RESTO de código cero con json-server

por Mitch Dresdner · 12 y 18 de abril · Zona de integración · Tutorial

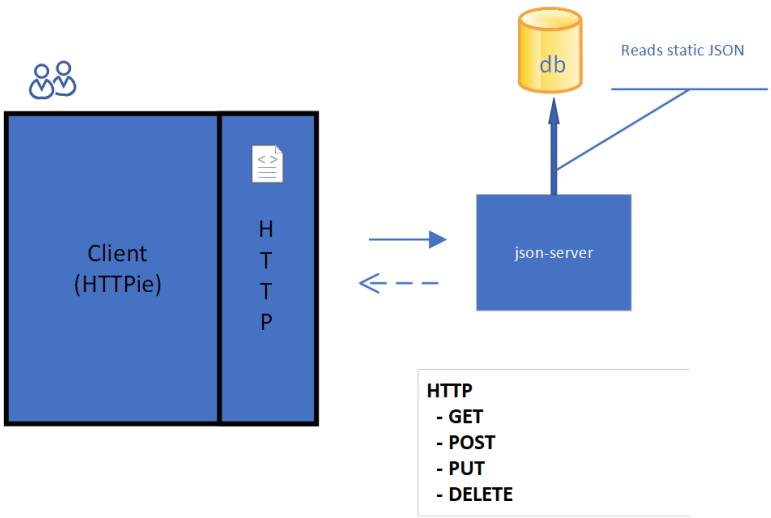
Resumen

Una y otra vez nos encontramos en la necesidad de construir un servidor JSON para compartir esquemas con nuestros clientes mientras nuestro desarrollo está en progreso o para apoyar nuestras propias pruebas de extremo a extremo.

El json-server es una aplicación de JavaScript. Como desarrollador de Java y MuleSoft, no me desconecto trabajando con otras pilas de aplicaciones cuando se adapta bien. Ser capaz de soportar un servidor JSON REST con codificación cero lo convierte en un caso convincente. Si bien puede agregar middleware JavaScript personalizado para mejorar su funcionalidad, para la mayoría de los desarrolladores, las características estándar deberían ser suficientes.

Cuando necesita una solución API JSON rápida y fácil de usar, no muchas soluciones son tan rápidas de

Otro caso de uso para json-server es cuando estás desarrollando una nueva aplicación que depende de las API que tienen un límite de velocidad. Un ejemplo de esto es la API Open Weather Map . Con el nivel gratuito, está limitado a un número fijo de llamadas en una ventana de tiempo móvil. Puede capturar estos tipos de resultados JSON, agregarlos a la base de datos json-server y reproducirlos haciendo solicitudes ilimitadas y sin límites.



Interacción cliente / servidor con el servidor json

Instalación

El json-server es una aplicación de JavaScript, que espero no asuste a muchos desarrolladores de Java o Mule, sin duda, ¡no asustará a ningún políglota! La instalación realmente es bastante simple.

global de json-server. La instalación agregará json-server a su ruta y le permitirá ejecutarlo desde una ventana de shell.

Nota: Si esta es la primera vez que instala npm, es posible que necesite abrir un nuevo shell para agregar la nueva ruta

Instalación del servidor JSON

```
npm install -g json-server

# verificar que la instalación fue exitosa
json-server -v
```

Configurando json-server

Deberá decidir dónde conservará la base de datos del esquema JSON, que conserva el esquema que se devolverá para las solicitudes del cliente.

Base de datos db.json

Con json-server instalado, cree una carpeta donde planee guardar cualquier muestra de datos y propiedades del proyecto.

Depende de usted dónde lo quiera ubicar, me gusta mantenerlo en mi Google Drive para poder reutilizarlo en diferentes máquinas; guardarlo en Git

Usando su editor favorito, ingrese los datos JSON de ejemplo a continuación en el archivo db.json, en la ubicación especificada arriba.

```
{
  "vinos" : [
    { "id" : 1 , "producto" : "SOMMELIER SELECT
    "desc" : "Viña vieja Cabernet Sauvignon"
    { "id" : 2 , "producto" : "MASTER VINTNER"
    "desc" : "Pinot Noir captura aromas delic
    { "id" : 3 , "producto" : "RESERVACIÓN DEL
    "desc" : "Merlot con sabores complejos de
    { "id" : 4 , "producto" : "SANGIOVESE ITALI
    "desc" : "La uva Sangiovese es famosa por
  ],
  "comentarios" : [
    { "id" : 1 , "cuerpo" : "como las pieles de
    { "id" : 1 , "cuerpo" : "las instrucciones
    { "id" : 3 , "cuerpo" : "recibí 3 paquetes
  ],
```

consulte este artículo de los verbos de DZone HTTP .

