Lab 9

Part 1

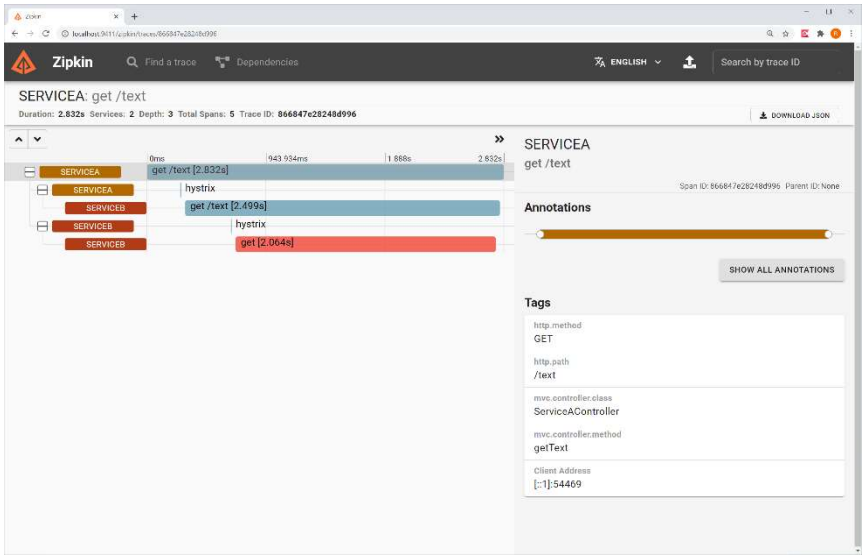
First download zipkin from:
https://drive.google.com/file/d/1WxBvlwhiAZMSkZxnUllKoJ3jSjtZiLPH/view?usp=sharing

Windows: Start Zipkin by double clicking the file **C:\springcloudtraining\zipkin\startzipkin.bat**

Mac: start zipkin with the command **java -jar zipkin.jar**

In the ProductService to the StockService that you wrote in lab 8 add the Zipkin and Sleuth libraries and configuration.
Then open the zipkin console on **http://localhost:9411/zipkin**

You can click the search button, then see the traces between the services, and you can see the dependencies between the services.



Part 2

First we need to install the ELK stack.

Window users can download it from:

https://drive.google.com/file/d/1mvMaAVdfUnG11sTn3m91OykCkK5a2iWp/view?usp=sharing

Elasticsearch

Start elasticsearch by dubble clicking **…\elasticsearch-7.10.0\bin\elasticsearch.bat**

