



Pruebas JavaScript



© JMA 2020. All rights reserved

INTRODUCCIÓN

© JMA 2020. All rights reserved

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.”

© JMA 2020. All rights reserved

Automatización de pruebas

- Las pruebas exploratorias (manuales) son muy costosas y difícilmente repetibles, por lo que se impone una estrategia de automatización.
- Las pruebas funcionales de usuario final 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 se puede hacer usando enfoques de prueba más livianos como las pruebas unitarias o con un enfoque de bajo nivel.
- JavaScript al ser interpretado directamente por el navegador (no requiere compilación) posibilita que hasta los errores sintácticos lleguen a ejecución. 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.

© JMA 2020. All rights reserved

Automatización de la Prueba

- La automatización de la prueba permite, mediante una Solución de Automatización Pruebas (SAP), ejecutar muchos casos de prueba de forma consistente y repetida en las diferentes versiones del sistema sujeto a prueba (SSP) y/o entornos.
- Los objetivos de la automatización de pruebas son:
 - Mejorar la eficiencia de la prueba.
 - Aportar una cobertura de funciones más amplia.
 - Reducir el coste total de la prueba.
 - Realizar pruebas imposibles de realizar manualmente.
 - Acortar el período de ejecución de la prueba.
 - Aumentar la frecuencia de la prueba y reducir el tiempo necesario para los ciclos de prueba.

© JMA 2020. All rights reserved

Ventajas y Desventajas de la automatización

Ventajas

- Se pueden realizar más pruebas por compilación.
- La posibilidad de crear pruebas que no se pueden realizar manualmente (pruebas en tiempo real, remotas, en paralelo).
- Las pruebas pueden ser más complejas.
- Las pruebas se ejecutan más rápido.
- Las pruebas están menos sujetas a errores del operador.
- Uso más eficaz y eficiente de los recursos de pruebas
- Información de retorno más rápida sobre la calidad del software.
- Mejora de la fiabilidad del sistema (por ejemplo, repetibilidad, consistencia).
- Mejora de la consistencia de las pruebas.

Desventajas

- Requiere tecnologías adicionales.
- Existencia de costes adicionales.
- Inversión inicial para el establecimiento de la SAP.
- Requiere un mantenimiento continuo de la SAP.
- El equipo necesita tener competencia en desarrollo y automatización.
- Puede distraer la atención respecto a los objetivos de la prueba, por ejemplo, centrándose en la automatización de casos de prueba a expensas del objetivo de las pruebas.
- Las pruebas pueden volverse más complejas.
- La automatización puede introducir errores adicionales.

© JMA 2020. All rights reserved

Entornos

- Un enfoque de varios entornos permite compilar, probar y liberar código con mayor velocidad y frecuencia para que la implementación sea lo más sencilla posible. Permite quitar la sobrecarga manual y el riesgo de una versión manual y, en su lugar, automatizar el desarrollo con un proceso de varias fases destinado a diferentes entornos.
 - Desarrollo: es donde se desarrollan los cambios en el software.
 - Prueba: permite que los evaluadores humanos o las pruebas automatizadas prueben el código nuevo y actualizado. Los desarrolladores deben aceptar código y configuraciones nuevos mediante la realización de pruebas unitarias en el entorno de desarrollo antes de permitir que esos elementos entren en uno o varios entornos de prueba.
 - Ensayo/preproducción: donde se realizan pruebas finales inmediatamente antes de la implementación en producción, debe reflejar un entorno de producción real con la mayor precisión posible.
 - UAT: Las pruebas de aceptación de usuario (UAT) permiten a los usuarios finales o a los clientes realizar pruebas para comprobar o aceptar el sistema de software antes de que una aplicación de software pueda pasar a su entorno de producción.
 - Producción: es el entorno con el que interactúan directamente los usuarios.

© JMA 2020. All rights reserved

Niveles de pruebas

- **Pruebas Unitarias o de Componentes:** verifican la funcionalidad y estructura de cada componente individualmente, una vez que ha sido codificado.
- **Pruebas de Integración:** verifican el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente, con el fin de comprobar que interactúan correctamente a través de sus interfaces, cubren la funcionalidad establecida y se ajustan a los requisitos no funcionales especificados en las verificaciones correspondientes.
- **Pruebas de Regresión:** verifican que los cambios sobre un componente de un sistema de información no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.
- **Pruebas del Sistema:** ejercitan profundamente el sistema comprobando la integración del sistema de información globalmente, verificando el funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica.
- **Pruebas de Aceptación:** validan que un sistema cumple con el funcionamiento esperado y permitir al usuario de dicho sistema, que determine su aceptación desde el punto de vista de su funcionalidad y rendimiento.

© JMA 2020. All rights reserved

Pruebas del Sistema

- **Pruebas funcionales:** dirigidas a asegurar que el sistema de información realiza correctamente todas las funciones que se han detallado en las especificaciones dadas por el usuario del sistema.
- **Pruebas de humo:** son un conjunto de pruebas aplicadas a cada nueva versión, su objetivo es validar que las funcionalidades básicas de la versión se cumplen según lo especificado. Impiden la ejecución el plan de pruebas si detectan grandes inestabilidades o si elementos clave faltan o son defectuosos.
- **Pruebas de comunicaciones:** determinan que las interfaces entre los componentes del sistema funcionan adecuadamente, tanto a través de dispositivos remotos, como locales. Asimismo, se han de probar las interfaces hombre-máquina.
- **Pruebas de rendimiento:** consisten en determinar que los tiempos de respuesta están dentro de los intervalos establecidos en las especificaciones del sistema.
- **Pruebas de volumen:** consisten en examinar el funcionamiento del sistema cuando está trabajando con grandes volúmenes de datos, simulando las cargas de trabajo esperadas.
- **Pruebas de sobrecarga o estrés:** consisten en comprobar el funcionamiento del sistema en el umbral límite de los recursos, sometiéndole a cargas masivas. El objetivo es establecer los puntos extremos en los cuales el sistema empieza a operar por debajo de los requisitos establecidos.

© JMA 2020. All rights reserved

Pruebas del Sistema

- **Pruebas de disponibilidad de datos:** consisten en demostrar que el sistema puede recuperarse ante fallos, tanto de equipo físico como lógico, sin comprometer la integridad de los datos.
- **Pruebas de usabilidad:** consisten en comprobar la adaptabilidad del sistema a las necesidades de los usuarios, tanto para asegurar que se acomoda a su modo habitual de trabajo, como para determinar las facilidades que aporta al introducir datos en el sistema y obtener los resultados.
- **Pruebas extremo a extremo (e2e):** consisten en interactuar con la aplicación como un usuario regular lo haría, cliente-servidor, y evaluando las respuestas para el comportamiento esperado.
- **Pruebas de configuración:** consisten en comprobar todos y cada uno de los dispositivos, en sus propiedades mínimo y máximo posibles.
- **Pruebas de operación:** consisten en comprobar la correcta implementación de los procedimientos de operación, incluyendo la planificación y control de trabajos, arranque y re-arranque del sistema, etc.
- **Pruebas de entorno:** consisten en verificar las interacciones del sistema con otros sistemas dentro del mismo entorno.
- **Pruebas de seguridad:** consisten en verificar los mecanismos de control de acceso al sistema para evitar alteraciones indebidas en los datos.

© JMA 2020. All rights reserved

Pruebas de extremo a extremo (e2e)

- Algunas pruebas tienen 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 que la aplicación como esté 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.

© JMA 2020. All rights reserved

Prueba exploratoria

- Incluso los esfuerzos de automatización de pruebas más diligentes no son perfectos. A veces se pierden ciertos casos extremos en sus pruebas automatizadas. A veces es casi imposible detectar un error en particular escribiendo una prueba unitaria. Ciertos problemas de calidad ni siquiera se hacen evidentes en las pruebas automatizadas (como en el diseño o la usabilidad).
- Las limitaciones de la automatización de pruebas son:
 - No todas las pruebas manuales se pueden automatizar y no son un sustituto de las pruebas exploratorias.
 - La automatización sólo puede comprobar resultados interpretables por la máquina.
 - La automatización sólo puede comprobar los resultados reales que pueden ser verificados por un oráculo de prueba automatizado.
- Las pruebas exploratorias es un enfoque de prueba manual que enfatiza la libertad y creatividad del probador para detectar problemas de calidad en un sistema en ejecución.
 - Simplemente tomate un tiempo en un horario regular, arremángate e intenta romper la aplicación.
 - Usa una mentalidad destructiva y encuentra formas de provocar problemas y errores en la aplicación.
 - Ten en cuenta los errores, los problemas de diseño, los tiempos de respuesta lentos, los mensajes de error faltantes o engañosos y, en general, todo lo que pueda molestarte como usuario de una aplicación.
 - Documenta todo lo que encuentre para más adelante.
- La buena noticia es que se puede automatizar la mayoría de los hallazgos con pruebas automatizadas. Escribir pruebas automatizadas para los errores que se detectan asegura que no habrá regresiones a ese error en el futuro. Además, ayuda a reducir la causa raíz de ese problema durante la corrección de errores.

© JMA 2020. All rights reserved

Pirámide de pruebas



© JMA 2020. All rights reserved

<https://martinfowler.com/bliki/TestPyramid.html>

Análisis estático con herramientas

- El objetivo del análisis estático es detectar defectos en el código fuente y en los modelos de software.
- El análisis estático se realiza sin que la herramienta llegue a ejecutar el software objeto de la revisión, como ocurre en las pruebas dinámicas, centrándose más en cómo está escrito el código que en cómo se ejecuta el código.
- El análisis estático permite identificar defectos difíciles de encontrar mediante pruebas dinámicas.
- Al igual que con las revisiones, el análisis estático encuentra defectos en lugar de fallos.
- Las herramientas de análisis estático analizan el código del programa (por ejemplo, el flujo de control y flujo de datos) y las salidas generadas (tales como HTML o XML).
- Algunos de los posibles aspectos que pueden ser comprobados con análisis estático:
 - Reglas, estándares de programación y buenas prácticas.
 - Diseño de un programa (análisis de flujo de control).
 - Uso de datos (análisis del flujo de datos).
 - Complejidad de la estructura de un programa (métricas, por ejemplo valor ciclomático).

© JMA 2020. All rights reserved

Valor del análisis estático

- La detección temprana de defectos antes de la ejecución de las pruebas.
- Advertencia temprana sobre aspectos sospechosos del código o del diseño mediante el cálculo de métricas, tales como una medición de alta complejidad.
- Identificación de defectos que no se encuentran fácilmente mediante pruebas dinámicas.
- Detectar dependencias e inconsistencias en modelos de software, como enlaces.
- Mantenibilidad mejorada del código y del diseño.
- Prevención de defectos, si se aprende la lección en la fase de desarrollo.

© JMA 2020. All rights reserved

Defectos habitualmente detectados

- Referenciar una variable con un valor indefinido.
- Interfaces inconsistentes entre módulos y componentes.
- Variables que no se utilizan o que se declaran de forma incorrecta.
- Código inaccesible (muerto).
- Ausencia de lógica o lógica errónea (posibles bucles infinitos).
- Construcciones demasiado complicadas.
- Infracciones de los estándares de programación.
- Vulnerabilidad de seguridad.
- Infracciones de sintaxis del código y modelos de software.

© JMA 2020. All rights reserved

Ejecución del análisis estático

- Las herramientas de análisis estático generalmente las utilizan los desarrolladores (cotejar con las reglas predefinidas o estándares de programación) antes y durante las pruebas unitarias y de integración, o durante la comprobación del código.
- Las herramientas de análisis estático pueden producir un gran número de mensajes de advertencias que deben ser bien gestionados para permitir el uso más efectivo de la herramienta.
- Los compiladores pueden constituir un soporte para los análisis estáticos, incluyendo el cálculo de métricas.
- El Compilador detecta errores sintácticos en el código fuente de un programa, crea datos de referencia del programa (por ejemplo lista de referencia cruzada, llamada jerárquica, tabla de símbolos), comprueba la consistencia entre los tipos de variables y detecta variables no declaradas y código inaccesible (código muerto).
- El Analizador trata aspectos adicionales tales como: Convenciones y estándares, Métricas de complejidad y Acoplamiento de objetos.

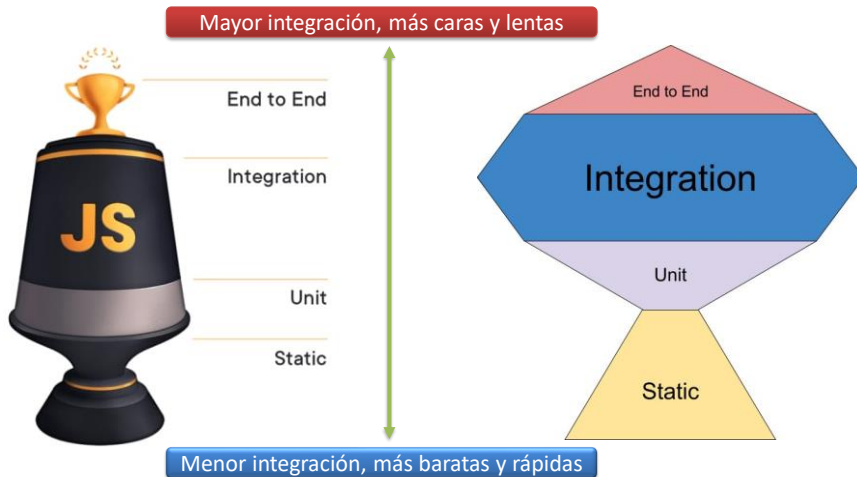
© JMA 2020. All rights reserved

El Trofeo de Pruebas

- Testing Trophy es un método de pruebas propuesto por Kent C. Dodds *para aplicaciones web*. Se trata de escribir suficientes pruebas, no muchas, pero si las pruebas correctas: proporciona la mejor combinación de velocidad, costo y confiabilidad.
- Se superpondrán las siguientes técnicas:
 - Usar un sistema de captura de errores de tipo, estilo y de formato utilizando linters, formateadores de errores y verificadores de tipo (ESLint, SonarQube, ...).
 - Escribir pruebas unitarias efectivas que apunten solo al comportamiento crítico y la funcionalidad de la aplicación.
 - Desarrollar pruebas de integración para auditar la aplicación de manera integral y asegurarse de que todo funcione correctamente en armonía.
 - Crear pruebas funcionales de extremo a extremo (e2e) para pruebas de interacción automatizadas de las rutas críticas y los flujos de trabajo más utilizados por los usuarios.

© JMA 2020. All rights reserved

Testing Trophy



© JMA 2020. All rights reserved

Diseñar la prueba

- Para diseñar la prueba empiezas por identificar y describir los casos de prueba de cada componente.
- La selección de las técnicas de pruebas depende factores adicionales como pueden ser requisitos contractuales o normativos, documentación disponible, tiempo, presupuesto, conocimientos, experiencia, ...
- Cuando dispongas de los casos de prueba, identificas y estructuras los procedimientos de prueba describiendo cómo ejecutar los casos de prueba.

© JMA 2020. All rights reserved

Características de una buena prueba unitaria

- El principio FIRST fue definido por Robert Cecil Martin en su libro Clean Code. Este autor, entre otras muchas cosas, es conocido por ser uno de los escritores del Agile Manifesto, escrito hace más de 15 años y que a día de hoy se sigue teniendo muy en cuenta a la hora de desarrollar software.
 - Fast: Los tests deben ser rápidos, del orden de milisegundos, hay cientos de tests en un proyecto.
 - Isolated/Independent (Aislado/Independiente). Los tests no deben depender del entorno ni de ejecuciones de tests anteriores.
 - Repeatable. Los tests deben ser repetibles y ante la misma entrada de datos, los mismos resultados.
 - Self-Validating. Los tests tienen que ser autovalidados, es decir, NO debe de existir la intervención humana en la validación
 - Thorough and Timely (Completo y oportuno). Los tests deben de cubrir el escenario propuesto, no el 100% del código, y se han de realizar en el momento oportuno

© JMA 2020. All rights reserved

Patrones

- Los casos de prueba se pueden estructurar siguiendo diferentes patrones:
 - ARRANGE-ACT-ASSERT: Preparar, Actuar, Afirmar
 - GIVEN-WHEN-THEN: Dado, Cuando, Entonces
 - BUILD-OPERATE-CHECK: Generar, Operar, Comprobar
- Con diferencias conceptuales, todos dividen el proceso en tres fases:
 - Una fase inicial donde montar el escenario de pruebas que hace que el resultado sea predecible.
 - Una fase intermedia donde se realizan las acciones que son el objetivo de la prueba.
 - Una fase final donde se comparan los resultados con el escenario previsto. Pueden tomar la forma de:
 - Aserción: Es una afirmación sobre el resultado que puede ser cierta o no.
 - Expectativa: Es la expresión del resultado esperado que puede cumplirse o no.

© JMA 2020. All rights reserved

Preparación mínima

- La sección de preparación, con la entrada del caso de prueba, debe ser lo más sencilla posible, lo imprescindible para comprobar el comportamiento que se está probando.
- Las pruebas se hacen más resistentes a los cambios futuros en el código base y más cercano al comportamiento de prueba que a la implementación.
- Las pruebas que incluyen más información de la necesaria tienen una mayor posibilidad de incorporar errores en la prueba y pueden hacer confusa su intención. Al escribir pruebas, el usuario quiere centrarse en el comportamiento. El establecimiento de propiedades adicionales en los modelos o el empleo de valores distintos de cero cuando no es necesario solo resta de lo que se quiere probar.

© JMA 2020. All rights reserved

Actuación mínima

- Al escribir las pruebas hay que evitar introducir condiciones lógicas como if, switch, while, for, etc.
- Minimiza la posibilidad de incorporar un error a las pruebas.
- El foco está en el resultado final, en lugar de en los detalles de implementación.
- Al incorporar lógica al conjunto de pruebas, aumenta considerablemente la posibilidad de agregar un error. Cuando se produce un error en una prueba, se quiere saber realmente que algo va mal con el código probado y no en el código que prueba. En caso contrario, restan confianza y las pruebas en las que no se confía no aportan ningún valor.
- El objetivo de la prueba debe ser único, si la lógica en la prueba parece inevitable, denota que el objetivo es múltiple y hay que considerar la posibilidad de dividirla en dos o más pruebas diferentes.

© JMA 2020. All rights reserved

Comprobación mínima

- Al escribir las pruebas, hay que intentar comprobar una única cosa, es decir, incluir solo una aserción por prueba.
 - Si se produce un error en una aserción, no se evalúan las aserciones posteriores.
 - Garantiza que no se estén declarando varios casos en las pruebas.
 - Proporciona la imagen exacta de por qué se producen errores en las pruebas.
- Al incorporar varias aserciones en un caso de prueba, no se garantiza que se ejecuten todas. Es un todas o ninguna, se sabe por cual fallo pero no si el resto también falla o es correcto, proporcionando la imagen parcial.
- Una excepción común a esta regla es cuando la validación cubre varios aspectos. En este caso, suele ser aceptable que haya varias aserciones para asegurarse de que el resultado está en el estado que se espera que esté.
- Los enfoques comunes para usar solo una aserción incluyen:
 - Crear una prueba independiente para cada aserción.
 - Usar pruebas con parámetros.

© JMA 2020. All rights reserved

Diseñar para probar

- Patrones:
 - Doble herencia
 - Inversión de Control
 - Inyección de Dependencias
 - Modelo Vista Controlador (MVC)
 - Model View ViewModel (MVVM)
- Metodologías:
 - Desarrollo Guiado por Pruebas (TDD)
 - Desarrollo Dirigido por Comportamiento (BDD)

© JMA 2020. All rights reserved

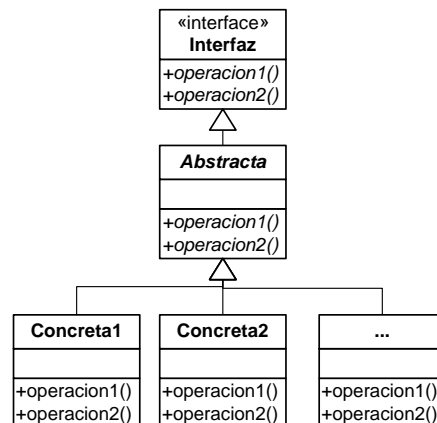
Programar con interfaces

- 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.
- Programar para las interfaz, no para la herencia. Favorecer la composición antes que la herencia.
- Ventajas:
 - Reducción de dependencias.
 - El cliente desconoce la implementación.
 - La vinculación se realiza en tiempo de ejecución.
 - Da consistencia (contrato).
- Desventaja:
 - Indireccionamiento.

© JMA 2020. All rights reserved

Doble Herencia

- Problema:
 - Mantener las clases que implementan como internas del proyecto (internal o Friend), pero la interfaz pública.
 - Organizar clases que tienen un comportamiento parecido para que sea consistente.
- Solución:
 - Clase base es abstracta.
 - La clase base puede heredar de mas de una interfaz.
 - Una vez que están escritos los métodos, verifico si hay duplicación en las clases hijas.



© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Simulación de objetos

- 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)
 - DI: Inyección de Dependencias (Dependency Injection)
 - Objetos Mock

© JMA 2020. All rights reserved

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.
 - Memorizan los valores con los que se llama a cada uno de sus miembros
 - Permiten verificar si los valores esperados coinciden con los recibidos

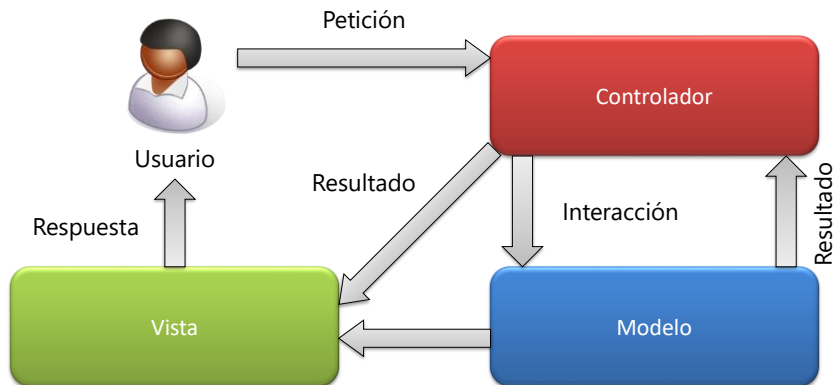
© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

El patrón MVC



© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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.
- 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 reglas básicas:
 - No añadir código sin escribir antes una prueba que falle
 - Eliminar el Código Duplicado empleando Refactorización

© JMA 2020. All rights reserved

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. Escribir una prueba que demuestre la necesidad de escribir código.
 2. Escribir el mínimo código para que el código de pruebas compile
 3. Implementar exclusivamente la funcionalidad demandada por las pruebas
 4. Mejorar el código (Refactoring) sin añadir funcionalidad
 5. 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 que se puede probar automáticamente.