Desde la carpeta json-server, ejecutaremos un comando rápido para imprimir la ayuda de la línea de comandos. Ejecute el siguiente comando:

```
1  json-server -h
```

Obtener la ayuda de la línea de comandos json-server

Como puede ver, hay muchas opciones para cambiar o anular los comportamientos predeterminados.

```
1  {
2    "puerto" : 9000
3  }
```

archivo de configuración json-server: json-server.json

Con los preliminares fuera del camino, comencemos json-server y prepárese para enviar algunas solicitudes de línea de comando.

```
1  json-server --watch json \ db.json
```

Iniciando el json-server

Debajo de la parte ASCII, debería ver lo siguiente:

1. El archivo de base de datos, ***json \ db.json*** , se cargó correctamente.

2. URI para recursos JSON que se cargaron.

3. El URI para el sitio web interno predeterminado (puede cambiar esto).

Ejemplos de HTTPie

Para instalar HTTPie para los ejemplos con los que trabajaremos, puede descargarlo usando este enlace: Descarga de HTTPie .

Siéntase libre de usar Postman o curl desde un shell de terminal Git bash en Windows si lo prefiere. Debería poder adaptar los ejemplos de HTTPie en consecuencia.

HTTPie es una herramienta de línea de comando tipo curl que se puede usar en Unix y Windows. Me gusta más que curl porque viene cargado con mucha azúcar sintáctica.

```
1  http localhost: 3000 / wines / 1 <1>
2  o
   http http: // localhost: 3000 / wines / 1 <2>
3
```

puerto que asignó, debe aparecer en la salida.

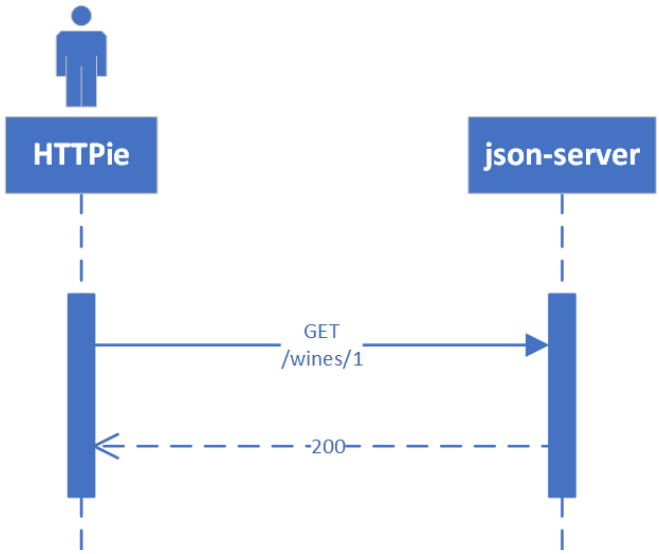
```
1   http: // localhost: 3000
```

Usa el navegador para acceder a json-server

En ***Recursos*** , debe observar que el *vinicultor* ha sido mal escrito como *vintnor* . Puede corregir el error utilizando su editor favorito para modificar la línea en * db.json * y guardar el archivo. Actualizando el enlace, debe notar que el cambio ya ha sido recogido por json-server.

Proporcionar la `--watch` opción le dijo al servidor json que se ejecute en modo de desarrollo, mirando y recargando los cambios.

Hacer solicitudes GET



OBTENER	http localhost: 3000 / wines / 1	Vino con ID = 1
OBTENER	http localhost: 3000 / wines? price_gte = 100	Vinos con precio> = 100
OBTENER	http localhost: 3000 / wines? id_ne = 2	filtro id = 2
OBTENER	http localhost: 3000 / wines? _embed = comentarios	incorpora todos los comentarios
OBTENER	http localhost: 3000 / wines / 1? _embed = comentarios	incrustar comentarios para ID = 1

