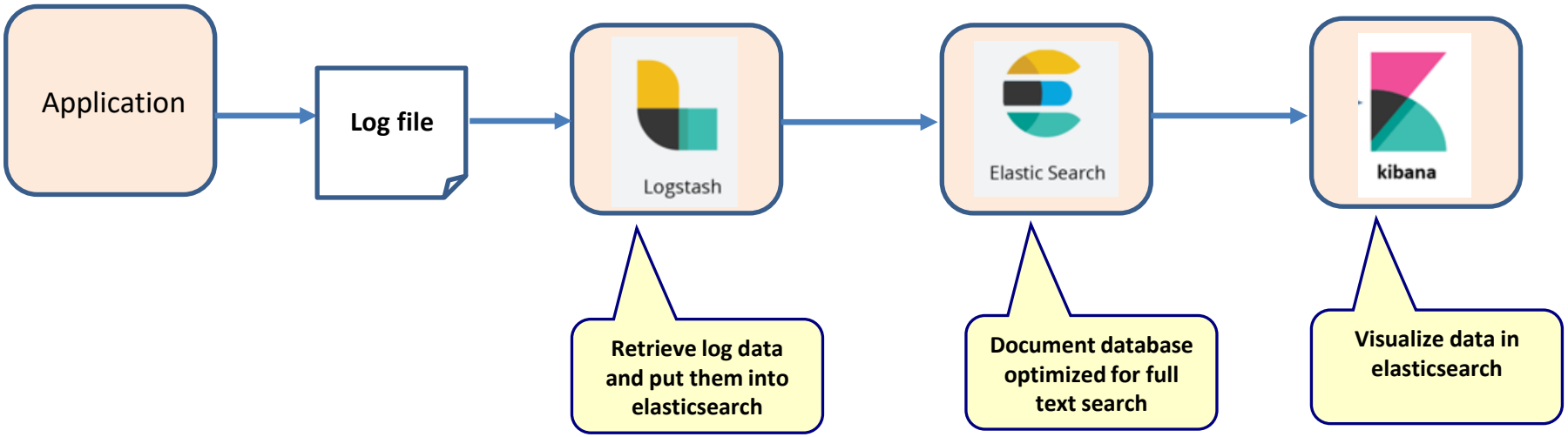
CS544

LESSON 13
MONITORING

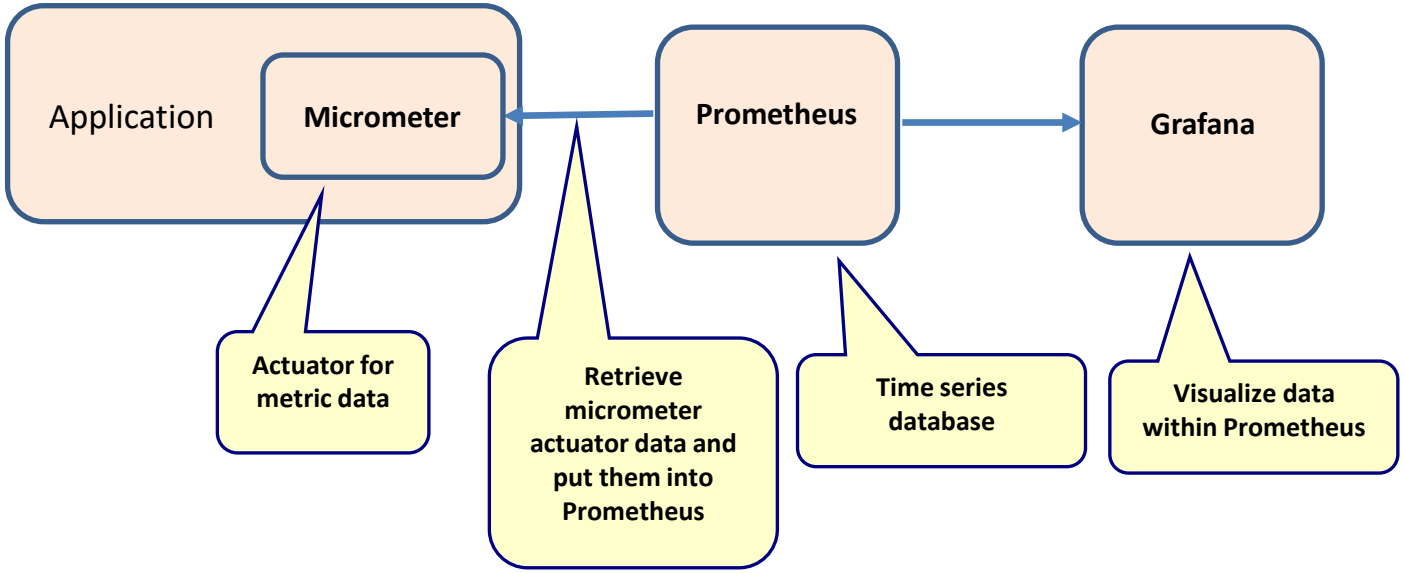
APPLICATION MONITORING

Approach 1: ELK stack



- Good for application specific log data

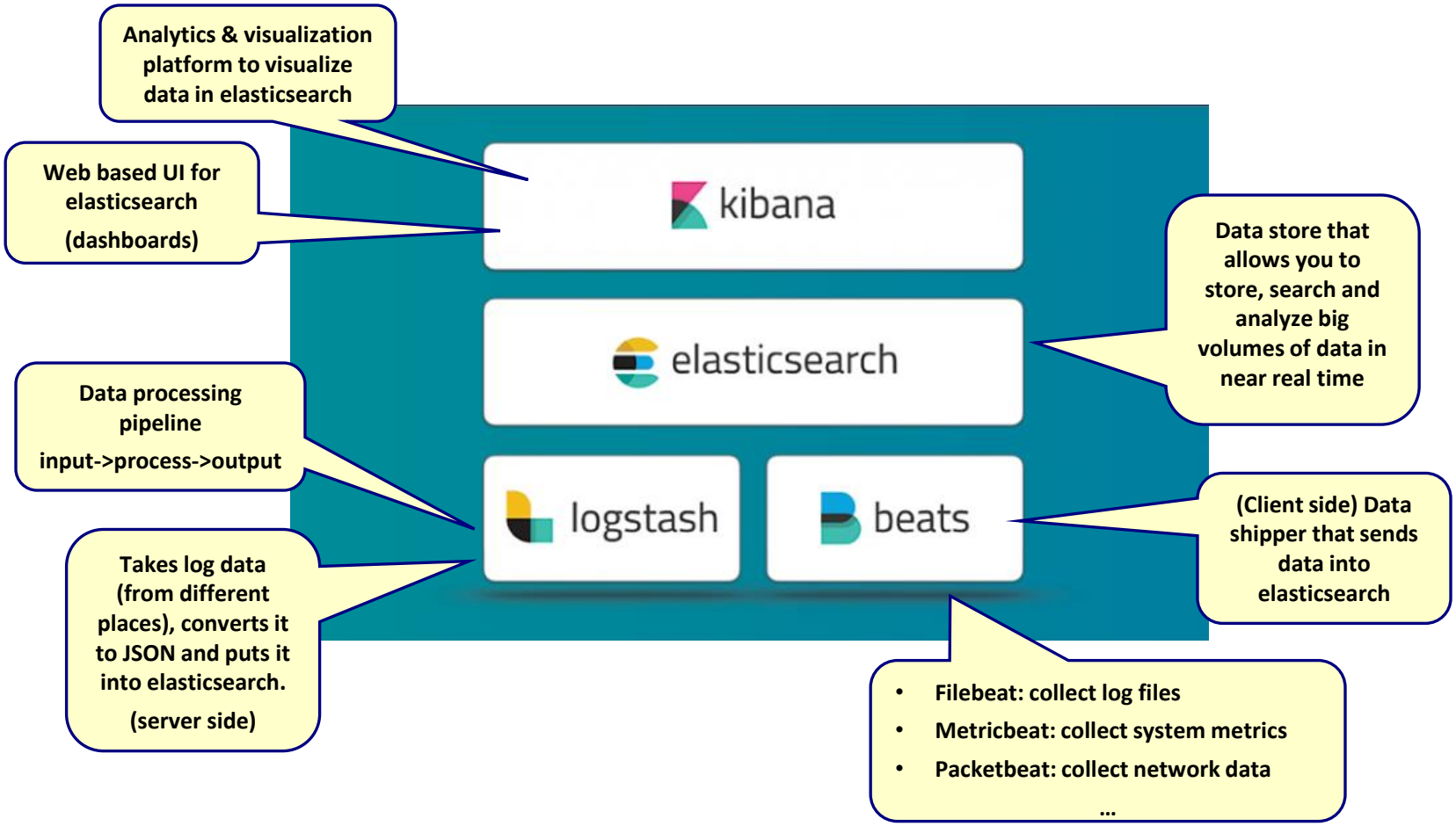
Approach 2: Prometheus/Grafana



- Good for metric data
  - Memory usage
  - CPU usage
  - JVM specific data

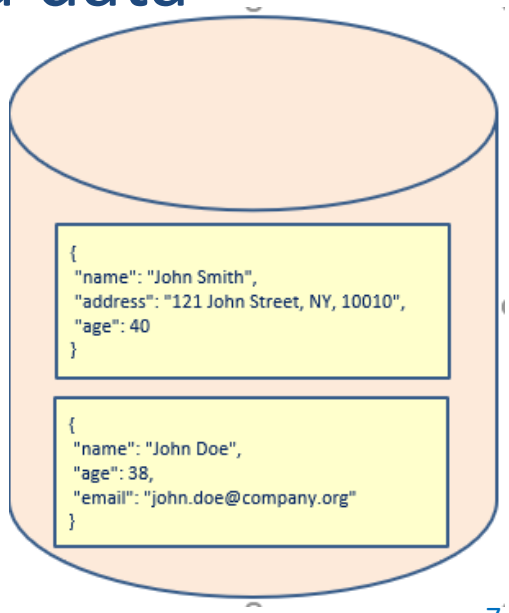
THE ELASTIC STACK

Elastic stack components



What is Elasicsearch?

