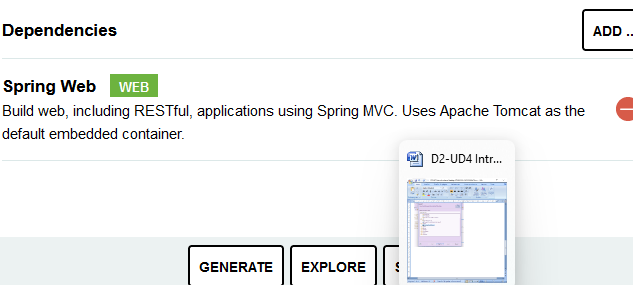
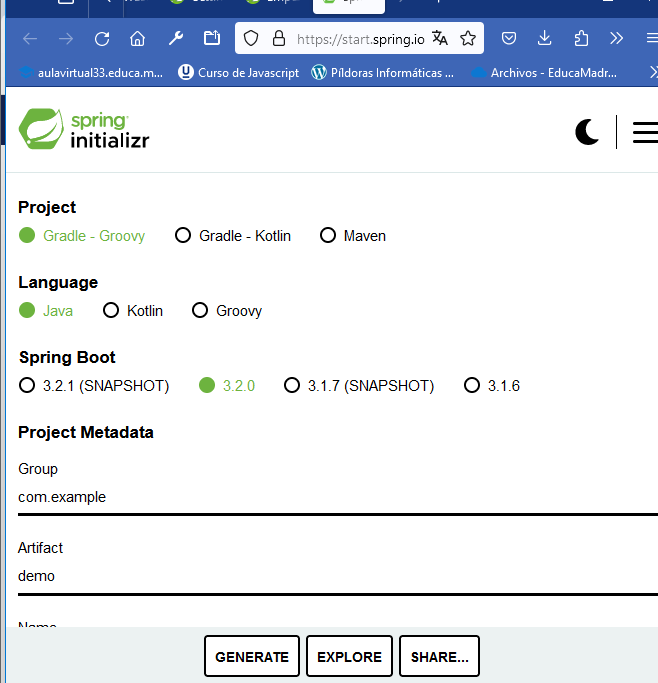
**Tarea 2 – UD4**

**Introducción al testing**

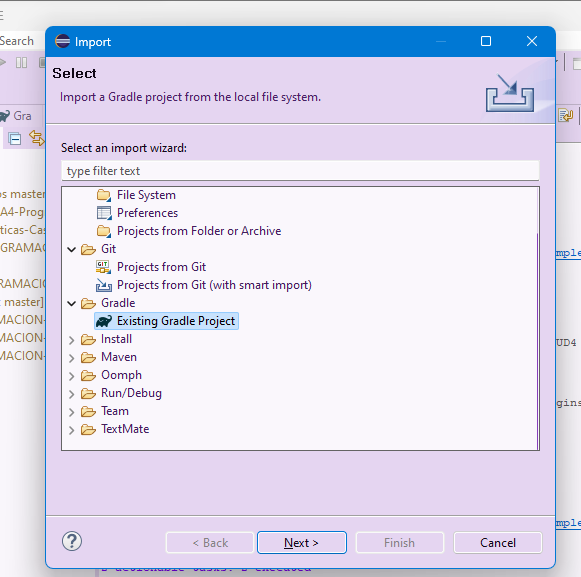
**TUTORIAL:** [**https://spring.io/guides/gs/testing-web/**](https://spring.io/guides/gs/testing-web/)

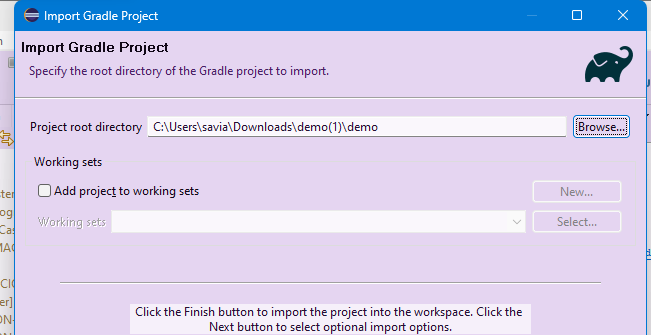
Voy a usar Eclipse.

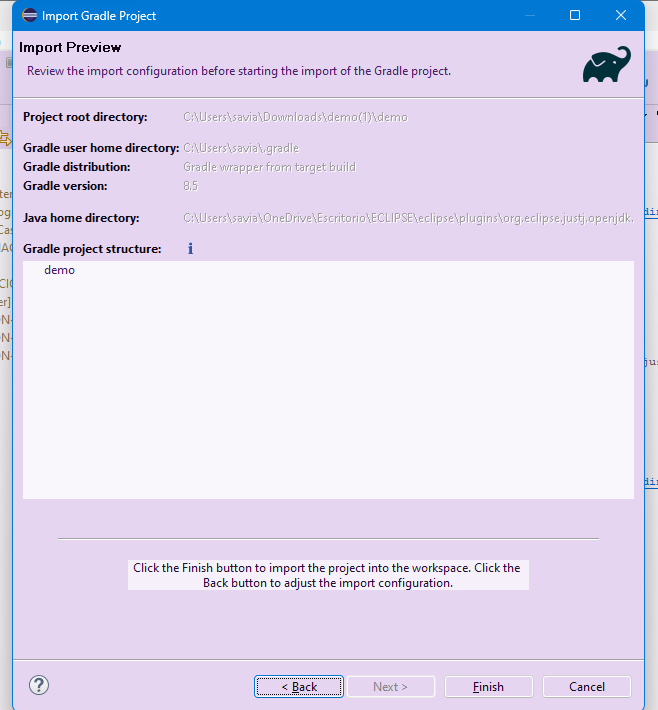
Genero zip de proyecto gradle, seleccionando SpringWeb en dependencias y java como lenguaje:

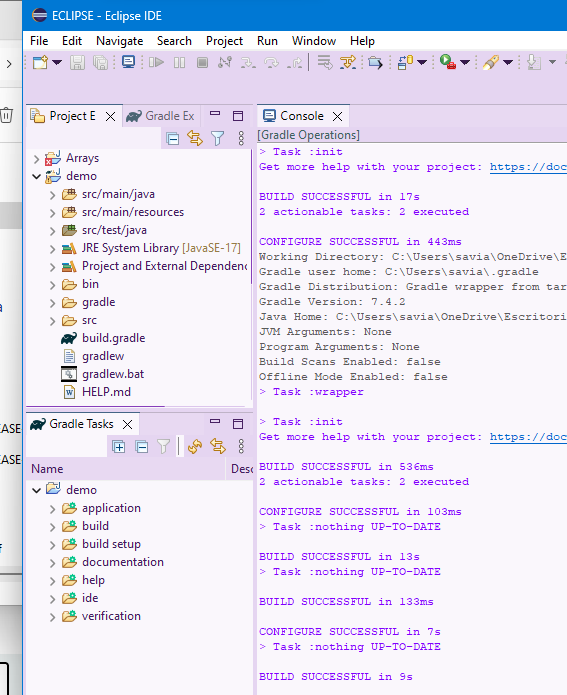


Y lo importo en Eclipse:

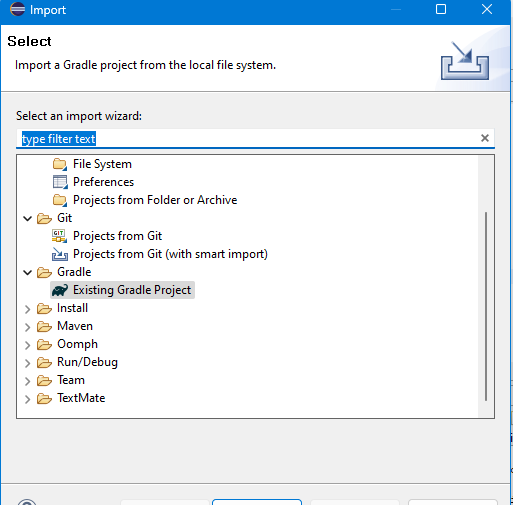


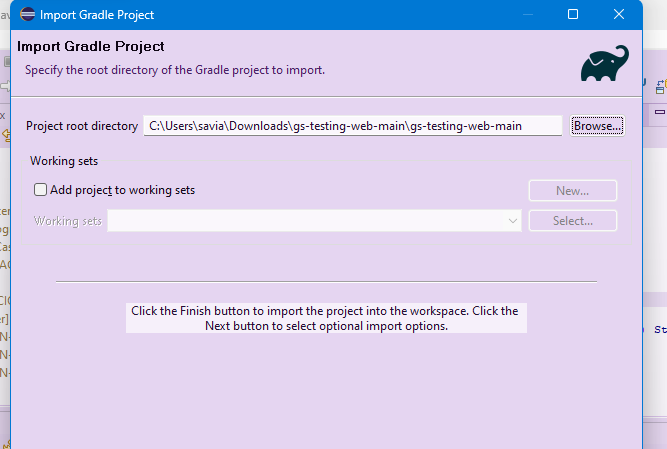




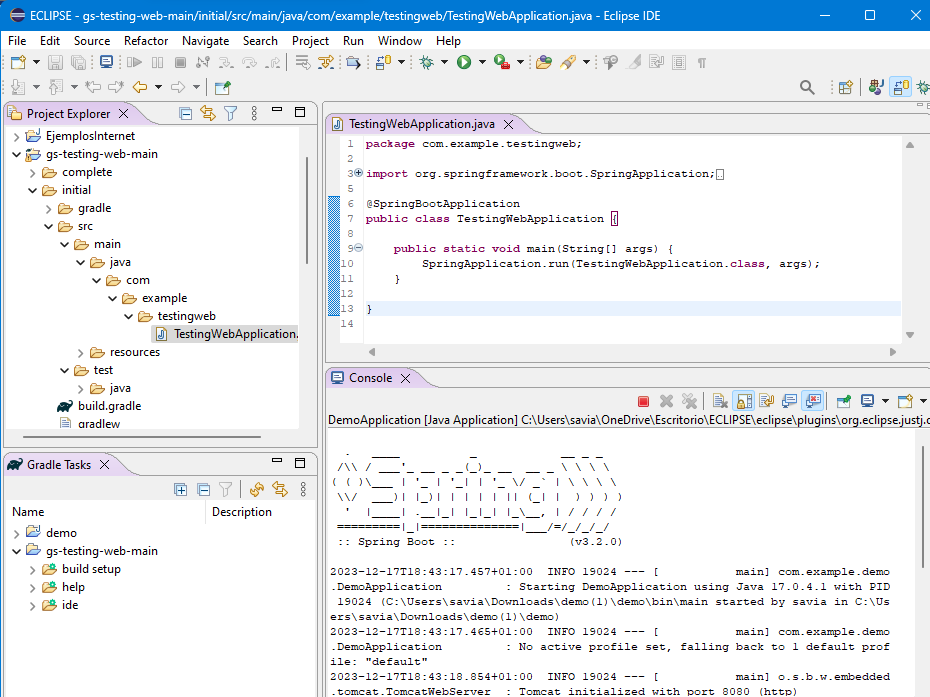


Descargo el repositorio de la guía y lo importo como Gradle Project:



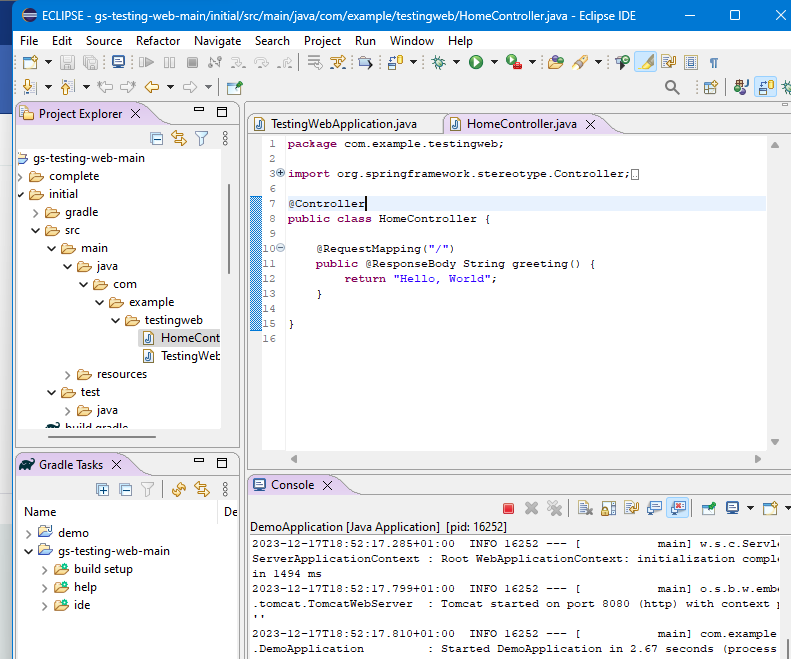


Y comprobamos que se ha importado bien y que se ejecuta:

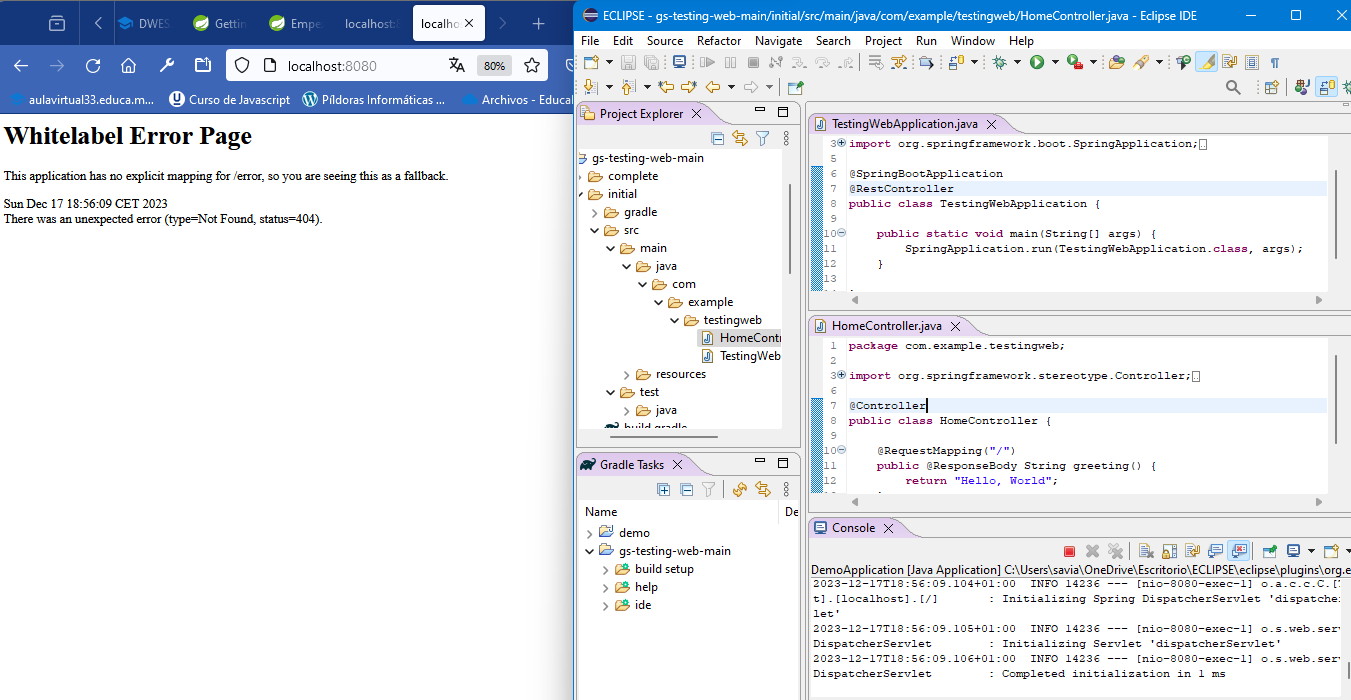


**Crear una aplicación simple:**

Creamos nuevo controlador en la clase HomeController:

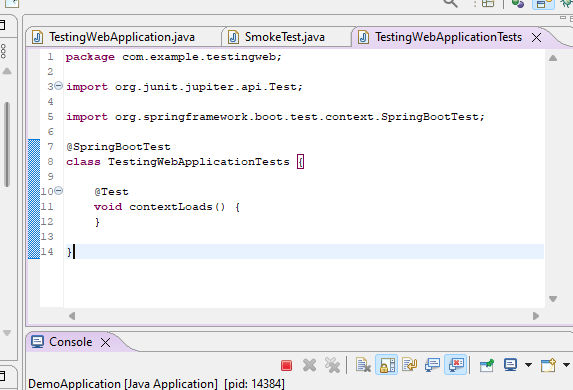


Ahora probamos a carga la página de inicio en el navegador, pero no sale nada, aunque en la consola de eclipse no sale ningún error:



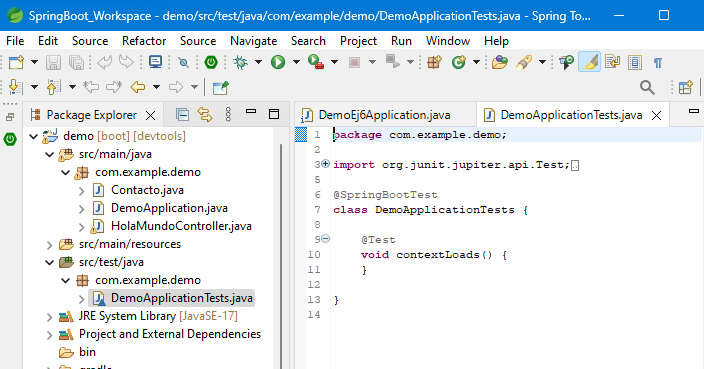
Aún así, sigo con el tutorial:

Voy a automizar la prueba. Escribo una prueba de control en TestingWebApplicationTest.java:



Aquí empecé a tener problemas por lo que voy al paso avanzado usando el código del desafío anterior:

1. **Salta el tutorial hasta el punto “Run the application” pues la parte anterior es lo ya hecho en el desafío anterior.**
2. **Mira si el proyecto ha creado algo en src/test/java. Si estuviera una clase tipo DemoApplicationTests o similar creada automáticamente en tu proyecto. Aprovecha para leer el significado de las anotaciones** **en el tutorial, aunque algunas todavía no las veas.** Así es, hay un archivo, lo abro y veo sus anotaciones:



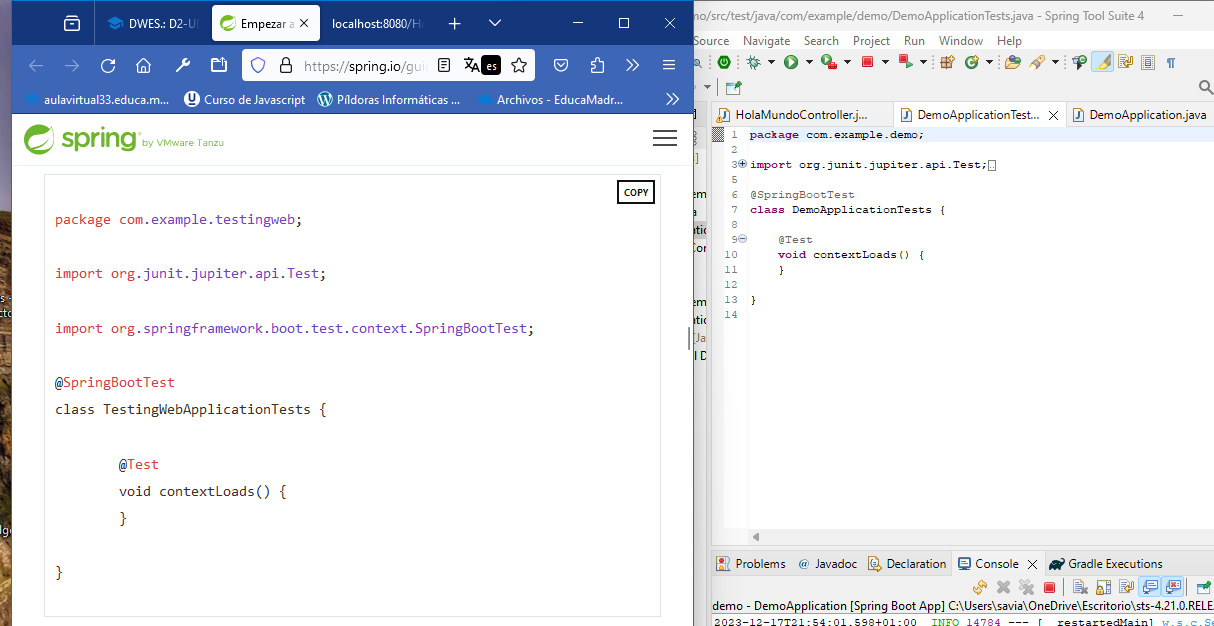
* @SpringBootTest
  + indica que esta clase debe cargar la configuración de la aplicación Spring Boot.
  + Cuando ejecutas pruebas con @SpringBootTest, Spring Boot configura el contexto de la aplicación y carga todas las configuraciones necesarias para ejecutar las pruebas.
* class DemoApplicationTests:
  + Esta clase representa la suite de pruebas para tu aplicación.
  + Las pruebas específicas se escribirán como métodos dentro de esta clase.
* @Test:
  + indica que el método siguiente es una prueba.
  + El nombre del método sugiere que la prueba verifica si el contexto de la aplicación se carga correctamente.
* void contextLoads() { }:
  + cuerpo de la prueba. El método está vacío, lo que significa que no contiene ninguna lógica de prueba específica.
  + La presencia de este método generalmente se debe a que, en la fase de desarrollo inicial, simplemente se verifica que el contexto de la aplicación se cargue correctamente.

Cuando ejecutas esta prueba (puedes hacerlo desde tu IDE o utilizando herramientas de construcción como Maven o Gradle), Spring Boot cargará el contexto de la aplicación y verificará que no haya errores de configuración. La prueba contextLoads suele ser la primera prueba que se ejecuta para asegurarse de que la aplicación se esté configurando correctamente.