Logstash

Start logstash by dubble clicking **…\logstash-7.10.1\startLogstash.bat**

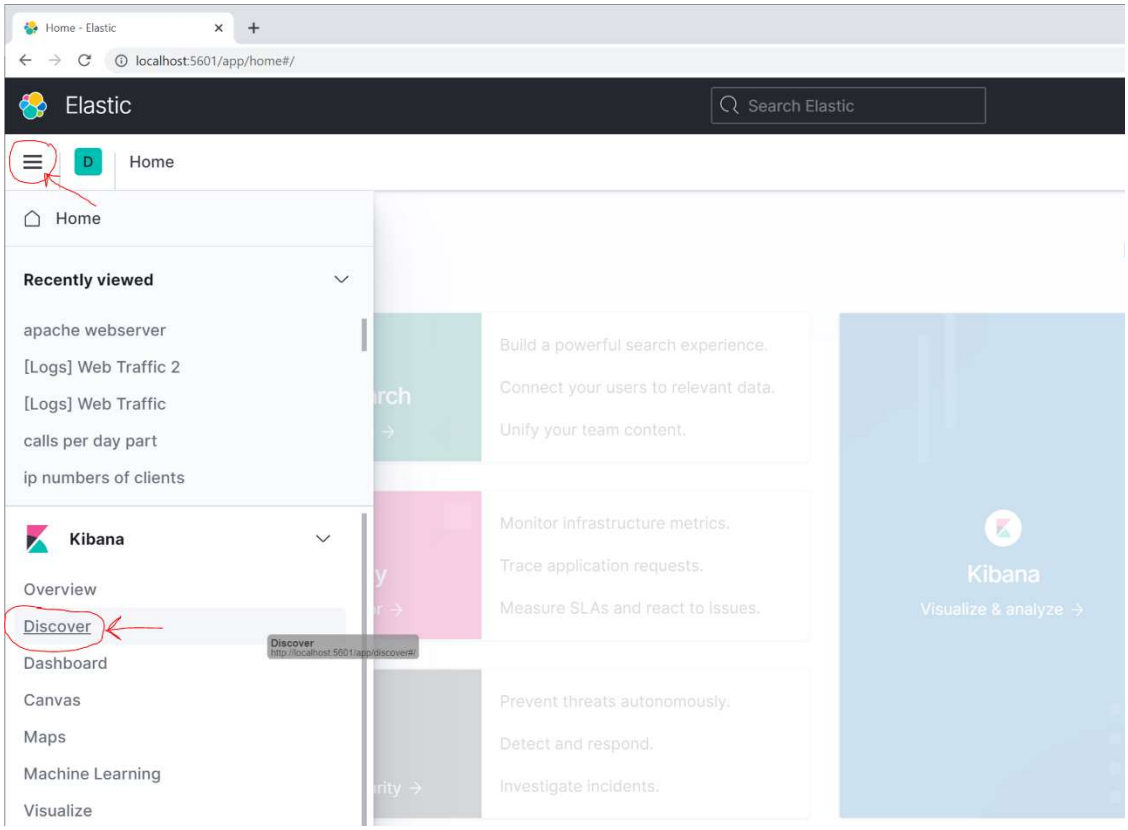
Kibana

Start kibana by dubble clicking **…\kibana-7.10.0-windows-x86_64\startkibana.bat**

Mac users can follow these steps**: https://www.devglan.com/elk/install-elk-mac**

Wait till kibana has started.

Open the browser on **http://localhost:5601** and click the **Discover** tab



Open the file **…\logstash-6.10.1/logstash.conf:**

```
input {
  file {
    type => "java"
    path => " C:/springcloudtraining/temp/spring-boot-elk.log"
    codec => multiline {
      pattern => "^%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME}.*"
      negate => "true"
      what => "previous"
    }
  }
}

output {
  stdout {
    codec => rubydebug
  }
   file {
   path => " C:/springcloudtraining/temp/testlog.log"
   create_if_deleted => true
  }

  # Sending log events to elasticsearch
  elasticsearch {
    hosts => ["localhost:9200"]
  }
}
```

Logstash will monitor log messages in the file **C:/springcloudtraining/temp/spring-boot-elk.log** and then write these messages to its console, to the file **C:/springcloudtraining/temp/testlog.log** and send them to elasticsearch.

Now we have to create a spring boot service that writes log messages to **C:/springcloudtraining/temp/spring-boot-elk.log**

Given is the Eclipse project **ServiceOne.** It is configured to write log messages to **C:/springcloudtraining/temp/spring-boot-elk.log**

