo, puede facilitarse la integración de nuevos elementos sin incurrir en errores nuevos.
- Además, cuanto más fácil les resulte a los programadores leer el código, más rápido se familiarizarán con él y podrán identificar y evitar los bugs de forma más eficiente.
- Otro objetivo de la refactorización es mejorar el análisis de errores y la necesidad de mantenimiento del software.

Cuando refactorizar

- Al terminar el primer borrador
  - Cuando escribimos la primera versión estamos mas centrado en encontrar una solución que funcione antes que en emplear tiempo en escribir un código de calidad que puede no ser definitivo. Que salga un código sucio no es un problema, el problema esta en dejarlo así. Una vez encontrada la solución es el momento de refactorizar.
- A la tercera va la vencida
  - Cuando hagas algo por primera vez, simplemente hazlo. Cuando hagas algo similar por segunda vez, no te avergüences de tener que repetirlo, pero haz lo mismo de todos modos. Cuando estés haciendo algo por tercera vez, comienza a refactorizar.
- Al agregar una característica
  - La refactorización te ayuda a comprender el código de otras personas. Si tienes que lidiar con el código sucio de otra persona, intenta refactorizarlo primero. El código limpio es mucho más fácil de entender. Lo mejorarás no sólo para ti sino también para quienes lo utilicen después de ti. La refactorización facilita la adición de nuevas funciones. Es mucho más fácil realizar cambios en un código limpio.
- Al corregir un error
  - Los errores en el código se comportan igual que los de la vida real: viven en los lugares más oscuros y sucios del código. Limpia tu código y los errores prácticamente se descubrirán solos. Los jefes aprecian la refactorización proactiva ya que elimina la necesidad de realizar tareas especiales de refactorización más adelante. ¡Los jefes felices hacen programadores felices!
- Durante una revisión de código
  - La revisión del código puede ser la última oportunidad para ordenarlo antes de que esté disponible para el público. Es mejor realizar este tipo de revisiones junto con un autor. De esta manera, podría solucionar problemas simples rápidamente y calcular el tiempo para solucionar los más difíciles.

Beneficios de la refactorización

- **Reducir la complejidad del código**: A medida que pasa el tiempo (incluso solo horas) mejora nuestra compresión del problema a resolver. A base de pequeñas refactorizaciones tenemos la posibilidad de ir reduciendo la complejidad del código haciéndolo más intuitivo y fácil de entender.
- **Código más fácil de leer**: La refactorización es ideal para mejorar la legibilidad de nuestro código, mejorando los nombres asignados a variables, funciones o clases.
- **Reducir tiempo de solución de errores**: Al mejorar la legibilidad del código y reducir complejidad, es más sencillo encontrar problemas en el código cuando alguien informa de un error.
- **Reducir el tiempo de añadir nuevas funcionalidades**: Al mejorar la legibilidad del código y reducir complejidad es más sencillo añadir código nuevo que se siga la misma dinámica de legibilidad y simplicidad.
- **Reducir la deuda técnica**: Al aplicar refactorizaciones podemos reducir la deuda técnica que nos hayamos ido generando por el camino debido a diferentes motivos como fechas de entrega, solución rápida de bugs, etc..
- **Evitar errores en el futuro**: Cuando identificamos duplicidades en el código y tras evaluar si es necesario, podemos crear mejores abstracciones que lo reducen. De esta manera estamos también evitando posibles errores en el futuro y facilitando la mantenibilidad.
- **Pon el tiempo a correr a tu favor**: El último beneficio, como suma de los anteriores, es que pone el tiempo a correr a tu favor: Si creas el hábito diario de pequeñas refactorizaciones rápidas a medida que haces tus tareas, es cuestión de tiempo que veas los resultados. No necesariamente trabajaras menos, pero si vivirás mucho mejor.

Objetivos de la refactorización

- **Estructuras complicadas o demasiado largas**: cadenas y bloques de comandos tan largos que la lógica interna del software se vuelve incomprensible para lectores externos.
- **Redundancias en el código**: los códigos poco claros suelen contener repeticiones que han de corregirse una a una durante el mantenimiento, por lo que consumen mucho tiempo y recursos.
- **Listas de parámetros demasiado largas**: los objetos no se asignan directamente a un método, sino que se indican sus atributos en una lista de parámetros.
- **Clases con demasiadas funciones**: no cumplen con SOLID clases con demasiadas funciones definidas como método, también llamadas god objects, que hacen que adaptar el software se vuelva casi imposible.
- **Clases anémicas**: clases con tan pocas funciones definidas como método que se vuelven innecesarias.
- **Código demasiado general con casos especiales**: funciones con casos especiales demasiado específicos que apenas se usan y que, por lo tanto, dificultan la incorporación de ampliaciones necesarias.
- **Middle man:** una clase separada actúa como intermediaria entre los métodos y las distintas clases, en lugar de direccionar las solicitudes de los métodos directamente a una clase.

Técnicas

- Hay muchísimas técnicas concretas de refactorización. Para conocerlas todas, se puede consultar la exhaustiva obra sobre este tema de Martin Fowler y Kent Beck: Refactoring: Improving the Design of Existing Code.
- El llamado *desarrollo red-green* (rojo-verde) es un método ágil de desarrollo de software basado en test. Una vez obtenido el código funcional correcto (supera las pruebas, verde), se refactoriza: rojo mientras es incorrecto que pasa a verde cuando vuelve a ser correcto.
- El método *Branching by abstraction* de refactorización aplica cambios graduales a un sistema y va modificando elementos anticuados del código y sustituyéndolos por segmentos nuevos. El branching by abstraction suele utilizarse cuando se realizan cambios grandes que afectan a la jerarquía de clases, a las herencias y a la extracción. Al implementar una abstracción que se mantiene enlazada con una implementación antigua, pueden enlazarse con la abstracción otros métodos y clases. De esta manera, es posible sustituir la funcionalidad del segmento antiguo por la abstracción.
- En refactoring.guru hay un catalogo muy amplio de técnicas de refactorización: componer métodos, mover funcionalidad entre clases, organizar datos, simplificar expresiones condicionales, simplificar las llamadas a métodos y lidiar con la herencia

Componer métodos

- La refactorización de código tiene el objetivo de que los métodos puedan leerse de la manera más fácil posible. En el mejor de los casos, los programadores externos que lean el código deberían poder captar la lógica interna del método. Para combinar métodos de forma eficiente, el refactoring cuenta con diversas técnicas. El objetivo de cada cambio es unificar métodos, eliminar redundancias y dividir métodos largos en segmentos separados que puedan ser modificados fácilmente en el futuro.
- Algunas de estas técnicas son las siguientes:
  - Extraer métodos
  - Convertir métodos a inline
  - Eliminar variables temporales
  - Sustituir variables temporales por métodos de solicitud
  - Introducir variables descriptivas
  - Separar variables temporales
  - Eliminar redireccionamientos a variables de parámetro
  - Sustituir métodos por un objeto método
  - Sustituir el algoritmo

Mover funcionalidad entre clases

- Para mejorar un código, es necesario poder mover atributos o métodos entre clases. Las siguientes técnicas sirven para realizar estos cambios:
  - Mover el método
  - Mover el atributo
  - Extraer la clase
  - Convertir una clase a inline
  - Ocultar el delegado
  - Eliminar una clase en el centro
  - Introducir métodos ajenos
  - Introducir una ampliación local

Organización de los datos

- El objetivo de estas técnicas es clasificar los datos en clases, que deben ser tan pequeñas y fáciles de comprender como sea posible. Deben ser eliminados y divididos en clases lógicas los enlaces innecesarios entre clases que perjudiquen la funcionalidad del software ante pequeños cambios.
- Algunos ejemplos de este tipo de técnicas son los siguientes:
  - Encapsular los propios accesos a atributos
  - Sustituir un atributo propio por una referencia de objeto
  - Sustituir un valor por una referencia
  - Sustituir una referencia por un valor
  - Agrupar datos observables
  - Encapsular atributos
  - Sustituir un conjunto de datos por una clase de datos

Simplificando las expresiones condicionales

- Las fórmulas expresiones deberían simplificarse tanto como sea posible durante la refactorización del código. Para hacerlo, existen varias técnicas:
  - Dividir las condiciones
  - Agrupar expresiones condicionadas
  - Agrupar comandos redundantes en expresiones condicionadas
  - Eliminar elementos de control
  - Reemplazar condicionales anidadas con cláusulas de protección
  - Sustituir distinciones de caso por polimorfismo
  - Introducción de objetos nulos

Simplificando las llamadas a métodos

- Las llamadas o solicitudes a método pueden ejecutarse más fácil y rápidamente al:
  - Cambiar el nombre de los métodos
  - Agregar parámetros
  - Eliminar parámetros
  - Cambiar nombre del método
  - Separar consulta del modificador
  - Parametrizar métodos
  - Agrupar parámetros en objetos
  - Reemplazar parámetro con métodos explícitos
  - Reemplazar parámetro con llamada a método
  - Preservar el objeto completo
  - Eliminar métodos de configuración
  - Ocultar método
  - Reemplazar constructores con métodos de fábrica
  - Reemplazar código de error con excepción
  - Reemplazar excepción con prueba

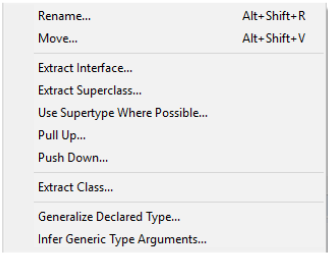
Lidiando con la herencia

- La abstracción tiene su propio grupo de técnicas de refactorización, principalmente asociadas con mover funcionalidad a lo largo de la jerarquía de herencia de clases, crear nuevas clases e interfaces y reemplazar la herencia con delegación y viceversa.
  - Subir un campo, un método o el cuerpo de constructor a la clase base
  - Bajar un campo o método a las clases derivadas
  - Extraer clases derivadas
  - Extraer una clase base
  - Extraer interfaz
  - Contraer jerarquía
  - Generalizar métodos a plantillas de clases
  - Reemplazar herencia con delegación
  - Reemplazar delegación con herencia

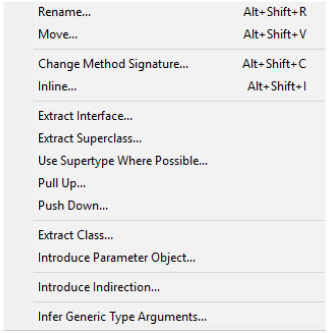
Refactorización en Eclipse IDE

- Eclipse tiene distintos métodos de refactorización. Dependiendo de sobre qué mostremos el menú de refactorización, nos ofrecerá unas u otras opciones. Para refactorizar pulsaremos click derecho sobre el nombre del elemento deseado, y desplegaremos la opción Refactor del menú contextual.

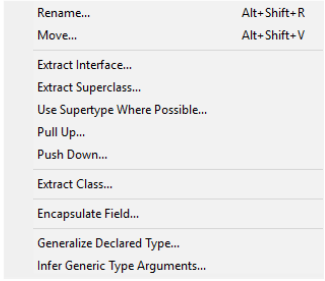
CLASE



MÉTODO



ATRIBUTO



Lista de verificación

- El código debería volverse más limpio.
  - Si el código sigue igual de sucio después de la refactorización... bueno, lo siento, pero acabas de desperdiciar una hora de tu vida. Intenta descubrir por qué sucedió esto. Sucede con frecuencia cuando dejas de refactorizar con pequeños cambios y mezclas un montón de refactorizaciones en un gran cambio. Así que es muy fácil perder la cabeza, especialmente si tienes un límite de tiempo.
  - Pero también puede suceder cuando se trabaja con código extremadamente descuidado: el código en su conjunto sigue siendo un desastre, independientemente de lo que se mejore. Llegado a un punto de no retorno, cuesta menos reescribir completamente partes del código que intentar arreglarlas. Pero antes de eso, deberías haber realizado pruebas automatizadas y reservar una buena cantidad de tiempo. De lo contrario, terminarás con el mismo resultado que en el párrafo anterior.
- No se deben crear nuevas funcionalidades durante la refactorización.
  - No mezcles refactorización y desarrollo directo de nuevas funciones. Una cosa cada vez, intenta separar estos procesos al menos dentro de los límites de las confirmaciones individuales.
- Todas las pruebas existentes deben pasar después de la refactorización.
  - Hay dos situaciones en las que las pruebas pueden fallar después de la refactorización: por lo obvio, cometiste un error durante la refactorización, corrige el error y sigue adelante o abandona dicho camino si empeora las cosas. Otra situación es que las pruebas son de nivel demasiado bajo (por ejemplo, estabas probando métodos privados de clases) y son las pruebas las culpables, se pueden refactorizar las pruebas en sí o escribir un conjunto completamente nuevo de pruebas de nivel superior.

http://junit.org/junit5/

PRUEBAS: JUNIT

Introducción

- JUnit es un entorno de pruebas opensource que nos permiten probar nuestras aplicaciones Java y permite la creación de los componentes de prueba automatiza que implementan uno o varios casos de prueba.
- JUnit 5 requiere Java 8 (o superior) en tiempo de ejecución. Sin embargo, aún puede probar el código que se ha compilado con versiones anteriores del JDK.
- A diferencia de las versiones anteriores de JUnit, JUnit 5 está compuesto por varios módulos diferentes de tres subproyectos diferentes:
  - La plataforma JUnit: sirve como base para lanzar marcos de prueba en la JVM.
  - JUnit Jupiter: es la combinación del nuevo modelo de programación y el modelo de extensión para escribir pruebas y extensiones en JUnit 5.
  - JUnit Vintage: proporciona una función TestEngine para ejecutar pruebas basadas en JUnit 3 y JUnit 4 en la plataforma.

Maven

```
<properties>
        <maven.compiler.release>17</maven.compiler.release>
        <junit.jupiter.version>5.10.1</junit.jupiter.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
        <dependency>
                <groupId>org.junit.jupiter</groupId>
                <artifactId>junit-jupiter</artifactId>
                <version>${junit.jupiter.version}</version>
                <scope>test</scope>
        </dependency>
</dependencies>
```


Casos de pruebas

- Los casos de prueba son clases que disponen de métodos para probar el comportamiento de una clase concreta. Así, para cada clase que quisiéramos probar definiríamos su correspondiente clase de casos de prueba. Los casos de prueba se definen utilizando:
  - Anotaciones: Automatizan el proceso de definición, sondeo y ejecución de las pruebas.
  - Aserciones: Afirmaciones sobre lo que se esperaba y deben cumplirse para dar la prueba como superada. Todas las aserciones del método deben cumplirse para superar la prueba. La primera aserción que no se cumpla detiene la ejecución del método y marca la prueba como fallida.
  - Asunciones: Afirmaciones que deben cumplirse para continuar con el método de prueba, en caso de no cumplirse se salta la ejecución del método y lo marca como tal.

Clases y métodos de prueba

- Clase de prueba: cualquier clase de nivel superior, clase miembro static o clase @Nested que contenga al menos un método de prueba, no deben ser abstractas y deben tener un solo constructor. El anidamiento de clases de pruebas @Nested permiten organizar las pruebas a diferentes niveles y conjuntos.
- Método de prueba: cualquier método de instancia anotado con @Test, @RepeatedTest, @ParameterizedTest, @TestFactory o @TestTemplate.
- Método del ciclo de vida: cualquier método anotado con @BeforeAll, @AfterAll, @BeforeEach o @AfterEach.
- Los métodos de prueba y los métodos del ciclo de vida pueden declararse localmente dentro de la clase de prueba actual, heredarse de superclases o de interfaces, no deben ser privados ni abstractos o devolver un valor.

Agrupar pruebas

- Las pruebas anidadas, clases de prueba anidadas dentro de otra clase de prueba y anotadas con @Nested, permiten expresar la relación entre varios grupos de pruebas.

```
class EntityTest {
        @Test
        void testCanCreate() { }
        @Nested
        class DataRulesTest {
                @Test
                void testProperty1() { }
        }
        @Nested
        class BusinessRulesTest {
                @Nested
                class PersistenceTest {

                }
        }
}
```

Documentar los resultados

- Las pruebas son una parte importante parte de la documentación: los casos de uso. Por defecto, al ejecutar las pruebas, se muestran los nombres de las clases y los métodos de pruebas.
- Se puede personalizar los nombre mostrados mediante la anotación @DisplayName:

```
@DisplayName("A special test case")
class DisplayNameDemo {
    @Test
    @DisplayName("Custom test name")
    void testWithDisplayName() {    }
```

- Se puede anotar la clase con @DisplayNameGeneration para utilizar un generador de nombres personalizados y transformar los nombres de los métodos a mostrar.

```
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
class Display_Name_Demo {
      @Test
      void if_it_is_zero() { }
```


Fixtures

- Es probable que en varias de las pruebas implementadas se utilicen los mismos datos de entrada o de salida esperada, o que se requieran los mismos recursos.
- Para evitar tener código repetido en los diferentes métodos de *test*, podemos utilizar los llamados *fixtures*, que son elementos fijos, escenarios de entrada que permiten prever las salidas, que se crearán antes de ejecutar cada prueba.
- Se pueden crear en métodos de la instancia y ser invocados directamente por cada caso de prueba o delegar su invocación en el ciclo de vida de la pruebas.

Ciclo de vida de instancia de prueba

- Para permitir que los métodos de prueba individuales se ejecuten de forma aislada y evitar efectos secundarios inesperados debido al estado de instancia de prueba mutable, JUnit crea una nueva instancia de cada clase de prueba antes de ejecutar cada método de prueba.
- Este ciclo de vida de instancia de prueba "por método" es el comportamiento predeterminado en JUnit Jupiter y es análogo a todas las versiones anteriores de JUnit.
- Los métodos anotados con @BeforeEach y @AfterEach se ejecutan antes y después de cada método de prueba.
- Anotando la clase de prueba con @TestInstance(Lifecycle.PER_CLASS) se pueden ejecutar todos los métodos de prueba en la misma instancia de prueba.
- Si los métodos de prueba dependen del estado almacenado en atributos de instancia, es posible que se deba restablecer ese estado en métodos @BeforeEach o @AfterEach.
- Los métodos de clase anotados con @BeforeAll y los @AfterAll se ejecutan al crear la instancia y al destruir la instancia, solo tienen sentido en el ciclo de vida de instancia "por clase". Deben ser métodos de clase (static).

Ciclo de vida de instancia de prueba

- @**BeforeAll** public **static** void method()
  - Este método es ejecutado una vez antes de ejecutar todos los test. Se usa para ejecutar actividades intensivas como conectar a una base de datos. Los métodos marcados con esta anotación necesitan ser definidos como static para trabajar con JUnit.
- @**BeforeEach** public void method()
  - Este método es ejecutado antes de cada test. Se usa para preparar el entorno de test (p.ej., leer datos de entrada, inicializar la clase).
- @**AfterEach** public void method()
  - Este método es ejecutado después de cada test. Se usa para limpiar el entorno de test (p.ej., borrar datos temporales, restaurar valores por defecto). Se puede usar también para ahorrar memoria limpiando estructuras de memoria pesadas.
- @**AfterAll** public **static** void method()
  - Este método es ejecutado una vez después que todos los tests hayan terminado. Se usa para actividades de limpieza, como por ejemplo, desconectar de la base de datos. Los métodos marcados con esta anotación necesitan ser definidos como static para trabajar con JUnit.

Ejecución de pruebas

- Las pruebas se pueden ejecutar desde:
  - IDE: IntelliJ IDEA , Eclipse , NetBeans y Visual Studio Code
  - Herramientas de compilación: Gradle, Maven y Ant (y en los servidores de automatización que soporte alguna de ellas)
  - Lanzador de consola: aplicación Java de línea de comandos que permite iniciar JUnit Platform desde la consola.
- Al ejecutarse las pruebas se marcan como:
  - Successful: Se ha superado la prueba.
  - Failed: Fallo, no se ha superado la prueba.
  - Aborted (Skipped): Se ha cancelado la prueba por una asunción.
  - Disabled (Skipped): No se ha ejecutado la prueba.
  - Error: Excepción en el código del método de prueba, no se ha ejecutado la prueba.

Aserciones

- assertTrue(boolean condition, [message]): Verifica que la condición booleana sea true.
- assertFalse(boolean condition, [message]): Verifica que la condición booleana sea false.
- assertNull(object, [message]): Verifica que el objeto sea nulo.
- assertNotNull(object, [message]): Verifica que el objeto no sea nulo.
- assertEquals(expected, actual, [message]): Verifica que los dos valores son iguales.
  - assertEquals(expected, actual, tolerance, [message]): Verifica que los valores float o double coincidan. La tolerancia es el número de decimales que deben ser iguales.
- assertArrayEquals (expected, actual, [message]): Verifica que el contenido de dos arrays son iguales.
- assertIterableEquals(expected, actual, [message]): Verifica (profunda) que el contenido de dos iterables son iguales.
- assertLinesMatch(expectedLines, actualLines, [message]): Verifica (flexible) que el contenido de dos listas de cadenas son iguales.

Aserciones

- assertNotEquals(expected, actual, [message]): Verifica que los dos valores NO son iguales.
- assertSame(expected, actual, [message]): Verifica que las dos variables referencien al mismo objeto.
- assertNotSame(expected, actual, [message]): Verifica que las dos variables no referencien al mismo objeto.
- assertThrows(exceptionType, executable, [message]): Verifica que se genera una determinada excepción.
- assertDoesNotThrow(executable, [message]): Verifica que no lanza ninguna excepción.
- assertTimeout(timeout, executable , [message]): Verifica que la ejecución no exceda el timeout dado.
- assertAll(title, asserts): Verifica que se cumplan todas las aserciones de una colección.
- fail([message]): Hace que el método falle. Debe ser usado para probar que cierta parte del código no es alcanzable para que el test devuelva fallo hasta que se implemente el método de prueba.

Aserciones

- Aserciones de comparación:

```
assertEquals(2, calculator.add(1, 1));
assertTrue(Double.isInfinite(calculator.divide(1, 0)) , "División por cero");
assertNotNull(lst.get(1));
```

- Verificaciones de excepciones:

```
Exception forValidateMessageException = assertThrows(ArithmeticException.class, () -> divide(1.0, 0.0));
assertDoesNotThrow(() -> lst.remove(1));
```

- Verificación SLA:

```
String actualResult = assertTimeout(Duration.ofMillis(90), () -> {
   Thread.sleep(50);
   return "OK";
 });
assertEquals("OK", actualResult);
```

- Fallar directamente:

```
fail("Not yet implemented");
```

Hamcrest

- Si queremos aserciones todavía más avanzadas, se recomienda usar librerías específicas, como por ejemplo Hamcrest: http://hamcrest.org
- Hamcrest es un marco para escribir objetos de coincidencia que permite que las restricciones se definan de forma declarativa. Hay una serie de situaciones en las que los comparadores son invariables, como la validación de la interfaz de usuario o el filtrado de datos, pero es en el área de redacción de pruebas flexibles donde los comparadores son los más utilizados.

```
import static org.hamcrest.MatcherAssert.assertThat;

assertThat(calculator.subtract(4, 1), not(is(equalTo(2))));
assertThat(Arrays.asList("foo", "bar", "baz"), hasItem(startsWith("b") , endsWith("z")));
assertThat(person, has(
    property("firstName", equalTo("Pepito")),
    property("lastName", equalTo("Grillo")),
    property("age", greaterThan(18))));
```


AssertJ

- Otra alternativa para aserciones todavía más avanzadas es AssertJ: https://assertj.github.io/doc/
- AssertJ es una biblioteca de Java que proporciona una api fluido con un amplio conjunto de aserciones y mensajes de error realmente útiles, que mejora la legibilidad del código de prueba y está diseñado para ser súper fácil de usar dentro de su IDE favorito. AssertJ es un proyecto de código abierto que ha surgido a partir del desaparecido Fest Assert. También dispone de un generador automático de comprobaciones para los atributos de las clases, lo que añade más semántica a nuestras clases de prueba.

```
import static org.assertj.core.api.Assertions.*;

assertThat(fellowshipOfTheRing).hasSize(9)
                               .contains(frodo, sam)
                               .doesNotContain(sauron);
```

Agrupar aserciones

- La primera aserción que no se cumpla detiene la ejecución del método y marca la prueba como fallida.
- Si un método de prueba cuenta con varias aserciones, el fallo de una de ellas impedirá la evaluación de las posteriores por lo que no se sabrá si fallan o no, lo cual puede ser un inconveniente.
- Para solucionarlo se dispone de assertAll: ejecuta todas las aserciones contenidas e informa del resultado y, en caso de que alguna falle, assertAll falla.

```
@Test
void groupedAssertions() {
    assertAll("person",
        () -> assertEquals("Jane", person.getFirstName()),
        () -> assertEquals("Doe", person.getLastName())
    );
}
```


Pruebas repetidas

- Se puede repetir una prueba un número específico de veces con la anotación @RepeatedTest. Cada iteración se ejecuta en un ciclo de vida completo.
- Se puede configurar el nombre de salida con:
  - {displayName}: nombre del método que se ejecuta
  - {currentRepetition}: recuento actual de repeticiones
  - {totalRepetitions}: número total de repeticiones
- Se puede inyectar RepetitionInfo para tener acceso a la información de la iteración.

```
@RepeatedTest(value = 5, name = "{displayName} {currentRepetition}/{totalRepetitions}")
void repeatedTest(RepetitionInfo repetitionInfo) {
    assertEquals(5, repetitionInfo.getTotalRepetitions());
}
```

Pruebas parametrizadas

- Las pruebas parametrizadas permiten ejecutar una prueba varias veces con diferentes argumentos. Se declaran como los métodos @Test normales, pero usan la anotación @ParameterizedTest y aceptan parámetros. Además, se debe declarar al menos una fuente que proporcionará los argumentos para cada invocación y utilizar los parámetros en el método de prueba.
- Las fuentes disponibles son:
  - @ValueSource: un array de valores String, int, long, o double
  - @EnumSource: valores de una enumeración (java.lang.Enum)
  - @CsvSource: valores separados por coma, en formato CSV (comma-separated values)
  - @CsvFileSource: valores en formato CSV en un fichero localizado en el classpath
  - @MethodSource: un método estático de la clase que proporciona un Stream de valores
  - @ArgumentsSource: una clase que proporciona los valores e implementa el interfaz ArgumentsProvider

Pruebas parametrizadas

- Se puede configurar el nombre de salida con:
  - {displayName}: nombre del método que se ejecuta
  - {index}: índice de invocación actual (basado en 1)
  - {arguments}: lista completa de argumentos separados por comas
  - {argumentsWithNames}: lista completa de argumentos separados por comas con nombres de parámetros
  - {0} {1}: argumentos individuales

```
@ParameterizedTest(name = "{index} => ''{0}'' –> {1}")
@CsvSource({ "uno,3", "dos,3", "'tres, cuatro',12" })
void testWithCsvSource(String str, int len) {
```

@ValueSource

- Es la fuente más simples posibles, permite especificar una única colección de valores literales y solo puede usarse para proporcionar un único argumento por invocación de prueba parametrizada.
- Los tipos compatibles son: short, byte, int, long, float, double, char, boolean, String, Class.

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue(argument > 0 && argument < 4);
}
```


@EnumSource

- Si la lista de valores es una enumeración se anota con @EnumSource:

```
@ParameterizedTest
@EnumSource(TimeUnit.class)
void testWithEnumSource(TimeUnit unit) {
```

- En caso de no indicar el tipo, se utiliza el tipo declarado del primer parámetro del método. La anotación proporciona un atributo names opcional que permite: listar los valores (como cadenas) de la enumeración incluidos o excluidos o el patrón inclusión o exclusión. El atributo opcional mode determina el uso de names: INCLUDE (por defecto), EXCLUDE, MATCH_ALL, MATCH_ANY.

```
@ParameterizedTest
@EnumSource(mode = EXCLUDE, names = { "DAYS", "HOURS" })
@ParameterizedTest
void testWithEnumSource(TimeUnit unit) {
```

@CsvSource y @CsvFileSource

- La anotación @CsvSource permite expresar la lista de argumentos como valores separados por comas (cadenas literales con los valores separados). El delimitador predeterminado es la coma, pero se puede usar otro carácter configurando el atributo delimiter.
- @CsvSource usa el apostrofe (') como delimitador de cita literal (ignora los delimitadores de valor). Un valor entre apostrofes vacío ('') se interpreta como cadena vacía. Un valor vacío (,,) se interpreta como referencia null, aunque con el atributo nullValues se puede establecer el literal que se interpretará como null (nullValues = "NIL").

```
@ParameterizedTest
@CsvSource({ "uno,3", "dos,3", "'tres, cuatro',12" })
void testWithCsvSource(String str, int len) {
```

- Con la anotación @CsvFileSource se puede usar archivos CSV desde el classpath. Cada línea de un archivo CSV es una invocación de la prueba parametrizada (salvo que comience con # que se interpretará como un comentario y se ignorará). Con el atributo numLinesToSkip se pueden saltar las líneas de cabecera.

```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSource(String str, int len) {
```


@MethodSource

- La lista de valores se puede generar mediante un método factoría de la clase de prueba o de clases externas.
- Los métodos factoría deben ser static y deben devolver una secuencia que se pueda convertir en un Stream. Cada elemento de la secuencia es un juego de argumentos para cada iteración de la prueba, el elemento será a su vez una secuencia si el método de prueba requiere múltiples parámetros.
- El método factoría se asocia con la anotación @MethodSource donde se indica como cadena el nombre de la factoría (el nombre es opcional si coincide con el del método de prueba), si es un método externo de static se proporciona su nombre completo: com.example.clase#método.