Para ver más ejemplos, consulte el sitio web json-server

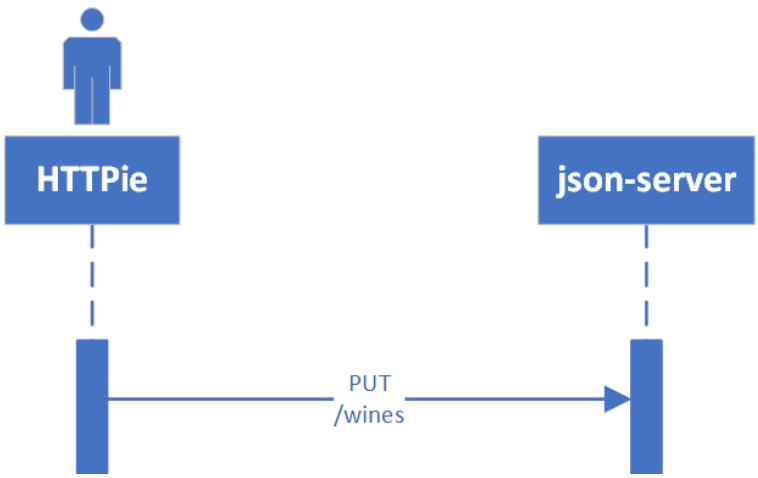
Hacer una solicitud POST

Con POST, agregaremos un nuevo registro a la base de datos.

Solicitud	URI	Resultado
ENVIAR	http POST localhost: 3000 / wines ... (ver arriba)	Nueva entrada de vino con id = 5
OBTENER	http localhost: 3000 / vinos	Todas las entradas de vino
OBTENER	http localhost: 3000 / wines? desc_like = grape	Todos los vinos con *uva* en desc

Hacer una solicitud PUT

En nuestro ejemplo de PUT, haremos un cambio en el ***producto*** para el registro que acabamos de agregar con POST.

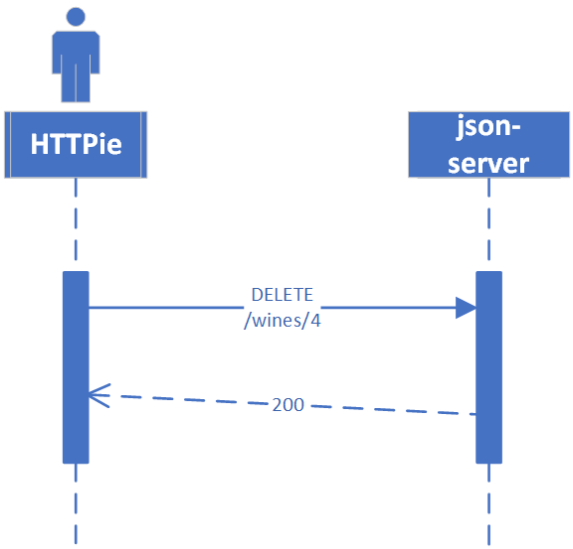


	arriba)	con id = 5
OBTENER	http localhost: 3000 / vinos	Todas las entradas de vino

NOTA: si no ingresa todos los campos, PUT reemplazará solo lo que proporcione.

Finalmente, una solicitud DELETE

Para concluir nuestras operaciones CRUD de ejemplo, eliminaremos el registro con ID = 5




```
1  http DELETE localhost: 3000 / wines / 5
```

Use HTTPie, curl o cartero

Borrar petición

Solicitud	URI	Resultado

sfrutado escribiéndolo, ¡estoy
ntarios!

La guía de Java: desarrollo y evolución

Los mejores patrones de diseño GoF para microservicios: estrategia, fachada y más
Beneficios de usar el futuro API en Kotlin
Archivos JAR de liberación múltiple, capacidades de Jigsaw y la herramienta Jlink en Java 9

Descargar My Free PDF

Temas: JSON API, NPM, MULA, JAVASCRIPT, INTEGRACIÓN

Recursos para socios de **integración**

Una guía completa para la nube de integración empresarial
SnapLogic

Cómo la transmisión de eventos puede beneficiar el diseño de API.
Capital uno

Libro de arquitectura de O'Reilly Microservice: Alineación de principios, prácticas y cultura
CA Technologies

Converting your Swagger 2.0 API Definition to OpenAPI 3.0
Runscope

Note: This article dives into the details of Java 8 Streams and its "under the hood" implementation in Java. This is a continuation to JλVλ 8: A Comprehensive Look article, where Java 8 Streams are discussed. So its recommended to first go through the discussion on Java 8 Streams here.

You can also find Part 1.1, which discusses Lambdas, here.