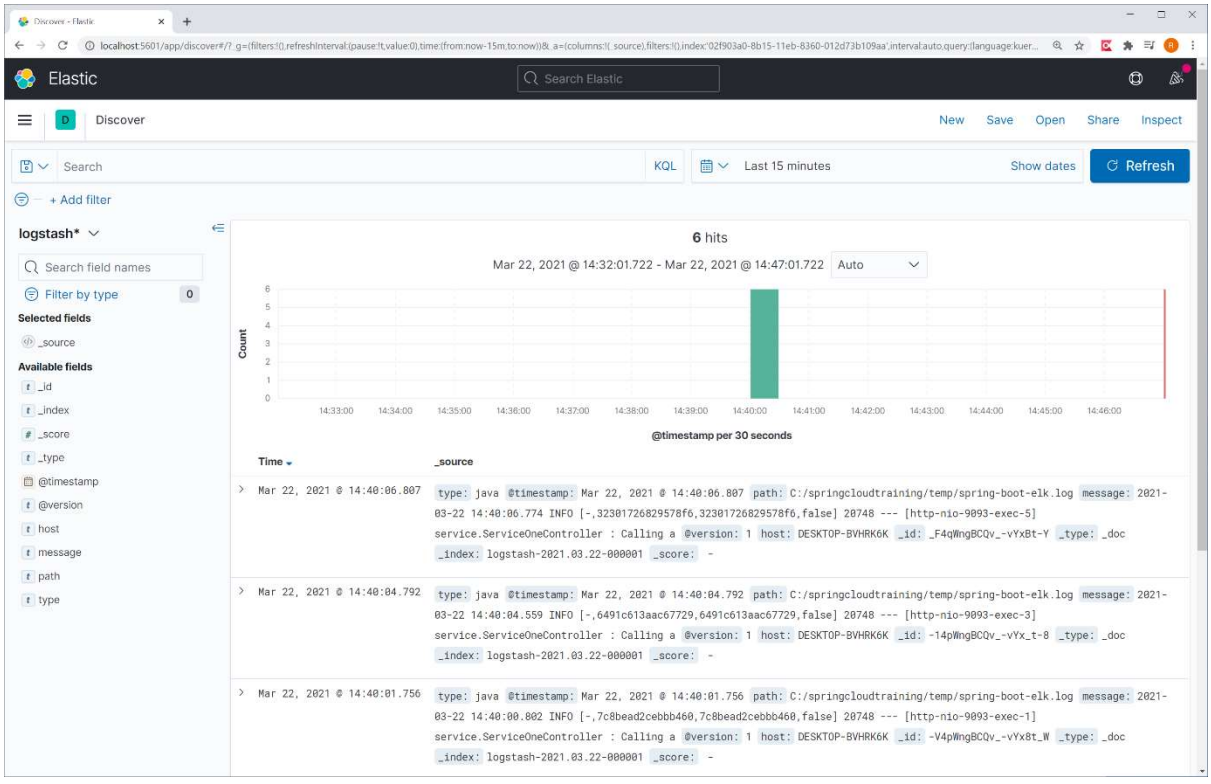
application.properties;

```
logging.file=C:/springcloudtraining/temp/spring-boot-elk.log

server.port=9093
```

```
@RestController
public class ServiceOneController {
      private static final Logger logger =
LoggerFactory.getLogger(ServiceOneController.class.getName());

      @RequestMapping("/a")
    public String one() {
        logger.info("Calling a");
        return "Calling a";
    }

      @RequestMapping("/b")
    public String two() {
        logger.debug("Calling b");
        return "Calling b";
    }
}
```

Start the application.

Anytime you access the webpage, a log is written to the logfile
Logstash sends the log to elasticsearch and the logs can be displayed in kibana.



You can also click the Auto-Refresh option in the upper hand corner on the right so that Kibana will update itself every 5 seconds.

Now create a new **ServiceTwo** project, and let it write its output to **C:/springcloudtraining/temp/spring-boot-elk2.log**

Then we have to tell logstash to also monitor this log file. Modify the file: **…\logstash-6.7.0/logstash.conf** so that we also see the logging of ServiceTwo in kibana

Part 3

Add the Hystrix circuit breaker to the remote call from the ProductService to the StockService. Test its working.

