Mejores prácticas de Camunda

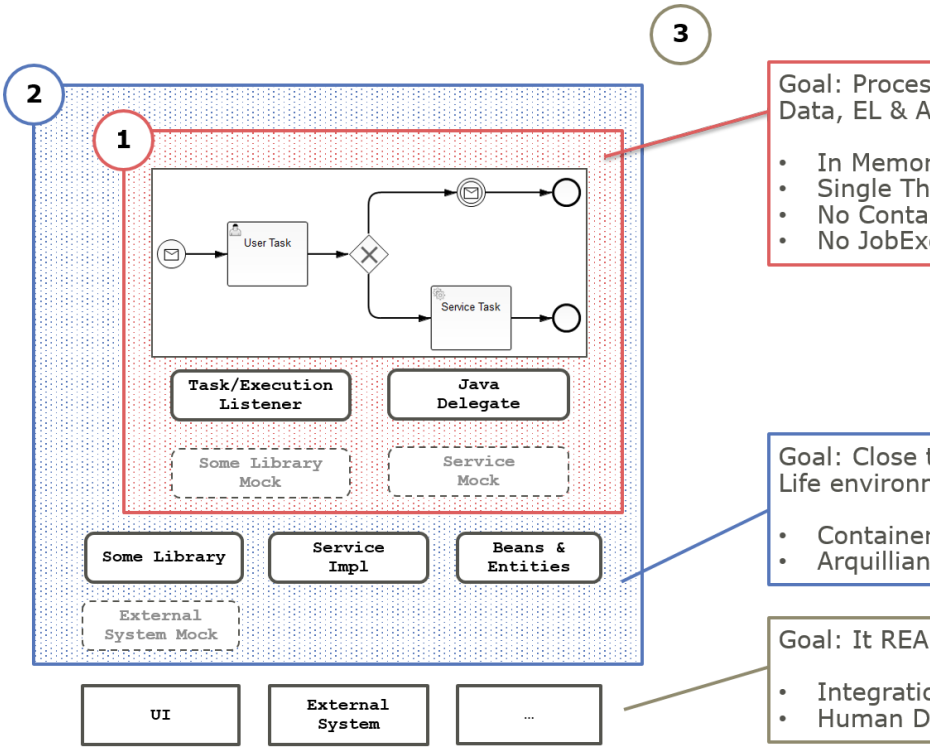
Definiciones de procesos de prueba

Pruebe sus procesos ejecutables de BPMN: son software. Si es posible, realic
unitarias automatizadas con un motor rápido en memoria (Alcance 1). Si tier
dependencias en su entorno, automatice una prueba cercana a su entorno c
(Alcance 2). Antes de su lanzamiento, verifique con pruebas de integración e
impulsadas por humanos (o un marco de automatización de pruebas) que s
"realmente funciona" (Alcance 3).

Definiciones del proceso de prueba ¿

? best-prac

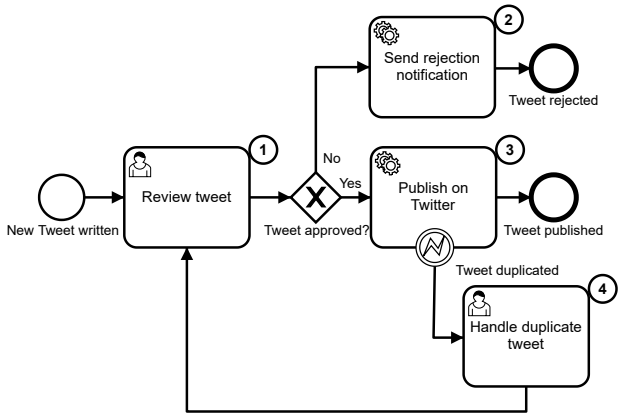
Descripción general : Pruebas de los alcanc 3



Alcance 1 : Unidad que prueba la definición proceso "más amplio"

Pruebe el comportamiento de ejecución de una definición de proceso mediante la ejecución de un **ún memoria** de prueba **sin un recipiente** . El * JobExecutor * [1] está **apagado** , se **utiliza** el **MockExpre [2]**

Para dar un ejemplo, ahora probamos el **proceso de aprobación de Tweet** , un proceso de ejemplo s en diversas situaciones.



