1. Goals

The goal of this lesson is to introduce a simple solution for protecting the system against abusive clients.

2. Lesson Notes

First, it's important to understand that this is a good base to build on, and not something you go pick up and go right into production with.

We're naturally going to start with the foundations - understanding what API Throttling is and why we need it.

On the implementation side of things, we're going use Guava and also make good use of the AOP support in Spring.

2.1. Intro to Rate Limiting

Simply put - API throttling/limiting is making sure that a single user can only access the system for fair use.

We of course need to be careful with what "fair use" is and not limit "well behaved" clients from consuming the API.

But, beyond fair use - we need to make sure that abusive clients are stopped (or at least slowed down) from accessing the system.

2.2. The Simple Implementation

We're going to drive the implementation using Aspect Oriented Programming and the Guava Rate Limiter

Let's first define the annotation:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RateLimit {
    /** @return rate limit in queries per second */
    int value();
    /** @return rate limiter identifier (optional) */
    String key() default "";
}
```

So as you can immediately see, the annotation itself is pretty simple.

We only have the optional key - in case we want to group limited operations together by using a common (or a shared) key - and then we have the value - which is the actual limit.

Now, let's define the actual aspect that's going to drive this annotation at runtime:

```
@Aspect
@Component
public class RateLimiterAspect {
    @Before("@annotation(limit)")
    public void rateLimit(JoinPoint jp, RateLimit limit) {
        // ...
    }
}
```

Notice that this is just the skeleton of an implementation - so let's start filling it in.

First, we're going to store our rate limiters into a concurrent HashMap:

```
private final Map<String, RateLimiter> limiters = new ConcurrentHashMap<>();
```

Now, we're going to set up the actual logic in the aspect:

```
@Before("@annotation(limit)")
public void rateLimit(JoinPoint jp, RateLimit limit) {
    String key = getOrCreateKey(jp, limit);
    RateLimiter limiter = limiters.computeIfAbsent(
      key, (name -> RateLimiter.create(limit.value())));
    limiter.acquire();
}
private String getOrCreateKey(JoinPoint jp, RateLimit limit) {
    if( limit.key() == null ){
        return limit.key();
    }
    return JoinPointToStringHelper.toString(jp);
}
```

First we retrieve the key from the annotation (the key was optional, so if there's no key, we'll have to generate one).

Then based on the key, we do the same thing - we go into the map and get the limiter associated with it (or we create one).

Finally, we're acquiring the limiter.

2.3. Using the Annotation

OK, so now that we’re good to go - let’s go to the PrivilegeController and let’s use the annotation.

We’re going to basically annotate the *findOne* API with:

```
@RateLimit(1)
```

So we're basically ensuring that no client can send more than 1 request per second to this particular API.

That's of course a very strict limit - one that is not a good idea for a production environment - but we’re using it here so that the limitation is immediately apparent when we test things out.

2.4. A Live Test

Now, with everything set up - write a simple live test and hit that rate-limited operation:

```
@Test
public void whenSingleResourceIsRetrievedMultipleTimes_thenThrottled() {
    // Given
    String uriOfExistingResource = getApi().createAsUri(createNewResource());
    ExecutorService executor = Executors.newCachedThreadPool();
    // When
    for (int i = 0; i < 10; i++) {
        executor.submit(() -> {
            getApi().read(uriOfExistingResource);
        });
    }
}
```

So we're first creating a new resource and then hitting the limited operation and trying to retrieve that resource concurrently.

When we run this test we'll be able to easily see that the operation is actually limiting our throughtput and only allowing us, as a client - to hit it once per second.

3. Resources

- Aspect Oriented Programming with Spring (the Spring reference)

- The Guava Rate Limiter (javadoc)