Part 4

Given is the project **EvenoddApplication**. This project contains the following controller:

```
@RestController
public class EvenOddController {

    @GetMapping("/validate")
    public String isNumberPrime(@RequestParam("number") Integer number) {
        return number % 2 == 0 ? "Even" : "Odd";
    }
}
```

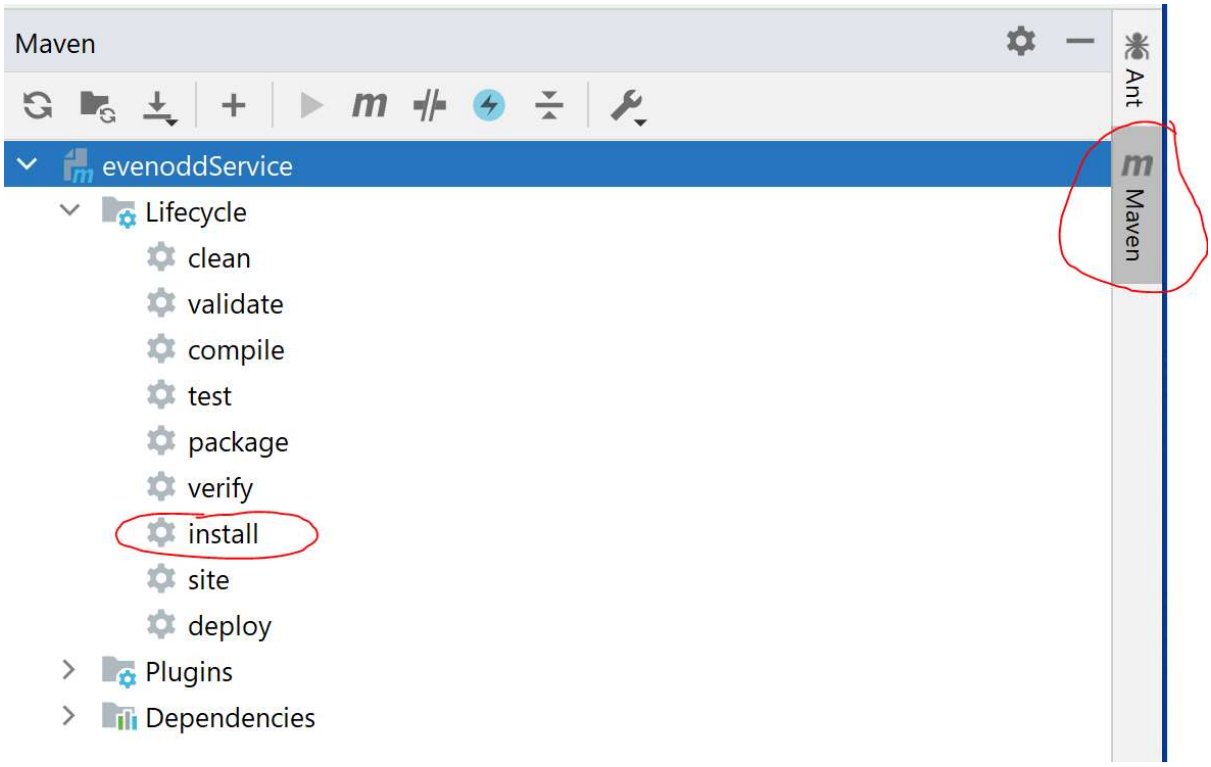
And the following contracts (in test/resources/contracts):

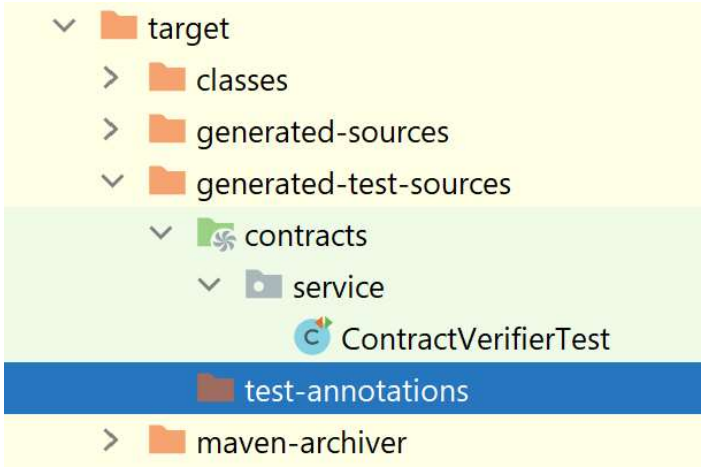
```
import org.springframework.cloud.contract.spec.Contract
Contract.make {
      description "should return even when number input is even"
      request{
            method GET()
            url("/validate") {
                  queryParameters {
                        parameter("number", "2")
                  }
            }
      }
      response {
            body("Even")
            status 200
      }
}
```