Filter-Map-Reduce Through Streams

Filter-map-reduce is a pattern and facility available in most programming languages and it allows developers to **perform bulk data operations on collections of data**.

- **Filter** simply means to filter out some data that should not be processed through the pipeline.

- **Map** simply means to map the input to some other output. There is another intermediate operation, **flatMap()**, that allows you to flatten stream elements if they are some sort of collections themselves that are required for individual processing, e.g: { {1,2}, {3,4}, {5,6} } -> flatMap -> {1,2,3,4,5,6}. `FlatMap` returns a stream of elements flattened from an original

Whereas **Reduce is a fold operation**, i.e. it folds the elements to reduce them to a single result. It applies a binary operator to each element of the stream where the first argument is the return value of the execution from the previous iteration and the second is the current stream element.

- **Collect is a mutable reduction operation**, i.e., the reduced value is a mutable container in which the results are added by updating the state of this container (reduce value) whereas, in **reduce, the result is replaced** as a consequence of the fold operation.

- **The Collect operation always returns a mutated state** of the initially supplied value for each iteration, whereas the **Reduce operation always returns a new value.**

Reduce Method:

Optional<T> reduce(BinaryOperator<T> accumulator): No identity/initial value is supplied to it, an accumulator is executed on the stream elements. **Accumulator** accepts two arguments of stream element type and returns an element (reduced value) of the stream element type. The first argument is the result of the accumulator's execution from the previous iteration and the second

In accumulator: 6, 4
In accumulator: 10, 5
15

T reduce(T identity, BinaryOperator<T> accumulator): An **identity**/initial value of stream element type is supplied to it and **accumulator** is applied in the same way as described above. The reduce operation **returns the reduced value**.

For example:

```
System.out.println(
    Stream.of(1, 2, 3, 4, 5).reduce(0, (a, b) -
      System.out.println("In accumulator: " + a
      return a + b;
}));
```

Output:

In accumulator: 0, 1
In accumulator: 1, 2
In accumulator: 3, 3
In accumulator: 6, 4
In accumulator: 10, 5
15

U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner): An **identity**/initial value of any type can be supplied to it, an **accumulator** is executed on

```
System.out.println(
    Stream.of(1, 2, 3, 4, 5).reduce(0, (a, b) -
      System.out.println("In accumulator: " + a
      return a + b;
}, (a, b) -> {
  System.out.println("In combiner: " + a + ", "
  return a + b;
}));
```

Output:

In accumulator: 0, 1
In accumulator: 1, 2
In accumulator: 3, 3
In accumulator: 6, 4
In accumulator: 10, 5
15

```
System.out.println(
    Stream.of(1, 2, 3, 4, 5).parallel().reduce(
      System.out.println("In accumulator: " + a
      return a + b;
}, (a, b) -> {
  System.out.println("In combiner: " + a + ", "
  return a + b;
}));
```

Output:

BiConsumer<R,R> combiner). A **supplier** has to be defined which serves as the container that will be used to collect the values. An **accumulator** has to be specified which accumulates the results into the container specified in the supplier, it accepts two arguments - the first one is the supplied container and the other one is the current stream element. A **combiner** also has to be specified which is executed when a stream is executed in parallel, it combines the collected values from parallel executions. The collect operation **returns the container with all the values collected/accumulated in it**.

For example:

```
List<String> x1 = Stream.of("R", "O", "B", "I",
.collect(() -> new ArrayList<>(), (a, e) -> {
  System.out.println("In accumulator: " + a + "
  a.add(e.toLowerCase());
}, (a, e) -> {
  System.out.println("In combiner: " + a + ", "
  a.addAll(e);
});
```

Output:

In accumulator: [], R
In accumulator: [r], O
In accumulator: [r, o], B
In accumulator: [r, o, b], I

In accumulator: [], B
In accumulator: [], O
In accumulator: [], R
In accumulator: [], N
In accumulator: [], I
In combiner: [r], [o]
In combiner: [i], [n]
In combiner: [b], [i, n]
In combiner: [r, o], [b, i, n]

R collect(Collector<? super T,A,R> collector):
A **collector** has to be specified which serves the same purpose as the above overload. There are different pre-built collectors already present in the JDK library which can be used as is or we can also create our own collector by implementing one or using **static factory methods (of(...)) from the Collector class** supplying it with supplier, accumulator, combiner, and stream characteristics.
The factory methods in Collector class to create collectors are:

```
1 Collector.of(Supplier supplier, BiConsumer accu
2 Collector.of(Supplier supplier, BiConsumer accu
```

You can have a look at the **different collectors available** in the Oracle docs.

