

Computer System Design & Application

计算机系统设计与应用A

陶伊达 (TAO Yida)

taoyd@sustech.edu.cn



Lecture 13

- Logging
 - Logging for Java
 - Logging for Spring Boot
- Testing
 - Software Testing Overview
 - JUnit Testing
 - Spring Boot Testing

What is Logging?

Logging in Java refers to the process of recording information, events, or errors that occur during the execution of a program.

Why Logging Matters?

- Troubleshooting
- Monitoring
- Performance optimization
- Security
- Historical analysis

Logging with `system.out.println`

- Every Java programmer is familiar with inserting calls to `System.out.println` into troublesome code to gain insight into program behavior.
- Of course, once you have figured out the cause of trouble, you remove the print statements, only to put them back in when the next problem surfaces.
- Performance could be affected for many `println()`
- All things will be printed with no filter (flooded console)

Java's logging API is designed to overcome this problem!

Requirements for Logging

1

Logs can have different formats: plain text, XML, HTML, etc.

2

Logs can be handled differently: display in consoles, save to files, etc.

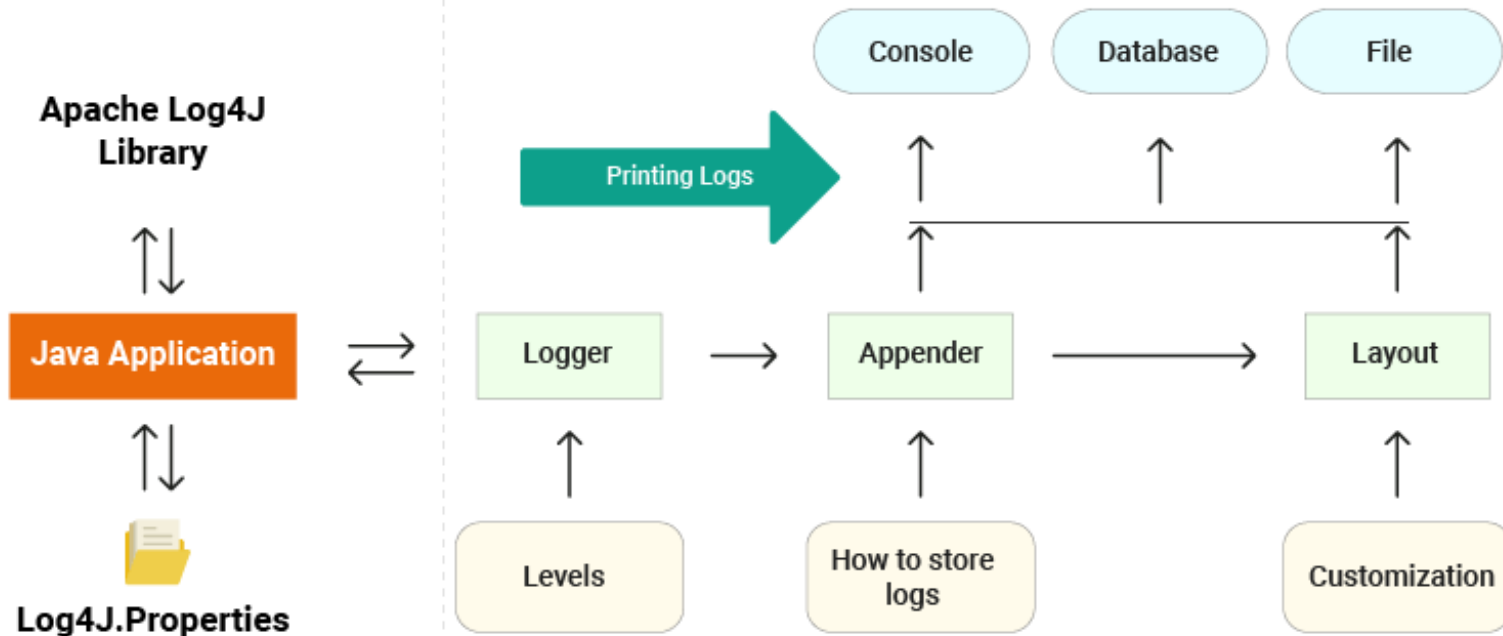
3

Logs can have different levels and different filters.

4

Application can have different logging mechanisms for different classes and packages.

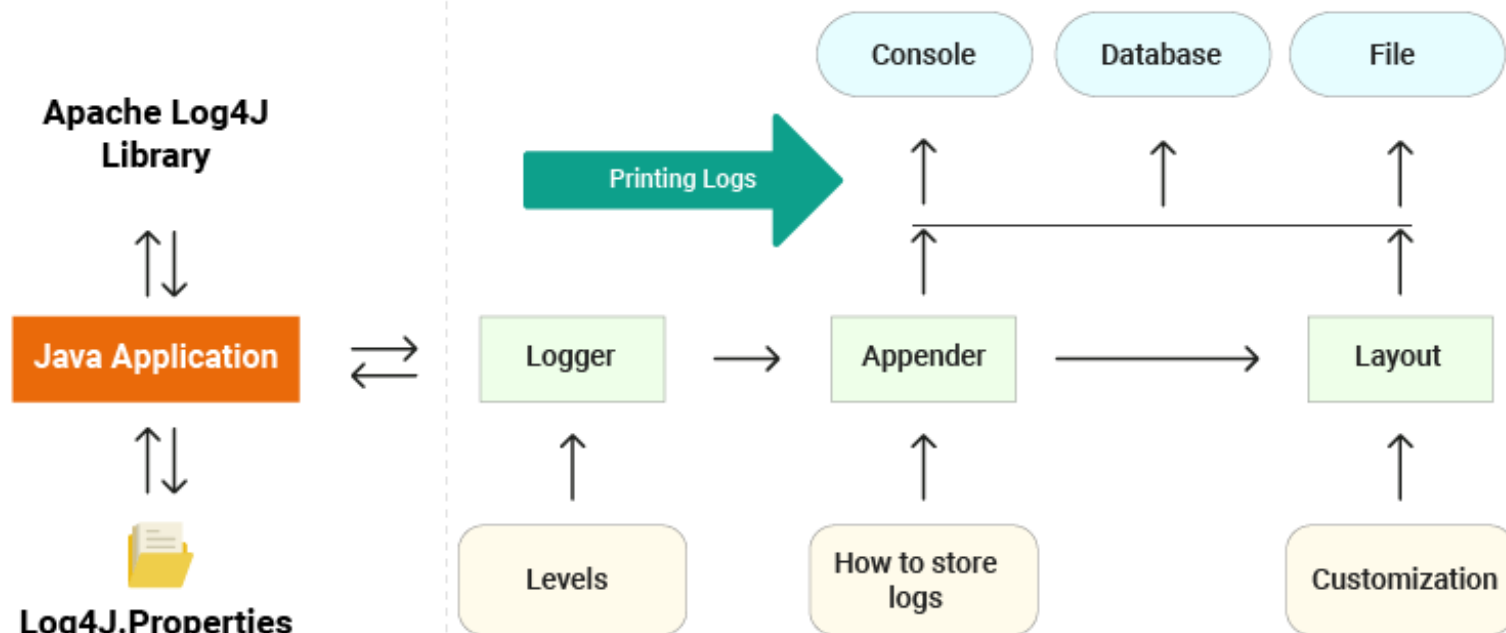
Log4J Engine



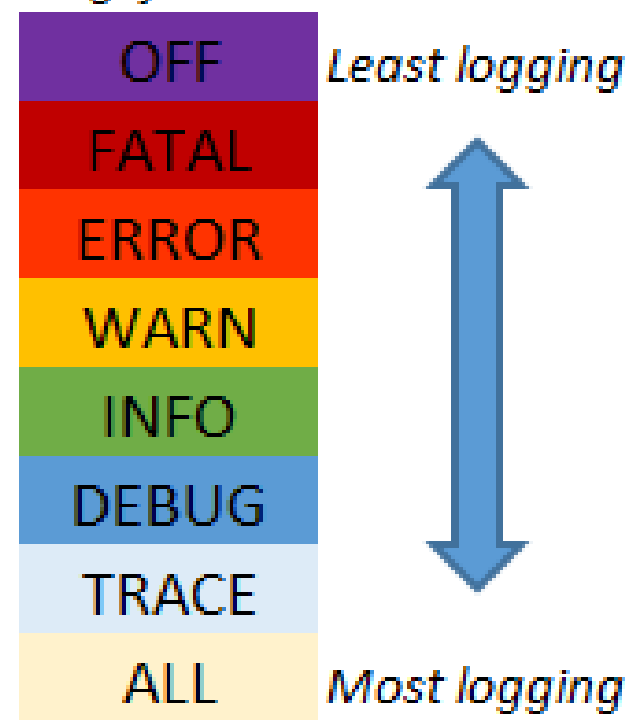
A **logger** is a named entity in Log4j that is responsible for capturing and processing log messages.

