

[Home](#) / [Tutorial](#) / [Project Reactor](#)

Project Reactor - CacheMono & CacheFlux with Caffeine Examples

POSTED ON 📅 24 JUL 2021 BY 👤 IVAN ANDRIANTO

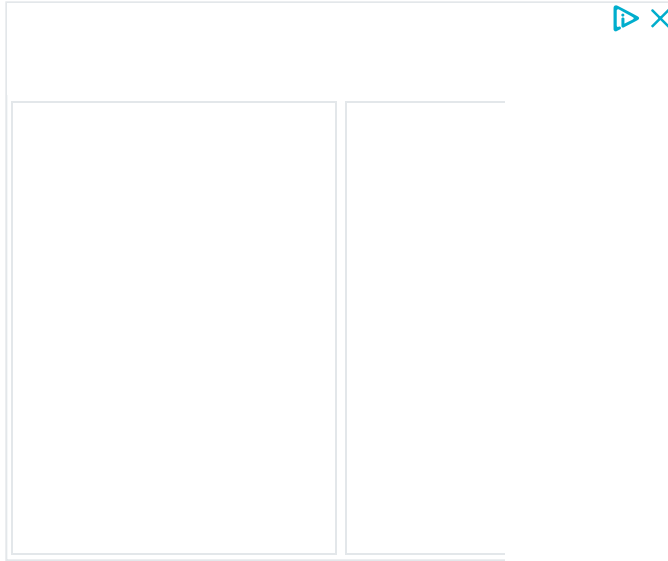


This tutorial shows you how to use **CacheMono** and **CacheFlux** in Project Reactor, including how to use them with Caffeine.

If you're using Project Reactor, sometimes you may want to cache a Mono or a Flux. Fortunately, it already provides an opinionated caching helper called **CacheMono** and **CacheFlux**. They define a cache abstraction for storing and restoring a **Mono** or a **Flux**. This tutorial explains how to use those two and also how to integrate them with **Caffeine**, a popular caching library.

Using **CacheMono**

Using **CacheMono** basically consists of three steps. The first one is looking up the value from the source based on the given key. The second step is handling cache missing, which will be done if the first step results in a cache miss. The last step is writing the value to the source in case of cache miss.



There are different ways to use **CacheMono**. One of which requires you to handle all the steps above (lookup value, handle cache misses, write value to the cache) manually. Alternatively, you can also provide a **Map** and let Project Reactor handle the lookup and write value to the cache – you only need to handle cache misses.

Manually Handle Lookup and Write

Let's start with the first way in which you need to handle lookup and write data to the cache manually. First, you need to handle how to retrieve the value from the cache using the below **lookup** method.

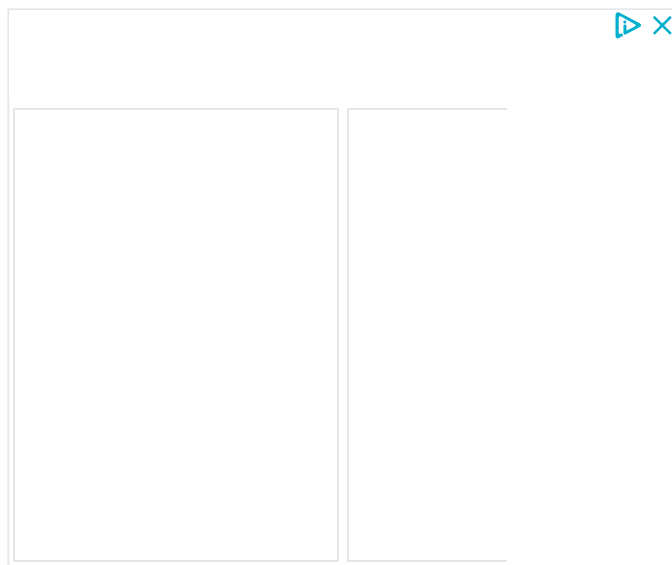
```
public static <KEY, VALUE>  
MonoCacheBuilderCacheMiss<KEY, VALUE>  
lookup(Function<KEY, Mono<Signal<? extends VALUE>>>  
reader, KEY key);
```



It requires you to pass a **Function** as the first argument. The passed **Function** needs to accept a key as the parameter and returns a **Mono**. So, you need to pass a function which is responsible for retrieving a value based on the given key. You can use any source to store the value of each key. For example, you can use Reactor's **Context**, **Map**, **Tree**, or any data structure. For the second argument, you need to pass the key of the value to be retrieved.