© JMA 2020. All rights reserved

Beneficios de TDD

- Reducen el número de errores y bugs ya que éstos, aplicando TDD, se detectan antes incluso de crearlos.
- Facilitan entender el código y que, eligiendo una buena nomenclatura, sirven de documentación.
- Facilitan mantener el 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.
- Facilitan desarrollar ciñéndose a los requisitos.
- Ayudan a encontrar inconsistencias en los requisitos
- Ayudan a especificar comportamientos
- Ayudan a refactorizar para mejorar la calidad del código (Clean code)
- A medio/largo plazo aumenta (y mucho) la productividad.

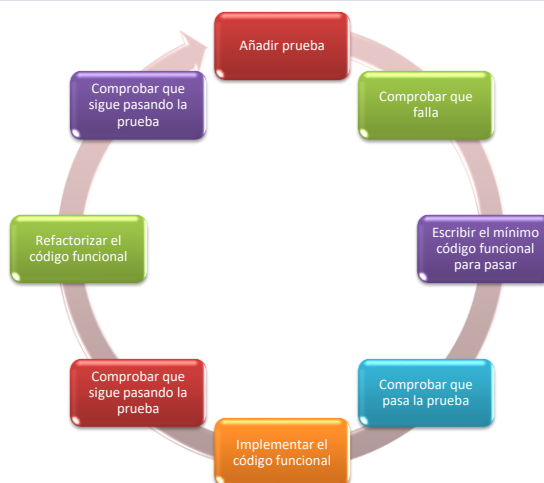
© JMA 2020. All rights reserved

Estrategia RED – GREEN

- 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 **GREEN**)
 - 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 del test 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.

© JMA 2020. All rights reserved

Ritmo TDD



© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 prerequisites 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.

© JMA 2020. All rights reserved

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 software lo hace.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 generar datos de métricas de código que medir la complejidad y el mantenimiento del código administrado.

© JMA 2020. All rights reserved

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, ramas, 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 todos 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.

© JMA 2020. All rights reserved

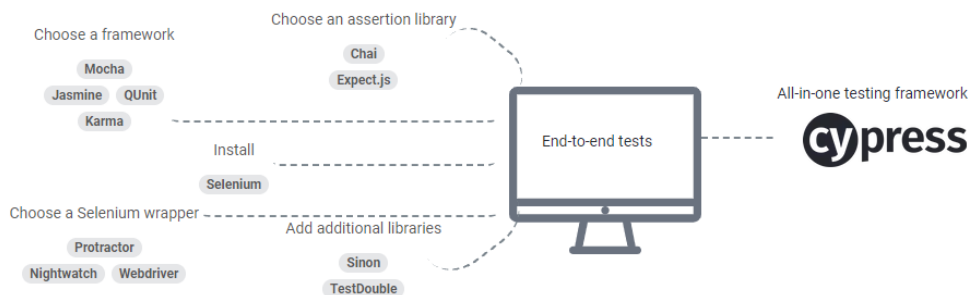
Pruebas de mutaciones

- 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. En este caso el mutante se considera eliminado.
 - 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.

© JMA 2020. All rights reserved

Herramientas

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



© JMA 2020. All rights reserved

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/>

© JMA 2020. All rights reserved

HERRAMIENTAS DE DESARROLLO

© JMA 2020. All rights reserved

IDEs

- Visual Studio Code - <http://code.visualstudio.com/>
 - VS Code is a Free, Lightweight Tool for Editing and Debugging Web Apps.
 - Visual Studio Code for the Web provides a free, zero-install Microsoft Visual Studio Code experience running entirely in your browser: <https://vscode.dev>
- StackBlitz - <https://stackblitz.com>
 - The online IDE for web applications. Powered by VS Code and GitHub.
- CodeSandbox - <https://codesandbox.io/>
 - Create, share, and get feedback with collaborative sandboxes.
- IntelliJ IDEA - <https://www.jetbrains.com/idea/>
 - Capable and Ergonomic Java * IDE
- Webstorm - <https://www.jetbrains.com/webstorm/>
 - Lightweight yet powerful IDE, perfectly equipped for complex client-side development and server-side development with Node.js

© JMA 2020. All rights reserved

Instalación de utilidades

Consideraciones previas

- Las utilidades son de línea de comandos.
- Para ejecutar los comandos es necesario abrir la consola comandos (Símbolo del sistema)
- Siempre que se realice una instalación o creación es conveniente “Ejecutar como Administrador” para evitar otros problemas.
- En algunos casos el firewall de Windows, la configuración del proxy y las aplicaciones antivirus pueden dar problemas.

GIT: Software de control de versiones

- Descargar e instalar: <https://git-scm.com/>
- Verificar desde consola de comandos:
 - git

Node.js: Entorno en tiempo de ejecución

- Descargar e instalar: <https://nodejs.org>
- Verificar desde consola de comandos:
 - node --version

© JMA 2020. All rights reserved

npm: Node Package Manager

- Aunque se instala con el Node es conveniente actualizarlo:
 - `npm update -g npm`
- Verificar desde consola de comandos:
 - `npm --version`
- Configuración:
 - `npm config edit`
 - `proxy=http://usr:pwd@proxy.dominion.com:8080` ← Símbolos: %HEX ASCII
- Generar fichero de dependencias `package.json`:
 - `npm init`
- Instalación de paquetes:
 - `npm install -g grunt-cli karma karma-cli` ← Global (CLI)
 - `npm install jasmine-core tslint --save --save-dev`
 - `npm install` ← Dependencias en `package.json`
- Arranque del servidor:
 - `npm start`

© JMA 2020. All rights reserved

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.
- Se puede instalar ESLint usando npm:
 - `npm install eslint --save-dev`
- Luego se debe crear un archivo de configuración `.eslintrc.json` en el directorio, se puede crear con `--init`:
 - `npx eslint --init`
- Se puede ejecutar ESLint con cualquier archivo o directorio:
 - `npx eslint **/*.js`
- Para habilitarlo y ejecutarlo en las últimas versiones de Angular:
 - `ng add @angular-eslint/schematics`
 - `ng lint`

© JMA 2020. All rights reserved

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 @stencil-community/eslint-plugin`
- Luego se debe crear un archivo de configuración `.eslintrc.json` en el directorio:
 - `npm init @eslint/config`
`{ "parserOptions": { "project": "./tsconfig.json" }, "extends": ["plugin:@stencil-community/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}"`

© JMA 2020. All rights reserved

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 de 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>

© JMA 2020. All rights reserved

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`

© JMA 2020. All rights reserved

GIT

- Preséntate a Git
 - `git config --global user.name "Your Name Here"`
 - `git config --global user.email your_email@youreemail.com`
- Crea un repositorio central
 - <https://github.com/>
- Conecta con el repositorio remoto
 - `git remote add origin https://github.com/username/myproject.git`
 - `git push -u origin master`
- Actualiza el repositorio con los cambios:
 - `git commit -m "first commit"`
 - `git push`
- Para clonar el repositorio:
 - `git clone https://github.com/username/myproject.git local-dir`
- Para obtener las últimas modificaciones:
 - `git pull`

© JMA 2020. All rights reserved

DOCUMENTADOR

© JMA 2020. All rights reserved

Documentación

- Hacer la documentación del código fuente puede llegar a ser muy tedioso, todos los programadores prefieren ir directo al grano, escribir su código y pasar de largo esta aburrida tarea. Por fortuna, actualmente existen un montón de herramientas para agilizar la documentación del código sin tener que dedicarle más tiempo del imprescindible.
- JSDoc es una sintaxis para agregar comentarios con documentación al código fuente de JavaScript.
- La sintaxis JSDoc es similar a la sintaxis de Javadoc, usado para documentar el código de Java, pero se ha especializado para trabajar con la sintaxis de JavaScript, es más dinámico y, por tanto único, ya que no es totalmente compatible con Javadoc. Sin embargo, como Javadoc, JSDoc permite al programador crear Doclets y Taglets que luego se pueden traducir en formatos como HTML o RTF.

```
/**
 * Create a dot.
 * @param {number} x - The x value.
 * @param {number} y - The y value.
 * @param {number} width - The width of the dot, in pixels.
 */
constructor(x, y, width) {
```

© JMA 2020. All rights reserved

JSDoc

Etiqueta	Descripción
@author	nombre del autor.
@constructor	indica el constructor.
@deprecated	indica que ese método es deprecated.
@exception	sinónimo de @throws.
@param	parámetros de documentos y métodos.
@private	indica que el método es privado.
@return	indica que devuelve el método.
@see	Indica la asociación con otro objeto.
@throws	Indica la excepción que puede lanzar un método.
@version	indica el número de versión o librería.

© JMA 2020. All rights reserved

Documentador

- JDOC
 - <https://jsdoc.app/>
- Typedoc
 - <https://typedoc.org>
- Compodoc
 - <https://compodoc.github.io/website/>
- Doxygen
 - <https://www.doxygen.nl/>
- Guía de estilo
 - http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=881:guia-de-estilo-javascript-comentarios-proyectos-jsdoc-param-return-extends-ejemplos-cu01192e&catid=78&Itemid=206

© JMA 2020. All rights reserved

<https://jasmine.github.io/>

JASMINE

© JMA 2020. All rights reserved

Jasmine

- Jasmine es un framework de desarrollo dirigido por comportamiento (behavior-driven development, BDD) para probar código JavaScript.
 - No depende de ninguna otra librería JavaScript.
 - No requiere un DOM.
 - Tiene una sintaxis obvia y limpia para que se pueda escribir pruebas fácilmente.
- En resumen, podríamos decir que desde que los creadores del conocido PivotalTracker sacaron a la luz este framework de test, prácticamente se ha convertido en el estándar de facto para el desarrollo con JavaScript.
- Para su instalación “standalone”, descargar y descomprimir:
 - <https://github.com/jasmine/jasmine/releases>
- Mediante npm: (node)
 - npm install -g jasmine

© JMA 2020. All rights reserved

SpecRunner.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Jasmine Spec Runner v2.5.0</title>
<link rel="shortcut icon" type="image/png" href="lib/jasmine-2.5.0/jasmine_favicon.png">
<link rel="stylesheet" href="lib/jasmine-2.5.0/jasmine.css">
<script src="lib/jasmine-2.5.0/jasmine.js"></script>
<script src="lib/jasmine-2.5.0/jasmine-html.js"></script>
<script src="lib/jasmine-2.5.0/boot.js"></script>
<script type="text/javascript" src="angular.js"></script>
<script type="text/javascript" src="angular-mocks.js"></script>
<!-- include source files here... -->
<script src="src/Player.js"></script>
<script src="src/Song.js"></script>
<!-- include spec files here... -->
<script src="spec/SpecHelper.js"></script>
<script src="spec/PlayerSpec.js"></script>
</head>
<body></body>
</html>
```

© JMA 2020. All rights reserved

Test Runner de Navegador

- Instalar el paquete como dependencia de desarrollo:
 - npm install --save-dev jasmine-browser-runner
- Inicializar para crear directorios y configuración:
 - npx jasmine-browser-runner init
- Ajustar configuración en:
 - spec/support/jasmine-browser.json
- Incorporar el comando al package.json
 - "scripts": { "test": "jasmine-browser-runner runSpecs" }
- Para ejecutar los test interactivamente:
 - npx jasmine-browser-runner serve
- Ejecutar los test:
 - npm test

© JMA 2020. All rights reserved

ESLint for Jasmine

- Instalar el paquete como dependencia de desarrollo:
 - `npm install --save-dev eslint-plugin-jasmine`
- Ajustar configuración en `.eslintrc`:
 - plugins:
 - jasmine
 - env:
 - jasmine: true
 - extends: 'plugin:jasmine/recommended'

© JMA 2020. All rights reserved

Suites

- Una “suite” es un nombre que describe a qué género o sección 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 al 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 de la prueba unitaria.
- El segundo parámetro es una función donde está el código que ejecutará con 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.

© JMA 2020. All rights reserved

Especificaciones

- Una especificación contiene una o más expectativas (algo que se espera) que ponen a prueba el estado del código. Una expectativa de Jasmine 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 Jasmine **it**, que al igual que describe, recibe una cadena y una función, mas un timeout opcional a cuya expiración fallara. La cadena es el título de la especificación y la función es la especificación o prueba.

```
it("y así es una especificación", function() {  
  //...  
});
```
- **describe** y **it** son funciones: pueden contener cualquier código ejecutable necesario para implementar la prueba y se aplican las reglas de alcance de JavaScript: las variables declaradas en un describe están disponibles para cualquier bloque it dentro de la suite.

© JMA 2020. All rights reserved

Expectativas

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

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

```
expect(valor actual).not.matchers(valor esperado);
```
- También existe la posibilidad de escribir comparadores personalizados para cuando el dominio de un proyecto consiste en afirmaciones específicas que no se incluyen en los ya definidos.
- Cuando hay múltiples expectativas se puede documentar la expectativa concreta que falla.

```
expect(valor actual).withContext('mensaje a mostrar').matchers(valor esperado);
```

© JMA 2020. All rights reserved

Comparadores

- `.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.
- `.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.
- `.nothing()`; no espera nada, nunca falla.
- `.toBeCloseTo(n, d)`; verifica la precisión matemática (número de decimales).
- `.toContain(y)`; verifica si el valor actual contiene el esperado.
- `.toBeInstanceOf(tipo)`; verifica si es del tipo esperado.
- `.toHaveClass(className)`; verifica que la etiqueta tenga el class esperado.
- `.toHaveLength(y)`; verifica que la colección tenga el tamaño esperado.

© JMA 2020. All rights reserved

Comparadores

- `.toBeTruthy()`; verifica si el valor es verdadero.
- `.toBeTrue()`; verifica si el valor es estrictamente true.
- `.toBeFalsy()`; verifica si el valor es falso.
- `.toBeFalse()`; verifica si el valor es estrictamente false.
- `.toBeNegativeInfinity()`; verifica si el valor es infinito negativo.
- `.toBePositiveInfinity()`; verifica si el valor es infinito positivo.
- `.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.
- `.toThrow()`; verifica si una función lanza una excepción.
- `.toThrowError(e)`; verifica si una función lanza una excepción específica.

© JMA 2020. All rights reserved

Forzar fallos

- La función `fail(msg)` hace que una especificación falle. Puede llevar un mensaje de fallo o error de un objeto como un parámetro.

```
describe("Una especificación utilizando la función a prueba", function() {
  var foo = function(x, callBack) {
    if (x) {
      callBack();
    }
  };
  it("no debe llamar a la devolución de llamada", function() {
    foo(false, function() {
      fail("Devolución de llamada ha sido llamada");
    });
  });
});
```

© JMA 2020. All rights reserved

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, Jasmine suministra las funciones globales :
 - `beforeEach(fn)` se ejecuta antes de cada especificación dentro del "describe".
 - `beforeAll(fn)` se ejecuta solo una vez antes de empezar a ejecutar las especificaciones 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(){
  var cal;
  beforeEach(function(){
    cal = new Calculadora();
  });
  it("adicion", function(){
    expect(cal.suma(4)).toEqual(4);
  });
  it("multiplicacion", function(){
    expect(cal.multiplica(7)).toEqual(0);
  });
  // ...
});
```

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

© JMA 2020. All rights reserved

Desactivación parcial

- Las Suites se pueden desactivar renombrando la función describe por xdescribe. Estas suites y las especificaciones dentro de ellas se omiten cuando se ejecuta y por lo tanto sus resultados no aparecerán entre los resultados de la prueba.
- De igual forma, las especificaciones se desactivan renombrando it por xit, pero en este caso aparecen en los resultados como pendientes (pending).
- Cualquier especificación declarada sin un cuerpo función también estará marcada pendiente en los resultados.
`it('puede ser declarada con "it", pero sin una función');`
- Y si se llama a la función de pending en cualquier parte del cuerpo de las especificaciones, independientemente de las expectativas, la especificación quedará marcada como pendiente. La cadena que se pasa a pending será tratada como una razón y aparece cuando termine la suite.
`it('se puede llamar a "pending" en el cuerpo de las especificaciones', function() {
 expect(true).toBe(false);
 pending('esto es por lo que está pendiente');
});`

© JMA 2020. All rights reserved

Ejecución de pruebas específicas

- En determinados casos (desarrollo) interesa limitar las pruebas que se ejecutan. Si se pone el foco en determinadas suites o especificaciones solo se ejecutaran las pruebas que tengan el foco, marcando el resto como pendientes.
- Las Suites se enfocan renombrando la función describe por fdescribe. Estas suites y las especificaciones dentro de ellas son las que se ejecutan.
- De igual forma, las especificaciones se enfocan renombrando it por fit.
- Si se enfoca una suite que no tiene enfocada ninguna especificación se ejecutan todas sus especificaciones, pero si tiene alguna enfocada solo se ejecutaran las que tengan el foco.
- Si se enfoca una especificación se ejecutara independientemente de que su suite esté o no enfocada.
- Las funciones de montaje y desmontaje se ejecutaran si la suite tiene alguna especificación con foco.
- Si ninguna suite o especificación tiene el foco se ejecutaran todas las pruebas normalmente.

© JMA 2020. All rights reserved

Pruebas asíncronas

- El código asíncrono es común y Jasmine necesita saber cuándo finaliza el trabajo asíncrono para evaluar el resultado.
- Jasmine admite dos formas de gestionar el trabajo asíncrono: basado en promesas (async/ await) o en devoluciones de llamada.

```
it('does a thing', async function() {  
  const result = await someAsyncFunction();  
  expect(result).toEqual(someExpectedValue);  
});
```

- Cuando no se puedan utilizar promesas, si la función de la especificación, montaje o desmontaje define un argumento (tradicionalmente llamado done), Jasmine pasará una función para ser invocada cuando se haya completado el trabajo asíncrono, debe ser lo último que realice la función asíncrona o cualquiera de las funciones a las que llama para evitar errores o anomalías.

```
it('does a thing', function(done) {  
  someAsyncFunction(function(result) {  
    expect(result).toEqual(someExpectedValue);  
    done();  
  });  
});
```

© JMA 2020. All rights reserved

Comparadores asíncronos

- Los comparadores asíncronos operan con un valor real que es una promesa y devuelven una promesa.
- La mayoría de los comparadores asíncronos esperarán indefinidamente a que la promesa se resuelva o se rechace, lo que dará como resultado la especificación falle por timeout si nunca sucede.
- Son accesibles mediante expectAsync(promesa) o expectAsync(promesa).already (cuando no esta pendiente):

```
await expectAsync(promesa).toBeResolved().then(function() {  
  // more spec code  
});  
await expectAsync(aPromise).toBeRejected();
```
- Los comparadores asíncronos son: toBePending(), toBeResolved(), toBeResolvedTo(expected), toBeRejected(), toBeRejectedWith(expected), toBeRejectedWithError(expected, message).
- Como cualquier comparador, pueden evaluarse como una afirmación negativa mediante el encadenamiento de un not antes de llamar al comparador.

© JMA 2020. All rights reserved

Pruebas dinámicas

- Se pueden generar pruebas dinámicamente utilizando JavaScript para crear múltiples funciones de especificación.

```
describe('Calculos', () => {  
  describe('Sumas', function () {  
    [[1, 2, 3], [2, 2, 4], [3, -2, 1]].forEach(item => {  
      it(`Prueba que ${item[0]} mas ${item[1]} es ${item[2]}`, () =>  
        expect(item[0] + item[1]).toBe(item[2]))  
    });  
  });  
})
```

© JMA 2020. All rights reserved

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, dobles de pruebas, 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)
 - DI: Inyección de Dependencias (Dependency Injection)
 - Simuladores de objetos

© JMA 2020. All rights reserved

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.
- Usar dobles de prueba (como los dobles en el cine) tiene ventajas:
 - Devuelven resultados determinísticos
 - Permiten crear o reproducir determinados estados
 - 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.
 - Memorizan los valores con los que se llama a cada uno de sus miembros
 - Permiten verificar si los valores esperados coinciden con los recibidos

© JMA 2020. All rights reserved

Simulación de objetos

- **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 preprogramado 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.

© JMA 2020. All rights reserved

Espías

- Jasmine tiene funciones dobles de prueba llamados espías.
- Un espía puede interceptar cualquier función y hacer un seguimiento a las llamadas y todos los argumentos.

```
beforeEach(function() {  
  fnc = spyOn(calc, 'suma');  
  prop = spyOnProperty(calc, 'pantalla', 'set')  
});
```
- Un espía sólo existe en el bloque describe o it en que se define, y se eliminará después de cada especificación.
- 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.
 - `.toHaveBeenCalledBefore(esperado)`: pasará si el espía se llama antes que el espía pasado por parámetro.