Streams Execution Order

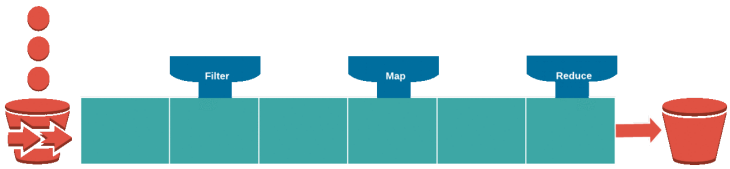
Streams are evaluated lazily and the stream is

RROOBB (because the source was traversed vertically)
And not ROBROB (if the source was traversed horizontally)

```
Stream.iterate(0, i -> i + 1) // infinite strea
.peek(System.out::println)
.findFirst();
```

In spite of being an infinite stream, the above operation completes almost immediately because of the lazy characteristic of Stream and its vertical traversal.

Let's see this vertical traversal of streams works, through a visualization:



Keeping the vertical traversal aspect of streams in view can help to **optimize the performance of stream execution by ordering the operations sensibly** in a stream pipeline. Let's see through an example how the number of operations that are executed can decrease/increase based on the ordering of operations in the pipeline:

```
Stream.of("R", "O", "B")
.map(e -> {
```

In map: O -> In filter: o ->
In map: B -> In filter: b ->

In this example, **7 operations** in total in the pipeline are executed

```
Stream.of("R", "O", "B")
.filter(e-> {
  System.out.println("");
  System.out.print("In filter: " + e + " -> ");
  return e.equalsIgnoreCase("R");
 })
.map(e -> {
  System.out.print("In map: " + e +" -> ");
  return e.toLowerCase();
})
.forEach(e -> {
  System.out.print("In forEach: " + e);
});
```

Output:

In filter: R -> In map: R -> In forEach: r
In filter: O ->
In filter: B ->

This is the same example as above with the same output, the **difference being that the order of operations is changed**, i.e. filter and map are interchanged. Here, **only 5 operations** are executed in total

result depends on any state that might change during the execution of the stream. Such a function in terms of functional programming would be seen as an **un-pure function since it is depending or mutating some state**.

Java Streams API has **some intermediate operations that are stateful**, they maintain some state internally to achieve their purpose. Those intermediate operations are:

- distinct()
- sorted()
- limit()
- skip()

These operations are stateful in the sense that they keep the state from previously processed elements while processing current elements. **Since they just use this state internally, they are deterministic and safe in nature** and don't have adverse side effects.

When **Java Stream's stateful intermediate operations are used, they process the stream horizontally** instead of vertically, as an example, think about the `sorted()` intermediate operation - if it would process stream elements vertically it would

sorted: RO
sorted: OB
forEach: B
forEach: O
forEach: R

Since `sorted` is a stateful operation, here the stream elements are processed horizontally instead of vertically. Let's compare it with a stateless operation and see how the stream elements are processed vertically instead:

```
Stream.of("R", "O", "B")
.map(a->{
  System.out.println("map: " + a);
  return a;
})
.forEach(e -> System.out.println("forEach: " +
```

Output:

map: R
forEach: R
map: O
forEach: O
map: B
forEach: B

Here, instead of sorted (stateful operation), the `map` (stateless) operation is used, and you can see from the output that stream elements are processed

synchronization (thread-safety) in such cases. Using a **stateful function dissolves all our parallelism advantage** that is inherent with streams. Let's see how this affects the code through an example:

```
for (int i = 0; i < 5; i++) {
  Set<String> stringSet = new HashSet<>();
  System.out.println(Stream.of("R", "B", "I", "
    .parallel()
    .map(
        // stateful function
        e -> {
          if (stringSet.add(e))
            return e.toLowerCase();
          else
            return "";
        })
    .collect(Collectors.joining()));
}
```

Output:

rbioiin
robin
robin
rnobi
rnobi

As you can see, this stream pipeline is **executed 5 times and the results are not consistent** and are non-deterministic due to the usage of a stateful

Output:

rbinobinin
rbinobinin
rbinobinin
rbinobinin
rbinobinin

As you can see, since a stateless function is used in a `map` operation, the output of the stream execution is consistent.