Apache Log4j

Log4J Engine



Log4j levels



Apache Log4j

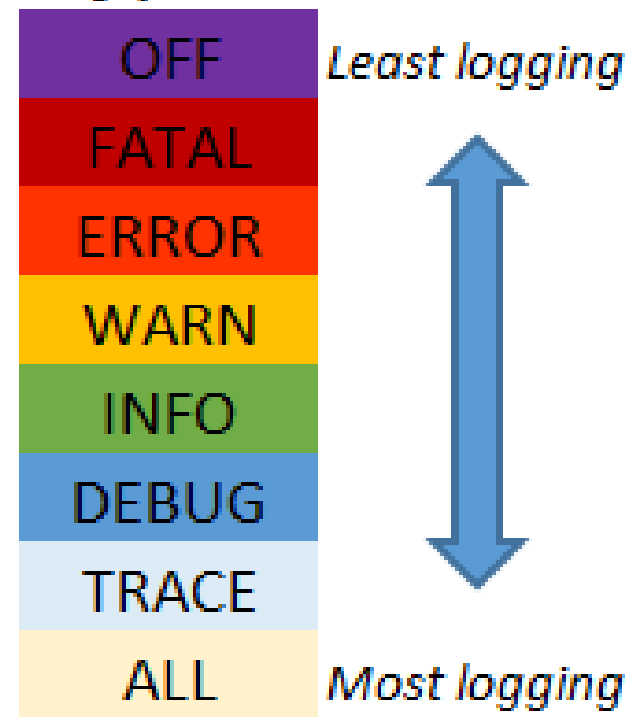
Authentication Module

```
INFO : User alice logged in.  
WARN : User bob entered an invalid password.  
ERROR: User bob entered an invalid password three times, user locked out.
```

Email Agent

```
DEBUG: Loading user bob from database jdbc:...  
INFO : Emailed user bob: three login attempts rejected, user locked out.  
ERROR: Email to alice bounced back; subject: Your weekly summary.
```

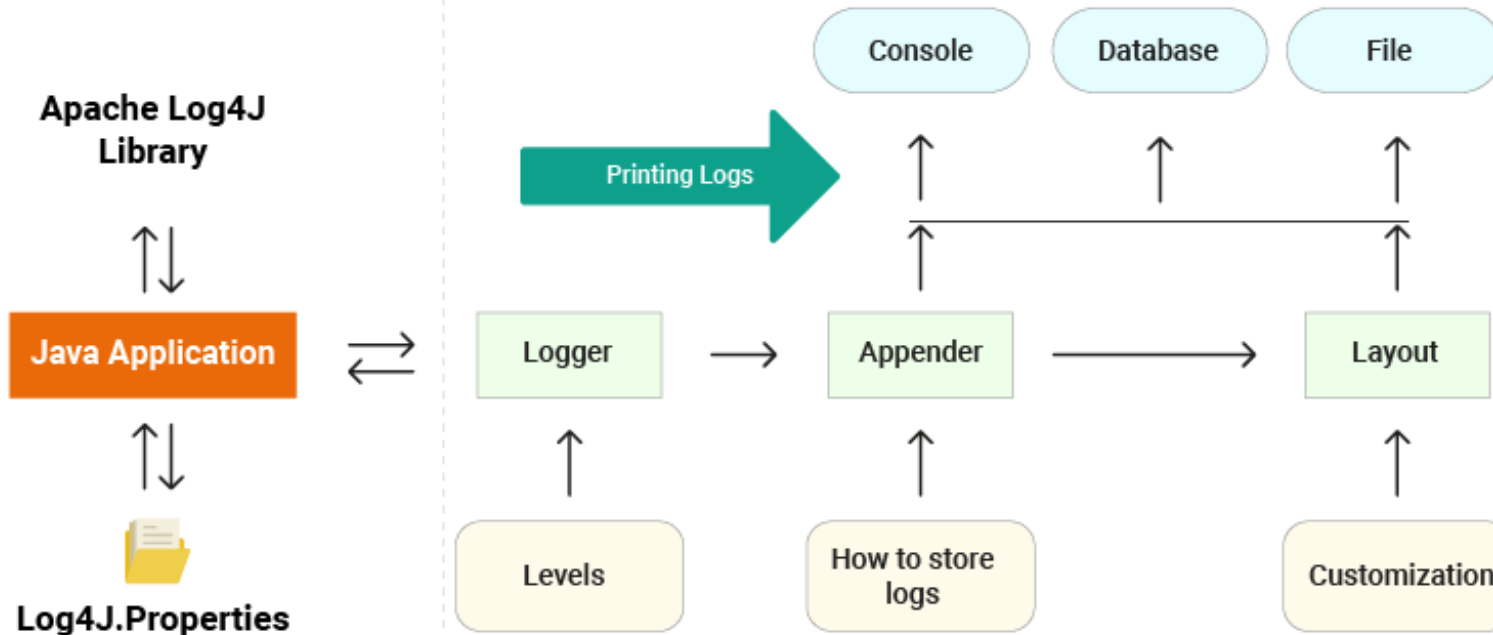
Log4j levels



<https://garygregory.wordpress.com/2015/09/10/the-art-of-test-driven-development-understanding-logging/>

Apache Log4j

Log4J Engine



- **Appenders** are responsible for outputting log messages to various destinations, such as the console, files, databases, or remote servers.
- Log4j provides a variety of built-in appenders, and users can also create custom appenders to suit their specific needs.

Apache Log4j

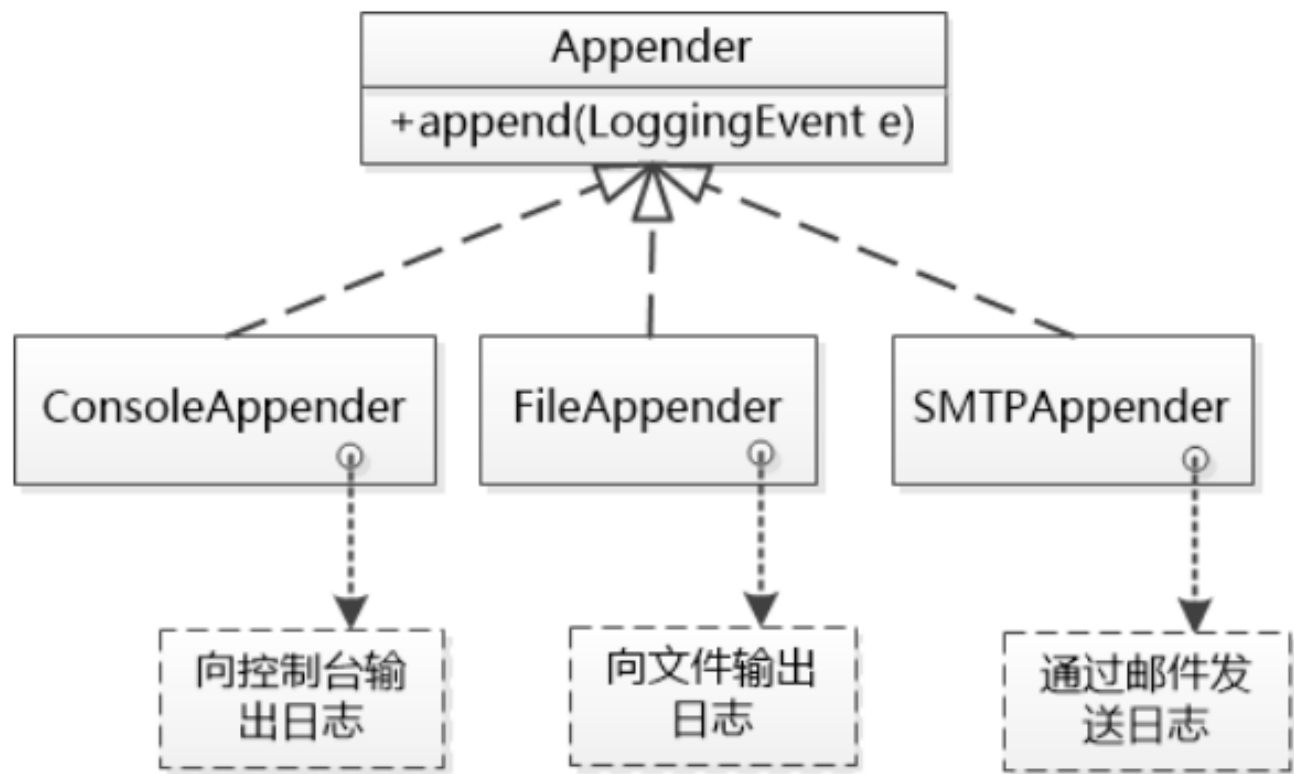
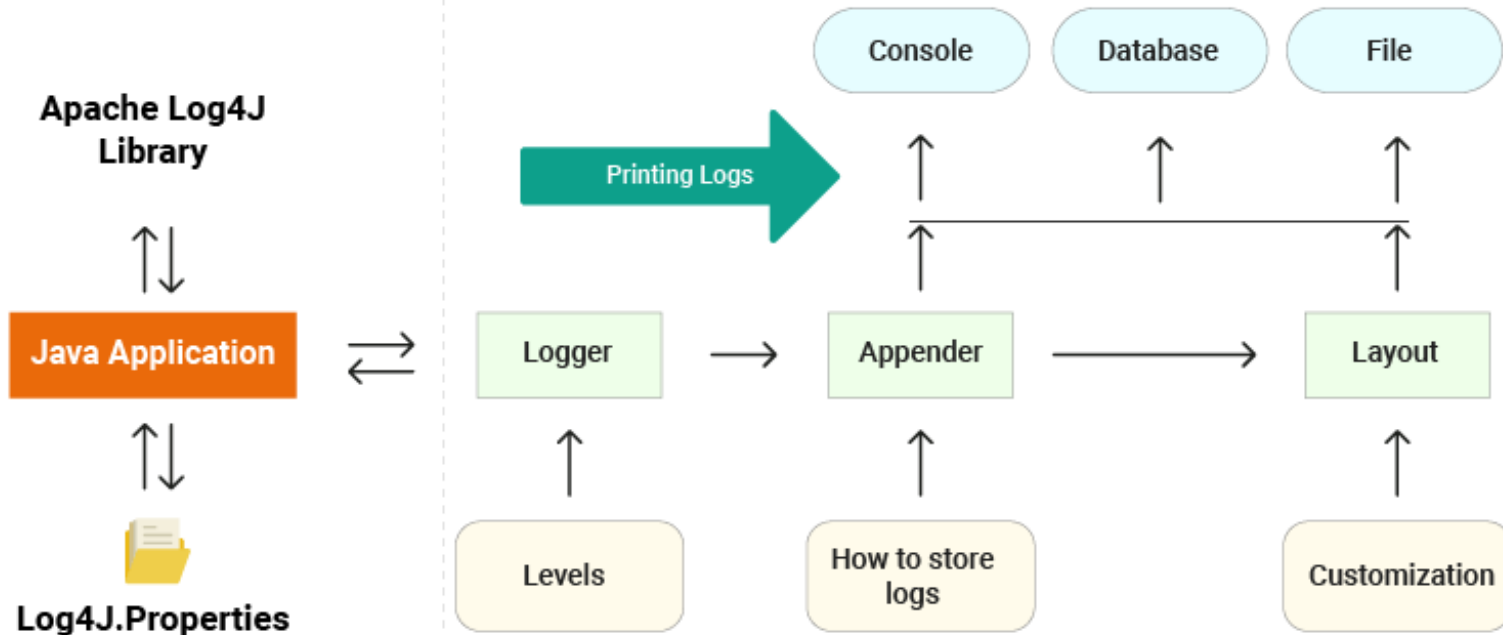


Image source: 《码农翻身》刘欣

- **Appenders** are responsible for outputting log messages to various destinations, such as the console, files, databases, or remote servers.
- Log4j provides a variety of built-in appenders, and users can also create custom appenders to suit their specific needs.