© JMA 2020. All rights reserved

Seguimiento de llamadas

- El proxy del espía añade la propiedad `calls` que permite:
 - `all()`: Obtener la matriz de llamadas sin procesar para este espía.
 - `allArgs()`: Obtener todos los argumentos para cada invocación de este espía en el orden en que fueron recibidos.
 - `any()`: Comprobar si se ha invocado este espía.
 - `argsFor(índice)`: Obtener los argumentos que se pasaron a una invocación específica de este espía.
 - `count()`: Obtener el número de invocaciones de este espía.
 - `first()`: Obtener la primera invocación de este espía.
 - `mostRecent()`: Obtener la invocación más reciente de este espía.
 - `reset()`: Restablecer el espía como si nunca se hubiera llamado.
 - `saveArgumentsByValue()`: Establecer que se haga un clon superficial de argumentos pasados a cada invocación.

```
spyOn(foo, 'setBar');  
expect(foo.setBar.calls.any()).toEqual(false);  
foo.setBar();  
expect(foo.setBar.calls.count()).toBe(1);
```

© JMA 2020. All rights reserved

Cambiar comportamiento

- Adicionalmente el proxy del espía puede añadir los siguientes comportamiento:
 - `callFake(fn)`: Llamar a una implementación falsa cuando se invoca.
 - `callThrough()`: Llamar a la implementación real cuando se invoca.
 - `exec()`: Ejecutar la estrategia de espionaje actual.
 - `identity()`: Devolver la información de identificación para el espía.
 - `returnValue(valor)`: Devolver un valor cuando se invoca.
 - `returnValues(... values)`: Devolver uno de los valores especificados (secuencialmente) cada vez que se invoca el espía.
 - `stub()`: No haga nada cuando se invoca. Este es el valor predeterminado.
 - `throwError(algo)`: Lanzar un error cuando se invoca.

```
spyOn(foo, "getBar").and.returnValue(745);
spyOn(foo, "getBar").and.callFake(function(arguments, can, be, received) {
  return 745;
});
spyOn(foo, "forbidden").and.throwError("quux");
```

© JMA 2020. All rights reserved

Reloj simulado

- Hay situaciones en las que es útil controlar el objeto `date` y los temporizadores para anular su comportamiento o evitar pruebas lentas:
 - Operaciones que dependen de marcas temporales obtenida a través del objeto `Date`.
 - Operaciones diferidas con `setTimeout` donde no es necesario esperar.
 - Sondeos con `setInterval` que se quieren acelerar.
- El reloj simulado de Jasmine se utiliza al probar el código dependiente del tiempo.
- `jasmine.clock().install()` anula las funciones globales nativas relacionadas con el tiempo para que puedan ser controladas sincrónicamente a través de `jasmine.clock().tick()`, que mueve el reloj hacia adelante, ejecutando los tiempos de espera en cola por el camino. Esto incluye controlar: `setTimeout`, `clearTimeout`, `setInterval`, `clearInterval` y el objeto `Date`.
- Por defecto el reloj comienza en la época de Unix (marca de tiempo de 0). Esto significa que cuando se cree una instancia `new Date` en la aplicación, es inicializa al 01/01/1970 00:00:00. La marca temporal inicial se puede modificar con `jasmine.clock().mockDate()`.

© JMA 2020. All rights reserved

Reloj simulado

- `jasmine.clock().uninstall()` se restauran los métodos integrados originales.
`jasmine.clock().withMock()` permite ejecutar una función con un reloj simulado: llamara a `install` antes de ejecutar la función y a `uninstall` después de que se complete la función.

```
beforeEach(function() {
  jasmine.clock().install();
});
afterEach(function() { jasmine.clock().uninstall(); });
it('does something after 10 seconds', function() {
  const callback = jasmine.createSpy('callback');
  doSomethingLater(callback);
  jasmine.clock().tick(10000);
  expect(callback).toHaveBeenCalled();
});
```

© JMA 2020. All rights reserved

Solicitudes de red

- Las solicitudes a recursos externos tienen el potencial de impactar negativamente en las ejecuciones de prueba debido a tiempos de carga lentos.
- En otros escenarios es difícil de obtener estados específicos del servidor, incluyendo status, headers o body de la respuesta o retrasos de la red, para poder realizar los casos de pruebas apropiados.
- Jasmine suministra un complemento llamado `jasmine-ajax` que permite simular las llamadas ajax en las pruebas. Para usarlo, se debe descargar el archivo `mock-ajax.js` y agregarlo a los ayudantes de Jasmine para que se cargue antes de cualquier especificación que lo use (<https://github.com/jasmine/jasmine-ajax>).
- `jasmine-ajax` proporciona un sustituto del objeto `XMLHttpRequest` intercepta las peticiones y simula la respuesta de la solicitud.
- Las respuestas simuladas pueden establecer directamente estados específicos.
- Las simulaciones son extremadamente rápidas, la mayoría de las respuestas se devolverán en menos de 20 ms. Dado que las respuestas reales pasan por cada capa del servidor (controladores, modelos, vistas, etc.) y, es posible, que se deba inicializar una fuente de datos antes de cada prueba para generar un estado predecible, pueden ser mucho más lentas que las respuestas simuladas.

© JMA 2020. All rights reserved

Solicitudes de red

```
describe('AJAX Mock', () => {
  beforeEach(function () { jasmine.Ajax.install(); });
  afterEach(function () { jasmine.Ajax.uninstall(); });

  it("specifying response when you need it", function () {
    var doneFn = jasmine.createSpy("success");
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function (args) {
      if (this.readyState == this.DONE) {
        doneFn(this.responseText);
      }
    };
    xhr.open("GET", "/some/cool/url");
    xhr.send();
    expect(jasmine.Ajax.requests.mostRecent().url).toBe('/some/cool/url');
    expect(doneFn).not.toHaveBeenCalled();
    jasmine.Ajax.requests.mostRecent().respondWith({
      "status": 200, "contentType": 'text/plain', "responseText": 'awesome response'
    });
    expect(doneFn).toHaveBeenCalled();
  });
});
```

© JMA 2020. All rights reserved

Depuración de pruebas

1. Seleccionar la ventana del navegador Karma.
2. Hacer clic en el botón DEBUG; se abre una nueva pestaña del navegador que permite volver a ejecutar las pruebas.
3. Abrir “Herramientas de Desarrollo” del navegador.
4. Seleccionar la sección de código fuentes.
5. Abrir el archivo con el código de prueba.
6. Establecer un punto de interrupción en la prueba.
7. Actualizar el navegador, que se detiene en el punto de interrupción.

© JMA 2020. All rights reserved

PRUEBAS ANGULAR

© JMA 2020. All rights reserved

Visión general

- A diferencia de otras bibliotecas de JavaScript de front-end populares, Angular es un marco completo y estricto que cubre todos los aspectos importantes del desarrollo de una aplicación web de JavaScript. Angular proporciona una estructura de alto nivel, bloques de construcción de bajo nivel y medios para agrupar todo en una aplicación utilizable.
 - La complejidad de Angular no se puede entender sin considerar las pruebas automatizadas. La arquitectura de Angular garantiza que todas las partes de la aplicación se puedan probar fácilmente de manera similar.
 - Angular proporciona herramientas de prueba sólidas listas para usar. Cuando se crea un proyecto Angular usando la interfaz de línea de comando, viene con una configuración de prueba completamente funcional para pruebas unitarias, de integración y de extremo a extremo. Así mismo, cuando genera un nuevo elemento, crea el correspondiente fichero de pruebas, usando .spec como sub extensión.
-

© JMA 2020. All rights reserved

Visión general

- Hay varias formas de probar una aplicación Angular, la mayoría se agrupan en tres categorías:
 - Pruebas unitarias aisladas del código TypeScript.
 - Crear un módulo de pruebas con el contexto mínimo de importaciones e inyecciones para ejecutar las pruebas un entorno de prueba simplificado y comprobando sus salidas.
 - Ejecutando la aplicación completa en un entorno de prueba más realista utilizando un navegador web (más conocido como pruebas “end-to-end”).
- Mientras las pruebas de tipo “end-to-end” pueden ser muy útiles para prever regresiones a flujos de trabajos importantes, estas pruebas no están relacionadas con las aplicaciones Angular específicamente.

© JMA 2020. All rights reserved

Consideraciones

- Velocidad de iteración vs Entorno realista: Algunas herramientas ofrecen un ciclo de retroalimentación muy rápido entre hacer un cambio y ver el resultado, pero no modelan el comportamiento del navegador con precisión. Otras herramientas pueden usar un entorno de navegador real, pero reducen la velocidad de iteración y son menos confiables en un servidor de integración continua.
- Cuanto abarcar: Cuando se prueban componentes la diferencia entre Prueba Unitaria y Prueba de Integración puede ser borrosa. ¿Si se está probando un formulario, se deben probar los botones del formulario en esta prueba? ¿O el componente botón debe tener su propia suite de pruebas? ¿Debería la refactorización del botón afectar el resultado de las pruebas del formulario?

© JMA 2020. All rights reserved

Pruebas Angular

- Angular, por defecto, se encuentra alineado con la calidad de software.
- Cuando Angular-CLI crea un nuevo proyecto:
 - Descarga TSLint, Jasmine, Karma y Protractor (e2e)
 - Configura el entorno de pruebas
 - Habilita un servidor Karma de pruebas continuas (puerto: 9876)
- Para ejecutar el servidor de pruebas:
 - **ng test --code-coverage** (alias: -cc)
 - **ng test --single-run** (alias: -sr)
- No se debe cerrar la instancia de Chrome mientras duren las pruebas.
- Angular suministra una serie de clases, funciones, mock y módulos específicos para las pruebas, comúnmente denominadas Utilidades Angular para pruebas.
- Permite la creación tanto de pruebas unitarias aisladas como casos de prueba que interactúan dentro del entorno Angular.

© JMA 2020. All rights reserved

Herramientas y tecnologías

Tecnología	Propósito
Jasmine	El marco de trabajo Jasmine proporciona todo lo necesario para escribir pruebas unitarias. Cuenta con una página HTML que ejecuta pruebas en el navegador.
Utilidades Angular de pruebas	Las utilidades Angular de pruebas crean casos de prueba para el código de la aplicación Angular bajo prueba. Permiten añadir y controlar partes de la aplicación a medida que interactúan dentro del entorno Angular.
Karma	El lanzador de pruebas Karma es ideal para escribir y ejecutar pruebas unitarias, mientras se desarrolla la aplicación. Puede ser una parte integral de los procesos de desarrollo e integración continua del proyecto.
Protractor	Permite escribir y ejecutar pruebas de extremo a extremo (E2E), para explorar la aplicación tal y como los usuarios la experimentan.
Selenium WebDriver	El Selenium es un conjunto de herramientas para automatizar los navegadores web, un robot que simula la interacción del usuario con el navegador.

© JMA 2020. All rights reserved

ESLint

- 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.
- Se puede instalar ESLint usando npm:
 - `npm install eslint --save-dev`
- Luego se debe crear un archivo de configuración `.eslintrc.json` en el directorio, se puede crear con `--init`:
 - `npx eslint --init`
- Se puede ejecutar ESLint con cualquier archivo o directorio:
 - `npx eslint **/*.js`
- Para habilitarlo y ejecutarlo en las últimas versiones de Angular:
 - `ng add @angular-eslint/schematics`
 - `ng lint`

© JMA 2020. All rights reserved

e2e: Cypress

- Escribir pruebas e2e requiere muchas herramientas diferentes trabajando juntas. Cypress es un todo en uno, no es necesario instalar 10 herramientas y bibliotecas independientes para configurar el entorno de pruebas. Han tomado algunas de las mejores herramientas de su clase y las han hecho funcionar juntas sin problemas.
- Las pruebas de Cypress solo están escritas en JavaScript, el código de prueba se ejecuta dentro del propio navegador.
- Para habilitarlo y ejecutarlo en las últimas versiones de Angular:
 - `ng add @cypress/schematic`
 - `ng e2e`
 - `ng g spec`
- Para evitar conflictos entre Jasmine (unit tests) y Chai (e2e tests), añadir al principio del `tsconfig.json` de la raíz del proyecto:

```
{ "exclude": ["cypress", "./cypress.config.ts"],
```
- Y en el `tsconfig.json` del directorio `cypress`, sustituir `"include": ["**/*.ts"]`, por:

```
"include": ["**/*.ts", "cypress", "../cypress.config.ts"],  
"exclude": [],
```

© JMA 2020. All rights reserved

Jest

- Hay un [schema](#) disponible para:
 - instalar Jest, tipos y un builder
 - agregar archivos de configuración de Jest
 - modificar la configuración en package.json, angular.json y tsconfig.spec.json
 - eliminar Karma y Jasmine junto con sus archivos de configuración
- Para sustituir Karma y Jasmine junto por Jest:
 - ng add @briebug/jest-schematic
- Las pruebas se siguen ejecutando con:
 - ng test

© JMA 2020. All rights reserved

Pruebas Unitarias Aisladas

- Las Pruebas Unitarias Aisladas examinan una instancia de una clase por sí misma sin ninguna dependencia Angular o de los valores inyectados.
- Se crea una instancia de prueba de la clase con new, se le suministran los parámetros al constructor y se prueba la superficie de la instancia.
- Se pueden escribir pruebas unitarias aisladas para pipes y servicios.
- Aunque también se puede probar los componentes y alguna directivas de forma aislada, las pruebas aisladas no revelan cómo interactúan entre si los elementos Angular. En particular, no pueden revelar cómo una clase de componente interactúa con su propia plantilla o con otros componentes.
- Las directivas que accedan o manipulen directamente el DOM no pueden ser probadas aisladamente.

© JMA 2020. All rights reserved

Pruebas Unitarias Aisladas

- De servicios sin dependencias
 - `let srv = new MyService();`
- De servicios con dependencias
 - `let srv = new MyService(new OtherService());`
- De Pipes
 - `let pipe = new MyPipe();`
 - `expect(pipe.transform('abc')).toBe('Abc');`
- De la clase del componente:
 - `let comp = new MyComponent();`
- De la clase del componente con dependencias:
 - `let comp = new MyComponent(new MyService());`

© JMA 2020. All rights reserved

Utilidades Angular para pruebas

- Para realizar pruebas dentro del contexto de Angular, las Utilidades Angular para pruebas cuentan con las clases TestBed, ComponentFixture, DebugElement y By, así como varias funciones de ayuda para sincronizar, inyectar, temporizar, ...
- Para probar los componentes, lo mas correcto es crear dos juegos de pruebas, a menudo en el mismo archivo de especificaciones.
 - Un primer conjunto de pruebas aisladas que examinan la corrección de la clase del componente.
 - Un segundo conjunto de pruebas que examina como se comporta el componente dentro del Angular, como interactúa con las plantillas, si actualiza el DOM y colabora con el resto de la aplicación.

© JMA 2020. All rights reserved

TestBed

- TestBed representa un módulo Angular para la prueba, proporciona el medio ambiente del módulo para la clase que desea probar. Extrae el componente a probar desde su propio módulo de aplicación y lo vuelve a conectar al módulo de prueba construido a medida, de forma dinámica, específicamente para una serie de pruebas.
- El método `configureTestingModule` reemplaza a la anotación `@NgModule` en la declaración del módulo. Recibe un objeto `@NgModule` que puede tener la mayoría de las propiedades de metadatos de un módulo normal de Angular.
- El estado base incluye la configuración del módulo de prueba predeterminado con las declaraciones (componentes, directivas y pipes) y los proveedores (servicios inyectables) necesarios para el entorno de prueba.
- El método `configureTestingModule` se suele invocar dentro de un método `beforeEach` de modo que TestBed pueda restablecer el estado base antes de cada ejecución de pruebas.

© JMA 2020. All rights reserved

Preparación de la prueba

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { DebugElement } from '@angular/core';
import { By } from '@angular/platform-browser';

import { MyComponent } from './my.component';

describe('Prueba de MyComponent', () => {
  let fixture: ComponentFixture<MyComponent>;
  let comp: MyComponent;
  let de: DebugElement;
  let tag: HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({ declarations: [ MyComponent ], });
    // ...
  })
})
```

© JMA 2020. All rights reserved

Pruebas poco profundas

- En TestBed.configureTestingModule hay que declarar todas las dependencias del componente a probar: otros componentes, pipes y directivas propias, proveedores de los servicios utilizados, incluso los módulos importados.

```
TestBed.configureTestingModule({  
  imports: [MyCoreModule],  
  declarations: [MyComponent, OtherComponent, MyPipe, MyDirective, ...],  
})
```

- Agregando NO_ERRORS_SCHEMA (en @angular/core) a los metadatos del esquema del módulo de prueba se indica al compilador que ignore los elementos y atributos no reconocidos. Ya no es necesario declarar los elementos de plantilla irrelevantes.

```
TestBed.configureTestingModule({  
  declarations: [ MyComponent ],  
  schemas: [ NO_ERRORS_SCHEMA ]  
});
```

© JMA 2020. All rights reserved

ComponentFixture y DebugElement

- Un ComponentFixture es un contexto (fixture) que envuelve el componente creado en el entorno de prueba.
 fixture = TestBed.createComponent(MyComponent);
- El fixture proporciona acceso a si mismo, a la instancia del componente y al envoltorio del elemento DOM del componente.
 comp = fixture.componentInstance; // instancia del componente
- El DebugElement es un envoltorio del elemento DOM del componente o de los elementos localizados.
 de = fixture.debugElement;
 tag = de.nativeElement;

NOTA: Una vez ejecutado el método createComponent se cierra la configuración del TestBed, si se intenta cambiar la configuración dará un error.

© JMA 2020. All rights reserved

Consultar DebugElement

- La clase By es una utilidad Angular de pruebas para consultar el árbol del DOM. By.css utiliza un selector CSS estándar para generar un predicado (devuelve un valor booleano) que filtra de la misma manera que un selector de jQuery y By.directive filtra los nodos que tienen presente la directiva del tipo dado.
- Un predicado de consulta recibe un DebugElement y devuelve true si el elemento cumple con los criterios de selección.
- La clase DebugElement dispone de métodos para, utilizando una función de predicado, buscar en todo el árbol DOM del fixture:
 - El método query devuelve el primer elemento que satisface el predicado.
 - El método queryAll devuelve una matriz de todos los DebugElement que satisfacen el predicado.
- La propiedad nativeElement de DebugElement obtiene el elemento DOM.
let tag: HTMLElement = fixture.debugElement.query(By.css('#myId')).nativeElement;

© JMA 2020. All rights reserved

Consultas By

- Recuperar un componente con elemento HTML:
tag = fixture.debugElement.query(By.css('my-component'));
- Recuperar el valor de la primera etiqueta:
it('should render title in a h1 tag', waitForAsync(() => {
 fixture.detectChanges();
 const tag = fixture.debugElement.query(By.css('h1'));
 expect(tag.nativeElement.textContent).toContain('Welcome to app!!');
}));
- Recuperar los elementos de un listado:
it('renders the list on the screen', () => {
 fixture.detectChanges();
 const li = fixture.debugElement.queryAll(By.css('li'));
 expect(li.length).toBe(2);
});

© JMA 2020. All rights reserved

triggerEventHandler

- El método `DebugElement.triggerEventHandler` permite simular que el elemento ha lanzado un determinado evento.
`fixture.debugElement.triggerEventHandler('click', eventData);`
- Según sea el evento, `eventData` representa el objeto event del DOM o el valor emitido. En algunos casos el `eventData` es obligatorio y con una determinada estructura.
- Dado que los eventos están vinculados a comandos, suele ser preferible invocar directamente a los métodos comando de la clase componente.
- Puede ser necesario para probar la interacción de determinadas directivas con el elemento y para probar las vinculaciones `@HostListener`.

© JMA 2020. All rights reserved

Detección de cambios

- La prueba puede decir a Angular cuándo realizar la detección de cambios, lo que provoca el enlace de datos y la propagación de las propiedades al elemento DOM.
`fixture.detectChanges()`
- Cuando se desea que los cambios se propaguen automáticamente sin necesidad de invocar `fixture.detectChanges()`:

```
import { ComponentFixtureAutoDetect } from '@angular/core/testing';  
// ...  
TestBed.configureTestingModule({  
  declarations: [ MyComponent ],  
  providers: [  
    { provide: ComponentFixtureAutoDetect, useValue: true }  
  ]  
})
```
- El servicio `ComponentFixtureAutoDetect` responde a las actividades asíncronas como la resolución de la promesa, temporizadores y eventos DOM. Pero una actualización directa, síncrona de una propiedad de componente es invisible. La prueba debe llamar `fixture.detectChanges()` manualmente para desencadenar otro ciclo de detección de cambios.

© JMA 2020. All rights reserved

Inyección de dependencias

```
TestBed.configureTestingModule({  
  declarations: [ MyComponent ],  
  providers: [ MyService ]  
});
```

- Un componente bajo prueba no tiene por qué ser inyectado con servicios reales, por lo general es mejor si son dobles de pruebas (stubs, fakes, spies o mocks), dado que el propósito de la especificación es probar el componente y no el servicio o servicios reales que pueden ser el origen del error.

```
providers: [ {provide: MyService, useValue: MyServiceFake },  
             { provide: Router, useClass: RouterStub} ]
```
- Si la prueba necesita tener acceso al servicio inyectado, la forma más segura es obtener el servicio desde el fixture.

```
srv = fixture.debugElement.injector.get(MyService);
```
- También se puede obtener el servicio desde TestBed:

```
srv = TestBed.inject(MyService);
```

© JMA 2020. All rights reserved

inject

- La función `inject` es una de las utilidades Angular de prueba.
- Inyecta servicios en la función especificación donde se los puede alterar, espiar y manipular.
- La función `inject` tiene dos parámetros:
 - Una matriz de tokens de inyección de dependencias Angular.
 - Una función de prueba cuyos parámetros corresponden exactamente a cada elemento de la matriz de tokens de inyección.

