



Descargue el Informe de tendencias Kubernetes 2019 de DZone para ver el impacto futuro que tendrá Kubernetes.

[Descargar informe](#)

Trabajando con el Programador Java

por Joydip Kumar MVB · 28 y 18 de noviembre · Zona Java · Tutorial

Enjoy productive Java and discover how every aspect of [IntelliJ IDEA](#) is specifically designed to maximize developer

Presented by JetBrains

En este artículo, vamos a cubrir los siguientes temas relacionados con el Programador Java:

- Programando una tarea en Java
- `SchedulerConfigurer` vs. `@Scheduled`
- Cambiar la `cron` expresión dinámicamente
- Ejecución de dependencia entre dos tareas.

Programando una Tarea en Java

El planificador se usa para programar un hilo o tarea que se ejecuta en un cierto período de tiempo o periódicamente en un intervalo fijo. Hay varias formas de programar una tarea en Java.

- `java.util.TimerTask`
- `java.util.concurrent.ScheduledExecutorService`
- Programador de cuarzo
- `org.springframework.scheduling.TaskScheduler`

`TimerTask` es ejecutado por un hilo demoníaco. Cualquier retraso en una tarea puede retrasar la otra tarea en un horario. Por lo tanto, no es una opción viable cuando se deben ejecutar varias tareas de forma asíncrona en un momento determinado.

Veamos un ejemplo:

```
1  paquete com . ejemplo . timerExamples ;
2
3  Importar Java . util . Temporizador ;
4
5  público de clase ExecuteTimer {
6
7      public static void main ( String [] args ) {
8          TimerExample te1 = new TimerExample ( "Tarea1" );
9          TimerExample te2 = new TimerExample ( "Tarea2" );
10
11          Temporizador t = nuevo Temporizador ();
12          t . scheduleAtFixedRate ( te1 , 0 , 5 * 1000 );
```

```

13         t . scheduleAtFixedRate ( te2 , 0 , 1000 );
14     }
15 }

1  clase pública TimerExample extiende TimerTask {
2
3     nombre de cadena privada ;
4     public TimerExample ( String n ) {
5         esta . nombre = n ;
6     }
7
8     @Anular
9     public void run () {
10         Sistema . a cabo . println ( Thread . currentThread (). getName () + " " + name + "la tarea se ejec
11         if ( "Tarea1" . equalsIgnoreCase ( nombre )) {
12             prueba {
13                 Hilo . dormir ( 10000 );
14             } catch ( InterruptedException e ) {
15                 // TODO Bloque de captura generado automáticamente
16                 e . printStackTrace ();
17             }
18         }
19     }

```

Salida:

```

1  Temporizador-0 Tarea1 la tarea se ejecutó correctamente Mié 14 de noviembre 14:32:49 GMT 2018
2  Temporizador-0 Tarea2 la tarea se ejecutó con éxito Mié 14 de noviembre 14:32:59 GMT 2018
3  Temporizador-0 Tarea2 la tarea se ejecutó con éxito Mié 14 de noviembre 14:32:59 GMT 2018
4  Temporizador-0 Tarea2 la tarea se ejecutó con éxito Mié 14 de noviembre 14:32:59 GMT 2018
5  Temporizador-0 Tarea2 la tarea se ejecutó con éxito Mié 14 de noviembre 14:32:59 GMT 2018
6  Temporizador-0 Tarea2 la tarea se ejecutó con éxito Mié 14 de noviembre 14:32:59 GMT 2018
7  Temporizador-0 Tarea2 la tarea se ejecutó con éxito Mié 14 de noviembre 14:32:59 GMT 2018
8  Temporizador-0 Tarea1 la tarea se ejecutó con éxito Mié 14 de noviembre 14:32:59 GMT 2018
9  Timer-0 Task2 la tarea se ejecutó con éxito el miércoles 14 de noviembre 14:33:09 GMT 2018
10 Timer-0 Task2 la tarea se ejecutó con éxito el miércoles 14 de noviembre 14:33:09 GMT 2018
11 Timer-0 Task2 la tarea se ejecutó con éxito el miércoles 14 de noviembre 14:33:09 GMT 2018
12 Timer-0 Task2 la tarea se ejecutó con éxito el miércoles 14 de noviembre 14:33:09 GMT 2018
13 Timer-0 Task2 la tarea se ejecutó con éxito el miércoles 14 de noviembre 14:33:09 GMT 2018
14 Temporizador-0 Tarea1 la tarea se ejecutó con éxito Mié 14 de noviembre 14:33:09 GMT 2018

```

En la ejecución anterior, está claro que la tarea 2 se atasca porque el subproceso que maneja la tarea1 va a dormir durante 10 segundos. Por lo tanto, solo hay un hilo de demonio que funciona tanto en la tarea 1 como en la tarea 2, y si uno recibe un golpe, todas las tareas se retrasarán.

`ScheduledExecutorService` y `TaskScheduler` funciona de la misma manera. La única diferencia con la primera es la biblioteca Java y la segunda es el marco Spring. Entonces, si la aplicación está en primavera, `TaskScheduler` puede ser una mejor opción para programar trabajos.

Ahora, veamos el uso de la `TaskScheduler` interfaz y podemos usarla en Spring.

SchedulerConfigurer Vs. @Programado

Spring proporciona una programación basada en anotaciones con la ayuda de `@Scheduled`.

