Automatización de pruebas funcionales

Servicio Andaluz de Salud
Consejería de Sanidad, Presidencia y Emergencias

Contenido

Resumen

Automatización de pruebas funcionales

Versión: v02r01

Estado: ACTIVO

Entrada en vigor desde: 18 nov 2025

Obligado cumplimiento desde: 26 ene 2026

Automatización de pruebas funcionales PRE-RELEASE

Versión: N/A

Estado:

Entrada en vigor: N/A

Enlaces relacionados:

Cumplimiento normativo

Las normas expuestas son de obligado cumplimiento. La STIC podrá estudiar los casos excepcionales los cuales serán gestionados a través de los responsables del proyecto correspondiente y autorizados por el Área de Gobernanza de la STIC. Asimismo cualquier aspecto no recogido en estas normas deberá regirse en primera instancia por las guías técnicas correspondientes al esquema nacional de seguridad y esquema nacional de interoperabilidad según correspondencia y en su defecto a los marcos normativos y de desarrollo software establecidos por la Junta de Andalucía, debiendo ser puesto de manifiesto ante la STIC.

La STIC se reserva el derecho a la modificación de la norma sin previo aviso, tras lo cual, notificará del cambio a los actores implicados para su adopción inmediata según la planificación de cada proyecto.

En el caso de que algún actor considere conveniente y/o necesario el incumplimiento de alguna de las normas y /o recomendaciones, deberá aportar previamente la correspondiente justificación fehaciente documentada de la solución alternativa propuesta, así como toda aquella documentación que le sea requerida por la STIC para proceder a su validación técnica.

Contacto Dpto: **Oficina de Calidad**

Histórico de cambios

Los cambios en la normativa vendrán acompañados de un registro de las modificaciones. De este modo se podrá realizar un seguimiento y consultar su evolución.

Versión:	v01pr06	**Fecha de publicación:**	18 de noviembre de 2025	**Fecha de entrada en vigor:**	18 de noviembre de 2025
Alcance:					

- Se incorpora y actualiza la información relativa a la Directriz 3: La carpeta Scripts y a la Directriz 12: Script de datos, alineandolas con la estructura del repositorio definida en la normativa de entregas.
- Se actualiza Directriz 13: Entrega de los .features, proporcionando un enfoque aplicable tanto a proyectos ágiles como a proyectos en cascada
- Se añade la nueva Directriz 15: Trazabilidad. Etiquetado de las pruebas
- Se elimina la "Directriz 13. Reporte" incluira en la versión normativa anterior, relativa al informe de resultados de la ejecución de las pruebas automatizadas, dado que dicha información ya queda recogida dentro del modelo de desarrollo cascada/agile y en la documentación requerida en cada caso.

Versión:	v01r05	**Fecha de publicación:**	14 de mayo de 2024	**Fecha de entrada en vigor:**	14 de mayo de 2024
Alcance:					

- Se incluye la directriz 13 sobre el informe de resultados de la ejecución de las pruebas automatizadas.
- Se incorpora nuevo apartado "Entregables" que agrupa:
  - Directriz 12. Script de datos
  - Directriz 13. Reporte

Versión:	v01r04	**Fecha de publicación:**	23 de abril de 2024	**Fecha de entrada en vigor:**	23 de abril de 2024
Alcance:					

- Se modifica la introducción.
- Se modifica directriz 1. Se añade la parte de la configuración de los pom.xml del proyecto.
- Se incluye la directriz 5 sobre la configuración de datos sensibles.
- Se renumeran las directrices a partir de la 5.

Versión:	v01r03	**Fecha de publicación:**	10 de febrero de 2023	**Fecha de entrada en vigor:**	10 de febrero de 2023
Alcance:					

- Se actualiza la directriz 10 (Etiquetado), indicando la opcionalidad de su uso.
- Se incluye la directriz 11 sobre los scripts de base de datos.

Versión:	v01r02	Fecha de publicación:	23 de mayo de 2022	Fecha de entrada en vigor:	23 de mayo de 2022
Alcance:					

- Se actualiza directriz 7 para alinear la entrega de los .feature tanto en pruebas manuales como automatizadas

Versión:	v01r01	Fecha de publicación:	4 de junio de 2020	Fecha de entrada en vigor:	6 de julio de 2020
Alcance:					

- Directriz 1: Se actualiza añadiendo ejemplos de cómo deben definirse las propiedades del pom.xml del proyecto (tanto en el caso del pom.xml padre como en el de los hijos).
- Directriz 2: Se modifican las imágenes de las carpetas principales para que sean coherentes con las rutas que aparecen. Se explica la casuística de una aplicación dividida en módulos.
- Directrices 3, 4 y 5: Se incorporan indicaciones para el caso de pruebas funcionales de Servicios Web.
- Directriz 7: Se añade un ejemplo de cómo deben nombrarse los archivos .feature

Introducción

El objetivo de esta normativa es facilitar a los proveedores una serie de pautas para la correcta implementación de las pruebas funcionales automatizadas.