```
private static Stream<Arguments> paramProvider() {
   return Stream.of(Arguments.of("uno", 3), Arguments.of("dos", 3), Arguments.of("tres", 4));
}
@ParameterizedTest
@MethodSource("paramProvider")
void testWithMethodSource(String str, int len) {
```

- Arguments es una abstracción que proporciona acceso a una matriz de objetos que se utilizarán para invocar un método @ParameterizedTest.

@ArgumentsSource

- Como alternativa a los métodos factoría se pueden utilizar clases proveedoras de argumentos. Mediante la anotación @ArgumentsSource se asociara al método de prueba. La clase proveedora debe declararse como una clase de nivel superior o como una clase anidada static. Debe implementar la interfaz ArgumentsProvider con el método provideArguments, similar a los métodos factoría.

```
static class CustomArgumentProvider implements ArgumentsProvider {
   @Override
   public Stream<? extends Arguments> provideArguments(ExtensionContext context) throws
Exception {
      return Stream.of(Arguments.of("uno", 2), Arguments.of("dos", 3), Arguments.of("tres", 4));
   }
}
@ParameterizedTest
@ArgumentsSource(CustomArgumentProvider.class)
void testWithArgumentsSource(String str, int len) {
```


Fuentes nulas y vacías

- Con el fin de comprobar los casos límite y verificar el comportamiento adecuado del código cuando se suministra una entrada incorrecta, puede ser útil suministrar valores null y vacíos en las pruebas con parámetros.
- Las siguientes anotaciones lo permiten:
  - @NullSource: proporciona al @ParameterizedTest un único argumento null, no se puede usar para un parámetro que tiene un tipo primitivo.
  - @EmptySource: proporciona un único argumento vacío al método, solo para parámetros de los tipos: String, List, Set, Map, las matrices primitivas (por ejemplo, int[], char[][], etc.), matrices de objetos (por ejemplo, String[], Integer[][], etc.). Los subtipos de los tipos admitidos no son compatibles.
  - @NullAndEmptySource: anotación compuesta que combina la funcionalidad de @NullSource y @EmptySource.

```
@ParameterizedTest
@NullSource
@ValueSource(strings = { "uno", "dos", "tres" } )
void testOrdinales(String candidate) {
```

Agregación de argumentos

- De forma predeterminada, cada argumento proporcionado a un método @ParameterizedTest corresponde a un único parámetro de método. En consecuencia, las fuentes de argumentos que se espera que proporcionen una gran cantidad de argumentos pueden generar firmas de métodos grandes.
- En tales casos, se puede utilizar un ArgumentsAccessor en lugar de varios parámetros. Con esta API, se puede acceder a los argumentos proporcionados a través de un único argumento pasado al método de prueba. Además, se puede recuperar el índice de la invocación de prueba actual con ArgumentsAccessor.getInvocationIndex().

```
@ParameterizedTest
@CsvSource({ "1,uno", "2,dos", "3,tres" })
void testArgumentsAccessor(ArgumentsAccessor args) {
    Elemento<Integer, String> ele = new Elemento<Integer, String>(args.getInteger(0),
            args.getString(1));
```


Plantillas de prueba

- Las pruebas parametrizadas nos permiten ejecutar un solo método de prueba varias veces con diferentes parámetros.
- Hay escenarios de prueba donde necesitamos ejecutar nuestro método de prueba no solo con diferentes parámetros, sino también bajo un contexto de invocación diferente cada vez:
  - usando diferentes parámetros
  - preparar la instancia de la clase de prueba de manera diferente, es decir, inyectar diferentes dependencias en la instancia de prueba
  - ejecutar la prueba en diferentes condiciones, como habilitar/deshabilitar un subconjunto de invocaciones si el entorno es "QA"
  - ejecutándose con un comportamiento de devolución de llamada de ciclo de vida diferente; tal vez queramos configurar y eliminar una base de datos antes y después de un subconjunto de invocaciones
- Los métodos de la plantilla de prueba, anotados con @TestTemplate, no son métodos normales, se ejecutarán múltiples veces dependiendo de los valores devueltos por el proveedor de contexto.

Plantillas de prueba

- Las pruebas repetidas y las pruebas parametrizadas son especializaciones integradas de las plantillas de prueba.
- Un método @TestTemplate solo puede ejecutarse si está registrada al menos una extensión TestTemplateInvocationContextProvider. La ejecución del proveedor suministra contextos con los que se invocan todos los métodos de plantilla como si fueran métodos @Test regulares con soporte completo para las mismas devoluciones de llamada y extensiones del ciclo de vida.

```
final List<String> fruits = Arrays.asList("apple", "banana", "lemon");

@TestTemplate
@ExtendWith(MyTestTemplateInvocationContextProvider.class)
void testTemplate(String fruit) {
   assertTrue(fruits.contains(fruit));
}
```


Plantillas de prueba

```
public class MyTestTemplateInvocationContextProvider implements TestTemplateInvocationContextProvider {
   @Override
   public boolean supportsTestTemplate(ExtensionContext context) { return true; }
   @Override
   public Stream<TestTemplateInvocationContext> provideTestTemplateInvocationContexts(ExtensionContext context) {
      return Stream.of(invocationContext("apple"), invocationContext("banana"));
   }
   private TestTemplateInvocationContext invocationContext(String parameter) {
      return new TestTemplateInvocationContext() {
         @Override
         public String getDisplayName(int invocationIndex) { return parameter; }
         @Override
         public List<Extension> getAdditionalExtensions() {
            return Collections.singletonList(new ParameterResolver() {
               @Override
               public boolean supportsParameter(ParameterContext parameterContext,
                     ExtensionContext extensionContext) {
                  return parameterContext.getParameter().getType().equals(String.class);
               }
               @Override
               public Object resolveParameter(ParameterContext parameterContext,
                     ExtensionContext extensionContext) {
                  return parameter;
               }
            });
         }
      };
   }
}
```

Pruebas Dinámicas

- Las pruebas dinámicas permiten crear, en tiempo de ejecución, un número variable de casos de prueba. Cada uno de ellos se lanza independientemente, por lo que ni su ejecución ni el resultado dependen del resto. El uso más habitual es generar batería de pruebas personalizada sobre cada uno de los estados de entrada de un conjunto (dato o conjunto de datos).
- Para implementar los pruebas dinámicas disponemos de la clase DynamicTest, que define un caso de prueba: el nombre que se mostrará en el árbol al ejecutarlo y la instancia de la interfaz Executable con el propio código que se ejecutará (método de prueba).
- Un método anotado con @TestFactory devuelve un conjunto iterable de DynamicTest (este conjunto es cualquier instancia de Iterator, Iterable o Stream).
- Al ejecutar se crearán todos los casos de prueba dinámicos devueltos por la factoría y los tratará como si se tratasen de métodos regulares anotados con @Test pero los métodos @BeforeEach y @AfterEach se ejecutan para el método @TestFactory pero no para cada prueba dinámica.

Pruebas Dinámicas

```
@TestFactory
Collection<DynamicTest> testFactory() {
   ArrayList<DynamicTest> testBattery = new ArrayList<DynamicTest>();
   DynamicTest testKO = dynamicTest("Should fail", () -> assertTrue(false));
   DynamicTest testOK = dynamicTest("Should pass", () -> assertTrue(true));
   int state = 1; // entity.getState();
   boolean rslt = true; // entity.isEnabled();
   if (rslt)
      testBattery.add(dynamicTest("Enabled", () -> assertTrue(true)));
   else
      testBattery.add(dynamicTest("Disabled", () -> assertTrue(true)));
   switch (state) {
   case 1:
      testBattery.add(testOK);
      break;
   case 2:
      testBattery.add(testKO);
      break;
   case 3:
      testBattery.add(testOK);
      testBattery.add(testKO);
      break;
   }
   return testBattery;
}
```

Desactivar pruebas y ejecución condicional

- Se pueden deshabilitar clases de prueba completas o métodos de prueba individuales mediante anotaciones.
- @Disabled: Deshabilita incondicionalmente todos los métodos de prueba de una clase o métodos individuales.
- @EnabledOnOs y @DisabledOnOs: Habilita o deshabilita una prueba para un determinado sistema operativo.

  @EnabledOnOs({ OS.LINUX, OS.MAC })
- @EnabledOnJre, @EnabledForJreRange, @DisabledOnJre y @DisabledForJreRange: Habilita o deshabilita para versiones particulares o un rango de versiones del Java Runtime Environment (JRE).

  @EnabledOnJre(JAVA_8), @EnabledForJreRange(min = JAVA_9, max = JAVA_11)
- @EnabledIfSystemProperty y @DisabledIfSystemProperty: Habilita o deshabilita una prueba en función del valor de una propiedad del JVM.

  @EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
- @EnabledIfEnvironmentVariable y @DisabledIfEnvironmentVariable: Habilita o deshabilita en función del valor de una variable de entorno.

  @EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")

Condiciones personalizadas

- Se puede habilitar o deshabilitar un contenedor o prueba según un método de condición configurado mediante las anotaciones @EnabledIfy @DisabledIf. El método de condición debe tener un tipo de retorno boolean, sin parámetros o un parámetros de tipo ExtensionContextar.

```
boolean customCondition() {
   return true;
}
@Test
@EnabledIf("customCondition")
void enabled() {
   // ...
}
@Test
@DisabledIf("customCondition")
void disabled() {
   // ...
}
```

Ejecución condicional

- Las clases y métodos de prueba se pueden etiquetar mediante la anotación @Tag. Las etiquetas se pueden usar más tarde para incluir y/o excluir (filtrar) del descubrimiento y la ejecución de las pruebas.

```
@Tag("Model")
class EntidadTest {
```

- Se recomienda crear anotaciones propias para las etiquetas:

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("smoke")
public @interface Smoke { }
```

En Eclipse: Run configurations → Test → Include and exclude tags

Asunciones

- Como alternativa a las anotaciones @EnabledOn y @DisabledOn, la clase org.junit.jupiter.api.Assumptions ofrece una colección de métodos de utilidad que permiten la ejecución de pruebas condicionales basadas en suposiciones.
- En contraste directo con afirmaciones fallidas, las asunciones ignoran la prueba (skipped) cuando no se cumplen.
- Las asunciones se usan generalmente cuando no tiene sentido continuar la ejecución de un método de prueba dado que depende de algo que no existe en el entorno de ejecución actual.
- Las asunciones disponibles son:
  - assumeTrue(boolean assumption, [message]): Continua si la condición booleana es true.
  - assumeFalse(boolean assumption, [message]): Continua si la condición booleana es false.
  - assumingThat(boolean assumption, executable): Ejecuta solo si se cumple la condición.

Orden de ejecución de prueba

- Cada caso de prueba debe ser independiente del resto de los casos, por lo que deben poderse ejecutar en cualquier orden (desordenados). Por defecto, los métodos de prueba se ordenarán usando un algoritmo que es determinista pero intencionalmente no obvio.
- Para controlar el orden en que se ejecutan los métodos de prueba, hay que anotar la clase con @TestMethodOrder:
  - MethodOrderer.DisplayName, MethodOrderer.MethodName, ~~MethodOrderer.Alphanumeric~~: ordena los métodos de prueba alfanuméricamente en función de sus nombres y listas de parámetros formales.
  - MethodOrderer.OrderAnnotation: ordena los métodos de prueba numéricamente según los valores especificados a través de la anotación @Order.

```
@TestMethodOrder(OrderAnnotation.class)
class OrderedTestsDemo {
   @Test
   @Order(1)
   void nullValues() { }
```


Tiempos de espera

- La anotación @Timeout permite declarar que una prueba, factoría de pruebas, plantilla de prueba o un método de ciclo de vida deberían dar error si su tiempo de ejecución excede una duración determinada.
- La unidad de tiempo para la duración predeterminada es segundos pero es configurable.

```
@Test
@Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
void failsIfExecutionTimeExceeds100Milliseconds() {
      try {
         Thread.sleep(150);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
}
```

- Las aserciones de timeout solo miden el tiempo de la propia aserción, en caso de excederlo dan la prueba como fallida.

Modelo de extensión

- El nuevo modelo de extensión de JUnit 5, la Extension API, permite ampliar el modelo de programación con funcionalidades personalizadas.
- Gracias al modelo de extensiones, frameworks externos pueden proporcionar integración con JUnit 5 de una manera sencilla.
- Hay 3 formas de usar una extensión:
  - Declarativamente, usando la anotación @ExtendWith (se puede usar a nivel de clase o de método)

```
@ExtendWith({ DatabaseExtension.class, WebServerExtension.class })
class MyFirstTests {
```

  - Programáticamente, usando la anotación @RegisterExtension (anotando campos en las clases de prueba)

```
@RegisterExtension
static WebServerExtension server = WebServerExtension.builder()
    .enableSecurity(false)
    .build();
```

  - Automáticamente, usando el mecanismo de carga de servicios de Java a través de la clase java.util.ServiceLoader

Inyección de dependencia

- La inyección de dependencia en JUnit Jupiter permite que los constructores y métodos de prueba tengan parámetros. Estos parámetros deben ser resueltos dinámicamente en tiempo de ejecución por un resolutor de parámetros registrado.
- ParameterResolveres es la extensión para proporcionar resolutores de parámetros que se pueden registrar mediante anotaciones @ExtendWith. Aquí es donde el modelo de programación de Júpiter se encuentra con su modelo de extensión.
- JUnit Jupiter proporciona algunos solucionadores integrados que se registran automáticamente:
  - TestInfo para acceder a información sobre la prueba que se está ejecutando actualmente.
  - TestReporter para publicar en la consola datos adicionales sobre la ejecución de prueba actual.

```
@Test @Smoke @DisplayName("Cotilla")
void cotilla(TestInfo testInfo, TestReporter testReporter) {
    assertEquals("Cotilla", testInfo.getDisplayName());
    assertTrue(testInfo.getTags().contains("smoke"));
    for(String tag: testInfo.getTags()) testReporter.publishEntry(tag);
}
```

Dobles de prueba

- La regla de oro de las pruebas unitarias, es que una unidad (unit) tiene que ser testeada sin utilizar ninguna de sus dependencias.
- Siguiendo la misma regla de oro, las pruebas de integración y sistema deben estar aisladas de sus dependencias salvo cuando se estén probando dichas dependencias. Así mismo, el resultado de las pruebas debe ser previsible.
- Entre las ventajas de esta aproximación se encuentran:
  - Devuelven resultados determinísticos
  - Permiten crear o reproducir determinados estados (por ejemplo errores de conexión)
  - Obtienen resultados mucho mas rápidamente y a menor coste, incluso offline.
  - Permiten el inicio temprano de las pruebas incluso cuando las dependencias todavía no están disponibles.
  - Permiten incluir atributos o métodos exclusivamente para el testeo.

Extensiones

- DBUnit: para realizar pruebas que comprueben el correcto funcionamiento de las operaciones de acceso y manejo de datos que se encuentran dentro de las bases de datos.
- Mockito: es uno de los frameworks de Mock más utilizados en la plataforma Java.
- JMockit: es uno de los frameworks mas completos.
- EasyMock: es un framework basado en el patrón record-replay-verify.
- PowerMock: es un framework que extiende tanto EasyMock como Mockito complementándolos

https://site.mockito.org/

MOCKITO

Introducción

- Mockito es un marco de simulación de Java que tiene como objetivo proporcionar la capacidad de escribir con claridad una prueba de unidad legible utilizando un API simple. Se diferencia de otros marcos de simulacros al dejar el patrón de esperar-ejecutar-verificar que la mayoría de los otros marcos utilizan.
- En su lugar, solo conoce una forma de simular las clases e interfaces (no final) y permite verificar y apilar basándose en comparadores de argumentos flexibles.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <scope>test</scope>
</dependency>
```

https://jar-download.com/artifacts/org.mockito

Extensión Junit 5

- Para poder utilizar la extensión se requiere la dependencia:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
</dependency>
```

- Para registran la extensión:

```
@ExtendWith(MockitoExtension.class)
class MyTest {
```

- Las sentencias de importar son:

```
import org.mockito.junit.jupiter.MockitoExtension;
import static org.mockito.Mockito.*;
```

- Para minimiza el código repetitivo y hacer la clase de prueba más legible dispone de anotaciones como alternativa al código imperativo.

Dobles de prueba

- Mockito provee de dos mecanismos para crear dobles de prueba.
  - Mock: Stub que solo proporciona respuestas predefinidas, sin responder a cualquier otra cosa fuera de aquello para lo que ha sido programado. Toman como referencia una clase o interface existente.

```
@Mock private Calculator calculator;  // declarativo

Calculator calculator = mock(Calculator.class); // imperativo
```

  - Spy: Objetos real al que se le suplantan determinados métodos.

```
@Spy private Calculator calculator;

Calculator calculator = spy(Calculator.class);
```

Suplantación

- El proceso de creación de los métodos de suplantación sigue los siguientes patrones:
  - Devolución de un valor:
    ```
    when(mockedList.get(0)).thenReturn("first");
    doReturn("first").when(mockedList).get(0);
    doNothing().when(mockedList).clear(); // Procedimientos
    ```
  - Devolución de multiples valores (iteradores):
    ```
    when(mockedList.get(0)).thenReturn(10).thenReturn(20).thenReturn(30);
    ```
  - Devolución de una excepción:
    ```
    when(mockedList.get(1)).thenThrow(new RuntimeException());
    doThrow(new RuntimeException()).when(mockedList).clear();
    ```
  - Invocación del método real.
    ```
    when(mockedList.get(0)).thenCallRealMethod();
    doCallRealMethod().when(mockedList).clear();
    ```
- Es obligatorio utilizar en la prueba todos los métodos de suplantación o se genera una excepción.

Argumentos

- Mockito solo suplanta con los valores literales definidos y verifica los valores de los argumentos siguiendo el estilo natural de Java: mediante el uso de un método equals().
- A veces, cuando se requiere una mayor flexibilidad que los valores literales, se pueden usar los emparejadores de argumentos.
  ```
  when(mockedList.get(anyInt())).thenReturn("element");
  ```
- Dispone de una amplia colección de matchers como emparejadores:
  - any, is, that, eq, some, notNull, …
- Si se está utilizando emparejadores de argumentos, todos los argumentos deben ser proporcionados por emparejadores.

Preparar valores

- La mayoría de las pruebas requieren que los valores devueltos sean constantes.
- Para aquellas que requieran la preparación de los valores devueltos se dispone de la interfaz genérica Answer:

```
when(mock.someMethod(anyString())).thenAnswer(
    new Answer() {
        public Object answer(InvocationOnMock invocation) {
            Object[] args = invocation.getArguments();
            Object mock = invocation.getMock();
            // …
            return rslt;
        }
});
```

- También está disponible la versión:

```
doAnswer(Answer).when(mockedList).someMethod(anyString());
```


Verificación

- Mockito suministra el método verify para validar la interacción con los métodos suplantados o espiar métodos no suplantados. Se comporta como una aserción: si falla la verificación, falla la prueba.
- Verificar que se ha invocado el método:

```
verify(mockedList, description("This will print on failure")).get(0);
```

- Verificar que se ha invocado el método un determinado número de veces:

```
verify(mockedList, times(3)).get(0);
```

- Verificar que se ha invocado el método al menos o como mucho un número de veces:

```
verify(mockedList, atLeast(2)).get(0);
verify(mockedList, atMost(5)).get(0);
```

- Verificar que se no ha invocado el método:

```
verify(mockedList, never()).get(0);
```

Verificación

- Verificación en orden:

```
InOrder inOrder = inOrder(mockedList);
inOrder.verify(mockedList).get(0);
inOrder.verify(mockedList).get(1);
```

- Verificar que no se produjeron mas invocaciones que las explícitamente verificadas:

```
verifyNoMoreInteractions(mockedList);
```

- Verificar que no se produjeron mas invocaciones a ninguno de los métodos:

```
verifyZeroInteractions(mockedList);
```

- Tiempos de espera antes de la verificación:

```
verify(mock, timeout(100)).someMethod();
verify(mock, timeout(100).times(2)).someMethod();
verify(mock, timeout(100).atLeast(2)).someMethod();
```


Verificando argumentos

- A veces, cuando se requiere una flexibilidad adicional para verificar los argumentos, se necesita capturar con qué parámetros fueron invocadas utilizando ArgumentCaptor y @Captor: permiten extraer los argumentos en le método de prueba y realizar aseveraciones en ellos.
- Para extraer los parámetros se utiliza el método capture en el verify del método que se quiere capturar.

```
ArgumentCaptor<List<String>> captor = ArgumentCaptor.forClass(List.class);
List mockedList = mock(List.class);
mockedList.addAll(Arrays.asList("uno", "dos", "tres"));

verify(mockedList).addAll(captor.capture());

assertEquals(3, captor.getValue().size());
assertEquals("dos", captor.getValue().get(1));
```

- Los capturadores se pueden declarar mediante anotaciones:

```
@Captor
ArgumentCaptor<List<String>> captor
```

Inyección de dependencias

- Utilizando la anotación @InjectMocks, se pueden inyectar los mocks y spy que requieren los objetos con dependencias.
- Mockito intentará resolver la inyección de dependencia en el siguiente orden:
  - Inyección basada en el constructor: los simulacros se inyectan en el constructor a través de los argumentos (si no se pueden encontrar alguno de los argumentos, se pasan nulos). Si un objeto es creado con éxito a través del constructor, entonces no se aplicarán otras estrategias.
  - Inyección basada en Setter: los mock son inyectados por los tipos de las propiedades. Si hay varias propiedades del mismo tipo, inyectará en aquellas que los nombres de las propiedades y los nombres simulados coincidirán.
  - Inyección directa en el campo: igual que la inyección basada en Setter.
- Hay que tener en cuenta que no se notifica ningún error en caso de que alguna de las estrategias mencionadas fallara.

Inyección de dependencias

```
public class ArticleManagerTest extends SampleBaseTestCase {
    @Mock private ArticleCalculator calculator;
    @Mock(name = "database") private ArticleDatabase dbMock;
    @Spy private UserProvider userProvider = new ConsumerUserProvider();

    @InjectMocks
    private ArticleManager manager;

    @Test public void shouldDoSomething() {
       manager.initiateArticle();
       verify(database).addListener(any(ArticleListener.class));
    }
  }

public class ArticleManager {
    private ArticleCalculator calculator;
    ArticleManager(ArticleCalculator calculator, ArticleDatabase database) {
       // parameterized constructor
    }
    void setDatabase(ArticleDatabase database) { }
```

BDDMockito

- El término BDD fue acuñado por primera vez por Dan North, en 2006. BDD fomenta la escritura de pruebas en un lenguaje natural legible por humanos que se centra en el comportamiento de la aplicación.
- Define una forma claramente estructurada de escribir pruebas siguiendo tres secciones (similares Preparar, Actuar, Afirmar):
  - Given: dadas algunas condiciones previas (Arrange)
  - When: cuando ocurre una acción (Act)
  - Then: luego verifique la salida (Assert)
- La biblioteca Mockito ahora trae una clase BDDMockito que presenta un API compatible con BDD. Esta API permite adoptar un enfoque más amigable con BDD preparando las pruebas con given() y haciendo afirmaciones usando then().
- Las pruebas se vuelven más legibles con la importación estática:
  import static org.mockito.BDDMockito.*;

BDDMockito

- BDDMockito proporciona alias BDD para varios métodos de Mockito: given(), en lugar when(), y then(), en lugar de verify().

```
given(phoneBookRepository.contains(momContactName)) // fijo
  .willReturn(false);
given(phoneBookRepository.getPhoneNumberByContactName(momContactName)) // dinámico
  .will((InvocationOnMock invocation) ->
      invocation.getArgument(0).equals(momContactName) ? momPhoneNumber : null);
willThrow(new RuntimeException()) // excepción
  .given(phoneBookRepository)
  .insert(any(String.class), eq(tooLongPhoneNumber));

try {
    phoneBookService.register(xContactName, tooLongPhoneNumber);
    fail("Should throw exception");
} catch (RuntimeException ex) { }

then(phoneBookRepository)
  .should(never())
  .insert(momContactName, tooLongPhoneNumber);
```

- Se debe intercambiar give y will* para los métodos que no tiene valor de retorno.

Servidores simulados (Mock Servers)

- Los retrasos en el front-end o back-end dificultan que los equipos dependientes completen su trabajo de manera eficiente. Los servidores simulados pueden aliviar esos retrasos en el proceso de desarrollo.
- Esto permite implementar y probar clientes mucho más rápido que si se hubiera tenido que esperar a que se desarrollara la solución real.
- Antes de enviar una solicitud real, se puede crear un mock server para simular cada punto final y su respuesta correspondiente, esto permite obtener un resultado rápido y determinista, incluso off-line.
- Las herramientas de pruebas e2e permiten la creación de mock server:
  - SoapUI (https://www.soapui.org)
  - Postman (https://www.getpostman.com/)

MÉTRICAS PARA EL PROCESO DE PRUEBAS DE SOFTWARE

Métricas de código

- La mayor complejidad de las aplicaciones de software moderno también aumenta la dificultad de hacer que el código confiable y fácil de mantener.
- Las métricas de código son un conjunto de medidas de software que proporcionan a los programadores una mejor visión del código que están desarrollando.
- Aprovechando las ventajas de las métricas del código, los desarrolladores pueden entender qué tipos o métodos deberían rehacerse o más pruebas.
- Los equipos de desarrollo pueden identificar los posibles riesgos, comprender el estado actual de un proyecto y realizar un seguimiento del progreso durante el desarrollo de software.
- Los desarrolladores pueden usar el IDE para generar datos de métricas de código que medir la complejidad y el mantenimiento del código administrado.

Contadores de cobertura de código

- Instrucciones
  - Los recuentos de unidades más pequeñas son instrucciones de código. La cobertura de instrucciones proporciona información sobre la cantidad de código que se ha ejecutado o se ha perdido. Esta métrica es completamente independiente del formato de origen.
- Líneas
  - Se puede calcular la información de cobertura para líneas individuales, pero un línea puede contener varias instrucciones o una instrucción contar con varias líneas. Una línea fuente se considera cubierta cuando se ha ejecutado al menos una instrucción asignada a esta línea.
- Bifurcaciones, ramas, bloques
  - Calcula la cobertura de bifurcaciones para todos los if y switch. Esta métrica cuenta el número total de tales ramas en un método y determina el número de ramas ejecutadas o perdidas.

Contadores de cobertura de código

- Métodos, funciones, procedimientos, …
  - Como bloques funcionales, se considera cubierto cuando se ha ejecutado al menos una instrucción.
- Clases, tipos, …
  - Una clase (o interfaz con código) se considera cubierta cuando se ha ejecutado al menos uno de sus métodos (incluidos constructores e inicializadores).
- Complejidad ciclomática:
  - Mide la complejidad del código estructural. Se crea, calculando el número de rutas de acceso de código diferente en el flujo del programa. Un programa que tiene un flujo de control complejo requiere más pruebas para lograr una buena cobertura de código y es mas difícil de mantener.

Calidad de las pruebas

- Se insiste mucho en que la cobertura de test unitarios de los proyectos sea lo más alta posible, pero es evidente que cantidad (de test, en este caso) no siempre implica calidad, la calidad no se puede medir "al peso", y es la calidad lo que realmente importa.
- La cobertura de prueba tradicional (líneas, instrucciones, rama, etc.) mide solo qué código ejecuta las pruebas. No comprueba que las pruebas son realmente capaces de detectar fallos en el código ejecutado, solo pueden identificar el código que no se ha probado.
- Los ejemplos más extremos del problema son las pruebas sin afirmaciones (poco comunes en la mayoría de los casos). Mucho más común es el código que solo se prueba parcialmente, cubrir todo los caminos no implica ejercitar tolos las clases de equivalencia y valores límite.
- La calidad de las pruebas también debe ser puesta a prueba: No serviría de tener una cobertura del 100% en test unitarios, si no son capaces de detectar y prevenir problemas en el código.
- La herramienta que testea los test unitarios son los test de mutaciones: Es un test de los test.

Pruebas de mutaciones

- Los pruebas de mutaciones son las pruebas de las pruebas unitarias y el objetivo es tener una idea de la calidad de las pruebas en cuanto a fiabilidad.
- Su funcionamiento es relativamente sencillo: la herramienta que se utilice debe generar pequeños cambios en el código fuente. A estos pequeños cambios se les conoce como mutaciones y crean mutantes.
- Una vez creados los mutantes, se lanzan todos los tests:
  - Si los test unitarios fallan, es que han sido capaces de detectar ese cambio de código. A esto se le llama matar al mutante.
  - Si, por el contrario, los test unitarios pasan, el mutante sobrevive y la fiabilidad (y calidad) de los tests unitarios queda en entredicho.
- Los test de mutaciones presentan informes del porcentaje de mutantes detectados: cuanto más se acerque este porcentaje al 100%, mayor será la calidad de nuestros test unitarios.