Los subprocesos son manejados por el marco de Spring, y no tendremos ningún control sobre los subprocesos que funcionarán en las tareas. Echemos un vistazo al siguiente ejemplo:

```

1  @Configuración
2  @EnableScheduling
3  clase pública ScheduledConfiguration {
4
5      @Scheduled ( fixedRate = 5000 )
6      public void executeTask1 () {
7          Sistema . a cabo . println ( Thread . currentThread (). getName () + "La Tarea1 ejecutada en"
8
9          prueba {
10              Hilo . dormir ( 10000 );
11          } catch ( InterruptedException e ) {
12              // TODO Bloque de captura generado automáticamente
13              e . printStackTrace ();
14          }
15      }
16      @Scheduled ( fixedRate = 1000 )
17      public void executeTask2 () {
18          Sistema . a cabo . println ( Thread . currentThread (). getName () + "La Tarea2 ejecutada en"
19      }
20  }

```

Salida:

```

1  programación - 1 El Task2 ejecutado en Mie Nov 14 14 : 22 : 59 GMT 2018
2  programación - 1 El Task2 ejecutado en Mie Nov 14 14 : 22 : 59 GMT 2018
3  programación - 1 El Task2 ejecutado en Mie Nov 14 14 : 22 : 59 GMT 2018
4  programación - 1 El Task2 ejecutado en Mie Nov 14 14 : 22 : 59 GMT 2018
5  programación - 1 El Task2 ejecutado en Mie Nov 14 14 : 22 : 59 GMT 2018
6  programación - 1 El Task1 ejecutado en Mie Nov 14 14 : 22 : 59 GMT 2018
7  programación - 1 El Task2 ejecutado en Mie Nov 14 14 : 23 : 09 GMT 2018
8  programación - 1 El Task2 ejecutado en Mie Nov 14 14 : 23 : 09 GMT 2018
9  programación - 1 El Task2 ejecutado en Mie Nov 14 14 : 23 : 09 GMT 2018
10 programación - 1 El Task2 ejecutado en Mie Nov 14 14 : 23 : 09 GMT 2018
11 programación - 1 El Task2 ejecutado en Mie Nov 14 14 : 23 : 09 GMT 2018
12 programación - 1 El Tarea1 ejecutado en Mie Nov 14 14 : 23 : 09 GMT 2018

```

Hay un hilo de programación-1, que maneja tanto la tarea1 como la tarea2. En el momento en que la tarea 1 se va a dormir durante 10 segundos, la tarea 2 también la espera. Por lo tanto, si hay dos trabajos ejecutándose al mismo tiempo, uno esperará a que se complete otro.

Ahora, intentaremos escribir una tarea de planificador donde queremos ejecutar la tarea1 y la tarea2 de forma asíncrona. Habrá un grupo de subprocesos y programaremos cada tarea en el `ThreadPoolTaskScheduler`. La clase necesita implementar la `SchedulingConfigurer` interfaz. Da más control a los hilos del planificador en comparación con `@Scheduled`.

```

1  @Configuración
2  @EnableScheduling
3  public class ScheduledConfiguration implementa SchedulingConfigurer {
4
5      TaskScheduler taskScheduler ;
6      Private ScheduledFuture <?> trabajo1 ;
7      Private ScheduledFuture <?> trabajo2 ;
8      @Anular
9      public void configureTasks ( ScheduledTaskRegistrar taskRegistrar ) {
10
11          ThreadPoolTaskScheduler threadPoolTaskScheduler = new ThreadPoolTaskScheduler ();
12          threadPoolTaskScheduler . setPoolSize ( 10 ); // Establecer el grupo de hilos
13          threadPoolTaskScheduler . setThreadNamePrefix ( "hilo-planificador" );
14          threadPoolTaskScheduler . initialize ();
15          job1 ( threadPoolTaskScheduler ); // Asigna el trabajo1 al planificador
16          job2 ( threadPoolTaskScheduler ); // Asigna el trabajo1 al planificador
17          esta . taskScheduler = threadPoolTaskScheduler ; // esto se usará en la parte posterior de
18
19          taskRegistrar . setTaskScheduler ( threadPoolTaskScheduler );
20      }
21
22      Private void job1 ( TaskScheduler Scheduler ) {
23          trabajo1 = planificador . horario ( nuevo Runnable () {
24              @Anular
25              public void run () {
26                  Sistema . a cabo . println ( Thread . currentThread (). getName () + "La Tarea1 ejecutando" );
27
28                  prueba {
29                      Hilo . dormir ( 10000 );
30                  } catch ( InterruptedException e ) {
31                      // TODO Bloque de captura generado automáticamente
32                      e . printStackTrace ();
33                  }
34              }, nuevo Trigger () {
35                  @Anular
36                  public Date nextExecutionTime ( TriggerContext triggerContext ) {
37                      String cronExp = "0/5 * * * *?" ; // Se puede extraer de un db.
38                      devolver nuevo CronTrigger ( cronExp ). nextExecutionTime ( triggerContext );
39                  }
40              });
41      }
42
43      Private void job2 ( TaskScheduler Scheduler ) {
44          trabajo2 = planificador . horario ( nuevo Runnable () {
45              @Anular
46              public void run () {
47                  Sistema . a cabo . println ( Thread . currentThread (). getName () + "La Tarea2 ejecutando" );
48
49              }, nuevo Trigger () {

```

```

50         @Anular
51         public Date nextExecutionTime ( TriggerContext triggerContext ) {
52             String cronExp = "0/1 * * * *?" ; // Se puede extraer de un db. Esto se ejec
53             devolver nuevo CronTrigger ( cronExp ). nextExecutionTime ( triggerContext
54         }
55     });
56 }
57 }