```
it('demo inject', inject([Router], (router: Router) => {  
  // ...  
}));
```
- La función `inject` utiliza el inyector del módulo `TestBed` actual y sólo puede devolver los servicios proporcionados a ese nivel. No devuelve los servicios de los proveedores de componentes.

NOTA: Una vez ejecutado el método `createComponent` se cierra la configuración del `TestBed`, si se intenta cambiar la configuración dará un error.

© JMA 2020. All rights reserved

Sobre escritura de @Component

- En algunos casos, sobre todo en la inyección de dependencias a nivel de componentes es necesario sobreescibir la definición del componente.
- La estructura MetadataOverride establece las propiedades de @Component a añadir, modificar o borrar:
type MetadataOverride = { add?: T; remove?: T; set?: T; };
- Donde T son las propiedades de la anotación:
selector?: string;
template?: string; ó templateUrl?: string;
providers?: any[];
...

© JMA 2020. All rights reserved

Sobre escritura de @Component

- El método overrideComponent recibe el componente a modificar y los metadatos con las modificaciones:
TestBed.configureTestingModule({
 declarations: [MyComponent, MyChildComponent],
 providers: [{ provide: Router, useClass: RouterStub}]
})
.overrideComponent(MyChildComponent, {
 set: {
 providers: [
 { provide: MyService, useClass: MyServiceSpy }
]
 }
});

© JMA 2020. All rights reserved

Creación asíncrona

- Si el componente tiene archivos externos de plantillas y CSS, que se especifican en las propiedades `templateUrl` y `styleUrls`, supone un problema para las pruebas.
- El método `TestBed.createComponent` es síncrono.
- Sin embargo, el compilador de plantillas Angular debe leer los archivos externos desde el sistema de archivos antes de que pueda crear una instancia de componente. Eso es una actividad asíncrona.
- El método `compileComponents` devuelve una promesa para que se puedan realizar tareas adicionales inmediatamente después de que termine.
- La función `waitForAsync` es una de las utilidades Angular de prueba que esconde la mecánica de ejecución asíncrona, envuelve una función de especificación en una zona de prueba asíncrona, la prueba se completará automáticamente cuando se finalicen todas las llamadas asíncronas dentro de esta zona.

© JMA 2020. All rights reserved

Preparación de la prueba asíncrona

```
import { TestBed, waitForAsync } from '@angular/core/testing';
// ...
describe('AppComponent', () => {
  beforeEach(waitForAsync() => {
    TestBed.configureTestingModule({
      declarations: [ AppComponent ],
    }).compileComponents();
  });

  it('should create the app', waitForAsync() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  });
});
```

© JMA 2020. All rights reserved

Inyección de servicios asíncronos

- Muchos servicios obtienen los valores de forma asíncrona. La mayoría de los servicios de datos hacen una petición HTTP a un servidor remoto y la respuesta es necesariamente asíncrona.
- Salvo cuando se estén probando los servicios asíncronos, las pruebas no deben hacer llamadas a servidores remotos. Las pruebas deberían emular este tipo de llamadas.
- Disponemos de las siguientes técnicas:
 - Sustituir el servicio asíncrono por un servicio síncrono.
 - Sustituir el método asíncrono por un espía.
 - Crear una zona de pruebas asíncrona.
 - Crear una falsa zona de pruebas asíncrona.

© JMA 2020. All rights reserved

Espías

- Los espías de Jasmine permiten interceptar y sustituir métodos de los objetos.
- Mediante el espía se sustituye el método asíncrono de tal manera que cualquier llamada al mismo recibe una promesa resuelta de inmediato con un valor de prueba (stub).
- El espía no invoca el método real, por lo que no entra en contacto con el servidor.
- En lugar de crear un objeto de servicio sustituto, se inyecta el verdadero servicio y se sustituye el método crítico con un espía Jasmine.

© JMA 2020. All rights reserved

Espías

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ MyComponent ],
    providers:    [ MyService ],
  });
  fixture = TestBed.createComponent(MyComponent);
  comp    = fixture.componentInstance;
  srv     = fixture.debugElement.injector.get(MyService);
  spy = spyOn(srv, 'myAsyncMethod')
    .and.returnValue(Promise.resolve('result value'));
  // ...
});
it('Prueba asíncrona con espía', () => {
  // ...
  fixture.detectChanges();
  expect(...).matcher(...);
  expect(spy.calls.any()).toBe(true, 'myAsyncMethod called');
  expect(spy.calls.count()).toBe(1, 'stubbed method was called once');
  expect(myService.myAsyncMethod).toHaveBeenCalled();
});
```

© JMA 2020. All rights reserved

whenStable

- En algunos casos, la prueba debe esperar a que la promesa se resuelva en el siguiente ciclo de ejecución del motor de JavaScript.
- En este escenario la prueba no se tiene acceso directo a la promesa devuelta por la llamada del método asíncrono dado que está encapsulada en el interior del componente, inaccesibles desde la superficie expuesta.
- Afortunadamente, la función `waitForAsync` genera una zona de prueba asíncrona, que intercepta todas las promesas emitidas dentro de la llamada al método asíncrono sin importar dónde se producen.
- El método `ComponentFixture.whenStable` devuelve su propia promesa que se resuelve cuando todas las actividades asíncronas pendientes dentro de la prueba se han completado (cuando sea estable).

```
it('Prueba asíncrona cuando sea estable', waitForAsync() => {
  fixture.detectChanges();
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    expect(...).matcher(...);
  });
});
```

© JMA 2020. All rights reserved

fakeAsync

- La función `fakeAsync`, otra de las utilidades Angular de prueba, es una alternativa a la función `waitForAsync`. La función `fakeAsync` permite un estilo de codificación secuencial mediante la ejecución del cuerpo de prueba en una zona especial de ensayo propia de `fakeAsync`, haciendo que la prueba aparezca como si fuera síncrona.
- Se apoya en la función `tick()`, que simula el paso del tiempo hasta que todas las actividades asíncronas pendientes concluyen, similar al `wait` en concurrencia. Sólo se puede invocar dentro del cuerpo de `fakeAsync`, no devuelve nada, no hay promesa que esperar.

```
it('Prueba asíncrona cuando con fakeAsync', fakeAsync(() => {  
  fixture.detectChanges();  
  tick();  
  fixture.detectChanges();  
  expect(...).matcher(...);  
}));
```

© JMA 2020. All rights reserved

Componentes contenidos

- Para probar los componentes simulando que están contenidos en una plantilla es necesario crear un componente Angular para la prueba (wrapper):

```
@Component({  
  template: `<my-component [myInput]="MyInput"  
    (myOutput)="onMyOutput($event)"></my-component>`  
})  
class TestHostComponent {  
  @ViewChild(MyComponent) myComponent: MyComponent;  
  MyInput: any = null;  
  MyOutput: any;  
  onMyOutput(rslt: any) { this.MyOutput = rslt; }  
}
```

© JMA 2020. All rights reserved

Componentes contenidos

- Para posteriormente instanciarlo:

```
beforeEach( waitForAsync() => {  
  TestBed.configureTestingModule({  
    declarations: [ MyComponent, TestHostComponent ],  
  }).compileComponents();  
});  
  
beforeEach(() => {  
  // create TestHostComponent instead of MyComponent  
  fixture = TestBed.createComponent(TestHostComponent);  
  testHost = fixture.componentInstance;  
  tag = fixture.debugElement.query(By.css('my-component'));  
  fixture.detectChanges(); // trigger initial data binding  
});
```

© JMA 2020. All rights reserved

Entradas y Salidas

- Entrada: Se modifican las entradas a través del componente de pruebas y se comprueba que las modificaciones se reflejan en el componente contenido.

```
it('input test', () => {  
  testHost.MyInput = '666';  
  fixture.detectChanges();  
  expect(testHost.myComponent.getInit()).toBe('666');  
});
```

- Salida: Se interactúa con el componente contenido para que se disparen los eventos de salida y se comprueba en el componente de pruebas que las modificaciones se han reflejado en el.

```
it('output test', () => {  
  testHost.myComponent.exec();  
  fixture.detectChanges();  
  expect(testHost.MyOutput).toBe('666');  
});
```

© JMA 2020. All rights reserved

Pruebas de observables

- Para poder probar los observables es necesario:
 - Crear una zona asíncrona
 - Convertir el observable en una promesa
 - Definir las expectativas dentro del .then de la promesa.

```
import 'rxjs/add/operator/toPromise';

it('get http', waitForAsync(inject([HttpClient], (http: HttpClient) => {
  http.get(url)
    .toPromise().then(
      data => { expect(data).toBeTruthy(); },
      err => { fail(); }
    );
})));
```

© JMA 2020. All rights reserved

Pruebas de observables

- Para poder probar los observables mediante devolución de llamadas:

```
it('callback', (done: DoneFn) => inject([ValueService], (service) => {
  service.subscribe(
    data => { expect(data).toBe(0); done(); },
    () => fail()
  );
}()));
```

- Se puede utilizar el marco de pruebas suministrado por RxJS.

```
import { TestScheduler } from 'rxjs/testing';
:
it('decrement contador', () => {
  let obs$ = service.getObservable();
  service.operate();
  testScheduler.run(helpers => {
    const { expectObservable } = helpers;
    expectObservable(obs$).toBe('a', {a:-1});
  });
});
```

© JMA 2020. All rights reserved

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.
- Eso quiere decir que si una unidad tiene dependencias hay que reemplazarlas por mocks.
- Un ejemplo de mock de un servicio sería:

```
class MyServiceSpy {  
  getData = jasmine.createSpy('getData').and.callFake(() => {  
    return of([{ id: 0, name: 'Uno'}, { id: 1, name: 'Dos' }]);  
  });  
}
```

- Para crear las especificación:

```
it('fetches all data', () => {  
  // ...  
  expect(instance.names.length).toBe(2);  
  expect(MyService.getData).toHaveBeenCalled();  
});
```

© JMA 2020. All rights reserved

Doble de prueba Observable

```
import { of, Observable } from 'rxjs';
```

```
export class DAOServiceMock {  
  constructor(private listado: Array<any>) { }  
  query(): Observable<any> { return of(this.listado); }  
  get(id: number) { return of(this.listado[0]); }  
  add(item: any) { return of(item); }  
  change(id: number, item: any) { return of(item); }  
  remove(id: number) { return of(id); }  
}
```

```
{provide: MyDAOService, useValue: new DAOServiceMock([  
  { id: 0, name: 'Uno'}, { id: 1, name: 'Dos' }])}
```

© JMA 2020. All rights reserved

Prueba de peticiones HTTP

- La biblioteca de pruebas HTTP de Angular está diseñada siguiendo un patrón de pruebas donde la especificación empieza haciendo las solicitudes.
- Después de eso, las pruebas esperan a que ciertas solicitudes hayan sido o no realizadas, se cumplan determinadas afirmaciones contra esas solicitudes y, finalmente, se proporcionan respuestas "descargando" cada solicitud esperada, lo que puede activar más solicitudes nuevas, etc.
- Al terminar, se pueden verificar que la aplicación no ha hecho peticiones inesperadas.
- Para disponer de la biblioteca de pruebas HTTP:
imports: [HttpClientTestingModule,],

© JMA 2020. All rights reserved

Prueba de peticiones HTTP

- El módulo instala un mock que sustituye el acceso real al servidor.
- `HttpTestingController` es el controlador que se inyecta en las pruebas, permite la inspección y el volcado de las solicitudes.

```
it('query', inject([DAOService, HttpTestingController], (dao: DAOService, httpMock:  
HttpTestingController) => {  
  dao.query().subscribe(  
    data => { expect(data.length).toEqual(2); },  
    data => { fail(); }  
  );  
  const req = httpMock.expectOne('http://localhost:4321/data');  
  expect(req.request.method).toEqual('GET');  
  req.flush([{ name: 'Data' }, { name: 'Data2' }]);  
  httpMock.verify();  
});
```

© JMA 2020. All rights reserved

Enrutado

- Para disponer del enrutador se utilizará el módulo:

```
imports: [ // ...
  RouterTestingModule.withRoutes([
    {path: '', component: HomeComponent},
    {path: 'simple', component: SimpleCmp}
  ])
]
```

- Para crear y registrar un sustituto del Router:

```
class RouterStub {
  navigateByUrl(url: string) { return url; }
  navigate(commands: Array<any>) { return url; }
}
{ provide: Router, useClass: RouterStub },
```

- Para interceptar las llamadas al Router:

```
it('Demo ROUTER', inject([Router], (router: Router) => { // ...
  const spy = spyOn(router, 'navigateByUrl');
  // ...
  const navArgs = spy.calls.first().args[0];
  expect(navArgs).toBe(...);
}));
```

© JMA 2020. All rights reserved

ng-mocks

- [ng-mocks](#) es una biblioteca rica en funciones para probar componentes con dependencias falsas o dobles de prueba.
- La función `MockComponent` espera el componente original y devuelve un doble de prueba que se parece al original ya que tiene todas las propiedades y métodos que tiene el original. TypeScript utiliza un sistema de tipos estructural que comprueba si se cumplen todos los requisitos de tipo y, desde el punto de vista de TypeScript, el doble de prueba se ajusta al tipo del componente original. También se pueden crear otros dobles de prueba con las funciones `MockService`, `MockDirective`, `MockPipe`, `MockProvider` o `MockModule`.

```
beforeEach(async () => {
  await TestBed.configureTestingModule({
    declarations: [HomeComponent, MockComponent(CalculadoraComponent)],
    schemas: [NO_ERRORS_SCHEMA],
  }).compileComponents();
});
```

© JMA 2020. All rights reserved

Spectator

- [Spectator](#) es una poderosa herramienta para simplificar la pruebas de Angular y ayuda a deshacerse de todo el trabajo pesado repetitivo, dejándonos con pruebas unitarias legibles, elegantes y optimizadas.
 - `npm install @ngneat/spectator --save-dev`
- Opcionalmente, agregar esquema de generación a `angular.json`:

```
"cli": {  
  "schematicCollections": [  
    "@angular-eslint/schematics",  
    "@ngneat/spectator"  
  ]  
}
```
- Con el esquema se pueden generar con Angular CLI componentes, servicios y directivas con especificaciones en Spectator:
 - `ng g cs MiComponente [--with-host=true] [--with-custom-host=true]`
 - `ng g ss MiServicio [--is-data-service=true]`
 - `ng g ds MiDirectiva`

© JMA 2020. All rights reserved

Spectator

- La idea principal de Spectator es unificar las API TestBed, ComponentFixture y DebugElement en una interfaz coherente, potente y fácil de usar: el objeto Spectator. Spectator recurre a la biblioteca ng-mocks para falsificar componentes secundarios.
- Para crear el objeto Spectator se utiliza un factoría. Para crear las factorías, el API Spectator suministra las funciones `create...Factory()` a las que se les pasa la clase del componente, directiva, pipe o servicio que se desea probar o un objeto configuración.
- El API Spectator incluye métodos para consultar el DOM, mediante selectores, como parte de una prueba : `query`, `queryAll`, `queryLast`, `queryHost` y `queryHostAll`.
- El objeto Spectator suministra métodos para invocar eventos, por defecto sobre el componente bajo prueba pero aceptan selectores para indicar sobre cual, y asistentes de teclado (`spectator.keyboard`) y ratón (`spectator.mouse`)

© JMA 2020. All rights reserved

Spectator: Consultas

- Los métodos de consulta del API Spectator son polimórficos y permiten consultar utilizando:
 - Un selector CSS para consultar elementos que coincidan en el DOM (equivalente al predicado `By.css` de Angular) y se devolverán elementos HTML nativos.
 - `spectator.query('div > ul.nav li:first-child');`
 - Un tipo (como un componente, directiva o clase del proveedor) para consultar instancias de ese tipo en el DOM (`By.directive` del Angular). Se puede pasar un segundo parámetro para leer un token de inyección específico de los inyectores de los elementos coincidentes.
 - `host.queryLast(ChildComponent);`
 - Una función selectora (inspiradas en `dom-testing-library`), están disponibles: `byTestId()`, `byRole()`, `byPlaceholder()`, `byValue()`, `byTitle()`, `byAltText()`, `byLabel()`, `byText()` y `byTextContent()`.
 - `spectator.query(byLabel('By label'));`

© JMA 2020. All rights reserved

Spectator: Componentes

```
describe('ButtonComponent', () => {
  let spectator: Spectator<ButtonComponent>;
  const createComponent = createComponentFactory(ButtonComponent);

  beforeEach(() => spectator = createComponent());

  it('should have a success class by default', () => {
    expect(spectator.query('button')).toHaveClass('success');
  });

  it('should set the class name according to the [className] input', () => {
    spectator.setInput('className', 'danger');
    expect(spectator.query('button')).toHaveClass('danger');
    expect(spectator.query('button')).not.toHaveClass('success');
  });
});
```

© JMA 2020. All rights reserved

Spectator: Componentes en host

```
describe('ZippyComponent', () => {
  let spectator: SpectatorHost<ZippyComponent>;
  const createHost = createHostFactory(ZippyComponent);

  it('should display the title from host property', () => {
    spectator = createHost(`<zippy [title]="title"></zippy>`, { hostProps: { title: 'Spectator is Awesome' } });
    expect(spectator.query('.zippy__title')).toHaveText('Spectator is Awesome');
  });

  it('should display the "Close" word if open', () => {
    spectator = createHost(`<zippy title="Zippy title">Zippy content</zippy>`);
    spectator.click('.zippy__title');
    expect(spectator.query('.arrow')).toHaveText('Close');
    expect(spectator.query('.arrow')).not.toHaveText('Open');
  });
});
```

© JMA 2020. All rights reserved

Spectator: Componentes con rutas

```
describe('ProductDetailsComponent', () => {
  let spectator: SpectatorRouting<ProductDetailsComponent>;
  const createComponent = createRoutingFactory({
    component: ProductDetailsComponent,
    params: { productId: '3' },
    data: { title: 'Some title' }
  });

  beforeEach(() => spectator = createComponent());

  it('should display route data title', () => {
    expect(spectator.query('.title')).toHaveText('Some title');
  });

  it('should react to route changes', () => {
    spectator.setRouteParam('productId', '5');

    // your test here...
  });
});
```

© JMA 2020. All rights reserved

Spectator: Directivas

```
describe('HighlightDirective', () => {
  let spectator: SpectatorDirective<HighlightDirective>;
  const createDirective = createDirectiveFactory(HighlightDirective);

  beforeEach(() => {
    spectator = createDirective(`<div highlight>Testing Highlight Directive</div>`);
  });

  it('should change the background color', () => {
    spectator.dispatchMouseEvent(spectator.element, 'mouseover');
    expect(spectator.element).toHaveStyle({ backgroundColor: 'rgba(0,0,0, 0.1)' });
    spectator.dispatchMouseEvent(spectator.element, 'mouseout');
    expect(spectator.element).toHaveStyle({ backgroundColor: '#fff' });
  });

  it('should get the instance', () => {
    const instance = spectator.directive;
    expect(instance).toBeDefined();
  });
});
```

© JMA 2020. All rights reserved

Spectator: Pipes

```
describe('SumPipe', () => {
  let spectator: SpectatorPipe<SumPipe>;
  const createPipe = createPipeFactory(SumPipe);

  it('should sum up the given list of numbers (template)', () => {
    spectator = createPipe('{{ [1, 2, 3] | sum }}');
    expect(spectator.element).toHaveText('6');
  });

  it('should sum up the given list of numbers (prop)', () => {
    spectator = createPipe('{{ prop | sum }}', { hostProps: { prop: [1, 2, 3] } });
    expect(spectator.element).toHaveText('6');
  });

  it('should delegate the summation to the service', () => {
    const sum = () => 42;
    const provider = { provide: StatsService, useValue: { sum } };
    spectator = createPipe('{{ prop | sum }}', { hostProps: { prop: [2, 40] }, providers: [provider] });
    expect(spectator.element).toHaveText('42');
  });
});
```

© JMA 2020. All rights reserved

Spectator: Servicios

```
describe('HttpClient testing', () => {
  let spectator: SpectatorHttp<TodosDataService>;
  const createHttp = createHttpFactory(TodosDataService);

  beforeEach(() => spectator = createHttp());
  it('can test HttpClient.get', () => {
    spectator.service.getTodos().subscribe();
    spectator.expectOne('api/todos', HttpMethod.GET);
  });
  it('can test HttpClient.post', () => {
    spectator.service.postTodo(1).subscribe();
    const req = spectator.expectOne('api/todos', HttpMethod.POST);
    expect(req.request.body['id']).toEqual(1);
  });
  it('can test current http requests', () => {
    spectator.service.getTodos().subscribe();
    const reqs = spectator.expectConcurrent([
      { url: '/api1/todos', method: HttpMethod.GET },
      { URL: '/api2/todos', method: HttpMethod.GET }
    ]);
    spectator.flushAll(reqs, [{}, {}, {}]);
  });
});
```

© JMA 2020. All rights reserved

<http://www.protractortest.org/>

TEST E2E

© JMA 2020. All rights reserved

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.
- Las pruebas de extremo a extremo (E2E: end to end) se hacen para encontrar estos problemas.
- El equipo de Angular ha desarrollado Protractor que simula las interacciones del usuario con el interfaz (navegador) y que ayudará a verificar el estado de una aplicación Angular.
- Protractor es una aplicación Node.js para ejecutar pruebas de extremo a extremo que también están escritas en JavaScript y se ejecutan con el propio Node.
- Protractor utiliza WebDriver para controlar los navegadores y simular las acciones del usuario.

© JMA 2020. All rights reserved

Introducción

- Protractor utiliza Jasmine para su sintaxis prueba.
- Al igual que en las pruebas unitarias, el archivo de pruebas se compone de uno o más bloques describe de it que describen los requisitos de su aplicación.
- Los bloques it están hechos de comandos y expectativas.
- Los comandos indican a Protractor que haga algo con la aplicación, como navegar a una página o hacer clic en un botón.
- Las expectativas indican a Protractor afirmaciones sobre algo acerca del estado de la aplicación, tales como el valor de un campo o la URL actual.
- Si alguna expectativa dentro de un bloque it falla, el ejecutor marca en it como "fallido" y continúa con el siguiente bloque.
- Los archivos de prueba también pueden tener bloques beforeEach y afterEach, que se ejecutarán antes o después de cada bloque it, independientemente de si el bloque pasa o falla.

