- Spring Cache
  - Cache Managers
- Spring Boot Actuator
  - Health
  - Metrics
  - Loggers
  - Security
  - Prometheus
- Spring Sleuth / Zipkin
- Reactive programming
- DeferredResult (Servlet 3.0 spec)
- Dependency Injection
- Springdoc OpenAPI (Swagger)
- Vulnerabilities
  - log4shell
    - CVE-2021-44228
    - CVE-2021-45046, CVE-2021-45105
  - spring4shell
    - CVE-2022-22965

Spring Cache

To enable caching:

- @EnableCaching
- Declare a bean implementing CacheManager interface

In Spring Boot, spring-boot-starter-cache + @EnableCaching would register the ConcurrentMapCacheManager.

Also, we can customize the auto-configured CacheManager using one or more **CacheManagerCustomizer**<T> beans:

```
@Component
public class SimpleCacheCustomizer
  implements CacheManagerCustomizer<ConcurrentMapCacheManager> {

        @Override
        public void customize(ConcurrentMapCacheManager cacheManager) {
            cacheManager.setCacheNames(asList("users", "transactions"));
        }
}
```

The CacheAutoConfiguration auto-configuration picks up these customizers and applies them to the current CacheManager before its complete initialization.

—

Key generation

Spring Cache uses **SimpleKeyGenerator** to calculate the key to be used for retrieving or updating an item in the cache from the method parameters. It's also possible to define a custom key generation by specifying a SpEL expression in the key attribute of the @Cacheable annotation, or by implementing the KeyGenerator interface.

```
public static Object generateKey(Object... params) {
        if (params.length == 0) {
                return SimpleKey.EMPTY;
        }
        if (params.length == 1) {
                Object param = params[0];
                if (param != null && !param.getClass().isArray()) {
                        return param;
                }
        }
        return new SimpleKey(params); //inside: Arrays.deepHashCode(params);
   }
```

SimpleKeyGenerator implementation

Cacheable

```
@Cacheable("addresses")
public String getAddress(Customer customer) {...}
```

The getAddress() call will first check the cache *addresses* before actually invoking the method and then caching the result. While in most cases one cache is enough, the Spring framework also supports multiple caches to be passed as parameters:

```
@Cacheable({"addresses", "directory"})
public String getAddress(Customer customer) {...}
```

In this case, if any of the caches contain the required result, the result is returned and the method is not invoked.

—

CacheEvict

If we delete data from our primary storage, we would have stale data in the cache. We can annotate the delete() method to update the cache:

```
@CacheEvict(value = "cars", key = "#uuid")
  public void delete(UUID uuid) {
        carRepository.deleteById(uuid);
  }
```

The @CacheEvict annotation deletes the data from the cache. We can define the key that is used to identify the cache item that should be deleted. We can delete all entries from the cache if we set the attribute allEntries to true.

—

CachePut

The data in the cache is just a copy of the data in the primary storage. If this primary storage is changed, the data in the cache may become stale. We can solve this by using the @CachePut annotation:

```
@CachePut(value = "cars", key = "#car.id")
  public Car update(Car car) {
        return carRepository.save(car);
  }
```

The body of the update() method will always be executed. Spring will put the result of the method into the cache. In this case, we also defined the key that should be used to update the data in the cache.

The difference between @Cacheable and @CachePut is that @Cacheable will skip running the method, whereas @CachePut will actually run the method and then put its results in the cache.

—

CacheConfig

With the @CacheConfig annotation, we can streamline some of the cache configuration into a single place at the class level, so that we don't have to declare things multiple times:

```
@CacheConfig(cacheNames={"addresses"})
public class CustomerDataService {
        @Cacheable
        public String getAddress(Customer customer) {...}
```

Cache Managers

ConcurrentMapCacheManager - CacheManager implementation that lazily builds ConcurrentMapCache instances for each getCache(java.lang.String) request. Also supports a 'static' mode where the set of cache names is pre-defined through setCacheNames(java.util.Collection), with no dynamic creation of further cache regions at runtime.

if you know how many elements there will be in your map, and you know they never expire unless you replace them, then a ConcurrentHashMap will do what you need.

—

EhCacheCacheManager - CacheManager backed by an EhCache CacheManager.

Ehcache (on-heap) is a ConcurrentHashMap with **expiration and eviction handling**.

It is also **JSR107** compliant so it makes it easier to have transparent caching with many frameworks.

