



Flujos reactivos en Java 9

por Justin Albano MVB · 10 y 18 de febrero · Zona de Java

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

El lanzamiento de Java Development Kit (JDK) 9 en septiembre del año pasado trajo consigo muchas mejoras tanto para Java Virtual Machine (JVM) como para Java Language Specification (JLS). La principal de estas adiciones fue la inclusión de JDK Enhancement Proposal (JEP) 261: The Module System (apodado Project Jigsaw), que introdujo una de las revisiones más completas de la JVM desde su creación. Detrás de toda la fanfarria, también había 85 JEP no modulares contenidos en el lanzamiento, incluida la adición de un cliente del protocolo de transferencia de hipertexto (HTTP) 2 (JEP 110), jshell (JEP 222) y actualizaciones de concurrencia importantes (JEP 266).) Dentro de estas actualizaciones de concurrencia, se realizó una mejora importante en las bibliotecas estándar de Java: Flujos reactivos.

La iniciativa Reactive Streams se inició en 2013 por algunas de las empresas de aplicaciones web más influyentes (incluida Netflix) como medio para estandarizar el intercambio de datos asincrónicos entre los componentes de software. A medida que más frameworks y constructores de bibliotecas apoyaron esta iniciativa y su especificación, varias implementaciones de Java de este estándar comenzaron a aparecer. Debido a la simplicidad del estándar Reactive Stream, muchas de las interfaces que conformaron estas implementaciones fueron idénticas, separadas solo por sus nombres de paquete.

Para reducir esta duplicación e incompatibilidad de importación, Java 9 ahora incluye interfaces básicas para cada uno de los conceptos fundamentales de Flujo reactivo en la biblioteca de Concurrencia de flujo. Esto permite que todas las aplicaciones Java dependan de esta biblioteca para las interfaces de flujo reactivo, en lugar de decidir sobre una implementación específica.

En este artículo, exploraremos el estándar Reactive Streams y su implementación en Java. Si bien hay muchos marcos de flujos de reacción ricos, como ReactiveX , no nos centraremos en estas implementaciones de terceros. En su lugar, profundizaremos en la implementación oficial de Java de este estándar y cómo esta implementación se puede usar para crear aplicaciones que respondan a flujos de datos asíncronos.

Cabe señalar que el objetivo de este conjunto común de interfaces reactivas oficiales de Java es consolidar las distintas interfaces reactivas en una única ubicación y no está destinado a implementaciones independientes o personalizadas. La creación de una implementación personalizada del estándar Reactive Streams puede ser propensa a errores y debe ser verificada por el Kit de Compatibilidad de Tecnología de Flujos Reactivos (TCK) . Si un desarrollador desea usar Flujos reactivos en su aplicación, debe usar una implementación existente del estándar, como ReactiveX o Akka.. Del mismo modo, los ejemplos en este artículo son pedagógicos y están destinados solo a fines de demostración; no deben usarse en aplicaciones de producción (en su lugar, se debe usar una implementación de Flujos reactivos bien probada).

¿Qué son las corrientes reactivas?

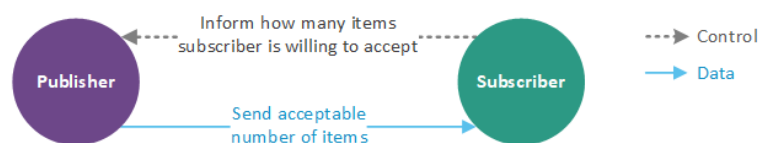
En muchas aplicaciones, los datos no se recuperan de un dispositivo de almacenamiento fijo, sino que se manejan casi en tiempo real, con usuarios u otros sistemas que inyectan información rápidamente en nuestro sistema. La mayoría de las veces esta inyección de datos es asíncrona, en la que no

sabemos con anticipación cuándo estarán presentes los datos. Para facilitar este estilo asíncrono de manejo de datos, tenemos que replantear los modelos anteriores basados en encuestas y, en cambio, usar un método más ligero y más sencillo.