Apache Log4j

Log4J Engine



A **layout (formatter)** defines the format of log messages. Common layouts include simple text, HTML, and XML formats.

Apache Log4j

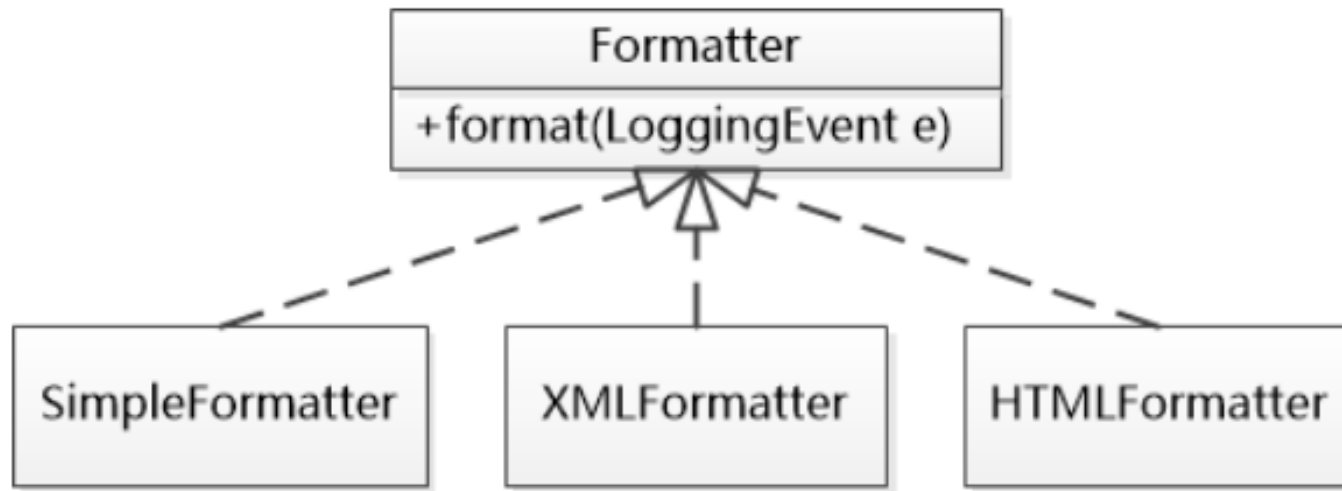
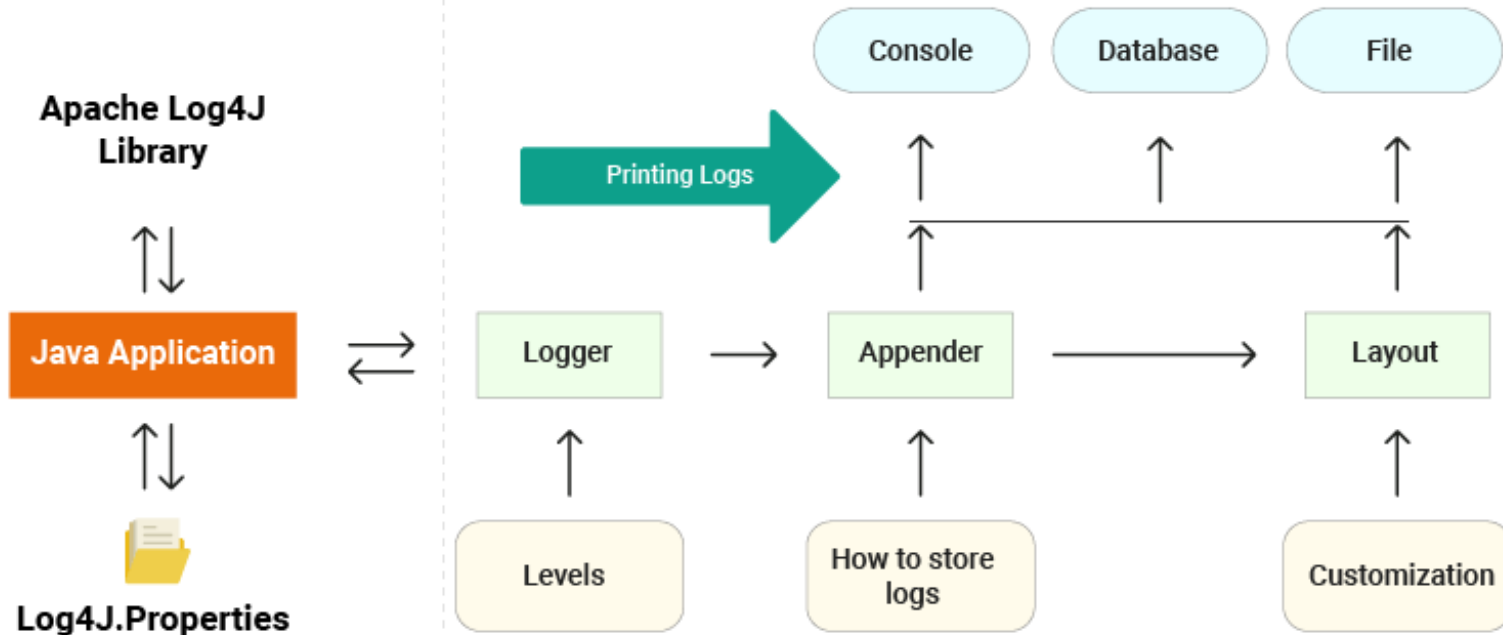


Image source: 《码农翻身》刘欣

A **layout (formatter)** defines the format of log messages. Common layouts include simple text, HTML, and XML formats.

Apache Log4j

Log4J Engine



- The **log4j.properties** file is a log4j configuration file which stores properties in key-value pairs.
- This file contains the entire runtime configuration used by log4j, such as appenders information, log level information and output file names for file appenders.

Apache Log4j

```
log4j.rootLogger=日志等级, AppenderNameA, , , ,
log4j.appender.AppenderNameA=要使用的Appender
log4j.appender.AppenderNameA.PropertyA=PropertyA的值
log4j.appender.AppenderNameA.PropertyB=PropertyB的值
log4j.appender.AppenderNameA.PropertyC=PropertyC的值

log4j.appender.AppenderNameA.layout=要使用的Layout
log4j.appender.AppenderNameA.layout.PropertyA=PropertyA的值
log4j.appender.AppenderNameA.layout.PropertyB=PropertyB的值
log4j.appender.AppenderNameA.layout.PropertyC=PropertyC的值
```

```
log4j.rootLogger=debug,cons

log4j.appender.cons=org.apache.log4j.ConsoleAppender
log4j.appender.cons.target=System.out
log4j.appender.cons.layout=org.apache.log4j.PatternLayout
log4j.appender.cons.layout.ConversionPattern=%m%n
```

Image source: <https://zhuanlan.zhihu.com/p/138026497>

- The **log4j.properties** file is a log4j configuration file which stores properties in key-value pairs.
- This file contains the entire runtime configuration used by log4j, such as appenders information, log level information and output file names for file appenders.

Apache Log4j

```
# ===== 控制台 日志记录 =====
log4j.appender.cons=org.apache.log4j.ConsoleAppender
# 设置当前appender的日志等级为info, 当方法的优先级大于info时, 控制台才会有输出
log4j.appender.cons.threshold=info
# 设置日志输出方式, System.out 和 System.err 两种选择
log4j.appender.cons.target=System.out
# 设置为true, 表示创建新的System.out 对象, 不使用System类中的out属性
log4j.appender.cons.follow=true
log4j.appender.cons.layout=org.apache.log4j.SimpleLayout
```

```
# ===== 文件 日志记录 =====
log4j.appender.myFile=org.apache.log4j.FileAppender
# 文件存储路径
log4j.appender.myFile.file=./log.txt
# 是否以追加的形式向日志文件中写入内容, 默认为true, 不会覆盖之前的内容, 否则只会保留最后一次写
log4j.appender.myFile.append=false
log4j.appender.myFile.layout=org.apache.log4j.SimpleLayout
```

