Async Request processing in Spring

1. Overview

Asynchronous support was introduced in Servlet 3.0 and, simply put, it allows processing an HTTP request in another thread than the request receiver thread. *DeferredResult,* available from Spring 3.2 onwards, assists in offloading a long-running computation from an http-worker thread to a separate thread.

Although the other thread will take some resources for computation, the worker threads are not blocked in the meantime and can handle incoming client requests.

The async request processing model is very useful as it helps scale an application well during high loads, especially for IO intensive operations.

2. Setup

In AppConfig.java override configureAsyncSupport

```
/* To support async processing
 *
 *
*/
@Override
protected void configureAsyncSupport(AsyncSupportConfigurer asyncSupportConfigurer) {
	asyncSupportConfigurer.setDefaultTimeout(5000);
	//this set method needs an instance of AsyncTaskExecutor
	// so we will get it via a Bean mvcTaskExecutor
	asyncSupportConfigurer.setTaskExecutor(mvcTaskExecutor());
}

//Bean for returning the AsyncTaskExecutor
@Bean
public AsyncTaskExecutor mvcTaskExecutor() {
		ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
		taskExecutor.setThreadNamePrefix("hplus-thread-");
	return taskExecutor;
}
```

3. Blocking REST Controller

Let's start with developing a standard blocking Controller:

```
//DeferredResult Implementation of the controller

	@Autowired
	private AsyncTaskExecutor taskExecutor;

	DeferredResult<String> deferredResult = new DeferredResult<>();

	@GetMapping("/search")
	public DeferredResult<String> search(@RequestParam("search")String search,Model
model, HttpServletRequest request ) {
		System.out.println("Inside search controller");
		System.out.println("Search Criteria "+search);
		System.out.println("Is Async supported "+request.isAsyncSupported());
		System.out.println("Thread from controller
"+Thread.currentThread().getName());
		taskExecutor.execute(()->{
			try {
			Thread.sleep(3000);
			//To invoke AsyncRequestTimeoutError and return error.jsp
			//Thread.sleep(7000);
			}catch(Exception e) {
				e.printStackTrace();
			}
			System.out.println("Async Thread "+Thread.currentThread().getName());
			List<Product> products = new ArrayList<Product>();
			products = productRepository.searchByName(search);
			if(products==null)
				throw new ApplicationException("No product found with search
key "+search);
			model.addAttribute("products",products);
			deferredResult.setResult("search");
		});
		return deferredResult;

	}
```

Request processing is done in a separate thread and once completed we invoke the *setResult* operation on the *DeferredResult* object.

Let's look at the log output to check that our threads behave as expected:

```
[nio-8080-exec-6] com.baeldung.controller.AsyncDeferredResultController:
Received async-deferredresult request
[nio-8080-exec-6] com.baeldung.controller.AsyncDeferredResultController:
Servlet thread freed
[nio-8080-exec-6] java.lang.Thread : Processing in separate thread
```

Internally, the container thread is notified and the HTTP response is delivered to the client. The connection will remain open by the container(servlet 3.0 or later) until the response arrives or times out.

4. *DeferredResult* Callbacks

We can register 3 types of callbacks with a DeferredResult: completion, timeout and error callbacks.

Let's use the *onCompletion()* method to define a block of code that's executed when an async request completes:

```
deferredResult.onCompletion(() -> LOG.info("Processing complete"));
```

Similarly, we can use *onTimeout()* to register custom code to invoke once timeout occurs. In order to limit request processing time, we can pass a timeout value during the *DeferredResult* object creation:

```
DeferredResult<ResponseEntity<?>> deferredResult = new
DeferredResult<>(500l);

deferredResult.onTimeout(() ->
  deferredResult.setErrorResult(
    ResponseEntity.status(HttpStatus.REQUEST_TIMEOUT)
      .body("Request timeout occurred.")));
```

In case of timeouts, we're setting a different response status via timeout handler registered with *DeferredResult*.

Let's trigger a timeout error by processing a request that takes more than the defined timeout values of 5 seconds:

```

ForkJoinPool.commonPool().submit(() -> {
    LOG.info("Processing in separate thread");
    try {
        Thread.sleep(6000);
    } catch (InterruptedException e) {
        ...
    }
    deferredResult.setResult(ResponseEntity.ok("OK")));
});

```

Let's look at the logs:

```
[nio-8080-exec-6] com.baeldung.controller.DeferredResultController:
servlet thread freed
[nio-8080-exec-6] java.lang.Thread: Processing in separate thread
[nio-8080-exec-6] com.baeldung.controller.DeferredResultController:
```

```
Request timeout occurred

```

There will be scenarios where long-running computation fails due to some error or exception. In this case, we can also register an *onError()* callback:

```
deferredResult.onError((Throwable t) -> {
    deferredResult.setErrorResult(
      ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
        .body("An error occurred."));
});
```

In case of an error, while computing the response, we're setting a different response status and message body via this error handler.