- Database
  - Data is stored as documents
  - Data is structured in JSON format
- Full text search engine
- Analytics platform for structured data



Inverted index

1. **Analyze the document**
2. **Retrieve the individual terms from the document and put them in the inverted index**

Document 1

The bright blue butterfly hangs on the breeze.

Document 2

It's best to forget the great sky and to retire from every wind.

Document 3

Under blue sky, in bright sunlight, one need not search around.

Stopword list

a
and
around
every
for
from
in
is
it
not
on
one
the
to
under

Inverted index

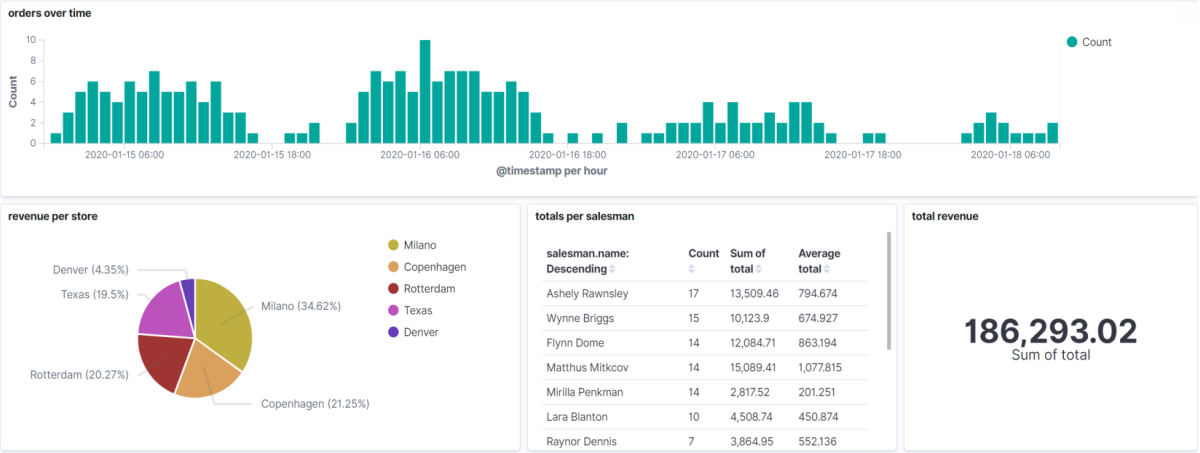
ID	Term	Document
1	best	2
2	blue	1, 3
3	bright	1, 3
4	butterfly	1
5	breeze	1
6	forget	2
7	great	2
8	hangs	1
9	need	3
10	retire	2
11	search	3
12	sky	2, 3
13	wind	2

Queries are executed on the inverted index

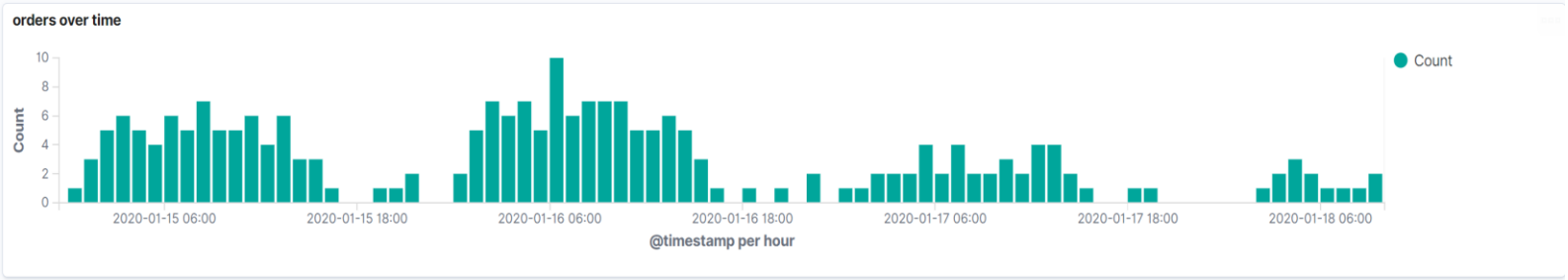
KIBANA

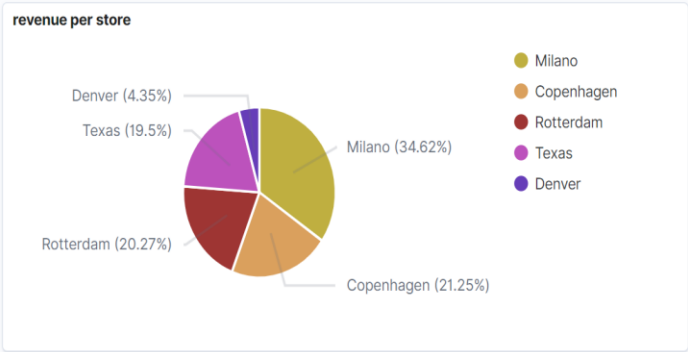
Kibana

- Web UI on top of elasticsearch
- Has its own Kibana query language (KQL)
- Objects (Queries, visualizations, dashboards, etc.) are saved in elasticsearch



Dashboard





totals per salesman

salesman.name: Descending	Count	Sum of total	Average total
Ashely Rawnsley	17	13,509.46	794.674
Wynne Briggs	15	10,123.9	674.927
Flynn Dome	14	12,084.71	863.194
Matthus Mitkcov	14	15,089.41	1,077.815
Mirilla Penkman	14	2,817.52	201.251
Lara Blanton	10	4,508.74	450.874
Raynor Dennis	7	3,864.95	552.136

total revenue

186,293.02

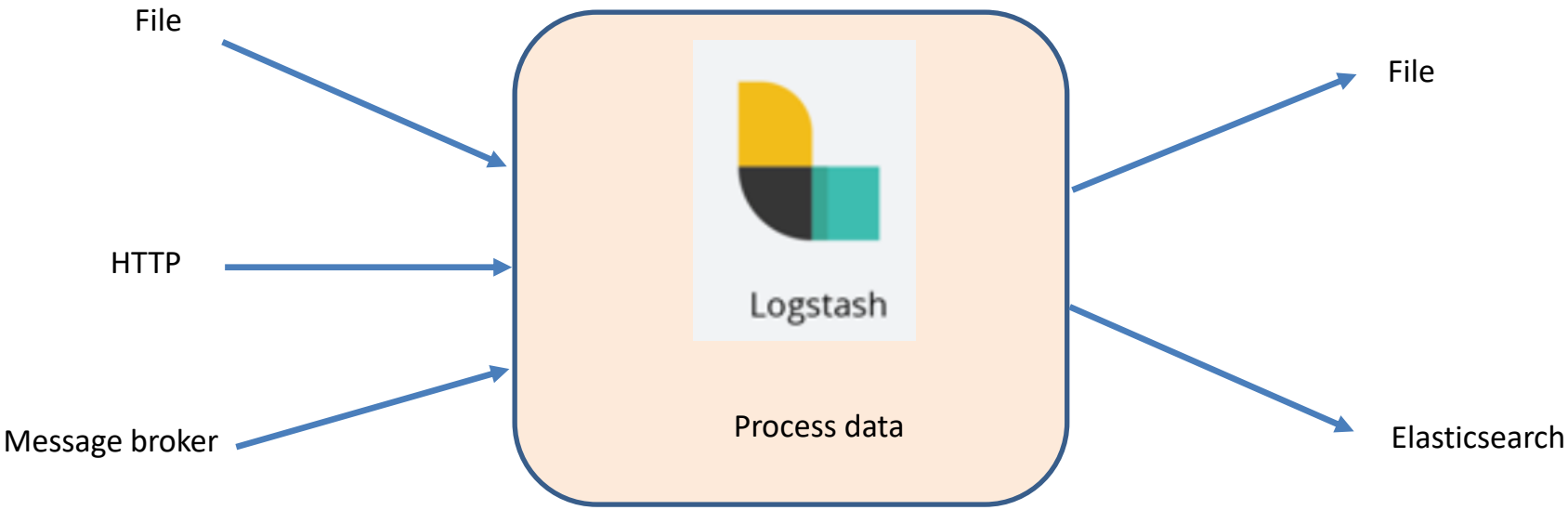
Sum of total

Contains visualizations

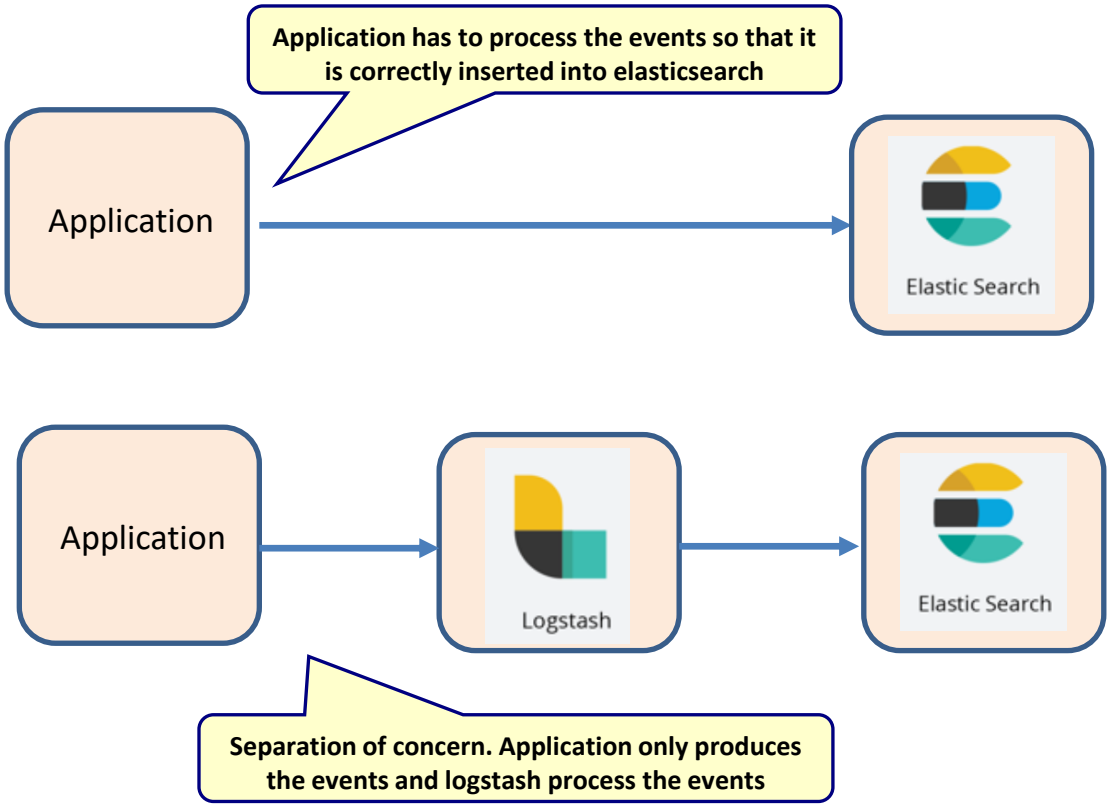
LOGSTASH

Logstash

- Event processing engine



Why logstash in ELK?



