



Spring Webflux: Primeros pasos

por Gunter Rotsaert · Mar. 09, 18 · Zona de Java

Descargue *Microservices for Java Developers* : una introducción práctica a frameworks y contenedores. Presentado en asociación con Red Hat .

Esta publicación será sobre cómo comencé con Spring Webflux. Hace algunas semanas, comencé a leer en detalle acerca de Spring Webflux, Reactive Streams, etc. Mirando hacia atrás en mi viaje, hice las cosas en el orden incorrecto. En esta publicación, intenté estructurar la información para darle un plan para comenzar con Spring Webflux. Los ejemplos se pueden encontrar en GitHub .

Manifiesto reactivo

La base de las corrientes reactivas se puede encontrar en el Manifiesto reactivo . Se recomienda leer este manifiesto, tiene solo 2 páginas, por lo que no tardará mucho en leerlo.

Flujos reactivos

Reactive Streams es una iniciativa para proporcionar un estándar para el procesamiento de flujo asíncrono con contrapresión no bloqueante, como se puede leer en el sitio web. Esto significa que una fuente puede enviar datos a un destino sin sobrecargar el destino con demasiados datos. El destino le dirá a la fuente cuántos datos puede manejar. De esta forma, los recursos se usan de manera más eficiente. La especificación Reactive Streams también quiere establecer un estándar.

El estándar para JVM está disponible en GitHub . Resumido, la especificación consta de los siguientes elementos:

- **Editor** : el editor es la fuente que enviará los datos a uno o más suscriptores.
- **Suscriptor** : un suscriptor se suscribirá a un editor, indicará la cantidad de datos que el editor puede enviar y procesará los datos.
- **Suscripción** : en el lado del editor, se creará una suscripción, que se compartirá con un suscriptor.
- **Procesador** : un procesador se puede usar para ubicarse entre un editor y un suscriptor, de esta forma se puede realizar una transformación de los datos.

La especificación Reactive Streams es un estándar y desde Java 9, se incluye en Java con Flow API . Eche un vistazo más de cerca a la especificación de Flujos reactivos y la API de flujo de Java 9. Como notará, las interfaces para Publisher, Subscriber, Subscription y Processor son idénticas.

Proyecto Reactor

El siguiente paso para emprender nuestro viaje es el Proyecto Reactor . Este proyecto proporciona una biblioteca reactiva basada en la especificación de Flujos reactivos. En otras palabras, una implementación de la especificación. Básicamente, Reactor proporciona dos tipos:

- **Mono** : implementa Publisher y devuelve 0 o 1 elementos;
- **Flux** : implementa Publisher y devuelve N elementos.

But what is its relation with Spring Webflux? Spring Webflux is based on Project Reactor and this brings us to the main goal of this post: learn more about Spring Webflux.

Spring Webflux

Spring Webflux se presenta con Spring 5, la documentación oficial de Spring se puede encontrar aquí . Puede leer toda la documentación, pero es bastante información. Sugiero que primero vea un

bastante información. Sugiero que primero vea un ejemplo (este artículo) y luego vuelva a consultar la documentación. Es importante saber que hay dos formas de usar Spring Webflux. Una usando anotaciones, que es bastante similar a Spring MVC, y otra que usa una forma funcional. Usaré la forma funcional. Además de eso, debes saber que Spring MVC y Spring Webflux existen uno al lado del otro. Spring MVC se utiliza para procesamiento síncrono, mientras que Spring Webflux para procesamiento asíncrono.

Ejemplo de Spring Webflux Getting Started

Para que sea fácil para mí, traté de seguir el ejemplo de Spring Webflux getting started: Creación de un servicio web RESTful reactivo . Seguí la guía y usé también Java 9. Sin embargo, no se ejecutó fuera de la caja. Tuve que hacer algunas adaptaciones. No repetiré los pasos a seguir (puede leerlo en la guía), pero documentaré mis hallazgos cuando intente construir y ejecutar la aplicación. El ejemplo se puede encontrar en el paquete *com.mydeveloperplanet.myspringwebfluxplanet.greeting* .