© JMA 2020. All rights reserved

Instalación

- Se utiliza npm para instalar globalmente Protractor:
 - `npm install -g protractor`
- Esto instalará dos herramientas de línea de comandos, protractor y WebDriver-manager, para asegurarse de que está funcionando.
 - `protractor -versión`
- El WebDriver-Manager es una herramienta de ayuda para obtener fácilmente una instancia de un servidor en ejecución Selenium. Para descargar los binarios necesarios:
 - `webdriver-manager update`
- Para poner en marcha el servidor:
 - `webdriver-manager start`
- Las pruebas Protractor enviarán solicitudes a este servidor para controlar un navegador local, el servidor debe estar en funcionamiento durante todo el proceso de pruebas.
- Se puede ver información sobre el estado del servidor en:
 - `http://localhost:4444/wd/hub`

© JMA 2020. All rights reserved

Configuración y Ejecución

- Se configura un fichero con las pruebas a realizar:

```
// Fichero: e2e.conf.js
exports.config = {
  framework: 'jasmine',
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['test/*.e2e.js'],
  multiCapabilities: [
    //{ browserName: 'firefox' },
    { browserName: 'chrome' }
  ]
};
```
- Se lanzan las pruebas (con Selenium Server en ejecución):
 - `protractor e2e.conf.js`

© JMA 2020. All rights reserved

Elementos Globales

- **browser:** Envoltura alrededor de una instancia de WebDriver, utilizado para la navegación y la información de toda la página.
 - El método `browser.get` carga una página.
 - Protractor espera que Angular esté presente la página, por lo que generará un error si la página que está intentando cargar no contiene la biblioteca Angular.
- **element:** Función de ayuda para encontrar e interactuar con los elementos DOM de la página que se está probando.
 - La función `element` busca un elemento en la página.
 - Se requiere un parámetro: una estrategia de localización del elemento dentro de la página.
- **by:** Colección de estrategias elemento localizador.
 - Por ejemplo, los elementos pueden ser encontrados por el selector CSS, por el ID, por el atributo `ng-model`, ...
- **protractor:** Espacio de nombres de Angular que envuelve el espacio de nombres WebDriver.
 - Contiene variables y clases estáticas, tales como `protractor.Key` que se enumera los códigos de teclas especiales del teclado.

© JMA 2020. All rights reserved

Visión de conjunto

- Protractor exporta la función global `element`, que con un localizador devolverá un `ElementFinder`.
- Esta función recupera un solo elemento, si se necesita recuperar varios elementos, la función `element.all` obtiene la colección de elementos localizados.
- El `ElementFinder` 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.
- Cuando se buscan elementos en Protractor todas las acciones son asíncronas:
 - Por debajo, todas las acciones se envían al navegador mediante el protocolo SON Webdriver Wire Protocol.
 - El navegador realiza la acción tal y como un usuario lo haría de forma nativa o manual.

© JMA 2020. All rights reserved

Localizadores

- Un localizador de Protractor dice cómo encontrar un cierto elemento DOM.
- Los localizadores más comunes son:
 - `by.css('.myclass')`
 - `by.id('myid')`
 - `by.model('name')`
 - `by.binding('bindingname')`
- Los localizadores se pasan a la función `element`:
 - `var tag = element(by.css('some-css'));`
 - `var arr = element.all(by.css('some-css'));`
- Aunque existe una notación abreviada para CSS similar a jQuery:
 - `var tag = $('some-css');`
- Para encontrar subelementos o listas de subelementos:
 - `var uno = element(by.css('some-css')).element(by.tagName('tag-within-css'));`
 - `var varios = element(by.css('some-css')).all(by.tagName('tag-within-css'));`

© JMA 2020. All rights reserved

ElementFinder

- La función `element()` devuelve un objeto `ElementFinder`.
- El `ElementFinder` sabe cómo localizar el elemento DOM utilizando el localizador que se pasa como un parámetro, pero en realidad no lo ha hecho todavía.
- No va a ponerse en contacto con el navegador hasta que un método de acción sea llamado.
- `ElementFinder` permite invocar acciones como si se produjesen directamente en el navegador.
- Dado que todas las acciones son asíncronas, todos los métodos de acción devuelven una promesa.
- Las acciones sucesivas se encolan y se mandan al navegador ordenadamente.
- Para acciones que deban esperar se usan las promesas:

```
element(by.model('nombre')).getText().then(function(text) {  
  expect(text).toBe("MUNDO");  
});
```

© JMA 2020. All rights reserved

La prueba

```
describe('Primera prueba con Protractor', function() {
  it('introducir nombre y saludar', function() {
    browser.get('http://localhost:4200/');
    var txt = element(by.model('vm.nombre'));
    txt.clear();
    txt.sendKeys('Mundo');
    browser.sleep(5000);
    element(by.id('btnSaluda')).click();
    expect(element(by.binding('vm.msg')).getText()).
      toEqual('Hola Mundo');
    browser.sleep(5000);
  });
});
```

© JMA 2020. All rights reserved

Selenium

- 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.

© JMA 2020. All rights reserved

Selenium IDE

- Es el entorno de desarrollo integrado para pruebas con Selenium que permite grabar, editar y depurar fácilmente las pruebas.
- Solo está disponible como una extensión de Firefox y Chrome.
- 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.

© JMA 2020. All rights reserved

Selenium IDE

- 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.

© JMA 2020. All rights reserved

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>
```

© JMA 2020. All rights reserved

Ejecutar en línea de comandos

- Requiere tener instalado NodeJS (<https://nodejs.org>)
- Instalar CLI
 - npm install -g selenium-side-runner
- Instalar los WebDriver:
 - Crear una carpeta y referenciarla en el PATH del sistema.
 - Descargar los drivers (<https://www.seleniumhq.org/download/>)
 - 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"

© JMA 2020. All rights reserved

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.
- Cree 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.

© JMA 2020. All rights reserved

Selenium

- <http://www.seleniumhq.org/>
- 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.

© JMA 2020. All rights reserved

Selenium IDE

- Es el entorno de desarrollo integrado para pruebas con Selenium que permite grabar, editar y depurar fácilmente las pruebas.
- Solo está disponible como una extensión de Firefox.
- 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.

© JMA 2020. All rights reserved

Selenium IDE

- 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.

© JMA 2020. All rights reserved

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());
    }
}
```

© JMA 2020. All rights reserved



Cypress



<https://www.cypress.io/>

© JMA 2020. All rights reserved

INTRODUCCIÓN

© JMA 2020. All rights reserved

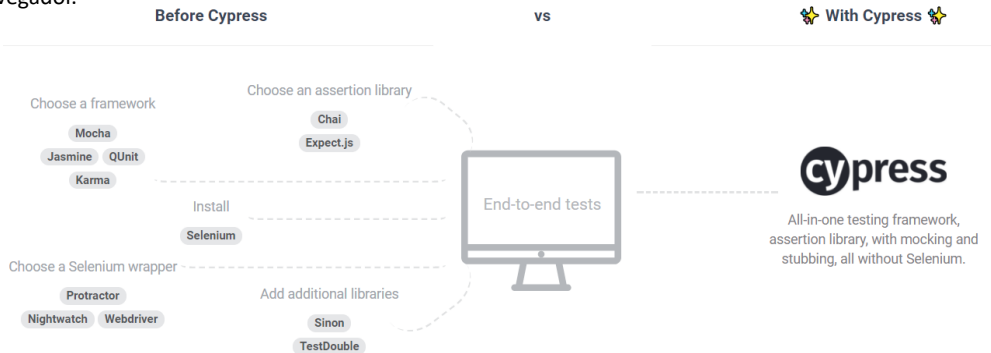
Introducción

- Cypress es una herramienta de prueba de front-end de próxima generación creada para la web moderna, que permite configurar, escribir, ejecutar y depurar pruebas.
- Aunque Cypress se compara con Selenium, es fundamental y arquitectónicamente diferente, controla los navegadores a través de JavaScript y las DevTools, por lo que no está limitado por las mismas restricciones que el Selenium. Esto permite escribir pruebas más rápidas, fáciles y confiables. Esta destinado a desarrolladores o ingenieros de control de calidad que crean aplicaciones web utilizando marcos de JavaScript modernos.
- Aunque Cypress puede probar cualquier cosa que se ejecute en un navegador, esta destinado a:
 - Pruebas de un extremo a extremo
 - Pruebas de integración
 - Pruebas unitarias (componentes)
- Cypress se ha construido para que las pruebas y el desarrollo se puedan realizar simultáneamente. Es suficientemente rápido para habilitar metodologías TDD, BDD o ATDD, donde los cambios en el código o en las pruebas se reflejan en tiempo real.

© JMA 2020. All rights reserved

Introducción

- Escribir pruebas e2e requiere muchas herramientas diferentes trabajando juntas. Cypress es un todo en uno, no es necesario instalar 10 herramientas y bibliotecas independientes para configurar el entorno de pruebas. Han tomado algunas de las mejores herramientas de su clase y las han hecho funcionar juntas sin problemas.
- Las pruebas de Cypress solo están escritas en JavaScript, el código de prueba se ejecuta dentro del propio navegador.



© JMA 2020. All rights reserved

Características

- Time Travel: Toma instantáneas mientras se ejecutan las pruebas, permitiendo desplazarse sobre los comandos para ver exactamente qué sucedió en cada paso.
- Capturas de pantalla y videos: Permite tomar automáticamente capturas de pantalla en casos fallidos, o un videos de todo el conjunto de pruebas cuando se ejecuta desde la CLI.
- Capacidad de depuración: Los errores legibles y el seguimientos de pila hacen que la depuración sea muy rápida.
- Espera automática: Cypress espera automáticamente los comandos y las afirmaciones antes de continuar. No más infierno asíncrono.
- Espías, Stubs y Temporizadores: permite verificar y controlar el comportamiento de las funciones, las respuestas del servidor o los temporizadores.
- Control de tráfico de red: Permite controlar, resguardar y probar casos sin involucrar al servidor. Se puede bloquear el tráfico de la red como se desee.
- Resultados consistentes: No depende de Selenium ni de los diferentes WebDriver, las pruebas son mas rápidas, consistentes y confiables.
- Pruebas en varios navegadores: Puede ejecutar pruebas dentro de los navegadores de la familia Firefox y Chrome (incluidos Edge y Electron) de manera local y óptima en un proceso de C.

© JMA 2020. All rights reserved

Ecosistema de Cypress

- Test Runner
 - gratuito,
 - de código abierto e
 - instalado localmente
- Dashboard Service
 - de pago
 - para ejecutar y registrar la pruebas en la nube
 - con paralelización automática y equilibrio de carga,
 - que permite optimizar la CI y ejecutar pruebas significativamente más rápidamente.

© JMA 2020. All rights reserved

Instalación

- Vía npm:
 - `npm install cypress --save-dev`
 - `npm i eslint-plugin-cypress -D`
- En package.json:

```
"scripts": {  
  ...  
  "e2e": "cypress open",  
  "ci": "cypress run"  
},  
"eslintConfig": {  
  "extends": [  
    ...  
    "plugin:cypress/recommended"  
  ]  
},
```
- Ejecución:
 - `npm run e2e`
 - `npx cypress info`
 - `node node_modules\cypress\bin\cypress info`

© JMA 2020. All rights reserved

e2e: Cypress

- Escribir pruebas e2e requiere muchas herramientas diferentes trabajando juntas. Cypress es un todo en uno, no es necesario instalar 10 herramientas y bibliotecas independientes para configurar el entorno de pruebas. Han tomado algunas de las mejores herramientas de su clase y las han hecho funcionar juntas sin problemas.
- Las pruebas de Cypress solo están escritas en JavaScript, el código de prueba se ejecuta dentro del propio navegador.
- Para habilitarlo y ejecutarlo en las últimas versiones de Angular:
 - `ng add @cypress/schematic`
 - `ng e2e`
 - `ng g spec`
- Para evitar conflictos entre Jasmine (unit tests) y Chai (e2e tests), añadir al principio del `tsconfig.json` de la raíz del proyecto:

```
{ "exclude": ["cypress", "../cypress.config.ts"],
```
- Y en el `tsconfig.json` del directorio cypress, sustituir `"include": ["**/*.ts"]`, por:

```
"include": ["**/*.ts", "cypress", "../cypress.config.ts"],  
"exclude": [],
```

© JMA 2020. All rights reserved

Extensiones Visual Studio Code

- Cypress Fixture-IntelliSense:
 - Supports your `cy.fixture()` and `cy.route(..., "fixture:")` commands by providing intellisense for existing fixtures.
- Cypress Helper:
 - Various helpers and commands for integration with Cypress.
- Cypress Snippets:
 - Useful Cypress code snippets.
- Open Cypress:
 - Allows you to open Cypress specs and single `it()` blocks directly from VS Code.
- Test Utils:
 - Easily add or remove `.only` and `.skip` modifiers with keyboard shortcuts or the command palette

© JMA 2020. All rights reserved

IntelliSense


- La forma más sencilla de ver IntelliSense al escribir un comando o afirmación de Cypress es agregar una directiva de triple barra al encabezado del archivo de prueba en JavaScript o TypeScript.
`/// <reference types="Cypress" />`
- Si se escriben comandos personalizados y se proporciona definiciones de TypeScript, también se puede usar las directivas de triple barra para mostrar IntelliSense, incluso si el proyecto solo usa JavaScript.
`/// <reference types="../support" />`
- En lugar de agregar directivas de triple barra a cada archivo de especificaciones de JavaScript, algunos IDE (como VS Code) utilizan un archivo común `jsconfig.json` en la raíz del proyecto.

```
{  "include": ["/node_modules/cypress", "cypress/**/*.js"]}
```
- Agregar un fichero local `tsconfig.json` en la carpeta `cypress` con la siguiente configuración debería hacer que la finalización inteligente del código funcione.

```
{  "compilerOptions": {    "allowJs": true,    "types": ["cypress"]  },  "include": ["**/*.ts"]}
```

© JMA 2020. All rights reserved

Cypress Test Runner

- El entorno Cypress muestra las pruebas disponibles, el Dashboard y la configuración. Permite seleccionar la prueba y el navegador para ejecutar y parar las pruebas.
- Cypress ejecuta la pruebas en un único Test Runner interactivo y continuo que permite ver los comandos a medida que se ejecutan mientras también se ve la aplicación bajo prueba.
- El Registro de comandos del lado izquierdo del Test Runner es una representación visual de su conjunto de pruebas y La Aplicación bajo prueba (AUT) del lado derecho se usa para mostrar la instantánea de la aplicación bajo prueba.  El Selector Playground, parte superior derecha, es una función interactiva que ayuda a:
 - Obtener un selector único para un elemento.
 - Ver qué elementos coinciden con un selector determinado.
 - Ver qué elemento coincide con una cadena de texto.

© JMA 2020. All rights reserved

Prioridad de Selector Playground

1. data-cy
2. data-test
3. data-testid
4. id
5. class
6. tag
7. attributes
8. nth-child

© JMA 2020. All rights reserved

Cypress CLI

- Cypress dispone de un interfaz de línea de comandos (CLI) que permite su ejecución sin interfaz gráfico para su integración en procesos de CI o en scripts:
 - `cypress run`
- Cuando se ejecuta sin interfaz gráfico genera capturas en video de las ejecuciones de cada una de las pruebas e instantáneas de los casos fallidos.
- Las pruebas se ejecutan por defecto Electron, pero el argumento “browser” puede ajustarse a chrome, chromium, edge, electron o firefox para lanzar otro navegador detectado en su sistema. Cypress intentará encontrar automáticamente el navegador instalado si no se le proporciona la ruta completa.
 - `cypress run --browser chrome`
- A través de los argumentos se puede personalizar la ejecución con valores alternativos para la configuración, variables de entorno ...

© JMA 2020. All rights reserved

Configuración

- La primera vez que se abre Cypress Test Runner, crea el archivo `cypress.json` de configuración en la carpeta raíz del proyecto. Se dispone de múltiples opciones de configuración (visibles desde la opción Settings), por ejemplo:
 - `baseUrl` URL utilizada como prefijo para la URL del comando `cy.visit()` o `cy.request()`
 - `testFiles` Una cadena o matriz de patrones globales de los archivos de prueba para cargar (Por defecto `**/*.spec.js`)

```
{
  "baseUrl": "http://localhost:8181",
  "testFiles": "**/*.spec.js",
  "$schema": "https://on.cypress.io/cypress.schema.json"
}
```

© JMA 2020. All rights reserved

Configuración

- La primera vez que se abre Cypress Test Runner genera la siguiente estructura de directorios para organizar las pruebas:

<code>cypress/fixtures</code>	Contiene los archivos de accesorios
<code>cypress/integration</code>	Contiene los archivos de prueba de integración
<code>cypress/plugins</code>	Contiene los archivos de complementos (se).
<code>cypress/support</code>	Contiene los archivos para cargar antes de cargar los de prueba.
- Se crearan dinámicamente en función a las operaciones realizadas:

<code>cypress/downloads</code>	Ruta a la carpeta donde se guardan los archivos descargados durante una prueba
<code>cypress/screenshots</code>	Ruta a la carpeta donde se guardarán las capturas de pantalla
<code>cypress/videos</code>	Ruta a la carpeta donde se guardarán los videos durante cypress run

© JMA 2020. All rights reserved

spec

```
describe('My First Test', () => {
  it('clicking "type" navigates to a new url', () => {
    cy.visit('https://example.cypress.io')

    cy.contains('type').click()
    cy.url().should('include', '/commands/actions')
    cy.get('.action-email')
      .type('fake@email.com')
      .should('have.value', 'fake@email.com')
    cy.get('#password1')
      .type('P@ssw0rd')
      .should('have.value', 'P@ssw0rd')
      .should('have.class', 'focus')
      .prev().should('have.attr', 'style', 'color: orange;')
  })
})
```

© JMA 2020. All rights reserved

ORGANIZAR LAS PRUEBAS

© JMA 2020. All rights reserved

Suites

- Una “suite” es un nombre que describe al género o sección que se va a pasar por un conjunto de pruebas unitarias, además de ser una herramienta que actúa de núcleo necesario para poder tener un orden en el momento de crear las pruebas.
- Las “suites” se crean con la función **describe** (o su alias **context**), que es una función global y con la cual se inicia toda prueba unitaria, y tiene 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.
- El segundo parámetro 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.

© JMA 2020. All rights reserved

Especificaciones

- Una especificación es un caso de prueba con una o más expectativas (algo que se espera) que ponen a prueba el estado del código. Una expectativa 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 **it** (o su alias **specify**) que, al igual que describe, recibe una cadena y una función. La cadena es el título de la especificación y la función es la especificación o prueba.

```
it('y así es una especificación', function() {  
  //...  
});
```
- **describe** y **it** 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.

© JMA 2020. All rights reserved

Especificaciones

- Al agregar un argumento de devolución de llamada (generalmente llamado `done`) a la función de la especificación o prueba, Cypress sabrá que debe esperar a que se llame a esta función para completar la prueba.
- Si se define dicho argumento es obligatorio invocar la función para que termine la prueba. Esta devolución de llamada acepta como argumento una instancia de `Error` (o una subclase de la misma) en cuyo caso se considerara la prueba fallida.

```
it('con control manual', function(done) {  
  //...  
  if(ok)  
    done()  
  else  
    done(new Error('Fallo forzado'))  
});
```

© JMA 2020. All rights reserved

Control de ejecución

- El marcador **.skip** (o la versión `xdescribe()`, `xcontext()`, `xit()` o `xspecify()`) permite desactivar las suites y las especificaciones, estas se omiten cuando se ejecutan las pruebas y aparecerán como pending entre los resultados de la prueba.
- De igual forma, las especificaciones pendientes de implementar (solo texto, sin función) también aparecerán como pending.
- 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 pending 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 solo cuando su suite este activada.
 - `describe.skip('Pruebas de expect', () => {`
 - `it('Pendiente...');`
 - `specify.only('null', () => {`

© JMA 2020. All rights reserved

Estados de prueba

- Una vez que se completa la especificación de Cypress, cada prueba tiene uno de 4 estados:
 - Passed: Las pruebas aprobadas han completado con éxito todos sus comandos sin fallar ninguna afirmación.
 - Failed: Buenas noticias: la prueba fallida ha detectado un problema. Podría ser mucho peor, ¡podría ser un usuario el que detecte el error!
 - Pending: No se ha ejecutado la prueba porque está marcada con skip, la ha excluido un only o está pendiente de implementar.
 - Skipped: Para las pruebas que se pretendían ejecutar pero que se omitieron debido a algún error en tiempo de ejecución.

© JMA 2020. All rights reserved

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, Cypress suministra las funciones globales (hooks), a nivel de fichero, o locales dentro de una suite:
 - `before(fn)` se ejecuta solo una vez antes empezar a ejecutar las especificaciones del “describe” o fichero.
 - `beforeEach(fn)` se ejecuta antes de cada especificación.
 - `afterEach(fn)` se ejecuta después de cada especificación.
 - `afterAll(fn)` se ejecuta solo una vez después de ejecutar todas las especificaciones del “describe” o fichero.

© JMA 2020. All rights reserved

Configuración de prueba

- Se puede aplicar un valor de configuración de Cypress específico a una suite o especificación con un parámetro intermedio. La configuración estará en vigor durante la ejecución de la suite o las especificaciones en las que se establezcan y volverá a sus valores predeterminados anteriores una vez se hayan completado.
 - `describe(name, config, fn)` o `context(name, config, fn)`
 - `it(name, config, fn)` o `specify(name, config, fn)`
- Para excluir o limitar los navegadores
`describe('When NOT in Chrome', { browser: '!chrome' }, () => {
 it('When in Firefox', { browser: 'firefox' }, () => {`
- Para controlar el tamaño de la pantalla:
`it('iPhone 4', { viewportWidth: 320, viewportHeight: 480 }, () => {`

© JMA 2020. All rights reserved

Pruebas dinámicas

- Se pueden generar pruebas dinámicamente utilizando JavaScript para crear múltiples funciones de especificación.

```
describe('Comprueba los enlaces principales', () =>
  ['/calculadora', '/compras', '/biblioteca', '/contactos', '/alertas',
    '/documentacion'].forEach((url) => {
    it('Ruta: ' + url, () => {
      cy.visit(url)
      cy.url().should('contain', url)
    })
  })
})
```

© JMA 2020. All rights reserved

COMANDOS

© JMA 2020. All rights reserved

API de Cypress

- El objeto global Cypress es el punto de entrada al API de Cypress que da acceso a la configuración, variables de entorno, entorno y contexto de ejecución y persiste durante la totalidad de las pruebas.
- La constante cy, alias de Cypress.cy, da accesos a los comandos que controlan al navegador durante el proceso de pruebas.
- Cypress incluye automáticamente jQuery y lo expone como Cypress.\$. Es una buena manera de consultar elementos de forma síncrona al depurar desde las herramientas de desarrollo.
- También se incluye las siguientes herramientas de terceros en el objeto Cypress.
 - Cypress._ ([lodash](#)): utilidades de JavaScript moderna
 - Cypress.minimatch ([minimatch.js](#)): para probar patrones glob
 - Cypress.Blob ([Blob utils](#)): convertir cadenas base64 en objetos Blob
 - Cypress.Promise ([Bluebird](#)): implementación avanzada de promesas
 - Cypress.dom: es una colección de métodos auxiliares del DOM.