Puedes agregar pruebas adicionales a esta clase a medida que desarrollas tu aplicación para verificar el comportamiento específico de los componentes y servicios

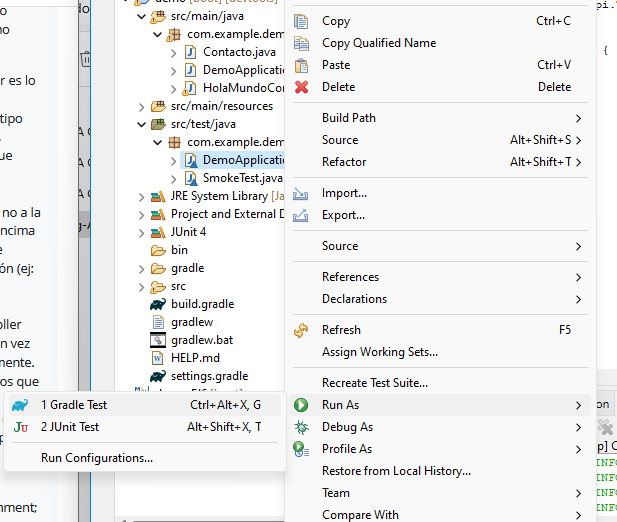
1. **En el apartado “Test the application” compara si el primer código (TestingWebApplicacionTest) a parte del nombre, su contenido es igual o no a la clase del punto anterior o en caso contrario créala.**

Vemos que es igual:

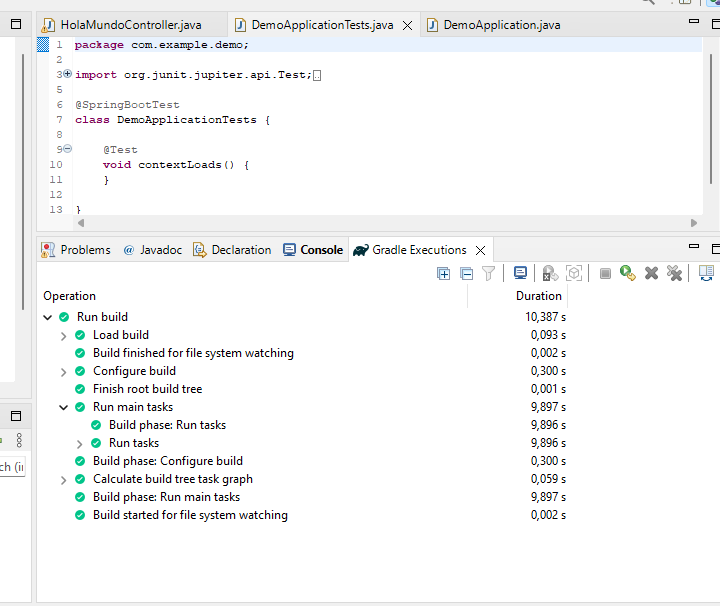


El @SpringBootTestanotación le dice a Spring Boot que busque una clase de configuración principal (una con @SpringBootApplication, por ejemplo) y utilizarlo para iniciar un contexto de aplicación de Spring.

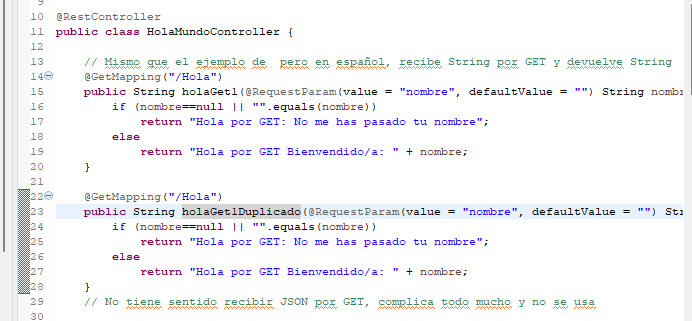
**Con un clic derecho encima de la clase dale a Run → Gradle Test y observa el resultado. Simplemente comprueba que la aplicación se levanta. Introduce un error en la aplicación (ej: duplicar un punto de acceso web) y vuelve a correr el test ¿cambia algo?**



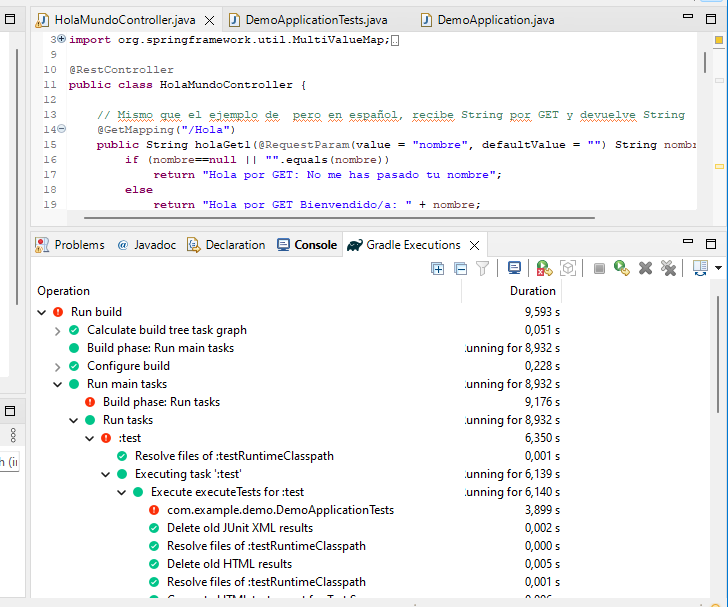
Vemos que todo es correcto, checks verdes:



Ahora vamos a provocar un error en el código, vamos a duplicar el método holaGet1 para así tener dos métodos con la misma ruta:

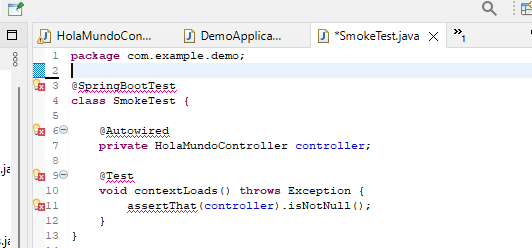


Y al volver a darle a run as Grasle test vemos que hay errores (también salían los errores en la consola):