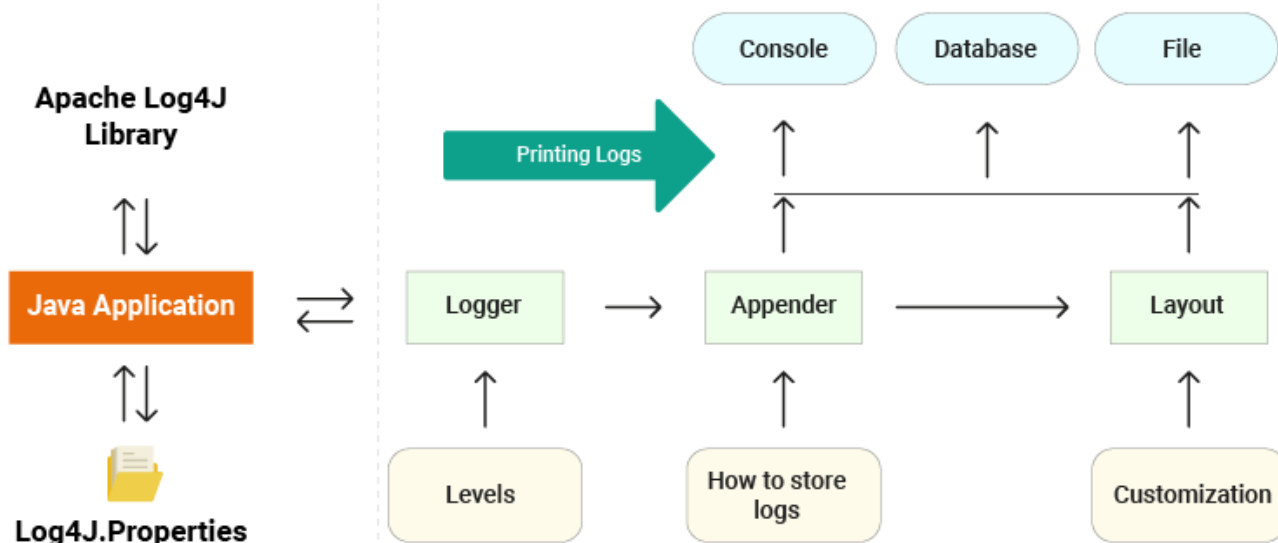
Image source: <https://zhuanlan.zhihu.com/p/138026497>

```
# ===== 滚动文件 日志记录 =====
# 当日志达到一定大小时, 将重新创建新文件记录日志
log4j.appender.myrFile=org.apache.log4j.RollingFileAppender
log4j.appender.myrFile.file=./log.txt
# 最多备份文件的个数, 当文件大小超过设置的值时, 会将原内容进行备份。
# 该值指定了备份文件的个数, 如果超过数量, 则会删除掉最早的备份文件, 如果为0 则不进行备份
log4j.appender.myrFile.maxBackupIndex=5
# 每个文件的最大容量 默认单位是b, 可以指定 "KB", "MB" 或者 "GB", 当文件超过该大小时, 会将其
log4j.appender.myrFile.maxFileSize=1024
# 每个文件的最大容量, 类似于 maxFileSize, 不过是long类型, 即不可有单位, 单位是b
log4j.appender.myrFile.maximumFileSize=1024
log4j.appender.myrFile.layout=org.apache.log4j.SimpleLayout
```

```
# ===== 每日滚动文件 日志记录 =====
# 日志按天进行备份
log4j.appender.mydFile=org.apache.log4j.DailyRollingFileAppender
# 前一天日志的备份文件的后缀格式(后缀为前一天日期, 格式为日期格式)
log4j.appender.mydFile.datePattern=yyyyMMdd
# 当天的日志的记录文件路径
log4j.appender.mydFile.file=./nl.txt
log4j.appender.mydFile.layout=org.apache.log4j.SimpleLayout
```

Apache Log4j

Log4J Engine



```
import com.foo.Bar;

// Import log4j classes.
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.LogManager;

public class MyApp {

    // Define a static logger variable so that it references the
    // Logger instance named "MyApp".
    private static final Logger logger = LogManager.getLogger(MyApp.class);

    public static void main(final String... args) {

        // Set up a simple configuration that logs on the console.

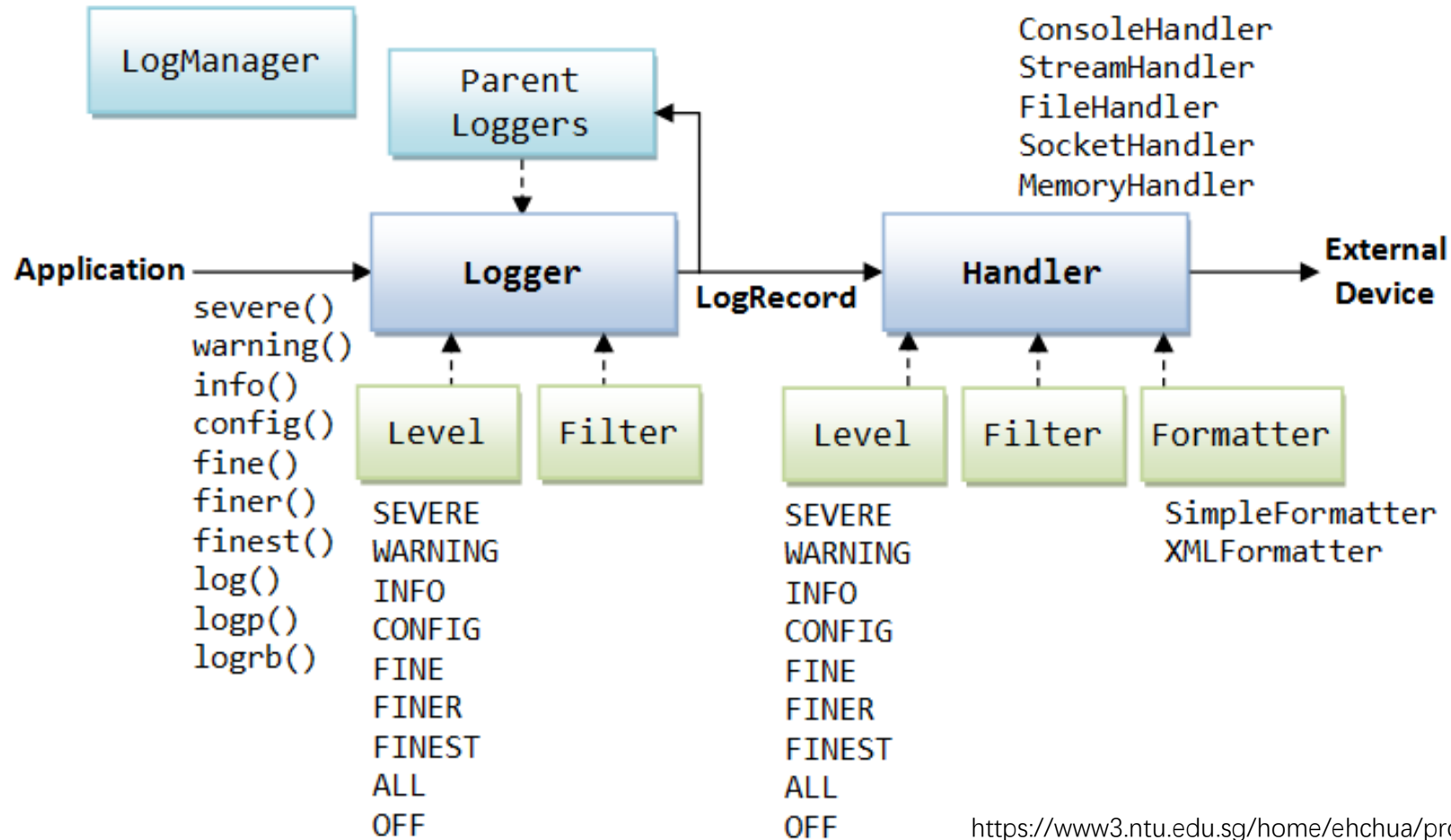
        logger.trace("Entering application.");
        Bar bar = new Bar();
        if (!bar.doIt()) {
            logger.error("Didn't do it.");
        }
        logger.trace("Exiting application.");
    }
}
```

<https://logging.apache.org/log4j/2.x/manual/configuration.html>

Apache Log4j

Java logging framework

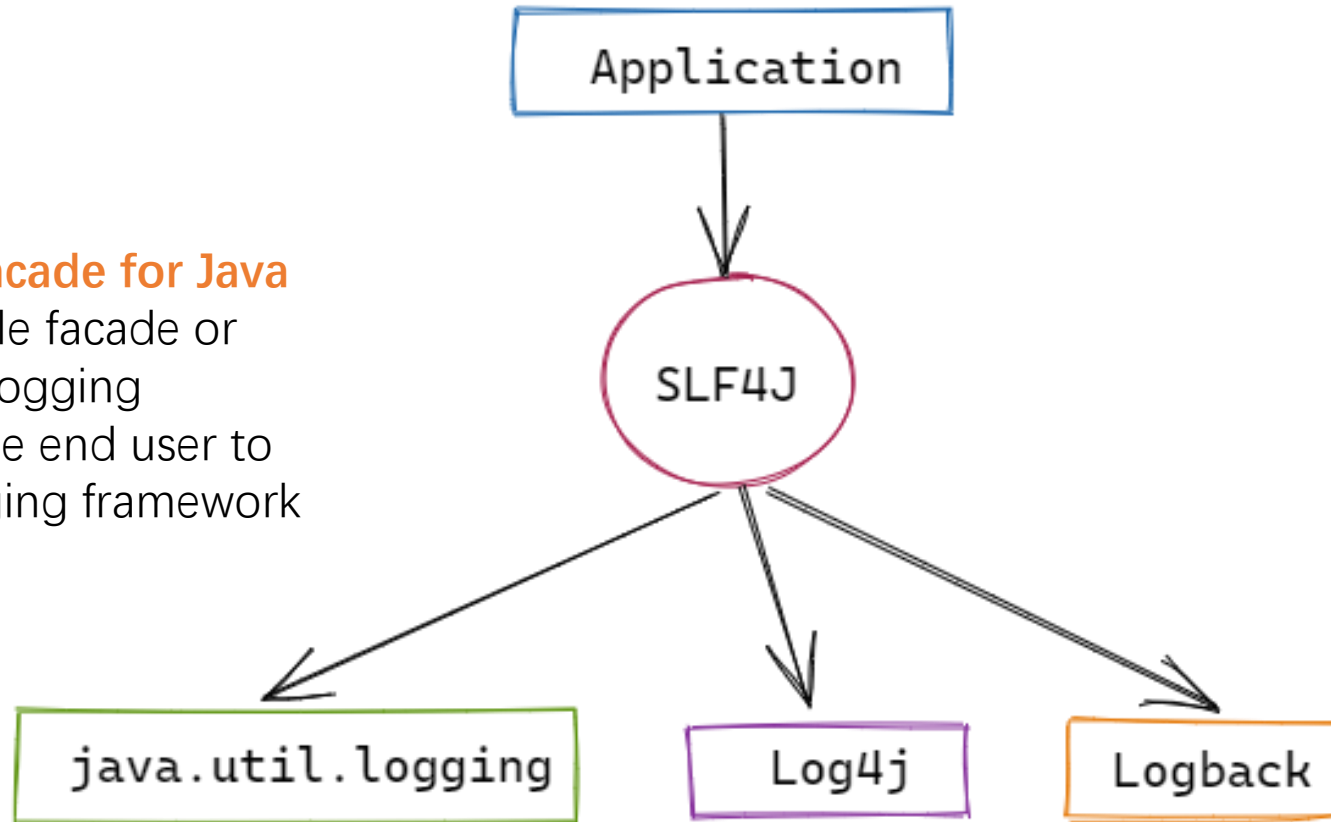
(java.util.logging, since JDK 1.4)