```

Salida:

```

1 Scheduler-thread1 La tarea2 ejecutada el miércoles 14 de noviembre 15:02:46 GMT 2018
2 Scheduler-thread2 La tarea 2 ejecutada el miércoles 14 de noviembre 15:02:47 GMT 2018
3 Scheduler-thread3 La tarea2 ejecutada el miércoles 14 de noviembre 15:02:48 GMT 2018
4 Scheduler-thread2 La tarea 2 se ejecutó el miércoles 14 de noviembre 15:02:49 GMT 2018
5 Scheduler-thread1 La tarea2 ejecutada el miércoles 14 de noviembre 15:02:50 GMT 2018
6 Scheduler-thread7 La tarea1 ejecutada el miércoles 14 de noviembre 15:02:50 GMT 2018
7 Scheduler-thread3 La tarea2 ejecutada el miércoles 14 de noviembre 15:02:51 GMT 2018
8 Scheduler-thread5 La tarea2 ejecutada el miércoles 14 de noviembre 15:02:52 GMT 2018
9 Scheduler-thread2 La tarea 2 se ejecutó el miércoles 14 de noviembre 15:02:53 GMT 2018
10 Scheduler-thread6 La tarea2 ejecutada el miércoles 14 de noviembre 15:02:54 GMT 2018
11 Scheduler-thread6 La tarea 2 ejecutada el miércoles 14 de noviembre 15:02:55 GMT 2018
12 Scheduler-thread6 La tarea 2 ejecutada el miércoles 14 de noviembre 15:02:56 GMT 2018
13 Scheduler-thread6 La tarea2 ejecutada el miércoles 14 de noviembre 15:02:57 GMT 2018
14 Scheduler-thread6 La tarea2 ejecutada el miércoles 14 de noviembre 15:02:58 GMT 2018
15 Scheduler-thread6 La tarea2 ejecutada el miércoles 14 de noviembre 15:02:59 GMT 2018
16 Scheduler-thread6 La tarea 2 ejecutada el miércoles 14 de noviembre 15:03:00 GMT 2018
17 Scheduler-thread2 La tarea 2 se ejecutó el miércoles 14 de noviembre 15:03:01 GMT 2018
18 Scheduler-thread2 La tarea 2 se ejecutó el miércoles 14 de noviembre 15:03:02 GMT 2018
19 Scheduler-thread2 La tarea 2 se ejecutó el miércoles 14 de noviembre 15:03:03 GMT 2018
20 Scheduler-thread2 La tarea 2 se ejecutó el miércoles 14 de noviembre 15:03:04 GMT 2018
21 Scheduler-thread10 La tarea 2 ejecutada el miércoles 14 de noviembre 15:03:05 GMT 2018
22 Scheduler-thread8 La Tarea1 ejecutada el miércoles 14 de noviembre 15:03:05 GMT 2018-

```

Estoy creando dos trabajos: trabajo1 y trabajo2. Entonces, lo programaré usando `TaskScheduler`. Esta vez, estoy usando una expresión Cron para programar el trabajo1 en cada intervalo de cinco segundos y el trabajo2 cada segundo. Job1 se atasca durante 10 segundos y veremos que job2 sigue funcionando sin interrupciones. Vemos que tanto la tarea 1 como la tarea 2 están siendo manejadas por un grupo de subprocesos que se crea usando `ThreadPoolTaskScheduler`

Cambiar una expresión de Cron dinámicamente

Siempre podemos mantener la expresión cron en un archivo de propiedades utilizando la configuración de Spring. Si el servidor Spring Config no está disponible, también podemos buscarlo desde el DB. Cualquier actualización de la expresión cron actualizará el Programador. Pero para cancelar la programación actual y ejecutar la nueva programación, podemos exponer una API para actualizar el trabajo cron:

```

1 public void refreshCronSchedule () {
2
3     if ( trabajo1 != nulo ) {

```

```

3     }
4     trabajo1 . cancelar ( verdadero );
5     scheduleJob1 ( taskScheduler );
6 }
7
8 if ( trabajo2 != nulo ) {
9     trabajo2 . cancelar ( verdadero );
10    scheduleJob2 ( taskScheduler );
11 }
12 }

```

Además, puede invocar el método desde cualquier controlador para actualizar la programación cron.

Ejecución de dependencia entre dos tareas

Hasta ahora, sabemos que podemos ejecutar los trabajos de forma asincrónica utilizando la interfaz `TaskScheduler` y `Schedulingconfigurer`. Ahora, supongamos que tenemos `job1` que se ejecuta durante una hora a la 1 am y `job2` que se ejecuta a las 2 am. Pero, `job2` no debe comenzar a menos que complete1. También tenemos otra lista de trabajos que pueden ejecutarse entre la 1 y las 2 am y son independientes de otros trabajos.

Veamos cómo podemos crear una dependencia entre el `trabajo1` y el `trabajo2`, pero ejecutar todos los trabajos de forma asincrónica a la hora programada.

Primero, declaremos una variable volátil:

```

1  privado booleano volátil job1Flag = false ;
2
3  privado void scheduleJob1 ( planificador de TaskScheduler ) {
4      trabajo1 = planificador . horario ( nuevo Runnable () {
5          @Anular
6          public void run () {
7              Sistema . a cabo . println ( Thread . currentThread (). getName () + "La Tarea1 eje
8              prueba {
9                  Hilo . dormir ( 10000 );
10             } catch ( InterruptedException e ) {
11                 // TODO Bloque de captura generado automáticamente
12                 e . printStackTrace ();
13             }
14             job1Flag = true ; // configurando la bandera como verdadera para marcarla como compl
15         }
16     }, nuevo Trigger () {
17         @Anular
18         public Date nextExecutionTime ( TriggerContext triggerContext ) {
19             String cronExp = "0/5 * * * *?" ; // Se puede extraer de una base de datos
20             devolver nuevo CronTrigger ( cronExp ). nextExecutionTime ( triggerContext );
21         }
22     }
23 }