Using **CacheMono** also requires you to handle cases when the given key cannot be found in the cache, usually known as cache miss. The return type of the first lookup method is **MonoCacheBuilderCacheMiss**. You can use one of its **onCacheMissResume** methods to handle cache misses.

```
1 MonoCacheBuilderCacheWriter<KEY, VALUE>  
  onCacheMissResume(Supplier<Mono<VALUE>>  
    otherSupplier);  
2 MonoCacheBuilderCacheWriter<KEY, VALUE>  
  onCacheMissResume(Mono<VALUE> other);
```

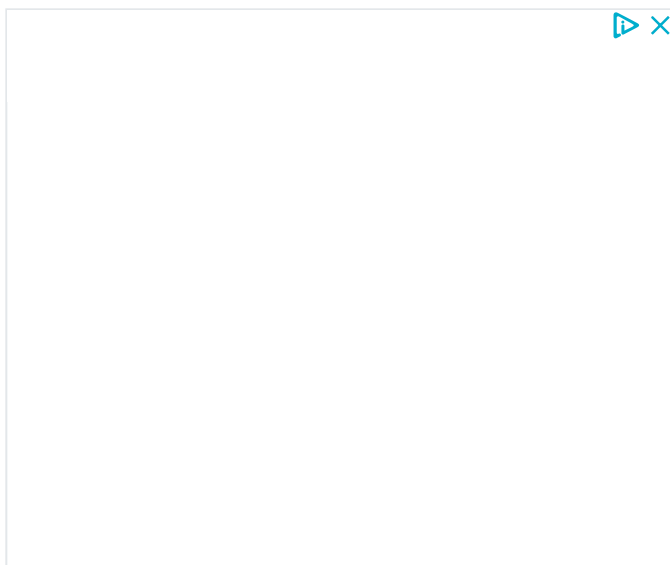


There are two `onCacheMissResume` variants. The first one requires you to pass a `Supplier` that returns a `Mono`. It can be used if you don't need to pass any argument for generating the value. However, if you need to pass an argument (usually the key) for generating the value, you should use the variant that accepts a `Mono` as the argument.

Another thing you need to handle is writing the data to the cache. The `MonoCacheBuilderCacheWriter` class has a method called `andWriteWith`. You need to call it and pass a `BiFunction` that writes the value returned by `onCacheMissResume` to the cache. The `andWriteWith` method is only called on cache misses.

```
Mono<VALUE> andWriteWith(BiFunction<KEY, Signal<?
extends VALUE>, Mono<Void>> writer)
```

With the above explanation, below is an example of how to use `CacheMono` using the above `lookup` method. In the example below, we are going to store the values in a `Map<String, String>`.



```
final Map<String, String> mapStringCache = new  
HashMap<>();
```

Below are the methods that can be passed as `onCacheMissResume` argument. The first one (with 0 parameter) can be passed as a `Supplier`, while the second one can be passed as a `Mono`.

```
1  private Mono<String> handleCacheMiss() {  
2      System.out.println("Cache miss!");  
3  
4      return Mono.just(ZonedDateTime.now().toString());  
5  }  
6  
7  private Mono<String> handleCacheMiss(String key) {  
8      System.out.println("Cache miss!");  
9  
10     return Mono.just(key + ": " +  
11     Instant.now().toString());  
11 }
```