© JMA 2020. All rights reserved

Comandos

- Los comandos simulan la interacción de un usuario con la aplicación. Por debajo, Cypress dispara los eventos que un navegador dispararía, lo que provocara que se disparen los controladores de eventos de la aplicación.
- Cypress, antes de emitir cualquiera de los comandos, verifica el estado actual del DOM y toma algunas acciones para asegurarse de que un elemento DOM está listo para recibir la acción:
 - Está visible (no está oculto o cubierto por otro elemento)
 - No está deshabilitado (disabled), separado (detached) o es de solo lectura (readonly) si se va a modificar
 - Esperará hasta que se detengan las animaciones
 - Desplazará (scrolling) la vista en caso de ser necesario
 - Las coordenadas de los eventos se disparan en el centro del elemento, pero la mayoría de los comandos permiten cambiar la posición a la que se dispara.
- Cypress esperará a que el elemento pase todas estas comprobaciones dentro del tiempo de espera establecido (defaultCommandTimeout).
- Se puede forzar la acción de la mayoría de los comandos con {force: true} para eludir la mayoría de las comprobaciones anteriores.

© JMA 2020. All rights reserved

Cadenas de comandos

- Los comandos de Cypress son asíncronos, no hacen nada en el momento en que se invocan, se colocan en cola para ejecutarse más tarde.
- Cypress es un API fluyente que permite encadenar una serie de comandos, hasta que la cadena finaliza o se encuentra un error.
`cy.get('textarea.post-body').type('This is an excellent post.')`
- Cypress, para encadenar comandos, gestiona una cadena de Promesas en nombre del desarrollador, con cada comando cediendo un 'sujeto' al siguiente comando, hasta que la cadena finaliza o se encuentra un error.
- El desarrollador no debería necesitar usar Promises directamente pero, cuando se desee interactuar con un elemento DOM directamente, se puede llamar a `.then()`

© JMA 2020. All rights reserved

Sujeto del comando

- Un sujeto es el resultado de un comando. Una nueva cadena de comandos siempre comienza con `cy.[command]`, que establece qué otros comandos se pueden llamar a continuación (encadenados).
 - Algunos ceden el sujeto original, otros generan un nuevo sujeto y algunos devuelven null (por lo que no se pueden encadenar nuevos comandos).
 - Algunos ceden un elemento DOM, lo que permite encadenar más comandos a ellos (asumiendo que esperan un sujeto DOM).
- Los comandos de Cypress no devuelven los sujetos, los ceden.
- Cuando se desee interactuar con un sujeto directamente, hay que llamar a `.then()` (o `.spread()` para que expanda una matriz en varios argumentos) con una función de devolución de llamada que recibirá el elemento como primer argumento.

```
cy.get('#element').then((myElement) => {  
  doSomething(myElement)  
})
```

© JMA 2020. All rights reserved

Características de los comandos

- Los comandos son asíncronos, no hacen nada en el momento en que se invocan, sino que se colocan en cola para ejecutarse más tarde.
- Las acciones de los comandos se ejecutan siempre en serie (una tras otra), en el mismo orden en que entraron en la cola, nunca en paralelo (al mismo tiempo).
- Los comandos son promesas, cuando se dice que se están poniendo en cola la acciones que se ejecutarán más adelante, sería más exacto decir “se agregan promesas a una cadena de promesas”.
- Los comandos no pueden tratarse como promesas: No se pueden ejecutar varios comandos al mismo tiempo, es obligatorio encadenar las promesas, no pueden tener catch para controlar los errores de un comando fallido.

© JMA 2020. All rights reserved

Comandos principales

- Los comandos principales siempre comienzan una nueva cadena de comandos. Incluso si se ha encadenado de un comando anterior, los comandos principales siempre iniciarán una nueva cadena e ignorarán los sujetos previamente entregados. Los mas comunes son:
 - `cy.visit()`: Carga una página remota.
 - `cy.go()`: Navega hacia atrás ('back') o hacia adelante ('forward') en el historial del navegador.
 - `cy.reload()`: Vuelve a cargar la página.
 - `cy.request()`: Realiza una solicitud HTTP, permite interactuar con servicios REST.
 - `cy.exec()`: Ejecuta un comando del sistema.
 - `cy.task()`: Ejecuta el código en Node a través del evento task de complemento.
 - `cy.wrap()`: Envuelve el objeto pasado para ceder un sujeto que se puede encadenar con otros valores.

© JMA 2020. All rights reserved

Consulta de Elementos

- Cypress aprovecha el potente motor de selección de jQuery (selectores CSS) para ayudar a que las pruebas sean familiares y legibles para los desarrolladores web modernos.
- Sin embargo, acceder a los elementos DOM devueltos por la consulta funciona de manera diferente: Cypress reintenta automáticamente la consulta hasta que encuentra el elemento o se alcanza el tiempo de espera establecido.
- Cypress envuelve todas las consultas DOM con una sólida lógica de reintento y tiempo de espera que se adapta mejor al funcionamiento de las aplicaciones web reales. En la mayoría de los comandos se pueden personalizar los timeout.
`cy.get('.my-selector', { timeout: 10000 })`
- Otra forma de ubicar las cosas, de forma más humana, es buscarlas por su contenido, por lo que el usuario vería en la página.
`cy.contains('Leer mas ...')`

© JMA 2020. All rights reserved

Selectores

- Un selector o localizador dice cómo encontrar un cierto elemento DOM o un conjunto de ellos. Los selectores mas comunes son:
 - La etiqueta con el atributo ID a un valor: #id
 - Las etiquetas que contengan un estilo en su atributo CLASS: .clase
 - Las etiquetas de un determinado tipo: input
 - Las etiquetas con un determinado atributo: [type]
 - Las etiquetas con un determinado valor en un atributo: [type=submit]
 - Las etiquetas que cumplan una relación jerárquica: :nth-child(\${index})
- Los diferentes tipos de selectores se pueden combinar para obtener un potente sistema de localización, aunque insuficiente o muy complicado en determinadas ocasiones.
- Como “trampilla de escape”, para elementos donde el contenido del texto y la etiqueta no tienen sentido o no son prácticos, la forma recomendada es la incorporación de atributos a las etiquetas para encontrar elementos como data-cy, data-test o data-testid, que se pueden eliminar en el paso a producción (la prueba no debería interferir con el objeto de la prueba).

© JMA 2020. All rights reserved

Comandos de selección

- `cy.root()`: Obtiene el elemento DOM raíz (<HTML>).
- `cy.window()`: Obtiene el objeto window de la página actual.
- `cy.document()`: Obtiene el window.document de la página actual.
- `cy.title()`: Obtiene la propiedad document.title de la página actual.
- `cy.location()`: Obtiene el objeto global window.location.
- `cy.url()`: Obtiene la URL actual de la página actual.
- `cy.hash()`: Obtiene la hash (#) URL actual de la página actual.
- `cy.focused()`: Obtiene el elemento DOM que tiene el foco.
- `cy.get()`: Obtiene los elementos DOM por un selector.
- `cy.contains()`: Obtiene los elementos DOM por un texto.

© JMA 2020. All rights reserved

Comandos de selección

- `.first()`: Obtiene el primer elemento DOM de la colección sujeto.
- `.last()`: Obtiene el último elemento DOM de la colección sujeto.
- `.eq()`: Obtiene el elemento DOM indicando su índice dentro de la colección sujeto.
- `.filter()`: Filtra los elementos DOM que coinciden con un selector.
- `.not()`: Filtra los elementos DOM que no coinciden con un selector.
- `.find()`: Busca los elementos DOM descendientes de un selector.
- `.contains()`: Obtiene los elementos DOM que contienen el texto.
- `.its()`: Obtiene el valor de una propiedad del sujeto.
- `.invoke()`: Invoca una función por su nombre (cadena) sobre el sujeto.
- `.each()`: Itera a través de la colección sujeto.
- `.within()`: Aplica todos los comandos cy subsiguientes dentro de este elemento.
- `.closest()`: Obtiene el primer elemento DOM que coincida con el selector (ya sea él mismo o uno de sus antepasados).

© JMA 2020. All rights reserved

Comandos de selección

```
cy.get('img').first().should('to.have.attr', 'alt', 'Calculadora')
cy.get('img').last().should('to.have.attr', 'alt', 'Alertas')
cy.get('img').eq(2).should('to.have.attr', 'alt', 'Contactos')
cy.get('.nav-link').filter('.active').should('contain', 'Inicio')
cy.get('.list>li').its('length').should('be.gt', 2)
cy.get('#btnOpen').should('be.hidden').invoke('show').should('be.visible')
cy.get('.connectors-each-ul>li').each(($el, index, $list) => {
  console.log($el, index, $list)
})
cy.get('.card>.card-body>.card-title').contains('Contactos')
  .closest('.card').within(() => {
    cy.contains('Ver mas').click()
  })
```

© JMA 2020. All rights reserved

Comandos de selección

- `.children()`: Obtiene los hijos de cada elemento DOM dentro de un conjunto de elementos DOM.
- `.parent()`: Obtiene el elemento DOM principal de un conjunto de elementos DOM.
- `.parents()`: Obtiene los elementos DOM antecesores de un conjunto de elementos DOM.
- `.parentsUntil()`: Obtiene todos los antepasados de cada elemento DOM en un conjunto de elementos DOM coincidentes hasta el elemento proporcionado, sin incluirlo.
- `.siblings()`: Obtiene todos los hermanos de cada elemento DOM dentro de un conjunto de elementos DOM.
- `.prev()`: Obtiene el hermano inmediatamente anterior de cada elemento DOM dentro de un conjunto de elementos DOM.
- `.prevAll()`: Obtiene todos los hermanos anteriores de cada elemento DOM dentro de un conjunto de elementos DOM.
- `.prevUntil()`: Obtiene todos los hermanos anteriores de cada elemento DOM dentro de un conjunto de elementos DOM hasta el elemento proporcionado, sin incluirlo.
- `.next()`, `.nextAll()`, `.nextUntil()`: Obtiene el hermano o todos los hermanos siguientes de cada elemento DOM dentro de un conjunto de elementos DOM.

© JMA 2020. All rights reserved

Comandos de selección

- `cy.get('.navbar-nav').find('.active').contains('Inicio')`
- `cy.get('.navbar-nav').find('.active').next().contains('Calculadora')`
- `cy.get('.navbar-nav').find('.active').nextAll().should('have.length', 7)`
- `cy.get('.navbar-nav').find('.active').siblings().last().find('.fa-info-circle')`
- `cy.get('.navbar-nav').find('.active').siblings().last().prev().contains('APIs')`
- `cy.get('.navbar-nav').find('.active').parent().should('to.have.class', 'mr-auto')`

© JMA 2020. All rights reserved

Comandos de Acción

- `.click()`: Hacer clic en un elemento DOM.
- `.dblclick()`: Hacer doble clic en un elemento DOM.
- `.rightclick()`: Hacer clic con el botón derecho en un elemento DOM.
- `.blur()`: Quitar el foco del elemento DOM que tiene el foco.
- `.focus()`: Poner el foco en un elemento DOM.
- `.type()`: Teclear (introducir un texto) en un elemento DOM.
- `.clear()`: Borrar el valor de un input o textarea.
- `.check()`: Marcar la(s) casilla(s) de verificación o de radio.
- `.uncheck()`: Desmarcar la(s) casilla(s) de verificación.
- `.select()`: Seleccionar un `<option>` dentro de un `<select>`.
- `.submit()`: Envía un formulario.
- `.trigger()`: Activa un evento en un elemento DOM.
- `.scrollIntoView()`: Desplazar hasta el sujeto.
- `.scrollTo()`: Desplazar hasta una posición específica.
- `.shadow()`: Atravesar la sombra DOM para acceder a un elemento.

© JMA 2020. All rights reserved

Comandos de Acción

```
cy.get('#btnOK').click()
cy.get('#scrollable-horizontal').scrollTo('right').click(80, 75).click(170, 75).click(80, 165)
cy.focused().dblclick()
cy.get('[type="email"]').clear().type('me@email.com').blur()
cy.get('[type="radio"]').first().not('[disabled]').check().should('be.checked')
cy.get('[type="checkbox"]').uncheck().should('not.be.checked')
cy.get('#city').select('Madrid').should('have.value', 'Madrid')
cy.get('form').submit().next().should('contain', 'Your form has been submitted!')
cy.get('#scroll-horizontal button').scrollIntoView().should('be.visible')
```

© JMA 2020. All rights reserved

Responsive Design

- El diseño web adaptable es una técnica de diseño y desarrollo cuyo objetivo es adaptar la apariencia de las páginas web al dispositivo que se esté utilizando para visitarlas.
- La apariencia de la pagina se adapta al tamaño de la pantalla (viewport) y debe ser validado con las correspondientes baterías de pruebas.
- `cy.viewport` controla el tamaño y la orientación de la pantalla de la aplicación.
`cy.viewport(550, 750)`
`cy.viewport('iphone-6')`
`cy.viewport('samsung-s10', 'landscape')`
- Se puede configurar el tamaño de la altura y el ancho de la ventana gráfica dentro de una suite o especificación pasando el nuevo valor de configuración:
`describe('ipad-mini', { viewportWidth: 768, viewportHeight: 1024}, () => {`

© JMA 2020. All rights reserved

Variables y alias

- No se puede asignar ni trabajar con los valores de retorno de ningún comando de Cypress. Los comandos se ponen en cola y se ejecutan de forma asíncrona.
- Para acceder al sujeto que produce cada comando de Cypress se usa `.then()`, `.each()`, `.within()` o los callback de los diferentes comandos.
- Los comandos fuera del `.then()` no se ejecutarán hasta que finalicen todos los comandos anidados en el. Las variables locales definidas anteriormente pueden ser usadas mediante clausura en los tratamientos anidados posteriores.
`let pedidos = 0;`
`cy.get('#listadoCarrito tr').then((rows) => pedidos = rows.length - 2)`
`cy.get('#filtroResult > li').first().find('button').click()`
`cy.get('#listadoCarrito tr').should((rows) => {`
 `expect(rows).toHaveLength(pedidos + 3)`
 `expect(localStorage).has.property('CarritoCompra')`
 `expect(JSON.parse(localStorage.CarritoCompra)).toHaveLength(pedidos + 1)`
})

© JMA 2020. All rights reserved

Variables y alias

- Los alias permiten asociar un nombre al sujeto cedido por un comando para compartir el contexto (hooks), reutilizar elementos seleccionados, interceptores, peticiones, ... Los alias son una poderosa construcción en Cypress que tiene muchos usos.
- El comando `.as` asigna un nombre (alias) al resultado del comando anterior para su uso posterior.
`cy.get('button[type=submit]').as('btnSubmit')`
- Para hacer referencia al alias más adelante dentro de un comando se utiliza el nombre con una `@` como prefijo.
`cy.get('@btnSubmit').click().should('be.disabled')`

© JMA 2020. All rights reserved

Variables de entorno

- Las variables de entorno son útiles cuando:
 - Los valores son diferentes en las máquinas de desarrollo.
 - Los valores difieren según los entornos (dev, staging, qa, prod).
 - Los valores cambian con frecuencia y son muy dinámicos.
- En Cypress, las "variables de entorno" son variables a las que se puede acceder mediante `Cypress.env`.
`cy.request(Cypress.env('EXTERNAL_API'))`
- No son lo mismo que las variables de entorno del sistema operativo, aunque es posible establecer variables de entorno de Cypress a partir de variables de entorno del sistema operativo (que comiencen con `CYPRESS_` o `cypress_`).
`export CYPRESS_API_SERVER=http://localhost:8888/api/v1/`

© JMA 2020. All rights reserved

Variables de entorno

- Se pueden establecer en el archivo de configuración cypress.json:

```
{
  :
  "env": { "login_url": "/login", "products_url": "/products" }
}
```

- Se puede crear el archivo cypress.env.json que Cypress comprobará automáticamente:

```
{
  "login_url": "/login",
  "products_url": "/products"
}
```

- Se pueden pasar variables de entorno como opciones cuando se utiliza la herramienta CLI:
cypress run --env mode=developer

© JMA 2020. All rights reserved

ASERCIONES

© JMA 2020. All rights reserved

Afirmaciones

- Las afirmaciones describen el estado deseado de los elementos, los objetos y las aplicación.
- Las afirmaciones permiten hacer cosas como asegurarse de que un elemento sea visible o tenga un atributo, una clase CSS o un estado en particular. Las afirmaciones son comandos que permiten describir el estado deseado de la aplicación. Cypress esperará automáticamente hasta que los elementos alcancen este estado, o fallará la prueba si las afirmaciones no pasan.
- Muchos comandos tienen una aserción predeterminada e incorporada, o más bien tienen requisitos que pueden hacer que falle sin necesidad de una aserción explícita. El comando `.should()` y su alias `.and()` permiten encadenar más fácilmente las afirmaciones a los comandos.

© JMA 2020. All rights reserved

Estilos de afirmación

- Cypress admite afirmaciones simples de estilo BDD (`expect/should`) y TDD (`assert`).
- Cypress incluye la popular biblioteca de aserciones Chai, así como útiles extensiones para Sinon y jQuery, que brindan docenas de poderosas aserciones para proporcionar aserciones integradas.
- Hay dos formas de escribir afirmaciones en Cypress:
 - Sujetos implícitos: usando `.should()` o `.and()`.
 - Sujetos explícitos: usando `expect`.

© JMA 2020. All rights reserved

Sujetos implícitos

- Los comandos `.should()` y `.and()` son la forma preferida de hacer afirmaciones en Cypress, se puede encadenar varias afirmaciones juntas usando `.and()`, que es un alias semántico de `.should()` que hace que las expresiones sean más legibles. Son comandos de Cypress, lo que significa que se aplican al sujeto generado en la cadena de comandos.

```
cy.get('#header a')  
  .should('have.class', 'active')  
  .and('have.attr', 'href', '/users')
```

© JMA 2020. All rights reserved

Sujetos explícitos

- El uso de `expect` permite pasar un sujeto específico y hacer una afirmación al respecto. Es el mecanismo habitual en las afirmaciones de las pruebas unitarias. Las afirmaciones explícitas permiten:
 - Realizar una lógica personalizada antes de realizar la aserción.
 - Hacer múltiples afirmaciones contra el mismo sujeto, incluidos sus elementos secundarios sin necesidad de consultarlos individualmente.
- El comando `.should()` nos permite pasar una función de devolución de llamada que toma el sujeto cedido como su primer argumento, como el `.then()`, excepto que Cypress espera que pase todo dentro del callback y lo vuelve a intentar automáticamente.

```
cy.get('tbody tr:first').should(($tr) => {  
  expect($tr).to.have.class('active')  
  expect($tr).to.have.attr('href', '/users')  
})
```

© JMA 2020. All rights reserved

Chai

- Los estilos BDD son expect y should. Ambos usan el mismo lenguaje encadenable para construir aserciones, pero difieren en la forma en que se construye inicialmente una aserción: texto o métodos.
- Los métodos y valores suministrados son: deep, nested, own, ordered, any, all, a, include, ok, true, false, null, undefined, NaN, exist, empty, arguments, equal, eql, above, least, below, most, within, instanceof, property, ownPropertyDescriptor, lengthOf, match, string, keys, throw, respondTo, itself, satisfy, closeTo, members, oneOf, change, increase, decrease, by, extensible, sealed, frozen, finite, fail
- Las siguientes propiedades se proporcionan como captadores encadenables para mejorar la legibilidad de la afirmaciones: to, be, been, is, that, which, and, has, have, with, at, of, same, but, does, still, also.
- Las afirmaciones negativas tienen el encabezado not. prefijando a la afirmación.