<https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaLogging.html>

Use Logging in Java Applications

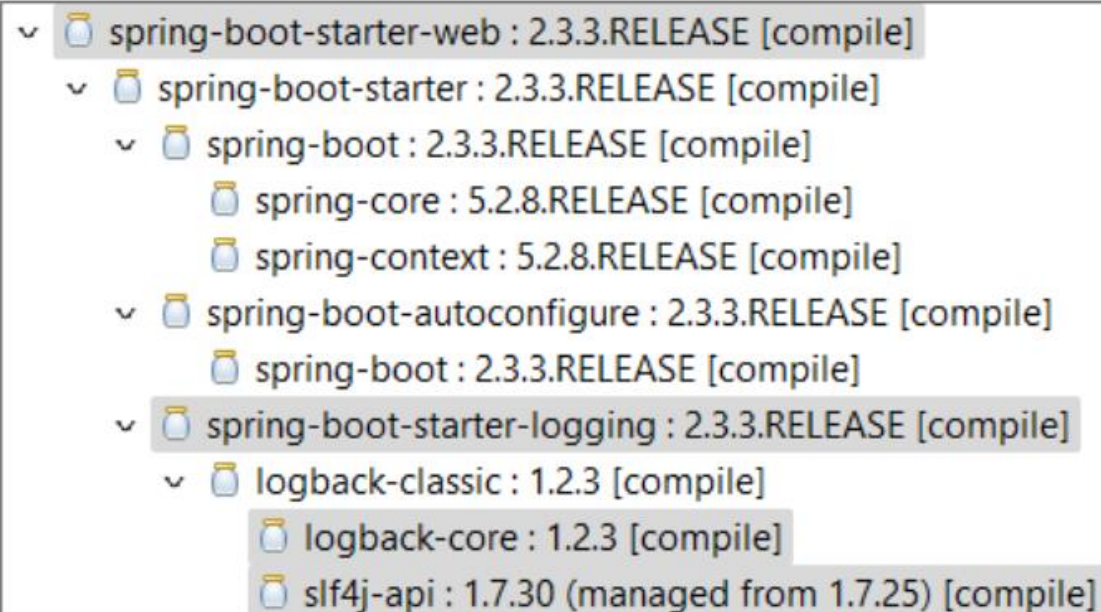
The **Simple Logging Facade for Java** (SLF4J) serves as a simple facade or abstraction for various logging frameworks, allowing the end user to plug in the desired logging framework at deployment time.



Logging in Spring Boot

- In Spring Boot, default logging uses **Logback** to log DEBUG messages into the Console.
- Most boot starters, such as `spring-boot-starter-web`, depends on `spring-boot-starter-logging`, which pulls in logback for us.

Dependency Hierarchy



Default Logging in Spring Boot

- By default, when no default configuration file is found, logback will add a ConsoleAppender to the root logger and this will log all the messages in the Console.
- By default, the output is formatted using a PatternLayoutEncoder
- The default logging level of the Logger is preset to INFO, meaning that TRACE and DEBUG messages are not visible

Default Logging in Spring Boot

[illegible]

Default logging for Spring Boot DOES NOT count for the logging requirements in the final project!

2023-12-12T20:20:18.661+08:00	INFO	28164	---	[
2023-12-12T20:20:18.668+08:00	INFO	28164	---	[
2023-12-12T20:20:19.454+08:00	INFO	28164	---	[
2023-12-12T20:20:19.474+08:00	INFO	28164	---	[
2023-12-12T20:20:19.475+08:00	INFO	28164	---	[
2023-12-12T20:20:19.532+08:00	INFO	28164	---	[
2023-12-12T20:20:19.532+08:00	INFO	28164	---	[

```
main] c.e.s.SpringLoggingDemoApplication : Starting SpringLoggin
main] c.e.s.SpringLoggingDemoApplication : No active profile set
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized wi
main] o.apache.catalina.core.StandardService : Starting service [Tom
main] o.apache.catalina.core.StandardEngine : Starting Servlet engi
main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring e
main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationCo
```

Use Logging in Spring Boot

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringLoggingDemoApplication {
    private static final Logger LOGGER= LoggerFactory.getLogger(SpringLoggingDemoApplication.class);

    public static void main(String[] args) {
        SpringApplication.run(SpringLoggingDemoApplication.class, args);

        LOGGER.info("Customized INFO log {}", 1);
        LOGGER.debug("Customized DEBUG log {}", 2);
        LOGGER.error("Customized ERROR log {}", 3);
        LOGGER.trace("Customized TRACE log {}", 4);
    }
}
```


Use Logging in Spring Boot

```
\W _ _ ) | | _ | | | | | | | ( | | ) ) ) )  
' | _ _ | . _ | | | | | | \ _ , | / / / /  
=====|_|=====|_ _/=/_/_/_/  
  
:: Spring Boot ::                (v3.2.0)
```

2023-12-12T20:30:59.865+08:00	INFO	24732	---	[main]	c.e.s.SpringLoggingDemoApplication	:	Starting SpringLoggingD
2023-12-12T20:30:59.867+08:00	INFO	24732	---	[main]	c.e.s.SpringLoggingDemoApplication	:	No active profile set,
2023-12-12T20:31:00.679+08:00	INFO	24732	---	[main]	o.s.b.w.embedded.tomcat.TomcatWebServer	:	Tomcat initialized with
2023-12-12T20:31:00.688+08:00	INFO	24732	---	[main]	o.apache.catalina.core.StandardService	:	Starting service [Tomca
2023-12-12T20:31:00.688+08:00	INFO	24732	---	[main]	o.apache.catalina.core.StandardEngine	:	Starting Servlet engine
2023-12-12T20:31:00.757+08:00	INFO	24732	---	[main]	o.a.c.c.C.[Tomcat].[localhost].[/]	:	Initializing Spring emk
2023-12-12T20:31:00.757+08:00	INFO	24732	---	[main]	w.s.c.ServletWebServerApplicationContext	:	Root WebApplicationCont
2023-12-12T20:31:01.040+08:00	INFO	24732	---	[main]	o.s.b.w.embedded.tomcat.TomcatWebServer	:	Tomcat started on port
2023-12-12T20:31:01.046+08:00	INFO	24732	---	[main]	c.e.s.SpringLogginqDemoApplication	:	Started SpringLogginqDe
2023-12-12T20:31:01.048+08:00	INFO	24732	---	[main]	c.e.s.SpringLoggingDemoApplication	:	Customized INFO log 1
2023-12-12T20:31:01.049+08:00	ERROR	24732	---	[main]	c.e.s.SpringLoggingDemoApplication	:	Customized ERROR log 3



Lecture 13

- Logging
 - Logging for Java
 - Logging for Spring Boot
- Testing
 - Software Testing Overview
 - JUnit Testing
 - Spring Boot Testing

Software Testing

- Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do.
- It involves execution of software/system components using manual or automated tools to evaluate one or more properties of interest.
- The benefits of testing include preventing bugs, reducing development costs and improving performance.

<https://www.ibm.com/topics/software-testing>

Types of Software Testing

- **Unit Test (单元测试)**: Test individual method/class in isolation. A unit is the smallest testable component of an application.
- **Integration Test (集成测试)**: Test a group of associated components/classes and ensure that they operate together.
- **System Test (系统测试)**: evaluating a complete software system to ensure that it functions as expected and meets the specified requirements
- **Acceptance Test (验收测试)**: operate on a fully integrated system, testing against the user interface
- **Regression Test (回归测试)**: Tests to ensure that a change does not break the system or introduce new faults.
- (there are more than 150 types of testing types and still adding)

Code Coverage (代码覆盖率)

- A measurement of how well your test set is covering your source code (i.e. to what extent is the source code covered by the set of test cases).
- It is generally considered (?) that 80% coverage is a good goal to aim for.
- Granularity
 - Statements/blocks/methods coverage
 - Condition/Decision/Loop coverage

JUnit

- JUnit is an open-source Unit Testing Framework for Java
- Initially designed by Erich Gamma and Kent Beck
- JUnit 5
 - JUnit 5 is the latest version and uses the new `org.junit.jupiter` package for its annotations and classes
 - JUnit 5 leverages features from Java 8 or later, such as lambda functions, making tests more powerful and easier to maintain.
 - JUnit 5 has added some very useful new features for describing, organizing, and executing tests

A Simple JUnit Example

```
public class Calculator {  
  
    static double add(double... operands) {  
        return DoubleStream.of(operands)  
            .sum();  
    }  
  
    static double multiply(double... operands) {  
        return DoubleStream.of(operands)  
            .reduce(1, (a, b) -> a * b);  
    }  
}
```

```
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*;
```

```
class CalculatorTest {
```

```
    @Test
```

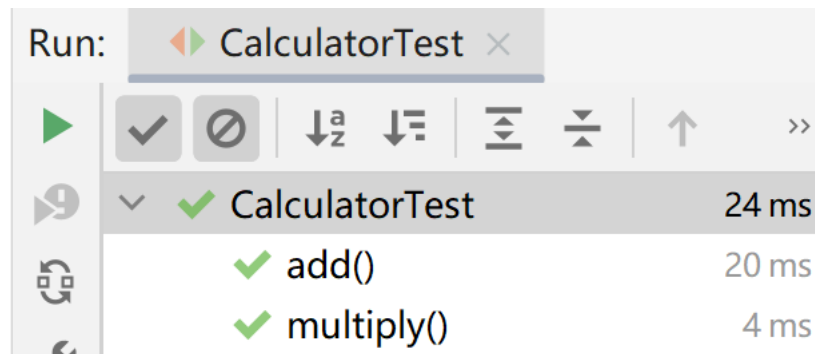
```
    void add() {  
        assertEquals(4, Calculator.add(2, 2));  
    }
```