Bottom line, Ehcache is an enhanced ConcurrentHashMap.

Spring Boot Actuator

Spring Boot Actuator module helps you monitor and manage your Spring Boot application by providing production-ready features like

- health check-up
- auditing
- metrics gathering
- HTTP tracing etc.

All of these features can be accessed over **JMX** or **HTTP** endpoints.

Actuator also integrates with external application monitoring systems like:

- Prometheus
- Graphite
- DataDog
- Influx
- Wavefront
- New Relic and many more.

These systems provide you with excellent dashboards, graphs, analytics, and alarms to help you monitor and manage your application from one unified interface

Actuator uses **Micrometer** - an application metrics facade to integrate with these external application monitoring systems. This makes it super easy to plug-in any application monitoring system with very little configuration.

Example actuator endpoints:

- /**actuator** - list all the actuator endpoints exposed over HTTP
- /**health** - provides basic information about the application's health
- /**metrics** - shows several useful metrics information like JVM memory used, system CPU usage, open files, etc
- /**loggers** endpoint shows the application's logs and also lets you change the log level at runtime

You can see the complete list of the endpoints on the official documentation page:
https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html

Note that every actuator endpoint can be explicitly enabled and disabled. Moreover, the endpoints also need to be exposed over HTTP or JMX to make them remotely accessible.

By default, **all endpoints** except for shutdown are **enabled**.

By default, only the **health** and **info** endpoints are **exposed** over HTTP.

For example, to enable the shutdown endpoint, add the following to your application.properties file:

```
management.endpoint.shutdown.enabled=true
```

Here is how you can expose actuator endpoints over HTTP and JMX using application properties:

```
#Exposing Actuator endpoints over HTTP
# Use "*" to expose all endpoints, or a comma-separated list to expose selected ones
management.endpoints.web.exposure.include=health,info
management.endpoints.web.exposure.exclude=

#Exposing Actuator endpoints over JMX
# Use "*" to expose all endpoints, or a comma-separated list to expose selected ones
management.endpoints.jmx.exposure.include=*
management.endpoints.jmx.exposure.exclude=
```

Health

The /health endpoint checks the health of your application by combining several health indicators. It uses these health indicators as part of the health check-up process.

Spring Boot Actuator comes with several predefined health indicators:

- DataSourceHealthIndicator
- DiskSpaceHealthIndicator
- MongoHealthIndicator
- RedisHealthIndicator
- CassandraHealthIndicator etc.

You can also disable a particular health indicator using application properties:

```
management.health.mongo.enabled=false
```

The health endpoint only shows a simple UP or DOWN status. To get the complete details including the status of every health indicator that was checked as part of the health check-up process, add the following property in the application.properties:

```
management.endpoint.health.show-details=always
```

Metrics

The /metrics endpoint lists all the metrics that are available for you to track.

To get the details of an individual metric, you need to pass the metric name in the URL like this -

```
http://localhost:8080/actuator/metrics/{MetricName}
```

For example, to get the details of system.cpu.usage metric, use the URL:

```
http://localhost:8080/actuator/metrics/system.cpu.usage
```

Loggers

The loggers endpoint, which can be accessed at http://localhost:8080/actuator/**loggers**, displays a list of all the configured loggers in your application with their corresponding log levels.

You can also view the details of an individual logger by passing the logger name in the URL like this:

```
http://localhost:8080/actuator/loggers/{name}
```

For example, to get the details of the root logger, use the URL http://localhost:8080/actuator/loggers/**root**.

The loggers endpoint also allows you to **change the log level of a given logger in your application at runtime**.

For example, To change the log level of the root logger to DEBUG at runtime, make a POST:

```
POST http://localhost:8080/actuator/loggers/root
{
  "configuredLevel": "DEBUG"
}
```

This functionality will really be useful in cases when your application is facing issues in production and you want to enable DEBUG logging for some time to get more details about the issue.

To reset the log-level to the default value, you can pass a value of null in the configuredLevel field.

Security

If Spring Security is present in your application, then the endpoints are secured by default using a form-based HTTP basic authentication.