Logstash configuration

input.txt

```
Hello world
```

pipeline.conf

```
input {
  file {
    path => "C:/elasticsearchtraining/temp/input.txt"
    start_position => "beginning"
  }
}

output {
  stdout {
    codec => rubydebug
  }
  file {
    path => "C:/elasticsearchtraining/temp/output.txt"
  }
}
```

output.txt


```
{
"host":"DESKTOP-BVHRK6K",
"@version":"1",
"path":"C:/elasticsearchtraining/temp/input.txt",
"message":"Hello world\r",
"@timestamp":"2021-01-16T13:52:32.726Z"
}
```


Anytime this file changes, read from this file

Write the output to the console

Write the output to the specified file

Logstash configuration

input.txt

```
Hi there
```

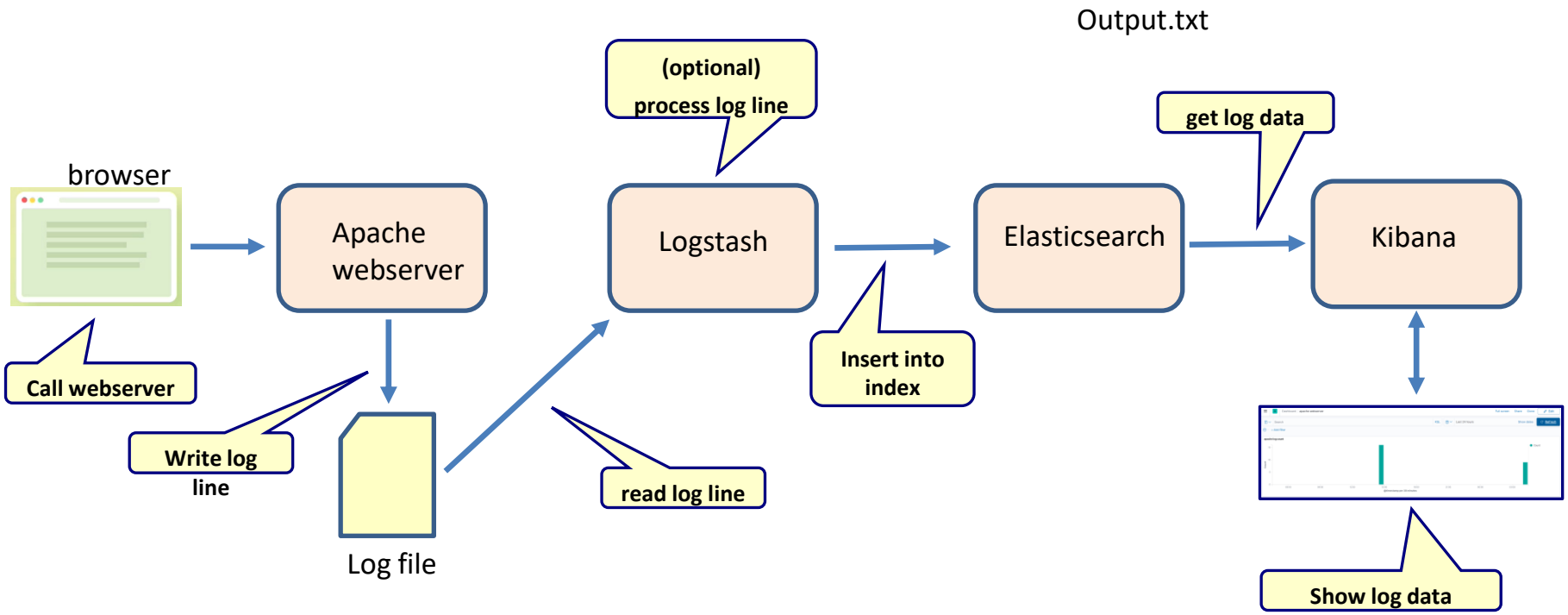
pipeline.conf

```
input {
  file {
    path => "C:/elasticsearchtraining/temp/input.txt"
          start_position => "beginning"
  }
}

filter {
    mutate {
      uppercase => ["message"]
    }
}

output {
  stdout {
    codec => rubydebug
  }
  file {
    path => "C:/elasticsearchtraining/temp/output.txt"
  }
}
```

output.txt


```
{
"path":"C:/elasticsearchtraining/temp/input.txt",
"message":"HI THERE\r",
"host":"DESKTOP-BVHRK6K",
"@version":"1",
"@timestamp":"2021-01-16T14:17:10.537Z"
}
```

logstash example



MONITOR ACTUATOR DATA

Micrometer

- Captures metric data and expose this data via an actuator endpoint

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

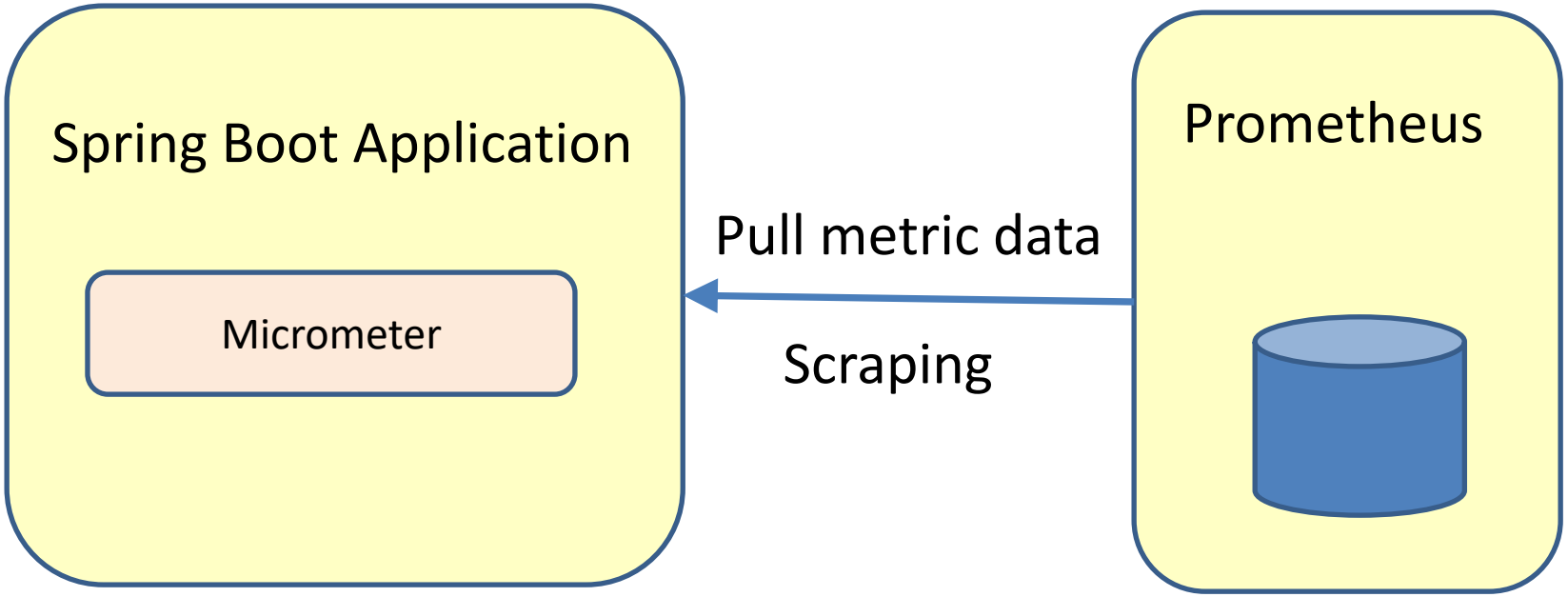
Actuator/prometheus

localhost:8080/actuator/prometheus

```
# HELP kafka_consumer_outgoing_byte_rate The number of outgoing bytes sent to all
servers per second
# TYPE kafka_consumer_outgoing_byte_rate gauge
kafka_consumer_outgoing_byte_rate{client_id="consumer-gid-
1",kafka_version="3.0.1",spring_id="kafkaConsumerFactory.consumer-gid-1",}
161.47368421052633
# HELP process_cpu_usage The "recent cpu usage" for the Java Virtual Machine
process
# TYPE process_cpu_usage gauge
process_cpu_usage 0.1281166160486457 7
# HELP logback_events_total Number of error level events that made it to the logs
# TYPE logback_events_total counter
logback_events_total{level="warn",} 2.0
logback_events_total{level="debug",} 0.0
logback_events_total{level="error",} 0.0
logback_events_total{level="trace",} 0.0
logback_events_total{level="info",} 32.0
# HELP kafka_consumer_network_io_total The total number of network operations
(reads or writes) on all connections
# TYPE kafka_consumer_network_io_total counter
kafka_consumer_network_io_total{client_id="consumer-gid-
```