```

```

1  privado void scheduleJob2 ( planificador TaskScheduler ) {
2      trabajo2 = planificador . horario ( nuevo Runnable () {
3
4          @Anular
5          public void run () {
6              sincronizado ( esto ) {
7                  while ( ! job1Flag ) {
8                      Sistema . a cabo . println ( Thread . currentThread (). getName () + "esperando que el
9
10                     prueba {
11                         espera ( 1000 ); // agrega cualquier número de segundos para esperar
12                     } catch ( InterruptedException e ) {
13                         e . printStackTrace ();
14                     }
15                 }
16
17                 Sistema . a cabo . println ( Thread . currentThread (). getName () + "La Tarea2 ejecutada er
18
19                 job1Flag = false ;
20             }, nuevo Trigger () {
21                 @Anular
22                 public Date nextExecutionTime ( TriggerContext triggerContext ) {
23                     String cronExp = "0/5 * * * *?" ; // Se puede extraer de un db. Esto se ejecutará cada mi
24
25                     devolver nuevo CronTrigger ( cronExp ). nextExecutionTime ( triggerContext );
26                 }
27             });
28         }
29     }

```

```

1  Scheduler-thread2 La tarea1 ejecutada el miércoles 14 de noviembre 16:30:50 GMT 2018
2  Scheduler-thread1 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:30:51 GMT 2018
3  Scheduler-thread1 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:30:52 GMT 2018
4  Scheduler-thread1 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:30:53 GMT 2018
5  Scheduler-thread1 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:30:54 GMT 2018
6  Scheduler-thread1 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:30:55 GMT 2018
7  Scheduler-thread1 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:30:56 GMT 2018
8  Scheduler-thread1 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:30:57 GMT 2018
9  Scheduler-thread1 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:30:58 GMT 2018
10 Scheduler-thread1 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:30:59 GMT 2018
11 Scheduler-thread1 La tarea2 ejecutada el miércoles 14 de noviembre 16:31:00 GMT 2018
12 Scheduler-thread2 La tarea 1 ejecutada el miércoles 14 de noviembre 16:31:05 GMT 2018

```

```

13 Scheduler-thread3 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:31:05 C
14 Scheduler-thread3 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:31:06 C
15 Scheduler-thread3 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:31:07 C
16 Scheduler-thread3 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:31:08 C
17 Scheduler-thread3 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:31:09 C
18 Scheduler-thread3 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:31:10 C
19 Scheduler-thread3 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:31:11 C
20 Scheduler-thread3 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:31:12 C
21 Scheduler-thread3 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:31:13 C
22 Scheduler-thread3 esperando que el trabajo1 se complete para ejecutarse Mié 14 de noviembre 16:31:14 C
23 Scheduler-thread3 La tarea2 ejecutada el miércoles 14 de noviembre 16:31:15 GMT 2018
24 Scheduler-thread1 La tarea1 ejecutada el miércoles 14 de noviembre 16:31:20 GMT 2018

```

Elegimos usar una bandera booleana volátil para que no se almacene en caché en el subproceso local, sino que se guarde en la memoria principal y pueda ser utilizada por todos los subprocesos del grupo. Según el indicador, el trabajo2 espera indefinidamente hasta que se complete el trabajo1. Ahora, si el trabajo1 se cuelga, existe la posibilidad de que el trabajo2 espere indefinidamente.

Conclusión

Hay varias formas de programar en Java, y ahora, sabemos cómo usar la API del planificador y la API del planificador Spring para controlar los subprocesos en el grupo.

Discover instant and clever code completion, on-the-fly code analysis, and reliable refactoring tools with

[IDEA](#)

Presented by JetBrains

¿Te gusta este artículo? Leer más de DZone



Spring Core Skills: Your First Spring Application [Video]



Cómo programar una tarea usando la expresión de Cron en Spring Boot [Video]




Implementación de un bloqueo de programador




Tarjeta DZone gratis Java 13

Temas: PLANIFICADOR SIN CRON, SPRING 5, CHRON, JAVA, TUTORIAL, PLANIFICADOR, PLANIFICADOR DE PRIMAVERA, CONFIGURACIÓN DE PRIMAVERA

Publicado en DZone con permiso de Joydip Kumar , DZone MVB . [Vea el artículo original aquí.](#) 
Las opiniones expresadas por los contribuyentes de DZone son propias.

Java en Azure Funciones

por Kiran Kumar · 25 de septiembre, 19 · Zona Java · Tutorial

Download DZone's 2019 Scaling DevOps Trend Report to learn how to ensure security as you scale 
"blameless retrospectives" are a myth, and the key considerations for getting the most out of DevOps

[Download Now](#)

Presented by DZone



Obtenga más información sobre las funciones de Azure en Java

Azure Functions es la informática sin servidor en la arquitectura de Azure que proporciona un modelo de programación para aplicaciones controladas por eventos

En febrero, Microsoft anunció el soporte de Java en Azure Functions 2.0. Esta es una buena noticia para los desarrolladores de Java que acceden a Azure como un proveedor basado en la nube y aprovechan Azure Functions como componentes informáticos.

También te puede interesar : Desarrolladores de Azure para Java

En este artículo, estudiaremos un ejemplo de desarrollo y ejecución de una aplicación basada en Java que se ejecuta en Azure Functions.

Crear aplicación de función desde Azure Portal



Function App

Microsoft

[Create](#)

Primero, debe proporcionar el nombre de la función y otros detalles como la suscripción, el grupo de recursos y los detalles del sistema operativo para aprovisionar las funciones de Azure

Function App □ ×

Create

* App name
mybrandorderservice ✓
.azurewebsites.net

* Subscription
[Dropdown menu]

* Resource Group ⓘ
☐ Create new ☒ Use existing
[Dropdown menu]

* OS
☒ Windows ☐ Linux

* Hosting Plan ⓘ
Consumption Plan ^
Consumption Plan
App Service Plan

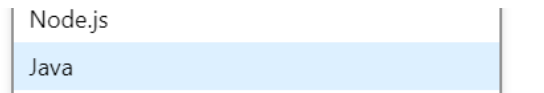
Seleccione el plan de alojamiento. Azure Functions se ejecuta en dos planes:

- **Plan de consumo** : en este plan, las instancias del host de Azure Functions se agregan o eliminan dinámicamente según la cantidad de eventos entrantes. La facturación se basa en el número de ejecuciones, la memoria utilizada y el tiempo de ejecución. La memoria máxima en este plan es de 1,5 GB por instancia
- **Plan de servicio de aplicaciones** : en este plan, Azure Functions se ejecuta en máquinas virtuales dedicadas como otras aplicaciones de servicio de aplicaciones. La memoria en este plan va de 1.75 GB a 14 GB por instancia

Hay un plan más: "Premium", que se encuentra en la etapa de vista previa. Elija cualquiera de los planes anteriores, según los requisitos comerciales y el uso.