Primero, tuve que agregar el repositorio Milestones al POM para usar spring-boot-starter-parent versión 2.0.0.RC2.

```
1 < padre >
2   < groupId > org.springframework.boot </ groupId >
3   < artifactId > spring-boot-starter-parent </ artifactId >
4   < version > 2.0.0.RC2 </ version >
5 </ parent >
6 ...
7 < repositorios >
8   < repositorio >
9     < id > primavera-hitos </ id >
10    < nombre > Spring Milestones </ nombre >
11    < url > https://repo.spring.io/milestone </ url >
12  < instantáneas >
13    < habilitado > falso </ enabled >
14  </ snapshots >
```

```

15     </ repository >
16 </ repositories >
17
18 < pluginRepositories >
19   < pluginRepository >
20     < id > primavera-hitos </ id >
21     < nombre > Spring Milestones </ name >
22     < url > https://repo.spring.io/milestone </
23   < instantáneas >
24     < habilitado > falso </ enabled >
25   </ snapshots >
26 </ pluginRepository >
27 </ pluginRepositories >

```

A partir del 1 de marzo, lo anterior ya no es necesario ya que Spring Boot ahora está disponible en general. Ahora es suficiente solo hacer referencia a esta versión de GA:

```

1   < padre >
2     < groupId > org.springframework.boot </ group
3     < artifactId > spring-boot-starter-parent </
4     < version > 2.0.0.RELEASE </ version >
5   </ parent >

```

Además, en las dependencias, tuve que usar spring-boot-starter-webflux en lugar de spring-boot-starter-web.

```

1   < dependencia >
2     < groupId > org.springframework.boot </ group
3     < artifactId > spring-boot-starter-webflux </
4   </ dependency >

```

Como utilicé Java 9, tuve que agregar la siguiente información de módulo:

```

1   módulo com . mydeveloperplanet . myspringwebfl
2   requiere reactor . núcleo ;

```

```

3     requiere primavera . web ;
4     requiere primavera . contexto ;
5     requiere primavera . webflux ;
6     requiere primavera . arranque ;
7     requiere primavera . arranque . autoconfigur
8 }

```

Ahora, construya y ejecute el ejemplo con Maven target spring-boot: run.

En la consola, vemos que se imprime la siguiente línea:

```

1 >> resultado = ¡Hola, primavera!

```

También puede probarlo en el navegador, usando la URL `http://localhost:8080/hello`.

Al mirar el registro de la consola, vemos que se está utilizando Netty Webserver. Este es el valor predeterminado para las aplicaciones Spring Webflux.

Ejemplo básico de Spring Webflux

Ahora, veamos si podemos crear un ejemplo básico y entender qué está sucediendo exactamente aquí.

Creamos el paquete

`com.mydeveloperplanet.myspringwebfluxplanet.examples` y agregamos un `ExampleRouter` y un `ExampleHandler`, similar al ejemplo de saludo, pero tratamos de hacer el mínimo para que sea accesible en la URL `http://localhost:8080/example`.

El `ExampleRouter` se convierte en lo siguiente:

```

1 @Configuración
2 public class ExampleRouter {
3
4     @Frijol
5     pública RouterFunction < ServerResponse > r
6

```

```

7         return RouterFunctions
8             . ruta ( RequestPredicates . GET ( "/" €
9         }
10    }

```

El ExampleHandler se convierte en lo siguiente:

```

1  @Componente
2  clase pública ExampleHandler {
3
4      public Mono < ServerResponse > hello ( soli
5          devuelve ServerResponse . ok () contentTyp
6          . body ( BodyInserters . fromObject (
7      }
8  }

```

Ejecute el *resorte de arranque* Maven : *ejecute* . Vaya a la URL `http: // localhost: 8080 / example` en su navegador. El siguiente error se muestra:

Whitelabel Error Page

This application has no configured error view, so you are seeing this as a fallback.

Sun Feb 11 12:54:04 CET 2018

There was an unexpected error (type=Not Found, status=404).