The code below is a complete chain that uses the `lookup`, `onCacheMissResume`, and `andWriteWith` methods.

```

1  final Mono<String> cachedMono1 = CacheMono
2      .lookup(
3          k ->
4      Mono.justOrEmpty(mapStringCache.get(key)).map(Signal::r
5          key
6      )
7      //      .onCacheMissResume(this::handleCacheMiss) //
8      Uncomment this if you want to pass a Supplier
9      .onCacheMissResume(this.handleCacheMiss(key))
10     .andWriteWith((k, sig) -> Mono.fromRunnable(() ->
11         mapStringCache.put(k,
12     Objects.requireNonNull(sig.get()))
13     ));

```

Provide a Map

Another way to use `CacheMono` is by providing a `Map`. Using this way requires you to use one of the `lookup` methods that accepts a `Map`.

```

    public static <KEY, VALUE>
1 MonoCacheBuilderMapMiss<VALUE> lookup(Map<KEY, ? super
    Signal<? extends VALUE>> cacheMap, KEY key);
    public static <KEY, VALUE>
2 MonoCacheBuilderMapMiss<VALUE> lookup(Map<KEY, ? super
    Signal<? extends VALUE>> cacheMap, KEY key,
    Class<VALUE> valueClass);

```

The first method requires you to pass a `Map` as the first argument and a key as the second argument. The key type of the `Map` must be compatible with the type of the passed key. The `Map`'s value type must be a `Signal<T>`. If you want to use the first method, you have to be able to provide the cache data representation in a `Map<KEY, ? super Signal<? extends VALUE>>`. That means you can store any type of `Signal` including `next`, `complete`, and `error`.

You also need to handle cache misses. The

`MonoCacheBuilderMapMiss` also has `onCacheMissResume` methods, as shown below. One of which accepts a `Supplier` and the other accepts a `Mono`. Below are the methods.

```
1  Mono<VALUE> onCacheMissResume(Supplier<Mono<VALUE>>  
   otherSupplier)  
2  Mono<VALUE> onCacheMissResume(Mono<VALUE> other)
```

If a given key doesn't exist in the `Map`, the `onCacheMissResume` method will be invoked. Unlike the previous `lookup` method (which doesn't use `Map`), the returned value will be stored to the `Map` cache automatically. Therefore, it doesn't provide `andWriteWith` method and you don't need to manually save the value to the cache.

Let's start with the example of using the first lookup method (the one with two parameters). First, you need to provide a `Map` whose value type is `Signal<? extends String>`.

```
final Map<String, Signal<? extends String>>  
mapStringSignalCache = new HashMap<>();
```

Below is the usage example.

```
1  final Mono<String> cachedMono2 = CacheMono  
2      .lookup(mapStringSignalCache, key)  
   //      .onCacheMissResume(this::handleCacheMiss) //  
3  Uncomment this if you want to pass a Supplier  
4      .onCacheMissResume(this.handleCacheMiss(key));
```

The second method (the one with three parameters) is similar to the first one. The difference is it accepts a third argument whose type is `Class<VALUE>` that indicates the generic class of the resulting `Mono`. You can use this method if you want to cast the cached signal value to a given type (must be a subtype of the signal value type).

For the second `lookup` method, we are going to create another `Map` whose type value is `Signal<?>`.

```
final Map<String, Signal<? extends Object>>
mapObjectSignalCache = new HashMap<>();
```

Below is the usage example which passes `String.class` as the third argument of the `lookup` method. As a result the value is casted to `String`. Be careful as it may throw `ClassCastException` if the value cannot be casted to the given class.

```
1 final Mono<String> cachedMono3 = CacheMono
2     .lookup(mapObjectSignalCache, key, String.class)
3 //     .onCacheMissResume(this::handleCacheMiss) //
4 Uncomment this if you want to pass a Supplier
   .onCacheMissResume(this.handleCacheMiss(key));
```

Using CacheFlux

The usage of `CacheFlux` is similar to `CacheMono`. You need to handle how to data lookup from cache, handle cache misses, and write data to the cache. The main difference is you need to work with a list of values or `Signals`. With `CacheFlux`, you can also choose whether to manually handle the lookup and write process or provide a `Map`, depending on the used `lookup` method.