Override existing security configuration:

```
@Override
	protected void configure(HttpSecurity http) throws Exception {
	http
		.authorizeRequests()
		  .requestMatchers(EndpointRequest.to(ShutdownEndpoint.class))
		    .hasRole("ACTUATOR_ADMIN")
		  .requestMatchers(EndpointRequest.toAnyEndpoint())
		    .permitAll()
		  .requestMatchers(PathRequest.toStaticResources().atCommonLocations())
		    .permitAll()
		  .antMatchers("/")
		    .permitAll()
		  .antMatchers("/**")
		    .authenticated()
		.and()
		  .httpBasic();
	}
```

To be able to test the above configuration with HTTP basic authentication, you can add a default spring security user in application.properties:

```
# Spring Security Default username and password
spring.security.user.name=actuator
spring.security.user.password=actuator
spring.security.user.roles=ACTUATOR_ADMIN
```

Prometheus

Prometheus is an open-source monitoring system that was originally built by SoundCloud. It consists of the following core components -

- A **data scraper** that pulls metrics data over HTTP periodically at a configured interval.
- A time-series **database** to store all the metrics data.
- A simple **user interface** where you can visualize, query, and monitor all the metrics.

Actuator endpoint: /actuator/prometheus

Spring Sleuth / Zipkin

https://howtodoinjava.com/spring-cloud/spring-cloud-zipkin-sleuth-tutorial/

Sleuth is a tool from the **Spring cloud** family. It is used to generate the trace id, span id and add this information to the service calls in the headers and MDC, so that It can be used by tools like Zipkin and ELK etc. to store, index and process log files.

As it is from the spring cloud family, once added to the CLASSPATH, it automatically integrated to the common communication channels like:

- requests made with the RestTemplate etc.
- requests that pass through a Netflix Zuul microproxy
- HTTP headers received at Spring MVC controllers
- requests over messaging technologies like Apache Kafka or RabbitMQ etc.

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

Zipkin was originally developed at Twitter, based on a concept of a Google paper that described Google's internally-built distributed app debugger – dapper. It manages both the collection and lookup of this data. To use Zipkin, applications are instrumented to report timing data to it.

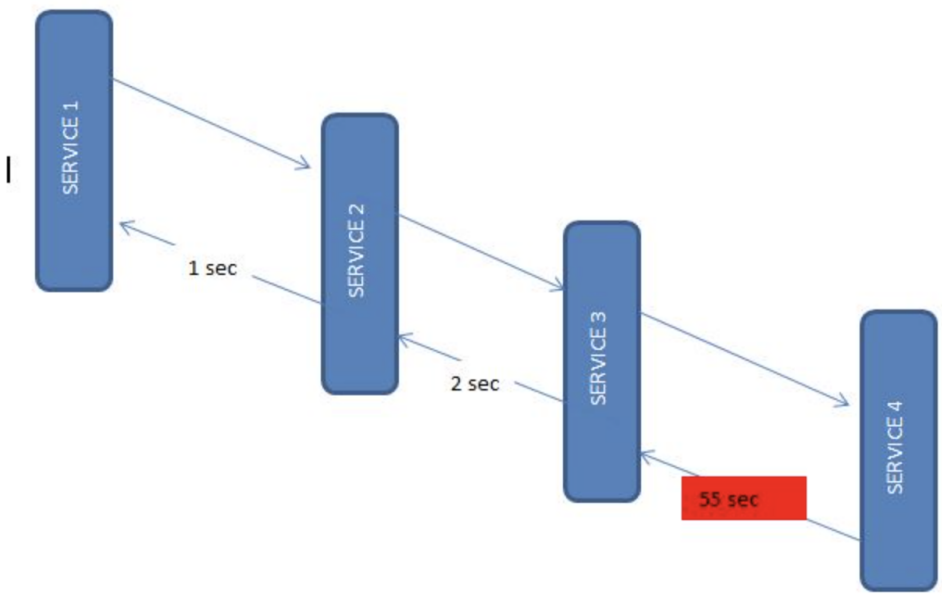
If you are troubleshooting latency problems or errors in an ecosystem, you can filter or sort all traces based on the application, length of trace, annotation, or timestamp. By analyzing these traces, you can decide which components are not performing as per expectations, and you can fix them.

Internally it has 4 modules:

- Collector – Once any component sends the trace data, it arrives to Zipkin collector daemon. Here the trace data is validated, stored, and indexed for lookups by the Zipkin collector.
- Storage – This module store and index the lookup data in backend. **Cassandra, ElasticSearch and MySQL** are supported.
- Search – This module provides a simple JSON API for finding and retrieving traces stored in the backend. The primary consumer of this API is the Web UI.
- Web UI – A very nice UI interface for viewing traces.