En el registro de la consola, vemos lo siguiente:

```

1  Reemplazando la definición de bean para bean 'r

```

Ahora cambie la *ruta* del método en la clase *ExampleRouter* en *routeExample* y vuelva a ejecutar la aplicación. Ahora el navegador muestra lo siguiente como se esperaba:

```

1  ¡Hola, Spring Webflux Example!

```

Como mínimo, solo necesitamos un Enrutador y un Manejador. Debemos asegurarnos de que Router Bean tiene un nombre único, de lo contrario, puede ser sobrescrito por otro Bean con el mismo nombre.

El RouterFunction determina la ruta que coincide y finalmente se resuelve en un método Handler.

Spring Webflux 'un paso más' Ejemplo

Ahora sabemos cómo configurar lo básico, vamos un paso más allá y exploramos algunas funcionalidades adicionales.

Creamos un método *routeExampleOneStepFurther* en la clase *ExampleRouter* y encadenamos dos rutas entre sí mediante el método *andRoute*. De esta forma, podemos encadenar varias rutas entre sí y dejar que se resuelvan con diferentes métodos de manejo. Las rutas se evalúan una a una, por lo que debe asegurarse de definir rutas específicas antes de las rutas más genéricas. El método *routeExampleOneStepFurther* es el siguiente:

```

1 public RouterFunction < ServerResponse > routeExampleOneStepFurther() {
2
3     return RouterFunctions
4         . ruta ( RequestPredicates . GET ( "/" + examplePath ) )
5         . andRoute ( RequestPredicates . GET ( "/" + examplePath ) )
6     }

```

Los métodos de manejo se crean en la clase *ExampleHandler* y simplemente devuelven una cadena:

```

1 public Mono < ServerResponse > helloFurther1() {
2     devuelve ServerResponse . ok () contentType ( "text/html" )
3     . body ( BodyInserters . fromObject ( "¡Hola!" ) )
4 }
5
6 public Mono < ServerResponse > helloFurther2() {
7     devuelve ServerResponse . ok () contentType ( "text/html" )
8     . body ( BodyInserters . fromObject ( "¡Hola!" ) )
9 }

```

9 }

Ahora compile y ejecute la aplicación e invoque la URL `http://localhost:8080/exampleFurther1`. El resultado es el siguiente:

Hola, Spring Webflux Ejemplo 1 !

Del mismo modo, la URL `http://localhost:8080/exampleFurther2` nos muestra el siguiente resultado:

```
1  Hola, Spring Webflux Ejemplo 2 !
```

Hemos probado manualmente, ahora vamos a crear una prueba de unidad para él. La prueba unitaria es la siguiente:

```

1  @RunWith ( clase SpringRunner . )
2  @SpringBootTest ( webEnvironment = SpringBootTest
3  clase pública ExampleRouterTest {
4
5  @Autowired
6  WebTestClient webTestClient privado ;
7
8  @Prueba
9  public void testExampleOneStepFurther () {
10
11  webTestClient
12  . get (). uri ( "/" exampleFurther1" )
13
14  . accept ( MediaType . TEXT_PLAIN )
15  . intercambio ()
16  . expectStatus (). isOk ()
17  . expectBody ( String . clase ). isEqualTo
18
19  webTestClient
20  . get (). uri ( "/" exampleFurther2" )
21
22  . accept ( MediaType . TEXT_PLAIN )
23  . intercambio ()
24  . expectStatus (). isOk ()
25  . expectBody ( String . clase ). isEqualTo

```



```

23     }
24
25 }
```

La prueba se basa en el ejemplo de Spring Webflux. Durante la prueba, un servidor se iniciará en un puerto aleatorio e invocará las URL una después de la otra. El método de *intercambio* nos da acceso a la respuesta. Primero, se verifica el estado de la respuesta y luego verificamos si la respuesta es correcta. En el ejemplo Spring Webflux, el método *equals* se usa para verificar la respuesta, pero esto siempre tiene éxito, incluso si la respuesta no es igual al resultado. Esto no es correcto, el método *isEqualTo* se debe usar para tener una prueba correcta.