Streams Under the Hood:

Let's have a peek under the hood to understand how Streams API in Java is implemented and how it executes. This will help us understand how the features like lazy evaluation, vertical and horizontal execution, etc., are achieved. **Having some insights into the implementation helps to easily identify bugs and bottlenecks, and helps to optimize performance and construct efficient pipelines** with Streams API.

Streams API in Java **follows a fluent API design** to construct a data processing pipeline. The data source constructors/generators and intermediate operations in the Streams API return a stream type themselves which allows linking more operations to the pipeline in a fluent manner.

As we have already discussed, Streams pipeline is

Spliterator allows you to access the elements in a more direct way. **Primary methods in a Spliterator are:**

- **boolean tryAdvance(Consumer<? super T> action):** If any element is remaining, it executes the specified action on it and returns true, otherwise, it returns false

- **void forEachRemaining(Consumer<? super T> action):** Executes the specified action on all remaining elements sequentially.

- **Spliterator<T> trySplit():** Splits the elements and represents the split elements in a new Spliterator and returns that Spliterator. The elements represented by the returned Spliterator are then removed and not represented by the original Spliterator. This is used for partitioning the elements for parallel stream execution.

- **int characteristics():** Returns a bitmap that represents the characteristics (ORDERED, DISTINCT, SORTED, SIZED, etc.) of the spliterator and it's elements.

An important part to understand in the Streams API is the **usage of characteristic bitmap flags** that

the Collection interface but it is overridden by sub-classes that return a more specialized and efficient Spliterator). If we have a **look at the java.util.ArrayList** class, we can see that it contains a static inner class **ArrayListSpliterator<E> implements Spliterator<E>**, which overrides the `characteristics()` method as below:

```
1  public int characteristics() {
     return Spliterator.ORDERED | Spliterator.SIZE
2
3  }
```

It indicates the characteristics of this collection:

- Spliterator.ORDERED - the elements of the array list are ordered.
- Spliterator.SIZED - the array list has a definite size.
- Spliterator.SUBSIZED - all the Spliterators returned after `trySplit` representing a section of this array list will also be sized.

Let's break down the example above to see characteristics at each step:

- **list.stream()** - Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED

SUBSIZED flags. And then a map operation is applied which will preesrve the ordered flag but clear the SORTED flag.

During the execution of the stream, the information gathered from the characteristics at each step **could be utilized to optimize** the stream execution, e.g., **some operations in the stream pipeline could even be skipped.**

For example:

```
1 HashSet<String> stringSet = new HashSet<>();
2 stringSet.stream().distinct().forEach(System.ou
```

Since `HashSet` 's Spliterator already defines the DISTINCT flag as it's characteristic, the `distinct()` operation would be skipped during stream evaluation.

Now, let's continue to understanding the streams execution process. For easy comprehension, you can view a stream setup in your code **as a linked list** (with a **HEAD node** representing the data source, linked to other **intermediate nodes** which represent the operations through which each element of the data source is passed, and then at the end of this linked list is the **terminal node** which actually computes the final result for you). Let's see how streams execute at runtime through an example:

the `Collection.stream()` default method and returns an object of Stream type

```
default Stream<E> stream() {
  return StreamSupport.stream(spliterator(), fa
}
```

The `spliterator()` method is overridden in the `ArrayList` class, which returns an instance of **ArrayListSpliterator**<E> and implements the Spliterator<E> class.
The `StreamSupport.stream(...)` method returns an instance of the `ReferencePipeline.Head` class which is an implementation of Stream type. This represents the head node of the stream's pipeline. This **head node stores the information about the source, spliterator,** which is provided to it.

```
public static <T> Stream<T> stream(Spliterator<
  Objects.requireNonNull(spliterator);
  return new ReferencePipeline.Head<>(spliterat
}
```

2. Next, the `filter()` **method** is executed on the head node returned in Step#1. This calls the `ReferencePipeline.filter(...)` method which returns an instance of the `ReferencePipe.StatelessOp` class (in fact, an instance of an anonymous class which extends `ReferencePipe.StatelessOp`) which is an

```
3
4       @Override
        Sink<P_OUT> opWrapSink(int flags, Sink<
5
            return new Sink.ChainedReference<P_
6
                @Override
7               public void begin(long size) {
8
                    downstream.begin(-1);
9               }
10              @Override
11              public void accept(P_OUT u) {
12
13                  if (predicate.test(u))
                        downstream.accept(u);
14
15              }
16          };
17      }
18  };
19  }
```