1. Los nuevos tweets deben revisarse antes de su publicación.
2. El empleado que tuitea es notificado sobre los tweets rechazados.
3. Los tweets aprobados se publican.
4. Los tweets duplicados deben ser tratados, por ejemplo, reformulados, y luego revisados nuev

1.1. Configure su prueba con las clases y bibliotecas de recomendadas

1. Use **JUnit** [3] como marco de prueba de unidad.
2. Use Camunda * JUnit Rule * [4] para acelerar un motor de proceso en memoria.
3. Use la anotación Camunda * @ Deployment * [5] para implementar y cancelar la implementa definiciones de proceso bajo prueba para un solo método de prueba.
4. Use Camunda **Assert** [6] para verificar fácilmente si se cumplen sus expectativas sobre el esta
5. Use la burla de su elección, por ejemplo, **Mockito** [7] más **PowerMock** [8] para burlarse de lo servicio y verificar que los servicios se llamen como se esperaba.
6. Use Camunda * MockExpressionManager * [2] para resolver los nombres de beans utilizados su proceso sin la necesidad de aumentar el marco de inyección de dependencias (como CDI o

```
// ...
import static org.camunda.bpm.engine.test.assertions.ProcessEngineTests.*; (4)
import static org.mockito.Mockito.*; (5)

@RunWith(PowerMockRunner.class)  (1) (5)
public class TwitterTest {

  @Rule
  public ProcessEngineRule processEngineRule = new ProcessEngineRule(); (2)

  @Mock // Mockito mock instantiated by PowerMockRunner (5)
  private TweetPublicationService tweetPublicationService;

  @Before
  public void setup() {
        // ...
    Mocks.register("tweetPublicationDelegate", tweetPublicationDelegate); (6)
  }

  @Test (1)
  @Deployment(resources = "twitter/TwitterDemoProcess.bpmn") (3)
  public void testTweetApproved() {
        // ...
  }
// ...
}
```

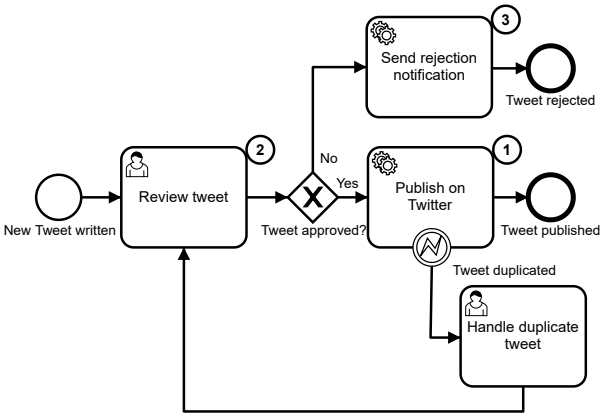
Por último, utilice una **base de datos In-Memory H2** [9] como base de datos predeterminada para pr máquinas de desarrollador.

Si es necesario, puede ejecutar las mismas pruebas en **varias bases de datos** , por ejer MS-SQL, ... en un servidor CI (por ejemplo , Jenkins [10] , ...). Para lograrlo, puede utilizar ejemplo, maven) y **propiedades** (archivos) de Java para la configuración de la base de d

Ahora echemos un vistazo más profundo a las partes de esta prueba de definición de proceso.

1.2. Concéntrese en probar la definición del proceso "r amplio", pero no más

Con el alcance 1, queremos probar la **definición de proceso más amplia** : esta es la definición de prc
ejecutable en sentido estricto más todo el código de cableado que todavía "pertenece" a la definición
sentido más amplio:



Considere el lenguaje de expresión (como, por ejemplo, JUEL) y la lógica del adaptador (como, por ejer
delegado de Java) como parte de esta definición de proceso "más amplia". Se puede hacer referencia a
cosas en el BPMN XML:

1. Una tarea de servicio normalmente llama a un **delegado de Java** o una **expresión** (por ejemp
2. Un **oyente de tareas que** envía un correo electrónico al jefe puede definirse detrás de la tare
3. Un **escucha de ejecución que** registra el correo de rechazo en una carpeta puede definirse d
servicio.

