**Lab 6**

**Caching with Spring Boot**

Building on top of the last lab, this lab will focus on caching. Adding caching to an operation is really easy, so let’s dive right in.

**Enable caching on the application**

Open /lab6/ProductService-start in IntelliJ or your favorite editor. This start situation is equal to the end situation of Lab 5.

To enable caching on the entire application we will need to add the ‘@EnableCaching’ annotation to the application class, like so:

@SpringBootApplication  
@EnableCaching  
public class ProductserviceApplication {

**Enable caching on the operations**

To use caching on any operation, we need to add annotations to the definition of the operation in its controller. Since all the operations we have defined are really fast, add the following bit of code to the top of the ‘retrieveProducts’ and ‘retrieveProduct’ operations to slow them down (by 3 seconds):

try {  
 long time = 3000L;  
 Thread.*sleep*(time);  
} catch (InterruptedException e) {  
 throw new IllegalStateException(e);  
}

To add caching to these operations we need to add a single annotation. Add the following annotation to the ‘retrieveProducts’ operaton:

@Cacheable(value="products")

This will create a cache by the name of ‘products’.

Lets also add caching to the ‘retrieveProduct’ operation. To do this, add the following annotation to the operation:

@Cacheable(value="product", key="#productId")

This will create a cache by the name of ‘products’ and will add values to the cache based on a productId. So now the cache will consist of a key-value pair of productId (key) coupled with the output of the operation (in this case a ‘Product’ object). We will later see why this is useful. For now, test the current setup. Note that the first time one of the retrieve operations is called, it will take at least 3 seconds. However, the second time we call the operation it returns with a reply almost instantaneous! Test what happens when after a fresh deployment of the application you retrieve all products (there should be no products available), then post a new product, and then retrieve all products again. The newly added product will not be returned by this service! The same is true if you retrieve a single specific product (‘retrieveProduct’). Try retrieving a product **before** submitting it in a post request. Since it doesn’t exist, nothing will be retrieved. Yet the cache for the productId you are submitting will be filled with a null value for the Product object!

**Empty/refresh cache on insert/update**

So, how do we fix this? Well, we can update/empty the cache each time we do a POST/PUT to the database (NOTE: this only works if your application is the only application altering the data). We can do this by adding annotations to the POST/PUT operations to evict (empty) the cache of the ‘retrieveProducts’ operation and update the cache of the ‘retrieveProduct’ operation for the specific product to be updated.

Do this by adding the following annotation to both the ‘addProduct’ and ‘modifyProduct’ operations:

@Caching(evict = {  
 @CacheEvict(value="products", allEntries=true)  
 }  
 ,  
 put = {  
 @CachePut(value="product", key="#product.productId")  
 }  
)

This will evict (empty) the ‘products’ cache entirely and update the ‘product’ cache for a specific product based on the key (since we stored the ‘product’ cache as a key-value pair). However, one problem arises with this approach. With the cacheput operation the entirety of the output of the operation is added to the cache for the given key. Since the output of both the ‘addProduct’ and ‘modifyProduct’ operations is an element of type ‘ResponseEntity<Product>’, this is the type of object that will be placed in the ‘product’ cache. The ‘retrieveProduct’ however expects elements of type ‘Product’ to be in the cache! We can fix this by changing the return element of the ‘retrieveProduct’ operation to ‘ResponseEntity<Product>’. Change the body of the ‘retrieveProduct’ operation to the following code (with the added bonus of adding a correct HttpStatus not found when the product does not exist):

@RequestMapping(value = "/{productId}", method = RequestMethod.*GET*)  
@Cacheable(value="product", key="#productId")  
public ResponseEntity<Product> retrieveProduct(@PathVariable("productId") String productId) {  
  
 try {  
 long time = 3000L;  
 Thread.*sleep*(time);  
 } catch (InterruptedException e) {  
 throw new IllegalStateException(e);  
 }  
 Optional<Product> product = productRepository.findByProductId(productId);  
 if (product.isPresent()){  
 return new ResponseEntity<>(product.get(),new HttpHeaders(), HttpStatus.*OK*) ;  
 }  
 else{  
 return new ResponseEntity<>(null,new HttpHeaders(), HttpStatus.*NOT\_FOUND*) ;  
 }  
}

Go ahead and play around with this for a while. See if you can get caching to work on the other operations!

In this example we have not set a specific caching provider. Setting a caching provider (for example Redis) will give you much more fine-grained control by extending an interface or two. For example, you can set a custom time-to-live on cache entries to evict entries after a certain time. Also, currently our implementation will propably not work when multiple instances of the service are running simultaneously since the cache is not shared. For more information on the different caching providers, check out:

<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-caching.html>