3. Next, the **`sorted()` method** is executed on the intermediate node returned in Step#2. This calls the `ReferencePipeline.sorted(...)` method which returns an instance of the `ReferencePipe.StatefulOp` class which is an implementation of the Stream type. This represents an **intermediate node which is a stateful** stream operation in the stream's pipeline. This intermediate node sets the characteristics at this step based on the stream operation, sets the source node (which is the

node, and establishes links. It is built **similarly to the filter method in Step#2**.

```
public final <R> Stream<R> map(Function<? super
    Objects.requireNonNull(mapper);
    return new StatelessOp<P_OUT, R>(this, Stre
                                 StreamOpFlag.N
        @Override
        Sink<P_OUT> opWrapSink(int flags, Sink<
            return new Sink.ChainedReference<P_
                @Override
                public void accept(P_OUT u) {
                    downstream.accept(mapper.ap
                }
            };
        }
    };
}
```

5. Next, the `forEach()` **method** is executed on the intermediate node returned in Step#4. This calls the `ReferencePipeline.forEach(...)` method which **does not return a Stream object** because forEach is a terminal operation. Most of the methods representing terminal operations, in turn, **create an object of TerminalOp type and execute the**

```
    assert getOutputShape() == terminalOp.input
    if (linkedOrConsumed)
        throw new IllegalStateException(MSG_STR
    linkedOrConsumed = true;

    return isParallel()
           ? terminalOp.evaluateParallel(this,
           : terminalOp.evaluateSequential(this
}
```

The `evaluateSequential` **method is responsible for creating a wrapped sink and passing stream elements through it**. In the current context, you can think of sink as a kitchen sink where each of the stream elements is dropped one by one. The sink here is, in fact, a **Sink type which extends the Consumer type,** i.e., it accepts an element of the stream and performs an operation on it. A **wrapped sink is one which wraps another sink** into it. So what really happens is that each node in the stream creates a sink which wraps a sink which is next to it in the stream pipeline, so this **creates a top-level sink for the first intermediate node in the pipeline which wraps the next sink and this process goes on until it reaches the terminal node**. The following method creates a wrapped sink. This method is passed with the terminal node which is of type Sink, which is then

is also similar to this, and you can have a look at the code for `filter()` above:

```
public final <R> Stream<R> map(Function<? super
    Objects.requireNonNull(mapper);
    return new StatelessOp<P_OUT, R>(this, Stre
                                 StreamOpFlag.N
        @Override
        Sink<P_OUT> opWrapSink(int flags, Sink<
            return new Sink.ChainedReference<P_
                @Override
                public void accept(P_OUT u) {
                    downstream.accept(mapper.ap
                }
            };
        }
    };
}
```

Have a look at the `opWrapSink` method above. It creates a new Sink which wraps the sink passed in arguments. Also, have a look at the **accept method, this is where the actual operation logic is written** which gets executed for an element of the stream. In the accept method, once the operation's own logic has been executed (mapper.apply(u)), **it calls the accept method of the sink it has**

```
            spliterator.forEachRemaining(wrappedSink
6
7           wrappedSink.end();
8       }
9       else {
            copyIntoWithCancel(wrappedSink, spliter
10
11      }
12  }
```

Here, the **spliterator.forEachRemaining(Consumer<? super T> action)** method, calls the `accept` method on the top level sink for each element sequentially and sink(s) in turn may call the next sink in the pipeline.

Note: `tryAdvance` in a loop is called in case of **short-circuiting terminal operations** (the operations that can break out in between without processing all remaining elements, like `findFirst`, `anyMatch`, etc.).

Let's get back to our original example and see how the pipeline is executed. The top-level sink would be the one that is returned when the `opWrapSink` method was executed on the Stream object returned for the filter operation. So when `spliterator.forEachRemaining` is executed, it will call the accept method of this sink for each element:

```
1  public void accept(P_OUT u) {
2    if (predicate.test(u))
3      downstream.accept(u);
```

```
public void begin(long size) {
    if (size >= Nodes.MAX_ARRAY_SIZE)
        throw new IllegalArgumentException(Node
    array = (T[]) new Object[(int) size];
}

@Override
public void end() {
    Arrays.sort(array, 0, offset, comparator);
    downstream.begin(offset);
    if (!cancellationWasRequested) {
        for (int i = 0; i < offset; i++)
            downstream.accept(array[i]);
    }
    else {
        for (int i = 0; i < offset && !downstre
            downstream.accept(array[i]);
    }
    downstream.end();
    array = null;
}

@Override
public void accept(T t) {
    array[offset++] = t;
}
```