En ella van a aparecer tanto frameworks a utilizar, como estrategias a la hora de la organización del código. En lo relacionado con las herramientas necesarias para llevar a cabo la automatización de las pruebas, el documento se basa en la triada Gherkins-Cucumber-Selenium. Sólo las dos primeras (Gherkins y Cucumber) son de obligado uso. Las pruebas funcionales a automatizar pueden ser de interfaz de usuario o de integración de componentes, por lo que se recomienda la utilización de Selenium cuando proceda. En cualquier caso, el SAS estudiará posibles propuestas alternativas que puedan plantearse mediante su solicitud formal a la Oficina de Calidad.

Las pruebas automatizadas deben estar preparadas para ejecutarse en las instalaciones de la DGSIC. No será suficiente que se puede lanzar de forma local desde el PC de uno de los miembros del equipo de la OCa. La STIC dispone de una infraestructura basada en Selenium Grid sobre la que realiza la ejecución de las pruebas.

Estructura del proyecto

DIRECTRIZ 1: Nombre y ubicación del código

Lo más importante a destacar en este punto es que las pruebas tendrán su propio repositorio, que se llamará *<código ALM de la aplicación en CMS>-testproject*.

Inicialmente, será un proyecto maven multimódulos.

El pom.xml de este proyecto de pruebas tendrá las siguientes propiedades:

> *<groupId>**es.ja.csalud.sas.$path de la aplicación$**</groupId> (en minúsculas) donde **path de la aplicación** == **Ambito.Categoría.CodALMAplicación***
>
> *<artifactId>**$código ALM del aplicativo en la CMS$-testproject**</artifactId> (en minúsculas)*
>
> *<name>**$código ALM del aplicativo en la CMS$-testproject**</name>*

La estructura de carpetas del proyecto quedaría:

> $código ALM del aplicativo en la CMS$-testproject/src/test/java/es/ja/csalud/sas/$path de la aplicación$.

Por ejemplo:

Aplicación: Estación Clínica Unificada - ECU: módulo de front

- ámbito: ASISTENCIAL
- categoría: DIRAYA
- código ALM aplicación CMS: ECUFRONT

> *pom.xml de la carpeta de pruebas*
>
> > *<groupId>**es.ja.csalud.sas.asistencial.diraya.ecufront**</groupId>*
> >
> > *<artifactId>**ecufront-testproject**</artifactId>*
> >
> > *<name>**ecufront-testproject**</name>*
> >
> > *<version>**1.0.0.1**</version>*

Nombre paquete: ecufront-testproject/src/test/java/es/ja/csalud/sas/asistencial/diraya/ecufront/

> **Modelo aplicación-componente**
>
> En el caso de que se adopte un modelo de aplicación-componente, la organización del proyecto de pruebas apenas cambia.
>
> Se organizará como un proyecto Maven multimódulo. Ahora, cada módulo del testproject corresponderá con un componente de la aplicación.

Así, el pom.xml del proyecto no cambiará:

> *<groupId>**es.ja.csalud.sas.$path de la aplicación$**</groupId> (en minúsculas) donde **path de la aplicación == Ambito.Categoría.CodALMAplicación.CodALMComponente***
>
> *<artifactId>**$código ALM del aplicativo en la CMS$-testproject**</artifactId> (en minúsculas)*
>
> *<name>**$código ALM del aplicativo en la CMS$-testproject**</name>*

Los archivos pom.xml de los módulos correspondientes a los diferentes componentes deberán cumplir:

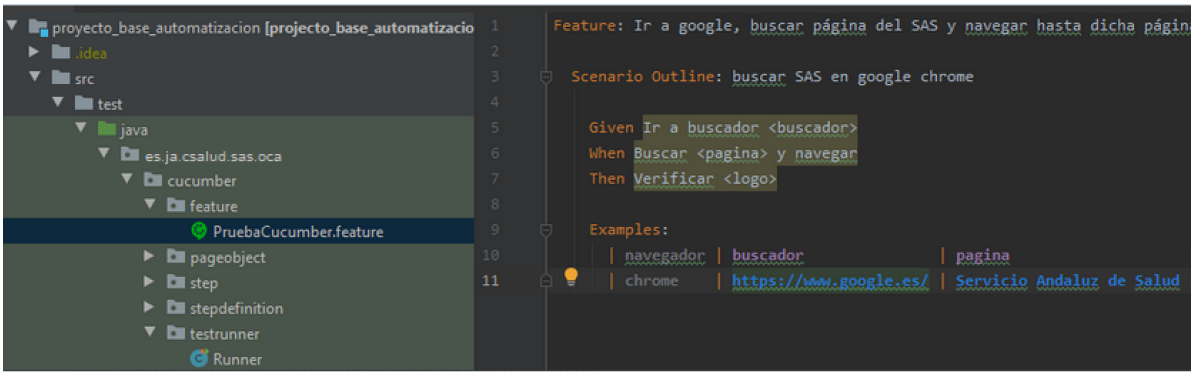
- <artifactId>***$código ALM del componente en la CMS$***</artifactId> (en minúsculas)
- <name>***$código ALM del componente en la CMS$***</name>

DIRECTRIZ 2: Carpetas principales

A continuación se listan las carpetas principales que aparecerán en cada módulo así como su ubicación. Importante recordar que ***path de la aplicación == Ambito.Categoría.CodALMAplicación.CodALMComponente***

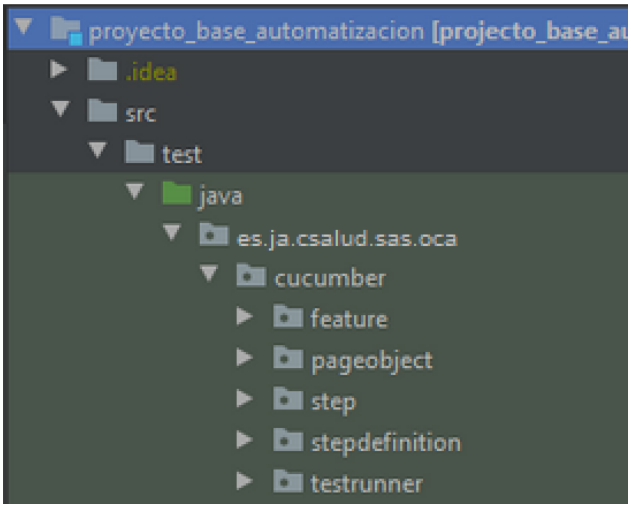
- ***es.ja.csalud.sas.$path de la aplicación$.cucumber.feature***

  Si procede, las pruebas se agruparán por subsistemas funcionales. En ese caso, habrá un directorio por subsitema, que contendrá sus pruebas entregadas en formato .feature.



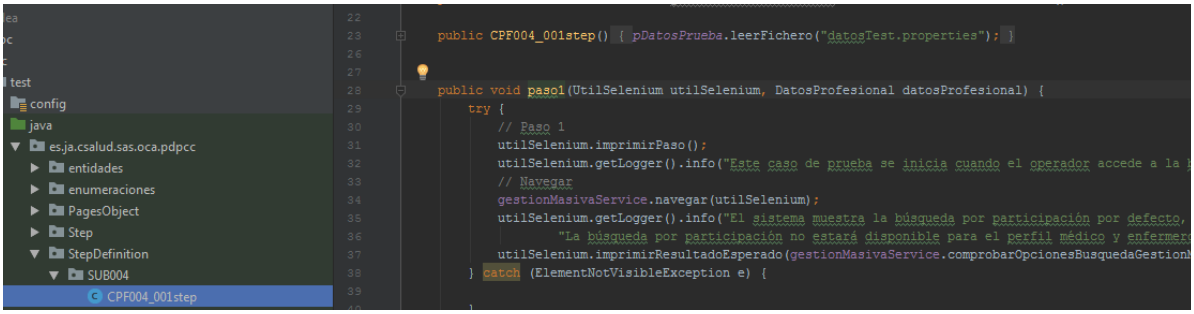
- ***es.ja.csalud.sas.$sistemaInformacion$.cucumber.testrunner***