Editores, suscriptores y suscripciones

Aquí es donde las corrientes reactivas entran en juego. En lugar de tener un estilo de manejo de datos de cliente y servidor, cuando un cliente solicita datos de un servidor y el servidor responde, posiblemente de forma asíncrona, al cliente con los datos solicitados, en su lugar utilizamos un mecanismo publicar-suscribir: un **suscriptor** informa a un **editor** que está dispuesto a aceptar un número determinado de **elementos** (**solicita** un número determinado de elementos), y si los artículos están disponibles, el editor empuja la cantidad máxima de artículos que se pueden recibir al suscriptor. Es importante tener en cuenta que se trata de una comunicación bidireccional, donde el suscriptor informa al editor cuántos artículos está dispuesto a manejar y el editor envía esa cantidad de artículos al suscriptor.

El proceso de restringir el número de elementos que un suscriptor está dispuesto a aceptar (como lo juzga el suscriptor) se denomina **contrapresión** y es esencial para prohibir la sobrecarga del suscriptor (empujando más elementos que el suscriptor puede manejar). Este esquema se ilustra en la figura a continuación.

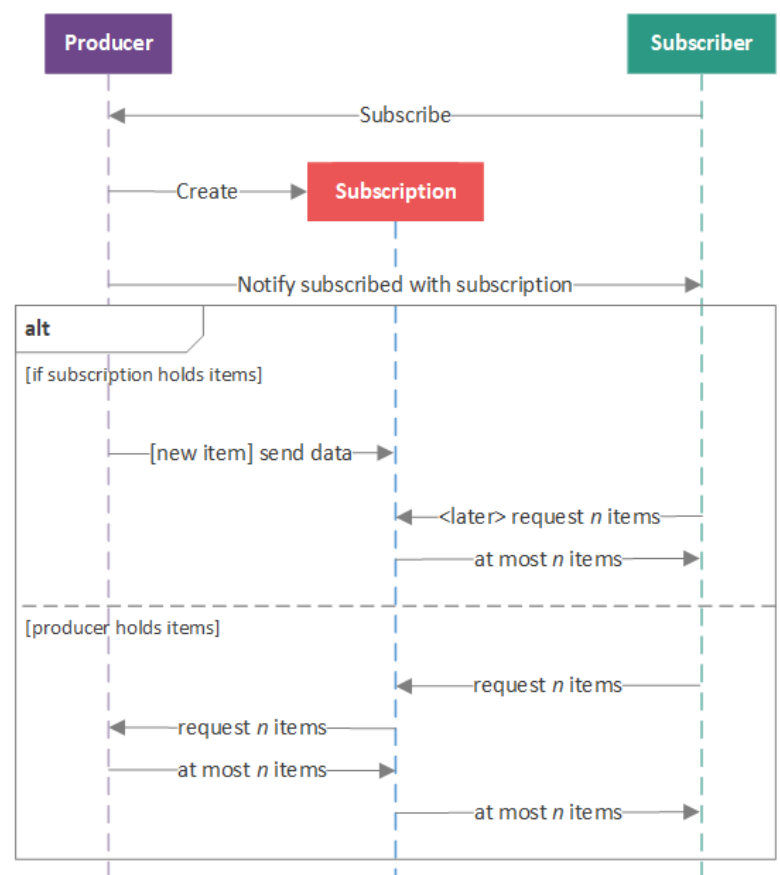


Esta conexión bidireccional entre un editor y un suscriptor se llama **suscripción**. Esta suscripción vincula a un solo editor a un único suscriptor (relación de uno a uno) y puede ser unidifusión o multidifusión. Del mismo modo, un solo editor puede tener suscriptores múltiples suscritos, pero un solo suscriptor solo puede estar suscrito a un solo

productor (un editor puede tener muchos suscriptores, pero un suscriptor puede suscribirse a lo sumo un editor).