Considere que los servicios que ejecutan el código de "negocio" independiente del motor de proceso y
definición del proceso en un sentido más amplio.

1.3. Burlarse de los métodos de servicio comercial

Se burlan de todo lo que no pertenece a la definición de proceso "más amplia" explicada anteriormen
método de servicio comercial llamado por un **delegado de Java** . Considere la tarea de servicio "Publi
delega al código Java:

```
<serviceTask id="service_task_publish_on_twitter" camunda:delegateExpression="#{twe
elegate}" name="Publish on Twitter">
</serviceTask>
```

Y este **delegado de Java en** sí mismo llama a otro método de "servicio comercial":

```
@Named
public class TweetPublicationDelegate implements JavaDelegate {

  @Inject
  private TweetPublicationService tweetPublicationService;

  public void execute(DelegateExecution execution) throws Exception {
    String tweet = new TwitterDemoProcessVariables(execution).getTweet(); (1)
    // ...
    try {
      tweetPublicationService.tweet(tweet); (2)
    } catch (DuplicateTweetException e) {
      throw new BpmnError("duplicateMessage"); (3)
    }
  }
// ...
```

(1) La recuperación de la variable de proceso pertenece al **código de delegado de cableado** , po de la definición de proceso "más amplia" y **no se burla** . (Para obtener una explicación de la c variable utilizada aquí, consulte Manejo de datos en procesos)

(2) Este método ejecuta **código de "negocio"** independiente del motor de proceso , por lo tanto, de la definición de proceso "más amplia" y **debe ser burlado** .

(3) La excepción específica del motor de proceso generalmente no es producida por su método c Por lo tanto, debemos **traducir la excepción comercial** a la excepción necesaria para impuls nuevamente, el código es parte de la definición de proceso "más amplia" y **no se burla** .

Veamos ahora cómo está conectada la burla en nuestra clase de prueba:

```
@Mock (1)
private TweetPublicationService tweetPublicationService;

@Before
public void setup() {
  // set up java delegate to use the mocked tweet service
  TweetPublicationDelegate tweetPublicationDelegate = new TweetPublicationDelegate
  tweetPublicationDelegate.setTweetService(tweetPublicationService);
  // register a bean name with mock expression manager
  Mocks.register("tweetPublicationDelegate", tweetPublicationDelegate); (3)
}

@After
public void teardown() {
  Mocks.reset(); (3)
}
```

(1) El simulacro anotado se instancia automáticamente (por PowerMockRunner).

(2) Java Delegate está preparado para trabajar con este servicio simulado.

(3) El delegado de Java se registra con el nombre del bean utilizado en la definición del proceso (p MockExpressionManager) y el registro se limpia después de cada prueba.

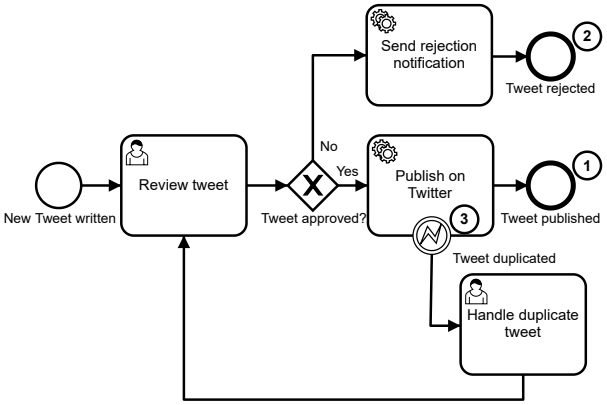
Por favor no!
Para enfatizarlo en otro momento: evite ejecutar métodos reales de servicio comercial c
1.

1.4. Conduzca el proceso y afirme el estado

Ahora **conduzca** el proceso de estado de espera a estado de espera y **afirme** que ve el proceso esper
variables. Divide y vence **probando tu proceso en trozos** .