Ejemplo de "últimos pasos" de Spring Webflux

Finalmente, crearemos una ruta que toma un parámetro, agrega una prueba y agrega una prueba para una llamada API fallida.

En nuestro método *routeExampleOneStepFurther* , agregamos otra ruta y la dejamos resolver con el método *helloFurther3* de la clase *ExampleHandler* . La ruta es la siguiente:

```

1 . andRoute ( RequestPredicates . GET ( "/" examp
```

El método *helloFurther2* extrae un parámetro de nombre y dice hola al nombre de pila:

```

1 public Mono < ServerResponse > helloFurther3
2     String name = solicitud . queryParams ( "nc
3     devuelve ServerResponse . ok () contentType
4     . cuerpo ( BodyInserters . fromObject ( "Hc
5 }
```

Cree y ejecute la aplicación y vaya a url <http://>

localhost: 8080 / exampleFurther3? Name = My%20Developer%20Planet . Esto nos da el resultado esperado:

```
1  Hola, mi desarrollador Planet!
```

La prueba que agregamos es similar a las pruebas anteriores que hemos escrito:

```
1  @Prueba
   public void testExampleWithParameter () {
2      webTestClient
3          . get (). uri ( "/" exampleFurther3? name =
4          . accept ( MediaType . TEXT_PLAIN )
5          . intercambio ()
6          . expectStatus (). isOk ()
7          . expectBody ( String . clase ). isEqualTo
8      }
9  }
```

Para concluir, agregamos una prueba que devolvería una respuesta http 404. El estado esperado para una ruta de URL no resuelta se prueba con el método *isNotFound* :

```
1  @Prueba
2  public void testExampleWithError () {
3      webTestClient
4          . get (). uri ( "/" ejemplo / algo" )
5          . accept ( MediaType . TEXT_PLAIN )
6          . intercambio ()
7          . expectStatus (). isNotFound ();
8  }
```

Resumen

En esta publicación, traté de brindar información sobre cómo comenzar con Spring Webflux:

- Lectura sugerida antes de comenzar con la programación reactiva
- Un ejemplo básico de Spring Webflux


Preparado por Spring

- Algunos ejemplos básicos de Spring Webflux para realizar algunos primeros pasos

Debe quedar claro que esta es solo información muy básica, y a partir de este punto, hay muchas otras cosas que explorar sobre Spring Webflux.

Descargar Building Reactive Microservices en Java : diseño de aplicaciones asíncronas y basadas en eventos. Presentado en asociación con Red Hat .

Temas: JAVA, SPRING WEBFLUX, JAVA 9, CORRIENTES ASINCRÓNICAS, REACTIVAS, TUTORIAL

Publicado en DZone con el permiso de Gunter Rotsaert. [Vea el artículo original aquí.](#) 
Las opiniones expresadas por los contribuidores de DZone son suyas.

Recursos para socios de Java

Creación de microservicios reactivos en Java: diseño de aplicaciones asíncronas y basadas en eventos

Programa de desarrollo de Red Hat



Migrar a bases de datos de Microservicio

Programa de desarrollo de Red Hat



Comience con Spring Security 5.0 y OpenID Connect (OIDC)
Okta



Microsistemas reactivos: la evolución de los microservicios a escala

Lightbend



Execute Automated Testing in 3 Steps With Katalon Automation

Recorder

by Alex Jones · Mar 08, 18 · DevOps Zone

The Nexus Suite is uniquely architected for a DevOps native world and creates value early in the development pipeline, provides precise contextual controls at every phase, and accelerates DevOps innovation with automation you can trust. Read how in this ebook.

1. Automated Testing Introduction

Automated testing is a method using an automated testing tool to write and execute automated test cases/test suites on a software application, comparing the actual results to the expected behavior and creating test reports.

In a nutshell, automated test scripts are the scripts prepared before running a test. When needed, these scripts will be executed the code of a web application to check whether it meets the requirements or not.

When it comes to automated testing, it's hard for testers to be good at it without programming skills. But over time, testing tools have matured to simplify the process of testing. There are a lot of codeless tools on the market providing a graphical environment so that testers can easily create test cases. One of these tools is Katalon Recorder (a perfect successor to Selenium IDE), which will be mentioned as the comprehensive solution in this post.