Cuando un suscriptor se suscribe a un editor, el editor notifica al suscriptor de la suscripción que se creó, permitiendo que el suscriptor almacene una referencia a la suscripción (si lo desea). Una vez que se completa este proceso de notificación, el suscriptor puede informarle al editor que está listo para recibir algunos n artículos.

Cuando el editor tiene elementos disponibles, envía como máximo n artículos al suscriptor. Si se produce un error en el editor, indica al suscriptor del **error**. Si el editor finaliza el envío de datos de forma permanente, indica al suscriptor que está **completo**. Si se notifica al suscriptor que se ha producido un error o que el editor está completo, la suscripción se considera **cancelada** y no se realizarán más interacciones entre el editor y el suscriptor (o la suscripción). Este flujo de trabajo de suscripción se ilustra en la figura a continuación.



Es importante tener en cuenta que hay dos enfoques teóricos para transmitir datos a un suscriptor: (1) la suscripción contiene los artículos o (2) el editor tiene

los artículos. En el primer caso, el editor empuja los artículos a la suscripción cuando están disponibles; cuando, en un momento posterior, el suscriptor solicita n elementos, la suscripción proporciona n o menos elementos que haya sido previamente entregados por el editor. Esto se puede usar cuando el editor gestiona elementos en cola, como las solicitudes HTTP entrantes. En el segundo caso, el suscriptor reenvía las solicitudes al editor, lo que empuja *no* menos elementos a la suscripción, que a su vez empuja esos mismos artículos al suscriptor. Este escenario puede ser más adecuado para instancias donde se generan elementos según sea necesario, como con un generador de números primos.

También es importante tener en cuenta que los artículos no tienen que estar presentes antes de que se pueda realizar una solicitud. Si un suscriptor realiza una solicitud de n elementos y no hay elementos disponibles, el suscriptor esperará hasta que haya al menos un artículo disponible y se envíe al suscriptor. Si hay i artículos disponibles cuando el abonado hace una solicitud de n elementos, donde i es menor que n , los i elementos son empujados al abonado. Una vez j más artículos están disponibles, $n - i$ artículos de los j elementos también son empujados al abonado hasta n número de elementos totales han sido empujados al abonado ($i + j = n$), o el suscriptor ha solicitado m más elementos; en este caso, todos los j ítems pueden ser enviados al suscriptor siempre que $i + j$ sea menor o igual a $n + m$. La cantidad de elementos que un suscriptor puede aceptar en cualquier momento dado (que puede o no ser igual a n , dependiendo de la cantidad de elementos que ya haya enviado al suscriptor) se denomina **demanda pendiente**.

Por ejemplo, supongamos que un suscriptor solicita 5 elementos y 7 están actualmente disponibles en el editor. La demanda pendiente para el suscriptor es 5, por lo que 5 de los 7 elementos se envían al suscriptor. Los 2 elementos restantes son mantenidos por el editor, esperando que el suscriptor solicite más artículos. Si el suscriptor solicita 10 elementos más, los 2 elementos restantes se envían al suscriptor, lo que da como resultado una demanda destacada de 8.

Si hay n elementos más disponibles en el editor, estos

Si hay 5 elementos mas disponibles en el editor, estos 5 elementos se envían al editor, lo que deja una demanda excepcional. de 3. La demanda pendiente se mantendrá en 3 a menos que el suscriptor solicite n más artículos, en cuyo caso la demanda pendiente aumentará a $3 + n$, o más i. los artículos se envían al suscriptor, en cuyo caso la demanda pendiente disminuirá a $3 - i$ (a un mínimo de 0).

Procesadores

Si una entidad es a la vez editor y suscriptor, se llama **procesador**. Un procesador comúnmente actúa como un intermediario entre otro editor y suscriptor (cualquiera de los cuales puede ser otro procesador), realizando alguna transformación en el flujo de datos. Por ejemplo, se puede crear un procesador que filtra elementos que coinciden con algunos criterios antes de pasarlos a su suscriptor. En la siguiente figura se ilustra una representación visual de un procesador.