- Pruebe completamente el **Happy Path** en un método de prueba (grande), ya que esto asegura qu
datos consistente en su proceso. Además, es fácil de leer y entender, lo que lo convierte en un exc
partida para que los nuevos desarrolladores entiendan su proceso / caso de prueba de proceso.
- Pruebe **bifurcaciones / desvíos** de la ruta feliz, así como **errores /** patches **excepcionales** como
métodos de prueba separados (más pequeños). Esto permite realizar pruebas unitarias en unidad

Considere los fragmentos importantes y las fallas que el proceso de aprobación de Tweet consiste en:



1. Puede ser que el tweet solo se publique. El **camino feliz** ¡☺
2. También puede ser que el tweet sea rechazado. ¡El empleado que tuitea tiene que vivir con es
3. También puede suceder que un tweet duplicado sea rechazado por Twitter. ¡Una ruta de erro
ocurre!

Para procesos más grandes, decida conscientemente si desea probar el **Happy Path** co
prueba de unidad larga o no. Por un lado, una prueba de unidad larga puede ser más
asegura que las variables / flujo de datos funcionen para esa ruta. Por otro lado, si dese
enfoque de prueba de unidad más "purista", pruebe también el camino feliz en trozos. I
absolutamente crucial que afirme las variables esperadas / estado de datos en los bord
fragmentos.

1.4.1. Prueba el camino feliz

El método de prueba **testTweetApproved ()** prueba la **ruta "feliz"** a un tweet publicado:

```
@Test
@Deployment(resources = "twitter/TwitterDemoProcess.bpmn")
public void testTweetApproved() {
  // given
  ProcessInstance processInstance = runtimeService().startProcessInstanceByKey(
    "TwitterDemoProcess",
    withVariables(TwitterDemoProcessConstants.VAR_NAME_TWEET, TWEET)); (1)
  assertThat(processInstance).isStarted();
  // when
  complete(task(), withVariables(TwitterDemoProcessConstants.VAR_NAME_APPROVED, tru
  // then
  assertThat(processInstance) (3)
    .hasPassed("end_event_tweet_published")
    .hasNotPassed("end_event_tweet_rejected")
    .isEnded();
  verify(tweetPublicationService).tweet(TWEET); (4)
  verifyNoMoreInteractions(tweetPublicationService);
}
```

(1) Crea una nueva instancia de proceso. Es posible que desee utilizar un submétodo para iniciar proceso, como se describe en la siguiente sección.

(2) Conduzca el proceso a su siguiente estado de espera, por ejemplo, completando una tarea de Puede usar métodos convenientes proporcionados por camunda-bpm-afirmar, pero también usar directamente la API del motor de procesos.

(3) Afirma que tu proceso está en el estado esperado.

(4) Verifique con su biblioteca burlona que sus métodos de servicio comercial se llamaron como

Tenga cuidado de no "sobreespecificar" su método de prueba afirmando demasiado. La proceso probablemente evolucionará en el futuro y dichos cambios deberían romper el prueba posible, ¡pero tanto como sea necesario! Como regla general, **siempre** afirme q **externos** esperados de su proceso realmente tuvieron lugar (por ejemplo, que los servi se llamaron como se esperaba). Además de eso, elija cuidadosamente qué aspectos del **proceso interno** son lo suficientemente importantes como para que desee que su mét advierta sobre cualquier cambio relacionado más adelante.

1.4.2. Crear submétodos para iniciar la instancia de proceso bajo prueba

Al probar **fragmentos** , es una buena práctica implementar submétodos para **navegar el proceso ha** necesarios para varios de sus métodos de prueba como nodo (s) de inicio. Aquí verá uno de estos sub simplemente crea una nueva instancia de proceso en su evento de inicio:

```
// create a new process instance
ProcessInstance newTweet(Map<String, Object> variables) {
  ProcessInstance processInstance = runtimeService().startProcessInstanceByKey( (1)
    "TwitterDemoProcess", variables
  );
  assertThat(processInstance) (2)
    .isStarted()
    .hasVariables(TwitterDemoProcessConstants.VAR_NAME_TWEET);
  return processInstance;
}
```

(1) **Cree una nueva** instancia de **proceso** (aquí "por clave") e **inicialice** algunas **variables de pro**

(2) Al final del submétodo, considere afirmar que deja el proceso en el estado esperado.