  Contendrá las clases lanzadoras de las pruebas para Cucumber



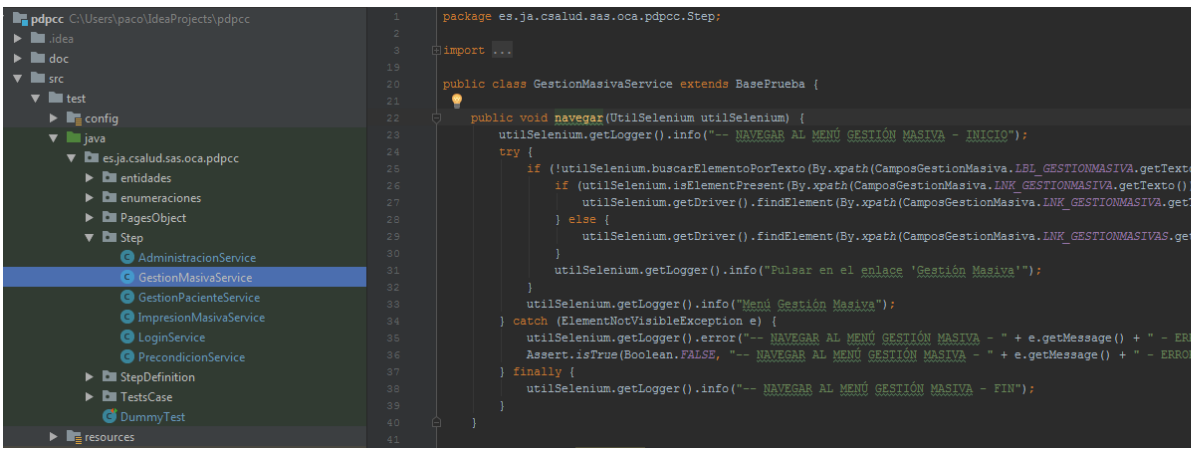
- ***es.ja.csalud.sas.$sistemaInformacion$.cucumber.stepdefinition***

  Este directorio stepdefinition también podrá estar dividido por subsistemas. Tendremos tantas clases como casos de pruebas. Estas clases tendrán instanciada un objeto de la clase caso de prueba con la coletilla step (por ejemplo CP001_001step) y tantos métodos (paso1) como pasos tenga dicho caso de prueba. También, deben tener como mínimo la llamada a los métodos imprimirPaso e imprimirResultadoEsperado de la librería utilidades (consultar el apartado correspondiente) para verificar que se está realizando el paso y cómo ha finalizado.



- ***es.ja.csalud.sas.$sistemaInformacion$.cucumber.step***

  El directorio step contiene las operaciones a bajo nivel. Habrá una clase por cada funcionalidad que tiene la aplicación.



- ***es.ja.csalud.sas.$sistemaInformacion$.cucumber.pageobject***

  Las clases que contendrán el DOM de los elementos de la aplicación a automatizar, o el patrón Page Object Model de las pruebas de interfaz de usuario.

- ***es.ja.csalud.sas.$sistemaInformacion$.cucumber.serviceobject***

  Las clases que contendrán el DOM de los elementos de la aplicación a automatizar de las pruebas funcionales de servicios web.

- ***es.ja.csalud.sas.$sistemaInformacion$.cucumber.common***

  En el caso de no utilizar la librería "Utilidades" ofrecida por la OCA, es necesario la creación de un paquete "common" en el que se inicialicen los distintos navegadores para los que está certificada la aplicación. Adem ás, deberá contener los métodos que sean reutilizables para todos los elementos, como las esperas (implícitas o explícitas), la generación de capturas de pantalla o la interacción con el log.

- ***es.ja.csalud.sas.$sistemaInformacion$.integración***

  Para el caso en el que se vayan a implementar pruebas de integración, este paquete contendrá los distintos ficheros "mockeados" en el formato que se considere oportuno (JSON, XML, YAML,...). Este directorio está orientado a reunir todo lo necesario para simular llamadas a Web Services.