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
      description "should return odd when number input is odd"
      request {
            method GET()
            url("/validate") {
                  queryParameters {
                        parameter("number", "1")
                  }
            }
      }
      response {
            body("Odd")
            status 200
      }
```

```
}
```

If you open the Mavan tab on the right-hand side of IntelliJ and you run the **Maven install** command, then you see that a test class is created and that these test runs without errors.






```
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, T
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
```

In this project, add the following contract:

```
import org.springframework.cloud.contract.spec.Contract
Contract.make {
	description "add 2 numbers"
	request{
		method GET()
		url("/add") {
			queryParameters {
				parameter("value1", "2")
				parameter("value2", "5")
			}
		}
	}
	response {
		body("7")
		status 200
	}
}
```

Also write a new Controller class that implements this contract.

Modify the BaseTestClass as follows:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@DirtiesContext
@AutoConfigureMessageVerifier
public class BaseTestClass {

    @Autowired
    private EvenOddController evenOddController;

    @Autowired
    private CalculatorController calculatorController;

    @BeforeEach
    public void setup() {
        StandaloneMockMvcBuilder standaloneMockMvcBuilder =

MockMvcBuilders.standaloneSetup(evenOddController,calculatorController);
        RestAssuredMockMvc.standaloneSetup(standaloneMockMvcBuilder);
    }
}
```

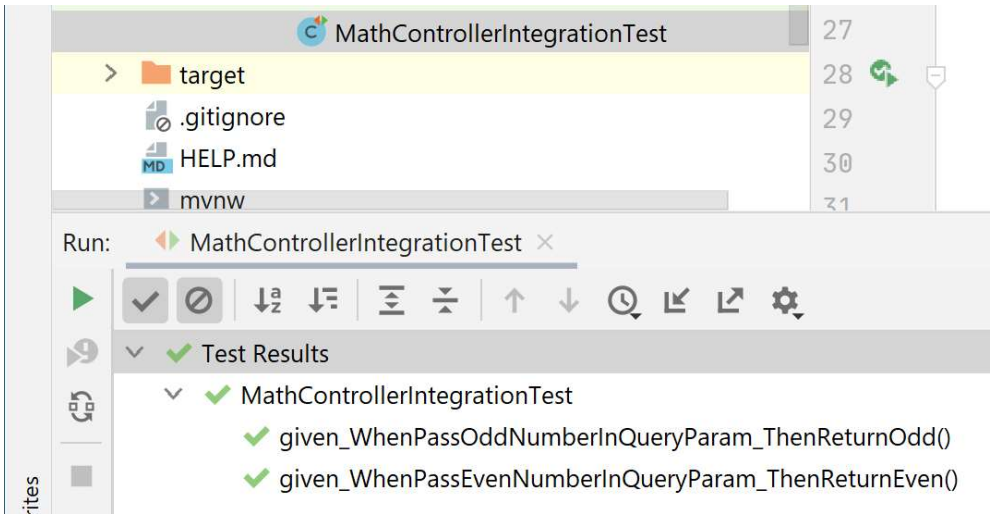
Run another Maven install, and see that the generated test class has another test method, and that this new test runs fine.

```
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
```

Also given is the **mathService** project that is the consumer of the evenoddService. The mathService contains the following test:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
@AutoConfigureJsonTesters
@AutoConfigureStubRunner(stubsMode = StubRunnerProperties.StubsMode.LOCAL,
        ids = "com.acme:evenoddService:+:stubs:8090")
public class MathControllerIntegrationTest {