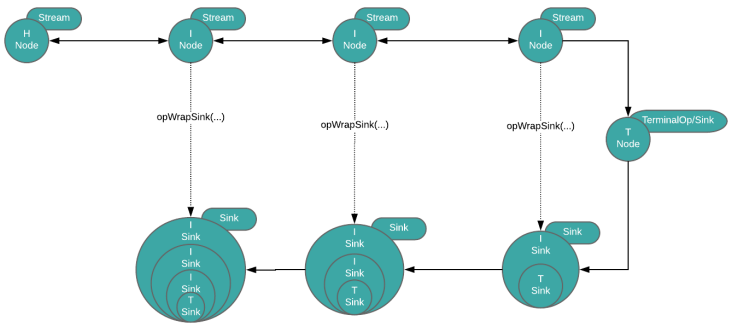
After the **sorting process is complete, the next downstream sink's `accept` method** is executed.

```
2            consumer.accept(t);
3        }
```

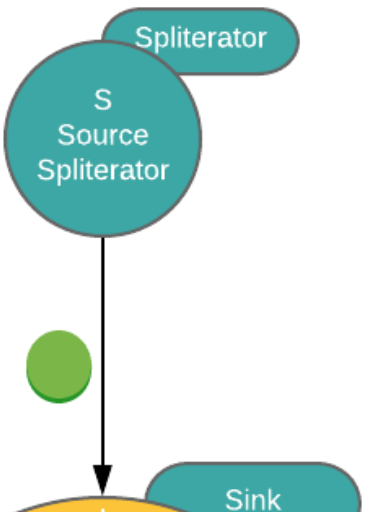
Which, finally, in our case, just prints the elements.

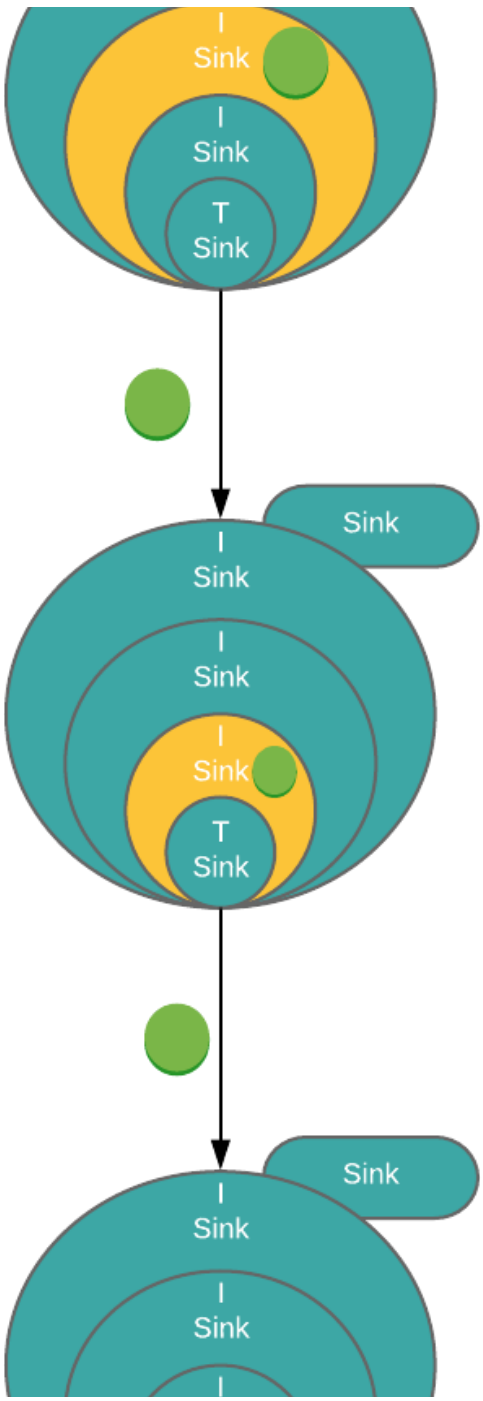
Let's see through a diagram the stream creation and evaluation/execution process:

Stream Creation:



Stream Evaluation:




Sink
Sink
T
Sink
Sink
Sink
Sink
Sink
T
Sink
Sink
Sink
Sink