Con una comprensión fundamental de cómo operan los flujos reactivos, podemos transformar estos conceptos en el reino de Java codificándolos en interfaces.

Representación de interfaz

Dada la descripción anterior, los flujos reactivos se componen de cuatro entidades principales: (1) editores, (2) suscriptores, (3) suscripciones y (4) procesadores. Desde una perspectiva de interfaz, los editores solo están obligados a permitir que los suscriptores se suscriban. Por lo tanto, podemos crear una interfaz simple para un editor, donde el parámetro de tipo genérico formal τ , representa el tipo de elementos que produce el editor:

```

1 public interface Publisher < T > {
2     public void suscribirse ( Subscriber <?
3 }
  
```

Esta definición de interfaz requiere que

Esta definición de interfaz requiere que posteriormente definamos la interfaz para un suscriptor. Como se indicó anteriormente, un suscriptor tiene cuatro interacciones principales: (1) notificación de suscripción, (2) aceptación de elementos enviados, (3) aceptación de errores que ocurren en un editor suscrito, y (4) notificación cuando un editor está completo. Esto da como resultado la siguiente interfaz, que también se parametriza por el tipo de elementos que solicita:

```

1  interfaz pública Suscriptor < T > {
    public void onSubscribe ( Subscription s );
2  vacío público onNext ( T t );
3  pública vacío onError ( Throwable t );
4  public void onComplete ();
5
6  }
```

A continuación, debemos definir la interfaz para una suscripción. Esta entidad es más simple que un suscriptor y es responsable de solo dos acciones: (1) aceptar solicitudes de artículos y (2) cancelarse. Esto da como resultado la siguiente definición de interfaz:

```

1  suscripción a la interfaz pública {
    solicitud de nulidad pública ( n largo );
2  anulación de anulación pública ();
3
4  }
```

Por último, definimos un procesador como una combinación de las interfaces del editor y del suscriptor, con una peculiaridad importante: un procesador puede producir elementos de un tipo diferente al tipo de los elementos que consume. Por lo tanto, utilizaremos el parámetro de tipo genérico formal τ para representar el tipo de ítems que el procesador consume y R para representar el tipo de ítems que devuelve (o produce). Tenga en cuenta que una implementación de un productor puede consumir y producir elementos del mismo tipo, pero no existe una restricción en tiempo de compilación que *deba* hacerlo. Esto da como resultado la siguiente interfaz:

```

1 El procesador de interfaz pública < T , R > ε

```

Si bien estas cuatro interfaces constituyen el contrato codificado para Flujos reactivos, existen otras restricciones y comportamientos previstos a los que estas interfaces deben ajustarse. Estas especificaciones, junto con las definiciones de interfaz anteriores, se pueden encontrar en la especificación JVM de Flujos reactivos . Como veremos en la siguiente sección, la implementación Java estándar de la especificación de Flujo reactivo es casi idéntica a la de la especificación JVM de Flujos reactivos y actúa como una estandarización de los contratos de Flujos reactivos dentro de la Biblioteca estándar de Java .

¿Cómo funcionan las corrientes reactivas en Java?