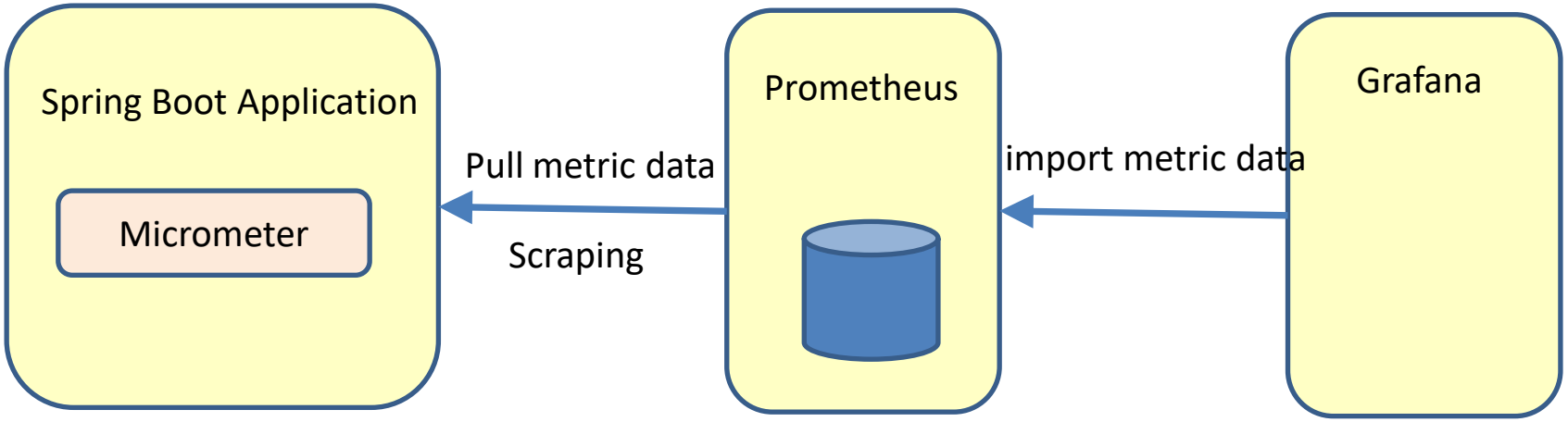
Prometheus

- Time series database
- Stores metric and performance data

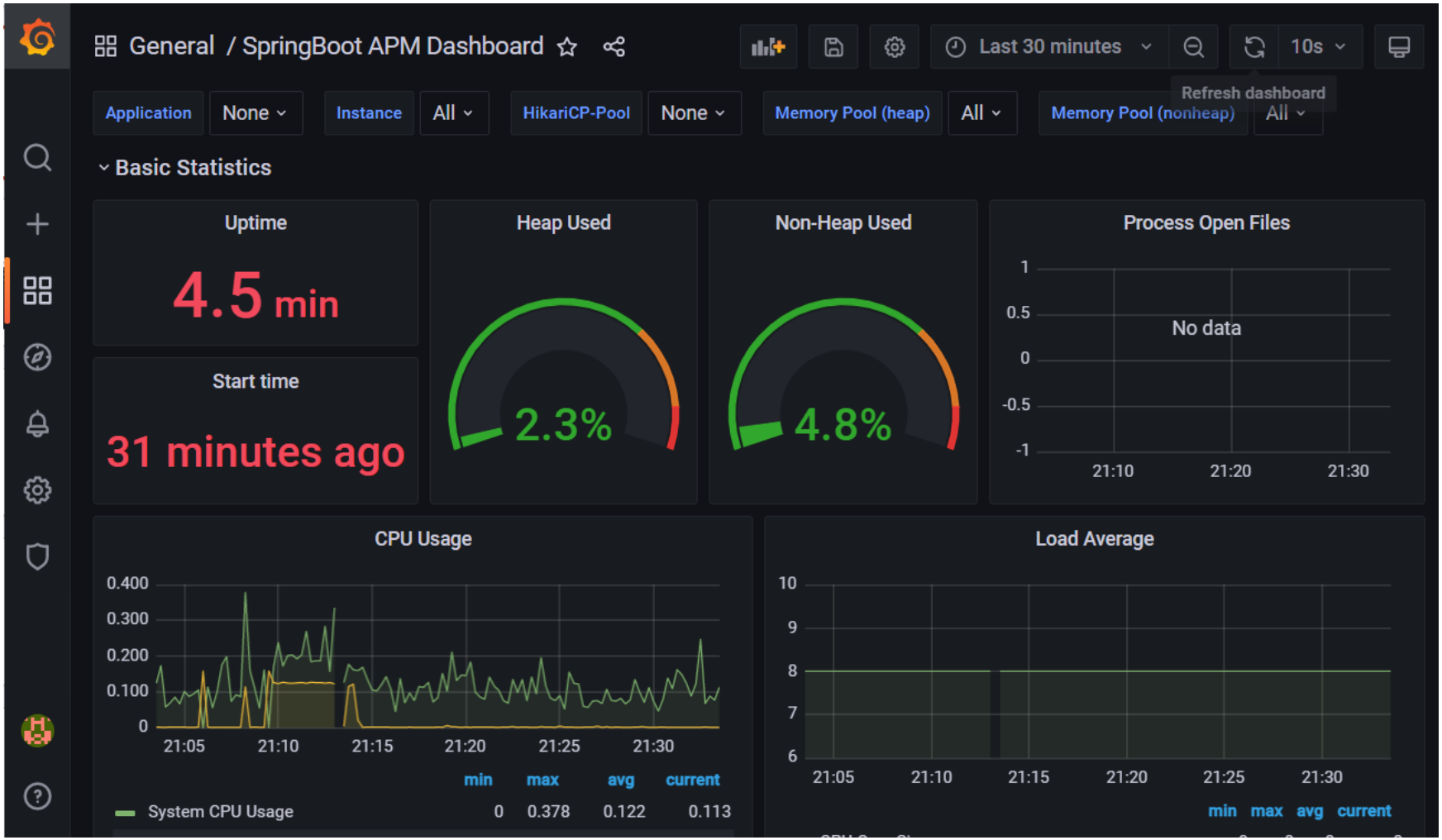


Grafana

- Dashboard to visualize metric data

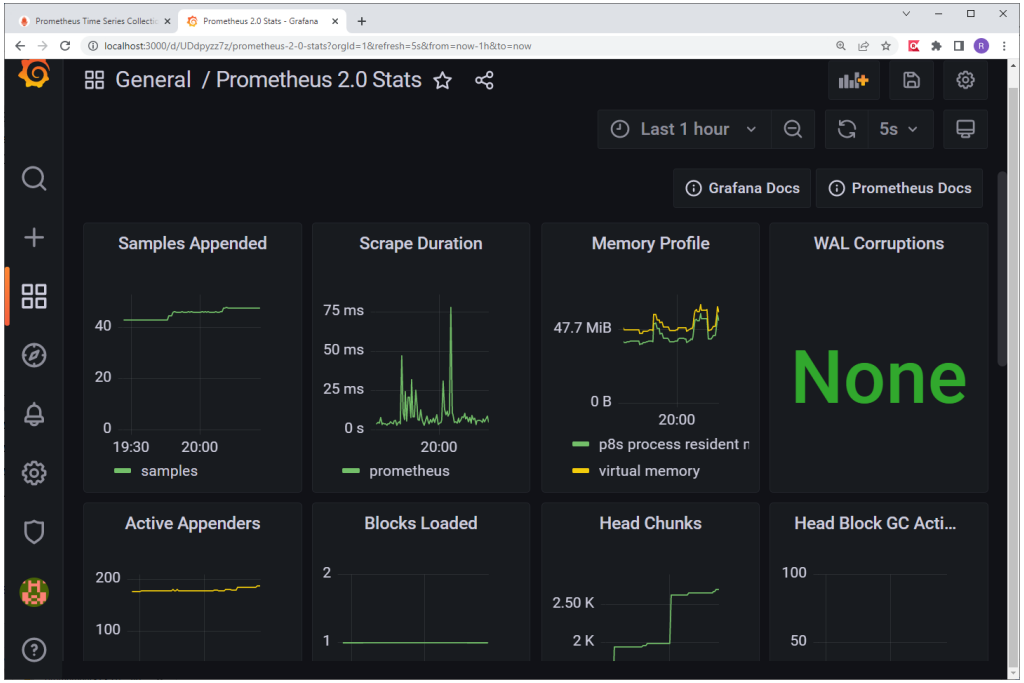


Grafana dashboard



Grafana

- Make your own dashboards
- Alerts
- Refresh interval
- Timespan



UNIT TESTING WITH JUNIT

What is unit testing?