Pruebas de mutaciones

- Los mutantes cuyo comportamiento es siempre exactamente igual al del programa original se los llama mutantes funcionalmente equivalentes o, simplemente, mutantes equivalentes, y representan “ruido” que dificulta el análisis de los resultados.
- Para poder matar a un mutante:
  - La sentencia mutada debe estar cubierta por un caso de prueba.
  - Entre la entrada y la salida debe crearse un estado intermedio erróneo.
  - El estado incorrecto debe propagarse hasta la salida.
- La puntuación de mutación para un conjunto de casos de prueba es el porcentaje de mutantes no equivalentes muertos por los datos de prueba:
  - Mutación Puntuación = 100 * D / (N - E)

  donde D es el número de mutantes muertos, N es el número de mutantes y E es el número de mutantes equivalentes

ENTORNO DE PRUEBAS WEB

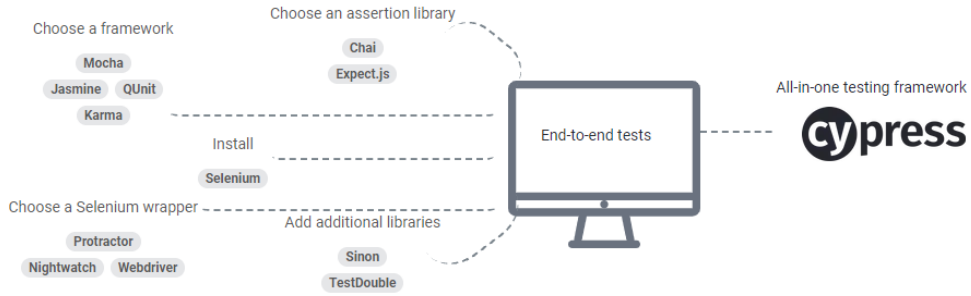
Calidad de Software

- El JavaScript es un lenguaje muy poco apropiado para trabajar en un entorno de calidad de software.
- En descargo del lenguaje JavaScript y de su autor, Brendan Eich, hay que decir que los problemas que han forzado esta evolución del lenguaje (así como las críticas ancestrales de la comunidad de desarrolladores) vienen dados por lo que habitualmente se llama “morir de éxito”.
- Jamás se pensó que un lenguaje que Eich tuvo que montar en 12 días como una especie de “demo” para Mozilla, pasase a ser omnipresente en miles de millones de páginas Web. O como el propio Hejlsberg comenta:

  “JavaScript se creó –como mucho- para escribir cien o doscientas líneas de código, y no los cientos de miles necesarias para algunas aplicaciones actuales.”
- JavaScript y el resto de los lenguajes web (HTML, CSS, …) son interpretados directamente por el navegador, por lo que hasta los errores sintácticos pueden llegar a producción, agravando los problemas de calidad.

Herramientas

- Escribir pruebas requiere muchas herramientas diferentes trabajando juntas: entornos de pruebas, bibliotecas de aserciones, dobles de pruebas, control remoto de navegadores, utilidades, …



Herramientas

- Test framework:
  - Jasmine: http://jasmine.github.io/
  - Jest: https://jestjs.io
  - Mocha: http://mochajs.org/
  - QUnit: http://qunitjs.com/
- Aserciones:
  - Chai: http://chaijs.com/
- Dobles de pruebas:
  - Sinon.JS: http://sinonjs.org/
- End to End (e2e)
  - Selenium: https://www.selenium.dev/
  - Cypress: https://www.cypress.io/
- Test runner:
  - Karma: https://karma-runner.github.io/
  - Web Test Runner: https://modern-web.dev/guides/test-runner/

ESLint

- Los analizadores de código son herramientas que realizan la lectura del código fuente y devuelve observaciones o puntos en los que tu código puede mejorarse desde la percepción de buenas prácticas de programación y código limpio.
- ESLint (https://eslint.org/) es una herramienta para identificar e informar sobre patrones encontrados en código ECMAScript/JavaScript, con el objetivo de hacer que el código sea más consistente y evitar errores: toma nuestro código, lo escanea y, si encuentra un problema, devuelve un mensaje describiéndolo y mostrando su ubicación aproximada.
- Se puede instalar ESLint usando npm (las dependencias se instalarán automáticamente):
  - npm install --save-dev eslint typescript-eslint
- Luego se debe crear un archivo de configuración .eslintrc.json en el directorio:
  - npm init @eslint/config
    { "parserOptions": { "project": "./tsconfig.json" }, "extends": ["plugin:typescript-eslint/recommended"]}
- Se puede ejecutar ESLint con cualquier archivo o directorio:
  - npx eslint src/**/*{.ts,.tsx}
- Se puede agregar un nuevo script al package.json:
  - "lint": "eslint src/**/*{.ts,.tsx}"

Karma: Gestor de Pruebas unitarias de JavaScript

- Instalación general:
  - npm install -g karma
  - npm install -g karma-cli
- Generar fichero de configuración karma.conf.js:
  - karma init
- Levantar el servidor se pruebas:
  - karma start
- Ingenierías de pruebas unitarias disponibles:
  - http://jasmine.github.io/
  - http://qunitjs.com/
  - http://mochajs.org
  - https://github.com/caolan/nodeunit
  - https://github.com/nealxyc/nunit.js

Web Test Runner

https://modern-web.dev/docs/test-runner/overview/

- Instalación como dependencia de desarrollo:
  - npm i --save-dev @web/test-runner
- Configurar su arranque en package.json:

  "test": "web-test-runner test/**/*.test.js --node-resolve"

  "test": "web-test-runner test/**/*.test.js --node-resolve --watch"

  "test": "web-test-runner test/**/*.test.js --node-resolve --coverage"
- Ejecutar las pruebas:
  - npm test

https://jestjs.io

JEST

Introducción

- Jest es un marco de pruebas (test runner) de JavaScript que se centra en la simplicidad.
- Jest es un marco de prueba diseñado para garantizar la corrección de cualquier base de código JavaScript. Permite escribir pruebas con una API accesible, familiar y rica en funciones que brinda resultados rápidamente.
- Jest está bien documentado, requiere poca configuración y puede ampliarse para adaptarse a las necesidades.
- Jest utiliza un cargador personalizado para las importaciones en sus pruebas, lo que facilita la simulación de cualquier objeto fuera del alcance de la prueba. Se puede usar importaciones simuladas con la rica API de funciones simuladas para espiar llamadas a funciones con sintaxis de prueba legible.
- Jest puede recopilar información de cobertura de código de proyectos completos, incluidos archivos no probados.

Instalación, configuración y ejecución

- Para su instalación mediante npm:
  - npm install --save-dev jest @types/jest
- Por defecto, se utiliza la extensión .test.js para los ficheros de pruebas.
- La configuración se puede definir en el archivo package.json del proyecto, a través de un archivo jest.config.js o jest.config.ts y a través de la opción --config <path/to/file.js|ts|cjs|mjs|json>.

```
"scripts": {
   :
   "test": "stencil test --spec --e2e",
   "test.watch": "stencil test --spec --e2e --watchAll",
  },
  "jest": {
   "displayName": "Curso de Stencil"
  },
```

- Se puede automatizar su ejecución con webpack, Babel, Parcel, …
- Se puede ejecutar directamente desde Jest CLI (npm i jest --global):
  - jest my-test --notify --config=config.json

Suites

- Una “suite” es un nombre que describe al género o sección qué se va a pasar por un conjunto de pruebas unitarias, además es una herramienta que es el núcleo que se necesita para poder tener un orden en el momento de crear las pruebas.
- Las “suites” se crean con la función **describe**, que es una función global y con la cual se inicia toda prueba unitaria, además consta con dos parámetros:

```
describe("Una suite es sólo una función", function() {
//…
});
```

- El primer parámetro es una cadena de caracteres donde se define el nombre descriptivo de la prueba unitaria y el segundo es una función con el código que ejecutará la prueba de código.
- Se pueden anidar “suites” para estructurar conjuntos complejos y facilitar la legibilidad y la búsqueda, basta con crear un describe dentro de otro.

Especificaciones

- Una especificación contiene una o más expectativas (algo que se espera) que ponen a prueba el estado del código. Una expectativa de Jest es una afirmación que debe ser verdadera pero puede ser falsa.
- Una especificación con todas las expectativas verdaderas es una especificación que pasa la prueba, pero con una o más falsas es una especificación que falla.
- Las especificaciones se definen dentro de una Suite llamando a la función global del **test** (o el alias **it**) que, al igual que describe, recibe una cadena (el título de la especificación) y una función (la especificación o caso de prueba), opcionalmente puede recibir un timeout.

```
test("y así es una especificación", function() {
//…
});
```

- **describe** y **test** son funciones: pueden contener cualquier código ejecutable necesario para implementar la prueba y las reglas de alcance de JavaScript se aplican, por lo que las variables declaradas en un describe están disponibles para cualquier bloque test dentro de la suite.

Expectativas

- Las expectativas se construyen con la función expect que obtiene un valor real de una expresión y lo comparan mediante una función Matcher con un el valor esperado (constante).

  expect(valor actual).*matchers*(valor esperado);
- Los matchers (comparadores) son funciones que implementan comparaciones booleanas entre el valor actual y el esperado, ellos son los responsables de informar a Jest si la expectativa se cumple o es falsa.
- Cualquier matcher puede evaluarse como una afirmación negativa mediante el encadenamiento a la llamada expect de un not antes de llamar al matcher.

  expect(valor actual)**.not**.matchers(valor esperado);
- También existe la posibilidad de escribir matchers personalizados para cuando el dominio de un proyecto consiste en afirmaciones específicas que no se incluyen en los ya definidos.

Matchers

.toEqual(y); verifica si ambos valores son iguales ==.

.toBe(y); verifica si ambos objetos son idénticos ===.

.toMatch(pattern); verifica si el valor pertenece al patrón establecido.

.toMatchObject(object); verifica que un objeto coincide en propiedad y valor con un subconjunto de otro objeto.

.toBeDefined(); verifica si el valor está definido.

.toBeUndefined(); verifica si el valor es indefinido.

.toBeNull(); verifica si el valor es nulo.

.toBeNaN(); verifica si el valor es NaN.

.toBeCloseTo (n, d); verifica la precisión matemática (número de decimales).

.toContain(y); verifica si el valor actual contiene el esperado.

Matchers

.toBeTruthy(); verifica si el valor es verdadero.
.toBeFalsy(); verifica si el valor es falso.
.toBeLessThan(y); verifica si el valor actual es menor que el esperado.
.toBeLessThanOrEqual (y); verifica si el valor actual es menor o igual que el esperado.
.toBeGreaterThan(y); verifica si el valor actual es mayor que el esperado.
.toBeGreaterThanOrEqual (y); verifica si el valor actual es mayor o igual que el esperado.
.toBeInstanceOf(Class); verifica que es instancia de cierta clase.
.toThrow(); verifica si una función lanza una excepción.
.toThrow(error?); verifica si una función lanza una excepción específica.

Código asincrónico

- Es común en JavaScript ejecutar código de forma asincrónica. En estos casos, Jest debe saber cuando ha terminado el código que se está probando, antes de que puede pasar a otra test.
- Jest tiene varias formas de manejar esto.
  - Devoluciones de llamada (Callbacks)
  - Promesas
  - Async / Await

Devoluciones de llamada (Callbacks)

- El patrón asincrónico más común son las devoluciones de llamada: pasar como argumento la función que se desea ejecutar cuando termine el proceso asíncrono.
- El problema es que la prueba se completará tan pronto como se complete la llamada asíncrona, antes de llamar a la devolución de llamada.
- Hay una forma alternativa de test que soluciona este problema: definir como parámetro (por convenio denominada done) con la función culla llamada Jest esperará para dar como finalizada la prueba.

```
test('the data is peanut butter', done => {
  function callback(data) {
    try {
      expect(data).toBe('peanut butter');
      done();
    } catch (error) { done(error); }
  }
  fetchData(callback);
});
```

- Si done() nunca se llama, la prueba fallará (con un error de tiempo de espera), que es lo que se espera que suceda. Si la declaración expect falla, arroja un error y done() no se llama por lo que hay que invocarla en un bloque try.

Promesas

- Si el código usa promesas, existe una forma más sencilla de manejar las pruebas asincrónicas.
- Si la función test devuelve una promesa, Jest esperará a que esa promesa se resuelva. Si se rechaza la promesa, la prueba fallará automáticamente.

```
test('the fetch fails with an error', () => {
  expect.assertions(1); // verificar que invoca la aserción
  return fetchData().catch(e => expect(e).toMatch('error'));
});
```

- También se puede utilizar los marcadores .resolves y .rejects en la declaración de "expect" y Jest esperará a que esa promesa resuelva o rechace.

```
test('the data is peanut butter', () => {
  return expect(fetchData()).resolves.toBe('peanut butter');
});
```

Async / Await

- Como sintaxis alternativa a las promesas se puede usar async/await en las pruebas. Para escribir una prueba asíncrona, hay que usar la palabra clave async delante de la función pasada a test.

```
test('the data is peanut butter', async () => {
  const data = await fetchData();
  expect(data).toBe('peanut butter');
});
test('the fetch fails with an error', async () => {
  expect.assertions(1);
  try {
    await fetchData();
  } catch (e) {
    expect(e).toMatch('error');
  }
});
```


Montaje y desmontaje

- Para montar el escenario de pruebas suele ser necesario definir e inicializar un conjunto de variables. Para evitar la duplicidad de código y mantener las variables inicializadas en un solo lugar además de mantener la modularidad, Jest suministra las funciones globales, a nivel de fichero, o local, dentro de una suite:
  - beforeAll(fn) se ejecuta solo una vez antes empezar a ejecutar las especificaciones del “describe”.
  - beforeEach(fn) se ejecuta antes de cada especificación dentro del “describe”.
  - afterEach(fn) se ejecuta después de cada especificación dentro del “describe”.
  - afterAll(fn) se ejecuta solo una vez después de ejecutar todas las especificaciones del “describe”.

```
describe("operaciones aritméticas", function(){
    let cal;
    beforeEach(function(){ calc = new Calculadora(); });
    it("adicion", function(){ expect(calc.suma(4)).toEqual(4); });
    it("multiplicacion", function(){ expect(calc.multiplica(7)).toEqual(0); });
    // …
```

- Otra manera de compartir las variables entre una beforeEach, test y afterEach es a través de la palabra clave this. Cada expectativa beforeEach/test/afterEach tiene el mismo objeto vacío this que se restablece de nuevo a vacío para de la siguiente expectativa beforeEach/test/afterEach.

Control de ejecución

- El marcador .**skip** permite desactivar las suites y las especificaciones, estas se omiten cuando se ejecutan las pruebas y aparecerán como skipped entre los resultados de la prueba.
- De igual forma, las especificaciones pendientes de implementar se pueden marcar con .**todo** (solo texto, sin función) y también aparecerán como skipped.
- El marcador .**only** permite limitar las pruebas que se ejecutan a determinadas suites y especificaciones, el resto se omiten cuando se ejecutan las pruebas y aparecerán como skipped entre los resultados de la prueba. Si una suite .only no tiene ninguna especificación .only se ejecutan todas sus especificaciones, pero si tiene alguna solo ellas se ejecutaran. Una especificación .only se ejecuta siempre aunque su suite este desactivada.

```
describe.skip('Pruebas de expect', () => {
  test.todo('Pendiente...');
  test.only('null', () => {
```


Pruebas parametrizadas

- Las pruebas parametrizadas permiten ejecutar una prueba, suite o especificación, varias veces con diferentes argumentos.
- Se declaran con el marcador .each asociado a una matriz que proporcionará los argumentos para cada invocación, una fila por cada juego de argumentos, a utilizar en los parámetros de la función de prueba.
- Se pueden generar títulos de prueba únicos mediante la inyección posicional de parámetros con formato printf:

```
describe.each([
  [1, 1, 2],  [1, 2, 3],  [2, 1, 3],
])('.add(%i, %i)', (a, b, expected) => {
  test(`returns ${expected}`, () => {
    expect(a + b).toBe(expected);
  });
  test(`returned value not be greater than ${expected}`, () => {
    expect(a + b).not.toBeGreaterThan(expected);
  });
  test(`returned value not be less than ${expected}`, () => {
    expect(a + b).not.toBeLessThan(expected);
  });
});
```

Mock

- Jest tiene funciones dobles de prueba llamados mock, también se conocen como "espías", porque permiten espiar el comportamiento de una función que es llamada indirectamente por algún otro código, en lugar de solo probar la salida.
- Se puede crear una función simulada con jest.fn(). Si no se proporciona una implementación, la función simulada devolverá undefined cuando se invoca.

```
const mockCallback = jest.fn(x => 42 + x);
```

- Las funciones mock también se pueden usar para inyectar valores de retorno simulados durante una prueba:

```
const myMock = jest.fn();
myMock.mockReturnValueOnce(false).mockReturnValueOnce(true);
const result = [11, 12, 15].filter(num => myMock(num));
expect(result).toHaveLength(2); // [12, 15]
expect(result[0]).toBe(12);
```


Valores de retorno

- .mockReturnValue(value) acepta un valor que se devolverá siempre que se llame a la función simulada, pero con .mockReturnValueOnce(value) solo se devolverá el valor para una llamada a la función simulada. Se pueden encadenar para que las sucesivas llamadas a la función simulada devuelvan valores diferentes.

```
const myMockFn = jest.fn()
  .mockReturnValue('default')
  .mockReturnValueOnce('first call')
  .mockReturnValueOnce('second call');
```

- Si solo cuenta con .mockReturnValueOnce, el último fijara el valor a devolver en las siguientes llamadas.
- .mockResolvedValue(value),.mockResolvedValueOnce(value), .mockRejectedValue(value) y .mockRejectedValueOnce(value) son versiones con azúcar sintáctica para devolver promesas.

```
test.only('async test', async () => {
    const asyncMock = jest.fn().mockResolvedValue(43);
    let rslt = await asyncMock(); // 43
    expect(rslt).toBe(43);
});
```

Seguimiento de llamadas

- Todas las funciones simuladas tienen la propiedad especial .mock, que es donde se guardan los datos sobre cómo se ha llamado a la función y qué devolvió. La propiedad .mock también rastrea el valor de this para cada llamada.
  - .calls: Una matriz que contiene arrays con los argumentos de llamada pasados en cada invocación a la función simulada.
  - .results: Una matriz que contiene los resultados de cada llamada: un objeto con las propiedades type (return, throw, incomplete) y value.
  - .instances: Una matriz que contiene todas las instancias que se han creado utilizando la función simulada con new (como función constructora).
- Las propiedades de .mock son muy útiles en las pruebas para afirmar cómo se llaman, instancian o devuelven estas funciones.

```
expect(someMockFunction.mock.calls.length).toBe(1);
expect(someMockFunction.mock.calls[1][0]).toBe('second exec, 'first arg');
expect(someMockFunction.mock.results[0].value).toBe('return value');
expect(someMockFunction.mock.instances.length).toBe(2);
expect(someMockFunction.mock.instances[0].name).toEqual('test');
```


Seguimiento de llamadas

- Hay comparadores (matchers) especiales para interactuar con los espías.
  - .toHaveBeenCalled() pasará si el espía fue llamado.
  - .toHaveBeenCalledTimes(n) pasará si el espía se llama el número de veces especificado.
  - .toHaveBeenCalledWith(…) pasará si la lista de argumentos coincide con alguna de las llamadas grabadas a la espía.
  - .toHaveBeenNthCalledWith(nthCall, arg1, arg2, ...) pasará si n llamada se invoco con determinados argumentos.
  - .toHaveReturned() pasará si el espía devolvió con éxito (es decir, no arrojó un error) al menos una vez.
  - .toHaveReturnedTimes(number) para asegurarse de que el espía devuelve correctamente (es decir, no arroje un error) un número exacto de veces.
  - .toHaveReturnedWith(value) pasará si el espía devolvió un valor específico.
  - .toHaveLastReturnedWith(esperado): pasará si el espía devolvió el valor en la última llamada.
  - .toHaveNthReturnedWith(nthCall, value) para probar el valor específico que devolvió el espía para la enésima llamada.

Cambiar comportamiento

- Un mock puede interceptar cualquier función y hacer un seguimiento a las llamadas y todos los argumentos.

```
let c = new Calc();
const spy = jest.spyOn(c, 'sum');
spy.mockReturnValue(4);
expect(c.sum(1,2)).toBe(4);
expect(spy).toBeCalled();
```

- Se pueden interceptar módulos completos:

```
import axios from 'axios';
import UsersRest from './users';
jest.mock('axios');
test('should fetch users', () => {
  const users = [{name: 'Bob'}];
  axios.get.mockResolvedValue({data: users});
  return UsersRest.getAll().then(data => expect(data).toEqual(users));
});
```


Otras opciones

- Opcionalmente, se puede proporcionar un nombre para las funciones simuladas, que se mostrarán en lugar de "jest.fn ()" en la salida de error de la prueba.

```
const spy = jest.spyOn(c, 'sum').mockName('Mock sum');
```

- jest.isMockFunction(fn) determina si la función dada es una función simulada.
- jest.clearAllMocks() borra las propiedades mock.calls y mock.instances de todos los simulacros.
- jest.resetAllMocks() restablece el estado de todos los simulacros.
- jest.restoreAllMocks() restaura todas las imitaciones a su valor original.
- mockFn.mockClear(), mockFn.mockReset() y mockFn.mockRestore() lo realiza a nivel de funciones simuladas individuales.

Cobertura de código

- Con Jest podemos ejecutar pruebas unitarias y crear informes de cobertura de código. Los informes de cobertura de código nos permiten ver que parte del código ha sido o no probada adecuadamente por nuestras pruebas unitarias.
- Para generar el informe de cobertura:

```
"scripts": {
  :
  "test": "react-scripts test --collectCoverage"
},
```

- Una vez que las pruebas se completen, aparecerá una nueva carpeta /coverage en el proyecto. Si se abre el archivo index.html en el navegador se debería ver un informe con el código fuente y los valores de cobertura del código.
- Usando los porcentajes de cobertura del código, podemos establecer la cantidad de código (instrucciones, líneas, caminos, funciones) que debe ser probado. Depende de cada organización determinar la cantidad de código que deben cubrir las pruebas unitarias.

Cobertura de código

- Para establecer un mínimo de 80% de cobertura de código cuando las pruebas unitarias se ejecuten en el proyecto:

```
"jest": {
  "displayName": "Curso de React",
  "coverageThreshold": {
    "global": { "branches": 80, "functions": 80, "lines": 80, "statements": 80 }
  }
```

- Si se especifican rutas al lado global, los datos de cobertura para las rutas coincidentes se restarán de la cobertura general y los umbrales se aplicarán de forma independiente.

```
"coverageThreshold": {
    "global": { "branches": 50,"functions": 50,"lines": 50,"statements": 50 },
    "./src/components/": { "branches": 40,"statements": 40 },
    "./src/reducers/**/*.js": { "statements": 90 },
    "./src/api/critical.js": { "branches": 100,"functions": 100,"statements": 100 }
  }
```

SUPERTEST

SuperTest

- Este módulo proporciona una abstracción de alto nivel para probar HTTP y, al mismo tiempo, permitirle acceder a la API de nivel inferior proporcionada por el módulo superagente. SuperTest funciona con cualquier marco de prueba. Para su instalación mediante npm:
  - npm install --save-dev supertest @types/supertest
- Puede pasar a request() un http.Server o un función, si el servidor aún no está escuchando las conexiones, entonces está vinculado a un puerto efímero, por lo que no es necesario realizar un seguimiento de los puertos.

```
const request = require('supertest');
const assert = require('assert');
const express = require('express');
const app = express();
app.get('/user', function(req, res) { res.status(200).json({ name: 'john' }); });
request(app).get('/user')
  .expect('Content-Type', /json/)
  .expect(200)
  .end(function(err, res) { if (err) throw err;  });
```

Petición

- Se puede iniciar una solicitud invocando el método apropiado (.get(), .post(), …) en el objeto request al que se le pasa la url de destino y luego llamando a .then()/.catch() o con sintaxis async/await para enviar la solicitud.

```
request.get('/fake')
    .then(res => {
      // res.body, res.headers, res.status
    })
    .catch(err => {
      // err.message, err.response
    });
```

- Con el método final .end() también se inicia la solicitud y admiten devoluciones de llamada al estilo antiguo, pero no se recomiendan. Si lanza un error la expectativa falla.

```
request('GET', '/fake').end(function(err, res) {
    if (err) throw err;
    // res.ok, res.body, res.headers, res.status
});
```


Configurar la petición

- Para configurar los campos del encabezado se invoca .set() con el nombre y el valor de campo:

```
.set('authorization', token)
```

- El método .query() acepta objetos que formarán una cadena de consulta:

```
.query({ page: 1, rows: 20 })
```

- Para rellenar el cuerpo de la solicitud se invoca .send() y se añade el encabezado Content-Type adecuado:

```
.set('Content-Type', 'application/json')
.send({ id: 0, name: "Nuevo" })
```

- El encabezado Content-Type es opcional, si .send() recibe un objeto, lo serializa a JSON y establecerá el Content-Type a application/json. Si recibe una cadena establecerá el Content-Type en application/x-www-form-urlencoded, y se concatenarán varias llamadas con &:

```
.send('id=0')
.send('name=Nuevo')
// id=0&name=Nuevo
```

Negociar contenidos

- Los formatos son extensibles, sin embargo, de forma predeterminada se admiten "json" y "form".
- Como abreviatura, el método .type() también está disponible, aceptando el nombre del tipo MIME completo con tipo/subtipo, o simplemente el nombre de la extensión como "xml", "json", "png", etc.:

```
.type('application/json')
.type('png')
```

- De manera similar al método .type(), también es posible configurar el encabezado Accept mediante el método abreviado .accept(). Lo cual hace referencia request.types y también permite especificar el nombre de tipo MIME completo como type/subtype, o el formato del sufijo de extensión como "xml", "json", "png", etc. para mayor comodidad:

```
.accept('application/json')
.accept('json')
```

- Si se desea enviar la carga útil en un formato personalizado, se puede reemplazar la serialización incorporada con el método .serialize() por solicitud.

Propiedades de respuesta

- La propiedad res.text contiene la cadena del sin analizar.
- Se des serializan automáticamente los datos del cuerpo de la respuesta, actualmente admite application/json, application/x-www-form-urlencoded y multipart/form-data. El objeto analizado estará disponible a través de res.body.
- La propiedad res.header contiene un objeto de campos de encabezado analizados, nombres de campos en minúsculas como lo hace Node.

```
res.header['content-length']
```

- El encabezado de respuesta Content-Type está en mayúsculas y minúsculas especiales, deja en res.type el formato MIME y en res.charset el juego de caracteres (si lo hay). Por ejemplo, el tipo de contenido "text/html; charset=utf8" proporcionará "text/html" como res.type y la propiedad res.charset propiedad contendrá "utf8".
- La propiedad res.status contiene el código de estado de la respuesta.

Expectativas

- Los métodos .expect() permiten establecer las afirmaciones que debe cumplir la respuesta. Una vez falla una no se evalúan el resto. Las expectativas se ejecutan en el orden de definición, esta característica se puede utilizar para modificar el cuerpo de la respuesta o los encabezados antes de ejecutar una aserción.

  .expect(status[, fn]): Establecer el código de estado esperado.

  ```
  .expect(200)
  ```

  .expect(field, value[, fn]): Establecer el valor esperado en una cabecera.

  ```
  .expect('Content-Type', /json/)
  ```

  .expect(body[, fn]): Establecer los valores esperados en el cuerpo.

  ```
  .expect({ id: 0, name: "Nuevo" })
  ```

  .expect(status, body[, fn]): Establecer el código de estado y los valores en el cuerpo esperados.

Expectativas

- El método .expect(function(res) {}) recibe la función que realiza la verificación, se le pasa el objeto de respuesta y lanzara un error para que la expectativa falle. También permite modificar la respuesta para las siguientes expectativas.

```
.expect(res => {
  if (res.status !== 200) throw new Error("error data");
  if (!res.body.data) throw new Error("missing data");
  if (!res.body.data.length) throw new Error("empty data");
})
```

- Los métodos .expect() reciben la función done cuando el caso de prueba trabaja con promesas:

```
it('responds with json', done => {
   request(app).get('/fake')
     .expect('Content-Type', /json/)
     .expect(200, done);
});
```

Casos de prueba

```
it('responds with json, inject done', done => {
   request(app).get('/fake')
     .then(res => {
       expect(res.header['content-length']).toContain('json')
       expect(res.status).toBe(200)
       done()
     }).catch(err => done(err))
});
it('responds with json, return promise', () => {
   return request(app).get('/fake')
     .then(res => {
       expect(res.header['content-length']).toContain('json')
       expect(res.status).toBe(200)
     })
});
it('responds with json, async/await', async () => {
   const res = await request(app).get('/fake')
   expect(res.header['content-length']).toContain('json')
   expect(res.status).toBe(200)
});
```


PRUEBAS E2E

Introducción

- A medida que las aplicaciones crecen en tamaño y complejidad, se vuelve imposible depender de pruebas manuales para verificar la corrección de las nuevas características, errores de captura y avisos de regresión.
- Las pruebas unitarias son la primera línea de defensa para la captura de errores, pero a veces las circunstancias requieran la integración entre componentes que no se pueden capturar en una prueba unitarias.
- Algunas pruebas deben tener una vista de pájaro de alto nivel de la aplicación. Simulan a un usuario interactuando con la aplicación: navegando a una dirección, leyendo texto, haciendo clic en un enlace o botón, llenando un formulario, moviendo el mouse o escribiendo en el teclado. Estas pruebas generan las expectativas sobre lo que el usuario ve y lee en el navegador.
- Desde la perspectiva del usuario, no importa como esté la aplicación implementada. Los detalles técnicos como la estructura interna de su código no son relevantes. No hay distinción entre front-end y back-end, entre partes del código. Se prueba la experiencia completa.
- Estas pruebas se denominan pruebas de extremo a extremo (E2E) ya que integran todas las partes de la aplicación desde un extremo (el usuario) hasta el otro extremo (los rincones más oscuros del back-end). Las pruebas de extremo a extremo también forman la parte automatizada de las pruebas de aceptación, ya que indican si la aplicación funciona para el usuario.