¡Y aquí ves un segundo submétodo que crea una nueva instancia de proceso **justo en el medio del p**

```
// create a process instance directly at the point at which a tweet was rejected
ProcessInstance rejectedTweet(Map<String, Object> variables) {
  ProcessInstance processInstance = runtimeService()
    .createProcessInstanceByKey("TwitterDemoProcess") (1)
    .startBeforeActivity("service_task_publish_on_twitter")
    .setVariables(variables)
  .execute();
  assertThat(processInstance) (3)
    .isStarted()
    .hasPassed("service_task_publish_on_twitter")
    .hasVariables(TwitterDemoProcessConstants.VAR_NAME_TWEET);
  return processInstance;
}
```

(1) **Cree una** instancia de **proceso modificada** por clave e **inicialice** algunas **variables de proce**

(2) Al final del submétodo, considere afirmar que deja el proceso en el estado esperado.

Como se muestra en el ejemplo, utilizamos la modificación de instancia de proceso [11] implementar dichos métodos de inicio. Esto permite **probar** fácilmente **los procesos e** como se muestra en la siguiente sección.

1.4.3. Probar las manchas excepcionales en trozos

Hay dos parches excepcionales que probamos como fragmentos en este ejemplo:

1. El método de prueba **testTweetRejected ()** prueba la ruta a un tweet rechazado, los mismos cua discutieron ocurren nuevamente, esta vez, la tarea del usuario se completa con un rechazo de twe

```
@Test
@Deployment(resources = "twitter/TwitterDemoProcess.bpmn")
public void testTweetRejected() {
  // given
  ProcessInstance processInstance = newTweet(withVariables(TwitterDemoProcessConsta
WEET, TWEET)); (1)
  // when
  complete(task(), withVariables(TwitterDemoProcessConstants.VAR_NAME_APPROVED, fa
  // then
  assertThat(processInstance) (3)
    .hasPassed("end_event_tweet_rejected")
    .hasNotPassed("end_event_tweet_published")
    .isEnded();
  verifyZeroInteractions(tweetPublicationService); (4)
}
```

2. El método de prueba **testTweetDuplicated ()** prueba lo que sucede en caso de que un tweet resu y sea rechazado por Twitter. Para este caso, adjuntamos un evento de error a la tarea de servicio " En el XML BPMN vemos un evento de error definido con un código de error "mensaje duplicado".

```
  <boundaryEvent id="boundary_event_tweet_duplicated" name="Tweet duplicated" atta
ice_task_publish_on_twitter">
    <errorEventDefinition id="error_event_definition_tweet_duplicated" errorRef="e
icated"/>
  </boundaryEvent>
<error id="error_tweet_duplicated" errorCode="duplicateMessage" name="Tweet duplic
```

Arriba, ya vimos el código de delegado de Java lanzando la expulsión de BpmnError con ese código "du aquí viene el método de prueba para el caso de que un tweet esté duplicado:

```
@Test
@Deployment(resources = "twitter/TwitterDemoProcess.bpmn")
public void testTweetDuplicated() {
  // given
  doThrow(new DuplicateTweetException()) (1)
    .when(tweetPublicationService).tweet(anyString());
  // when
  ProcessInstance processInstance = rejectedTweet(withVariables(TwitterDemoProcess
AME_TWEET, TWEET)); (2)
  // then
  assertThat(processInstance) (3)
    .hasPassed("boundary_event_tweet_duplicated")
    .hasNotPassed("end_event_tweet_rejected").hasNotPassed("end_event_tweet_publis
    .isWaitingAt("user_task_handle_duplicate");
  verify(tweetPublicationService).tweet(TWEET); (4)
  verifyNoMoreInteractions(tweetPublicationService);
  // when
  complete(task()); (5)
  // then
  assertThat(processInstance) (6)
    .isWaitingAt("user_task_review_tweet")
    .hasVariables(TwitterDemoProcessConstants.VAR_NAME_TWEET)
    .task().isAssignedTo("demo");
}
```

(1) Inicialice su método de servicio comercial burlado para lanzar la excepción comercial destinad

(2) Cree una nueva instancia de proceso llamando a un submétodo que comience el proceso "jus como ya se mostró anteriormente .