- A unit test is a test that test one single class.
  - A test case test one single method
  - A test class test one single class
  - A test suite is a collection of test classes
- Unit tests make use of a testing framework

- A unit test
  1. Create an object
  2. Call a method
  3. Check if the result is correct

Example of unit testing

```
package count;

public class Counter {
    private int counterValue=0;

    public int increment(){
        return ++counterValue;
    }
    public int decrement(){
        return --counterValue;
    }
    public int getCounterValue() {
        return counterValue;
    }
 }
```

Example of unit testing

```java
public class Counter {
    private int counterValue=0;

    public int increment(){
        return ++counterValue;
    }
    public int decrement(){
        return --counterValue;
    }
    public int getCounterValue() {
        return counterValue;
    }
}
```

```java
public class CounterTest {
    private Counter counter;

    @BeforeEach
    public void setUp() throws Exception {
      counter = new Counter();
    }
    @Test
    public void testIncrement(){
        assertEquals("Counter.increment does not work correctly",1,counter.increment());
        assertEquals("Counter.increment does not work correctly",2,counter.increment());
    }
    @Test
    public void testDecrement(){
        assertEquals("Counter.decrement does not work correctly",-1,counter.decrement());
        assertEquals("Counter.decrement does not work correctly",-2,counter.decrement());
    }
}
```

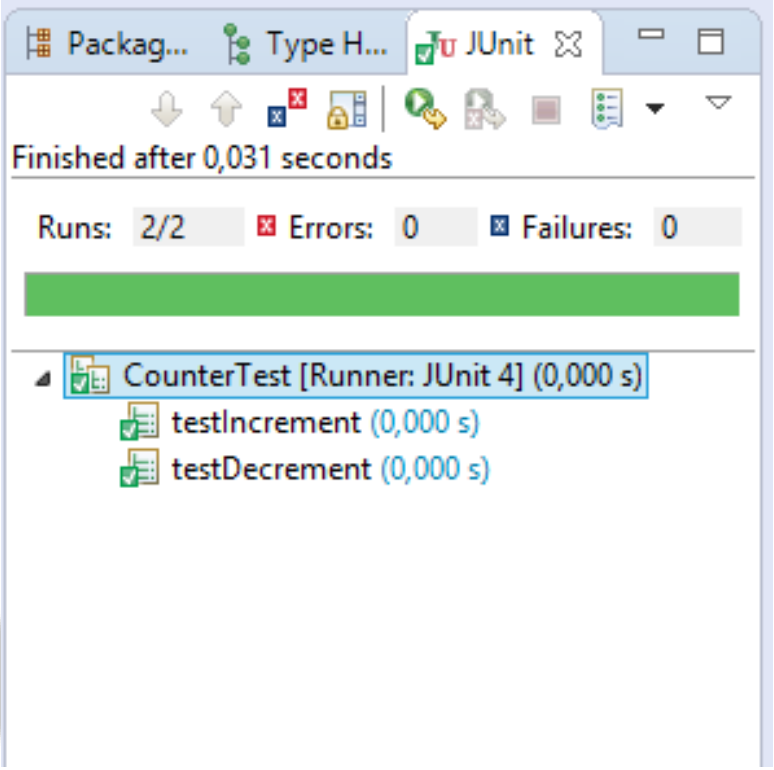
Initialization

Test method

Test method

Running the test

```java
package count;

public class Counter {
    private int counterValue=0;

    public int increment(){
        return ++counterValue;
    }
    public int decrement(){
        return --counterValue;
    }
    public int getCounterValue() {
        return counterValue;
    }
}
```



Running the test

```
package count;

public class Counter {
    private int counterValue=0;

    public int increment(){
        return ++counterValue;
    }
    public int decrement(){
        return counterValue;
    }
    public int getCounterValue() {
        return counterValue;
    }
}
```

Package Explorer | Type Hierarchy | JUnit

Finished after 0,032 seconds

Runs: 2/2 Errors: 0 Failures: 1

CounterTest [Runner: JUnit 4] (0,000 s)
- testIncrement (0,000 s)
- testDecrement (0,000 s)

Failure Trace

```
java.lang.AssertionError: Counter.decrement does not work correctly expected:<-1> but was:<0>
at CounterTest.testDecrement(CounterTest.java:21)
```

JUnit test case

```
public class Calculator
{
    public double add( double number1, double number2 )
    {
        return number1 + number2;
    }
}
```

```
public class CalculatorTest
{
    @Test
    public void add()
    {
        Calculator calculator = new Calculator();
        double result = calculator.add( 10, 50 );
        assertEquals( 60, result, 0 );
    }
}
```

expected

Value to assert

delta

Junit assert methods

- static void assertTrue(boolean ***test***)
- static void assertTrue(String **message**, boolean ***test***)

- static void assertFalse(boolean ***test***)
- static void assertFalse(String **message**, boolean ***test***)

- assertEquals(Object ***expected***, Object ***actual***)
- assertEquals(String **message**, ***expected***, ***actual***)

- assertSame (Object ***expected***, Object ***actual***)
- assertSame(String **message**, Object ***expected***, Object ***actual***)

- assertNotSame(Object ***expected***, Object ***actual***)
- assertNotSame(String **message**, Object ***expected***, Object ***actual***)

- assertNull(Object ***object***)
- assertNull(String **message**, Object ***object***)

- assertNotNull(Object ***object***)
- assertNotNull(String **message**, Object ***object***)

- fail()
- fail(String **message**)

http://junit.org/apidocs/org/junit/Assert.html

@Before and @After

```
public class CounterTest {
    private Counter counter;

    @BeforeEach
    public void setUp() throws Exception {
        counter = new Counter();
    }

    @AfterEach
    public void tearDown() throws Exception {
        counter=null;
    }
    @Test
    public void testConstructor(){
        assertEquals("Counter constructor does not set counter to
                    0",0,counter.getCounterValue());
    }
}
```

This method is called before every testmethod

This method is called after every testmethod

@BeforeClass and @AfterClass

```
public class CounterTest {
    private static Counter counter;

    @BeforeClass
    public static void setUpOnce() throws Exception {
        counter = new Counter();
    }

    @AfterClass
    public static void tearDownOnce() throws Exception {
        counter=null;
    }
    @Test
    public void testConstructor(){
        assertEquals("Counter constructor does not set counter to
                      0",0,counter.getCounterValue());
    }
}
```

This method is called once, before the testmethods are called

This method is called once, after the testmethods are called

Test suite

```
@RunWith(value=Suite.class)
@SuiteClasses(value={CalculatorTest.class, ParameterizedTest.class})
public class CalculatorTestSuite {

}
```

Suite of 2 test classes

- You can also have a suite of suites
- Organize your tests

JUnit example: Calculator

```
public class Calculator {
    private double value;

    public Calculator() {
      value =0.0;
    }
    public void add(double number) {
      value = value + number;
    }
    public void subtract (double number) {
      value = value - number;
    }
    public void multiply(double number) {
      value = value * number;
    }
    public void divide (double number) throws DivideByZeroException{
      if (number == 0){
        throw new DivideByZeroException();
      }
      value = value / number;
    }
    public double getValue() {
      return value;
    }
}
```

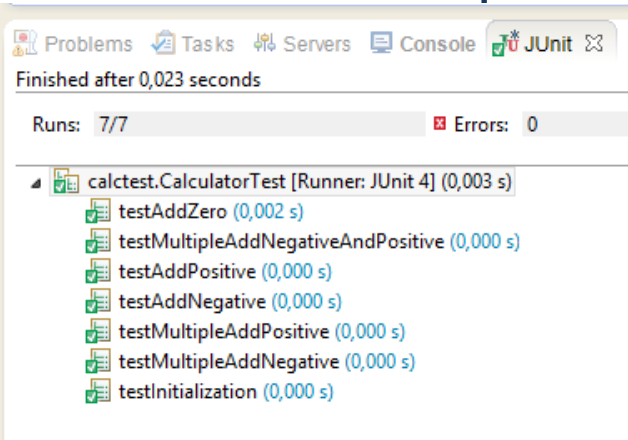
JUnit example: CalculatorTest

```
import calculation.Calculator;

public class CalculatorTest {

  private Calculator calculator;

  @BeforeEach
  public void setup(){
    calculator = new Calculator();
  }

  @Test
  public void testInitialization() {
    assertEquals(0.0, calculator.getValue(),0.0000001);
  }

  @Test
  public void testAddZero() {
    calculator.add(0.0);
    assertEquals(0.0, calculator.getValue(),0.0000001);
  }
```

JUnit example: CalculatorTest

```
@Test
public void testAddPositive() {
  calculator.add(23.255);
  assertEquals(23.255, calculator.getValue(),0.0000001);
}
@Test
public void testAddNegative() {
  calculator.add(-23.255);
  assertEquals(-23.255, calculator.getValue(),0.0000001);
}
@Test
public void testMultipleAddPositive() {
  calculator.add(23.255);
  calculator.add(10.255);
  assertEquals(33.510, calculator.getValue(),0.0000001);
}
@Test
public void testMultipleAddNegative() {
  calculator.add(-23.255);
  calculator.add(-10.255);
  assertEquals(-33.510, calculator.getValue(),0.0000001);
}
@Test
public void testMultipleAddNegativeAndPositive() {
  calculator.add(-23.255);
  calculator.add(10.250);
  assertEquals(-13.005, calculator.getValue(),0.0000001);
}
```

Only test methods for add()



HAMCREST MATCHERS

Traditional asserts

- Parameter order is counter-intuitive
- Assert statements don't read well

assertEquals(***expected***, ***actual***)

```
import static org.junit.Assert.*;

@Test
public void AssertEqualToRed(){
    String color = "red";
    assertEquals("red", color);
}
```

assertThat with hamcrest matchers

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.jupiter.api.Before;
import org.junit.jupiter.api.Test;

public class CalculatorHamcrestTest{
Calculator calculator=null;

    @BeforeEach
    public void createAcalculator(){
      calculator = new Calculator();
    }

    @Test
    public void add(){
        assertThat( calculator.add( 10, 50), equalTo (60.0));
    }

    @Test
    public void divide(){
        assertThat(calculator.divide( 10, 2 ), equalTo (5.0));
    }
}
```

Static import of matchers

matcher

assertThat

actual

expected

assert vs assertThat

```
@Test
public void AssertEqualToRed(){
    String color = "red";
    assertEquals("red", color);
}
```

assert

```
@Test
public void hamcrestAssertEqualToRed(){
    String color = "red";
    assertThat(color, equalTo("red"));
}
```

assertThat

assertThat equality tests

```
String color = "red";
assertThat(color, is("red"));
```

assertThat … is

```
String color = "red";
assertThat(color, equalTo("red"));
```

assertThat … equalTo

```
String color = "red";
assertThat(color, not("blue"));
```

assertThat … not

```
String color = "red";
assertThat(color, isOneOf("blue","red"));
```

assertThat … isOneOf