El puerto Java estándar de las interfaces de Flujos reactivos se encuentra en la clase `java.util.concurrent.Flow` y se agrupan como interfaces estáticas dentro de la `Flow` clase. Con los JavaDocs eliminados, la `Flow` clase se define de la siguiente manera:

```

1  clase final  pública Flow {
2
3      private  Flow () {} // desinstalable
4
5      @FunctionalInterface
6      public static interface Publisher < T >
7      {
8          public void suscribirse ( suscriptor
9
10         interfaz estática  pública Suscriptor < T
11         public void onSubscribe ( Suscripció
12         vacío  público onNext ( T artículo );
13         pública vacío onError ( Throwable tr

```



```

14         public void onComplete ();
15     }
16
17     suscripción a la interfaz pública estática
18     solicitud de nulidad pública ( n larg
19     anulación de anulación pública ();
20 }
21
22 El procesador de interfaz pública estática
23 }

```

Si bien no hay mucho nuevo que discutir cuando se compara la especificación de reactivos corrientes JVM a las definiciones estándar de Java, la versión estándar de Java sí incluye una implementación editor: `SubmissionPublisher`. La

`SubmissionPublisher` clase actúa como un simple editor, que acepta elementos para enviar a los suscriptores utilizando un `submit(T item)` método. Cuando un elemento se envía al `submit` método, se envía de forma asíncrona a los suscriptores, como en el siguiente ejemplo:

```

público de clase PrintSubscriber implementa
1
2
3     suscripción de suscripción privada ;
4
5     @Anular
6     public void onSubscribe ( Suscripción de
7     esta . suscripción = suscripción ;
8     suscripción . solicitud ( 1 );
9 }
10
11     @Anular
12     pública vacío onNext ( Entero elemento )
13     Sistema . a cabo . println ( "Artículo
14     suscripción . solicitud ( 1 );
15 }

```

```

16
17     @Anular
18     pública vacío onError ( Throwable de err
19     Sistema . a cabo . println ( "Error ocu
20     }
21
22     @Anular
23     public void onComplete () {
24         Sistema . a cabo . println ( "PrintSubs
25     }
26 }
27
28     clase pública SubmissionPublisherExample {
29
30     public static void main ( String ... arg
31
32     SubmissionPublisher < Integer > publis
33     editor . suscribirse ( nuevo PrintSubs
34
35     Sistema . a cabo . println ( "Envío de
36
37     para ( int i = 0 ; i < 10 ; i ++ )
38     editor . enviar ( i );
39     }
40
41     Hilo . dormir ( 1000 );
42     editor . close ();
43 }
44 }

```

Al ejecutar este ejemplo, se obtiene el siguiente resultado:

```

1 Envío de artículos ...
2 Artículo recibido: 0
3 Artículo recibido: 1
4 Artículo recibido: 2
5 Artículo recibido: 3

```

```
6  Artículo recibido: 4
7  Artículo recibido: 5
8  Artículo recibido: 6
9  Artículo recibido: 7
10 Artículo recibido: 8
11 Artículo recibido: 9
12 PrintSubscriber está completo
```

Dentro de nuestro suscriptor, capturamos el `Subscription` objeto que se pasó al `onSubscribe` método, lo que nos permite interactuar con el `Subscription` en un momento posterior. Una vez que almacenamos el `Subscription` objeto, inmediatamente informamos `Subscription` que nuestro suscriptor está listo para aceptar un artículo (llamando `subscription.request(1)`). Hacemos lo mismo dentro del `onNext` método después de imprimir el artículo recibido. Esto equivale a informar al editor que estamos listos para aceptar otro artículo tan pronto como hayamos terminado de procesar un artículo.

En nuestro método principal, simplemente instanciamos un `SubmissionPublisher` y nuestro `PrintSubscriber` y suscribimos el último al primero. Una vez que se establece la suscripción, sometemos los valores `0` a través `9` de la editorial, que a su vez empuja de forma asíncrona los valores para el abonado. El suscriptor maneja cada elemento imprimiendo su valor a la salida estándar e informa a la suscripción que está listo para aceptar otro valor. Luego pausamos el hilo principal durante 1 segundo para permitir que se completen las presentaciones asíncronas. Este es un paso muy importante ya que el `submit` método *asincrónicamente* empuja los artículos enviados a sus suscriptores. Por lo tanto, debemos proporcionar un período de tiempo razonable para que se complete la acción asíncrona. Por último, cerramos el editor, que a su vez notifica a nuestro suscriptor que la suscripción se ha completado.