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```



Reactive programming

https://www.baeldung.com/spring-webflux-concurrency
https://www.educba.com/rxjava-vs-reactor/
https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-new-framework

DeferredResult (Servlet 3.0 spec)

DeferredResult provides an alternative to using a **Callable** for asynchronous request processing. While a Callable is executed concurrently on behalf of the application, with a DeferredResult the application can produce the result from a thread of its choice.

Your controller is eventually a function executed by the servlet container (I will assume it is Tomcat) worker thread. Your service flow starts with Tomcat and ends with Tomcat. Tomcat gets the request from the client, holds the connection, and eventually returns a response to the client. Your code (controller or servlet) is somewhere in the middle.

Consider this flow:

- Tomcat get client request.
- Tomcat executes your controller.
- Release Tomcat thread but keep the client connection (don't return response) and run heavy processing on different thread.
- When your heavy processing complete, update Tomcat with its response and return it to the client (by Tomcat).

Because the servlet (your code) and the servlet container (Tomcat) are different entities, then to allow this flow (releasing tomcat thread but keep the client connection) we need to have this support in their contract, the package javax.servlet, which introduced in **Servlet 3.0**.

About servlet container worker threads: we can configure Tomcat's server thread pool:

```
server.tomcat.threads.min-spare: 10
server.tomcat.threads.max: 200
```

minSpare is the smallest the pool will be, including at startup. maxThreads is the largest the pool will be before the server starts queueing up requests.

Spring Boot defaults these to 10 and 200, respectively.

How and when to use DeferredResult?

With DeferredResult you can choose any thread (or threadpool) to run your async code, which has some beneficial advantages:

- Http worker threads released immediately by passing on the work to the non HTTP thread.
- The worker threads are able to handle additional HTTP request without waiting for the long-running process to finish the work.

Dependency Injection

“@Lazy” - Indicates whether a bean is to be lazily initialized. Can be used as hot fix for circular dependency.

Springdoc OpenAPI (Swagger)

Brief overview: https://www.youtube.com/watch?v=utRxyPfFlDw&ab_channel=SpringI%2FO

springfox –> springdoc migration guide: https://springdoc.org/#migrating-from-springfox

springdoc documentation: https://reflectoring.io/spring-boot-springdoc/

Vulnerabilities

log4shell

CVE-2021-44228

https://www.lunasec.io/docs/blog/log4j-zero-day/

Almost all versions of log4j version 2 are affected: 2.0-beta9 <= Apache log4j <= 2.14.1

Exploit Requirements

- A server with a vulnerable log4j version (listed above).
- An endpoint with any protocol (HTTP, TCP, etc), that allows an attacker to send the exploit string.
- A log statement that logs out the string from that request.

CVE-2021-45046, CVE-2021-45105

https://www.cyberkendra.com/2021/12/3rd-vulnerability-on-apache-log4j.html

https://nvd.nist.gov/vuln/detail/CVE-2021-45046

When the logging configuration uses a non-default Pattern Layout with a Context Lookup (for example, **$${ctx:loginId}**), attackers with control over Thread Context Map (MDC) input data can craft malicious input data that contains a recursive lookup, resulting in a StackOverflowError that will terminate the process. This is also known as a DOS (Denial of Service) attack.

spring4shell

CVE-2022-22965

https://www.cyberkendra.com/2022/03/springshell-rce-0-day-vulnerability.html

This vulnerability is NOT as bad a Log4Shell. All attack scenario are more complex because of the nature of Class Loader Manipulation attacks work in Java. Spring4Shell exploitation requires deep Java knowledge to get a functioning POC. Class Loader Manipulation is more complicated to understand than the Log4Shell vulnerability.

Exploitation requires an endpoint with DataBinder enabled (e.g. a POST request that decodes data from the request body automatically) and **depends heavily on the servlet container** for the application. For example, when Spring is deployed to Apache Tomcat, the **WebAppClassLoader** is accessible, which allows an attacker to call getters and setters to ultimately write a malicious JSP file to disk. However, if Spring is deployed using the Embedded Tomcat Servlet Container the classloader is a LaunchedURLClassLoader which has limited access.

the Spring team has fixed the vulnerability and released the latest versions of

- Spring Boot 2.6.6 and 2.5.12 that depend on
- Spring Framework 5.3.18