DIRECTRIZ 3: La carpeta Scripts

El proyecto de pruebas (*<código ALM de la aplicación en CMS>-testproject)* es un proyecto para el desarrollo de pruebas automatizadas. Como cualquier otro proyecto de desarrollo, contará con una carpeta script que tendrá la estructura que se pide en normativa, en el artículo *Gestión de entregas,* en el *ANEXO I: Estructura de la carpeta scripts.*

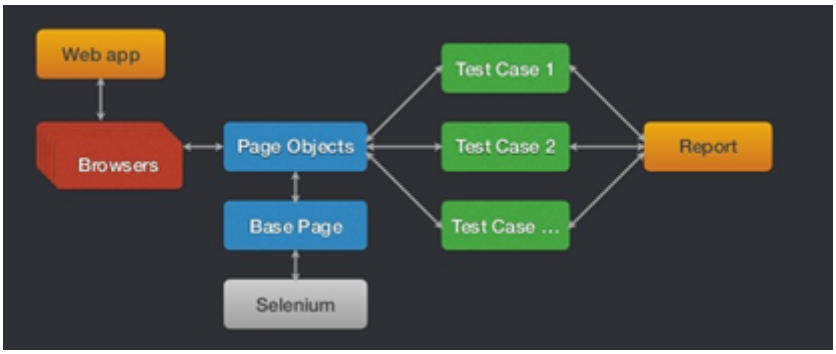
Buenas prácticas

DIRECTRIZ 4: Uso de patrones

Pruebas funcionales de interfaz de usuario (Uso de Page Object Model)

La ejecución automatizada de un conjunto de pruebas funcionales requiere de un modelo de clases que de soporte a su implementación y que sirva de marco para cualquier proyecto desarrollado por y para la STIC. Para ello, el patrón a seguir es “Page Object” que consiste en crear un objeto por cada conjunto de elementos significativos de la interfaz con la que se interactúa. Aunque el nombre sugiera (y generalmente suceda) que cada objeto debe representar una página de la navegación del sistema de información, si dentro de la página existen componentes visuales reutilizables en otras partes, es necesario construir un Page Object de ese elemento para poder reutilizarlo. De esta forma, si se realiza un cambio en la interfaz de usuario, el cambio solo se realiza una única vez, y éste se vea reflejado en todos los test del sistema.

El patrón indicado se sustenta bajo el siguiente modelo:



Por tanto, se encuentran las clases Page Objects donde se ha de implementar los elementos de la interfaz de usuario que permiten interactuar con el sistema de información. Y por otro lado las clases de test con las pruebas a realizar, donde se desarrolla la lógica del test utilizando los elementos descritos en las clases Page Object. Por último, es necesario y fundamental tener presente los distintos navegadores donde se pueden ejecutar los casos de prueba.

Pruebas funcionales de Servicios Web

Para las pruebas funcionales de servicios web, aunque no hay un patrón definido, se sigue una estrategia equivalente a la implementada con Page Object pues se sustituye selenium por el api que utilizamos para lanzar los servicios web y los navegadores por las entradas y salidas de cada uno de los servicios que vamos a probar.

DIRECTRIZ 5: Implementación de buenas prácticas

Pruebas funcionales de interfaz de usuario (utilización de Selenium)

Para la codificación de cada clase page object se utilizarán los métodos proporcionados por Selenium para interactuar con los diferentes elementos de la página web, como por ejemplo:

- Para la identificación de los elementos, sobre los que se interactúa en la prueba, es necesario la utilización de un ***id único en la etiqueta de cada componente***.

  Ejemplo:

  * *

  * *

  * *

  ...

- Hacer click en un botón o enlace *elemento.click();*
- Escribir información en un campo de texto *elemento.sendKeys("Texto a introducir");*
- Recuperar el texto de un elemento: *elemento.getText();*
- Verificar si el elemento se encuentra disponible en la página *inputSearch.isDisplayed();*

Pruebas funcionales de Servicios Web

La codificación de pruebas funcionales de Servicios Web se realizará en base al API que utilice el proyecto para realizar dichas pruebas. Partimos de la premisa de que cada uno de los test debe de tener definidas los siguientes elementos.

- Cabecera http necesaria
- login del servicio si tuviese
- Cuerpo del mensaje de entrada bien sea JSON o XML
- Cuerpo del mensaje de salida para tratar en la prueba.
- Posibles mensajes de error.

DIRECTRIZ 6: Los datos sensibles

Las configuraciones necesarias para la correcta ejecución de las pruebas, que pudieran contener información sensible (tipo usuarios/contraseñas, conexión a la base de datos, etc.) no podrán estar expuestas en archivos tales como el pom. xml o cualquier otro .properties que necesite ser almacenado junto con el código. Deberán ser archivos externos o definirse como variables de entorno, compatibles con Selenium Grid. De esta forma, se deberá entregar tanto el código como el listado de la variables a incluir, cumplimentando el apartado correspondiente en el readme.md.