Ventajas e inconvenientes

- Las pruebas de extremo a extremo han sido ampliamente adoptadas porque:
  - Ayuda a los equipos a expandir su cobertura de pruebas agregando casos de pruebas más detallados que otros métodos de prueba, como pruebas unitarias y funcionales.
  - Garantiza que la aplicación funcione correctamente ejecutando los casos de prueba en función del comportamiento del usuario final.
  - Ayuda a los equipos de lanzamiento a reducir el tiempo de paso a producción al permitirles automatizar las rutas críticas de los usuarios.
  - Reduce el costo general de creación y mantenimiento del software al disminuir el tiempo que lleva probar el software.
  - Ayuda a detectar errores de manera predecible y confiable, aunque no de forma concluyente.
- Los inconvenientes se presentan porque las pruebas de extremo a extremo:
  - Toman mucho tiempo y son frágiles.
  - Deben estar diseñadas para reproducir escenarios del mundo real.
  - Requiere una buena comprensión de los objetivos del usuario: Los usuarios no buscan características, buscan resolver sus problemas específicos.
- Las pruebas de extremo a extremo requieren un grupo multidisciplinario que incluye desarrolladores, probadores, administradores y usuarios.

Estrategias de automatización

- **Record-and-replay (R&R):** Se orientan a la grabación de pruebas exploratorias sobre el GUI que luego se pueden reproducir automáticamente. La fase de grabación (recording) incluye aserciones de verificación y genera un script con las instrucciones a reproducir. El script puede estar en un lenguaje propio de la herramienta usada o puede tener instrucciones de un API para automatización. Tienen el beneficio que son fáciles de aprender, grabar y reproducir, sin embargo, pero son difíciles de mantener o personalizar al estar limitadas por la herramienta utilizada.
- **APIs de automatización:** Existen librerías y frameworks que permiten la automatización de pruebas de GUI mediante código, a través de APIs que permiten controlar externamente el GUI. Si bien se benefician de un mayor control y flexibilidad, tienen una curva de aprendizaje mucho mayor, deben ocuparse de los detalles, como los tiempos de espera, y dependen de un código que hay que escribir y mantener, mucho mas costoso que una grabación.

Conceptos a tener en cuenta

- Flujo funcional:
  - Proceso en el que se define un comportamiento a partir de unas entradas, y un tratamiento de esas entradas por parte de todos los sistemas o componentes, hasta llegar a un resultado final.
- Camino crítico:
  - En un flujo funcional participan normalmente muchos sistemas o componentes. Sería un error pretender hacer la validación del todo, porque por una parte normalmente no hay tiempo material para hacerlo, y por otra, muchos de los sistemas no tienen relevancia. El camino crítico lo componen sólo los sistemas a validar en la prueba E2E.
- Pruebas horizontales o verticales:
  - Las horizontales cubren aspectos de toda la aplicación mientras que las pruebas verticales cubren un único aspecto.

Proceso de Prueba

- Identifica con qué navegadores probar
- Elige el mejor lenguaje/entorno para ti y tu equipo.
- Configura entorno para que funcione con cada navegador que te interese.
- Descompón la aplicación web existente para identificar qué probar
- Escribe pruebas mantenibles y reutilizables que sean compatibles y ejecutables en todos los navegadores.
- Crea un circuito de retroalimentación integrado para automatizar las ejecuciones de prueba y encontrar problemas rápidamente.
- Configura tu propia infraestructura o conéctate a un proveedor en la nube.
- Mejora drásticamente los tiempos de prueba con la paralelización
- Mantente actualizado en el mundo de las herramientas de pruebas.

Selenium Webdriver

INTRODUCCIÓN

Introducción

- El Selenium es un conjunto de herramientas para automatizar los navegadores web, robot que simula la interacción del usuario con el navegador, originalmente pensado como entorno de pruebas de software para aplicaciones basadas en la web.
- Como principales herramientas Selenium cuenta con:
  - Selenium IDE:
    - una herramienta para grabar y reproducir secuencias de acciones con el navegador que permite crear pruebas sin usar un lenguaje de scripting para pruebas.
  - Selenium Core:
    - API para escribir pruebas automatizadas y de regresión en un amplio número de lenguajes de programación populares incluyendo Java, C#, Ruby, Groovy, Perl, Php y Python.
  - WebDriver:
    - interfaces que permite ejecutar las pruebas de forma nativa usando la mayoría de los navegadores web modernos en diferentes sistemas operativos como Windows, Linux y OSX.
  - Selenium Grid:
    - Permite ejecutar muchas pruebas de un mismo grupo en paralelo o pruebas en múltiples entornos. Tiene la ventaja que un conjunto de pruebas muy grande puede dividirse en varias maquinas remotas para una ejecución más rápida o si se necesitan repetir las mismas pruebas en múltiples entornos.

Historia

- Cuando Selenium 1 fue lanzado en el año 2004, surgió por la necesidad de reducir el tiempo dedicado a verificar manualmente el comportamiento consistente en el front-end de una aplicación web. Hizo uso de las herramientas disponibles en ese momento y confió en gran medida en la inyección de JavaScript en la página web bajo prueba para emular la interacción de un usuario.
- Si bien JavaScript es una buena herramienta para permitirte la introspección de las propiedades del DOM y para hacer ciertas observaciones del lado del cliente que de otro modo no se podría hacer, se queda corto en la capacidad de replicar naturalmente las interacciones de un usuario como usar el mouse y el teclado.
- Desde entonces, Selenium ha crecido y ha madurado bastante, convirtiéndose en una herramienta ampliamente utilizada. Selenium ha pasado de ser de un kit de herramientas de automatización de pruebas de fabricación casera a la librería de facto de automatización de navegadores del mundo.
- Así como Selenium RC hizo uso de las herramientas de oficio disponibles en ese momento, Selenium WebDriver impulsa esta tradición al llevar la parte de la interacción del navegador al territorio del proveedor del mismo y pedirles que se responsabilicen de las implementaciones de back-end orientadas al navegador. Recientemente este esfuerzo se ha convertido en un proceso de estandarización del W3C donde el objetivo es convertir el componente WebDriver en Selenium en la librería de control remoto du jour para agentes de usuario.

Selenium controla los navegadores web

- Selenium significa muchas cosas pero en su núcleo, es un conjunto de herramientas para la automatización de navegadores web que utiliza las mejores técnicas disponibles para controlar remotamente las instancias de los navegadores y emular la interacción del usuario con el navegador.
- Permite al código simular interacciones básicas realizadas por los usuarios finales; insertando texto en los campos, seleccionando valores de menús desplegables y casillas de verificación, y haciendo clics en los enlaces de los documentos. También provee muchos otros controles tales como el movimiento del mouse, la ejecución arbitraria de JavaScript y mucho más.
- A pesar de que es usado principalmente para pruebas de front-end de sitios webs, Selenium es en esencia una librería de agente de usuario para el navegador. Las interfaces son ubicuas a su aplicación, lo que fomenta la composición con otras librerías para adaptarse a su propósito.

Un API para gobernarlos a todos

- Uno de los principios fundamentales del proyecto es permitir una interfaz común para todas las tecnologías de los (principales) navegadores.
- Los navegadores web son aplicaciones increíblemente complejas y de mucha ingeniería, realizando operaciones completamente diferentes pero que usualmente se ven iguales al hacerlo. Aunque el texto se presente con las mismas fuentes, las imágenes se muestren en el mismo lugar y los enlaces te llevan al mismo destino.
- Lo que sucede por debajo es tan diferente como la noche y el día. Selenium abstrae estas diferencias, ocultando sus detalles y complejidades a la persona que escribe el código. Esto le permite escribir varias líneas de código para realizar un flujo de trabajo complicado, pero estas mismas líneas se ejecutarán en Firefox, Internet Explorer, Chrome y los demás navegadores compatibles.

Herramientas y soporte

- El diseño minimalista de Selenium le da la versatilidad para que se pueda incluir como un componente en otras aplicaciones. La infraestructura proporcionada debajo del catálogo de Selenium te da las herramientas para que puedas ensamblar un grid de navegadores donde las pruebas se puedan ejecutar en diferentes navegadores a través de diferente sistemas operativos y plataformas.
- Imagina un granja de servidores en tu sala de servidores o en un centro de datos, todos ejecutando navegadores al mismo tiempo e interactuando con los enlaces en tu sitio web, formularios y tablas—probando tu aplicación 24 horas al día. A través de la sencilla interfaz de programación proporcionada para los lenguajes más comunes, estas pruebas se ejecutarán incansablemente en paralelo, reportando cuando ocurran errores.
- Es un objetivo ayudar a que esto sea una realidad para ti, proporcionando a los usuarios herramientas y documentación para controlar no solo los navegadores pero también para facilitar la escalabilidad e implementación de tales grids.

Automatización de pruebas

- Las pruebas funcionales de usuario final, como las pruebas de Selenium son caras de ejecutar, requieren abrir un navegador e interactuar con el. Además, normalmente requieren que una infraestructura considerable este disponible para estas ejecutarse de manera efectiva.
- Es una buena regla preguntarse siempre si lo que se quiere probar puede hacerse usando enfoques de prueba más livianos como las pruebas unitarias o con un enfoque de bajo nivel.
- Las pruebas manuales son muy costosas y difícilmente repetibles, por lo que se impone una estrategia de automatización. Selenium permite ejecutar y repetir las mismas pruebas en múltiples navegadores de diferentes sistemas operativos.
- Una vez tomada la decisión de hacer pruebas en el navegador, y que tengas tu ambiente de Selenium listo para empezar a escribir pruebas, generalmente realizaras alguna combinación de estos tres pasos:
  - Preparar los datos
  - Realizar un conjunto discreto de acciones
  - Evaluar los resultados
- Querrás mantener estos pasos tan cortos como sea posible; una o dos operaciones deberían ser suficientes la mayor parte del tiempo. La automatización del navegador tiene la reputación de ser “frágil”, pero en realidad, esto se debe a que los usuarios suelen exigirle demasiado.
- Manteniendo las pruebas cortas y usando el navegador web solo cuando no tienes absolutamente ninguna alternativa, puedes tener muchas pruebas con fragilidad mínima.
- Una clara ventaja de las pruebas de Selenium es su capacidad inherente para probar todos los componentes de la aplicación, desde el backend hasta el frontend, desde la perspectiva del usuario. En otras palabras, aunque las pruebas funcionales pueden ser caras de ejecutar, también abarcan a la vez grandes porciones críticas para el negocio.

Tipos de pruebas

- **Pruebas funcionales**: Este tipo de prueba se realiza para determinar si una funcionalidad o sistema funciona correctamente y sin problemas. Se comprueba el sistema en diferentes niveles para garantizar que todos los escenarios están cubiertos y que el sistema hace lo que se supone que debe de hacer. Es una actividad de verificación que responde la pregunta: ¿Estamos construyendo el producto correctamente?.
  - Para aplicaciones web, la automatización de esta prueba puede ser hecha directamente con Selenium simulando los retornos esperados. Esta simulación podría hacerse mediante grabación/reproducción o mediante los diferentes lenguajes soportados.
- **Pruebas de aceptación**: Este tipo de prueba se realiza para determinar si una funcionalidad o un sistema cumple con las expectativas y requerimientos del cliente. Este tipo de pruebas implican la cooperación o retroalimentación del cliente, siendo una actividad de validación que responde la pregunta: ¿Me están construyendo el producto correcto?.
  - Las pruebas de aceptación son un subtipo de pruebas funcionales, por lo que la automatización se puede hacer directamente con Selenium simulando el comportamiento esperado del usuario mediante grabación/reproducción.

Tipos de pruebas

- **Pruebas de regresión**: Este tipo de pruebas generalmente se realiza después de un cambio, corrección o adición de funcionalidad.
  - Para garantizar que el cambio no ha roto ninguna de las funcionalidades existentes, algunas pruebas ya ejecutadas se ejecutan nuevamente. El conjunto de pruebas ejecutadas nuevamente puede ser total o parcial, y puede incluir varios tipos diferentes, dependiendo del equipo de desarrollo y la aplicación.
- Las **pruebas no funcionales** tales como rendimiento, de carga, de estrés, … aunque se pueden realizar con Selenium hay otras opciones, como JMeter, que las realizan mas eficientemente y además suministran métricas que incluyen rendimiento, latencia, pérdida de datos, tiempos de carga de componentes individuales.
- De igual forma, las **pruebas unitarias** se pueden realizar con Selenium pero, como se deben ejecutar continuamente, suele ser demasiado costoso.

Proceso de Prueba

- Descomponer la aplicación web existente para identificar qué probar.
- Identificar con qué navegadores probar.
- Elige el mejor lenguaje para ti y tu equipo.
- Configura Selenium para que funcione con cada navegador que te interese.
- Escriba pruebas de Selenium mantenibles y reutilizables que serán compatibles y ejecutables en todos los navegadores.
- Crea un circuito de retroalimentación integrado para automatizar las ejecuciones de prueba y encontrar problemas rápidamente.
- Configura tu propia infraestructura o conéctate a un proveedor en la nube.
- Mejora drásticamente los tiempos de prueba con la paralelización.
- Mantente actualizado en el mundo Selenium.

Que probar

- La navegación
- La interacciones con la página y sus elementos
- El rellenado de formularios y sus validaciones
- El arrastrar y soltar, si procede
- La estética, presencia y visualización de elementos esenciales, con especial atención al responsive design.
- Moverse entre ventanas y marcos (aunque están prohibidos por la WAI)
- Uso de cookies, Local Storage, Session Storage, Service Worker, …

INSTALACIÓN

Drivers

- Instalar los WebDriver:
  - Crear una carpeta y referenciarla en el PATH del sistema.
    - Windows: *setx /m path "%path%;C:\Selenium\WebDriver\"*
    - macOS y Linux: *export PATH=$PATH:/opt/Selenium/WebDriver >> ~/.profile*
  - Descargar los drivers (https://www.selenium.dev/downloads/)
  - Copiarlos descomprimidos en la carpeta creada.
- Plataformas compatibles con Selenium:
  - Firefox: GeckoDriver está implementado y soportado por Mozilla.
  - Internet Explorer: Solo se admite la versión 11 y requiere una configuración adicional.
  - Safari: SafariDriver es compatible directamente con Apple.
  - Ópera: OperaDriver es compatible con Opera Software.
  - Chrome: ChromeDriver es compatible con el proyecto Chromium.
  - Edge: EdgeDriver está implementado y soportado por Microsoft.

Local

- Instalación de Eclipse.
- Descargar y descomprimir API: https://www.selenium.dev/downloads/
- Creación del proyecto:
  - Create a Java Project
    - Add Library Junit 5
    - Add External JARs: API descomprimida
  - Crear paquete de tests
    - Añadir JUnit Test Case
  - Ejecutar test
    - Run As → JUnit Test Case.

Maven

- Instalación de Eclipse.
- Creación del proyecto:
  - Create a New Maven Project
    - Create a simple project (skip archetype selection)
  - Editar pom.xml
    - Añadir dependencias
  - Seleccionar src/test/java
    - Añadir JUnit Test Case
  - Ejecutar test
    - Run As → JUnit Test Case.

pom.xml

```
<dependency>
	<groupId>org.junit.jupiter</groupId>
	<artifactId>junit-jupiter</artifactId>
	<version>${junit.jupiter.version}</version>
	<scope>test</scope>
</dependency>
<dependency>
	<groupId>org.seleniumhq.selenium</groupId>
	<artifactId>selenium-java</artifactId>
	<version>${selenium.version}</version>
	<scope>test</scope>
</dependency>
```

Dependencias únicas

- La dependencia selenium-java permite la ejecución del proyecto de automatización en todos los navegadores compatibles con Selenium.
- Si solo es necesario ejecutar las pruebas en un navegador en específico, se puede sustituir dependencia selenium-java por la dependencia para ese navegador en el archivo pom.xml. Por ejemplo, se debe agregar la siguiente dependencia en el archivo pom.xml para ejecutar las pruebas solamente en Chrome:

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-chrome-driver</artifactId>
  <version>3.X</version>
</dependency>
```


SELENIUM IDE

Introducción

- Es el entorno de desarrollo integrado para pruebas con Selenium que permite grabar, editar y depurar fácilmente las pruebas. Es una solución simple y llave en mano para crear rápidamente pruebas de extremo a extremo confiables.
- Selenium IDE no requiere una configuración adicional aparte de instalar una extensión en el navegador pero solo está disponible para Firefox y Chrome.
- Selenium IDE es muy flexible, registra múltiples localizadores para cada elemento con el que interactúa, si un localizador falla durante la reproducción, los demás se probarán hasta que uno tenga éxito.
- Mediante el uso del comando de ejecución se puede reutilizar un caso de prueba dentro de otro.
- Dispone una estructura de flujo de control extensa, con comandos condicionales, bucles, … que permite modelizar escenarios complejos.

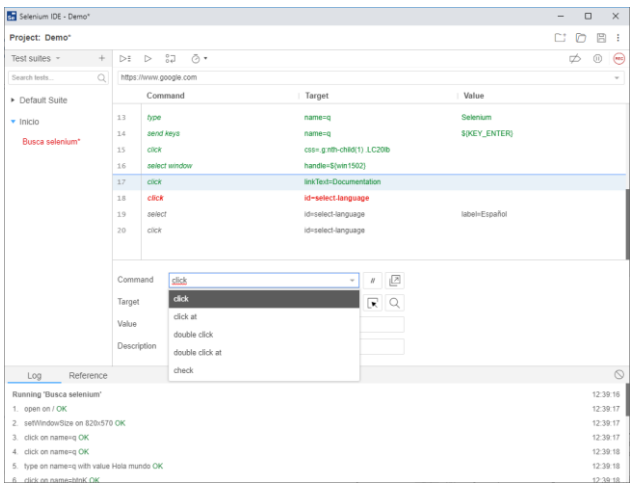
Características

- Dispone de una selección inteligente de campos usando ID, nombre, Xpath o DOM según se necesite.
- Para la depuración permite la configuración de los puntos de interrupción, iniciar y detener la ejecución de un caso de prueba desde cualquier punto dentro del caso de prueba e inspeccionar la forma en el caso de prueba se comporta en ese punto.
- Permite exportar los casos de prueba a Java, C# y Ruby, actuando como embriones en la creación de los casos de prueba para WebDriver.
- Selenium IDE dispone de un amplio conjunto de extensiones adicionales que ayudan o simplifican la elaboración de los casos de pruebas.

Empezando

- Instale Selenium IDE desde la tienda web Chrome o Firefox .
- Una vez instalado, inícielo haciendo clic en su icono desde la barra de menú de su navegador.
- Aparecerá un cuadro de diálogo de bienvenida con las siguientes opciones:
  - Grabar una nueva prueba en un nuevo proyecto
  - Abrir un proyecto existente
  - Crea un nuevo proyecto
  - Cierra el IDE
- Si se da grabar, pedirá el nombre del proyecto y la URL base de la aplicación que se está probando (se utiliza en todas las pruebas del proyecto).
- Después de completar esta configuración, se abrirá la ventana IDE y una nueva ventana del navegador, cargará la URL base y comenzará a grabar. Interactúe con la página y cada una de las acciones se registrará en el IDE. Para detener la grabación, cambie a la ventana IDE y haga clic en el icono de grabación.

Empezando



Empezando

- La grabación se puede editar, modificar, reproducir, depurar, guardar y exportar.
- Cada grabación es una prueba. El proyecto puede contener múltiples pruebas que se organizan agrupándolas en suites. Por defecto se crea la “Default Suite” a la que se le asigna la prueba de la grabación inicial. Las pruebas y las suites se pueden renombrar, ejecutar o eliminar.
- El proyecto se puede descartar o guardar un único archivo JSON con una extensión .side para recuperarlo posteriormente o ejecutarlo en la línea de comandos.
- La reproducción de la prueba en el IDE permite establecer puntos de ruptura e inspeccionar valores.
- Se puede reanudar la grabación en cualquier punto.

Scripts

- Se pueden desarrollar automáticamente scripts al crear una grabación y de esa manera se puede editar manualmente con sentencias y comandos para que la reproducción de nuestra grabación sea correcta
- Los scripts se generan en un lenguaje de scripting especial para Selenium a menudo denominado Selanese.
- Selanese provee comandos que dicen al Selenium que hacer y pueden ser:
  - **Acciones**: son comandos que generalmente manipulan el estado de la aplicación, ejecutan acciones sobre objetos del navegador, como hacer click en un enlace, escribir en cajas de texto o seleccionar de una lista de opciones. Muchas acciones pueden ser llamadas con el sufijo "AndWait“ que indica la acción hará que el navegador realice una llamada al servidor y que se debe esperar a una nueva página se cargue.
  - **Descriptores de acceso**: examinan el estado de la página y almacenan los resultados en variables.
  - **Aserciones**: son como descriptores de acceso, pero las muestras confirman que el estado de la solicitud se ajusta a lo que se esperaba, verifican la presencia de un texto en particular o la existencia de elementos.

Localizadores

- Localizar por Id:
  - id=loginForm
- Localizar por Name
  - name=username
- Localizar por XPath
  - xpath=//form[@id='loginForm']
- Localizar por el texto en los hipervínculos
  - link=Continue
- Localizar por CSS
  - css=input[name="username"]

Acciones

- Sobre elementos de los formularios:
  - click, click at, double click, double click at, check, uncheck, edit content, send keys, type, select, add selection, remove selection, submit
- Del ratón:
  - mouse move at, mouse over, mouse out, mouse down, mouse down at, mouse up, mouse up at, drag and drop to object
- Sobre cuadros de dialogo alert, prompt y confirm:
  - choose ok on next confirmation, choose cancel on next confirmation, choose cancel on next prompt, webdriver choose ok on visible confirmation, webdriver choose cancel on visible confirmation, webdriver choose cancel on visible prompt
- Sobre la ejecución de la prueba:
  - open, execute script, run, echo, debugger, pause, close

Variables

- Se puede usar variable en Selenium para almacenar constantes al principio de un script. Además, cuando se combina con un diseño de prueba controlado por datos, las variables de Selenium se pueden usar para almacenar valores pasados a la prueba desde la línea de comandos, desde otro programa o desde un archivo.
  - store target:valor value:varName
- Para acceder al valor de una variable:
  - ${userName}
- Hay métodos diponibles para recuperar información de la página y almacenarla en variables:
  - storeAttribute, storeText, storeValue, storeTitle, storeXpathCount

Flujo de control

- Selenium IDE viene con comandos que permiten agregar lógica condicional y bucles a las pruebas, para ejecutar comandos (o un conjunto de comandos) solo cuando se cumplen ciertas condiciones o repetidamente en función de criterios predefinidos.
- Las condiciones en su aplicación se verifican mediante el uso de expresiones de JavaScript.
- Los comandos de flujo de control Control Flow funcionan son comandos de apertura y cierre para denotar un conjunto (o bloque) de comandos.
- Aquí están cada uno de los comandos de flujo de control disponibles acompañados de sus comandos complementarios de cierre.
  - if … else if … else … end // else if y else son opcionales
  - times … end // Bucle for
  - forEach … end
  - while … end
  - do … repeat if // la condicion en repeat if
- Se pueden anidar los comandos de control de flujo según sea necesario.

Afirmar y Verificar

- Una "afirmación" hará fallar la prueba y abortará el caso de prueba actual, mientras que una "verificación" hará fallar la prueba pero continuará ejecutando el caso de prueba.
  - Tiene muy poco sentido para comprobar que el primer párrafo de la página sea el correcto si la prueba ya falló al comprobar que el navegador muestra la página esperada. Por otro lado, es posible que desee comprobar muchos atributos de una página sin abortar el caso de prueba al primer fallo, ya que esto permitirá revisar todos los fallos en la página y tomar la acción apropiada.
- Selenese permite múltiples formas de comprobar los elementos de la interfaz de usuario pero hay que decidir el métodos mas apropiado:
  - ¿Un elemento está presente en algún lugar de la página?
  - ¿El texto especificado está en algún lugar de la página?
  - ¿El texto especificado está en una ubicación específica en la página?
- Métodos:
  - assert, assertAlert, assertChecked, assertNotChecked, assertConfirmation, assertEditable, assertNotEditable, assertElementPresent, assertElementNotPresent, assertPrompt, assertSelectedValue, assertNotSelectedValue, assertSelectedLabel, assertText, assertNotText, assertTitle, assertValue
  - verify, verifyChecked, verifyNotChecked, verifyEditable, verifyNotEditable, verifyElementPresent, verifyElementNotPresent, verifySelectedValue, verifyNotSelectedValue, verifyText, verifyNotText, verifyTitle, verifyValue, verifySelectedLabel

Ejecutar en línea de comandos

- Requiere tener instalado NodeJS (https://nodejs.org)
- Instalar CLI
  - npm install -g selenium-side-runner chromedriver edgedriver geckodriver
- Instalar los WebDriver:
  - Crear una carpeta y referenciarla en el PATH del sistema.
  - Descargar los drivers (https://www.selenium.dev/downloads/)
  - Copiarlos a la carpeta creada.
- Para ejecutar las suites de pruebas:
  - selenium-side-runner project.side project2.side *.side
- Para ejecutar en diferentes navegadores:
  - selenium-side-runner *.side -c "browserName=Chrome"
  - selenium-side-runner *.side -c "browserName=firefox"

Exportación

- Se puede exportar una prueba o un conjunto de pruebas a código de WebDriver haciendo clic con el botón derecho en una prueba o un conjunto, seleccionando Export, eligiendo el idioma de destino y haciendo clic Export.
- Actualmente, se admite la exportación a los siguientes idiomas y marcos de prueba.
  - C# NUnit
  - C# xUnit
  - Java JUnit
  - JavaScript Mocha
  - Python pytest
  - Ruby RSpec
- Esta previsto admitir todos los lenguaje de programación soportados oficialmente para Selenium en al menos un marco de prueba para cada lenguaje.

WebDriver

```
@BeforeClass
public static void setUpClass() throws Exception {
        System.setProperty("webdriver.chrome.driver", "C:/Archivos/.../chromedriver.exe");
}
@Before
public void setUp() throws Exception {
  driver = new ChromeDriver();
  baseUrl = "http://localhost/";
  driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}
@Test
public void testLoginOK() throws Exception {
  driver.get(baseUrl + "/login.php");
  driver.findElement(By.id("login")).sendKeys("admin");
  driver.findElement(By.id("password")).sendKeys("admin");
  driver.findElement(By.cssSelector("input[type=\"submit\"]")).click();
  try {
    assertEquals("", driver.findElement(By.cssSelector("img[title=\"Main Menu\"]")).getText());
  } catch (Error e) {
    verificationErrors.append(e.toString());
  }
}
```

Maven

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>3.13.0</version>
</dependency>
```

WEBDRIVER

WebDriver

- Selenium permite la automatización de todos los principales navegadores del mercado mediante el uso de WebDriver.
- WebDriver es una API y un protocolo que define una interfaz de idioma neutral para controlar el comportamiento de los navegadores web. Cada navegador está respaldado por un Driver, una implementación específica de WebDriver, llamada controlador. El controlador es el componente responsable de delegar en el navegador, y maneja la comunicación entre Selenium y el navegador.
- Esta separación es parte de un esfuerzo consciente para hacer que los proveedores de navegadores asuman la responsabilidad de la implementación para sus navegadores. Selenium utiliza estos controladores de terceros cuando es posible, pero también proporciona sus propios controladores mantenidos por el proyecto para los casos en que esto no es una realidad.
- El framework de Selenium unifica todas estas piezas a través de una interfaz orientada al usuario que habilita que los diferentes backends de los navegadores sean utilizados de forma transparente, permitiendo la automatización cruzada entre navegadores y plataformas diferentes.

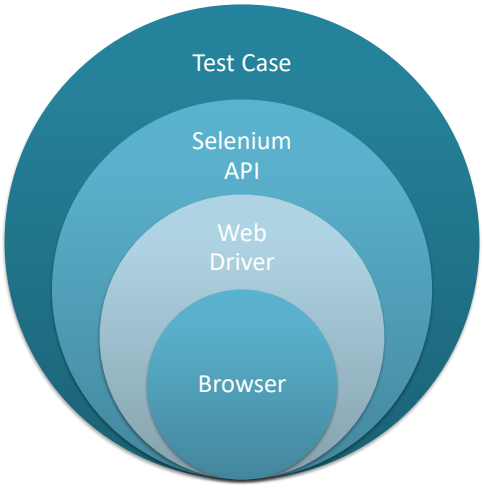
WebDriver

- WebDriver controla un navegador de forma nativa, como lo haría un usuario, ya sea localmente o en una máquina remota utilizando el servidor Selenium, marca un salto adelante en términos de automatización de navegadores.
- Selenium WebDriver se refiere tanto a los enlaces de lenguajes como también a las implementaciones individuales del código controlador del navegador. Esto se conoce comúnmente solo como WebDriver.
- Selenium WebDriver es una Recomendación W3C (https://www.w3.org/TR/webdriver1/)
  - WebDriver está diseñado como una interfaz de programación simple y más concisa.
  - WebDriver es una API compacta orientada a objetos.
  - Controla el navegador de manera efectiva.