          @Autowired
          private MockMvc mockMvc;

          @Test
          public void
given_WhenPassEvenNumberInQueryParam_ThenReturnEven() throws Exception {


mockMvc.perform(MockMvcRequestBuilders.get("/calculate?number=2")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().string("Even"));
          }

          @Test
          public void given_WhenPassOddNumberInQueryParam_ThenReturnOdd()
throws Exception {


mockMvc.perform(MockMvcRequestBuilders.get("/calculate?number=1")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().string("Odd"));
          }
}
```

This test uses the generated stubs from the evenoddService. When you run this test class as a JUnit test you see that the tests are correct.



Now we are going to modify the evenoddService. Change the contracts as follows:

```
import org.springframework.cloud.contract.spec.Contract
Contract.make {
	description "should return even when number input is even"
	request{
		method GET()
		url("/validate") {
			queryParameters {
				parameter("number1", "2")
				parameter("number2", "2")
			}
		}
	}
	response {
		body("Even")
		status 200
	}
}
```

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
	description "should return odd when number input is odd"
	request {
		method GET()
		url("/validate") {
			queryParameters {
				parameter("number1", "1")
				parameter("number2", "1")
			}
		}
	}
	response {
		body("Odd")
		status 200
	}
}
```

Run a Maven install on the POM file so that new stubs get generated. In the output we see that our local test fails.
Modify the controller so that the tests will pass.

```
	@GetMapping("/validate")
    public String isNumberPrime(@RequestParam("number1") Integer
                number1,@RequestParam("number2") Integer number2) {
        return number1 % 2 == 0  &&  number1 % 2 == 0 ? "Even" : "Odd";
    }
```

If we run the test in the mathService again, we see that the tests do not pass.
Update the mathService, so the tests pass again.

Now add the following method to the controller on the producer side:

```
    @GetMapping("/double")
    public String add(@RequestParam("number") Integer number) {
        return number+number+"";
    }
```

Write a new contract for this.
Also modify mathService so that it uses this method.
Make both the producer side (evenoddService) as the consumer side (mathService) work correctly.

Now add a new Controller in the mathService:

```
@RestController
public class CalculatorController {

      @GetMapping("/add")
    public String add(@RequestParam("value1") Integer value1,
                      @RequestParam("value2") Integer value2) {
        return value1+value2+"";
    }
}
```

Modify the baseTestClass as follows:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@DirtiesContext
@AutoConfigureMessageVerifier
public class BaseTestClass {

    @Autowired
    private EvenOddController evenOddController;

    @Autowired
    private CalculatorController calculatorController;

    @BeforeEach
    public void setup() {
        StandaloneMockMvcBuilder standaloneMockMvcBuilder =

MockMvcBuilders.standaloneSetup(evenOddController,calculatorController);
        RestAssuredMockMvc.standaloneSetup(standaloneMockMvcBuilder);
    }
}
```

Write a new contract for this.
Also modify mathService so that it uses this method.
Make both the producer side (evenoddService) as the consumer side (mathService) work correctly.

Part 5

Suppose we have a microservice architecture with many different services. We also have an app that gets data from these different services. The app runs on both Android and IOS. The problem we face is that when we start this app, the first page on the app needs to show data that comes from 15 different microservices. The time to retrieve all this data from all 15 microservices takes 10 seconds, which is too slow and not acceptable.

The app should get all the required data as fast as possible. It should not take more than 2.5 seconds to retrieve all necessary data.

How can we solve this problem?

What to hand in?

1. A zip file containing all services for part 1
2. A zip file containing all services for part 2
3. A zip file containing all services for part 3
4. A zip file containing all services for part 4
5. A PDF for part 5
6. Write a readme.txt file with the following statement and sign with your name:

   I hereby declare that this submission is my own original work and to the best of my knowledge it contains no materials previously published or written by another person. I am aware that submitting solutions that are not my own work will result in an NC of the course.

   [your name as signature]