DIRECTRIZ 7: Uso de framework y librerías

Para la codificación de las pruebas automatizadas se recomienda el uso del siguiente conjunto de herramientas:

- **JUnit**: *Framework* para la ejecución de test unitarios y de integración.

- **TestNG**: *Framework basado en JUnit pero que cubre el espectro completo de pruebas: unitarias, integración, funcionales, end-to-end.*
- **Selenium**: *Framework* para la ejecución de scripts de pruebas sobre un navegador web.
- **Pruebas de Servicios web:** Utilización de framework que realicen las pruebas funcionales de Servicios Web, como puede ser SoapUI
- **LogBack**: *Framework* para la generación de trazas. Teniendo que guardar en ellas los pasos del proceso de cada prueba.
- **Librería de Utilidades (Solo interfaz de usuario):** Se recomienda el uso de una librería de ***Utilidades*** que ha sido desarrollada por el equipo de la Oficina de Calidad y que contiene las acciones básicas para inicializar el driver, escribir el log, etc. A continuación se describe cómo hacer buen uso de ella, así como alguna de sus funcionalidades.
    - Se tendrá que instanciar un objeto de tipo UtilSelenium. La clase que lo instancie hereda de la clase BasePrueba de la librería. Para instanciar dicho objeto se utiliza el método getInstancia el cual recibe tres parámetros.
        - Nombre del log
        - Navegador a utilizar en la prueba
        - Versión del Selenium WebDriver a utilizar
    - imprimirPaso: Utilizado para escribir en el log el paso X (autoincremental) y la información que se pasa como texto como resultado de la realización de dicho paso. Recibe un parámetro String con la descripción del paso a realizar.
    - getLogger: Embebe las utilidades del LogBack dentro de la librería.
    - getDriver: Embebe las utilidades del WebDriver dentro de la librería.
    - imprimirResultadoEsperado: Comprueba el parámetro que indica si la prueba ha ido bien con un Assert y escribe en el log dicho resultado. Este método recibe un parámetro Booleano que indica si la prueba ha ido bien o no.
    - cerrarDriver finaliza la instancia del objeto UtilSelenium.

```
utilSelenium = UtilSelenium.getInstancia("PruebasSeleniumTest", "chrome",
utilSelenium.imprimirPaso("Acceder a Google");
utilSelenium.getLogger().info("URL acceso a Google: https://www.google.es
utilSelenium.getDriver().navigate().to("https://www.google.es");
utilSelenium.getLogger().info("Cerrar navegador");
utilSelenium.imprimirResultadoEsperado(Boolean.TRUE);
utilSelenium.cerrarDriver();
```

Para hacer uso de la librería, es necesario incorporarla como dependencia en el pom.xml:

```
<dependency>
        <groupId>es.ja.csalud.sas.oca</groupId>
        <artifactId>utilidades</artifactId>
        <version>1.3141.26</version>
</dependency>
```

Redacción de pruebas

DIRECTRIZ 8: Uso de Gherkin

Cada caso de prueba deberá contener definido el escenario de la prueba a realizar en lenguaje GHERKIN pudiendo elegir entre los tipos “Scenario” o “Scenario outline” (caso de prueba parametrizado mediante una tabla).

Dentro de las *features* solo debe incluir información relativa al negocio, y nunca aspectos técnicos de configuración; estos irían dentro de *profiles* o directamente en los *pipelines* del Jenkins.

A continuación se indica un ejemplo de para la descripción de un caso de prueba utilizando Gherkin:

Scenario: descripción del escenario a ejecutar

Given Cumplo una pre condición

[And]

When [Ejecuto una acción]

Then [Observo este resultado]

But [No debería poder observar este otro resultado]

Scenario outline: descripción del escenario parametrizable a ejecutar

Given Cumplo una precondición con <*input_1*>

[And] y cumplo con <*input_2*>

When Ejecuto una acción sobre <*button*>

Then Observo este resultado <*output*>

But No debería poder observar este otro resultado

Examples:

| *input_1* | *input_2* | *button* | *output* |

| valor1 | valor2 | etiqueta | salida |

Codificación de pruebas

DIRECTRIZ 9: Codificación de tests unitarios

Para la ejecución de las pruebas dentro del flujo de integración continua, es necesario crear las clases de tipo TestNG. Para la codificación de cada clase TestNG se utilizarán mínimo las siguientes anotaciones:

- **@DataProvider**: Es un método para suministrar los datos necesarios al @test. Este método debe devolver un Object[x][y] que corresponde con los distintos test case que van a ejecutarse en el @test.
- **@Parameters**: Describe cómo pasar los parámetros al método @Test.
- **@Test**: Método para ejecutar el test. Es necesario pasarle por parámetros el DataProvider.