```
List myList = new ArrayList();
assertThat(myList, is(Collection.class));
```

assertThat … is a class

assertThat testing for null values

```
String color = "red";
assertThat(color, is(notNullValue()));
assertNotNull(color);
```

notNullValue

```
String color = null;
assertThat(color, is(nullValue()));
assertNull(color);
```

nullValue

assertThat testing with collections

```
List<String> colors = new ArrayList<String>();
colors.add("red");
colors.add("green");
colors.add("blue");
assertThat(colors, hasItem("blue"));
```

hasItem

```
assertThat(colors, hasItems("red","blue"));
```

hasItems

```
String[] colors = new String[] {"red","green","blue"};
assertThat(colors, hasItemInArray("blue"));
```

hasItemInArray

```
assertThat("red", isIn(colors));
```

isIn

```
List<Integer> ages = new ArrayList<Integer>();
ages.add(20);
ages.add(30);
ages.add(40);
assertThat(ages, not(hasItem(lessThan(18))));
```

Combined matchers

Hamcrest matchers

- Core
  - anything - always matches, useful if you don't care what the object under test is
  - describedAs - decorator to adding custom failure description
  - is - decorator to improve readability
- Logical
  - allOf - matches if all matchers match, short circuits (like Java &&)
  - anyOf - matches if any matchers match, short circuits (like Java ||)
  - not - matches if the wrapped matcher doesn't match and vice versa
- Object
  - equalTo - test object equality using Object.equals
  - hasToString - test Object.toString
  - instanceOf, isCompatibleType - test type
  - notNullValue, nullValue - test for null
  - sameInstance - test object identity
- Beans
  - hasProperty - test JavaBeans properties
- Collections
  - array - test an array's elements against an array of matchers
  - hasEntry, hasKey, hasValue - test a map contains an entry, key or value
  - hasItem, hasItems - test a collection contains elements
  - hasItemInArray - test an array contains an element
- Number
  - closeTo - test floating point values are close to a given value
  - greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo - test ordering
- Text
  - equalToIgnoringCase - test string equality ignoring case
  - equalToIgnoringWhiteSpace - test string equality ignoring differences in runs of whitespace
  - containsString, endsWith, startsWith - test string matching

UNIT TESTING BEST PRACTICES

Good unit tests: FIRST

- Fast
- Isolated
- Repeatable
- Self-validating
- Timely

Fast

- It should be comfortable to run all unit tests often
- Isolate slow tests from fast tests
  - Separate unit and integration tests

Isolated

- Only two possible results: PASS or FAIL
- No partially successful tests.
  - If a test can break for more than one reason, consider splitting it into separate tests

- Isolation of tests:
  - Different execution order must yield same results.
  - Test B should not depend on outcome of Test A

Repeatable

- A test should produce the same results each time you run it.
- Watch out for
  - Dates, times
  - Random numbers
  - Data from a datastore
- Use mock objects to give consistent data

Self-validating

- Your tests should be able to run anywhere at any time
- They should not depend on
  - Manual interaction
  - External setup

Timely

- Do not defer writing unit tests
  - For every method you write, write the corresponding unit tests at the same time
- Use test rules in your project
  - Review process
  - Test coverage tools

Unit test best practices

- Write tests for every found bug
- Fix failing tests immediately
- Make unit tests simple to run
  - Test suites can be run by a single command or a one button click.
- An incomplete set of unit tests is better than no unit tests at all.
- Don’t repeat production logic
- Reuse test code (setup, manipulate, assert)
- Don’t run a test from another test

Single Responsibility

- One test should be responsible for one scenario only.

- Test behavior, not methods:
  - One method, multiple behaviors → Multiple tests
  - One behavior, multiple methods → One test

Single Responsibility

```java
@Test
public void testMethod() {
    assertTrue(behaviour1);
    assertTrue(behaviour2);
    assertTrue(behaviour3);
}
```

```java
@Test
public void testMethodCheckBehaviour1() {
    assertTrue(behaviour1);
}

@Test
public void testMethodCheckBehaviour2() {
 assertTrue(behaviour2);
}

@Test
public void testMethodCheckBehaviour3() {
  assertTrue(behaviour3);
}
```

Self Descriptive

- Unit test must be easy to read and understand
  - Variable Names
  - Method Names
  - Class Names
  - No conditional logic
  - No loops

Self descriptive

- Name tests to represent PASS conditions:
  - canMakeReservation()
  - totalBillEqualsSumOfMenuItemPrices()

No conditional logic

- Test should have no uncertainty:
  - All inputs should be known
  - Method behavior should be predictable
  - Expected output should be strictly defined
  - Split in to two tests rather than using “If” or “Case”

- Tests should not contain conditional logic.
  - If test logic has to be repeated, it probably means the test is too complicated.

No conditional logic

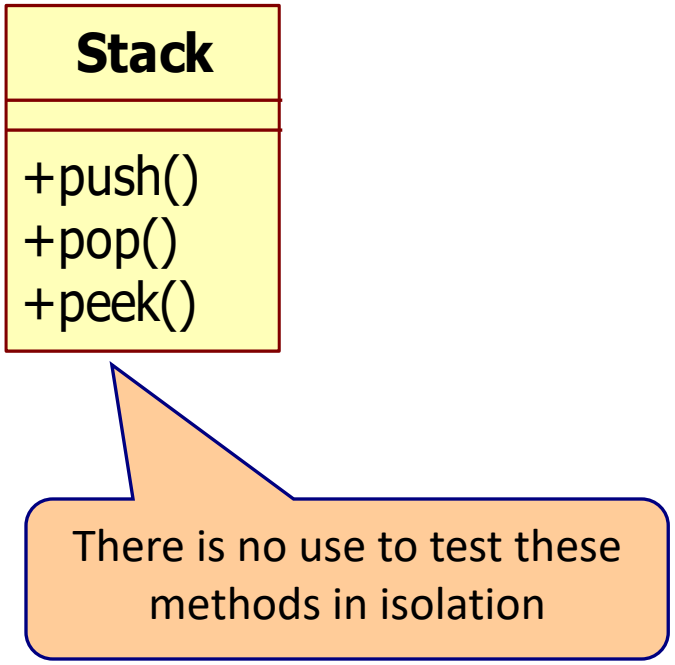
```java
@Test
public void testMethod() {
  if (before)
    assertTrue(behaviour1);
  else if (after)
    assertTrue(behaviour2);
  else
    assertTrue(behaviour3);
}
```

```java
@Test
public void testBefore() {
  boolean before = true;
  assertTrue(behaviour1);
}

@Test
public void testAfter() {
  boolean after= true;
  assertTrue(behaviour2);
}