Automated testing helps eliminate tasks that are too time-consuming and laborious to be performed manually, plus, automated tests can be run repeatedly once they have been created. In other words, automated testing helps increase the effectiveness and efficiency of software testing.

2. The Challenges of Executing Automated

Tests

As I stated above about the complexity when running tests with a lack of programming knowledge, there are still so many other challenges:

- **Preparing test scripts**

In automated testing, a test script is a short program written in a programming language; that's one of the reasons why programming skills are required in automated testing. It is one of the constraints for manual and non-technical testers when executing automated testing.

- **Managing tests**

Not only is scripting challenging, but so is managing tests if you are a manual or non-technical tester. Why do you have to control tests? If you don't manage your tests and scripts, many things can occur. First, the effort is duplicated because many people can build the same test scripts. Second, test scripts are created for a single purpose and cannot be reused. Existing automated test scripts are at risk of corruption if they are modified without the knowledge of the original author, and many other things.

- **Reporting**

Every tool has a standard for reporting the test results. However, sometimes these generic reports do not fulfill your needs. It is challenging to customize the report because it requires a great deal of effort, good planning, and maintenance.

In order to crack all these challenges, in this article, I will not only show you how to create a test case and execute automated tests in three easy steps, but also the solution to manage tests and reports efficiently.

3. 3 Magical Steps to Execute Automated Web Testing

All magicians need supporting tools, and so does web testing execution. My magical trick will be performed

with a tool called Katalon Recorder that can be installed in 3 seconds on both Chrome and Firefox's latest versions.

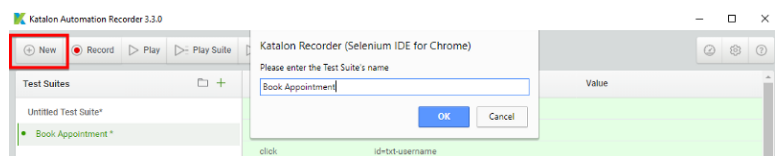
A few words about Katalon Recorder: it has been hailed as one of the alternative solutions since Selenium IDE no longer works from Firefox 55 onwards (read the official announcement). As the toolmakers stated, “Katalon Automation Recorder records actions and captures web elements on web applications to let you generate, edit, and execute automated test cases quickly and easily without programming knowledge requirements.”

And yes, Katalon Recorder is not the only one; there are the same solutions out there, such as Protractor, Kantu, or Robot framework, but I will just show the easiest way and the most proficient tool to do the magical things.

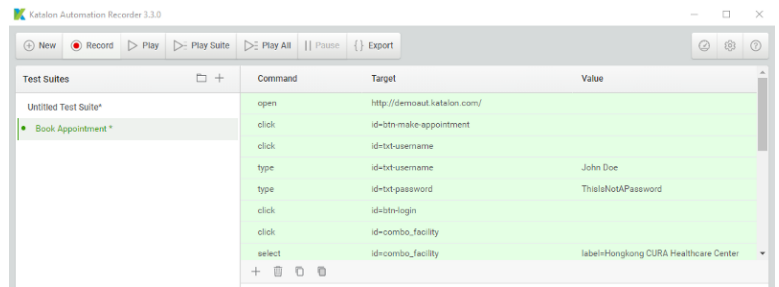
Let's use a sample test case whose scenario is as follows. Imagine you have a service booking site that requires the user to log in, fill in a form, and submit it to accomplish the booking process. After changing a little bit in the theme, you may wonder whether this function is still working or not. The 3 easy steps below will help you get rid of the confusion.