Manually Handle Lookup and Write

The first way is to manually handle lookup and write values to the cache, using the below `lookup` method.

```
1 public static <KEY, VALUE>
2 FluxCacheBuilderCacheMiss<KEY, VALUE> lookup(
3     Function<KEY, Mono<List<Signal<VALUE>>>> reader,
4     KEY key
5 )
```


You need to pass a function that accepts a key as the argument. Inside the passed function, you need to obtain the values from the cache based on the given key and return it as a **Mono** of **List** of **Signal** (**Mono<List<Signal<VALUE>>>**). If the key doesn't exist in the cache, you have to return an empty **Mono**.

Next, you have to handle how to generate values on cache misses. The **FluxCacheBuilderCacheMiss** class has two methods named **onCacheMissResume**. You can choose to pass a **Flux** or a **Supplier**. The former should be used if you need to generate the value based on the given key.

```
FluxCacheBuilderCacheWriter<KEY, VALUE>  
1 onCacheMissResume(Supplier<Flux<VALUE>>  
  otherSupplier);  
2 FluxCacheBuilderCacheWriter<KEY, VALUE>  
  onCacheMissResume(Flux<VALUE> other);
```

Lastly, you need to handle how to store the data generated by **onCacheMissResume** method by using **FluxCacheBuilderCacheMiss**'s **onCacheMissResume** method.

```
Flux<VALUE> andWriteWith(BiFunction<KEY,  
List<Signal<VALUE>>, Mono<Void>> writer);
```

Let's start with the example. This time, we have a **Map** whose key type is **Integer** and value type is **List<Integer>**.

```
final Map<Integer, List<Integer>> mapIntCache = new  
HashMap<>();
```

Below are the methods that can be passed as **onCacheMissResume** argument. The first one (with 0 parameter) can be passed as a **Supplier**, while the other can be passed as a **Mono**.

```
1  private Flux<Integer> handleCacheMiss() {
2      System.out.println("Cache miss!");
3      final List<Integer> values = new ArrayList<>();
4
5      for (int i = 1; i <= 5; i++) {
6          values.add(i);
7      }
8
9      return Flux.fromIterable(values);
10 }
11
12 private Flux<Integer> handleCacheMiss(Integer key)
13 {
14     System.out.println("Cache miss!");
15     final List<Integer> values = new ArrayList<>();
16
17     for (int i = 1; i <= 5; i++) {
18         values.add(i * key);
19     }
20
21     return Flux.fromIterable(values);
22 }
```

Below is a complete chain that uses the `lookup`, `onCacheMissResume`, and `andWriteWith` methods.

```

1  final Flux<Integer> cachedFlux1 = CacheFlux
2      .lookup(
3          k -> {
4              if (mapIntCache.get(k) != null) {
5                  Mono<List<Signal<Integer>>> res =
6                      Flux.fromIterable(mapIntCache.get(k))
7                          .map(Signal::next)
8                          .collectList();
9                  return res;
10             } else {
11                 return Mono.empty();
12             }
13         },
14         key
15     )
16     .onCacheMissResume(this::handleCacheMiss) //
17     // Uncomment this if you want to pass a Supplier
18     // .onCacheMissResume(() -> Flux.defer(() ->
19     //     this.handleCacheMiss(key)))
20     .andWriteWith((k, sig) -> Mono.fromRunnable(() ->
21         mapCache.put(
22             k,
23             sig.stream()
24                 .filter(signal -> signal.getType() ==
25                     SignalType.ON_NEXT)
26                 .map(Signal::get)
27                 .collect(Collectors.toList())
28         )
29     ));

```

Provide a Map

Another way to use `CacheFlux` is by passing a `Map`. `CacheFlux` has another static `lookup` method that allows you to pass a `Map`.

```
    public static <KEY, VALUE>
1   FluxCacheBuilderMapMiss<VALUE> lookup(
2       Map<KEY, ? super List> cacheMap,
3       KEY key,
4       Class<VALUE> valueClass
5   )
```

The **Map** has to be passed as the first argument. The key type of the **Map** must be compatible with the type of the key passed as the second argument. The **Map**'s value type must be a **List** or another type that extends a **List**. Unfortunately, you cannot define a generic type for the **List**. However, it doesn't mean you can pass any value as the element of the **List**. The **List** can only contain Project Reactor's **Signal**. For the third argument, you have to pass a **Class** which is used to cast each **Flux** element.