(3) Afirma que tu proceso está en el estado esperado. Nuevamente, tenga cuidado de no "sobree método de prueba afirmando demasiado, pero hay un área gris de lo que considera "lo sufici

(4) Verifique con su biblioteca burlona que sus métodos de servicio comercial se llamaron como

(5) Puede seguir adelante y decidir probar aún más dentro de ese método. Aquí completamos la tweet duplicado" en orden ...

(6) ... Para afirmar que el proceso vuelve nuevamente a la tarea "Revisar tweet" y, por ejemplo, q se asigna al usuario esperado, etc.

Para que su código de prueba sea más legible, use convenciones de nomenclatura amig desarrolladores para ids (vea Nombrar ID técnicamente relevantes). Si tiene muchas de proceso para probar, considere generar clases constantes (por ejemplo, XSLT) directam XML.

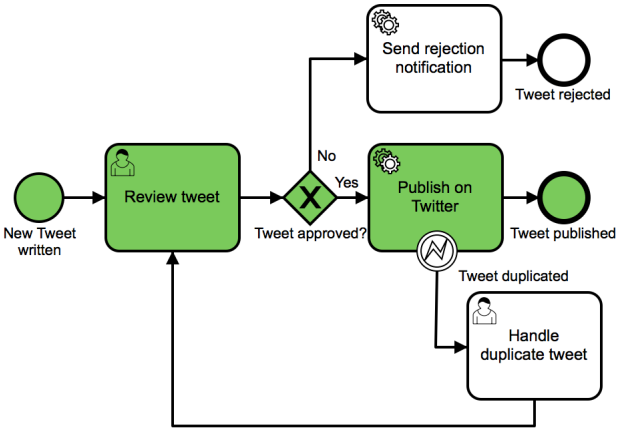
1.5. Controle su cobertura de prueba de proceso

Apunte a **una cobertura de prueba de nodo de flujo del 100%** cuando pruebe las definiciones de p
1. Eso significa que, básicamente, todos los "nodos de flujo" (por ejemplo, tareas, puerta de enlace, ev
por al menos un caso de prueba. Normalmente **no** apuntamos a una cobertura de ruta del 100% (lo q
prueban todas las posibles rutas a través del modelo), ya que esto es simplemente un gran esfuerzo.
correctamente en fragmentos es suficiente.

Considere aprovechar la herramienta **visual** Cobertura de prueba de proceso [12] , actualmente dispc
fragmento de consultoría. Agregue las siguientes líneas a su clase de prueba:

```
@After
public void calculateProcessTestCoverage() {
  ProcessTestCoverage.calculate(processEngine());
}
```

Imagine que acaba de implementar la prueba para la ruta feliz, luego su archivo de cobertura de prue
generado en `target/process-test-coverage/TwitterDemoProcess.html` se vería de la siguiente m



Cuando se busca una **cobertura de prueba de nodo de flujo del 100%,** todas las tareas, puertas de
deben ser verdes antes de dejar de escribir métodos de prueba. En nuestro ejemplo, después de habe
tres métodos de prueba mostrados anteriormente, nuestra definición de proceso está completament
diagrama de cobertura de prueba de proceso ahora está coloreado en verde .

La herramienta también puede mostrarle los parches / fragmentos específicos que prue
métodos de prueba individuales. Consulte todos los detalles en el repositorio de Proces
[12] GitHub.

Alcance 2 : El entorno de la "vida real"

Pruebe el proceso cerca de un entorno de la vida real mediante la ejecución de una prueba **en memo
contenedor** , que es potencialmente **multiproceso** .