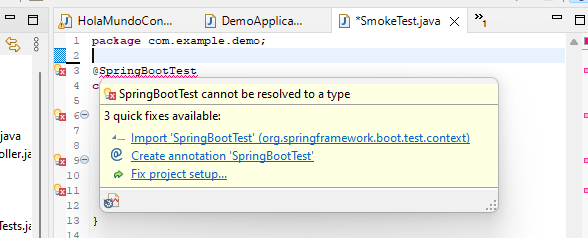


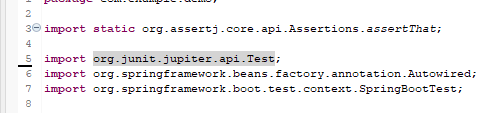
1. **Copia el segundo código de ese apartado del tutorial (SmokeTest) sin los imports a una clase dentro de src/test/java pero cambiando HomeController por HolaMundoController, hace algo similar a lo anterior, pero esta vez en vez de la aplicación comprueba que el controlador se ha levantado correctamente. Pruébalo y documenta el resultado .Documenta las anotaciones y métodos que te han pedido incluir nuevos imports. Analiza el código.**

Vamos a añadir una afirmación creando la clase SmokeTest en src/test/java, sin los imports:

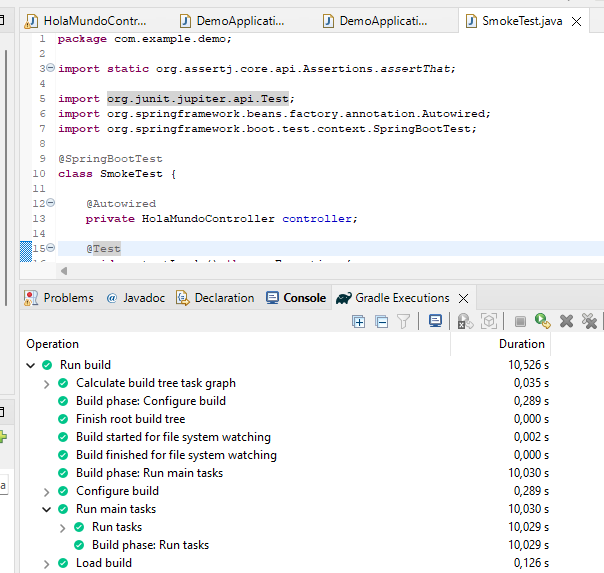


Ahora importamos los imports solicitados para que no salten los errores:





Y probamos con run as gradle test:

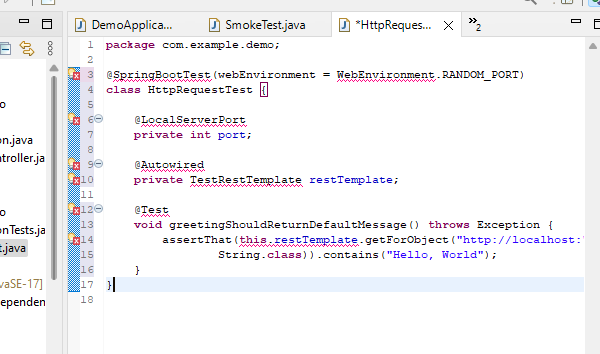


Spring interpreta la anotación @Autowired y el controlador se inyecta antes de ejecutar los métodos de test. No veo diferencias de los resultados con el test del código anterior. Solo que este código de pruebas está destinado a verificar que el contexto de la aplicación HolaMundoController carga/inyecta correctamente y el código de DemoApplicationTests es un test vacío que lo estamos usando por defecto, y en un futuro puede usarse para para agregar pruebas más específicas.  
  
Ahora voy a documentar el código, las anotaciones y métodos:

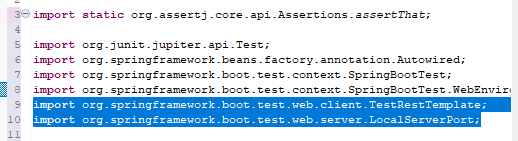
* @**SpringBootTest** anotación: indica que este caso de prueba debe cargar la configuración de la aplicación completa.
* @**Autowired** anotación: se utiliza para inyectar HolaMundoController en el campo **private HolaMundoController controller** lo que significa que Spring Boot se encargará de crear una instancia del controlador y asignarla al campo controller antes de ejecutar el método de prueba.
* @**Test anotación**: indica que el método anotado es un método de prueba. Este método será ejecutado cuando se ejecute la prueba.
* **void contextLoads() throws Exception { ... } método:** método de prueba se que está destinado a verificar la carga del contexto de la aplicación. Está marcado con **throws Exception**, para manejar excepciones que puedan ocurrir durante la ejecución del test.
* **assertThat(controller).isNotNull();:** Utiliza la biblioteca AssertJ para realizar una aserción que verifica que el controlador (controller) no sea nulo. Si el controlador es nulo, la aserción fallará y la prueba se considerará como fallida.

Este código está probando que HolaMundoController se carga correctamente en el contexto de la aplicación de Spring Boot y que no es nulo. Este tipo de prueba se utiliza para verificar rápidamente que la aplicación puede arrancar y que algunos de sus componentes clave están disponibles.