Componentes

- API: Interfaz de Programación de Aplicaciones. Es un conjunto de “comandos” que se utilizan para manipular el WebDriver.
- Library: Un módulo de código que contiene las APIs y el código necesario para implementarlos. Las librerías son específicas para cada lenguaje, por ejemplo ficheros .jar en Java, ficheros .dll para .NET, etc.
- Driver: El responsable de controlar un navegador concreto (proxies). La mayoría de los drivers son creados por los fabricantes del navegador. Los Drivers son generalmente módulos ejecutables que corren en el sistema con el propio navegador, no en el sistema ejecutando la suite de test.
- Remote WebDrive: Permite controlar instancias remotas de navegadores web.
- Framework: Una librería adicional usada como un soporte para la suites de WebDriver. Estos frameworks pueden ser test frameworks como JUnit o NUnit. También pueden ser frameworks soportando lenguaje natural como Cucumber o Robotium. Los frameworks también pueden ser escritos y usados para cosas como la manipulación o configuración del sistema bajo la prueba, creación de datos, test oracles, etc

Componentes



Navegadores

El framework de Selenium soporta oficialmente los siguientes navegadores:

Navegador	Proveedor	Versiones Soportadas
Chrome	Chromium	Todas las Versiones
Firefox	Mozilla	54 y más recientes
Internet Explorer	Selenium	6 y más recientes
Opera	Opera Chromium / Presto	10.5 y más recientes
Safari	Apple	10 y más recientes

También hay un conjunto de navegadores especializados utilizados típicamente en entornos de desarrollo.

Controlador	Propósito	Mantenedor
HtmlUnitDriver	Emulador de navegador headless respaldado por Rhino	Proyecto Selenium

Navegadores simulados

- HtmlUnit
  - HtmlUnit es un navegador sin interfaz grafica para programas basados en Java. Modela documentos HTML y proporciona un API que permite invocar las paginas, rellanar formularios, hacer clic en enlaces, etc. Soporta JavaScript y es capaz de funcionar con librerías AJAX, simulando Chrome, Firefox o Internet Explorer dependiendo de la configuración usada.
- PhantomJS
  - PhantomJS es un navegador sin interfaz grafica basado en Webkit, aunque es una versión mucho mas antigua que las usadas por Chrome o Safari. El proyecto esta sin soporte desde que se anunció que Chrome y Firefox tendrían la capacidad de ser navegadores sin interfaz grafica.

Instalación

- La gran mayoría de controladores necesitan de un ejecutable extra para que Selenium pueda comunicarse con el navegador. Para ejecutar tu proyecto y controlar el navegador, debes tener instalados los binarios de WebDriver específicos para el navegador.
- Crea un directorio para almacenar los ejecutables en el, como C:\WebDriver\bin o /opt/WebDriver/bin. A través de los diferentes proveedores, descargar y descomprimir en la carpeta recién creada.
- Se puede especificar manualmente donde esta ubicado el ejecutable antes lanzar el WebDriver, pero esto hará que los tests sean menos portables, ya que los ejecutables necesitan estar en el mismo lugar en todas las maquinas. Si la ruta de la carpeta está incluida en el PATH no es necesario especificarla en cada test:
  - Windows: setx /m path "%path%;C:\WebDriver\bin\"
  - macOS y Linux: export PATH=$PATH:/opt/WebDriver/bin >> ~/.profile
- Para probar los cambios, en una terminal de comando escribe el nombre de uno de los binarios que has añadido en la carpeta en el paso previo:
  - chromedriver

Cargar el driver

- Importar el espacio de nombres de WebDriver.
  ```
  import org.openqa.selenium.WebDriver;
  ```
- Si la ruta a los dirvers está en el PATH no debe fijarse con System.setProperty.
- Chromium/Chrome
  ```
  import org.openqa.selenium.chrome.ChromeDriver;
  System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
  WebDriver driver = new ChromeDriver();
  ```
- Firefox
  ```
  import org.openqa.selenium.Firefox.FirefoxDriver;
  System.setProperty("webdriver.gecko.driver", "/path/to/geckodriver");
  WebDriver driver = new FirefoxDriver();
  ```
- Edge
  ```
  import org.openqa.selenium.edge.EdgeDriver;
  System.setProperty("webdriver.edge.driver", "C:/path/to/MicrosoftWebDriver.exe");
  WebDriver driver = new EdgeDriver();
  ```
- Internet Explorer
  ```
  import org.openqa.selenium.ie.InternetExplorerDriver;
  System.setProperty("webdriver.ie.driver", "C:/path/to/IEDriver.exe");
  WebDriver driver = new InternetExplorerDriver();
  ```


Estrategia de carga de página

- pageLoadStrategy permite los siguientes valores:
  - NORMAL (por defecto) : Esto hará que WebDriver espere a que se cargue toda la página. Cuando se establece en NORMAL, WebDriver espera hasta que se dispare el evento load y concluya.
  - EAGER: Esto hará que WebDriver espere hasta que el documento HTML inicial se haya cargado y analizado por completo, descartando la carga de hojas de estilo, imágenes y sub marcos. Cuando se establece en EAGER, WebDriver espera hasta que se dispare el evento DOMContentLoaded y concluya.
  - NONE: Esto hará que WebDriver solo espera hasta que se descargue la página inicial.

```
ChromeOptions chromeOptions = new ChromeOptions();
chromeOptions.setPageLoadStrategy(PageLoadStrategy.EAGER);
WebDriver driver = new ChromeDriver(chromeOptions);
```

Aislar las pruebas

- Hay muchos recursos externos que se cargan en una página que no son directamente relevantes para la funcionalidad que está probando (por ejemplo, widgets de Facebook, Analytics, fragmentos de JavaScript, CDN, etc.). Estos recursos externos tienen el potencial de impactar negativamente las ejecuciones de prueba debido a tiempos de carga lentos.
- Entonces, ¿cómo proteger nuestras pruebas de estas cosas que están fuera de su control?
- Una posible solución es utilizar un servidor proxy en nuestras pruebas, que actúe como un doble de pruebas, donde podremos bloquear los recursos externos que no queremos cargar agregándolos a una lista negra o suministrando una versión ligera.

Proxy

- Un servidor proxy actúa como intermediario para solicitudes entre un cliente y un servidor. En forma simple, el tráfico fluye a través del servidor proxy en camino a la dirección solicitada y de regreso.
- Un servidor proxy para scripts de automatización con Selenium podría ser útil para:
  - Capturar el tráfico de la red
  - Simular llamadas de backend realizadas por el sitio web
  - Accede al sitio web requerido bajo topologías de red complejas o restricciones/políticas corporativas estrictas.
- Si te encuentras en un entorno corporativo, y un navegador no puede conectarse a una URL, esto es muy probablemente porque el ambiente necesita un proxy para acceder.
- Selenium WebDriver proporciona una vía para configurar el proxy:

```
Proxy proxy = new Proxy();
proxy.setHttpProxy("<HOST:PORT>");
ChromeOptions options = new ChromeOptions();
options.setCapability("proxy", proxy);
WebDriver driver = new ChromeDriver(options);
```

Perfiles Firefox

- Firefox guarda tu información personal (marcadores, contraseñas y preferencias de usuario) en un conjunto de archivos llamado perfil, que se almacena en una ubicación de disco diferente a la que se utiliza para almacenar los archivos de ejecución de Firefox. Puedes crear distintos perfiles, cada uno de ellos con un conjunto de datos de usuario diferente. Solo puede estar activo un perfil en un momento determinado. Mediante el Administrador de perfiles de Firefox puedes crear, eliminar y renombrar los perfiles.
- Cuando se desee ejecutar una automatización confiable en un navegador Firefox, se recomienda crear un perfil separado.
- Para iniciar el Administrador de perfiles se introduce en la barra de about:profiles.
- Para abrir Firefox con un determinado perfil: firefox.exe -P webdrive

```
ProfilesIni profile = new ProfilesIni();
FirefoxProfile myprofile = profile.getProfile("testProfile");
WebDriver driver = new FirefoxDriver(myprofile);
```


Visión de conjunto

- Una vez referenciado el controlador ya se puede interactuar con el navegador, todas las acciones se realizaran a través de dicha referencia.
- Con el controlador se navega a una nueva página.
- En la página, se buscaran los diferentes elementos (tag) con el método findElement y un localizador, que devolverá un WebElement. Este método recupera un solo elemento, si se necesita recuperar varios elementos, el método findElements obtiene la colección de elementos localizados.
- Un localizador es un objeto que define el selector de los elementos web basándose en diferentes estrategia como ID, Nombre, Clase, XPath, Selectores CSS, Texto de enlace, etc. Los WebElements se pueden encontrar buscando desde la raíz del documento o buscando en otra WebElement.
- Un WebElement representa un elemento del DOM. El WebElement tiene un conjunto de métodos de acción, tales como click(), getText() y sendKeys().
- Esta es la forma principal para interactuar con un elemento (etiqueta) de la página y obtener información de respuesta de el.

Navegación

- Navegar hacia
  driver.get("https://selenium.dev"); // Recomendada
  driver.navigate().to("https://selenium.dev");
- Retroceder
  driver.navigate().back();
- Avanzar
  driver.navigate().forward();
- Actualizar
  driver.navigate().refresh();
- Obtener la URL actual
  url = driver.getCurrentUrl();
- Obtener el título
  driver.getTitle();

Navegación

- Salir y cerrar el navegador al final de una sesión

  driver.quit();
  - Muy importante porque:
    - Cerrará todas las ventanas y pestañas asociadas a esa sesión del WebDriver.
    - Cerrará el proceso de navegador.
    - Cerrará el proceso en segundo plano del driver.
    - Notificará al Grid de Selenium que el navegador ya no esta en uso y que puede ser usado por otra sesión del Grid de Selenium.
  - Un fallo en la llamada del método salir dejará procesos corriendo en segundo plano y puertos abiertos en tu maquina lo que podría llevar a problemas en un futuro.

Estrategia de carga de página

- Por defecto, cuando WebDriver carga una página, sigue la estrategia de carga NORMAL. Siempre se recomienda detener la descarga de más recursos adicionales (como imágenes, css, js) cuando la carga de la página lleva mucho tiempo.
- La propiedad document.readyState de un documento describe el estado de carga del documento actual. Por defecto, WebDriver esperará responder a una llamada driver.get() o driver.navigate().to() hasta que el estado de documento listo esté completo
- En aplicaciones SPA (como Angular, react, Ember) una vez que el contenido dinámico ya está cargado (es decir, una vez que el estado de readyState es COMPLETE), hacer clic en un enlace o realizar alguna acción dentro de la página no disparara una nueva solicitud al servidor ya que el contenido se carga dinámicamente en el lado del cliente sin una actualización de la página. Las aplicaciones de SPA pueden cargar muchas vistas dinámicamente sin ninguna solicitud al servidor, por lo que pageLoadStrategy siempre mostrará el estado 'COMPLETE' hasta que se haga un nuevo driver.get() y driver.navigate().to().

Cookies

- Una cookie es una pequeña pieza de datos que es enviada desde el sitio web y es almacenada en el ordenador. Las cookies son usadas principalmente para reconocer al usuario y cargar la información personalizada.
- Se gestionan a traves de driver.manage(), una ver descargada la pagina:

```
driver.manage().addCookie(new Cookie("key", "value"));
Cookie cookie = driver.manage().getCookieNamed("foo");
Set<Cookie> cookies = driver.manage().getCookies();
driver.manage().deleteCookieNamed("foo");
driver.manage().deleteCookie(cookie);
driver.manage().deleteAllCookies();
```

- Por defecto, cuando el atributo sameSite esta fijado como Strict (estricto en español), la cookie no será enviada junto a las peticiones iniciadas por paginas web externas. Cuando se fija como Lax, será enviada junto con la petición GET iniciada por paginas web externas.

```
Cookie cookie = new Cookie.Builder("key", "value").sameSite("Strict").build();
Cookie cookie1 = new Cookie.Builder("key", "value").sameSite("Lax").build();
```

Ventanas, solapas, Frames e Iframes

- El uso de múltiples ventanas, o solapas, Frames e Iframes está prohibido por la reglas de WAI, dado construyem sitios desde múltiples documentos en el mismo dominio.
- Para trabajar con ventanas o solapas:

```
driver.getWindowHandle();
driver.switchTo().window(windowHandle);
driver.close();
```

- Para trabajar con Frames e Iframes

```
driver.switchTo().frame("myframe");
driver.switchTo().defaultContent();
```


Tamaño del navegador

- La resolución de las pantallas puede impactar en como la aplicación se renderiza.
- El W3C impone restricciones sobre los elementos presentes pero no visibles.
- WebDriver provee de mecanismos para mover y cambiar el tamaño de la ventana del navegador:

```
driver.manage().window().setSize(new Dimension(1024, 768));
driver.manage().window().maximize();
driver.manage().window().minimize();
driver.manage().window().fullscreen();
```

Buscar elementos

- Los WebElements se pueden encontrar buscando desde la raíz del documento utilizando una instancia de WebDriver o buscando en otra WebElement. Las búsquedas son costosas por lo que conviene guardar el resultado para no repetirlas.
- Localizar un elemento o el primer elemento:

```
WebElement searchForm = driver.findElement(By.id("myForm"));
WebElement searchBox = searchForm.findElement(By.name("q"));
```

- Localizar múltiples elementos:

```
List<WebElement> elements = driver.findElements(By.tagName("p"));
```

- Localizar el elemento activo (que tiene el foco en el contexto de navegación actual).

```
WebElement active = driver.SwitchTo().ActiveElement();
```


Localización de elementos

- La interfaz By encapsula las diferentes estrategias de localización de elementos. Hay ocho estrategias diferentes de ubicación de elementos integradas en WebDriver.

- By.id: Localiza elementos cuyo atributo ID coincide con el valor de la búsqueda

```
WebElement tag = driver.findElement(By.id("myForm"));
```

- By.name: Localiza elementos cuyo atributo NAME coincide con el valor de la búsqueda

```
WebElement tag = driver.findElement(By.name("username"));
```

- By.className: Localiza elementos en el que el nombre de su clase contiene el valor de la búsqueda (no se permiten nombres de clase compuestos)

```
WebElement tag = driver.findElement(By.className("container-fluid"));
```

Localización de elementos

- By.cssSelector: Localiza elementos que coinciden con un selector CSS
  WebElement tag = driver.findElement(By.cssSelector(".header img"));
- By.linkText: Localiza hipervínculos cuyo texto visible coincide con el valor de búsqueda
  WebElement tag = driver.findElement(By.linkText("Close"));
- By.partialLinkText: Localiza hipervínculos cuyo texto visible coincide con el valor de búsqueda
  WebElement tag = driver.findElement(By.partialLinkText("Next"));
- By.tagName: Localiza elementos cuyo nombre de etiqueta (tagName) coincide con el valor de búsqueda
  List<WebElement> elements = driver.findElements(By.tagName("img"));
- By.xpath : Localiza elementos utilizando una expresión Xpath
  WebElement tag = driver.findElement(By.xpath("//div[2]/input"));

Localización de elementos

- En general, si los ID o Name del HTML están disponibles, son únicos y predecibles, son el método preferido para ubicar un elemento en una página. Tienden a trabajar muy rápido y evitan costosos recorridos DOM.
- Si las ID únicas no están disponibles, un selector CSS bien escrito es el método preferido para localizar un elemento. XPath funciona tan bien como los selectores CSS, pero la sintaxis es complicada y con frecuencia difícil de depurar. Aunque los selectores XPath son muy flexibles, generalmente su desempeño no es probado por lo proveedores de navegadores y tienden a ser bastante lentos.
- Las estrategias de selección basadas en enlaces de texto y enlaces de texto parciales tienen el inconveniente en que solo funcionan en elementos de enlace. Además, internamente llaman a los selectores XPath.
- El nombre de la etiqueta puede ser una forma peligrosa de localizar elementos. Existen frecuentemente múltiples elementos con la misma etiqueta presentes en la página. Es útil sobre todo cuando se llama al método findElements(By) que devuelve una colección de elementos.
- La recomendación es mantener los localizadores tan compactos y legibles como sea posible. Pedirle a WebDriver que atraviese la estructura del DOM es una operación costosa y, cuanto más se pueda reducir el alcance de tu búsqueda, mejor.

Operar con elementos

- Recuperar el estado del elemento
  .getText(): Obtiene el texto visible (es decir, no oculto por CSS) del elemento, incluidos los subelementos.
  .isDisplayed(): Determina si el elemento es visible. Este método evita el problema de tener que analizar el atributo "estilo" de un elemento.
  .isEnabled(): Determina si el elemento está habilitado actualmente.
  .isSelected(): Determina si este elemento está seleccionado, tiene el foco.
  .getAttribute(): Obtiene el valor del atributo dado del elemento.
  .getCssValue(): Obtiene el valor de una propiedad CSS dada.
  .getLocation(): Obtiene en qué parte de la página se encuentra la esquina superior izquierda del elemento renderizado.
  .getSize(): Obtiene el ancho y el alto del elemento renderizado
  .getTagName(): Obtiene el nombre de la etiqueta de este elemento.

```
assertEquals("WebDriver", driver.findElement(By.id("titulo")).getText());
```


Operar con elementos

- Acciones
  - .click(): Haz clic en este elemento.
  - .sendKeys(): Simula escribir en un elemento.
  - .clear(): Si es un elemento de entrada de texto, borra el valor.
  - .submit(): Si el elemento es un formulario o un elemento dentro de un formulario, se enviará el formulario al servidor remoto.

```
driver.findElement(By.name("password")).clear();
driver.findElement(By.name("password")).sendKeys("god");
driver.findElement(By.id("myForm")).submit();
```

- Buscar subelementos:
  - .findElement(): Encuentra el primer WebElement usando el localizador dado.
  - .findElements(): Encuentra todos los elementos dentro del elemento usando el localizador dado.

Esperas

- Generalmente se puede decir que WebDriver posee una API de bloqueo. Porque es una biblioteca fuera-de-proceso que instruye al navegador qué hacer, y debido a que la plataforma web tiene una naturaleza intrínsecamente asíncrona, WebDriver no rastrea el estado activo y en tiempo real del DOM.
- Afortunadamente, el conjunto normal de acciones disponibles en la interfaz WebElement tales como click o sendKeys estan garantizadas para ser síncronas, es decir, las llamadas a métodos no vuelven hasta que el comando se haya completado en el navegador. Las API avanzadas de interacción del usuario, Keyboard y Mouse, son excepciones ya que están explícitamente pensadas como comandos asíncronos “Haz lo que te digo”.
- Esperar es hacer que transcurra una cierta cantidad de tiempo antes de continuar con el siguiente paso en la ejecución automatizada de la tarea.

Esperas explícitas

- Permiten que el código detenga la ejecución del programa, o congelar el hilo, hasta que la condición pasada sea resuelva. La condición se llama con cierta frecuencia, hasta que transcurra el tiempo de espera. Esto significa que mientras la condición devuelva un valor falso, seguirá intentando y esperando.
- Dado que las esperas explícitas permiten esperar a que ocurra una condición, hacen una buena combinación para sincronizar el estado entre el navegador, y su DOM, con el código de WebDriver.
- Para esperar a que la llamada findElement espere hasta que el elemento agregado dinámicamente desde el script se haya agregado al DOM:

```
WebElement firstResult = new WebDriverWait(driver, 10)
      .until(driver -> driver.findElement(By.xpath("//li/a")));
assertEquals(firstResult.getText());
```

- La condición de espera se puede personalizar para optimizar la espera:

```
WebElement firstResult = new WebDriverWait(driver, 10)
      .until(ExpectedConditions.elementToBeClickable(By.xpath("//li/a")));
```

Condiciones esperadas

- Es una necesidad bastante común tener que sincronizar el DOM con el código de prueba, hay condiciones predefinidas para operaciones frecuentes de espera.
- Las condiciones disponibles varían en las diferentes librerías de los lenguajes, pero esta es una lista no exhaustiva de algunos:
  - alert is present (la alerta esta presente)
  - element exists (el elemento existe)
  - element is visible (el elemento es visible)
  - title contains (el titulo contiene)
  - title is (el titulo es)
  - visible text (el texto es visible)
  - text to be (el texto es)

Espera predefinida

- Una instancia de FluentWait define la cantidad máxima de tiempo para esperar por una condición, así como la frecuencia con la que verificar dicha condición.
- Se puede configurar la espera para ignorar tipos específicos de excepciones mientras esperan, como NoSuchElementException cuando buscan un elemento en la página.
- La instancia de FluentWait se puede reutilizar en tantas esperas, con la misma latencia, como sea necesario.

```
Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)
        .withTimeout(Duration.ofSeconds(30))
        .pollingEvery(Duration.ofSeconds(5))
        .ignoring(NoSuchElementException.class);

WebElement firstResult = wait.until(driver -> driver.findElement(By.xpath("//li/a")));
```

Espera implícita

- Una espera implícita es decirle a WebDriver que sondee el DOM durante un cierto período de tiempo al intentar encontrar un elemento o elementos si no están disponibles de inmediato. Esto puede ser útil cuando ciertos elementos en la página web no están disponibles de inmediato y necesitan algo de tiempo para cargarse.
- La espera implícita está deshabilitada de forma predeterminada y deberá habilitarse manualmente por sesión.

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://somedomain/url_that_delays_loading");
WebElement element = driver.findElement(By.id("myDynamicElement"));
```

- Si se habilita la espera implícita se pierde la posibilidad de fijar esperas explícitas. No se debe mezclar esperas implícitas y explícitas. Mezclar esperas explícitas y esperas implícitas causará consecuencias no deseadas, es decir, esperara el máximo de tiempo incluso si el elemento está disponible o la condición es verdadera.

Mensajes emergentes

- WebDriver proporciona una API para trabajar con los tres tipos nativos de mensajes emergentes ofrecidos por JavaScript: alertas, prompts y confirmaciones. Estas ventanas emergentes están diseñadas por el navegador y ofrecen personalización limitada.
- **Alertas**: muestra un mensaje personalizado y un solo botón que cierra la alerta, etiquetado en la mayoría de los navegadores como OK.

```
Alert alert = wait.until(ExpectedConditions.alertIsPresent());
assertEquals(msgEsperado, alert.getText());
alert.accept(); // Cerrar
```

Mensajes emergentes

- **Confirm**: muestra un mensaje de confirmación y dos botones: uno para confirmar y otro para descartar.

```
Alert alert = wait.until(ExpectedConditions.alertIsPresent());
assertEquals(msgEsperado, alert.getText());
alert.accept();  // confirmar
alert.dismiss(); // descartar
```

- **Prompt**: solicita un texto al usuario, muestra un mensaje de petición, una caja de texto y dos botones: uno para aceptar y otro para cancelar.

```
Alert alert = wait.until(ExpectedConditions.alertIsPresent());
assertEquals(msgEsperado, alert.getText());
alert.sendKeys("Selenium");
alert.accept();  // aceptar
alert.dismiss(); // cancelar
```


Interacciones de usuario avanzadas

- WebDrive suministra los métodos de alto nivel sendKeys y click para simular las entradas habituales de teclado y ratón, pero dispone de la posibilidad de crear acciones compuestas con interacciones de menor nivel.
- La API de interacciones de usuario avanzadas es una API más completa para describir acciones que un usuario puede realizar en una página web. Esto incluye acciones como arrastrar y soltar o hacer clic en varios elementos mientras mantiene presionada la tecla Ctrl.
- La clase Actions implementa los patrones Builder y Composite para crea una acción compuesta que contiene todas las acciones especificadas por las llamadas al método. Para generar una secuencia de acciones, se usa el generador de acciones para construirla:

```
Actions provider = new Actions(driver);
Action action = provider.keyDown(Keys.CONTROL)
    .click(someElement).click(someOtherElement)
    .keyUp(Keys.CONTROL)
    .build();
```

Interacciones de usuario avanzadas

- El objetos Action solo permite la ejecución de las secuencia:
  action.perform();
- Si la secuencia es de un solo uso, se puede construir y ejecutar en un solo paso:

```
provider.keyDown(Keys.CONTROL)
    .click(someElement).click(someOtherElement)
    .keyUp(Keys.CONTROL)
    .perform();
```

- La secuencia de acciones debe ser corta: es mejor realizar una secuencia corta de acciones y verificar que la página esté en el estado correcto antes de que tenga lugar el resto de la secuencia.

Teclado

- Hasta ahora, la interacción del teclado se realizaba a través de un elemento específico y WebDriver se aseguraba de que el elemento estuviera en el estado adecuado para esta interacción. Esto consistía principalmente en desplazar el elemento a la ventana gráfica y centrarse en el elemento.
- Dado que el API de interacciones adopta un enfoque orientado al usuario, es más lógico interactuar explícitamente con el elemento antes de enviarle texto, como lo haría un usuario. Esto significa hacer clic en un elemento o enviar un mensaje Keys.TAB cuando se enfoca en un elemento adyacente.
- En el API de interacciones se define la secuencia de acciones de teclado sin proporcionar un elemento, se aplican al elemento actual que tiene el foco. Posteriormente se puede cambiar el foco a un elemento antes de enviarle eventos de teclado.

```
Actions provider = new Actions(driver);
Action action = provider.action.keyDown(Keys.SHIFT).sendKeys(search,"Hola
        ").keyUp(Keys.SHIFT).sendKeys("Mundo").sendKeys(Keys.TAB).build();
action.perform();
```

Ratón

- Las acciones del mouse tienen un contexto: la ubicación actual del mouse. Entonces, al establecer un contexto para varias acciones del mouse (usando onElement), la primera acción será relativa a la ubicación del elemento utilizado como contexto, la siguiente acción será relativa a la ubicación del mouse al final de la última acción, etc.
- Mientras que el método de alto nivel sendKeys cubre la mayoría de los escenarios de teclado y no suele requerir acciones de bajo nivel, el método click cubre el escenario mas común para el ratos pero deja fuera operaciones habituales como el doble click, los desplazamientos, el menú contextual, arrastrar y soltar, … Para todas estas acciones es necesario el API de interacciones.

```
Actions provider = new Actions(driver);

provider.clickAndHold(sourceEle).moveToElement(targetEle) .release().build().perform();
```


Acciones

- Teclado:
  - SendKeysAction: equivalente a WebElement.sendKey(...)
  - KeyDownAction: mantener presionada una tecla modificadora.
  - KeyUpAction: liberar la tecla modificadora.

- Ratón:
  - ClickAction: equivalente a WebElement.click()
  - DoubleClickAction: hacer doble clic en un elemento.
  - ClickAndHoldAction: mantenga presionado el botón izquierdo del mouse.
  - ButtonReleaseAction: liberar un botón del ratón retenido.
  - ContextClickAction: hacer clic con el botón secundario del ratón, (que generalmente abre el menú contextual).
  - MoveMouseAction: mover el ratón de su ubicación actual a otro elemento.
  - MoveToOffsetAction: mover el ratón una cantidad (desplazamiento) desde un elemento (el desplazamiento podría ser negativo y el elemento podría ser el mismo elemento al que se acaba de mover el ratón).

Trabajando con elementos select

- A la hora de seleccionar elementos puede ser necesario código repetitivo para poder ser automatizado.
- Para reducir esto y hacer tus test mas limpios, existe un clase Select en los paquetes de soporte de Selenium.

```
import org.openqa.selenium.support.ui.Select;
```

- El WebElement debe ser envuelto por un objeto Select para acceder a la funcionalidad extendida:

```
Select selectObject = new Select(driver.findElement(By.id("selectElementID")));
```

- La funcionalidad extendida permite:

```
selectObject.selectByIndex(1);
selectObject.selectByValue("value1");
selectObject.selectByVisibleText("Bread");
WebElement firstSelectedOption = selectObject.getFirstSelectedOption();
List<WebElement> allSelectedOptions = selectObject.getAllSelectedOptions();
selectObject.deselectByIndex(1);
selectObject.deselectByValue("value1");
selectObject.deselectByVisibleText("Bread");
selectObject.deselectAll();
```


Trabajando con colores

- En algunas ocasiones es posible que sea necesario querer validar el color de algo como parte de las pruebas. El problema es que las definiciones de color en la web no son constantes.
- La clase org.openqa.selenium.support.Color es una forma sencilla de comparar una representación de color HEX con una representación de color RGB, o una representación de color RGBA con una representación de color HSLA.

```
private final Color HEX_COLOUR = Color.fromString("#2F7ED8");
private final Color RGB_COLOUR = Color.fromString("rgb(255, 255, 255)");
private final Color RGB_COLOUR = Color.fromString("rgb(40%, 20%, 40%)");
private final Color RGBA_COLOUR = Color.fromString("rgba(255, 255, 255, 0.5)");
private final Color RGBA_COLOUR = Color.fromString("rgba(40%, 20%, 40%, 0.5)");
private final Color HSL_COLOUR = Color.fromString("hsl(100, 0%, 50%)");
private final Color HSLA_COLOUR = Color.fromString("hsla(100, 0%, 50%, 0.5)");
private final Color BLACK = Color.fromString("black");
private final Color TRANSPARENT = Color.fromString("transparent");
```

- Para crear las aserciones:

```
Color color = Color.fromString(driver.findElement(By.id("login")).getCssValue("color"));
assertEquals(TRANSPARENT, color);
assertEquals("#2F7ED8", color.asHex());
```

Excepciones

- NoSuchElementException: Se produce cuando se intenta interactuar con un elemento que no satisface la estrategia de selector.