Then, you need to handle cache misses by using one of the **onCacheMissResume** methods. One of the methods requires you to pass a **Supplier**, while the other requires you to pass a **Flux**. You should use the latter if you need to generate the values based on the given key. When a cache miss occurs, the **onCacehMissResume** method will be called and the resulting values will be stored to the **Map** automatically.

```
1   Flux<VALUE> onCacheMissResume(Supplier<Flux<VALUE>>
   otherSupplier);
2   Flux<VALUE> onCacheMissResume(Flux<VALUE> other);
```

To use **CacheFlux** by providing a **Map**, you need to have a **Map** whose value type is **List**.

```
final Map<Integer, List> mapCache = new HashMap<>();
```

Below is the usage example.

```
1  final Flux<Integer> cachedFlux2 = CacheFlux
2      .lookup(
3          mapCache,
4          key,
5          Integer.class
6      )
7      .onCacheMissResume(this::handleCacheMiss);
8  //      .onCacheMissResume(Flux.defer(() ->
9  this.handleCacheMiss(key)));
10
11 return cachedFlux2
    .doOnNext(res -> System.out.println("Value is " +
    res));
```

Using CacheMono and CacheFlux with *Caffeine*

If you need more advanced features for caching such as expiration time, most likely you'll use a cache library that provides the features you need. The **CacheMono** and **CacheFlux** can be used with any caching library, such as *Caffeine*. This tutorial doesn't explain how to use *Caffeine* in detail, as it will be very long to explain it. We only focus on how to use it with **CacheMono** and **CacheFlux**. First of all, we need to create a *Caffeine* cache. *Caffeine* has some classes for creating caches, such as **Cache**, **AsyncCache**, and **LoadingCache**. In this example, we are going to use the **Cache** class.

```
1  final Cache<String, String> caffeineCache =
2  Caffeine.newBuilder()
3      .expireAfterWrite(Duration.ofSeconds(30))
4      .recordStats()
5      .build();
```

As I have explained above, to use **CacheMono**, you can provide a **Map** whose value type is Reactor's **Signal**. If not possible, you need to pass a **Function** for looking up the value and handle write data to the cache

manually. *Caffeine* has a method for converting the cache into a **Map**, but the value type is not a **Signal**, which means it's not compatible with the lookup methods that accept a **Map** parameter. Therefore, you need to handle the lookup by passing a **Function** and store the generated values to the *Caffeine* cache.

```

1  final Mono<String> cachedMonoCaffeine = CacheMono
2      .lookup(
3          k ->
4      Mono.justOrEmpty(caffeineCache.getIfPresent(k)).map(Sig
5          key
6      )
7      //      .onCacheMissResume(this::handleCacheMiss) // Ur
8      this if you want to pass a Supplier
9      .onCacheMissResume(this.handleCacheMiss(key))
10     .andWriteWith((k, sig) -> Mono.fromRunnable(() ->
11         caffeineCache.put(k, Objects.requireNonNull(s
12     ));

```

The usage for **CacheFlux** is also similar. Below is another *Caffeine* cache whose value type is **List<Integer>**.

```

1  final Cache<Integer, List<Integer>> caffeineCache =
2      Caffeine.newBuilder()
3          .expireAfterWrite(Duration.ofSeconds(30))
4          .recordStats()
5          .build();