Luego, elija una pila de tiempo de ejecución para el entorno de ejecución. En este requisito, seleccionamos Java:

* Runtime Stack
Java ^
.NET Core



Seleccione **Cuenta de almacenamiento** y **Application Insights** ya que el registro de Azure Functions está directamente integrado con Application Insights:

* Storage ⓘ

☐ Create new ☒ Use existing

Application Insights
mybrandorderservice >

Con los detalles anteriores, la aplicación de función se aprovisiona en el plan de uso dado. Con este sencillo ejemplo, podemos ver cómo podemos implementar un servicio Java publicado y alojado en Azure Functions.

Cree un proyecto Maven en su sistema local con el arquetipo de Maven para un entorno de Azure Functions.

A continuación se muestra el comando para crear un proyecto Maven:

```
1 mvn archetype: generar ".microsoft.azure" "-functions-archetype"
```

Importe el proyecto Maven en su IDE favorito, como Eclipse o Visual Studio Code. Luego, abra el pom.xml generado y actualice los siguientes detalles:

```
1 < functionName > </ functionName >
2 < functionAppRegion > </ functionAppRegion >
3 < functionResourceGroup > </ functionResourceGroup >
```

Estos detalles se obtienen cuando ha aprovisionado Azure Functions desde el portal.

Además, agregue cualquier otra biblioteca de terceros como dependencias en el proyecto Maven requerido para su aplicación Java

En nuestro ejemplo, creamos un servicio Java simple con dos métodos: uno que atiende solicitudes HTTP GET y otro que atiende solicitudes HTTP POST.

El siguiente código es para una solicitud POST que crea orden. En nuestro ejemplo, estamos creando una orden ficticia y no hay interacción con ningún sistema de fondo. El propósito de este ejemplo es mostrar cómo podemos construir una solicitud POST en Java publicada en el entorno de Azure Function:

```
1 @FunctionName ( "createorder" )
2 public String createOrder (
3     @HttpTrigger ( name = "req" , métodos = { HttpMethod . POST } ,
4         authLevel = AuthorizationLevel . FUNCION )
5     HttpRequestMessage < Opcional < Cadena >> solicitud ,
6     contexto de ExecutionContext final ) {
7     Cadena orderDetails = solicitud . getBody (). obtener ();
8     contexto . getLogger (). info ( "detalles del pedido" + detalles del pedido );
9     devuelve "orden creada o-8976" ;
10 }
```

La función anterior se anota con `functionName`, que es `CreateOrder` y toma el objeto `HttpRequestMessage` y el `ExecutionContext` objeto como parámetros.

El cuerpo de la solicitud pasado en la solicitud POST se puede obtener utilizando el `getBody` método del `HttpRequestMessage` objeto.

`ExecutionContext` tiene métodos que obtienen configuraciones relacionadas con la función, como la identificación de invocación, el nombre de la función y el objeto de registro que registra los mensajes en Application Insights.

De manera similar, a continuación se muestra un ejemplo para un servicio de solicitud GET y proporciona detalles del pedido al pasar el Id de pedido como un parámetro de consulta en la URL:

```
1 @FunctionName ( "listorder" )
2 public HttpResponseMessage listOrder (
3     @HttpTrigger ( name = "req" , métodos = { HttpMethod . GET },
4         authLevel = AuthorizationLevel . FUNCTION )
5     HttpRequestMessage < Opcional < Cadena >> solicitud ,
6     contexto de ExecutionContext final ) {
7     contexto . getLogger (). info ( "En orden de lista" );
8     // Parse parámetro de consulta
9     Cadena orderRequest = solicitud . getQueryParameters (). get ( "orderId" );
10    String orderId = solicitud . getBody (). o Else ( orderRequest );
11    if ( orderRequest == null ) {
12        volver solicitud . createResponseBuilder ( HttpStatus . BAD_REQUEST )
13            . cuerpo ( "Pase el ID de pedido" ). construir ();
14    } más {
15        volver solicitud . createResponseBuilder ( HttpStatus . OK )
16            . body ( "este pedido" + orderId + "pertenece al pedido móvil" ). construir ();
17    }
18 }
```

La función anterior se anota con el nombre de la función `ListOrder`. Se accede a esta función mediante una solicitud GET con el `orderId` parámetro "" pasado como parámetro de consulta en la URL

El parámetro de consulta pasado en la URL se puede obtener por el `getQueryParameters` método de `HttpRequestMessage`.

Las clases requeridas para que el código Java se ejecute en Azure Functions son del paquete Java "com.microsoft.azure.functions".

El archivo `host.json` también se genera en el proyecto Maven; entonces puede agregar la siguiente entrada:

```
1 {
2     "versión" : "2.0" ,
3     "extensionBundle" : {
4         "id" : "Microsoft.Azure.Functions.ExtensionBundle" ,
5         "versión" : "[1. *, 2.0.0)"
6     }
7 }
```

Cuando habilita paquetes, se instala automáticamente un conjunto predefinido de paquetes de extensión.

Ahora, ejecute el proyecto Maven localmente:

```
1  paquete limpio mvn
2  Funciones de mvn azure: ejecutar
```

Cuando las funciones se ejecutan localmente (configuración inicial), obtenemos esta excepción:

Causado por: java.lang. Excepción: Herramientas principales de Azure Functions no encontradas. Vaya a <https://aka.ms/azfunc-install> para instalar Azure Functions Core Tools primero. a

Solución: instale las herramientas de Azure Function-core con npm (Node Package Manager) con el siguiente comando:

```
1  npm i -g azure-functions-core-tools --unsafe -perm true
```

Pruebe la función en el entorno local y, una vez que todo esté bien, comience a publicar en Azure.