Por otro lado, podrían ser de utilidad las siguientes:

- **@BeforeSuite** - **@AfterSuite**: Este método se ejecutará antes - después de lanzar todos los test suite.
- **@BeforeTest** - **@AfterTest**: Este método se ejecutará antes - después de lanzar los métodos que pertenecen a las clases <test>.
- **@BeforeClass** - **@AfterClass**: Este método se ejecutará antes - después del método @test de la propia clase.
- **@BeforeMethod** - **@AfterMethod**: Este método se ejecutará antes - después de cada método @test.

Aquellos tests en los que no sean necesario incluir su ejecución por el momento se pueden excluir de la ejecución global indicándolo en el XML Suite de la siguiente forma:

testng.xml

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Suite" verbose="1" parallel="classes" thread-count="5">

        <parameter name="browse" value="firefox"/>
        <!-- "ie"// "firefox" //"chrome" -->

    <test name="Test">
        <classes>
            <class name="src.test.java.es.ja.csalud.sas.$sistemInformacion$.funcionales.pageObject.claseTestNG">
                <methods>
                    <exclude name="test"></exclude> <!-- excluir el método que no es necesario ejecutar -->
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

DIRECTRIZ 10: Uso de cucumber

Para la codificación de cada background y de los pasos de cada scenario se utilizarán las siguientes keywords:

- **@Given** ("^we are a user$") definirá el cumplimiento de una precondición.
- **@And** ("^we enter to application$") definirá los pasos de creación del driver de Appium y carga de la aplicación en el dispositivo móvil.

- **@When** ("^we make login with user and password $") definirá los pasos que realizará la prueba.
- **@Then** ("^the login is succesffull $") definirá los pasos para la verificación correcta de la prueba.

Adicionalmente se puede incluir la keyword **@After** para definir los pasos a ejecutar después de la prueba.

Para la ejecución de las pruebas dentro del flujo de integración continua, es necesario crear una clase lanzadora con la siguiente información como mínimo:

```
import orj.junit.runner.RunWith;
import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)
@Cucumber.Options(
        features = "MockApp", //ruta donde se alojan los ficheros .features
        format = {"json:target/cucumber.json"},  //que formatters se usaran
        tags = {"@run"},  //tags para ejecutar/no ejecutar -> include(@)/exclude(@~)
        plugin = {
                "pretty",
                "html:target/cucumber",
                "json:target/cucumber.json"
        }
)
public class LanzadorDePruebas {
}
```

DIRECTRIZ 11: Etiquetado

A continuación se describen un conjunto de tags a usar para la ejecución o no de escenarios durante las pruebas. Los tags son anotaciones que sirven para agrupar y organizar escenarios e incluso features. Se escriben con el símbolo @ seguido de un texto significativo. No son de obligado uso, y el procedimiento correcto es que se consensue con la STIC su utilización.

Ejemplos:

- **tags = {“@SmokeTest”}** Ejecuta todos los escenarios bajo el tag @SmokeTest.
- **tags = {“@gui”}** Ejecuta todos los escenarios bajo el tag @gui, como en el ejemplo tenemos una feature bajo este tag, se ejecutarán todos los escenarios de esa feature.
- **tags = {“@SmokeTest, @wip”}** Ejecuta todos los escenarios que estén bajo el tag @SmokeTest o bajo el tag @wip (condición OR).
- **tags = {“@SmokeTest”, “@RegressionTest”}** Ejecuta todos los escenarios que estén bajo los tags @SmokeTest y @RegressionTest (condición AND).
- **tags = {“~@SmokeTest”}** ignora todos los escenarios bajo el tag @SmokeTest.
- **tags = {“@RegressionTest, ~@SmokeTest”}** ejecuta todos los escenarios bajo el tag @RegressionTest, pero ignora todos los escenarios bajo el tag @SmokeTest.

- **tags = {“@gui“, “~@SmokeTest”, “~@RegressionTest”}** ignora todos los escenarios bajo el tag @SmokeTest y @RegressionTest pero ejecuta todos los que estén bajo el tag “@gui”, si seguimos el ejemplo es como ejecutar todos los escenarios de la feature que no estén bajo ningún otro tag.