@Test
public void testNow() {
   boolean before = false;
   boolean after= false;
   assertTrue(behaviour3);
}
```

Test only the public interface

- Every method has a side effect
  - Test this side effect
  - Test behavior, not methods
- What if this side effect is not visible (private attributes and methods)?
  - Do not sacrifice good design just for testing
  - Test behavior, not state

Test behavior, not methods/state



- Unit tests:
  - Pop of an empty stack should return null
  - Peek of an empty stack should return null
  - Push first x on the stack, then a peek should return x
  - Push first x on the stack, then a pop should remove x from the stack
  - Push first x, then y. A pop should return y and another pop should return x.

Summary

- Fast
- Isolated
- Repeatable
- Self-validating
- Timely
- Single responsibility
- No conditional logic
- Test behavior, not methods
  - Test the public interface

Treat test code as production code

Keep your tests

- Simple
- Short
- Understandable
- Loosely coupled

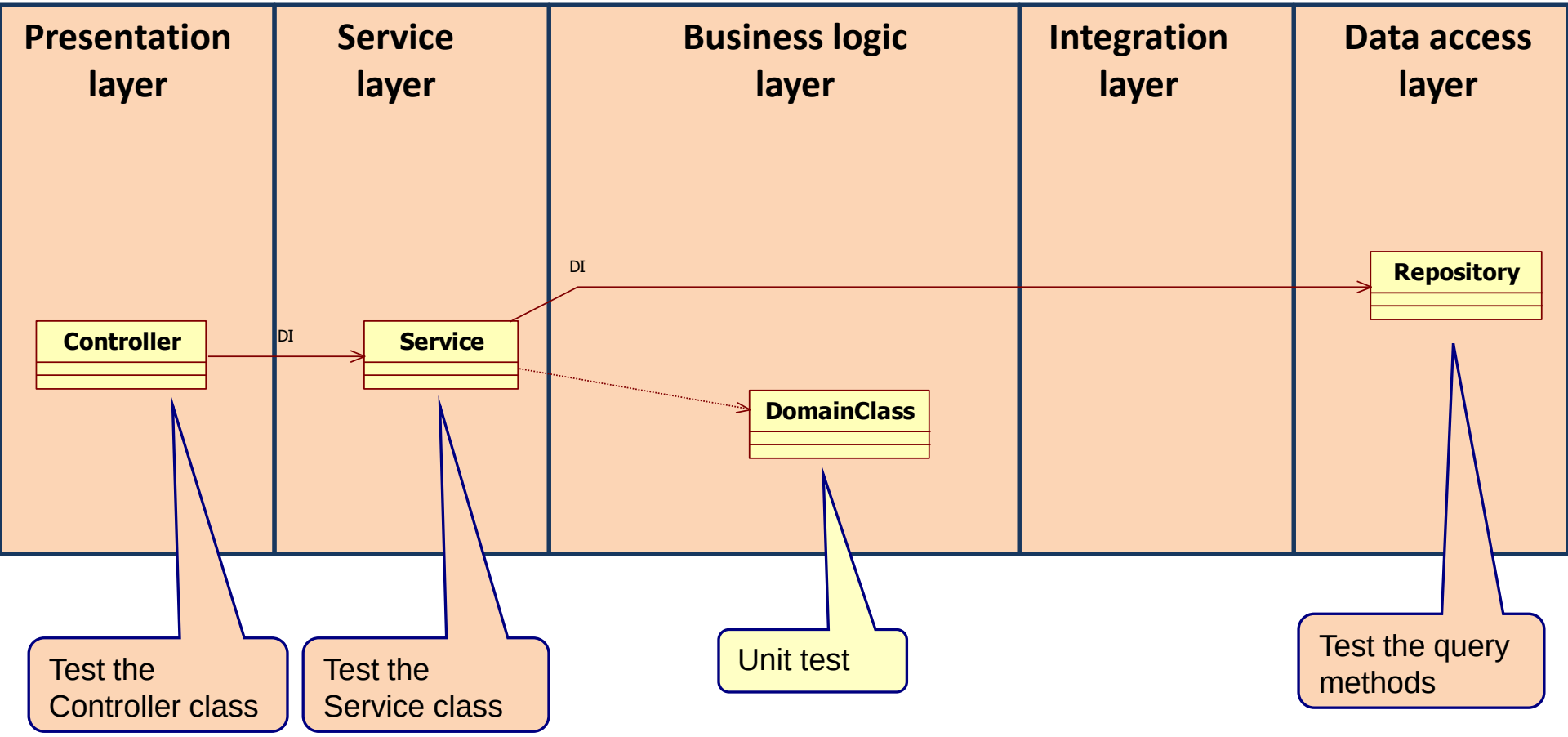
Main point

- The test code and the application code should be loosely coupled.

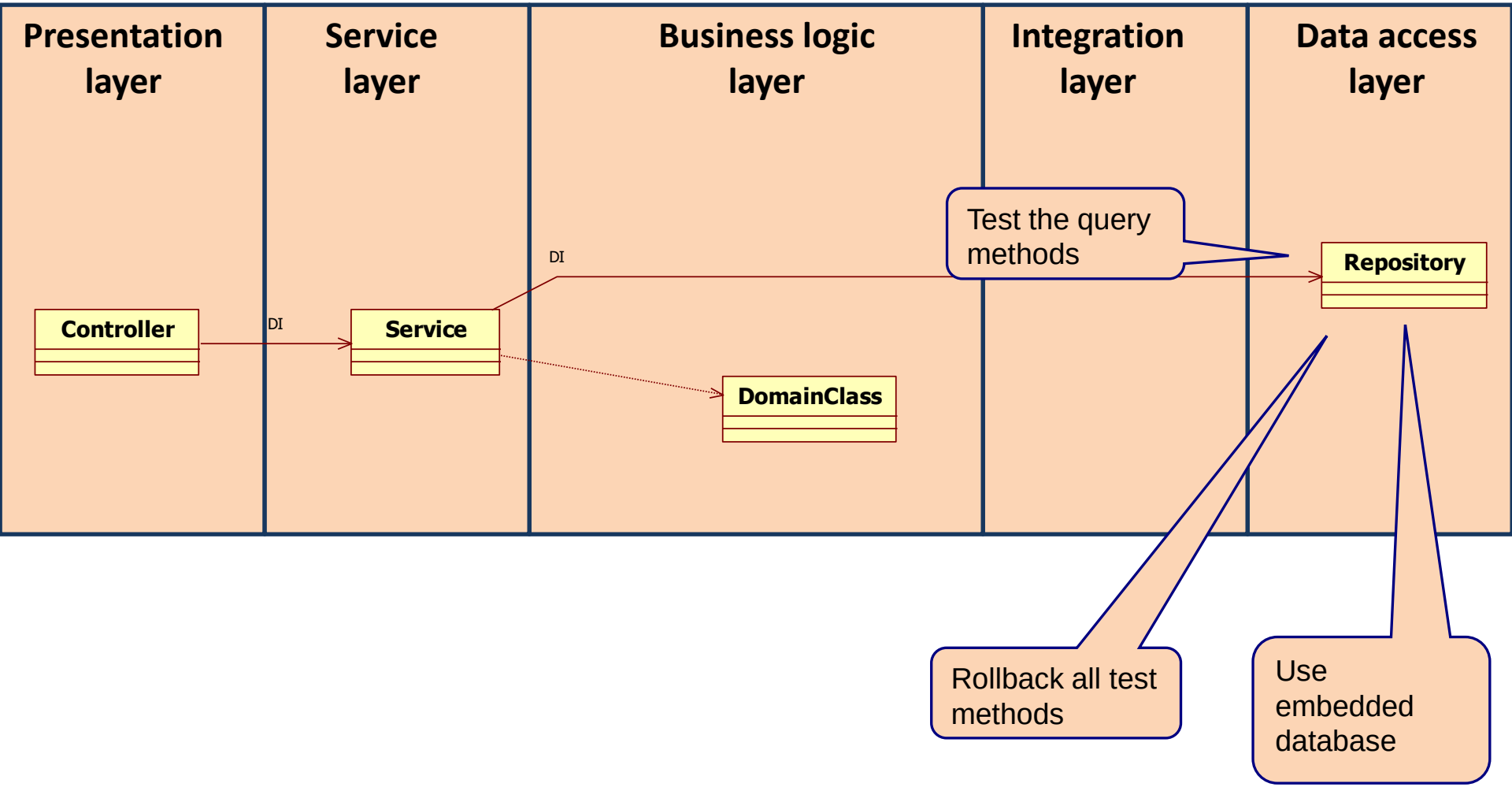
  Science of Consciousness: When you take care of yourself, nature will take care of you. When you go against nature, nature will not support you.

SPRING TESTING

Spring testing



Test the repository



Testing the repository

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

  Customer findByName(String name);
}
```

Auto configure JPA
- Scan entities
- Setup database and datasource
- Create entityManager
- Create repository

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class CustomersRepositoryTests {
    @Autowired
    private TestEntityManager entityManager;
    @Autowired
    private CustomerRepository customerRepository;

    @Test
    public void whenFindByName_thenReturnEmployee() {
        // given
        Customer frank = new Customer(123L,"Frank Brown","fbrown@gmail.com");
        entityManager.persist(frank);
        entityManager.flush();
        // when
        Customer found = customerRepository.findByName(frank.getName());
        // then
        assertThat(found.getName())
          .isEqualTo(frank.getName());
    }
}
```

Data JPA tests are transactional and rolled back at the end of each test

Use the entityManager to persist a Customer

Call the method on the repository

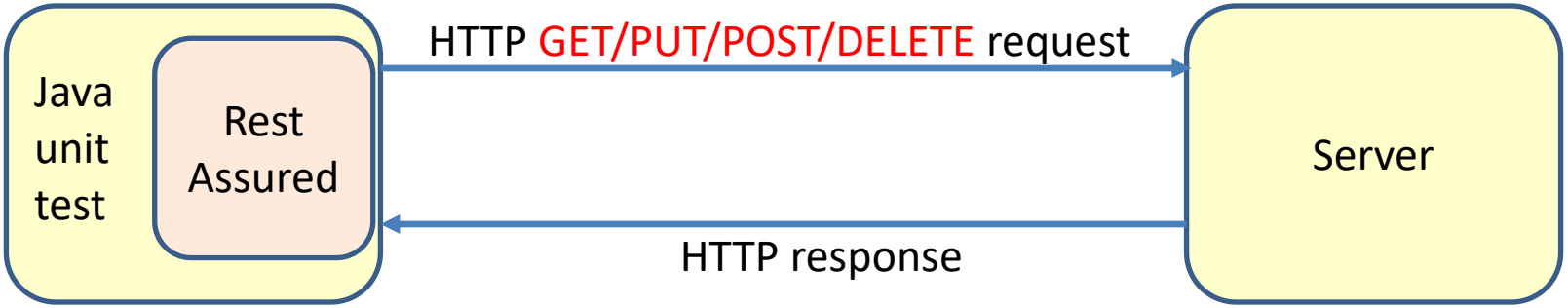
Using an embedded database

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>test</scope>
  <version>1.4.194</version>
</dependency>
```

```
Replacing 'dataSource' DataSource bean with embedded versionStarting embedded database:
url='jdbc:h2:mem:cda533b4-a53f-4fb6-8f00-8a608a533537;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=false',
username='sa'
Hibernate: drop table customer if exists
Hibernate: create table customer (customer_number bigint not null, email varchar(255), name
varchar(255), primary key (customer_number))
Started CustomersRepositoryTests in 3.128 seconds (JVM running for 3.832)
Began transaction (1) for test context
Hibernate: insert into customer (email, name, customer_number) values (?, ?, ?)
Hibernate: select customer0_.customer_number as customer1_0_, customer0_.email as email2_0_,
customer0_.name as name3_0_ from customer customer0_ where customer0_.name=?
Rolled back transaction for test:
Closing JPA EntityManagerFactory for persistence unit 'default'
Hibernate: drop table customer if exists
```