Step 1: Create a New Test Case Using the Record Function

- Click “New” to create a new test case



- Click “Record” to generate a test case with the following steps:
 - Access <http://demoaut.katalon.com/profile.php#login> (the Katalon demo AUT website)
 - Enter username/password (John Doe/ThisIsNotAPassword)
 - Fill in and submit the form
 - Log out

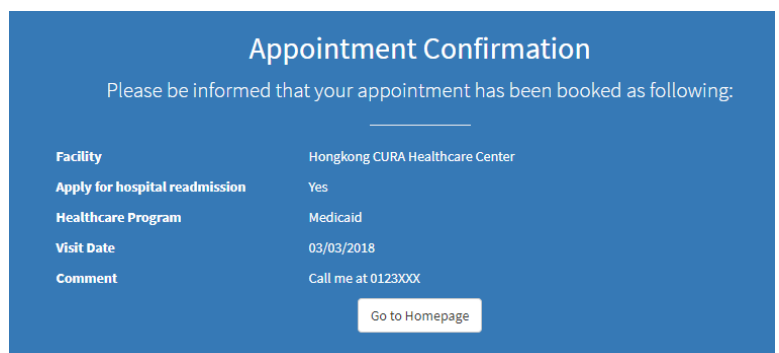


Step 2: Keep Calm and Execute Automated Testing

Click “Play” to run the test and monitor the test case. If the test case fails somewhere, you can stop the playback and remove the bug right there.

You can change the “Value” input manually to test the function with more accounts.

A happy case should be shown as in the image below:



It will bring you back to the homepage without logging into any account.

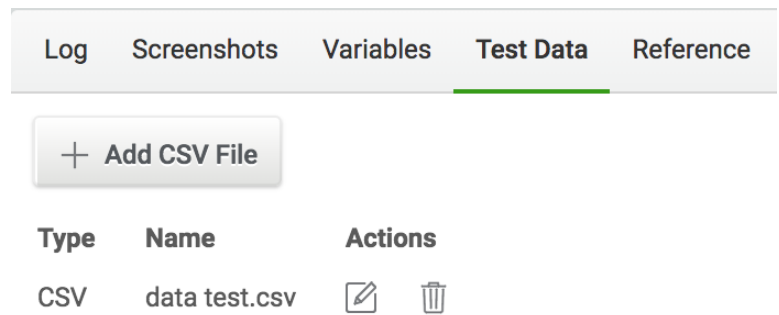


Step 3: Advanced Steps for Automated Test Experts

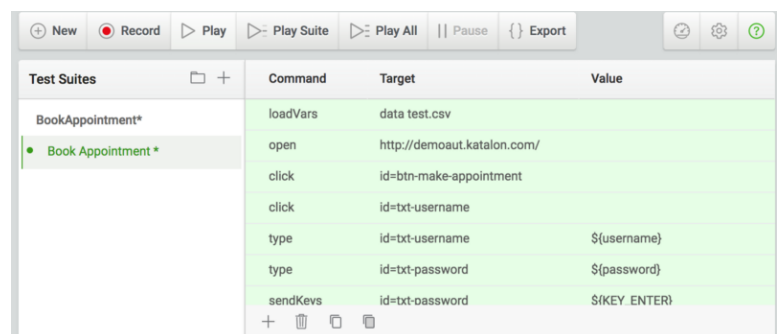
If you want to use more variable test data to cover different cases and run the test with these data automatically, Katalon Recorder supports data-driven testing, which allows users to define data sets and execute test scripts that use these data.

- You should download the sample test data here and add the file named “test data.csv” in the

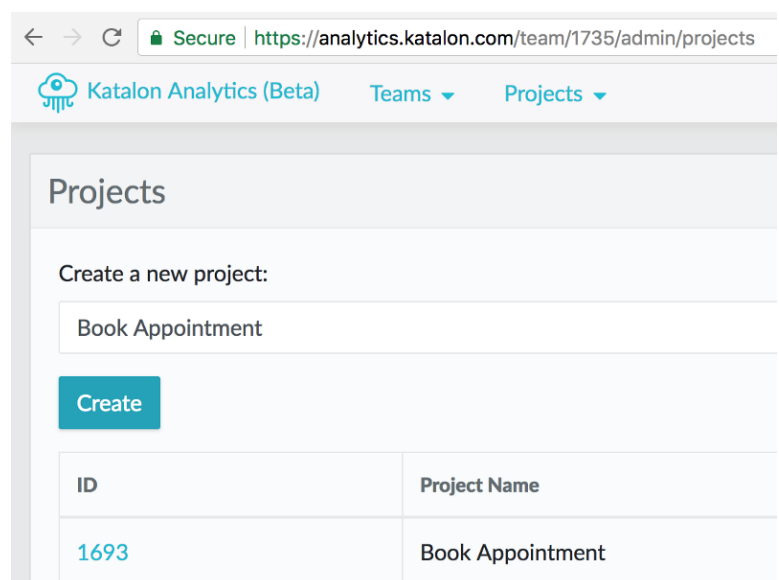
“Test Data” tab.