1. **Vamos con el tercer código de ese apartado, HttpRequestTest , de nuevo copia la clase sin los imports (en clase explicaré por qué) Sí necesitarás este import que no es capaz de localizar:**

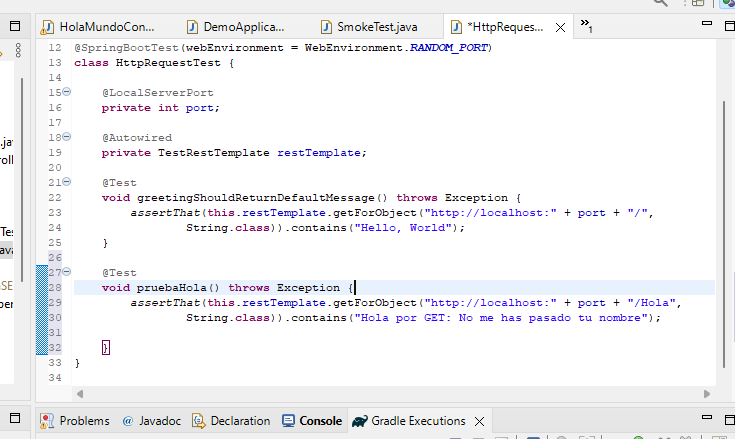
****

E importo los imports:

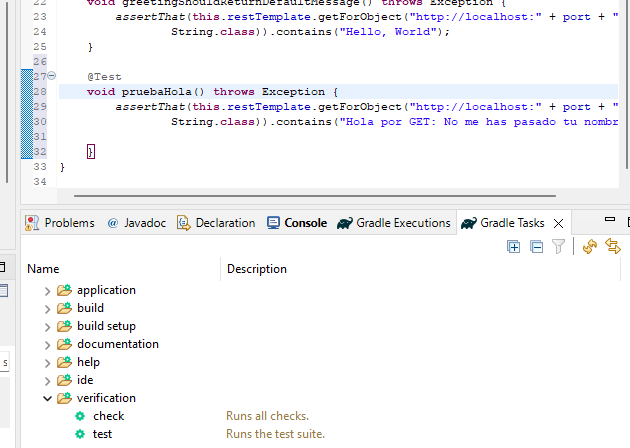


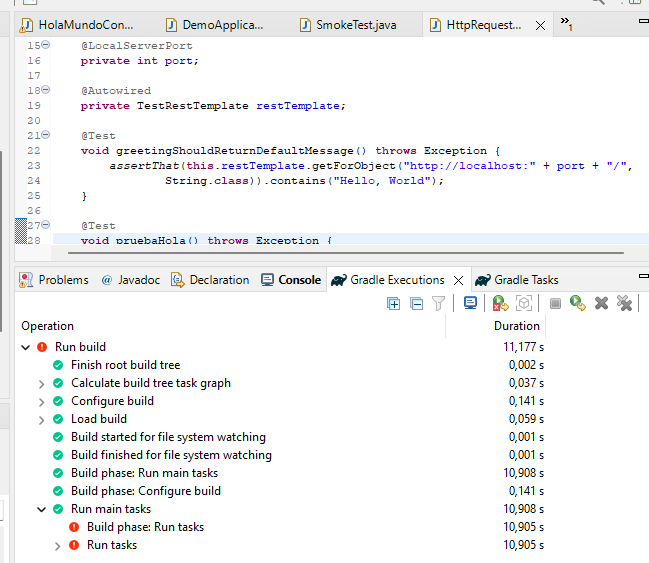
**Añade el punto de acceso (/Hola) y el valor esperado (Hola por GET: No me has pasado tu nombre) y pruébalo con esos datos y con otros para ver cuándo falla. Localiza el fallo ¿se entiende? Prueba con Run → Junit Test ¿mejor?**

Añado el punto de acceso /Hola:

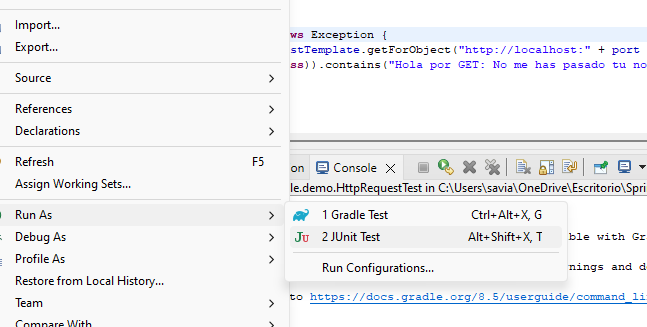


Ejecuto pruebas con Gradle, doblel click en test:

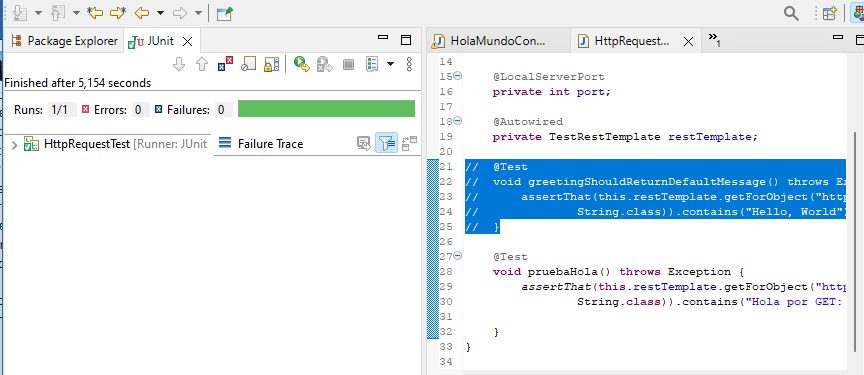
  
Capturo resultados pero no entiendo:



Pruebo con Junit:

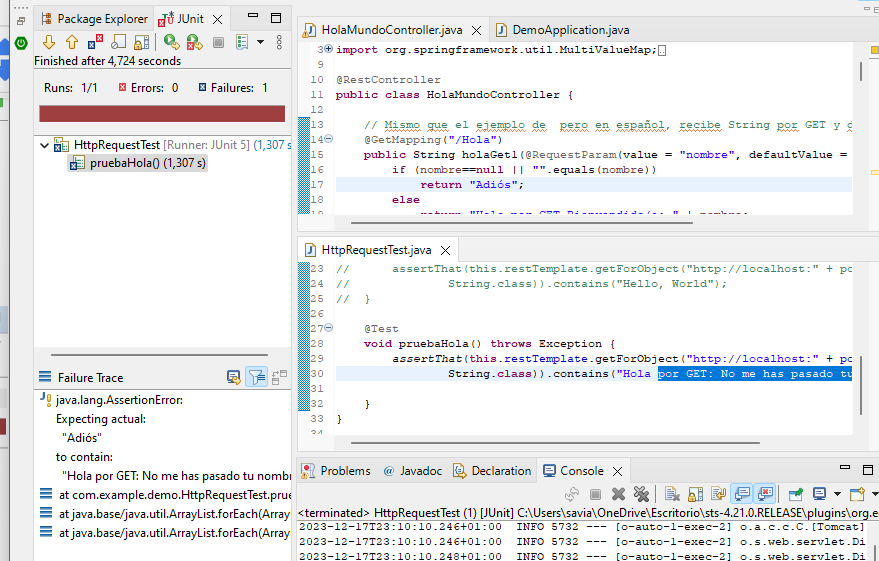


He comentado las líneas que salían en la copia del código, y no sale ningún error, el código es correcto:

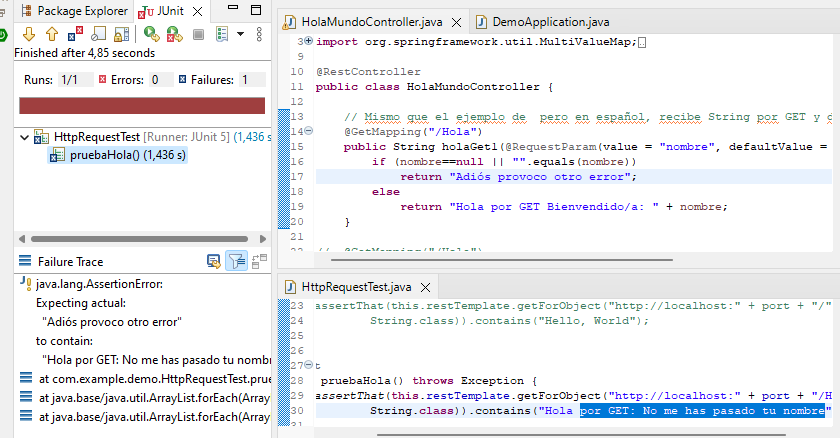


Procedo a realizar pruebas:

- En la clase HolaMundoControler, en el getMappint /Hola, en el primer return ponemos “Adiós” y ejecutamos la prueba nuevamente, vemos el error, ya que no devuelve el valor esperado:

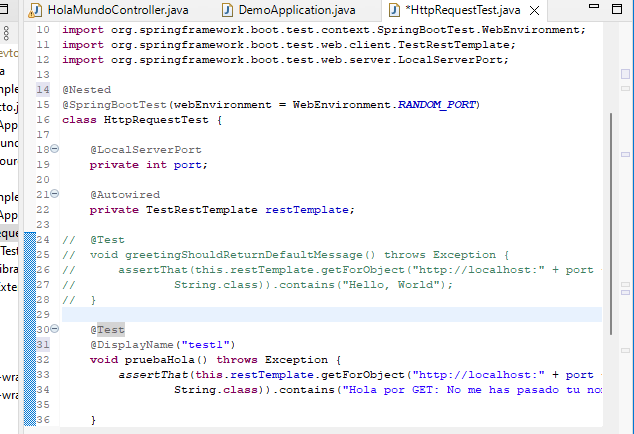


Hacemos otras pruebas:

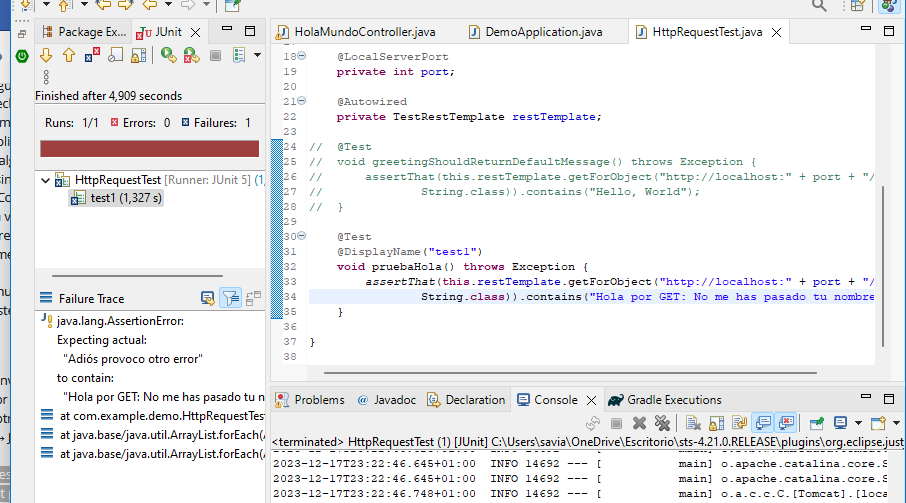


Mucho mejor hacer las pruebas con JUnit, mucho más claro.

**Puedes añadir la anotación @Nested antes de @SpringBootTest y la etiqueta @DisplayName(“Nombre del test”) después de @Test para mejorar la salida del Junit test.**



Y ejecuto Junit Test nuevamente.

Podemos ver ahora el nombre del test, será útil para cuando tengamos varios test:  


Estas dos anotaciones son características que ofrecen funcionalidades específicas para estructurar y personalizar la visualización de tus pruebas.

* @Nested se utiliza para crear clases internas anidadas dentro de una clase de prueba. Esto permite organizar y agrupar pruebas relacionadas de una manera más clara y semántica.
* @DisplayName se utiliza para personalizar el nombre que se mostrará para una clase de prueba o un método de prueba en los informes y en la salida de la consola.