```
    @Test
```

```
    void multiply() {  
        assertAll(() -> assertEquals(4,  
            Calculator.multiply(2, 2)),  
            () -> assertEquals(-4,  
            Calculator.multiply(2, -2)),  
            () -> assertEquals(4,  
            Calculator.multiply(-2, -2)),  
            () -> assertEquals(0,  
            Calculator.multiply(1, 0)));  
    }  
}
```

A Simple JUnit Example

- @Test annotation denotes that this method is a test method
- Assertions (断言) is a collection of utility methods that support asserting conditions in tests.
- Run the test class CalculatorTest will execute all its test methods



```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
```

```
class CalculatorTest {
```

```
    @Test
```

```
    void add() {
```

```
        assertEquals(4, Calculator.add(2, 2));
```

```
    }
```

```
    @Test
```

```
    void multiply() {
```

```
        assertAll(() -> assertEquals(4,
```

```
            Calculator.multiply(2, 2)),
```

```
            () -> assertEquals(-4,
```

```
                Calculator.multiply(2, -2)),
```

```
            () -> assertEquals(4,
```

```
                Calculator.multiply(-2, -2)),
```

```
            () -> assertEquals(0,
```

```
                Calculator.multiply(1, 0)));
```

```
    }
```

```
}
```


Test Classes and Methods

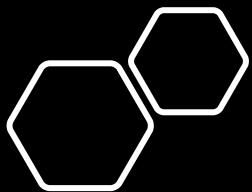
- **Test Class:** any class that contains at least one test method. Test classes must not be abstract and must have a single constructor.
- **Test Method:** any instance method that is annotated with @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, or @TestTemplate.
- **Lifecycle Method:** any method that is annotated with @BeforeAll, @AfterAll, @BeforeEach, or @AfterEach

<https://junit.org/junit5/docs/current/user-guide/#writing-tests-classes-and-methods>

Test Classes and Methods

- Test methods and lifecycle methods may be declared locally within the current test class, inherited from superclasses, or inherited from interfaces
- Test methods and lifecycle methods must **not** be abstract and must **not** return a value (except @TestFactory methods which are required to return a value).
- Test classes, test methods, and lifecycle methods are not required to be public, but they **must not** be private

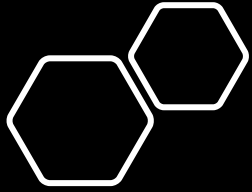
<https://junit.org/junit5/docs/current/user-guide/#writing-tests-classes-and-methods>



Test Instance Lifecycle

In order to allow individual test methods to be executed in isolation and to avoid unexpected side effects due to mutable test instance state, JUnit **creates a new instance of each test class before executing each test method** (default)

```
class CalculatorTest {  
    @Test  
    void add() {  
        assertEquals( expected: 4, Calculator.add(2, 2));  
    }  
  
    @Test  
    void multiply() {  
        assertAll(() -> assertEquals( expected: 4,  
                                     Calculator.multiply( ...operands: 2, 2)),  
                () -> assertEquals( expected: -4,  
                                     Calculator.multiply( ...operands: 2, -2)),  
                () -> assertEquals( expected: 4,  
                                     Calculator.multiply( ...operands: -2, -2)),  
                () -> assertEquals( expected: 0,  
                                     Calculator.multiply( ...operands: 1, 0)));  
    }  
}
```



Test Instance Lifecycle

- In order to allow individual test methods to be executed in isolation and to avoid unexpected side effects due to mutable test instance state, JUnit creates a new instance of each test class before executing each test method (default)
- If you would prefer that JUnit Jupiter execute all test methods on the same test instance, annotate your test class with `@TestInstance(Lifecycle.PER_CLASS)`
 - A new test instance will be created once per test class.
 - If your test methods rely on state stored in instance variables, you may need to reset that state in `@BeforeEach` or `@AfterEach` methods.

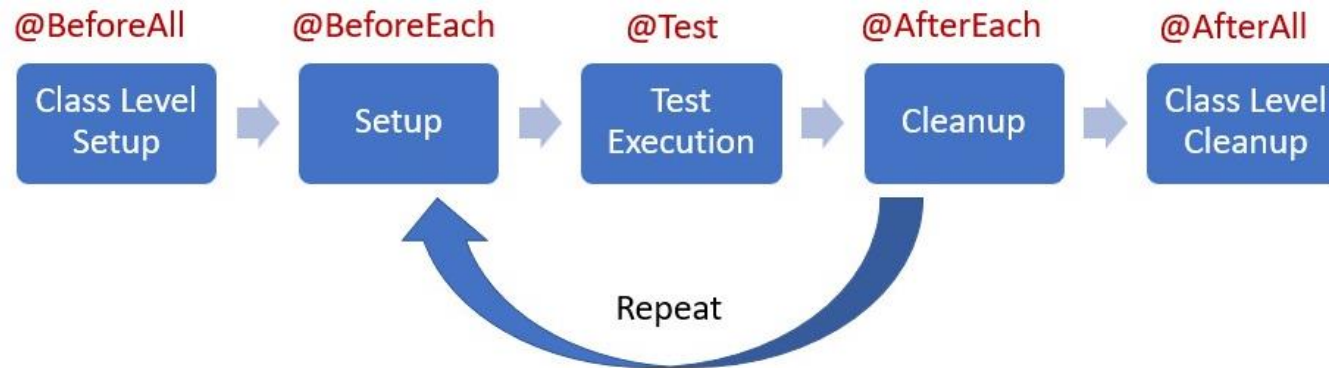
Further reading:

<https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle>

Test Lifecycle

The complete lifecycle of a test case can be seen in 3 phases

1. **Setup:** This phase puts the test infrastructure in place. JUnit provides class level setup (`@BeforeAll`) and method level setup (`@BeforeEach`). Generally, heavy objects like database connections are created in class level setup while lightweight objects like test objects are reset in the method level setup.
2. **Test Execution:** In this phase, the test execution and assertion happen, and results signify a success or failure.
3. **Cleanup:** This phase is used to cleanup the test infrastructure setup in the first phase. Just like setup, teardown also happen at class level (`@AfterAll`) and method level (`@AfterEach`).



Reference: <https://howtodoinjava.com/junit5/junit-5-test-lifecycle/>

@BeforeEach & @AfterEach

```
class CalculatorTest {  
  
    @Test  
    void add() {  
        Calculator c = new Calculator();  
        assertEquals( expected: 4, c.add(2, 2));  
    }  
  
    @Test  
    void multiply() {  
        Calculator c = new Calculator();  
        assertEquals( expected: 6, c.multiply( ...operands: 3, 2, 1));  
    }  
}
```

- @BeforeEach is used to signal that the annotated method should be executed before each @Test method in the current test class.
- @BeforeEach methods must have a void return type, must NOT be private, and must NOT be static

```
class CalculatorTest {  
  
    Calculator c;  
  
    @BeforeEach  
    public void setUp() {  
        this.c = new Calculator();  
    }  
  
    @AfterEach  
    public void tearDown() {  
        this.c = null;  
    }  
  
    @Test  
    void add() {  
        assertEquals( expected: 4, c.add(2, 2));  
    }  
  
    @Test  
    void multiply() {  
        assertEquals( expected: 6, c.multiply( ...operands: 3, 2));  
    }  
}
```

@BeforeAll & @AfterAll

- Generally, heavy objects like database connections are created in class level setup

```
public class DatabaseTest {  
    static Database db;  
  
    @BeforeAll  
    public static void initDatabase() {  
        db = createDb(...);  
    }  
  
    @AfterAll  
    public static void dropDatabase() {  
        ...  
    }  
}
```

- @BeforeAll is used to signal that the annotated method should be executed before all tests in the current test class.
- In contrast to @BeforeEach methods, @BeforeAll methods are only executed once for a given test class.
- @BeforeAll methods must have a void return type, must not be private, and **must be static** by default (unless the PER_CLASS test instance lifecycle is used)

Image: <https://www.liaoxuefeng.com/wiki/1252599548343744/1304049490067490>

Assertions

```
@Test
void standardAssertions(){
    assertEquals(2, Calculator.add(1,1));

    assertEquals(4, Calculator.multiply(2,2),
                 "Optional failure messages");

    assertTrue(Calculator.add(1,1) == 2);

    assertEquals(new int[]{1,2}, new int[]{1,2,3});

    assertNull(null);
}
```

java.lang.Object
org.junit.jupiter.api.Assertions

Assertions is a class/collection of utility methods that support asserting conditions in tests.

If one assert fails, the test will stop and you won't see the results of the remaining asserts

assertAll

```
public static void assertAll(String heading,  
                             Executable... executables)  
    throws MultipleFailuresError
```

Asserts that all supplied executables do not throw exceptions.