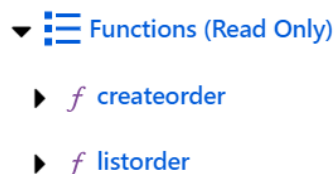
Para comenzar a publicar en Azure, inicie sesión en Azure Portal con el siguiente comando:

```
1  inicio de sesión az
```

Una vez que el inicio de sesión sea un éxito, publique e implemente el proyecto Java en Azure Functions:

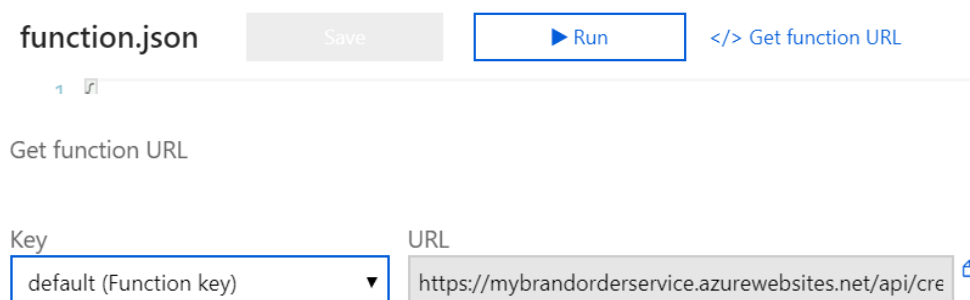
```
1  Funciones de mvn azure: desplegar
```

Una vez que se implementa el proyecto, se pueden visualizar dos funciones en el portal desde el recurso Función de Azure:



Las funciones `createorder` y `listorder`, donde hemos anotado con el nombre de la función en el código Java para el servicio POST y el servicio GET, se ven como funciones individuales separadas con el mismo nombre que el anotado.

Para obtener la URL `createorder`, haga clic en la `createorder` función y obtenga la URL haciendo clic en Obtener URL de función



El formato es el siguiente:

`https://mybrandorderservice.azurewebsites.net/api/createorder?code= <code>`

`mybrandorderservice` es el nombre de la aplicación de función que proporcionamos durante el aprovisionamiento de

Azure Functions, `createorder` es el nombre de la función que proporcionamos en el método Java y el parámetro de consulta de código es el código de seguridad generado en el nivel de Azure Functions.

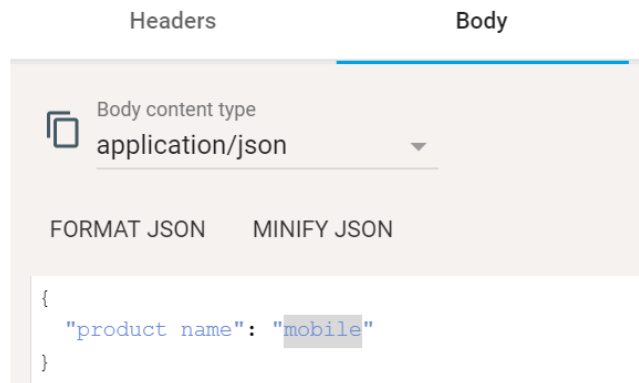
Del mismo modo, para la solicitud GET en nuestro ejemplo, el formato de URL se muestra a continuación:

`https://mybrandorderservice.azurewebsites.net/api/listorder?orderId=8976&code= <codeid>`

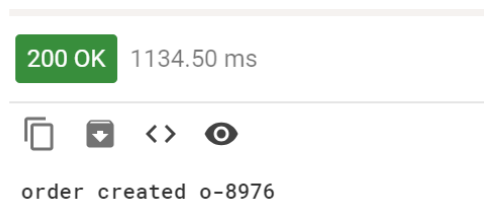
`orderId` es el parámetro de consulta que definimos para pasar un valor personalizado (en este ejemplo, el `orderId` valor).

El patrón de URL comienza con `/api` la función y se publica y se aloja como Azure Functions.

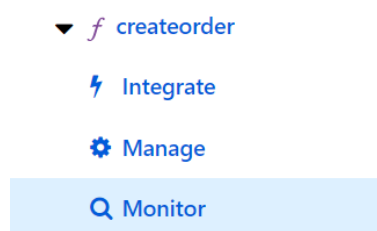
Si podemos probar nuestro servicio POST de ejemplo del cliente REST:



Recibimos la siguiente respuesta del `createorder` servicio:



Para ver los registros, navegue a la sección Monitor de la función particular en Azure Portal y vea los registros de la `createorder` función



Haga clic en el siguiente registro generado:

DATE (UTC) ▾	SUCCESS ▾	RESULT CODE ▾	DURATION (MS) ▾
2019-08-31 09:29:44.326	✓	200	1964.7582

El registro detallado se presentará desde Application Insights. Podemos ver el registro de información del código Java que se registró para la `createorder` función, como se muestra a continuación.

De esta manera, podemos monitorear cualquier error, depurar y recopilar rastros de información del proyecto Java.



DATE (UTC)	MESSAGE	LOG
2019-08-31 09:29:45.178	Executing 'Functions.createorder' (Reason='Th...	Info
2019-08-31 09:29:45.749	order details { "product name": "mobile" }	Info

Conclusión

Esperamos que haya disfrutado de esta descripción general de cómo ejecutar un proyecto Maven basado en Java y publicarlo en Azure Functions. Hemos visto cómo podemos desarrollar servicios para los métodos de solicitud POST y GET en Java, y luego alojarlo en Azure Functions con la supervisión habilitada al rastrear los detalles del registrador desde Application Insights.