```
driver.findElement(By.linkText("Este no existe")).click();
```

- StaleElementReferenceException: Se produce cuando un elemento que se identificó y almaceno previamente ha sido modificado en el DOM y se intentó interactuar con él después de la mutación.

```
element = wait.until(driver -> driver.findElement(By.id("txtSaluda")));
driver.findElement(By.linkText("Next")).click();
element.sendKeys("oHola");
```

- ElementNotInteractableException: Lanzada para indicar que aunque un elemento está presente en el DOM, no está en un estado con el que se pueda interactuar: fuera de la vista (scroll, solapado, oculto …).

```
Actions provider = new Actions(driver);
provider.sendKeys(Keys.PAGE_DOWN).perform();
provider.keyDown(Keys.CONTROL).sendKeys(Keys.END) .keyUp(Keys.CONTROL).perform();
```


RECOMENDACIONES

Patrones de diseño

- Page Objects: una simple abstracción de la interfaz de usuario de su aplicación web.
- Componente Loadable: Modelado de PageObjects como componentes.
- BotStyleTests: Uso de un enfoque basado en comandos para automatizar pruebas, en lugar del enfoque basado en objetos que PageObjects fomenta
- Domain Driven Design: Exprese las pruebas en el idioma del usuario final de la aplicación.
- Acceptance Test Driven Development (ATDD): técnica conocida también como Story Test-Driven Development (STDD), utiliza las pruebas de aceptación para ayudar a estructurar el trabajo de desarrollo.

Lenguaje de dominio específico

- Un lenguaje de dominio específico (DSL) es un sistema que proporciona al usuario un medio expresivo para resolver un problema. Permite a un usuario interactuar con el sistema en sus términos, no solo en jerga del programador.
- A los usuarios, en general, no les importa principalmente cómo se ve su sitio, no están preocupados por la decoración, animaciones o gráficos. Lo que realmente les importa es como el sistema se impulsa a través de los procesos con una mínima dificultad. El trabajo del probador debe acercarse lo más que pueda a “capturar” esta mentalidad utilizando un lenguaje ubicuo (la terminología del usuario). Con eso en mente, los scripts de prueba deberían se legibles al usuario.
- Con Selenium, el DSL generalmente se representa por métodos, la elección de los sus nombres es de suma importancia, así como la refactorización para ocultar la complejidad.
- Beneficios
  - Legible: Las partes interesadas del negocio pueden entenderlo.
  - Escribible: Fácil de escribir, evita duplicaciones innecesarias.
  - Extensible: Se puede agregar funcionalidad (razonablemente) sin romper los contratos y la funcionalidad existente.
  - Mantenible: Al dejar los detalles de implementación fuera de casos de prueba, está bien aislado contra cambios en el aplicación.

Patrón Page Objects

- https://martinfowler.com/bliki/PageObject.html
- Cuando se escriben pruebas de una página web, hay que acceder a los elementos dentro de esa página web para hacer clic en los elementos, teclear entradas y determinar lo que se muestra.
- Sin embargo, si se escriben pruebas que manipulan los elementos HTML directamente, las pruebas serán frágiles ante los cambios en la interfaz de usuario.
- Un objeto de página envuelve una página HTML, o un fragmento, con una API específica de la aplicación, lo que permite manipular los elementos de la página sin excavar en el HTML.

Patrón Page Objects

- La regla básica para un objeto de página es que debe permitir que un cliente de software haga cualquier cosa y vea todo lo que un humano puede hacer.
- El objeto de página debe proporcionar una interfaz que sea fácil de programar y oculta en la ventana.
- Entonces, para acceder a un campo de texto, debe tener métodos de acceso que tomen y devuelvan una cadena, las casillas de verificación deben usar valores booleanos y los botones deben estar representados por nombres de métodos orientados a la acción.
- El objeto de la página debe encapsular los mecanismos necesarios para encontrar y manipular los datos en el propio control GUI.
- A pesar del término objeto de "página", estos objetos no deberían construirse para cada página, sino para los elementos significativos en una página.
- Los problemas de concurrencia y asincronía son otro tema que un objeto de página puede encapsular.

Ventajas del patrón Page Objects

- De acuerdo con patrón Page Object, deberíamos mantener nuestras pruebas y localizadores de elementos por separado, esto mantendrá el código limpio y fácil de entender y mantener.
- El enfoque Page Object hace que el programador de marcos de automatización de pruebas sea más fácil, duradero y completo.
- Otra ventaja importante es que nuestro repositorio de objetos de página es independiente de las pruebas de automatización. Mantener un repositorio separado para los objetos de la página nos ayuda a usar este repositorio para diferentes propósitos con diferentes marcos como, podemos integrar este repositorio con otras herramientas como JUnit / NUnit / PhpUnit, así como con TestNG / Cucumber / etc.
- Los casos de prueba se vuelven cortos y optimizados, ya que podemos reutilizar los métodos de objetos de página.
- Los casos de prueba se centran solamente en el comportamiento.
- Cualquier cambio en la IU se puede implementar, actualizar y mantener fácilmente en los objetos y clases de página sin afectar a los casos de pruebas que no estén implicados.

Page Objects

```
public class LoginPage {
    private WebDriver driver;
    private final By txtUsuarioBy = By.id("txtUsuario");
    private final By txtPasswordBy = By.id("txtPassword");
    private final By btnSendLoginBy = By.id("btnSendLogin");
    private final By userDataBy = By.id("userData");

    public LoginPage(WebDriver driver) {
        this.driver = driver;
        driver.get("http://localhost/login");
    }
    public void ponUsuario(String valor) { driver.findElement(txtUsuarioBy).sendKeys(valor); }
    public void ponPassword(String valor) { driver.findElement(txtPasswordBy).sendKeys(valor); }
    public void enviarLogin() { driver.findElement(btnSendLoginBy).click();          }
    public String textoSaludo() {
        WebElement element = new WebDriverWait(driver, Duration.ofSeconds(3).getSeconds()).until(driver ->
            driver.findElement(userDataBy));
        return element.getText();
    }
}
```

Prueba usando un Page Objects

```
@Test
public void loginTest() {
   LoginPage page = new LoginPage(driver);
   page.ponUsuario("admin");
   page.ponPassword("P@$$w0rd");
   page.enviarLogin();
   assertThat(page.textoSaludo(), is("Hola Administrador"));

}
```


PageFactory

- La biblioteca de soporte de WebDriver contiene una clase de PageFactory que simplifica la implementación del patón PageObject.
- La PageObject declara sus campos como WebElements o List <WebElement>:
  `private WebElement txtUsuario;`
- Cuando ejecutamos la factoría, PageFactory buscará un elemento en la página que coincida con el nombre del campo WebElement en la clase e inyectara el elemento en el campo. Para ello, primero busca un elemento con un atributo de ID coincidente. Si esto falla, recurre a la búsqueda de un elemento por el valor de su atributo "name".
  ```
  LoginPageFactory page = new LoginPageFactory(driver);
  PageFactory.initElements(driver, page);
  ```
- La anotación @FindBy permite elegir un nombre significativo de campo y cambiar la estrategia utilizada para buscar el elemento:
  ```
  @FindBy(how = How.CSS, css = "#userData")
  private WebElement saludo;
  ```

PageFactory

```
public class LoginPageFactory {
        private WebDriver driver;
        private WebElement txtUsuario;
        private WebElement txtPassword;
        private WebElement btnSendLogin;
        @FindBy(how = How.CSS, css = "#userData")
        private WebElement saludo;

        public LoginPageFactory(WebDriver driver) { this.driver = driver; }
        public void ponUsuario(String valor) { txtUsuario.sendKeys(valor); }
        public void ponPassword(String valor) { txtPassword.sendKeys(valor); }
        public void enviarLogin() { btnSendLogin.click(); }
        public String textoSaludo() { return saludo.getText(); }
}

LoginPageFactory page = PageFactory.initElements(driver, LoginPageFactory.class);
```


Pruebas de estilo bot

- Un "bot" es una abstracción orientada a la acción sobre las API de Selenium sin procesar. Permite ocultar la complejidad del API con comandos mayor nivel que son fácil de cambiar.

```
public class ActionBot {
  private final WebDriver driver;
  public ActionBot(WebDriver driver) { this.driver = driver; }
  public void click(By locator) { driver.findElement(locator).click(); }
  public void submit(By locator) { driver.findElement(locator).submit(); }
  public void type(By locator, String text) {
    WebElement element = driver.findElement(locator);
    element.clear();
    element.sendKeys(text);
  }
}
```

- Una vez que se han construido estas abstracciones y se ha identificado la duplicación en sus pruebas, es posible colocar los PageObjects encima de los bots.

API fluída

- Un API fluida, termino acuñado por Martin Fowler y Eric Evans, permite encadenar varias acciones consecutivas proporcionando una sensación más fluida al código.

```
String saludo = sitio.pedirLogin().rellenarFormulario("admin",
        "P@$$w0rd").enviarFormulario().dameTextoSaludo();
```

- Crear un API fluida significa construirla de tal manera que cumpla con los siguientes criterios:
  - El usuario de la API puede entenderla API muy fácilmente.
  - El API puede realizar una serie de acciones para finalizar una tarea.
  - El nombre de cada método debe utilizar la terminología específica del dominio.
  - La API debe ser lo suficientemente sugerente como para guiar a los usuarios del API sobre qué hacer a continuación y qué posibles operaciones se pueden en un determinado momento.
- Son interfaces o clases que cuando invocamos a un método concreto nos devuelve el mismo objeto modificado u otro objeto. De tal forma que podemos volver a solicitar otro método del mismo objeto, y encadenar más operaciones, o continuar las operaciones con el nuevo objeto.
- Se recomienda considerar el uso del patrón de diseño Fluent API en el objeto de página. El API de Seleniun WebDriver suministra la clase base LoadableComponent que tiene como objetivo simplificar la creación de PageObjects fluidos.

LoadableComponent

- LoadableComponent proporciona una forma estándar de garantizar que las páginas se carguen y facilita la depuración de la errores de carga de la página. Ayuda a reducir la cantidad de código repetitivo en la pruebas, lo que a su vez hace que el mantenimiento de la pruebas sea menos agotador.
- La clase cuenta con tres métodos:
  - public T get():
    - Devuelve el PageObject si la página esta cargada o cargándola si es necesario.
  - protected abstract void load():
    - Cuando este método termine, el componente modelado por la subclase debería estar completamente cargado. Se espera que esta subclase navegue a una página apropiada si fuera necesario.
  - protected abstract void isLoaded() throws Error
    - Determine si el componente está cargado o no, lanzando un Error (no una excepción) cuando no este cargada u obsoleta. se carga el componente, este método volverá, pero cuando no se carga, se debe lanzar un. Esto permite la verificación compleja y el informe de errores al cargar una página o cuando una página no se carga.

LoadableComponent

```
public class LoginPageFluent {
	private WebDriver driver;
	private WebElement txtUsuario;
	private WebElement txtPassword;
	private WebElement btnSendLogin;
	@FindBy(how = How.CSS, css = "#userData")
	private WebElement saludo;

	public LoginPageFactory(WebDriver driver) { this.driver = driver; }
	@Override
	protected void load() {
		driver.get("http://localhost/login");
		PageFactory.initElements(driver, this);
	}
	@Override
	protected void isLoaded() throws Error {
		if(driver.getTitle() == null || !driver.getTitle().contains("Login"))
				throw new Error();
	}
```

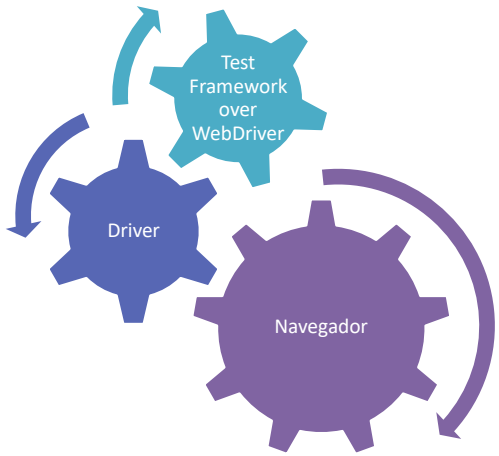

LoadableComponent

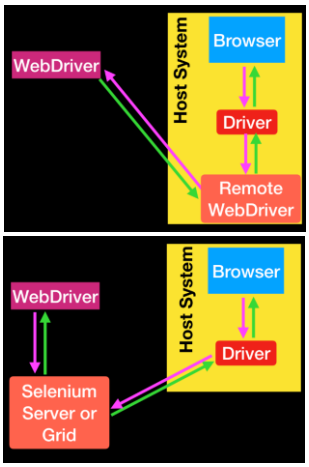
```
	public LoginPageFluent ponUsuario(String valor) {
	    txtUsuario.sendKeys(valor);  return this;
	}
	public LoginPageFluent ponPassword(String valor) {
	    txtPassword.sendKeys(valor); return this;
	}
	public LoginPageFluent enviarLogin() {
	    btnSendLogin.click(); return this;
	}
	public String textoSaludo() { return saludo.getText(); }
}

LoginPageFluent page = new LoginPageFluent();
assertEquals("Hola Administrador", page.ponUsuario("admin")
		.ponPassword("P@$$w0rd").enviarLogin().textoSaludo());
```

SELENIUM GRID

Interacciones





Remote WebDriver

- Se puede usar WebDriver de forma remota de la misma manera que se usaría localmente. La principal diferencia es que un WebDriver remoto debe ser configurado para que pueda ejecutar las pruebas en una máquina diferente.
- Un WebDriver remoto se compone de dos piezas: un cliente y un servidor. El cliente es la prueba de WebDriver y el servidor es simplemente un servlet Java, que se puede alojar en cualquier servidor moderno de aplicaciones JEE.
- Pros
  - Separa dónde se ejecutan las pruebas desde donde está el navegador.
  - Permite que las pruebas se ejecuten con navegadores no disponibles en el sistema operativo actual
- Contras
  - Requiere que se ejecute un contenedor de servlet externo
  - Puede encontrar problemas con los finales de línea al obtener texto del servidor remoto
  - Introduce latencia adicional a las pruebas, sobre todo con las excepciones.

Servidor

- El servidor siempre se ejecutará en la máquina con el navegador que deseas probar. Debe tener instalados los diferentes driver para los navegadores.
- El servidor se puede usar desde la línea de comandos o mediante configuración de código. Standalone combina todos los componentes de Grid todo en uno en una sola máquina.

```
java -jar selenium-server-<version>.jar standalone
```

- Para personalizar la configuración por defecto, se utiliza un archivo de configuración JSON:

```
{
  "server": {
    "port": 4449
  },
}
```

```
java -jar selenium-server-<version>.jar standalone -config standaloneConfig.json
```

- Esta disponible una consola en: http://localhost:4444/

Cliente (java)

- Las pruebas que quieran ejecutarse en remoto deben sustituir la instancia de WebDriver por una instancia de RemoteWebDriver.
- El constructor recibe dos parámetros:
  - URL con la dirección del servidor que ejecutara las pruebas.
  - Una instancia de las opciones del navegador que debe usar el servidor remoto (browserName).
- Para implementar la prueba de ejecución remota:

```
FirefoxOptions firefoxOptions = new FirefoxOptions();
WebDriver driver = new RemoteWebDriver(new URL("http://localhost:4444"),
    firefoxOptions);
driver.get("http://www.google.com");
driver.quit();
```


Cliente (js)

- Las pruebas que quieran ejecutarse en remoto deben sustituir la instancia de WebDriver por una instancia de RemoteWebDriver.
- El constructor recibe dos parámetros:
  - URL con la dirección del servidor que ejecutara las pruebas.
  - Una instancia de las opciones del navegador que debe usar el servidor remoto (browserName).
- Para implementar la prueba de ejecución remota:

```
const { Builder, Capabilities } = require("selenium-webdriver");
let driver = new Builder()
      .usingServer("http://localhost:4444")
      .withCapabilities(Capabilities.firefox())
      .build();
await driver.get('http://www.google.com');
await driver.quit();
```

Grid

- Selenium Grid es un servidor inteligente que actúa de proxy lo que permite a los tests de Selenium enrutar sus comandos hacia instancias remotas de navegadores web. La intención es proporcionar una forma sencilla de ejecutar los tests en paralelo en múltiple maquinas.
- Con Selenium Grid un servidor actúa como el centro de actividad (hub) encargado de enrutar los comandos de los tests en formato JSON hacia uno o mas nodos registrados en el Grid. Los tests contactan con el hub para obtener acceso a las instancias remotas de los navegadores.
- Selenium Grid permite ejecutar los tests en paralelo en múltiples maquinas y gestionar diferentes versiones de navegadores y configuraciones de manera centralizada (en lugar de hacerlo de manera individual en cada test).

Grid

- Selenium Grid no es una solución mágica para todos los problemas. Permite resolver un subconjunto de problemas comunes de delegación y distribución, pero no administrará su infraestructura y podría no satisfacer necesidades concretas.
- Sus principales características son:
  - Punto de entrada centralizado para todos los tests
  - Gestión y control de los nodos / entornos donde se ejecutan los navegadores
  - Escalado
  - Balanceo de carga
  - Ejecución de los tests en paralelo
  - Testing cruzado entre diferentes sistemas operativos

Casos de uso

- El Grid se usa para acelerar la ejecución de los test usando múltiples maquinas para ejecutarlos en paralelo: en cuatro máquinas tardaría aproximadamente una cuarta parte de lo que tardaría en una sola. Esto permite ejecutar suites muy grandes, y de larga duración (horas), en un tiempo razonable, así como obtener una retroalimentación temprana.
- El Grid permite ejecutar la suite múltiples entornos, especialmente, contra diferentes navegadores y versiones al mismo tiempo, lo cual seria imposible en una sola maquina. Cuando la suite de test es ejecutada, el Selenium Grid recibe cada combinación de test-navegador y lo asigna para su ejecución a la maquina o maquinas con el navegador requerido.
- Remote WebDriver trabaja en escenarios de uso con un cliente para un remoto, Selenium Grid reutiliza la misma infraestructura para que un cliente use múltiples remotos.

Hub

- El Hub es un punto central donde se envían todos los tests. Cada Grid tiene un único hub. El hub necesita ser accesible desde la perspectiva de los clientes (ej. Servidor de la CI, maquina del desarrollador). El hub se conectará a uno o mas nodos en los que los tests serán ejecutados.
- Características:
  - Ejerce como mediador y administrador
  - Acepta peticiones para ejecutar los tests
  - Recoge instrucciones de los clientes y las ejecuta de forma remota en los nodos
  - Gestiona los hilos

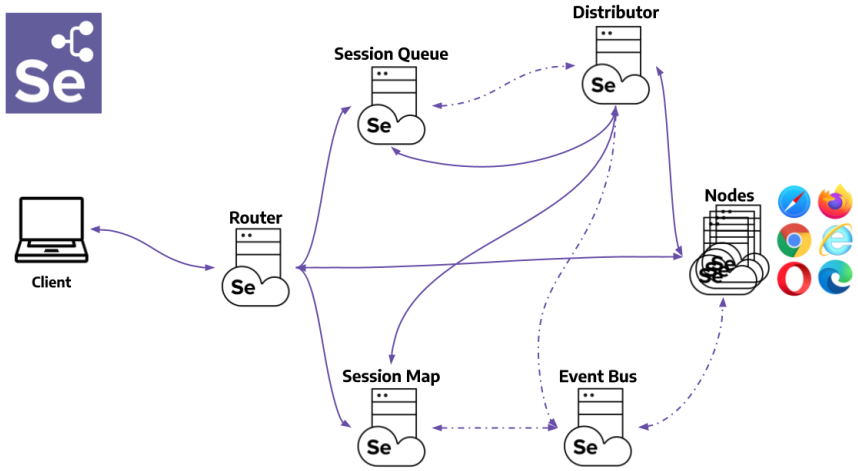
Hub

- Para arrancar el Hub:
  java -jar selenium-server-<version>.jar hub
- El hub escuchará al puerto 4444 por defecto. Se puede ver el estado del hub abriendo una ventana del navegador y navegando a http://localhost:4444/ o consultando http://localhost:4444/status.
- Un Hub es la unión de los siguientes componentes:
  - Router
  - Distributor
  - Session Map
  - New Session Queue
  - Event Bus
- Cuando se utiliza un Grid distribuido, cada componente debe iniciarse por separado. Esta configuración es más adecuada para Grids grandes.

Configuración distribuida

- Enrutador: redirige las solicitudes al componente correcto
  - java -jar selenium-server-<version>.jar router --sessions http://localhost:5556 --distributor http://localhost:5553 --sessionqueue http://localhost:5559
- Cola de sesiones: agrega la solicitud de nueva sesión a una cola para que el distribuidor la procese
  - java -jar selenium-server-<version>.jar sessionqueue
- Mapa de sesión: asigna ID de sesión al nodo donde se ejecuta la sesión
  - java -jar selenium-server-<version>.jar sessions
- Distribuidor: asigna un Nodo para una solicitud de sesión. Otros Nodos se registran en el Distribuidor de la misma manera que se habrían registrado en el Hub en una Red no distribuida
  - java -jar selenium-server-<version>.jar distributor --sessions http://localhost:5556 --sessionqueue http://localhost:5559 --bind-bus false
- Bus de eventos: sirve como ruta de comunicación entre los nodos, el distribuidor, la cola de nuevas sesión y el mapa de sesión.
  - java -jar selenium-server-<version>.jar event-bus

Arquitectura



Nodos

- Los nodos son diferentes instancias de Selenium que ejecutarán los tests en sistemas informáticos individuales. Puede haber tantos nodos en un grid como sean necesarios.
- Las maquinas que contienen los nodos no necesitan disponer del mismo sistema operativo o el mismo conjunto de navegadores que el hub o los otros nodos. Un nodo en Windows podría tener la capacidad de ofrecer Internet Explorer como opción del navegador mientras que esto no podría ser posible en Linux o Mac.
- Características:
  - Donde se ubican los navegadores
  - Se registra a si mismo en el hub y le comunica sus capacidades
  - Recibe las peticiones desde el hub y las ejecuta
- Se pueden iniciar uno o más Nodos y el servidor detectará en el PATH los controladores disponibles:
  - java -jar selenium-server-<version>.jar node

Nodos

- Al iniciar los nodos deben indicar la url del hub (si no se especifica un puerto a través del parámetro -port se elegirá un puerto libre):

  java -jar selenium-server-standalone.jar -role node -hub http://localhost:4444

- Para personalizar la configuración por defecto, se utiliza un archivo de configuración JSON:

```
{
  "capabilities": [
    { "browserName": "firefox", "maxInstances": 2, "seleniumProtocol": "WebDriver" },
    { "browserName": "chrome", "maxInstances": 3, "seleniumProtocol": "WebDriver" }
  ],
  "proxy": "org.openqa.grid.selenium.proxy.DefaultRemoteProxy",
  "maxSession": 5, "port": -1, "register": true, "registerCycle": 5000,
  "hub": "http://localhost:4444", "nodeStatusCheckTimeout": 5000, "nodePolling": 5000,
  "unregisterIfStillDownAfter": 60000, "downPollingLimit": 2, "debug": false,
  "servlets": [], "withoutServlets": [], "custom": {}
}
```

```
java -jar selenium-server-standalone.jar -role node -nodeConfig nodeConfig.json
```


Seguridad

- El Grid de Selenium debe estar protegido contra accesos externos mediante el uso apropiado de los permisos del firewall.
- Fallar a la hora de proteger el Grid puede ocasionar uno o mas de los siguientes problemas:
  - Permitir acceso abierto a tu infraestructura del Grid.
  - Permitir a terceros el acceso a aplicaciones web y archivos interno.
  - Permitir a terceros ejecutar tus ejecutables.

Cloud

- Para usar Selenium Grid, se necesita mantener una infraestructura para los nodos. Como esto puede suponer un engorro y un gran esfuerzo de tiempo y recursos, se pueden usar proveedores de IaaS (Infraestructura como servicio) en la nube como Amazon EC2 o Google Compute para proveer de la infraestructura. Otras opciones incluyen usar proveedores como Sauce Labs or Testing Bot los cuales proveen Selenium Grid como servicio en la nube.
- Docker provee una forma conveniente de aprovisionar y escalar la infraestructura de Selenium Grid en unidades conocidas como contenedores. Los contenedores son unidades estandarizadas de software que contienen todo lo necesario para ejecutar la aplicación deseada, incluidas todas las dependencias, en un entorno confiable y regenerable en diferentes sistemas.
- El proyecto de Selenium mantiene un conjunto de imágenes Docker, las cuales se pueden descargar y ejecutar para tener un Grid funcionando rápidamente. Los nodos están disponibles para los navegadores Edge, Firefox y Chrome.
  - https://github.com/SeleniumHQ/docker-selenium

Standalone con Docker

- Iniciar un contenedor Docker con Firefox
  - docker run -d -p 4444:4444 -p 7900:7900 --shm-size="2g" --name standalone-firefox selenium/standalone-firefox:4.4.0-20220812
- Establecer RemoteWebDriver o usingServer apuntando a http://localhost:4444
- Para ver lo que sucede dentro del contenedor:
  - http://localhost:7900 (la contraseña es secret)
- Al ejecutar docker run para una imagen que contiene un navegador, debería indicarse la opción --shm-size=2g para usar la memoria compartida del host.
- Se debería utilizar siempre una imagen de Docker con una etiqueta completa para anclar un navegador específico y una versión de Grid.

Ejecución en paralelo

- Para habilitar la ejecución de test paralelos en junit-platform.properties:

```
junit.jupiter.execution.parallel.enabled = true
#junit.jupiter.execution.parallel.mode.default = concurrent
```

- Para marcar que suites o test deben ejecutarse en paralelo:

```
@ParameterizedTest
@ValueSource(classes = {ChromeOptions.class, FirefoxOptions.class, EdgeOptions.class })
@Execution(ExecutionMode.CONCURRENT)
void paralellTest(Class<? extends Capabilities> options) throws Exception {
    WebDriver driver = new RemoteWebDriver(new URL("http://localhost:4444"),
options.newInstance());
    try {
        driver.get("https://www.selenium.dev/");
        assertEquals("Selenium", driver.getTitle());
    } finally { driver.quit(); }
}
```


Selenium Hub: docker-compose.yaml

```
version: "3"
services:
  chrome:
    image: selenium/node-chrome:4.4.0-20220812
    shm_size: 2gb
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443

  edge:
    image: selenium/node-edge:4.4.0-20220812
    shm_size: 2gb
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443

  firefox:
    image: selenium/node-firefox:4.4.0-20220812
    shm_size: 2gb
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443

  selenium-hub:
    image: selenium/hub:4.4.0-20220812
    container_name: selenium-hub
    ports:
      - "4442:4442"
      - "4443:4443"
      - "4444:4444"