```

To use *Caffeine* cache with **CacheFlux**, you need to use the **lookup** method that accepts a **Function** as the first parameter. The passed **Function** is responsible to get the value from the cache or return an empty **Mono** if the key is not present. You also need to store the values to the *Caffeine* cache.

```

1  final Flux<Integer> cachedFluxCaffeine = CacheFlux
2      .lookup(
3          k -> {
4              final List<Integer> cached =
5                  caffeineCache.getIfPresent(k);
6
7                  if (cached == null) {
8                      return Mono.empty();
9                  }
10                 return Mono.just(cached)
11                     .flatMapMany(Flux::fromIterable)
12                     .map(Signal::next)
13                     .collectList();
14             },
15             key
16         )
17 //         .onCacheMissResume(this::handleCacheMiss)
18 // Uncomment this if you want to pass a Supplier
19 //         .onCacheMissResume(this.handleCacheMiss(key))
20 //         .andWriteWith((k, sig) -> Mono.fromRunnable(()
21 //             ->
22 //                 caffeineCache.put(
23 //                     k,
24 //                     sig.stream()
25 //                         .filter(signal -> signal.getType()
26 //                             == SignalType.ON_NEXT)
27 //                         .map(Signal::get)
28 //                         .collect(Collectors.toList())
29 //                     )
30 //             ));

```

Summary

That's how to use **CacheMono** and **CacheFlux** in Project Reactor. Basically, there are two options: handle lookup and store values manually or provide a compatible **Map** whose value type is a **Signal** (for **CacheMono**) or **List<Signal>** (for **CacheFlux**). The latter option

is preferred if possible because it's simpler. **CacheMono** and **CacheFlux** can also be used with any caching library such as *Caffeine*. The full code of this tutorial is available on GitHub.

- [CacheMono Examples](#)
- [CacheFlux Examples](#)

[Share on Facebook](#)[Share on Twitter](#)[Share on Tumblr](#)[Share by E-Mail](#)[Share on Pinterest](#)[Share on LinkedIn](#)[Share on Reddit](#)[Share on Hacker News](#)

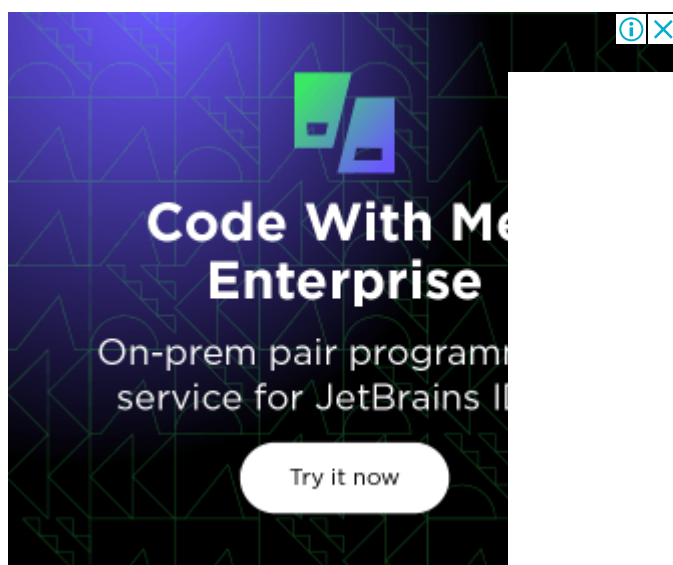
Ivan Andrianto



Ivan Andrianto is a software engineer and the founder of woolha.com.




📁 [Project Reactor, Java Spring](#)




Latest Posts

[Flutter - Using ReorderableListView Widget Examples](#)

 02 Oct 2021


[Flutter - Using TabBar & TabBarView Examples](#)

 22 Aug 2021

[Flutter - Creating New Project](#)

 13 Aug 2021

[Flutter - Upgrade/Downgrade Flutter SDK Version](#)

 11 Aug 2021

[Flutter - Check Internet Connection Examples](#)

 07 Aug 2021

Categories

[Dart](#)

- [Rx Dart](#)
-

[Deno](#)

[Flutter](#)

[Java](#)

[Node.js](#)

Copyright © 2019 Woolha

[Terms of Service](#) | [Privacy Policy](#)