```
Address address = unitUnderTest.methodUnderTest();  
assertAll("Should return address of Oracle's headquarter",  
    () -> assertEquals("Redwood Shores", address.getCity()),  
    () -> assertEquals("Oracle Parkway", address.getStreet()),  
    () -> assertEquals("500", address.getNumber())  
);
```

```
org.opentest4j.MultipleFailuresError:  
    Should return address of Oracle's headquarter (3 failures)  
    expected: <Redwood Shores> but was: <Walldorf>  
    expected: <Oracle Parkway> but was: <Dietmar-Hopp-Allee>  
    expected: <500> but was: <16>
```

If any supplied Executable throws an AssertionError, all remaining executables will still be executed, and all failures will be aggregated and reported in a MultipleFailuresError.

Example: <https://stackoverflow.com/questions/40796756/assertall-vs-multiple-assertions-in-junit5>

Assumptions

- Assumptions is a collection of utility methods that support conditional test execution based on assumptions.
- In contrast to failed assertions, which result in a test failure, a failed assumption results in a test being *aborted*.
- Assumptions are typically used whenever it does not make sense to continue execution of a given test method (e.g., if the test depends on something that does not exist in the current runtime environment)

Assumptions

```
private final Calculator calculator = new Calculator();

@Test
void testOnlyOnCiServer() {
    assumeTrue("CI".equals(System.getenv("ENV")));
    // remainder of test
}

@Test
void testOnlyOnDeveloperWorkstation() {
    assumeTrue("DEV".equals(System.getenv("ENV")),
        () -> "Aborting test: not on developer workstation");
    // remainder of test
}

@Test
void testInAllEnvironments() {
    assumingThat("CI".equals(System.getenv("ENV")),
        () -> {
            // perform these assertions only on the CI server
            assertEquals(2, calculator.divide(4, 2));
        });

    // perform these assertions in all environments
    assertEquals(42, calculator.multiply(6, 7));
}
```

Conditional Test Execution

- Entire test classes or individual test methods may be disabled via the `@Disabled` annotation
- Developers could either enable or disable a test based on certain conditions programmatically

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@API(
    status = Status.STABLE,
    since = "5.0"
)
public @interface Disabled {
    String value() default "";
}
```

```
@Test
@EnabledOnOs({ LINUX, MAC })
void onLinuxOrMac() {
    // ...
}

@Test
@DisabledOnOs(WINDOWS)
void notOnWindows() {
    // ...
}
```

```
@Test
@EnabledForJreRange(min = JAVA_9, max = JAVA_11)
void fromJava9to11() {
    // ...
}
```

```
@Test
@DisabledForJreRange(min = JAVA_9)
void notFromJava9toCurrentJavaFeatureNumber() {
    // ...
}
```

<https://junit.org/junit5/docs/current/user-guide/#writing-tests-conditional-execution>

Parameterized Test

- Parameterized tests make it possible to run a test multiple times with different arguments.
- Use the `@ParameterizedTest` annotation
- you must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the test method.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

```
palindromes(String) ✓
├─ [1] candidate=racecar ✓
├─ [2] candidate=radar ✓
└─ [3] candidate=able was I ere I saw elba ✓
```

<https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests>



Lecture 13

- Logging
 - Logging for Java
 - Logging for Spring Boot
- Testing
 - Software Testing Overview
 - JUnit Testing
 - Spring Boot Testing

Preparation

- Finished Lab 13 (and understand it)
- Add 2 services to be tested
 - getOneStudent(Long studentId)
 - addOneStudent(Student student)

```
@Service
public class StudentService {
    7 usages
    private final StudentRepository studentRepository;

    yidatao
    @Autowired
    public StudentService(StudentRepository studentRepository) {
        this.studentRepository = studentRepository;
    }

    1 usage yidatao
    public Student getOneStudent(Long studentId){
        return studentRepository.findById(studentId).get();
    }

    1 usage yidatao
    public void addOneStudent(Student student){
        studentRepository.save(student);
    }

    2 usages yidatao
    public List<Student> getStudents(){
        return studentRepository.findAll();
    }
}
```

Preparation

Add 2 corresponding REST endpoints

- GET /api/students/getOne/{id}
- POST /api/students/save

```
@RestController
@RequestMapping("/api/students")
public class StudentRestController {
    6 usages
    private final StudentService studentService;

    yidatao
    public StudentRestController(StudentService studentService) {
        this.studentService = studentService;
    }

    yidatao
    @GetMapping("/getOne/{id}")
    public Student getOneStudent(@PathVariable("id") Long studentId){
        return studentService.getOneStudent(studentId);
    }

    yidatao
    @PostMapping("/save")
    public String addOneStudent(@RequestBody Student student){
        studentService.addOneStudent(student);
        return "success";
    }

    yidatao
    @GetMapping
    public List<Student> getStudentsByEmail(@RequestParam(value = "email")
                                           Optional<String> email) {
        if (email.isPresent()){
            return studentService.findByEmailLike(email.get());
        }
        return studentService.getStudents();
    }
}
```


Preparation

Manually test the new features

▶ GET <http://localhost:8080/api/students/getOne/1>

<http://localhost:8080/api/students/getOne/1>

HTTP/1.1 200
Transfer-Encoding: chunked
Connection: keep-alive
Content-Type: application/json
Date: Thu, 11 May 2023 03:29:59 GMT
Keep-Alive: timeout=4
Proxy-Connection: keep-alive

```
{  
  "id": 1,  
  "name": "Mary",  
  "email": "mary@gmail.com"  
}
```

Response file saved.
> [2023-05-11T112959.200.json](#)

Response code: 200; Time: 114ms; Content length: 47 bytes

▶ POST <http://localhost:8080/api/students/save>
Content-Type: application/json

```
{"name": "zoe", "email": "zoe@sustech.edu.cn"}
```

<http://localhost:8080/api/students/save>

HTTP/1.1 200
Content-Length: 7
Connection: keep-alive
Content-Type: text/plain;charset=UTF-8
Date: Thu, 11 May 2023 03:58:42 GMT
Keep-Alive: timeout=4
Proxy-Connection: keep-alive

success

Response code: 200; Time: 20ms; Content length: 7 bytes

However, to give yourself more confidence that the application works when you make changes, you want to automate the testing.

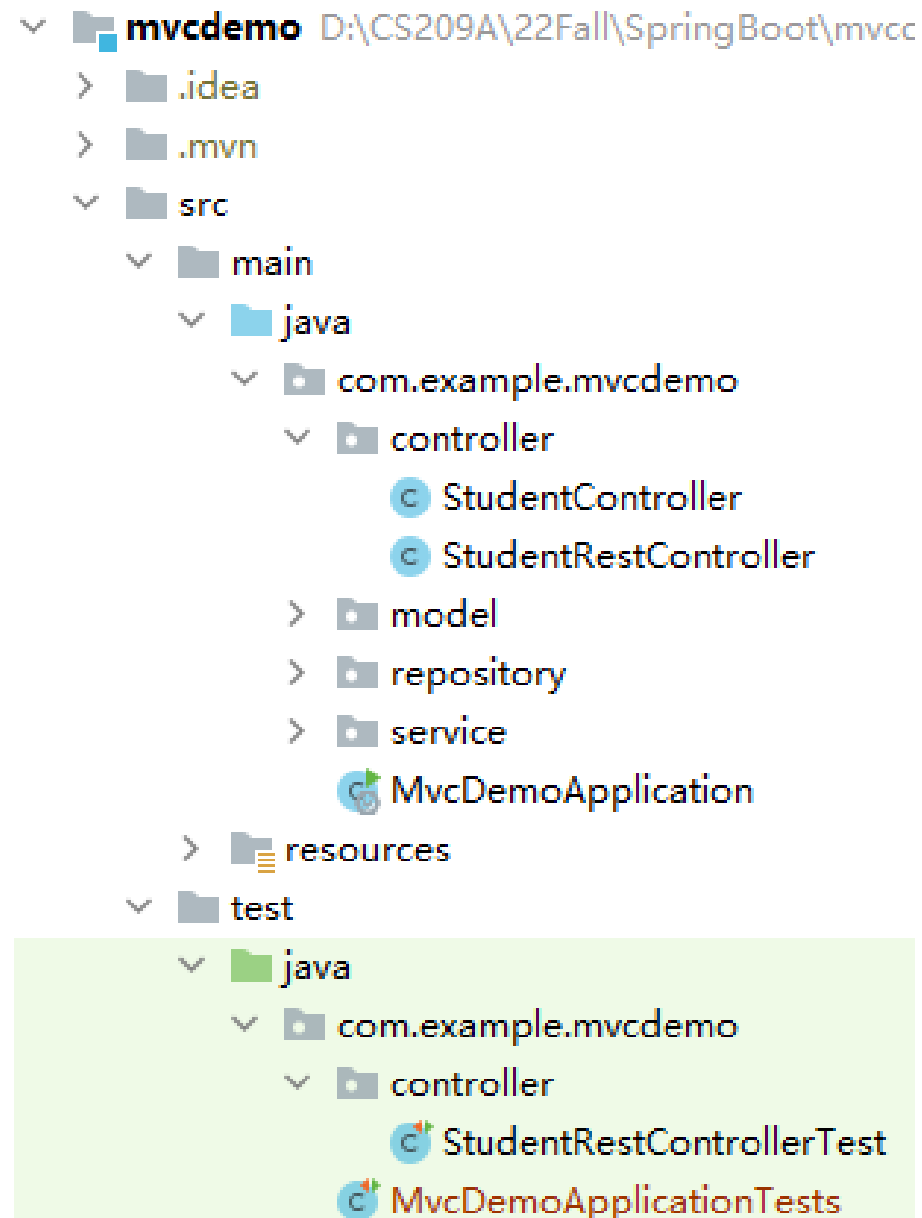

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-test</artifactId>  
  <scope>test</scope>  
</dependency>
```

Convention over Configuration

Spring Boot assumes you plan to test your application, so it adds the necessary dependencies to your pom.xml

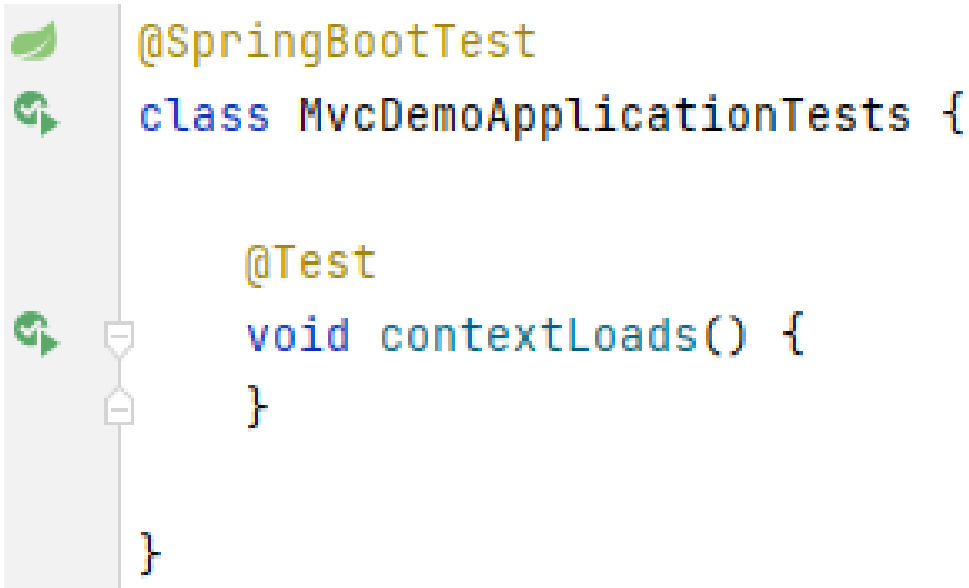
Test Structure

Tests will be organized using the same directory structure as the source



Sanity Check

- A basic test or verification of a system or application's fundamental functionality
- The `@SpringBootTest` annotation tells Spring Boot to look for a main configuration class (one with `@SpringBootApplication`) and use that to start a Spring application context