© JMA 2020. All rights reserved

Comandos

```
.should('to.not.equal', 'Jane')  
.should('be.oneOf', [200, 201, 204, 304])  
.should('to.deep.equal', { name: 'Jane' })  
.should('to.have.any.keys', 'name')  
.should('be.gt', 2)  
.should('to.be.a', 'string')  
.should('to.include', 'Jane')  
.should('to.be.true')  
.should('to.exist')  
.should('to.be.empty')  
.should('to.have.property', 'name')  
.should('have.property', 'name', 'Jane')  
.should('to.have.lengthOf', 3)
```

© JMA 2020. All rights reserved

Chai-jQuery

- Los encadenadores disponibles cuando se afirma sobre un objeto DOM son: attr, prop, css, data, class, id, html, text, value, visible, hidden, selected, checked, focus, enabled, disabled, empty, exist, match, contain, descendants.

```
cy.get('li.selected').should('have.length', 3)
cy.get('span').first().should('to.have.class', 'fa-info-circle')
cy.get('form').find('input').should('not.have.class', 'disabled')
cy.get('.completed').should('have.css', 'text-decoration', 'line-through')
cy.get('textarea').should('have.value', 'foo bar baz')
cy.get('#address').should('include.text', 'Atlanta')
cy.get('a').parent('span.help').should('not.contain', 'click me')
cy.get('button#form-submit').should('be.visible')
cy.get(':radio').should('be.checked')
cy.get('#loading').should('not.exist')
```

© JMA 2020. All rights reserved

Timeout

- Cypress ofrece varios valores de tiempo de espera diferentes según el tipo de comando (requestTimeout, defaultCommandTimeout, execTimeout, ...).
- Casi todos los comandos pueden expirar de alguna manera. Se puede modificar el tiempo de espera de un comando, pero este tiempo de espera afecta tanto a sus aserciones predeterminadas (si las hay) como a las aserciones específicas que se hayan agregado.
`cy.get('.mobile-nav', {timeout: 10000}).should('be.visible')`
- El comando `cy.wait()` permite esperar una cantidad fija de milisegundos o a que se resuelva un recurso con alias antes de pasar al siguiente comando.
`cy.wait(500)`

© JMA 2020. All rights reserved

Reintentos

- Una característica principal de Cypress que ayuda a probar aplicaciones web dinámicas es la capacidad de reintento.
- Con los reintentos de prueba, Cypress puede reintentar automáticamente las pruebas fallidas para ayudar a reducir el número final de pruebas fallidas y los fallos de compilación en la integración continua (CI).
- Los comandos no se reintentan cuando potencialmente podrían cambiar el estado de la aplicación bajo prueba. Solo se reintentan el último comando, el que ha fallado.
- De forma predeterminada, las pruebas no se reintentan cuando fallan, se deberá habilitar los reintentos en la configuración:

```
{  
  "retries": {  
    "runMode": 2,  
    "openMode": 0  
  }  
}
```

© JMA 2020. All rights reserved

DEPURACIÓN

© JMA 2020. All rights reserved

Depuración

- El código de prueba de Cypress se ejecuta en el mismo ciclo de ejecución que la aplicación. Esto significa que se tiene acceso al código que se ejecuta en la página, así como al BOM: `document`, `window`, ...
- Cypress suministra una extensa información sobre los errores que se producen y salta a la ubicación del mismo, lo que facilita la depuración de las pruebas. Cypress también expone un acceso directo para la depuración con el comando `.debug()` para inspeccionar el sujeto que sustituye a la instrucción síncrona `debugger`.
`cy.get('.selector-in-question').debug()`
- Se puede detener la ejecución de los comandos, para interactuar con la aplicación bajo prueba y, a continuación, "reanudar" la ejecución de todos los comandos o elegir pasar por los comandos "siguientes" del Registro de comandos.
`cy.get('nav').pause()`
- Así mismo, con `cy.log()` se puede imprimir un mensaje en Cypress Command Log.
`cy.log('created new user')`

© JMA 2020. All rights reserved

Capturas de pantalla y videos

- Cypress trae la capacidad de tomar capturas de pantalla, tanto para cuando se esté ejecutando a través de *cypress open* o como para *cypress run*.
- Cypress realizará automáticamente capturas de pantalla cuando ocurra un fallo durante *cypress run*, cosa que no ocurrirá con *cypress open*.
- Para tomar una captura de pantalla manual, se puede usar el comando `cy.screenshot()`.
- Las capturas de pantalla se almacenan en la carpeta `screenshotsFolder` que está configurada a `cypress/screenshots` de forma predeterminada.
- Además, Cypress graba un video con lo que se vería si se ejecutase con el interfaz gráfico del Test Runner, una grabación por cada archivo de especificación y solo cuando se ejecutan las pruebas con *cypress run*.

© JMA 2020. All rights reserved

Informes

- Debido a que Cypress está construido sobre Mocha, permite que cualquier generador de informes creado para Mocha puede usarse con Cypress.
- Cypress trae integrados los siguientes generadores de terceros:
 - mocha-junit-reporter: genera resultados XML al estilo JUnit.
 - mocha-teamcity-reporter: genera resultados de pruebas para TeamCity.
- Los informes de Mocha se muestran en consola con *cypress run* y se ajustan a la ventana de la terminal. Las opciones son: spec (por defecto), dot, nyan, tap, landing, list, progress, json, json-stream, min, doc, markdown, xunit. Se configura en cypress.json:
"reporter": "nyan",

© JMA 2020. All rights reserved

Opciones de informes

- Algunos informes aceptan opciones que personalizan su comportamiento. Estas opciones se pueden especificar en el archivo de configuración (cypress.json) o mediante las opciones de la línea de comandos.
- Las opciones difieren según el generador de informe, incluso puede que no acepten ninguna.
- Para generar el informe en JUnit y guardarlo en un archivo XML.
"reporter": "junit",
"reporterOptions": {
 "mochaFile": "cypress/reports/output.xml",
 "toConsole": true
}
- Para conservar informes únicos para cada ejecución específica, se puede utilizar [hash] en el nombre del archivo.
"mochaFile": "cypress/reports/output-[hash].xml",

© JMA 2020. All rights reserved

Informes HTML

- Para generar informes HTML es necesario instalar y configurar los siguientes módulos:
`npm i -D mochawesome mochawesome-merge mochawesome-report-generator`
- Para configurarlo en el archivo de configuración (`cypress.json`):

```
"reporterOptions": {  
  "reportDir": "cypress/reports",  
  "overwrite": false,  
  "html": false,  
  "json": true  
},
```
- Cada archivo de especificaciones se procesa completamente por separado durante cada ejecución de *cypress run*, por ello no se deben sobre escribir y se deben fusionar los ficheros JSON antes de generar el informe. Se deberían eliminar las ejecuciones anteriores.

© JMA 2020. All rights reserved

Informes HTML

- Para eliminar las ejecuciones anteriores se crea el script node `cypress/support/deleteReports.js`.

```
const fs = require('fs');  
fs.rmdirSync('./cypress/reports', {recursive: true});
```
- Se configuran los siguientes scripts en el `package.json`:

```
"scripts": {  
  :  
  "combine-reports": "mochawesome-merge cypress/reports/mocha*.json >  
cypress/reports/index.json",  
  "generate-report": "marge --autoOpen --charts true --reportDir ./cypress/reports/ --  
reportFilename index ./cypress/reports/index.json",  
  "preci": "node cypress/support/deleteReports.js",  
  "postci": "npm run combine-reports && npm run generate-report",  
  "ci": "cypress run"  
},
```

© JMA 2020. All rights reserved

Múltiples Informes

- A menudo es necesario utilizar varios informes. Cuando se ejecuta en CI, es posible que se desee generar un informe junit además del informe habitual de especificaciones.
- Para ello es necesario instalar y configurar el siguiente módulo:
npm i -D mocha-multi-reporters
- Para configurarlo en el archivo de configuración (cypress.json):

```
"reporter": "mocha-multi-reporters",
"reporterOptions": {
  "reporterEnabled": "mocha-junit-reporter, spec",
  "mochaJunitReporterReporterOptions": {
    "mochaFile": "cypress/reports/output-[hash].xml",
    "toConsole": false
  }
},
```

© JMA 2020. All rights reserved

DOBLES DE PRUEBA

© JMA 2020. All rights reserved

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, dobles de pruebas, 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)
 - DI: Inyección de Dependencias (Dependency Injection)
 - Simuladores de objetos

© JMA 2020. All rights reserved

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.
- Usar dobles de prueba (como los dobles en el cine) tiene ventajas:
 - Devuelven resultados determinísticos
 - Permiten crear o reproducir determinados estados
 - 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.
 - Memorizan los valores con los que se llama a cada uno de sus miembros
 - Permiten verificar si los valores esperados coinciden con los recibidos

© JMA 2020. All rights reserved

Simulación de objetos

- **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 preprogramado 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.

© JMA 2020. All rights reserved

Consideraciones

- Las Pruebas extremo a extremo (e2e) consisten en interactuar con la aplicación tal y como un usuario real lo haría, generando un ratón y teclado virtual que navegue por cada rincón, yendo del cliente al servidor y volviendo al cliente, y evaluando las respuestas para el comportamiento esperado. Se prueba todo el flujo del software desde el punto de vista del usuario final, utilizando la interfaz de usuario, y no desde el código interno. Están enfocadas en detectar posibles problemas que pudieran encontrar los usuarios en su interacción con el flujo general del programa.
- Sin embargo, en muchas ocasiones, sobre todo en aplicaciones web modernas (SPA), puede ser interesante tomar determinados atajos que agilicen las pruebas, por ejemplo: una vez probada la conexión al servicio REST no es necesario que el resto de las pruebas se ralenticen al pasar por el servicio real, una vez probado el interfaz de autenticación se puede realizar una autenticación directa para el resto de los caso o que se tenga que esperar el tiempo real de un temporizador. En otras ocasiones es muy complicado reproducir situaciones anómalas en escenarios reales para validarlas mediante casos de pruebas. Estas consideraciones pueden justificar el uso de dobles de pruebas en una parte de las pruebas e2e.

© JMA 2020. All rights reserved

Espías, Stubs y Temporizadores

- Cypress trae incorporado la capacidad de crear dobles de pruebas y espías con `cy.stub()` y `cy.spy()` o modificar los temporizadores de la aplicación con `cy.clock()` que permite manipular `Date`, `setTimeout`, `clearTimeout`, `setInterval` o `clearInterval`.
- Estos comandos son útiles al escribir pruebas unitarias y pruebas de integración, así como para agilizar las pruebas e2e.
- Cypress agrupa y envuelve automáticamente las bibliotecas:
 - sinon: proporciona las API `cy.stub()` y `cy.spy()`
 - sinon-chai: agrega afirmaciones chai para stubs y espías
 - lolex: proporciona las API `cy.clock()` y `cy.tick()`

© JMA 2020. All rights reserved

Espías

- Un espía intercepta una función, permitiendo capturar su ejecución para luego poder afirmar si la función fue llamada con los argumentos correctos o si la función fue llamada un cierto número de veces, incluso cuál fue el valor de retorno o con qué contexto se llamó a la función.
- Un espía no modifica el comportamiento de la función, la envuelve dejándola intacta. Un espía es más útil cuando se está probando el contrato entre múltiples funciones y no importan los efectos secundarios que la función real puede crear (si los hay).
- `cy.spy()` envuelve un método en un espía para registrar las llamadas y los argumentos de la función. A diferencia de la mayoría de los comandos de Cypress, `cy.spy()` es sincrónico y devuelve el espía de la función en lugar de un objeto encadenable similar a Promise.
- `cy.spy()` es un envoltorio de la biblioteca Sinon.js que es mucho mas completa.

© JMA 2020. All rights reserved

Espías

- Se pueden crear espías sobre funciones o métodos concretos y sobre todos los métodos de un objeto:

```
let sumaSpy = cy.spy(calc, 'suma')  
let calcSpy = cy.spy(calc)
```
- Después de las diferentes actuaciones, directas o indirectas, se puede consultar las llamadas recibidas:

```
expect(o.suma).to.be.called  
expect(o.suma).to.be.calledWith(2, 2)  
expect(o.suma).to.have.returned(4)
```
- El espía registra información sobre las invocaciones, el número y orden de las mismas, argumentos, valores de retorno, el valor `this` y las excepciones lanzadas (si corresponde) para todas sus llamadas.
- La información registrada puede ser consultada a través de los métodos específicos del espía:

```
expect(o.suma.callCount).to.be.equal(1)
```

© JMA 2020. All rights reserved

Espías

Cypress también tiene soporte incorporado `sinon-chai`, por lo que dispone de aserciones específicas sobre los valores registrados por los espías:

<pre>expect(spy).to.be.called expect(spy).to.have.callCount(n) expect(spy).to.be.calledOnce expect(spy).to.be.calledTwice expect(spy).to.be.calledThrice expect(spy1).to.be.calledBefore(spy2) expect(spy1).to.be.calledAfter(spy2) expect(spy).to.be.calledWithNew expect(spy).to.always.be.calledWithNew expect(spy).to.be.calledOn(context) expect(spy).to.always.be.calledOn(context) expect(spy).to.be.calledWith(...args)</pre>	<pre>expect(spy).to.always.be.calledWith(...args) expect(spy).to.be.calledWithExactly(...args) expect(spy).to.always.be.calledWithExactly(...args) expect(spy).to.be.calledWithMatch(...args) expect(spy).to.always.be.calledWithMatch(...args) expect(spy).to.have.returned(returnVal) expect(spy).to.have.always.returned(returnVal) expect(spy).to.have.thrown(error) expect(spy).to.have.always.thrown(error)</pre>
---	---

© JMA 2020. All rights reserved

Espías

- Las aserciones sinon-chai también se pueden utilizar con el comando should, para integrarlo con el resto de los comandos:

```
cy.visit('/compras', {
  onBeforeLoad(win) {
    cy.spy(win, 'fetch')
  },
})
cy.window().its('fetch').should('be.calledWith', 'api/peliculas')
```
- Al espía se le puede agregar un alias usando .as() para que sean más fáciles de localizar e identificar en los mensajes de error y en el registro de comandos de Cypress:

```
cy.spy(win, 'fetch').as('winSpy')
cy.get('@winSpy').should('be.calledWith', 'api/peliculas?_sort=titulo')
```

© JMA 2020. All rights reserved

Stubs

- Un stub es una forma de modificar una función y definir un comportamiento propio. Los stubs se usa con mayor frecuencia en pruebas unitarias, pero sigue siendo útil durante algunas pruebas de integración y e2e. Básicamente son espías con un comportamiento preprogramado.
- cy.stub() reemplaza una función, registra su uso y controla su comportamiento. A diferencia de la mayoría de los comandos de Cypress, cy.stub() es sincrónico y devuelve el stub de la función en lugar del habitual sujeto encadenable similar a Promise.
- cy.stub() es un envoltorio de la biblioteca Sinon.js que es mucho mas completa.

© JMA 2020. All rights reserved

Stubs

- Creación y reemplazo de métodos:

```
function calc() {  
  this.suma = (a, b) => a + b;  
}  
let o = new calc();  
const sumaStub = cy.stub(o, 'suma')  
o.resta = cy.stub().returns(1)
```

- Especificar el valor de retorno de un método
sumaStub.returns(5)
sumaStub.throws('Error', 'Fallo forzado') // Genera una excepción
- Reemplazar un método con una función
cy.stub(o, 'suma').callsFake((a, b) => 0)

© JMA 2020. All rights reserved

Stubs

- Reemplazar un método solo cuando reciba determinados argumentos (devolverá undefined para el resto si no se define un valor por defecto)
sumaStub
 .withArgs(2, 2).returns(5)
 .withArgs(2, 1).returns(0);
sumaStub.returns(3) // valor por defecto
- Especificar el valor de retorno en función al número de llamadas (onCall en base 0; onFirstCall, onSecondCall, onThirdCall son alias para onCall de 0, 1 y 3; el resto de llamadas devolverá undefined)
sumaStub.withArgs(2, 2)
 .onFirstCall().returns(1)
 .onSecondCall().returns(2)
 .onCall(3).returns(4); // onThirdCall queda undefined
sumaStub.returns(0); // valor por defecto para otros argumentos

© JMA 2020. All rights reserved

Stubs

- Acceder a la versión original
`sumaStub.callsFake(sumaStub.wrappedMethod)`
`sumaStub.callThrough() // Cuando fallan las demás`
- Restaurar la versión original
`o.suma.restore()`
`sumaStub.restore()`
- Deshabilitar el registro en el registro de comandos
`sumaStub = cy.stub(o, 'suma').log(false)`
- Agregar un alias usando `.as()` para que sean más fáciles de identificar en los mensajes de error y en el registro de comandos de Cypress.
`cy.stub(o, 'suma').as('fakeSuma')`

© JMA 2020. All rights reserved

Stubs

- Reemplazar métodos de los objetos del navegador

```
cy.visit('/alertas', {
  onBeforeLoad(win) {
    cy.stub(win, 'prompt').returns('Hola mundo')
  },
})
cy.get('#btnPrompt').click()
cy.window().its('prompt').should('be.called')
cy.get('#CuadroAlerta').contains('Hola mundo')

cy.visit('/compras', {
  onBeforeLoad(win) {
    cy.stub(win, 'fetch').withArgs('api/peliculas?_sort=titulo')
      .resolves({ ok: false, status: 404, statusText: 'Not found',
        json: () => Promise.resolve({ message: 'Error forzado' }), })
  }
})
```

© JMA 2020. All rights reserved

Temporizadores

- Hay situaciones en las que es útil controlar el objeto `date` y los temporizadores para anular su comportamiento o evitar pruebas lentas:
 - Operaciones que dependen de marcas temporales obtenida a través del objeto `Date`.
 - Operaciones diferidas con `setTimeout` donde no es necesario esperar.
 - Sondeos con `setInterval` que se quieren acelerar.
- `cy.clock()` anula las funciones globales nativas relacionadas con el tiempo para que puedan ser controladas sincrónicamente a través de `cy.tick()` o el objeto `clock` devuelto. Esto incluye controlar: `setTimeout`, `clearTimeout`, `setInterval`, `clearInterval` y el objeto `Date`.
- Por defecto el reloj comienza en la época de Unix (marca de tiempo de 0). Esto significa que cuando se cree una instancia `new Date` en la aplicación, es inicializa al 01/01/1970 00:00:00. Esta marca temporal se puede modificar al crear el reloj.

© JMA 2020. All rights reserved

Temporizadores

- Para crear un reloj:

```
let clock = cy.clock()
cy.clock()
```
- Para especificar la marca temporal de “ahora”:

```
// app code
$('#currentDate').text(new Date().toJSON())
// test code
cy.clock(new Date(Date.UTC(2031, 3, 14)).getTime()) // timestamp
:
cy.get('#currentDate').should('have.text', '2031-04-14T00:00:00.000Z')
```
- Se pueden anular solo algunas funciones y dejar las otras funciones relacionadas con el tiempo como están:

```
cy.clock(null, ['setTimeout', 'clearTimeout'])
cy.clock(Date.UTC(2018, 10, 30), ['Date'])
```

© JMA 2020. All rights reserved

Temporizadores

- Para mover el reloj el número de milisegundos especificado (forzara a que se active cualquier temporizador dentro del rango de tiempo afectado):
 `clock.tick(1000)`
 `cy.tick(1000)`
- `cy.tick()` actúa sobre el objeto `clock` generado con `cy.clock()`
- Para restaurar todas las funciones nativas anuladas (se llama automáticamente entre pruebas, por lo que generalmente no debería ser necesario utilizarlo):
 `clock.restore()`
- También se puede acceder al objeto `clock` actual a través de una devolución de llamada en `.then()`.
 `cy.clock().then((clock) => { clock.restore() })`

© JMA 2020. All rights reserved

Temporizadores

```
// app code
let seconds = 0
setInterval(() => {
    $('#crono').text(++seconds + ' seconds')
}, 1000)

// test code
cy.clock()
cy.visit('/alertas')
cy.tick(1000)
cy.get('#crono').should('have.text', '1 seconds')
cy.tick(3000)
cy.get('#crono').should('have.text', '4 seconds')
```

© JMA 2020. All rights reserved

Solicitudes de red

- 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 en las ejecuciones de prueba debido a tiempos de carga lentos.
- En otros escenarios es difícil de obtener estados específicos del servidor, incluyendo status, headers o body de la respuesta o retrasos de la red, para poder realizar los casos de pruebas apropiados.
- Cypress proporciona acceso a los objetos con información sobre una solicitud, lo que permite hacer afirmaciones sobre sus propiedades, así como simular la respuesta de dicha solicitud.
- Las simulaciones son extremadamente rápidas, la mayoría de las respuestas se devolverán en menos de 20 ms. Dado que las respuestas reales pasan por cada capa del servidor (controladores, modelos, vistas, etc.) y, es posible, que se deba inicializar una fuente de datos antes de cada prueba para generar un estado predecible, pueden ser mucho más lentas que las respuestas simuladas.
- Las respuestas simuladas pueden establecer directamente estados específicos.
- Se puede mezclar y combinar dentro de la misma prueba simular ciertas solicitudes, mientras se permite que otras lleguen a su servidor.

© JMA 2020. All rights reserved

Solicitudes directas

- Las solicitudes directas al servidor permiten crear atajos y realizar comprobaciones en el servidor.
- `cy.request()` realiza una solicitud HTTP, donde el cuerpo puede ser una cadena u objeto. Con `options` se pueden configurar todos los aspectos de la petición (método, url, cabeceras, cuerpo, ...):
 - `cy.request(url)`
 - `cy.request(url, body)`
 - `cy.request(method, url)`
 - `cy.request(method, url, body)`
 - `cy.request(options)`
- `cy.request()` cede un objeto `response` con, entre otras cosas:
 - `status`
 - `body`
 - `headers`
 - `duration`

© JMA 2020. All rights reserved

Solicitudes directas

- Se puede tratar la respuesta directamente:

```
cy.request('DELETE', 'api/contactos').then((response) => {  
  expect(response.status).to.eq(204)  
})
```
- Se puede asociar un alias a la respuesta para tratarla a posteriori:

```
Cypress.Commands.add('login', () => {  
  cy.request('POST', '/api/login?cookie=true',  
    { "name": "admin", "password": "P@$w0rd" }  
  ).as('login')  
  cy.get('@login').its('status').should('eq', 200)  
  cy.getCooki('Authorization').should('exist')  
})
```

© JMA 2020. All rights reserved

Interceptar solicitudes de red

- `cy.intercept()` se utiliza para controlar el comportamiento de las solicitudes HTTP, permitiendo espiar, modificar o suplantar las solicitudes.
- Para espiar el resultado de las solicitudes:

```
cy.intercept('POST', 'api/**').as('postREST')  
:  
cy.wait('@postREST').its('response.statusCode').should('eq', 201)
```
- Esperar un interceptor con alias tiene grandes ventajas:
 - Evita que los siguientes comandos se ejecuten hasta que lleguen las respuestas
 - Las pruebas son más robustas.
 - Los mensajes de error son mucho más precisos.
 - Se puede afirmar sobre el objeto de solicitud subyacente.