También podemos presentar un procesador y encadenar el editor y el suscriptor original con este procesador. En el siguiente ejemplo, creamos un procesador que incrementa los valores recibidos en 10

procesador que incrementa los valores recibidos en 10 y empuja los valores incrementados hacia su suscriptor:

```

1  public class PlusTenProcessor extends Subscriber<Integer> {
2
3      Subscription subscription;
4
5      @Override
6      public void onSubscribe (Subscription subscription) {
7          this.subscription = subscription;
8          subscription.request ( 1 );
9      }
10
11     @Override
12     public void onNext ( Integer elemento ) {
13         enviar ( elemento + 10 );
14         subscription.request ( 1 );
15     }
16
17     @Override
18     public void onError ( Throwable error ) {
19         error.printStackTrace ();
20         closeExceptionalmente ( error );
21     }
22
23     @Override
24     public void onComplete () {
25         Sistema.aCabo.println ( "PlusTenProcessor terminado" );
26         close ();
27     }
28 }
29
30 class SubmissionPublisherExample {
31
32     public static void main ( String ... args ) {
33
34         SubmissionPublisher<Integer> publisher = new SubmissionPublisher<> ();
35         Procesador PlusTenProcessor = new PlusTenProcessor ( publisher );

```

```

Suscriptor PrintSubscriber = nuevo
36
    editor . suscribirse ( procesador );
37
    procesador . suscribirse ( suscriptor )
38
39
    Sistema . a cabo . println ( "Envío de
40
41
    para ( int i = 0 ; i < 10 ; i ++ )
42
    editor . enviar ( i );
43
    }
44
45
    Hilo . dormir ( 1000 );
46
    editor . close ();
47
    }
48
    }
49 }
```

Al ejecutar este ejemplo, se obtiene el siguiente resultado:

```

1 Envío de artículos ...
2 Artículo recibido: 10
3 Artículo recibido: 11
4 Artículo recibido: 12
5 Artículo recibido: 13
6 Artículo recibido: 14
7 Artículo recibido: 15
8 Artículo recibido: 16
9 Artículo recibido: 17
10 Artículo recibido: 18
11 Artículo recibido: 19
12 PlusTenProcessor completado
13 PrintSubscriber está completo
```

Como esperábamos, cada uno de los valores enviados se incrementa en 10 y los eventos recibidos por el procesador (como recibir un error o completarlo) se envían al suscriptor, lo que da como resultado un mensaje completo impreso tanto para el PlusTenProcessor como para el PrintSubscriber .

Conclusión

En la era del procesamiento de datos casi en tiempo real, las corrientes reactivas son un estándar *de facto* . La ubicuidad de este estilo de programación ha llevado a numerosas implementaciones de este estándar, cada una con su propio conjunto duplicado de interfaces. En un esfuerzo por recopilar estas interfaces comunes en un estándar universal para Java, JDK 9 ahora incluye interfaces de flujos reactivos, junto con una poderosa implementación del editor, por defecto. Como hemos visto en este artículo, aunque estas interfaces son modestas en apariencia, proporcionan un método rico para manejar flujos de datos de transmisión de una manera estándar e intercambiable.

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

Temas: JAVA 9, JAVA, FLUJOS REACTIVOS, MENSAJERÍA ASINCRÓNICA, TUTORIAL

Las opiniones expresadas por los contribuidores de DZone son suyas.