# docker compose up -d
```

Extensiones

- Existen extensiones de Selenium Grid con grabación de video y logs, vista previa en tiempo real, autenticación básica y funciones de grid. También usan docker-selenium para ejecutar pruebas localmente en Chrome, Firefox, Opera, Android, … Se pueden usar múltiples versiones del mismo navegador. Si se necesita un navegador diferente, se pueden redirigir las pruebas a un proveedor de pruebas en la nube (Sauce Labs, BrowserStack, TestingBot). También se pueden utilizar en Kubernetes y entornos CI.

- Selenoid
  - GitHub: https://github.com/aerokube/selenoid
  - Sitio: http://aerokube.com/selenoid/latest/#_getting_started
- Zalenium
  - GitHub: https://github.com/zalando/zalenium
  - Sitio: https://zalando.github.io/zalenium/

https://www.soapui.org

SOAPUI

Contenidos

- Instalación
- Estructura de proyectos
- Preparación de la prueba
- Service Mocking
- Pruebas funcionales: Pasos, Aserciones, Propiedades
- Ejecución
- Pruebas de carga
- Informes
- Data Driven Testing
- TestRunner Command-Line

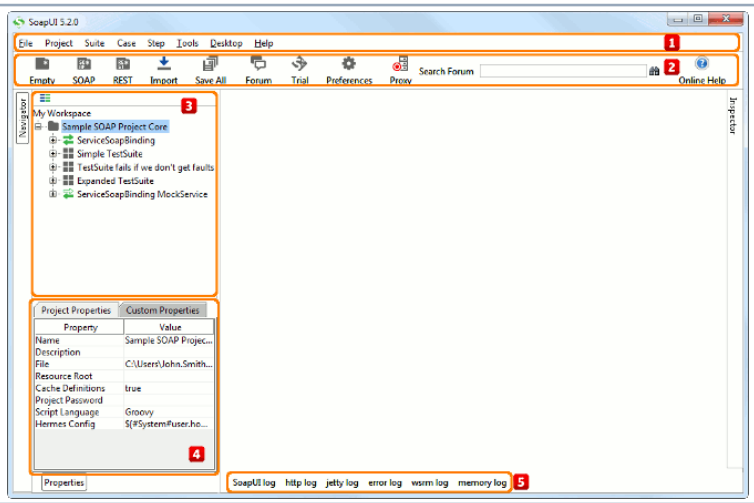
Introducción

- SoapUI es una herramienta para probar servicios web que pueden ser servicios web SOAP, servicios web RESTful u otros servicios basados en HTTP.
- SoapUI es una herramienta de código abierto y completamente gratuita con una versión comercial, ReadyAPI (anteriormente SoapUI Pro), que tiene una funcionalidad adicional para empresas con servicios web de misión crítica.
- SoapUI se considera el estándar de facto para las Pruebas de servicio API. Esto significa que hay mucho conocimiento en la red sobre la herramienta y blogs para obtener más información sobre el uso de SoapUI en la vida real.
- SoapUI permite realizar pruebas funcionales, pruebas de rendimiento, pruebas de interoperabilidad, pruebas de regresión y mucho más. Su objetivo es que la prueba sea bastante fácil de comenzar, por ejemplo, para crear una Prueba de carga, simplemente hay que hacer clic derecho en una prueba funcional y ejecutarla como una prueba de carga.
- SoapUI puede simular servicios web (mocking). Se puede grabar pruebas y usarlas más tarde.
- SoapUI puede crear apéndices de código desde el WSDL. Incluso puede crear especificaciones REST (WADL) a partir de la comunicación grabada.

Instalación

- SoapUI está basado en Java, por lo que se ejecuta en la mayoría de los sistemas operativos. Ha sido probado en varias versiones de Windows, así como en Mac y múltiples dialectos de Linux.
- SoapUI requiere una versión 1.6+ de JRE (Java Runtime Environment), se recomienda al menos 1 GB de memoria y aproximadamente 100 MB de espacio en disco.
- Para descargar e instalar accedemos a su página oficial:
  - https://www.soapui.org/downloads/soapui.html
- Si se está instalando con el instalador o las distribuciones independientes, el JRE está incluido y no es necesario en el sistema. De lo contrario, hay que asegúrese de que esté instalado y la variable de entorno JAVA_HOME esté configurada.

Interfaz de SoapUI



Log

- Las diferentes pestañas de log disponibles con las siguientes:
  - SoapUI log: Notificaciones generales y mensajes.
  - Http log: Muestra los datos enviados y recibidos por http. Deshabilitado durante las pruebas de stress.
  - Jetty log: Relacionado con las notificaciones de estado del mock-service.
  - Script log: Los scripts lanzan estos mensajes usando el objeto log disponible (está deshabilitado durante las pruebas de stress pero puede ser habilitado desde /File/Preferencess/UI settings).
  - Error log: Es un log con información sobre los errores ocurridos durante la ejecución. No tienen porque ser solo errores de soapUI, sino que pueden ser producidos por algún servicio o servidor que no esté disponible.
  - Memory Log: Muestra información del uso de la memoria.
- SoapUI utiliza log4j para crear los logs, es posible adaptar la configuración de log4j, renombrando el archivo log4j.xml, llamándolo “soapui-log4j.xml” y posteriormente moviéndolo al directorio bin de soapUI.

Espacio de trabajo

- En SoapUI, el trabajo se organiza en espacios de trabajo y proyectos. Un proyecto puede contener cualquier número de pruebas funcionales, pruebas de carga y simulaciones de servicio requeridas para los propósitos de prueba.
- SoapUI ofrece dos formatos de proyecto:
  - Proyectos independientes (predeterminado): los almacena como un único archivo XML que contiene todos los artefactos del proyecto, como interfaces, pruebas, servicios simulados, scripts, etc.
  - Proyectos compuestos (ReadyAPI): para equipos de prueba, esto permite que varias personas trabajen en el proyecto al mismo tiempo.

Proyectos

- Los proyectos SOAP se pueden crear desde un archivo WSDL o una llamada de servicio individual. Se puede usar estos proyectos para probar cada aspecto de sus servicios SOAP, verificar que los servicios sean compatibles con los estándares de uso común como WS-Security, WS-Addressing y MTOM, crear pruebas funcionales y de carga, y mucho más.
- Los proyectos REST se pueden crear desde un archivo WADL o, directamente, URI y sus parámetros. Se puede usar estos proyectos para probar servicios RESTful, crear varias solicitudes y verificar la información que recibe, probar una multitud de métodos y operaciones, etc.
- Se pueden importar proyectos existentes locales, compuestos, comprimidos o remotos. Al importar, SoapUI lo comprueba para verificar que sea coherente y que tenga todas las dependencias externas necesarias disponibles. Este proceso se llama resolución .

Propiedades del proyecto

- La propiedad de raíz de recursos controla cómo SoapUI maneja las rutas para los recursos del proyecto.
  - Un camino absoluto: Usa este camino absoluto.
  - $ {projectDir}: Resuelve archivos relativos a la carpeta del proyecto.
  - $ {workspaceDir}: Resuelve archivos relativos a la carpeta que contiene el archivo del espacio de trabajo
- Se pueden crear propiedades personalizadas.
- Se puede usar la propiedad del proyecto Contraseña del proyecto para cifrar todo el contenido del archivo del proyecto (el icono del proyecto contendrá un pequeño carácter E para indicar que se ha cifrado). Al abrir el proyecto SoapUI pedirá dicha contraseña. Para eliminar el cifrado, hay que borrar el valor de la contraseña del proyecto y guardar el proyecto. SoapUI no proporciona ningún medio para recuperar un archivo de proyecto cifrado si se ha perdido la contraseña.

Estructura del proyecto

- Interfaces
  - SOAP
  - REST
- Suite de pruebas (TestSuite)
  - Casos de prueba
    - Pasos de la prueba (pruebas funcionales)
    - Pruebas de carga
    - Pruebas de seguridad
- Simulaciones de servicio
  - SOAP MockServer
  - REST MockServer

Preparación de la prueba

- Importar definición y configurar endpoints:
  - Añadir WSDL (http://www.dneonline.com/calculator.asmx?wsdl)
  - Añadir WADL
  - Importar Swagger (https://petstore.swagger.io/v2/swagger.json)
- Explorar el servicio:
  - Cada servicio basado en WSDL expone una serie de operaciones que tienen un formato de mensaje de solicitud y respuesta (ambos opcionales).
  - Cada servicio REST expone una serie de URLs y el método de solicitud determina la operación a realizar. Las básicas son: POST, GET, PUT y DELETE; aunque SoapUI también admite solicitudes HEAD, OPTIONS, TRACE, PATCH, PROPFIND, LOCK, UNLOCK, COPY y PURGE.
- Preparar y probar peticiones al servicio
  - Para invocar una operación, puede agregar cualquier número de objetos de solicitud a una operación en el árbol del navegador.

Explorar el servicio

- Si al crear el proyecto se activa "Create Request" muestrea las posibles peticiones extraídas de las definiciones.
- Todas las peticiones se pueden completar con:
  - Auth: permite establecer el mecanismo de autenticación y la identidad con la que se realizará la petición.
  - Encabezados HTTP: permite agregar los valores a los encabezado de la solicitud
  - Adjuntos: permite agregar ficheros que se enviaran adjuntos con la solicitud
- Las peticiones SOAP adicionalmente se pueden completar con:
  - WS-A: las propiedades utilizadas para agregar encabezados WS-A a Request / MockResponse de acuerdo con la especificación WS Addressing.
  - WS-RM: las propiedades utilizadas para configurar y usar una secuencia WS-RM para la solicitud de acuerdo con la especificación de WS Reliable Messaging.

Parámetros REST

- Tipos de parámetros:
  - CONSULTA (QUERY): aparecen en la URL después del signo de interrogación (?) después del nombre del recurso (nombre=valor).
  - ENCABEZAMIENTO (HEADER): se pasan en los encabezados de las solicitudes salientes (encabezado: valor).
  - PLANTILLA (TEMPLATE): estos parámetros aparecen en la ruta encerrados entre {}.
  - MATRIX: estos parámetros también van en la URL de la solicitud. Residen entre la ruta del recurso y los parámetros QUERY, y están separados de la ruta del recurso por un punto y coma (;).
  - PLAIN: estos parámetros están presentes en el editor de solicitudes, pero SoapUI no los incluye en las solicitudes.
- Nivel de parámetro:
  - El nivel de RECURSO significa que el parámetro que cree se agregará a todos los elementos de método y solicitud en el elemento de recurso.
  - El nivel METHOD significa que el parámetro se agregará a los elementos de la solicitud debajo del elemento del método solamente. No afectará al recurso ni a otros elementos del método.

Opciones de parámetros

- SoapUI utiliza las siguientes opciones para los parámetros para configurar la solapa Forms y seleccionar el valor de una lista desplegable (solo en ReadyAPI):
  - Required: Establece si los pasos de prueba deben especificar el valor del parámetro o si puede omitirlo
  - Type: Especifica el tipo de datos del parámetro.
  - Options: Una lista de posibles valores para el parámetro que aparecen en la lista desplegable.
  - Description: Un texto libre que describe el parámetro y proporcionar una pista rápida para el parámetro en los editores de pasos de prueba.
  - Disable Encoding: Si los valores de un parámetro van en una URL y contienen algunos símbolos especiales como espacios o barras diagonales, SoapUI de forma predeterminada reemplazará estos símbolos con sus códigos (como %20 o %2F). En determinados casos, es posible que se desee desactivar dicha codificación marcando la casilla.

Representaciones

- Establece los diferentes formatos MIME en el que se puede solicitar u obtener las peticiones REST:
  - Type: Tipo de representación, puede tomar los siguientes valores:
    - REQUEST: formato del cuerpo de la solicitud
    - RESPONSE: formato del cuerpo de las respuestas correctas (2xx)
    - FAULT: formato del cuerpo de las respuestas fallidas (4xx y 5xx)
  - Media-Type: Formato MIME
  - Status Codes: Código numérico del estado HTTP (no aplicable en el tipo REQUEST)

Pruebas funcionales

- En SoapUI, las pruebas funcionales se pueden usar para validar requisitos funcionales, tanto para invocar los Servicios Web propios (=”pruebas unitarias”) como para una secuencia de peticiones (= " pruebas de integración"). Además, es posible añadir lógica a las pruebas mediante scripts de Groovy, lo que permite, por ejemplo, interactuar con una base de datos o realizar un flujo de pruebas complejo.
- Las pruebas funcionales, en SoapUI, se pueden usar para una variedad de propósitos:
  - Pruebas Unitarias: valida que cada operación del Servicio Web funciona como se indica
  - Pruebas de compatibilidad: valida que el resultado devuelto por el Servicio Web es compatible con su definición
  - Prueba de procesos: valida que una secuencia de invocaciones de Servicios Web ejecuta un proceso de negocio requerido
  - Pruebas guiadas por datos: valida que cualquiera de los anteriores funciona como requerimiento de datos de entrada procedentes de fuentes externas (por ejemplo, una base de datos u otro servicio Web)

Pruebas funcionales

- SoapUI soporta pruebas funcionales de Servicios Web suministrando un caso de prueba con un número de pasos que pueden ser ejecutados en secuencia. En la actualidad, hay seis tipos de pasos que proporcionan muchas posibilidades de prueba. Los casos de prueba están organizados en un grupo de pruebas y en un mismo proyecto se pueden crear varios grupos de pruebas.
- SoapUI estructura las pruebas funcionales en tres niveles: TestSuites, TestCases y TestSteps.
- Un TestSuite es una colección de TestCases que se pueden usar para agrupar pruebas funcionales en unidades lógicas. Se puede crear cualquier número de TestSuites dentro de un proyecto de SoapUI para admitir escenarios de pruebas masivas.
- Un TestCase es una colección de TestSteps que se ensamblan para probar algunos aspectos específicos de sus servicios. Puede agregar cualquier número de TestCases a el TestSuite que lo contenga e incluso modularizarlos para llamarse entre sí para escenarios de prueba complejos.
- TestSteps son los "bloques de construcción" de las pruebas funcionales en SoapUI. Se agregan a un TestCase y se utilizan para controlar el flujo de ejecución y validar la funcionalidad de los servicios que se probarán.

Tipos de Paso

- SOAP Request: Envía una petición SOAP y permite que la respuesta sea validada usando una variedad de aserciones
- REST Request: Ejecuta una Request Rest definida en el proyecto
- HTTP Request: Ejecuta una llamada http
- JDBC Request: Ejecuta una petición a una BBDD
- AMF Request: Ejecuta una petición AMF (es el formato de mensajería Adobe ActionScript utilizado por las aplicaciones Flash / Flex para interactuar con un servidor back-end).
- GraphQL Request: Ejecuta una consulta o una mutación en GraphQL.
- Properties: Se utiliza para definir propiedades globales que pueden ser leídas desde una fuente externa.
- Property Transfer: Utilizadas para transferir los valores de propiedad entre dos TestSteps
- Run TestCase Step: Ejecuta otro TestCase dentro de uno ya existente
- Conditional Goto: Permite cualquier número de saltos condicionales en la ruta de ejecución del TestCase.
- Delay Step: Pausa la ejecución de una TestCase durante un número especificado de milisegundos.
- Groovy Script: Ejecuta un script Groovy que puede hacer más o menos "cualquier cosa"
- Mock Response: Escucha un petición que se valida y devuelve una respuesta mock
- Manual TestStep: Se utiliza cuando se necesita interacción manual del tester dentro de la prueba
- Message Queuing Telemetry Transport: Receive MQTT Message, Publish using MQTT, Drop MQTT Connection

TestRequests

- Los TestRequests son una de las principales características cuando se trabaja con SoapUI.
- Extiende peticiones estándar con la posibilidad de añadir cualquier número de aserciones que se aplicarán a la respuesta recibida por la petición. Esto es, comprueba que la respuesta contenga lo que se espera que contenga.
- Los TestRequests son enviados manualmente a través de las acciones de envío de los editores o cuando se ejecuta el TestCase que contiene la petición.
- La respuesta de la petición es validada contra las aserciones de peticiones y el icono de la petición cambia para reflejar el resultado de la validación; verde significa que todas las validaciones fueron bien y rojo que algunas fallaron.
- Un icono con el fondo gris indica que la petición todavía no ha sido enviada para validar, un fondo blanco indica que los TestRequests carecen de aserciones.

Petición

- SOAP Request:
  - Vinculada a operación del interfaz contiene el fragmento SOAP de la petición en XML para su edición.
- REST Request:
  - Vinculada a un método del interfaz contiene los parámetros de solicitud HTTP para la asignación de sus valores.
- HTTP Request:
  - Permite crear una petición a una URL usando el método indicado y parámetros de solicitud HTTP para la asignación de sus valores.

Petición

- Todas las peticiones se pueden completar con:
  - Auth : permite establecer el mecanismo de autenticación y la identidad con la que se realizará la petición.
  - Encabezados HTTP: permite agregar los valores a los encabezado de la solicitud
  - Adjuntos: permite agregar ficheros que se enviaran adjuntos con la solicitud
- Las peticiones SOAP adicionalmente se pueden completar con:
  - WS-A : las propiedades utilizadas para agregar encabezados WS-A a Request / MockResponse de acuerdo con la especificación WS Addressing.
  - WS-RM : las propiedades utilizadas para configurar y usar una secuencia WS-RM para la solicitud de acuerdo con la especificación de WS Reliable Messaging.

Respuesta

- Una vez ejecutada manualmente la petición se mostrara la respuesta. Admite diferentes formatos: XML, JSON, HTML, Raw.
- Así mismo permite inspeccionar:
  - Encabezados HTTP : muestra los encabezado HTTP de la respuesta.
  - Adjuntos: muestra los adjuntos de la respuesta asociada.
  - Información SSL: muestra la respuesta detallada SSL para la respuesta actual.
  - WS-A: muestra las propiedades del encabezado WS-Addressing.
  - WSS: muestra las propiedades del encabezado WS-Security

Aserciones

- Las aserciones se utilizan para validar el mensaje recibido por un TestStep durante la ejecución, generalmente comparando partes del mensaje (o el mensaje completo) con algún valor esperado.
- Se puede agregar cualquier cantidad de aserciones a una muestra de TestStep, cada una validando algún aspecto o contenido diferente de la respuesta.
- Después de que se ejecuta una muestra TestStep, todas las aserciones se aplican a la respuesta recibida y si alguna de ellas falla, el TestStep se marca como fallido en la vista del TestCase y se muestra una entrada FALLO correspondiente en el Registro de ejecución de prueba.
- Las aserciones se pueden deshabilitar para que no se apliquen.
- Las aserciones se dividen en varias categorías para facilitar la gestión. Solo se habilitan las categorías que contienen aserciones aplicables para un tipo específico de muestra.

Categoría: contenido de propiedad

- Contains: busca la existencia de un token de cadena en el valor de la propiedad, admite expresiones regulares. Aplicable a cualquier propiedad.
- Not Contains: busca la inexistencia de un token de cadena en el valor de la propiedad, admite expresiones regulares. Aplicable a cualquier propiedad.
- XPath Match: utiliza una expresión XPath para seleccionar contenido de la propiedad de destino y compara el resultado con un valor esperado. Aplicable a cualquier propiedad que contenga XML.
- XQuery Match: utiliza una expresión XQuery para seleccionar contenido de la propiedad de destino y compara el resultado con un valor esperado. Aplicable a cualquier propiedad que contenga XML.

Categoría: contenido de propiedad

- JsonPath Count: usa una expresión JsonPath para contar las ocurrencias de un elemento. Aplicable a cualquier propiedad que contenga JSON.
- JsonPath Existence Match: utiliza una expresión JsonPath para seleccionar contenido de la propiedad de destino y compara el resultado con un valor esperado. Aplicable a cualquier propiedad que contenga JSON.
- JsonPath Match: utiliza una expresión JsonPath para seleccionar contenido de la propiedad de destino y compara el resultado con un valor esperado. Si los valores coinciden con las afirmaciones pasadas supera la prueba, de lo contrario falla. Aplicable a cualquier propiedad que contenga JSON.
- JsonPath RegEx Match: utiliza una expresión JsonPath para seleccionar contenido de la propiedad de destino y compara el resultado con una expresión regular especificada. Aplicable a cualquier propiedad que contenga JSON.

Categoría: Cumplimiento, estado y estándares

- HTTP Download all resource: descarga todos los recursos referidos como un documento HTML (imágenes, scripts, etc.) y valida que estén disponibles. Aplicable a cualquier propiedad que contenga HTML.
- Valid HTTP Status Codes: comprueba que se recibió un resultado HTTP con un código de estado en la lista de códigos definidos. Aplicable a cualquier TestStep que recibe mensajes HTTP.
- Invalid HTTP Status Codes: comprueba que se recibió un resultado HTTP con un código de estado que no figura en la lista de códigos definidos. Aplicable a cualquier TestStep que recibe mensajes HTTP
- SOAP Response: valida que la última respuesta recibida es una respuesta SOAP válida. Aplicable solo a los pasos de solicitud de prueba SOAP.
- SOAP Request: valida que la última solicitud recibida es una solicitud SOAP válida. Aplicable solo a MockResponse TestSteps.
- Not SOAP Fault: valida que el último mensaje recibido no es un fallo SOAP. Aplicable a SOAP TestSteps.

Categoría: Cumplimiento, estado y estándares

- SOAP Fault: valida que el último mensaje recibido es un fallo SOAP. Aplicable a la solicitud SOAP TestSteps
- Schema Compliance: valida que el último mensaje recibido sea compatible con la definición de esquema WSDL o WADL asociada. Aplicable a SOAP y REST TestSteps.
- WS-Addressing Request: valida que la última solicitud recibida contiene encabezados de direccionamiento WS válidos. Aplicable solo a MockResponse TestSteps.
- WS-Addressing Response: valida que la última respuesta recibida contiene encabezados de WS-Addressing válidos. Aplicable solo a los pasos de solicitud de prueba SOAP.
- WS-Security Status: valida que el último mensaje recibido contenía encabezados WS-Security válidos. Aplicable a SOAP TestSteps.

Otras categorías

- Script
  - Script Assertion: ejecuta una secuencia de comandos personalizada para realizar validaciones arbitrarias. Aplicable solo a TestSteps (es decir, no a las propiedades).
- SLA
  - Response SLA: valida que el último tiempo de respuesta recibido estuvo dentro del límite definido. Aplicable a Script TestSteps y TestSteps que envían solicitudes y reciben respuestas.
- JMS
  - JMS Status: valida que la solicitud JMS del TestStep objetivo se ejecutó correctamente. Aplicable a Request TestSteps con un punto final JMS.
  - JMS Timeout: valida que la instrucción JMS del TestStep de destino no tardó más de la duración especificada. Aplicable a Request TestSteps con un punto final JMS.
- JDBC
  - JDBC Status: valida que la instrucción JDBC del TestStep objetivo se ejecutó correctamente. Aplicable solo a JDBC TestSteps.
  - JDBC Timeout: valida que la instrucción JDBC del TestStep de destino no tardó más de la duración especificada. Aplicable solo a JDBC TestSteps.
- Security
  - Sensitive Information Exposure: comprueba que el último mensaje recibido no expone información confidencial sobre el sistema de destino. Aplicable a REST, SOAP y HTTP TestSteps.

Verificación SOAP

- SOAP Response
- Schema Compliance
- XPath Match
  - XPath Expression:

    declare namespace ns1='http://tempuri.org/';

    //ns1:AddResult
- XQuery Math
  - XQuery Expression:

    count(//Row)
- SOAP Fault

Verificación REST

- JsonPath Count
  - JsonPath Expression: $[*]
- JsonPath RegEx Match:
  - JsonPath Expression: $[0]
- JsonPath Existence Match
  - JsonPath Expression: $.id
- JsonPath RegEx Match:
  - JsonPath Expression: $.id
  - Regular Expression: \d
- Valid HTTP Codes:
  - Codes: 200, 201, 204
- Schema Compliance

Propiedades

- Las propiedades son un aspecto central de las pruebas más avanzadas con SoapUI.
- En lo que respecta a las propiedades de pruebas funcionales, se utilizan para parametrizar la ejecución y la funcionalidad de sus pruebas, por ejemplo:
  - Las propiedades se pueden utilizar para mantener los puntos finales de sus servicios, lo que facilita el cambio de los puntos finales reales utilizados durante la ejecución de la prueba (consulte el ejemplo a continuación).
  - Las propiedades se pueden usar para mantener las credenciales de autenticación, lo que facilita su administración en un lugar central o en un archivo externo.
  - Las propiedades se pueden usar para transferir y compartir identificadores de sesión durante la ejecución de la prueba, por lo que múltiples pasos de prueba o casos de prueba pueden compartir las mismas sesiones.
- Las propiedades se pueden definir en varios niveles en SoapUI:
  - En el nivel Proyecto, TestSuite y TestCase
  - En un Properties TestStep
  - En un DataGen TestStep
  - Como parte de una configuración de TestStep (DataSource TestStep y DataSink TestStep)
- Además, la mayoría de los otros TestSteps exponen propiedades para lectura y escritura, por ejemplo:
  - Script TestStep expone una propiedad "Result" que contiene el valor de cadena devuelto por el último script ejecutado.
  - Todos los pasos de Request exponen una propiedad con la última respuesta recibida.
- Las propiedades pueden leerse y escribirse fácilmente desde scripts y también transferirse entre TestSteps con Property-Transfer TestStep

Properties TestStep

- El Properties TestStep se utiliza para definir propiedades personalizadas que se utilizarán dentro de un TestCase.
- Sus principales ventajas sobre la definición de propiedades en el nivel TestCase son:
  - Se pueden organizar las propiedades en varios Properties TestStep (si se tienen muchas de ellas).
  - Se puede especificar los nombres de los archivos de origen y destino que se usarán para leer y escribir las propiedades contenidas cuando se ejecute TestStep. En formato .properties:

    Propiedad1=Valor1

    Propiedad2=Valor2

Property Transfer TestStep

- Los Property Transferes son TestStep que transfieren propiedades entre los contenedores de propiedades dentro del mismo alcance que el Property Transfer TestStep (es decir, que contenga su TestCase, su TestSuite, el proyecto y las propiedades del destino).
- El paso puede contener un número arbitrario de "transferencias" especificando una propiedad de origen y destino con expresiones opcionales de XPath/XQuery.
- Property Transferes utiliza el mismo motor de Saxon XPath/XQuery descrito para las aserciones XPath y XQuery.
- Tras la ejecución de un TestCase, cada transferencia se realiza mediante la selección de las propiedades especificadas por el paso de origen, por la propiedad y expresión XPath opcional y copiando su valor a la propiedad especificada por el paso de destino pudiendo usar una expresión Xpath opcional.
- Si las expresiones XPath están especificadas, SoapUI tratará de sustituir el nodo destino con el nodo de origen si son del mismo tipo. Si no (por ejemplo, cuando se asigna texto () a un @atributo), SoapUI hará todo lo posible para copiar el valor.
- El origen de la transferencia puede ser la respuesta de un paso anterior.
- El destino puede ser el contenido de un paso posterior, pero ser mas cómodo el uso de la expansión de propiedades.

Propiedades de Expansión

- SoapUI proporciona una sintaxis común para insertar dinámicamente ("expandir") valores de propiedad durante el procesamiento. La sintaxis es la siguiente:
  - ${[scope]propertyName[#xpath-expression]}
- donde alcance puede ser uno de los siguientes valores literales:
  - #Project#: hace referencia a una propiedad del Proyecto
  - #TestSuite#: hace referencia a una propiedad en el TestSuite que lo contiene
  - #TestCase#: hace referencia a una propiedad en el TestCase que lo contiene
  - #MockService#: hace referencia a una propiedad en el MockService que lo contiene
  - #Global#: hace referencia a una propiedad global en todos los proyectos. En File>Preferences>Global Properties.
  - #System#: hace referencia a una propiedad del sistema. En Help>System properties.
  - #Env#: hace referencia a una variable de entorno
  - [Nombre de TestStep]#: hace referencia a una propiedad TestStep
- Si no se especifica ningún alcance, la propiedad se resuelve de la siguiente manera:
  - Mira en el contexto actual (por ejemplo, TestRunContext) una propiedad con el nombre coincidente
  - Busca una propiedad global coincidente
  - Comprueba si hay una propiedad del sistema coincidente

Propiedades de Expansión

- Si la expansión de la propiedad incluye además una expresión XPath, se usará para seleccionar el valor correspondiente del valor de la propiedad referenciada (que debe contener XML):
  - ${Search Request#Response#//ns1:Item[1]/n1:Author[1]/text()}
- SoapUI 2.5 introdujo la posibilidad de escribir scripts directamente dentro de una PropertyExpansion; si se prefija el contenido con un '=' el contenido restante hasta la llave de cierre se evaluará como un script y se insertará su resultado:
  - ${=(int)(Math.random()*1000)}
- Dependiendo del contexto de la expansión, las variables relevantes estarán disponibles para acceder al modelo de objeto SoapUI.
  - $ {= request.name}
- Las siguientes variables están (casi) siempre disponibles en estos scripts:
  - log: un Logger log4j que registra en la ventana de registro
  - modelItem: el modelItem actual (por ejemplo, Request, MockResponse, etc.).
  - context: el contexto de ejecución actual (por ejemplo, un TestCase o MockService)

JDBC Request

- SoapUI 3.5 presento un nuevo TestStep para recuperar datos de una base de datos utilizando JDBC. El resultado está formateado como XML y se puede afirmar o procesar de la manera estándar (XPath , Xquery, …), pero también hay dos aserciones adicionales disponibles:
  - Afirmación de estado JDBC
  - La aserción JDBC Timeout verificará que el SQL se ejecutó dentro del tiempo de espera configurado.
- Para utilizar JDBC TestStep se deberá agregar el controlador JDBC a la carpeta soapui_home/bin/ext y reiniciar la aplicación.
- Para configurar la conexión:
  - Driver: com.mysql.jdbc.Driver
  - Connection String: jdbc:mysql://localhost:3306/sakila?user=root&password=root
- La consulta SQL, los parámetros pueden ser referenciados mediante la expansión de propiedades:
  - SELECT count(*) FROM `sakila`.`actor` WHERE `actor_id`=:id

Run TestCase TestStep

- Si los escenarios de prueba se vuelven cada vez más complejos, es posible que se desee compartir una serie de TestSteps entre diferentes TestCases, tal vez para configurar algunos requisitos previos (inicio de sesión, etc.) o para realizar ciertos pasos en diferentes contextos.
- Run TestCase TestStep presenta un enfoque flexible para la modularización; ejecuta el TestCase objetivo configurado opcionalmente, pasando y recuperando propiedades antes y después de la ejecución, por ejemplo, se podría usar esto para llamar a una secuencia compleja de TestSteps para realizar y validar un procedimiento de inicio de sesión, pasando las credenciales requeridas y recibiendo el sessionid resultante.

Conditional Goto TestStep

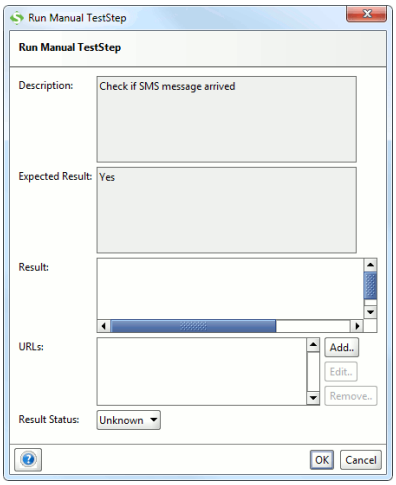
- Los pasos de condicional Goto evalúan un número arbitrario de condiciones XPath del mensaje de respuesta de la petición anterior y transfieren la ejecución del TestCase al TestStep asociado con la primera condición que se evalúe a verdadero.
- Esto permite la ejecución condicional de rutas de TestCase, donde el resultado de algunas peticiones controla cómo moverse por el TestCase.
- Si no hay condiciones que coincidan con la respuesta actual, la ejecución del TestCase continúa después del paso Goto de forma habitual.
- Ejemplo de escenarios:
  - Ramificaciones que dependen de los resultados devueltos por una petición
  - Reiniciar después de un largo retraso en las pruebas de vigilancia
  - En varias ocasiones esperar y chequear el valor de estado antes de continuar (por ejemplo en un proceso por lotes)
- Las condiciones usan el mismo motor que Saxon XPath descrito para las aserciones Xpath (una condición debe devolver un valor booleano para ser válida)

Delay TestStep

- Los TestStep de retardo se pueden insertar en cualquier posición en un TestCase para pausar la ejecución de un TestCase durante un número determinado de milisegundos.
- Ejemplos de escenarios:
  - Repetir varias veces el TestCase o determinados TestStep después de una demora de tiempo.
  - Demorar los siguientes pasos para dar tiempo a que un proceso ajeno se complete.

Manual TestStep

- Manual TestStep se utiliza cuando se necesita interacción manual del tester dentro de la prueba. Algunos ejemplos:
  - Iniciar dispositivos como teléfonos movil
  - Realizar una acción en el sitio web o en el servidor
  - Verificar manualmente la base de datos / registros para
- Para definirlo se puede especificar una descripción y el resultado esperado. Cuando la ejecución del caso de prueba alcanza un paso de prueba Manual, se muestra una ventana emergente con la descripción y el resultado esperado y permite agregar los detalles de los resultados reales y especificar el botón de opción Pasar-Fallar-N/A para establecer el estado del paso de prueba. Al hacer clic en continuar, SoapUI va al siguiente paso.
- Estos pasos de prueba manuales, se omiten cuando se inicia a través de testrunner ya que no hay forma de interactuar con ventanas emergentes.



Script Step

- SoapUI ofrece amplias opciones para la creación de secuencias de comandos, utilizando Groovy o Javascript (desde SoapUI 3.0) como su lenguaje de secuencias de comandos, que se utiliza para establecer el nivel del proyecto en la pestaña de detalles del proyecto en la parte inferior izquierda.
- Todos los scripts tienen acceso a una serie de variables específicas de la situación, que siempre incluyen un objeto de registro para iniciar sesión en Groovy Log y un objeto de contexto para realizar PropertyExpansions específicas de contexto o manejo de propiedades, si corresponde.
- Una variable específica de contexto siempre está disponible para acceder directamente al modelo de objetos SoapUI.

Script Step

- Los scripts se pueden usar en los siguientes lugares en SoapUI:
  - Como parte de un TestCase con Script TestStep, que permite que las pruebas realicen prácticamente cualquier funcionalidad deseada
  - Antes y después de ejecutar un TestCase o TestSuite para inicializar y limpiar antes o después de ejecutar sus pruebas.
  - Al iniciar/detener un MockService para inicializar o limpiar el estado de MockService
  - Para proporcionar despacho dinámico de MockOperation o crear contenido dinámico de MockResponse
  - Al abrir / cerrar un proyecto, para inicializar o limpiar configuraciones relacionadas con el proyecto
  - Como un DataSource o DataSink dinámico con los pasos de prueba de DataSource / DataSink correspondientes
  - Para crear aserciones arbitrarias con la aserción de script
  - Para extender SoapUI y agregar funcionalidad arbitraria al núcleo SoapUI.

SOAP Mock Response

- El paso SOAP Mock Response escucha una solicitud SOAP y devuelve una respuesta preconfigurada antes de continuar. La solicitud entrante se puede validar con las mismas aserciones de las solicitudes SOAP, específicamente con SOAP Request y Schema Compliance se puede validar la petición entrante.
- Cuando la ejecución de un caso de prueba llega al paso de prueba SOAP Mock Response, la ejecución de la prueba se detiene y espera una solicitud al recurso especificado en el puerto configurado con la ruta especificada. Una vez que se recibe una solicitud, SOAP Mock Response devuelve el resultado y continua con siguiente paso en el caso de prueba.
- Dichos servicios mock temporales pueden capturar solicitudes enviadas por webhooks. De esta manera, se puede configurar rápidamente un servicio temporal para capturar una sola solicitud, o hacer que esto sea parte de la prueba sin configurar una aplicación o una API virtual separada para manejar la solicitud.

Pruebas de carga

- Las pruebas de rendimiento son una de las tareas más comunes en las pruebas de servicios web. Lo que se llama una prueba de carga en SoapUI es en realidad una prueba de rendimiento.
- Las pruebas de rendimiento crean o simulan artificialmente la carga y miden cómo la maneja el entorno.
- Esto significa que no necesariamente tiene que ser el rendimiento de un sistema bajo una carga alta, sino también el rendimiento bajo la carga base o la carga esperada.
- En SoapUI, se crean las pruebas de carga sobre la base de pruebas funcionales existentes. Esto ayuda a crear rápida y fácilmente pruebas de rendimiento para el servicio web. Luego se puede validar el rendimiento del servicio web utilizando diferentes estrategias de carga, verificar que su funcionalidad no se rompa bajo carga, ejecutar varias pruebas de carga simultáneamente para ver cómo se afectan entre sí y mucho más.

Pruebas de carga

- Pruebas de línea de base
  - Técnicamente hablando, esto se puede definir como pruebas de rendimiento puro. Las pruebas de línea de base examinan cómo funciona un sistema bajo la carga esperada o normal y crea una línea de base con la que se pueden comparar otros tipos de pruebas.
  - Objetivo: encontrar métricas para el rendimiento del sistema bajo carga normal.
- Prueba de carga
  - La prueba de carga incluye aumentar la carga y ver cómo se comporta el sistema bajo una carga mayor. Durante las pruebas de carga, puede monitorear los tiempos de respuesta, el rendimiento, las condiciones del servidor y más. Sin embargo, el objetivo de las pruebas de carga no es romper el entorno de destino.
  - Objetivo: encontrar métricas para el rendimiento del sistema bajo alta carga.
- Prueba de esfuerzo o estrés
  - La prueba de esfuerzo consiste en aumentar la carga hasta encontrar el volumen de carga donde el sistema realmente se rompe o está cerca de romperse.
  - Objetivo: encontrar el punto de ruptura del sistema.

Pruebas de carga

- Pruebas de remojo
  - Para encontrar inestabilidades del sistema que ocurren con el tiempo, debe ejecutar pruebas durante un período prolongado. Para eso están las pruebas de remojo; ejecute pruebas de carga o incluso pruebas de línea de base durante un largo período de tiempo y vea cómo el entorno de destino maneja los recursos del sistema y si funciona correctamente. El defecto más común encontrado en las pruebas de remojo son las pérdidas de memoria. El escenario más común para las pruebas de remojo es activar una serie de pruebas cuando sale de su oficina un viernes y lo deja correr durante el fin de semana.
  - Objetivo: Asegurarse de que no surja ningún comportamiento no deseado durante un largo período de tiempo.
- Pruebas de escalabilidad
  - El propósito de las pruebas de escalabilidad es verificar si su sistema se adapta adecuadamente a la carga cambiante. Por ejemplo, un mayor número de solicitudes entrantes debería causar un aumento proporcional en el tiempo de respuesta. El aumento no proporcional indica problemas de escalabilidad.
  - Objetivo: buscar métricas y verificar si el rendimiento del sistema cambia adecuadamente a la carga.

Factores que afectan al rendimiento

- Si observamos características únicas del rendimiento del servicio web, se destacan dos aspectos: una gran cantidad de procesamiento XML / JSON en el lado del servidor (análisis y serialización XML / JSON) y procesamiento de cargas (cuerpos de solicitud y respuesta). Las razones por las cuales esto falla pueden ser múltiples; puede estar en la plataforma en forma de, por ejemplo, debilidades en el servidor de aplicaciones, y en la implementación en forma de WSDL innecesariamente complejos. También podría ser que el código está haciendo una solicitud a una base de datos que responde lentamente.
- Un factor más que afecta el rendimiento con frecuencia es la seguridad. Aquellos que realizan pruebas de rendimiento web saben que los sitios HTTPS tienen un rendimiento considerablemente menor que sus análogos HTTP, y en las pruebas de servicios web, podemos agregar una capa de WS-Security a la capa de seguridad HTTP, disminuyendo aún más el rendimiento.
- La complejidad de analizar las cargas útiles de XML / JSON significa que tenemos que poner un enfoque adicional en las pruebas de escalabilidad, mientras que el problema de la seguridad significa que tendremos que enfocarnos un poco más en hacer pruebas de solicitudes que son seguras. Si todo el servicio web es seguro, esto significa que la prueba de carga es más importante, especialmente si se utiliza WS-Security y el manejo de tokens.
- También significa que tenemos que examinar la especificación API de cerca. Si las solicitudes y respuestas son complejas o grandes, o si incluyen archivos adjuntos grandes, debemos centrarnos en enfatizar esa complejidad y ver cómo se comporta bajo carga.

LoadTest

- En SoapUI, los LoadTests se basan en los TestCases funcionales existentes. Un LoadTest ejecuta el TestCase repetidamente durante el tiempo especificado con un número deseado de subprocesos (o usuarios virtuales). Los LoadTests se muestran en el navegador como elementos secundarios del TestCase que usan. Se puede crear cualquier número de LoadTests para un TestCase.
- Las diferentes estrategias de carga disponibles en SoapUI permiten simular varios tipos de carga a lo largo del tiempo, lo que permite probar fácilmente el rendimiento de sus servicios de destino en una serie de condiciones. Dado que SoapUI también le permite ejecutar múltiples LoadTests simultáneamente, se puede usar una combinación de LoadTests para afirmar aún más el comportamiento de los servicios.
- Si queremos simular varios clientes consumiendo el servicio, vamos a la opción threads, por defecto aparecen 5 pero se puedan variar ese número a discreción, aunque mientras más hilos más consumo del CPU. Si queremos especificar una demora entre una petición y otra vamos a la opción delay (1000 milisegundos por defecto) y la cantidad aleatoria de la demora que se desea aleatorizar (es decir, establecerlo en 0.5 dará como resultado retrasos entre 0,5 y 1 segundo)
- Si queremos determinar el tiempo de duración del servicio vamos a la opción Limit y aquí se muestra que solo durará 60 segundos pero pueden cambiarla para especificar la cantidad de peticiones por hilo por ejemplo.

Estrategias

- Estrategia simple: pruebas de línea de base, carga y remojo
  - La estrategia simple ejecuta la cantidad especificada de subprocesos con el retraso especificado entre cada ejecución para simular un respiro para el servidor.
- Estrategia de tasa fija (Fixed Rate): simple con un toque especial
  - La estrategia de tasa fija ejecuta una cantidad de subprocesos dentro de un tiempo determinado, la estrategia iniciará automáticamente el número de subprocesos requeridos intentando mantener el valor configurado.
- Estrategias de carga variable (Variance)
  - La estrategia de variación cambia el número de subprocesos a lo largo del tiempo en "dientes de sierra" según lo configurado; se establece el Intervalo en el valor deseado y la Varianza en cuánto debe disminuir y aumentar el número de subprocesos.
- Estrategias de carga variable (Burst)
  - La estrategia Burst está diseñada específicamente para pruebas de recuperación y lleva la variación al extremo. No hace nada durante el retardo de ráfaga configurado, luego ejecuta la cantidad configurada de subprocesos para la "Duración de ráfaga" y vuelve al modo de suspensión.

Ejecución de LoadTest

- SoapUI permite ejecutar el LoadTest con tantos subprocesos (o usuarios virtuales) como el hardware pueda administrar, dependiendo principalmente de la memoria, la CPU, el tiempo de respuesta del servicio objetivo y otros factores. Establezca el valor deseado e inicie LoadTest con el botón Ejecutar en la barra de herramientas del editor LoadTest.
- El TestCase subyacente se clona internamente para cada subproceso configurado y se inicia en su propio contexto; scripts, transferencias de propiedades, etc. El TestCase accederá a una "copia" única de TestCase y TestSteps que evita problemas de subprocesos en el nivel TestStep y TestCase (pero no más arriba en la jerarquía).
- A medida que se ejecuta LoadTest, la tabla de estadísticas de LoadTest se actualiza continuamente con los datos recopilados cada vez que finaliza un TestCase ejecutado, lo que le permite controlar de forma interactiva el rendimiento de sus servicios de destino mientras se ejecuta LoadTest. Si el TestCase contiene un bucle o TestSteps de ejecución prolongada, las estadísticas pueden tardar un tiempo en actualizarse (ya que se recopilan cuando finaliza TestCase), en este caso, seleccione la opción "Estadísticas TestStep" en el cuadro de diálogo Opciones de LoadTest para actualizarse estadísticas en el nivel TestStep en lugar del nivel TestCase (esto requiere un poco más de procesamiento interno, por eso está desactivado de forma predeterminada).

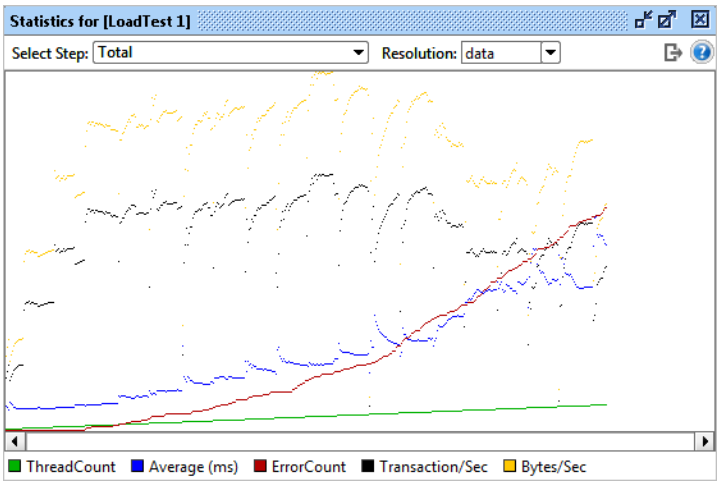
Estadísticas

- Tiempos mínimos (min), máximos (max), medios (avg) y de la última petición (last), todos en milisegundos.
- Número de ejecuciones (cnt), total de bytes (bytes), número errores (err) y el ratio de errores (rat).
- TPS (transacciones por segundo) y BPS (bytes por segundo) se pueden calcular de dos maneras diferentes (Opciones de LoadTest);
  - Según el tiempo real transcurrido (predeterminado):
    - TPS: cnt/segundos pasados, es decir, un TestCase que se ejecutó durante 10 segundos y manejó 100 solicitudes obtendrá un TPS de 10
    - BPS: bytes/tiempo transcurrido, es decir, un TestCase que se ejecutó durante 10 segundos y manejó 100000 bytes obtendrá un BPS de 10000.
  - Basado en el tiempo promedio de ejecución:
    - TPS: (1000/avg) * número de hilos, por ejemplo avg = 100 ms con diez hilos dará un TPS de 100
    - BPS: (bytes/cnt) * TPS, es decir, el número promedio de bytes por solicitud * TPS.
- Se pueden exportar a ficheros .csv

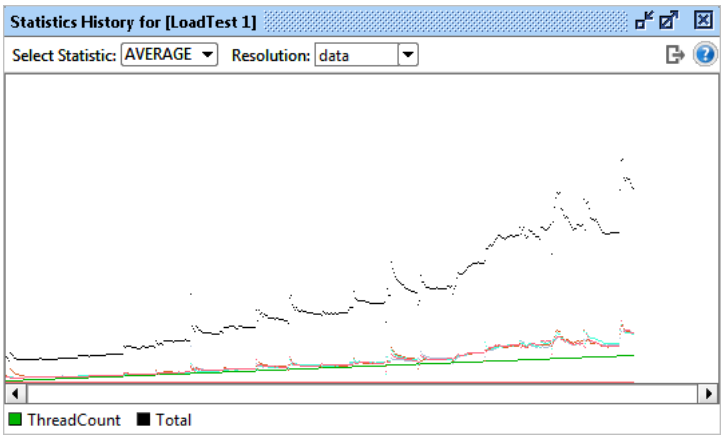
Gráficos estadísticos

- Hay dos tipos de gráficos disponibles en la barra de herramientas de LoadTest durante la ejecución; estadísticas e historial de estadísticas.
- El objetivo principal de estos es visualizar estadísticas seleccionadas a lo largo del tiempo para poder detectar cambios repentinos e inesperados. La visualización de estadísticas es relativa (no absoluta) y, por lo tanto, los gráficos no son muy útiles para analizar datos exactos.
- Ambos gráficos tienen una configuración de Resolución que controla con qué frecuencia se actualiza el gráfico; establecer esto en "datos" actualizará el gráfico con el mismo intervalo que la Tabla de estadísticas (que son los datos subyacentes para el gráfico, de ahí el nombre). Alternativamente, si desea una de las resoluciones fijas, seleccione estas.
- El gráfico de estadísticas muestra todas las estadísticas relevantes para un paso seleccionado o el caso de prueba completo a lo largo del tiempo, lo que le permite ver cómo cambian los valores cuando, por ejemplo, aumenta el número de subprocesos.
- El gráfico de historial de estadísticas muestra un valor estadístico seleccionado para todos los pasos que le permite compararlos y ver si la distribución de cualquier valor entre los pasos de prueba cambia con el tiempo.
- Sus datos se pueden exportar a ficheros .csv

Gráficos estadísticos



Gráficos estadísticos



Validando el rendimiento

- Al igual que puede agregar aserciones funcionales a ciertos TestSteps en un TestCase (para afirmar sus resultados), se pueden agregar aserciones a un LoadTest para validar tanto el rendimiento como la funcionalidad durante la ejecución de LoadTest. Las aserciones configuradas se aplican continuamente a los resultados de LoadTest y puede fallar el LoadTest al igual que su contraparte funcional.
- Las aserciones disponibles son:
  - Step Average: valida que el valor promedio de un TestStep o todo el TestCase no excede el límite especificado.
  - Step TPS: valida el valor de TPS (transacción por segundo) para el TestStep o TestCase correspondiente.
  - Step Maximum: valida el valor tiempo máximo para el TestStep o TestCase correspondiente.
  - Step Status: comprueba que el estado de ejecución subyacente del TestStep o TestCase sea exitoso; de lo contrario, la aserción falla.
  - Max Errors: compruebe que el número de fallos para el TestStep o TestCase correspondiente no exceda el valor configurado.

Pruebas de seguridad

- Las funciones de prueba de seguridad introducidas en SoapUI 4.0 facilitan enormemente la validación de la seguridad funcional de los servicios de destino, lo que permite evaluar la vulnerabilidad del sistema para los ataques de seguridad comunes. Esto es especialmente crítico si el sistema está disponible públicamente, pero incluso si ese no es el caso, garantizar un entorno completamente seguro es igualmente importante.
- Los siguientes análisis de seguridad están disponibles actualmente:
  - Inyección SQL: intenta aprovechar la mala codificación de integración de bases de datos
  - Inyección XPath: intenta explotar el procesamiento XML incorrecto dentro de su servicio de destino
  - Escaneo de límites: intenta aprovechar el mal manejo de valores que están fuera de los rangos definidos
  - Tipos no válidos: intenta aprovechar el manejo de datos de entrada no válidos
  - XML con formato incorrecto: intenta aprovechar el mal manejo de XML no válido en su servidor o en su servicio
  - XML Bomb: intenta aprovechar el mal manejo de solicitudes XML maliciosas (tenga cuidado)
  - Adjunto malicioso: intenta aprovechar el mal manejo de los archivos adjuntos
  - Cross Site Scripting: intenta encontrar vulnerabilidades de cross-site scripting
  - Secuencia de comandos personalizada: le permite utilizar una secuencia de comandos para generar valores de fuzzing de parámetros personalizados

Data Driven Testing

- Las pruebas basadas en datos son aquellas que almacenan los datos de la prueba (entrada, salida esperada, etc.) en algún almacenamiento externo (base de datos, hoja de cálculo, archivos xml, etc.) y luego usa esos datos de forma iterativa en las pruebas cuando se ejecutan.
- Las pruebas basadas en datos solo están disponibles en ReadyAPI y crearlas es realmente fácil.
- El DataSource TestStep permite la lectura de los datos de prueba en propiedades SoapUI estándar a partir de una serie de fuentes externas (archivos de Excel, propiedades XML, fuentes JDBC, archivos/directorios, etc.), estas propiedades se pueden utilizar en los siguientes TestSTEPS (solicitudes, aserciones, consultas XPath, scripts, etc.), ya sea a través de Transferencias de propiedad o Expansiones de propiedad. Finalmente un DataSource Loop TestStep permite recorrer la fuente de datos para aplicar los TestSteps anteriores para los datos de cada fila/registro del DataSource.
- Esto se reduce a la siguiente configuración:
  1. DataSource: lee datos de prueba en propiedades de una fuente externa
  2. TestSteps (cualquier número): prueba la funcionalidad del servicio utilizando las propiedades DataSource disponibles en (1)
  3. DataSource Loop: realiza un bucle de regreso a (2) para cada fila en (1)

Ejecución

- Ejecución de prueba
  - Todas las vistas anteriores tienen una barra de herramientas en la parte superior con botones que ejecutan los elementos de prueba contenidos.
- Salida de la Prueba
  - Todas las vistas anteriores también contienen un registro de ejecución en la parte inferior que muestra información continua sobre los pasos de prueba ejecutados y su estado.
- Informes
  - ReadyAPI también agrega un botón "Crear informe" a la barra de herramientas superior, lo que le permite exportar los resultados de la ejecución actual a un documento, por ejemplo, un PDF.

TestRunner Command-Line

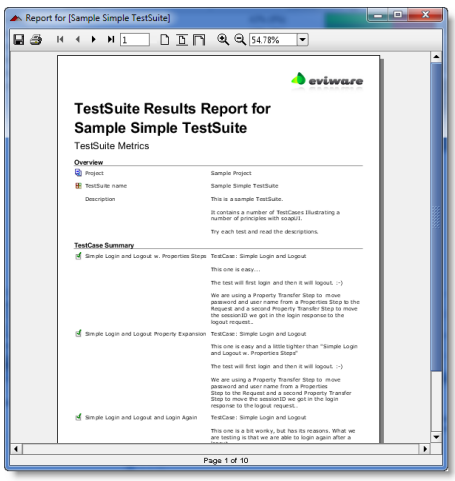
- Una vez creado, es posible que se desee ejecutar los Tests desde la línea de comandos, tal vez como parte de una compilación de integración continua o para monitorear el rendimiento diario de sus servicios.
- SoapUI proporciona utilidades de línea de comandos y complementos para que maven haga esto.
- Las utilidades están disponible en la carpeta SoapUI\bin (.bat o .sh):
  - testrunner.bat
  - loadtestrunner.bat
  - securitytestrunner.bat
  - mockservicerunner.bat
- Se necesitan muchos argumentos relacionados con informes, propiedades, etc., que pueden hacer que la creación de la lista de argumentos inicial sea bastante tediosa. Afortunadamente, la interfaz de usuario incluye un asistente para iniciar LoadTestRunner, que ayudará a evaluar la salida, así como a crear una cadena de invocación de línea de comandos.

https://www.soapui.org/test-automation/running-from-command-line/functional-tests.html

Informes

- ReadyAPI proporciona tres tipos de informes que se pueden generar desde el interior de la interfaz de usuario a nivel de Proyecto, TestSuite, TestCase y LoadTest:
  - Informes imprimibles: se pueden imprimir o guardar como PDF, HTML, RTF, Excel, etc. y son totalmente personalizables tanto a nivel global como de proyecto, lo que permite crear y personalizar cualquier tipo de informe que se pueda necesitar.
  - Exportación de datos: permite la exportación de datos de informes subyacentes en formato XML y CSV. Esto es útil si se desea importar datos de informes en otras herramientas para informes personalizados o integraciones.
  - Informes HTML: brinda una descripción general simplificada de los resultados de las pruebas funcionales en formato HTML (no disponible en el nivel de LoadTest).
- Se puede incluir fácilmente cualquier dato deseado en los informes generados, ya que puede exportar datos personalizados tanto en informes imprimibles como de exportación de datos a través del SubReport DataSink.

Informes



SOAP Service Mocking

- La funcionalidad de SOAP Service Mocking en SoapUI le permite crear una simulación compatible con los estándares de un servicio basado en WSDL solo a partir de su contrato WSDL, que en SoapUI se denomina "MockService". Se puede ejecutar directamente desde dentro de SoapUI, con la utilidad de línea de comandos o con un contenedor de servlet estándar.
- Los escenarios de uso son muchos:
  - Prototipos rápidos de servicios web: generar una implementación simulada completa estática de un WSDL en segundos y agregar funcionalidad dinámica usando Groovy. Esto permite implementar y probar clientes mucho más rápido que si se hubiera tenido que esperar a que se desarrollara la solución real.
  - Prueba o desarrollo del cliente: crear implementaciones simuladas de las operaciones deseadas y configurar una serie de respuestas alternativas (incluidos scripts, archivos adjuntos y encabezados http personalizados). Los clientes pueden desarrollarse contra el MockService y probarse sin acceso a los servicios en vivo. Las respuestas pueden ciclarse, aleatorizarse o seleccionarse con la expresión XPath de la solicitud entrante
  - Desarrollo dirigido por pruebas: crear pruebas funcionales y de carga en SoapUI contra un MockService antes o durante la implementacíon los servicios reales
- Un MockService cumple con los estándares aceptados WSDL, SOAP o HTTP y un cliente debe poder usarlo como si fuera un servicio real.

SOAP Service Mocking

- Un MockService puede simular cualquier número de contratos WSDL y la funcionalidad de scripting incorporada le permite simular básicamente cualquier tipo de comportamiento deseado; respuestas fijas, errores aleatorios, resultados dinámicos, etc. Incluso es posible implementar un servicio completo en SoapUI y desplegarlo en un contenedor de servlet estándar utilizando la funcionalidad DeployAsWar (aunque no se recomienda para uso en producción).
- MockServices proporciona la simulación del servicio al exponer un número arbitrario de MockOperations que, a su vez, pueden contener un MockRequest Dispatcher y cualquier número de mensajes MockResponse.
- Cuando el MockService recibe una solicitud SOAP y se envía a una MockOperation específica y se selecciona la MockResponse correspondiente en función del MockRequest Dispatcher configurado.
- Cada MockOperation se corresponde a una Operación WSDL de un Servicio WSDL del proyecto de prueba, y los mensajes MockResponse contenidos se crean a partir del esquema asociado con la operación. Esto no significa que un MockResponse deba cumplir con el esquema asociado, puede devolver cualquier mensaje de texto o XML arbitrario y, por ejemplo, se puede configurar para que devuelva un mensaje de respuesta para una operación completamente diferente, lo que permite probar en los clientes la 'capacidad para manejar mensajes de respuesta no válidos e inesperados.

SOAP Service Mocking

- El MockResponse es el mensaje que MockService debe devolver al cliente que realiza la llamada. MockResponse puede contener contenido personalizado, encabezados y archivos adjuntos, lo que le permite simular cualquier tipo de respuesta HTTP válida (o no válida), y las posibilidades de secuencias de comandos adicionales permiten agregar fácilmente contenido dinámico a la respuesta saliente. Los MockResponse se pueden generar desde los servicios reales.
- Una vez que MockService ha recibido una solicitud y se ha enviado a una MockOperation, SoapUI debe seleccionar el método de respuesta correcto y devolverlo al cliente. Hay varios métodos de envío diferentes:
  - SECUENCIA: este es el método de envío más simple; las respuestas se seleccionan y devuelven en el orden en que se agregaron al MockOperation.
  - ALEATORIO: casi tan simple, este despachador selecciona qué respuesta usar al azar, con el tiempo todas las respuestas se habrán devuelto la misma cantidad de veces.
  - QUERY-MATCH: es bastante versátil, puede devolver diferentes respuestas en función del contenido de la solicitud.
  - XPATH: similar a QUERY_MATCH pero no tan potente, solo permite una condición.
  - SCRIPT: la opción más versátil y más difícil de dominar, se crea un script que debería devolver el nombre de MockResponse para usar.

SOAP Service Mocking

- Al hacer clic en el botón Ejecutar , MockService se inicia inmediatamente dentro de SoapUI, se puede consultar jetty log.
- El MockService ahora está listo para manejar las solicitudes SOAP entrantes en la ruta y el puerto configurados. Estas se enviarán a la correspondiente MockOperation en función de su contenido, si no hay disponible ninguna MockOperation coincidente, se devolverá un fallo SOAP estándar.
- El registro de mensajes en la parte inferior muestra todos los mensajes que ha recibido el MockService desde que se inició por última vez y permite consultar, a través del visor de mensajes, la petición recibida así como la respuesta emitida.
- Además, también expone el WSDL de MockServic en la URL estándar de WSDL; por ejemplo, introduciendo http://localhost:8088/mockSampleServiceBinding?WSDL en un navegador se mostrará el WSDL.

REST Service Mocking

- De forma similar a los SOAP Service Mocking, la función de simulación del servicio REST permite simular un servicio REST creando un servicio simulado. Luego se puede ejecutar directamente desde SoapUI o usar la aplicación de línea de comandos mockservicerunner.bat
- Un servicio simulado simula un servicio en vivo al exponer un cierto número de acciones simuladas (método y URL). Las acciones simuladas (Mock Action), a su vez, contienen una serie de respuestas simuladas.
- Las respuestas simuladas permiten establecer el HTTP Status Code, los valores de las cabeceras, el contenido (response body) y su media-type (content-type).
- El MockRequest Dispatcher esta limitado a SECUENCIA y SCRIPT.
- Los REST Service Mocking se ejecutan y supervisan de la misma forma que los SOAP Service Mocking.