© JMA 2020. All rights reserved

Interceptar solicitudes de red

- Para espiar y verificar la petición:

```
cy.intercept('api/**', { method: 'POST' }, (req) => {
  expect(req.headers).to.have.property('authorization')
}).as('postREST')
```
- Para modificar la petición y la respuesta:

```
cy.intercept('api/**', { method: 'GET', middleware: true }, (req) => {
  req.headers['cache-control'] = 'no-cache,no-store,max-age=0,must-revalidate';
  delete req.headers['if-none-match']
  req.continue((res) => {
    expect(res.statusCode).be.oneOf([200, 201, 204, 304])
  })
  req.on('before:response', (res) => {
    res.headers['cache-control'] = 'no-store'
  })
}).as('getRest')
```

© JMA 2020. All rights reserved

Suplantar solicitudes de red

- Para generar directamente la respuesta de la solicitud:

```
let listado = [
  { "id": 98, "titulo": "Man of the House" },
  { "id": 99, "titulo": "Innocent Affair" },
  { "id": 100, "titulo": "Damn the Defiant!" }
]
cy.intercept('GET', 'api/peliculas', listado).as('getRest')
cy.intercept('GET', 'api/peliculas', {
  statusCode: 200,
  headers: { 'Content-Type': 'application/json' },
  body: listado
}).as('getRest')
cy.intercept('GET', 'api/peliculas', { statusCode: 404 }).as('getRest')
cy.intercept('GET', 'api/peliculas', { forceNetworkError: true
}).as('getRest')
```

© JMA 2020. All rights reserved

Controlando la respuesta

- La solicitud interceptada que se pasa al controlador del interceptor (req) contiene métodos para controlar dinámicamente la respuesta a una solicitud:
 - req.reply(): generar la respuesta sin realizar la petición real.
 - req.continue(): modificar o hacer afirmaciones sobre la respuesta real
 - req.destroy(): destruir la solicitud y responder con un error de red
 - req.redirect(): responder a la solicitud con una redirección a una ubicación específica
 - req.on(): modificar la respuesta adjuntando eventos (before:response, response, after:response)

© JMA 2020. All rights reserved

Suplantar solicitudes de red

```
cy.intercept('POST', '/users', (req) => {  
  req.reply({  
    headers: {  
      Set-Cookie: 'newUserName=admin';  
    },  
    statusCode: 201,  
    body: {  
      name: 'Administrador'  
    },  
    delay: 10, // milliseconds  
    throttleKbps: 1000, // to simulate a 3G connection  
    forceNetworkError: false // default  
  })  
})
```

© JMA 2020. All rights reserved

Fixtures

- Un fixture es un conjunto fijo de datos ubicados en un archivo que se utiliza en las pruebas. El propósito de un fixture de prueba es garantizar que exista un entorno bien conocido y fijo en el que se ejecuten las pruebas para que los resultados sean repetibles y predecibles.
- Se accede a los fixtures dentro de las pruebas llamando al comando `cy.fixture()`.
`cy.fixture('peliculas.json').as('pelis')`
`cy.fixture('users/admin.png').as('adminPhoto')`
- Cypress crea automáticamente la carpeta sugerida `/cypress/fixtures` para organizar los fixtures en cada nuevo proyecto. Se puede organizar en subcarpetas. Cypress determina automáticamente la codificación para los siguientes tipos de archivos:

<code>.json</code>	<code>.js</code>	<code>.coffee</code>	<code>.html</code>	<code>.txt</code>	<code>.csv</code>	<code>.png</code>
<code>.jpg</code>	<code>.jpeg</code>	<code>.gif</code>	<code>.tif</code>	<code>.tiff</code>	<code>.zip</code>	

© JMA 2020. All rights reserved

Fixtures

- Los fixtures son potencialmente grandes por lo que es conveniente asignarles un alias y esperar a que se terminen de cargar.
- Se puede usar `.then()` para acceder a los datos del fixture:
`cy.fixture('audio/sound.mp3', 'base64').then((mp3) => {`
 `const audio = new Audio('data:audio/mp3;base64,' + mp3)`
 `audio.play()`
 `})`
- Hay que tener en cuenta que se supone que los archivos de fixture no se modifican durante la prueba y, por lo tanto, Test Runner los carga solo una vez, si posteriormente se modifican no los tiene en cuenta.
- Como alternativas se dispone de `cy.readFile()`, que lee un archivo y cede su contenido, y `cy.writeFile()`, que escribe en un archivo con el contenido especificado.
`cy.readFile('cypress/fixtures/ personas.json').then((lst) => { expect(lst.length).eq(20)`
 `})`
 `cy.writeFile('cypress/fixtures/personas.json', response.body.slice(0, 20))`

© JMA 2020. All rights reserved

Fixtures y solicitudes de red

- Con Cypress, puede suplantar solicitudes de red y hacer que responda instantáneamente con datos del fixture.
- Al suplantar una respuesta, normalmente se necesita administrar objetos JSON potencialmente grandes y complejos a los que hay que esperar, por lo que Cypress permite integrar la sintaxis de fixtures directamente en la generación de respuestas que será a la que haya que esperar.

```
cy.intercept('/api/peliculas', { fixture: 'peliculas.json' })
cy.intercept('GET', '/api/peliculas', (req) => {
  req.reply({
    statusCode: 200, // default
    fixture: 'peliculas.json'
  })
}).as('getRest')
```

© JMA 2020. All rights reserved

Estado local

- Cypress borra automáticamente el localStorage, para el dominio y subdominio actual, y todas las cookies antes de cada prueba para evitar que el estado se comparta entre las pruebas. Con `cy.clearLocalStorage()`, `cy.clearCookie()` y `cy.clearCookies()` se puede realizar manualmente en mitad de una prueba.
- `cy.getCookie()` obtiene una cookie del navegador por su nombre y `cy.getCookies()` obtiene todas. Por cada cookie se produce un objeto de cookie con las siguientes propiedades: `domain`, `expiry` (si se especifica), `httpOnly`, `name`, `path`, `sameSite` (si se especifica), `secure`, `value`.
`cy.getCookie('Authorization').should('exist')`
- `cy.setCookie()` establecer una cookie del navegador:
`cy.setCookie('Authorization', Bearer eyJhbGciOiJIbnR5cCI6IkpXVCJ')`

© JMA 2020. All rights reserved

REUTILIZACIÓN

© JMA 2020. All rights reserved

Comandos personalizados

- Cypress viene con su propia API para crear comandos personalizados y sobrescribir los comandos existentes.
- El lugar para definir o sobrescribir comandos es en el archivo `cypress/support/commands.js` o en otros ficheros del mismo directorio.
- Los ficheros adicionales se deben importar mediante una declaración `import` en `cypress/support/index.js`.
- Los comandos se añaden o sobrescriben con:
`Cypress.Commands.add(name, callbackFn)`
`Cypress.Commands.add(name, options, callbackFn)`
`Cypress.Commands.overwrite(name, callbackFn)`

© JMA 2020. All rights reserved

Crear comandos

- Comandos principales finales, no devuelven nada y terminan la cadena de comandos:

```
Cypress.Commands.add('clickLink', (label) => {  
  cy.get('a').contains(label).click().end()  
})  
cy.clickLink('Leer mas ...')
```
- Comandos principales, no devuelven nada pero dejan que la cadena pueda continuar:

```
Cypress.Commands.add('clickButton', (label) => {  
  cy.get('button').contains(label).click()  
})
```

© JMA 2020. All rights reserved

Crear comandos

- Comandos secundarios, que reciben y devuelven el sujeto para que la cadena pueda continuar:

```
Cypress.Commands.add('console', { prevSubject: true, }, (subject, method) => {  
  console[method || 'log']('The subject is', subject)  
  return subject  
})
```
- La opción `prevSubject` acepta los valores siguientes:
 - `false`: ignora cualquier sujeto anterior (comando principal)
 - `true`: recibe el sujeto anterior (comando secundario)
 - `optional`: puede iniciar una cadena o usar una cadena existente (comando dual)
 - `element`: requiere que el sujeto anterior sea un elemento DOM
 - `document`: requiere que el sujeto anterior sea el documento
 - `window`: requiere que el sujeto anterior sea la ventana
- Se pueden establecer varios tipos:
`prevSubject: ['optional', 'document', 'element'],`

© JMA 2020. All rights reserved

Sobrescribir comandos existentes

- También se puede modificar el comportamiento de los comandos Cypress existentes. Esto es útil para establecer siempre algunos valores predeterminados para evitar crear otro comando que termine usando el original.

```
Cypress.Commands.overwrite('type', (originalFn, element, text, options) => {  
  if (options && options.secrete) {  
    options.log = false  
    Cypress.log({ $el: element, name: 'type', message:  
      '*'.repeat(text.length), })  
  }  
  return originalFn(element, text, options)  
})
```

```
cy.get('#txtUsuario').type('admin')  
cy.get('#txtPassword').type('P@$w0rd', {secrete: true })
```

© JMA 2020. All rights reserved

Registro de comandos

- Cuando se crea un comando personalizado, se puede controlar cómo aparece y se comporta en el Registro de comandos.
- El comando Cypress.log() permite escribir en el Registro de comandos.
- La opción { log: false } pasada a los comando deshabilita que se muestren por defecto en el registro.
- Cuando se implementa un comando personalizado que utiliza múltiples comandos internos, se debe evitar que aparezcan todos en el registro y controlar mediante programación lo que realmente se desea que aparezca.
- Esto limpiará el Registro de comandos que será mucho más atractivo y comprensible visualmente.

© JMA 2020. All rights reserved

Mejores prácticas

- No conviertas todo en un comando personalizado: Los comandos personalizados funcionan bien cuando necesitas describir un comportamiento deseable en muchas de tus pruebas.
- No compliques demasiado las cosas: No hay razón para agregar un nuevo nivel de complejidad por solo un par de comandos.
- No hagas demasiado con un solo comando: Es mejor que los comandos se puedan componer, siendo concretos, que abarrotarlos demasiado lo que los vuelve inflexibles y requieren muchas opciones para controlar su comportamiento.
- Omite el acceso al interfaz de usuario tanto como sea posible: Los comandos personalizados son una excelente manera de abstraer la configuración, si se pueden utilizar atajos (`cy.request()`) serán mas rápidos y predecibles.
- Escribe sus declaraciones en TypeScript: Describir la firma de los comandos personalizados permite que IntelliSense asista y muestre documentación útil.

© JMA 2020. All rights reserved

Declaraciones TypeScript

- Para que el compilador de TypeScript sepa que hemos agregado comandos personalizados y que Lint o IntelliSense funcione, hay que describir la firma de tipo de los comandos personalizados en el archivo `cypress/support/index.d.ts`.
- Las firmas de tipo deben coincidir con los argumentos que esperan los comandos personalizados.
- Cualquier usuario del comando personalizado apreciará mucho un comentario JSDoc detallado sobre el tipo de método.
- Para incluir las nuevas firmas en los ficheros de especificaciones:

```
/// <reference types="../support" />
```

© JMA 2020. All rights reserved

Declaraciones TypeScript

```
declare namespace Cypress {  
  interface Chainable {  
    clickLink(label: string): Chainable<Element>  
    /**  
     * Mostrar el sujeto en la consola de depuración  
     * @param method Permite cambiar el método de consola utilizado, log por defecto  
     * @example cy.console('info')  
     */  
    console(method: string): Chainable<Element>  
  }  
}
```

© JMA 2020. All rights reserved

Plugins

- Los complementos permiten aprovechar, modificar o ampliar el comportamiento interno de Cypress.
- Normalmente, todo el código de prueba, la aplicación y los comandos de Cypress se ejecutan en el navegador. Pero Cypress también es un proceso de Node que los complementos pueden usar.
- Los complementos permiten acceder al proceso de Node que se ejecuta fuera del navegador y son una vía para que código personalizado se ejecute durante etapas particulares del ciclo de vida de Cypress.
- Esto permite hacer cosas como:
 - Utilizar varios entornos con sus propias configuraciones y cambiar las variables de entorno
 - Utilizar preprocesadores
 - Preparar ficheros y recursos que se utilizaran en las pruebas
 - Operaciones de limpieza después de la ejecución de las pruebas
- Cypress mantiene una lista seleccionada de complementos de creación propia y por la comunidad:
 - <https://docs.cypress.io/plugins/directory>

© JMA 2020. All rights reserved

Plugins

- Los complementos de la lista oficial son módulos npm. Esto les permite ser versionados y actualizados por separado sin necesidad de actualizar Cypress.
- Se puede instalar cualquier complemento publicado usando npm:
`npm install <plugin name> --save-dev`
- En `cypress/plugins/index.js` se debe registrar el complemento según indique el desarrollador del mismo:

```
module.exports = (on, config) => {  
  on('<event>', (arg1, arg2) => {  
    // plugin stuff here  
  })  
}
```

© JMA 2020. All rights reserved

Eventos de Cypress

- Los eventos `before:run` y `after:run` ocurren antes y después de una ejecución, respectivamente.
- Los eventos `before:spec` y `after:spec` se ejecutan antes y después de que se ejecute una sola especificación, respectivamente.
- El evento `before:browser:launch` se puede utilizar para modificar los argumentos de lanzamiento para cada navegador en particular.
- El evento `after:screenshot` se llama después de que se toma una captura de pantalla y se guarda en el disco.
- El evento `file:preprocessor` se produce cuando una especificación o un archivo relacionado con la especificación debe transpilarse para el navegador.
- El evento `task`, que se usa junto con el comando `cy.task()`, permite escribir código arbitrario en Node para realizar tareas que no son posibles en el navegador.

© JMA 2020. All rights reserved

cy.task

- Se puedes usar el evento task para hacer cosas como:
 - Manipular una base de datos (lectura, escritura, etc.)
 - Almacenamiento en Node del estado para que persista
 - Ejecutar un proceso externo
 - Realización de tareas paralelas (como realizar múltiples solicitudes http fuera de Cypress)

```
on("task", {
  async "db:seed"() { // seed database with test data
    const { data } = await axios.post(`${testDataApiEndpoint}/seed`);
    return data;
  }, "filter:database"(queryPayload) { // fetch test data from a database
    return queryDatabase(queryPayload, (data, attrs) => _.filter(data.results, attrs));
  }, "find:database"(queryPayload) {
    return queryDatabase(queryPayload, (data, attrs) => _.find(data.results, attrs));
  } });
```

© JMA 2020. All rights reserved

EVENTOS

© JMA 2020. All rights reserved

Eventos

- Cypress emite una serie de eventos mientras se ejecuta en el navegador. Estos eventos son útiles para controlar el comportamiento de la aplicación y con fines de depuración.
- Hay dos tipos de eventos:
 - Eventos de la aplicación: provienen de la aplicación que se está probando actualmente.
 - Eventos de Cypress: provienen de Cypress, ya que emite comandos y reacciona a su estado.
- Hay una gran cantidad de otros eventos que Cypress que son estrictamente internos a la forma en que funciona Cypress y no son útiles para los usuarios.

© JMA 2020. All rights reserved

Vinculación a eventos

- Cypress es un objeto global que persiste durante la totalidad de las pruebas. Cualquier evento que se vincule a Cypress se aplicará a todas las pruebas y no se desvinculará a menos que se los desvincule manualmente. Esto es útil cuando está depurando y se desea agregar un solo evento "catch-all" para rastrear pruebas fallidas o excepciones no detectadas de la aplicación.
- El objeto `cy` está vinculado a cada prueba individual. Los eventos vinculados a `cy` se eliminan automáticamente cuando finaliza la prueba. No es necesario desvincularlos.
- Para vincular y desvincular los controladores de eventos disponen de las versiones jQuery:
 - `on`
 - `once`
 - `removeListener`
 - `removeAllListeners`

© JMA 2020. All rights reserved

Eventos de la aplicación

- `uncaught:exception` - Se activa cuando se produce una excepción no capturada por la aplicación. La prueba fallará cuando se produzca la excepción no capturada salvo que el controlador del evento devuelva `false` y así ya no fallará la prueba. También es útil para fines de depuración porque se proporciona la instancia real del error.
- `window:confirm` - Se activa cuando la aplicación llama al método global `window.confirm()`. Cypress acepta las confirmaciones automáticamente. Si el evento devuelve `false` y la confirmación será cancelada.
- `window:alert` - Se activa cuando la aplicación llama al método global `window.alert()`. Cypress acepta automáticamente las alertas. No puede cambiar este comportamiento pero permite hacer afirmaciones.
- `url:changed` - Se activa cada vez que Cypress detecta que la URL de la aplicación ha cambiado.

© JMA 2020. All rights reserved

Eventos de la aplicación

- `window:before:load` - Se activa cuando la página comienza a cargarse, pero antes de que se ejecute JavaScript de ninguna de las aplicaciones. Esto se activa exactamente al mismo tiempo que la devolución de llamada `onBeforeLoad` en `cy.visit()`. Útil para modificar la ventana en una transición de página.
- `window:load` - Se activa después de que todos sus recursos hayan terminado de cargarse después de una transición de página. Esto se activa exactamente al mismo tiempo que la devolución de llamada `onLoad` en `cy.visit()`.
- `window:before:unload` - Se activa cuando la aplicación está a punto de desaparecer. Se proporciona el objeto del evento real. Es posible que la aplicación haya configurado un evento `returnValue`, lo cual es útil para hacer afirmaciones.
- `window:unload` - Se activa cuando la aplicación se ha descargado y sigue navegando. Se le proporciona el objeto del evento real. Este evento no se puede cancelar.

© JMA 2020. All rights reserved

Eventos de la aplicación

```
it('Excepciones no controladas', (done) => {  
  cy.on('uncaught:exception', (err, runnable) => {  
    if (err.message.includes('Excepción no controlada'))  
      return false;  
    done(err)  
  })  
  cy.visit('/alertas')  
  cy.get('#btnExcepcion').click()  
  cy.get('#btnAlert').click()  
  cy.wrap({}).then(obj => done())  
})
```

© JMA 2020. All rights reserved

Eventos de la aplicación

```
it('Selecciona resultado de confirms', () => {  
  let rslt = true  
  cy.on('window:confirm', (str) => { return rslt; })  
  cy.visit('/alertas')  
  cy.get('#btnConfirm')  
    .then((s) => { rslt = false; return s; })  
    .click()  
  cy.get('#txtResultado').should('be.text', 'Respuesta negativa')  
  cy.get('#btnConfirm')  
    .then((s) => { rslt = true; return s; })  
    .click()  
  cy.get('#txtResultado').should('be.text', 'Respuesta positiva')  
})
```

© JMA 2020. All rights reserved

Eventos de Cypress

test:before:run - Se dispara antes de la prueba y de todos los **before** y **beforeEach**.

test:after:run - Se dispara después de la prueba y de todos los **afterEach** y **after**.

fail - Se dispara cuando la prueba ha fallado. Es técnicamente posible evitar que la prueba falle realmente enlazándose a este evento e invocando una devolución de llamada asíncrona **done**, aunque se se desaconseja enérgicamente. Este evento existe porque es extremadamente útil para fines de depuración.

viewport:changed - Se dispara cada vez que la ventana gráfica cambia a través de **cy.viewport()** o cuando se restablece la ventana gráfica a su valor predeterminado entre pruebas. Útil para fines de depuración.

scrolled - Se dispara siempre que Cypress se desplaza por la aplicación. Este evento se activa cuando Cypress está esperando y calculando la capacidad de acción. Se desplazará para 'descubrir' los elementos que están cubiertos actualmente. Este evento es extremadamente útil para depurar por qué Cypress puede pensar que un elemento no es interactivo.

© JMA 2020. All rights reserved

Eventos de Cypress

command:enqueued - Se activa cuando un comando **cy** se invoca por primera vez y se pone en cola para ejecutarse más tarde. Útil para fines de depuración si no se está seguro del orden en que se ejecutarán los comandos.

command:start - Se dispara cuando **cy** comienza a ejecutar el comando. Útil para depurar y comprender cómo la cola de comandos es asíncrona.

command:end - Se dispara cuando **cy** termina de ejecutar el comando. Útil para depurar y comprender cómo se manejan los comandos.

command:retry - Se activa cada vez que un comando comienza sus rutinas de reintento. Se llama después de que Cypress haya esperado internamente el intervalo de reintento. Es útil para comprender por qué se está reintentando un comando y, por lo general, incluye el error real que provoca el reintento. Cuando los comandos fallan, el error final es el que realmente aparece para la prueba fallida. Este evento es esencialmente para depurar por qué Cypress está fallando.

log:added - Se activa cada vez que un comando emite este evento para que pueda mostrarse en el Registro de comandos. Útil para ver cómo los comandos internos de Cypress utilizan la API **Cypress.log()**.

log:changed - Se dispara cuando cambian los atributos de un comando. Este evento tiene una función antirrebote para evitar que se dispare demasiado rápido y con demasiada frecuencia. Útil para ver cómo los comandos internos de Cypress utilizan la API **Cypress.log()**.

© JMA 2020. All rights reserved

Eventos

```
it('Capturar fail', () => {  
  Cypress.once('fail', (error, runnable) => {  
    debugger  
    throw error  
  })  
  cy.get('element-that-does-not-exist')  
})  
it('Ignora primer fallo', () => {  
  Cypress.once('fail', (error, runnable) => {  
    return false  
  })  
  cy.get('element-that-does-not-exist')  
})
```

© JMA 2020. All rights reserved

Suites

- Una “suite” es un nombre que describe al género o sección que 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.
- El segundo parámetro 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.

© JMA 2020. All rights reserved

Especificaciones

- Una especificación contiene una o más expectativas (algo que se espera) que ponen a prueba el estado del código. Una expectativa de Cypress 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 **it** que, al igual que describe, recibe una cadena y una función, opcionalmente puede recibir un timeout. La cadena es el título de la especificación y la función es la especificación o prueba.
 - `it("y así es una especificación", function() {`
 - `//...`
 - `});`
- **describe** y **it** 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.

© JMA 2020. All rights reserved

Elementos de consulta

- Cypress aprovecha el potente motor de selección de jQuery para ayudar a que las pruebas sean familiares y legibles para los desarrolladores web modernos.
- Sin embargo, acceder a los elementos DOM devueltos por la consulta funciona de manera diferente: Cypress reintenta automáticamente la consulta hasta que encuentra el elemento o se alcanza el tiempo de espera establecido.

© JMA 2020. All rights reserved