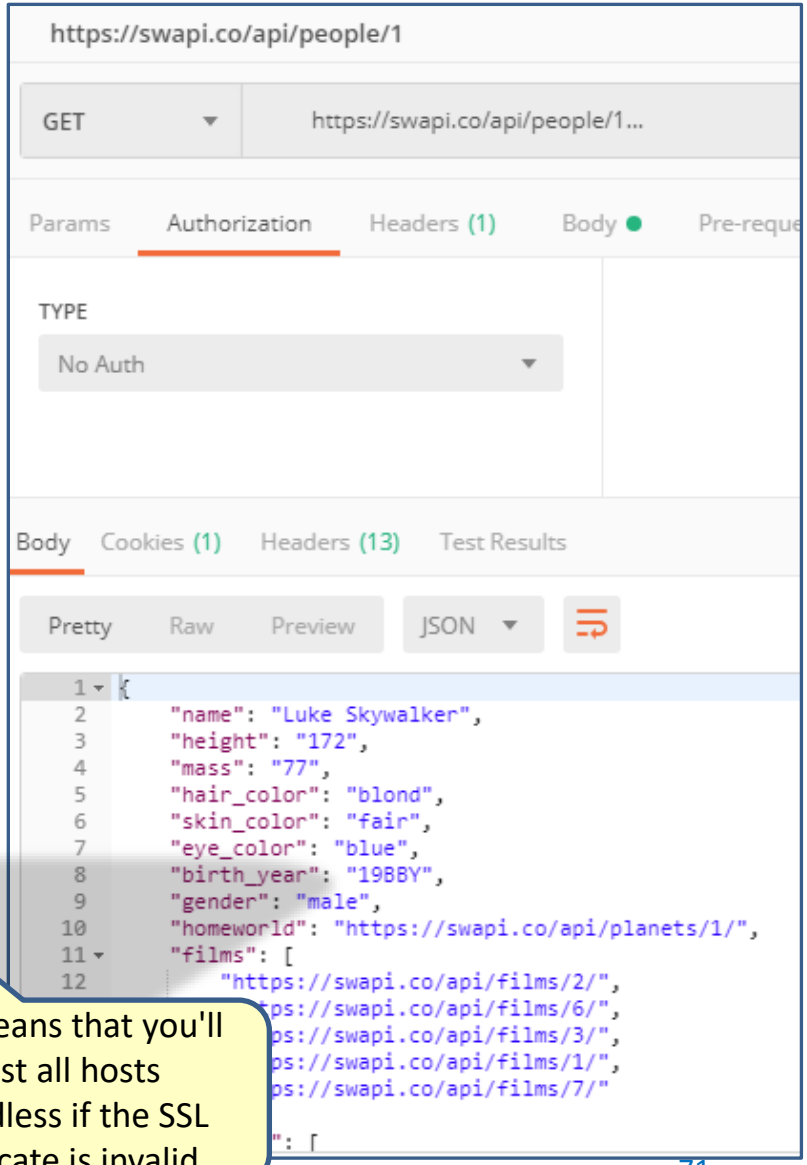
RESTASSURED

REST client


Java
unit
test
Rest
Assured
HTTP GET/PUT/POST/DELETE request
HTTP response
Server

RestAssured example

```java
import org.junit.BeforeClass;
import org.junit.Test;
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.equalTo;

public class RestTest {

  @BeforeClass
  public static void setup() {
        RestAssured.port = Integer.valueOf(8080);
        RestAssured.baseURI = "http://swapi.co";
        RestAssured.basePath = "/api/people/";
  }

  @Test
  public void test() {
    given()
      .relaxedHTTPSValidation("TLSv1.2")
      .when()
      .get("1")
      .then()
      .body("name",equalTo("Luke Skywalker"));
  }
}
```



This means that you'll trust all hosts regardless if the SSL certificate is invalid.

statusCode

```
@Test
public void testStatusLuke() {
  given()
   .relaxedHTTPSValidation("TLSv1.2")
   .when()
   .get("1")
   .then()
   .statusCode(200)
   .body("name",equalTo("Luke Skywalker"));
}
```

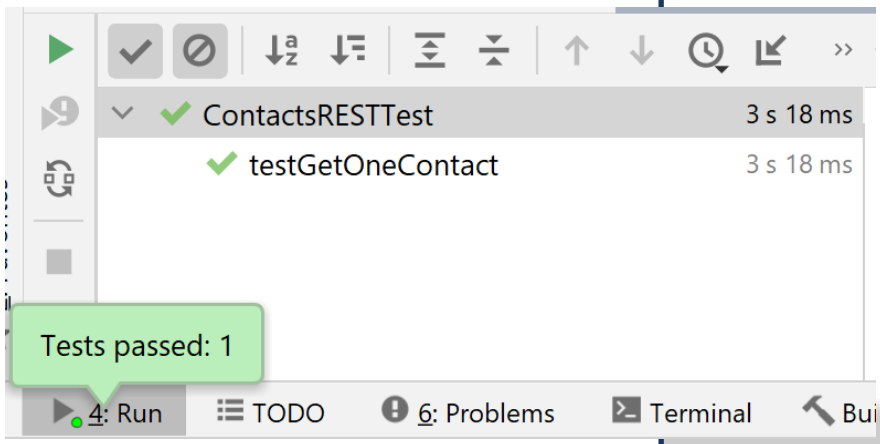
```
@Test
public void testStatusLuke() {
  given()
   .relaxedHTTPSValidation("TLSv1.2")
   .when()
   .get("123")
   .then()
   .statusCode(404);
}
```

contentType

```
@Test
public void test() {
  given().relaxedHTTPSValidation("TLSv1.2")
   .when()
   .get("1")
   .then()
   .contentType(ContentType.JSON)
   .and()
   .body("name",equalTo("Luke Skywalker"));
}
```

GET contact

```
public class ContactsRESTTest {
    @BeforeClass
    public static void setup() {
        RestAssured.port = Integer.valueOf(8080);
        RestAssured.baseURI = "http://localhost";
        RestAssured.basePath = "";
    }
  @Test
  public void testGetOneContact() {
    // add the contact to be fetched
    Contact contact = new Contact("Mary", "Jones", "mjones@acme.com", "2341674376");
    given()
          .contentType("application/json")
          .body(contact)
          .when().post("/contacts").then()
          .statusCode(200);
    // test getting the contact
    given()
          .when()
          .get("contacts/Mary")
          .then()
          .contentType(ContentType.JSON)
          .and()
          .body("firstName",equalTo("Mary"))
          .body("lastName",equalTo("Jones"))
          .body("email",equalTo("mjones@acme.com"))
          .body("phone",equalTo("2341674376"));
    //cleanup
    given()
          .when()
          .delete("contacts/Mary");
  }
}
```



DELETE contact

```
@Test
public void testDeleteContact() {
  // add the contact to be deleted book
  Contact contact = new Contact("Bob", "Smith", "bobby@hotmail.com", "76528765498");
  given()
        .contentType("application/json")
        .body(contact)
        .when().post("/contacts").then()
        .statusCode(200);

  given()
        .when()
        .delete("contacts/Bob");

  given()
        .when()
        .get("contacts/Bob")
        .then()
        .statusCode(404)
        .and()
        .body("errorMessage",equalTo("Contact with firstname= Bob is not available"));
}
```

✔ ContactsRESTTest	3 s 719 ms
✔ testGetOneContact	3 s 74 ms
✔ testDeleteContact	645 ms

POST contact

```
@Test
public void testAddContact() {
   // add the contact
   Contact contact = new Contact("Bob", "Smith", "bobby@hotmail.com", "76528765498");
   given()
         .contentType("application/json")
         .body(contact)
         .when().post("/contacts").then()
         .statusCode(200);
   // get the contact and verify
   given()
         .when()
         .get("contacts/Bob")
         .then()
         .statusCode(200)
         .and()
         .body("firstName",equalTo("Bob"))
         .body("lastName",equalTo("Smith"))
         .body("email",equalTo("bobby@hotmail.com"))
         .body("phone",equalTo("76528765498"));
   //cleanup
   given()
         .when()
         .delete("contacts/Bob");
}
```

✔ ContactsRESTTest	4 s 181 ms
✔ testGetOneContact	3 s 378 ms
✔ testDeleteContact	673 ms
✔ testAddContact	130 ms

PUT contact

```
@Test
public void testUpdateContact() {
  // add the contact
  Contact contact = new Contact("Bob", "Smith", "bobby@hotmail.com", "76528765498");
  Contact updateContact = new Contact("Bob", "Johnson", "bobby@gmail.com", "89765123");
  given()
      .contentType("application/json")
      .body(contact)
      .when().post("/contacts").then()
      .statusCode(200);
  //update contact
  given()
      .contentType("application/json")
      .body(updateContact)
      .when().put("/contacts/"+updateContact.getFirstName()).then()
      .statusCode(200);
  // get the contact and verify
  given()
      .when()
      .get("contacts/Bob")
      .then()
      .statusCode(200)
      .and()
      .body("firstName",equalTo("Bob"))
      .body("lastName",equalTo("Johnson"))
      .body("email",equalTo("bobby@gmail.com"))
      .body("phone",equalTo("89765123"));
  //cleanup
  given()
      .when()
      .delete("contacts/Bob");
}
```

✔ ContactsRESTTest	5 s 230 ms
✔ testGetOneContact	4 s 118 ms
✔ testDeleteContact	779 ms
✔ testUpdateContact	183 ms
✔ testAddContact	150 ms

Get all contacts

```
@Test
public void testGetAllContacts() {
  // add the contacts
  Contact contact = new Contact("Bob", "Smith", "bobby@hotmail.com", "76528765498");
  Contact contact2 = new Contact("Tom", "Johnson", "tomjohnson@gmail.com", "543256789");
  given()
      .contentType("application/json")
      .body(contact)
      .when().post("/contacts").then()
      .statusCode(200);
  given()
      .contentType("application/json")
      .body(contact2)
      .when().post("/contacts").then()
      .statusCode(200);
  // get all contacts and verify
  given()
      .when()
      .get("contacts")
      .then()
      .statusCode(200)
      .and()
      .body("contacts.firstName", hasItems("Bob", "Tom"))
      .body("contacts.lastName",hasItems("Smith", "Johnson"))
      .body("contacts.email",hasItems("bobby@hotmail.com", "tomjohnson@gmail.com"))
      .body("contacts.phone",hasItems("76528765498", "543256789"));
  //cleanup
  given()
      .when()
      .delete("contacts/Bob");
  given()
      .when()
      .delete("contacts/Tom");
}
```

✔ ContactsRESTTest	4 s 572 ms
✔ testGetOneContact	3 s 298 ms
✔ testDeleteContact	698 ms
✔ testUpdateContact	173 ms
✔ testGetAllContacts	214 ms
✔ testAddContact	189 ms

Main point

- Writing and maintaining tests takes time. You should write only those tests that you really need. Do not blindly write tests for every method you write.

Science of Consciousness: Do less and accomplish more.