Recursos para socios de Java

Microsistemas reactivos: la evolución de los microservicios a escala

Lightbend



jQuery UI y entrada de dirección de autocompletar

Melissa Data



Comandos avanzados de Linux [Cheat Sheet]

Programa de desarrollo de Red Hat



Seguridad de la aplicación de una sola página (SPA) con Spring Boot y OAuth

Okta

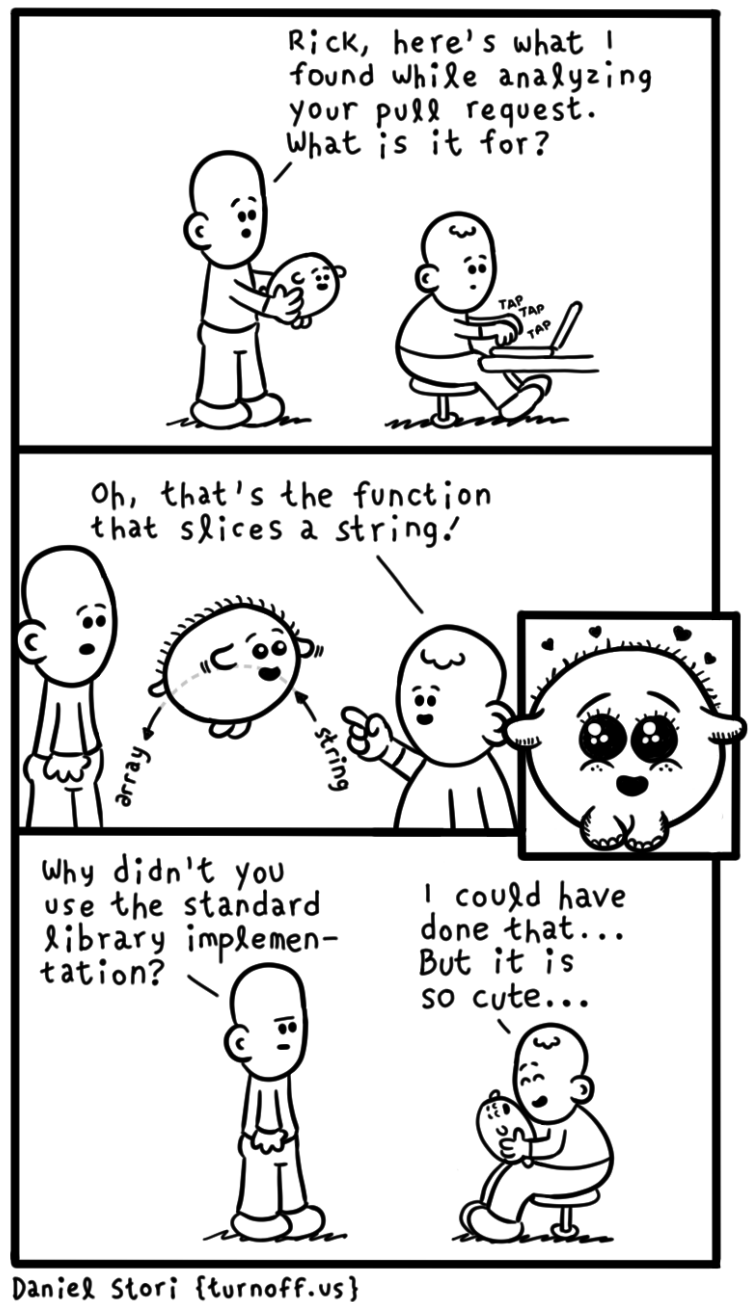


Mi códião inútil

adorable [Comic]

por Daniel Stori  MVB  · 11 y 18 de febrero · Zona de Java


Construir vs Comprar una solución de calidad de datos: ¿Cuál es el mejor para usted? Obtenga información sobre un enfoque híbrido. ¡Descargue el libro blanco ahora!



Build vs Buy a Data Quality Solution: Which is Best for You? Maintaining high quality data is essential for

For you: Maintaining high quality data is essential for operational efficiency, meaningful analytics and good long-term customer relationships. But, when dealing with multiple sources of data, data quality becomes complex, so you need to know when you should build a custom data quality tools effort over canned solutions. Download our whitepaper for more insights into a hybrid approach.

Topics: COMIC, JAVA, FUNCTION, ARRAY, STRINGS

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