Otras lecturas

Introducción a las funciones de Azure

Azure para desarrolladores de Java

Cuándo usar aplicaciones lógicas y funciones de Azure

Learn how to implement some solutions to tackle on the issues of time series forecasting at scale, including continuous accuracy evaluation and algorithm hyperparameters optimization. [Watch now!](#)

Presented by InfluxData

¿Te gusta este artículo? Leer más de DZone



Indexación y búsqueda de NuGet.org con Azure Functions and Search



Introducción a los servicios de Azure Event Grid



Creación de una aplicación de microservicio Spring Cloud nativo de Azure en Azure, parte 1



Tarjeta DZone gratis Java 13

Temas: JAVA , FUNCIONES AZULES , TUTORIAL , ORDEN DE CREACIÓN , NUBE , AZUL

Las opiniones expresadas por los contribuyentes de DZone son propias.

Pruebas unitarias para el cliente web de Spring

por Biju Kunjummen MVB · 25 de septiembre, 19 · Zona Java · Tutorial

Machine learning can help your business process and understand data insights faster. [Read this eBook](#) tricks on how to transform your data for machine learning.

Presented by Matillion



Aprenda a programar una prueba unitaria utilizando el WebClient de Spring.

WebClient , para citar su documentación de Java, es Spring Framework:

"Cliente reactivo no bloqueante para realizar solicitudes HTTP, exponiendo una API reactiva y fluida sobre bibliotecas de cliente HTTP subyacentes como Reactor Netty".

En mi proyecto actual, lo he estado utilizando `webClient` ampliamente para hacer llamadas de servicio a servicio y he encontrado que es una API increíble. Me encanta su uso de una interfaz fluida.

Considere un servicio remoto que devuelve una lista de " Cities ." Un código que usa `webClient` ve así:

```

1  ...
2  importar org . Springframework . http . Tipo de medio
3  importar org . Springframework . web . reactiva . la función . cliente . Cliente web
4  importar org . Springframework . web . reactiva . la función . cliente . bodyToFlux
5  importar org . Springframework . web . util . UriComponentsBuilder
6  la importación del reactor . núcleo . editorial . Flujo
7  Importar Java . neta . URI
8
9  class CitiesClient (
10     private val webClientBuilder : WebClient . Constructor ,
11     privada val citiesBaseUrl : Cadena
12 ) {
13
14     diversión getCities () : Flux < Ciudad > {
15         val buildUri : URI = UriComponentsBuilder

```



```

16      . fromUriString ( citiesBaseUrl )
17      . ruta ( "/" ciudades " )
18      . construir ()
19      . codificar ()
20      . toUri ()
21
22      val webClient : WebClient = esto . webClientBuilder . construir ()
23
24      volver webClient . obtener ()
25          . uri ( buildUri )
26          . aceptar ( MediaType . APPLICATION_JSON )
27          . intercambio ()
28          . flatMapMany { clientResponse ->
29              clientResponse . bodyToFlux < Ciudad > ()
30          }
31      }
32  }

```

Sin `WebClient` embargo, es difícil probar a un cliente haciendo uso de `.` En esta publicación, repasaré los desafíos para probar un cliente `WebClient` y una solución limpia.

Desafíos en la burla del cliente web

Una prueba de unidad efectiva de la `CitiesClient` clase " " requeriría burlarse de `WebClient` cada llamada de método en la cadena de interfaz fluida a lo largo de estas líneas:

```

1  val mockWebClientBuilder : WebClient . Constructor = simulacro ()
2  val mockWebClient : WebClient = mock ()
3  siempre que ( mockWebClientBuilder . build () ). thenReturn ( mockWebClient )
4
5  val mockRequestSpec : WebClient . RequestBodyUriSpec = mock ()
6  siempre que ( mockWebClient . get () ). thenReturn ( mockRequestSpec )
7  val mockRequestBodySpec : WebClient . RequestBodySpec = mock ()
8
9  siempre que ( mockRequestSpec . uri ( any < URI > () ) ). thenReturn ( mockRequestBodySpec )
10
11 siempre que ( mockRequestBodySpec . accept ( any () ) ). thenReturn ( mockRequestBodySpec )
12
13 val citiesJson : String = this . javaClass . getResource ( "/sample-cities.json" ). readText ()
14
15 val clientResponse : ClientResponse = ClientResponse
16     . crear ( HttpStatus . OK )
17     . header ("Content-Type", "application/json")
18     . body ( citiesJson ). build ()
19
20 whenever ( mockRequestBodySpec . exchange () ). thenReturn ( Mono . just ( clientResponse ) )
21
22 val citiesClient = CitiesClient ( mockWebClientBuilder, "http://somebaseurl" )
23
24 val cities : Flux < City > = citiesClient . getCities ()

```

This makes for an extremely hairy test as any change in the order of calls would result in new mocks that will need to be recorded.