Ahora desea tener su entorno disponible, como beans (p. Ej. CDI, Spring ...), transaccion
utilizando Spring, es completamente natural que tenga una configuración propia de Spr
pruebas. Cuando utilice Java EE, considere conducir sus pruebas con Arquillian [13] o ur
similar. Para facilitar la agrupación y el control de versiones de sus pruebas junto con su
producción, considere usar el aprovisionamiento de contenedores con Docker [14] .

Configure sus pruebas para que sean pruebas de integración dedicadas. Invoquelos por separado de alcance 1 (generalmente una ejecución mucho más rápida).

Por favor no!
Evite apagar el * JobExecutor * [1] . De forma predeterminada, está **activado** y lo dejam
2. También evite usar **MockExpressionManager** . De forma predeterminada, **no se us**
recomienda para probar en Scope 1. **[2]**

Alcance 3 : "Realmente funciona"

Compruebe que "realmente funciona" antes de lanzar una nueva versión de su definición del proceso, **impulsada por humanos** , **exploratorios** pruebas.

¡ **Defina** claramente **sus objetivos** para el alcance 3! Los objetivos pueden ser

- usuario final y pruebas de aceptación,
- pruebas completas de extremo a extremo,
- pruebas de rendimiento y carga, etc.

Considere cuidadosamente **automatizar las** pruebas en el alcance 3. Debe observar el esfuerzo gene escribir código de automatización de pruebas y mantenerlo, en comparación con la ejecución de prue humanos para la vida útil de su proyecto de software. ¡La mejor opción depende mucho de la frecuen regresión!

La mayor parte del esfuerzo generalmente se invierte en configurar datos de prueba adecuados en lo circundantes.

Mire JMeter [15] para pruebas de carga, SoapUI [16] para pruebas funcionales de servici para pruebas frontend y TestLink [18] para descripciones de escenarios de prueba. Tam usar una pila de JavaScript para las pruebas frontend: utilizamos Mocha [19] , Chai [20] , Karma [22] , Protractor [23] .

Links

[1] https://docs.camunda.org/manual/7.11/user-guide/process-engine/the-job-executor/
[2] https://docs.camunda.org/manual/7.11/reference/javadoc/?org/camunda/bpm/engine/test/mock/MockExpressionManager.html
[3] http://junit.org
[4] https://docs.camunda.org/manual/7.11/reference/javadoc/?org/camunda/bpm/engine/test/Proc
[5] https://docs.camunda.org/manual/7.11/reference/javadoc/?org/camunda/bpm/engine/test/Depl
[6] http://github.com/camunda/camunda-bpm-assert
[7] http://mockito.org
[8] https://github.com/jayway/powermock/
[9] http://www.h2database.com/html/features.html#in_memory_databases
[10] https://jenkins-ci.org/
[11] https://docs.camunda.org/manual/7.11/user-guide/process-engine/process-instance-modificati
[12] https://github.com/camunda/camunda-consulting/tree/master/snippets/process-test-coverage
[13] http://arquillian.org/
[14] https://www.docker.com/
[15] http://jmeter.apache.org/
[16] http://www.soapui.org/
[17] http://www.seleniumhq.org/
[18] http://testlink.org/
[19] http://mochajs.org/
[20] https://github.com/chaijs/chai
[21] http://gruntjs.com/

[22] http://karma-runner.github.io
[23] https://angular.github.io/protractor

Descargo de responsabilidad y derechos de autor

Sin garantía : las declaraciones hechas en esta publicación son recomendaciones basadas en la experiencia p
autores. No forman parte de la documentación oficial del producto de Camunda. Camunda no puede aceptar
responsabilidad por la exactitud o puntualidad de las declaraciones realizadas. Si se muestran ejemplos de có
puede garantizar una ausencia total de errores en el código fuente proporcionado. Se excluye la responsabilic
daño resultante de la aplicación de las recomendaciones presentadas aquí.

Printed November 19, 2019. Applies to Camunda **7.11**. Any feedback? **best-practices@camur**