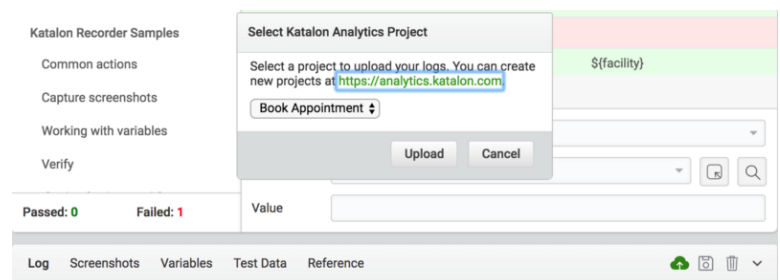
- Import the test cases that are saved as an HTML file by clicking on “Open test suite” instead of clicking on “Create test suite” and open my sample test case.
- Click “Play” to execute the test automatically with the variable input.



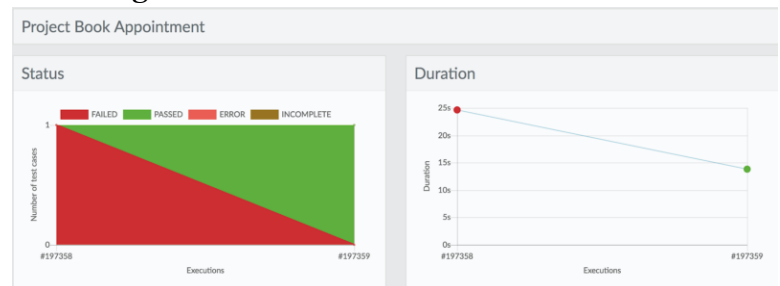
That is not all; the Katalon platform also offers a tool called Katalon Analytics, to help you save all the test reports and display them visually through colorful charts. You can upload your results directly to Katalon Recorder by clicking “Upload logs” (the cloud icon). You may be required for sign up a Katalon account at www.katalon.com (for free) and log into Katalon Analytics at <https://analytics.katalon.com> to create a new project.



Then, come back to the Katalon Recorder window and choose the project for which you want to upload the logs, and click “Upload”



Below is the chart that shows my test results after executing the test case twice:



Although Katalon Analytics is launched as the beta version, it meets all my demands for test reports, even shares the results with teammates easily.

4. Run Selenese (Selenium IDE) Scripts on Chrome and Firefox

This is the corner for Selenium IDE fans who are depressed and disappointed after the official announcement of the end of Selenium IDE from Firefox 55 onwards. Some guys at Katalon seems to be bringing the hope back by allowing users to import Selenese scripts from Selenium IDE to Katalon Recorder. Check it out here.

I will say no more about Katalon platform in this post, even it is great and free - it should belong to another post that may be named “How to leverage Katalon platform (Katalon Studio, Katalon Docker, Katalon Analytics, Katalon Recorder) to make your testing life easier.”

Executing automated test cases and managing test reports is not as painful as we thought. In general, with a wise testing strategy and a proper supporting tool collection, test execution or automated testing is

no longer a big deal.

The DevOps Zone is brought to you in partnership with Sonatype Nexus. See how the Nexus platform infuses precise open source component intelligence into the DevOps pipeline early, everywhere, and at scale. Read how in this ebook.

Topics: DEVOPS, AUTOMATED TESTING, SOFTWARE TESTING, TUTORIAL, KATALON STUDIO, SELENIUM, WEB TESTING

Opinions expressed by DZone contributors are their own.