Testing Using Real Endpoints

An approach that works well is to bring up a real server that behaves like the target of a client. Two mock servers that work really well are mockwebserver in okhttp library and WireMock. An example of Wiremock looks like this:

```

1  import com.github.tomakehurst.wiremock.WireMockServer
2  import com.github.tomakehurst.wiremock.client.WireMock
3  import com.github.tomakehurst.wiremock.core.WireMockConfiguration
4  import org.bk.samples.model.City
5  import org.junit.jupiter.api.AfterAll
6  import org.junit.jupiter.api.BeforeAll
7  import org.junit.jupiter.api.Test
8  import org.springframework.http.HttpStatus
9  import org.springframework.web.reactive.function.client.WebClient
10 import reactor.core.publisher.Flux
11 import reactor.test.StepVerifier
12
13 class WiremockWebClientTest {
14
15     @Test
16     fun testARemoteCall() {
17         val citiesJson = this.javaClass.getResource("/sample-cities.json").readText()
18         WIREMOCK_SERVER.stubFor(WireMock.get(WireMock.urlMatching("/cities"))
19             .withHeader("Accept", WireMock.equalTo("application/json"))
20             .willReturn(WireMock.aResponse()
21                 .withStatus(HttpStatus.OK.value())
22                 .withHeader("Content-Type", "application/json")
23                 .withBody(citiesJson)))
24
25         val citiesClient = CitiesClient(WebClient.builder(), "http://localhost:${WIREMOCK_SERVER.port()")
26
27         val cities: Flux<City> = citiesClient.getCities()
28
29         StepVerifier
30             .create(cities)
31             .expectNext(City(1L, "Portland", "USA", 1_600_000L))
32             .expectNext(City(2L, "Seattle", "USA", 3_200_000L))
33             .expectNext(City(3L, "SFO", "USA", 6_400_000L))
34             .expectComplete()
35             .verify()
36     }
37
38     companion object {
39         private val WIREMOCK_SERVER = WireMockServer(WireMockConfiguration.wireMockConfig().dynamicPort())
40
41         @BeforeAll
42         @JvmStatic
43         fun beforeAll() {
44             WIREMOCK_SERVER.start()

```

```

45     }
46
47     @AfterAll
48     @JvmStatic
49     fun afterAll() {
50         WIREMOCK_SERVER.stop()
51     }
52 }
53 }

```

Here, a server is being brought up at a random port; it is then injected with behavior, and then the client is tested against this server and validated. This approach works and there is no muddling with the internals of `WebClient` in mocking this behavior, but technically, this is an integration test and will be slower to execute than a pure unit test.

Prueba de la unidad por cortocircuito de la llamada remota

Un enfoque que he estado usando recientemente es cortocircuitar la llamada remota usando una función `Exchange`. Un `ExchangeFunction` representa los mecanismos reales para hacer la llamada remota y puede reemplazarse por uno que responda con lo que la prueba espera de la siguiente manera:

```

1  importar org . junit . Jupiter . api . Prueba
2  importar org . Springframework . http . HttpStatus
3  importar org . Springframework . web . reactiva . la función . cliente . Respuesta del cliente
4  importar org . Springframework . web . reactiva . la función . cliente . Función de intercambio
5  importar org . Springframework . web . reactiva . la función . cliente . Cliente web
6  la importación del reactor . núcleo . editorial . Flujo
7  la importación del reactor . núcleo . editorial . Mono
8  la importación del reactor . prueba . StepVerifier
9
10 class CitiesWebClientTest {
11
12     @Prueba
13     fun testCleanResponse () {
14         val citiesJson : String = this . javaClass . getResource ( "/sample-cities.json" ). readText ()
15
16         val clientResponse : ClientResponse = ClientResponse
17             . crear ( HttpStatus . OK )
18             . encabezado ( "Tipo de contenido" , "aplicación / json" )
19             . cuerpo ( ciudades Json ). construir ()
20         val shortCircuitingExchangeFunction = ExchangeFunction {
21             Mono . solo ( clientResponse )
22         }
23
24         val webClientBuilder : WebClient . Constructor = Cliente web . constructor (). exchangeFunction ( shortCircuitingExchangeFunction )
25
26         val citiesClient = CitiesClient ( webClientBuilder , "http://somebaseurl" )
27
28         val ciudades : Flux < City > = citiesClient . getCities ()
29
30         StepVerifier
31             . crear ( ciudades )
32             . expectNext ( Ciudad ( 1L , "Portland" , "EE. UU." , 1_600_000L ) )
33             . expectNext ( Ciudad ( 2L , "Seattle" , "EE. UU." , 3_200_000L ) )
34             . expectComplete ()
35             . verify ()
36     }
37 }

```

```

32         . expectNext ( Ciudad ( 2L , "SEATTLE" , "EE. UU." , 5_200_000L ))
33         . expectNext ( Ciudad ( 3L , "SFO" , "EE. UU." , 6_400_000L ))
34         . expectComplete ()
35         . verificar ()
36     }
37 }

```

Se `WebClient` inyecta un `ExchangeFunction`, que simplemente devuelve una respuesta con el comportamiento esperado del servidor remoto. Esto ha provocado un cortocircuito en toda la llamada remota y permite que el cliente sea probado exhaustivamente. Este enfoque depende de un poco de conocimiento de los aspectos internos de la `WebClient`. Sin embargo, este es un compromiso decente, ya que funcionaría mucho más rápido que una prueba con `WireMock`.

Sin embargo, este enfoque no es original; He basado esta prueba en algunas de las pruebas utilizadas para probarse a `WebClient` sí misma, por ejemplo, esta [aquí](#).

Conclusión

Personalmente prefiero el último enfoque; me ha permitido escribir pruebas unitarias bastante completas para `Client` hacer uso de `WebClient` llamadas remotas. Mi proyecto con muestras totalmente funcionales está [aquí](#).

Otras lecturas

[Spring Boot WebClient y pruebas unitarias](#)

[Pruebas de unidad e integración en Spring Boot](#)

¿Te gusta este artículo? Leer más de DZone



Esta semana en primavera: primavera 5.0 M4, AMQP y pruebas unitarias



SpringOne Platform 2016 Replay: Pruebas con Spring Framework 4.3, JUnit 5 y más allá [Video]



10 consejos para convertirse en un mejor desarrollador de Java



Tarjeta DZone gratis Java 13

Temas: [JAVA](#) , [LA UNIDAD DE PRUEBAS](#) , [LA PRIMAVERA](#) , [WEBCLIENT PRIMAVERA](#) , [WEBCLIENT](#) , [PRUEBA DE LA UNIDAD](#)

Publicado en DZone con permiso de Biju Kunjummen , DZone MVB . [Vea el artículo original aquí.](#)

Las opiniones expresadas por los contribuyentes de DZone son propias.