Para cada caso de prueba, en función de la criticidad acordada, se etiquetarán los scenarios con los siguientes tags:

- Prioridad Bloqueante: **@SmokeTest.**
- Prioridad Critica: **@RegressionTest.**

Para el resto de prioridades no será necesario asignar ningún tag por defecto.

Entregables

DIRECTRIZ 12: Script de datos

De todos es conocida la importancia de la disposición de datos correctos para la ejecución de las pruebas. Es un aspecto a tener muy en cuenta de cara a minimizar los casos de falsos negativos provocados por la utilización de unos datos inadecuados, y no por un error en la funcionalidad.

Al entregar el código, se entregarán también los scripts de inserción de los datos necesarios para la correcta ejecución de las pruebas, así como sus respectivos scripts de *"undo"*.

Dichos scripts se almacenarán en la carpeta *"scripts"* ubicada en la raíz del propio repositorio testproject, y cuya estructura se ha comentado anteriormente.

Dentro de la carpeta DML, la estructura de la carpeta será análoga a la de la carpeta *"feature"*.

- Se dividirá por subsistemas funcionales.
- El nombre del archivo .sql coincidirá con el nombre del .feature de la prueba.

    Por ejemplo: Si se entrega un archivo *ECU_AdquisicionPaciente.feature* del SUB001, con las pruebas, se deberá entregar un archivo *ECU_AdquisicionPaciente.sql* y un *ECU_AdquisicionPaciente_undo.sql* en la carpeta SUB001 dentro de *"scripts"*.

DIRECTRIZ 13: Entrega de los .features

Todas las pruebas, ya sean de ejecución manual o automatizada, deben registrarse en JIRA. El plugin Xray permite exportar una HU y/o un Caso de Uso (según corresponda), junto con sus pruebas asociadas, a un archivo de extensión . feature. Éste será el archivo que se subirá a Git.

DIRECTRIZ 14: Trazabilidad. Etiquetado de las pruebas

Identificación de las pruebas

Cada prueba deberá ir etiquetada con su ID en Jira.

Este etiquetado permite identificar de forma unívoca cada prueba en Jira. También es un mecanismo que nos permite el filtrado de las pruebas, útil cuando no se desea ejecutar el conjunto completo de pruebas.

Ejemplo:

@DBS-4791
Scenario: TABACO - Prueba insertar registro de no exposición al humo

Asimismo, los escenarios deben estar etiquetados también en los archivos .feature almacenados en Git.

Configuración técnica

Es necesario generar un archivo denominado *"Cucumber.json"*, que contendrá toda la información relativa a la ejecución de las pruebas. Este reporte *"Cucumber.json"* puede generarse por distintos medios, siempre que se respeten las directrices establecidas en esta normativa.

Una forma habitual de generarlo es mediante la **Class Runner**, utilizando la anotación: @CucumberOptions(plugin=" json:...")

Ejemplo de configuración:

> *Nota: Este es un ejemplo orientativo. Las rutas de los archivos .feature, los paquetes de step definitions, los plugins y las etiquetas (tasgs) deberán ajustarse a las necesidades específicas de cada proyecto*

ClassRunner

```
/** Nota:  Este es un ejemplo de código.
La ruta de los archivos .feature (features), los paquetes de step definitions (glue), los plugins, tags
deben ajustarse a las necesidades específicas  de cada proyecto. */

@CucumberOptions(
                features = "classpath:feature",
                glue = {
                                "classpath:es.ja.csalud.sas.mdbs.habitos.funcionales.cucumber.step.
definitions.common",
                                "classpath:es.ja.csalud.sas.mdbs.habitos.funcionales.cucumber.step.
definitions.tabaco",
                                "classpath:es.ja.csalud.sas.mdbs.habitos.funcionales.cucumber.step.
definitions.cuidador",
                                "classpath:es.ja.csalud.sas.mdbs.habitos.funcionales.cucumber.hooks"
                },
                plugin = {
                                "json:../target/surefire-reports/Cucumber.json",
                                "pretty",
```

```
                                "html:../target/surefire-reports/cucumber.html",
                                "es.ja.csalud.sas.mdbs.habitos.funcionales.common.Logs:../target
/surefire-reports"
                },
                tags = "(@EjemploTag)",
                dryRun = false
)
```

Generación del archivo JSON

- La generación del JSON no debe realizarse de forma manual.
- El fichero *Cucumber.json* se generará automáticamente en el directorio: "target/surefire-reports/Cucumber.json"

  En cualquier caso, se proporcionará la ruta exacta de creación del archivo según el entorno o proyecto.