```
@SpringBootTest
class MvcDemoApplicationTests {

    @Test
    void contextLoads() {
    }

}
```

The image shows a code editor snippet with a light gray background. On the left side, there is a vertical toolbar with icons: a green leaf, a green circular arrow, and a green square with a white arrow. The code is written in a monospaced font with syntax highlighting: `@SpringBootTest` is in orange, `class MvcDemoApplicationTests` is in blue, `@Test` is in orange, `void contextLoads()` is in blue, and the curly braces are in black.

Sanity Check

- To convince yourself that the context is indeed creating your controller, you could add an assertion

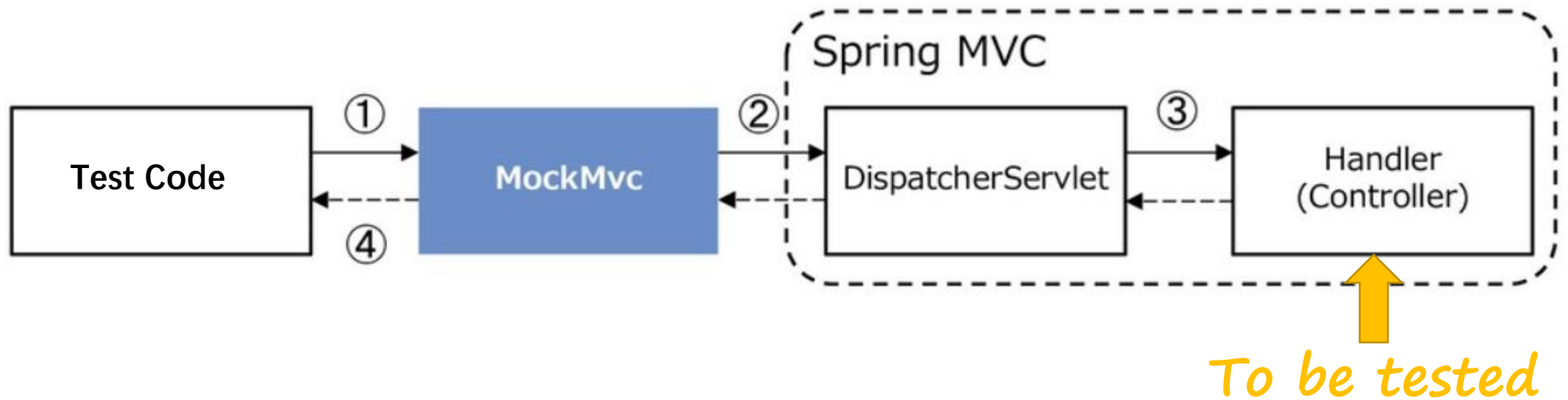
```
@SpringBootTest
class MvcDemoApplicationTests {

    1 usage
    @Autowired
    private StudentController controller;

    @Test
    void contextLoads() {
        assertThat(controller).isNotNull();
    }
}
```

Unit Test the Controller

- We want to unit test the controller (web layer) to check
 - URL mapping
 - Correctly handle incoming HTTP request
 - Correctly return HTTP response
- This means we need to start a web server and expose a port, which is
 - Slow
 - Dependent on network conditions



<https://speakerdeck.com/rshindo/spring-fest-2020?slide=34>

Spring MockMvc

- ① Test code: Set HTTP request URL and parameters
- ② MockMvc mocks a HTTP request and send it to DispatcherServlet
- ③ DispatcherServlet invokes the proper controller/handler
- ④ Test code use MockMvc to assert the response

Test Case

- `@AutoConfigureMockMvc` is a Spring Boot annotation that automatically configures a `MockMvc` instance in a test class.

```
@SpringBootTest
@AutoConfigureMockMvc
class StudentRestControllerTest {
    5 usages
    @Autowired
    private MockMvc mvc;

    yidatao
    @Test
    void getOneStudent() throws Exception {
        mvc.perform(get( urlTemplate: "/api/students/getOne/{id}", ...uriVars: 1))
            .andExpect(status().isOk())
            .andExpect(jsonPath( expression: "$.name").value( expectedValue: "Mary"))
            .andDo(print());
    }
}
```

Test Case

- `MockMvc.perform()`: construct HTTP requests
- `MockMvc.andExpect()`: assert that the response meets the expectation (e.g., status, response content type, response body, etc.)

```
@SpringBootTest
@AutoConfigureMockMvc
class StudentRestControllerTest {
    5 usages
    @Autowired
    private MockMvc mvc;

    yidatao
    @Test
    void getOneStudent() throws Exception {
        mvc.perform(get( urlTemplate: "/api/students/getOne/{id}", ...uriVars: 1))
            .andExpect(status().isOk())
            .andExpect(jsonPath( expression: "$.name").value( expectedValue: "Mary"))
            .andDo(print());
    }
}
```



```
@Test
void getStudentsByEmail() throws Exception {
    mvc.perform(get( urlTemplate: "/api/students/?email={email}", ...uriVars: "jack")).
        andExpect(status().isOk()).
        andExpect(jsonPath( expression: "$.*", hasSize(1))). // ensure 1 student matches
        andExpect(jsonPath( expression: "$.[0].name").value( expectedValue: "Jack")). // verify
        andDo(print());
}
```

JsonPath

- JsonPath is used for querying and manipulating JSON data
- It provides a simple and intuitive syntax for accessing values in a JSON document
- It can be used to assert and verify JSON responses in unit tests

Test Case

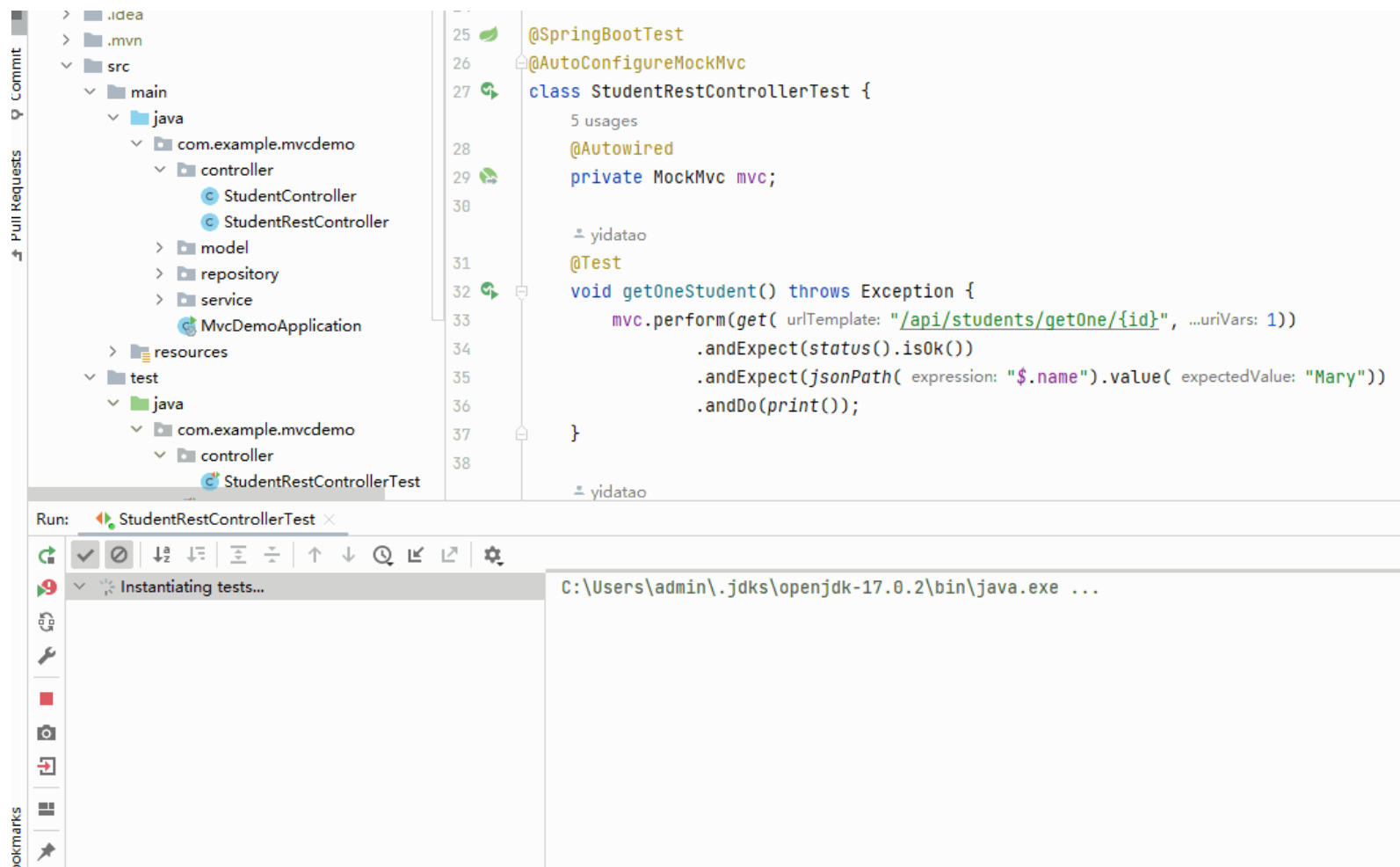
- ObjectMapper supports transformation between Objects and JSON strings
- Can be used to test POST requests, which requires a JSON string as request body

```
@Test
void addOneStudent() throws Exception {
    Student stu = new Student( name: "Jack", email: "jack@mail.com");
    ObjectMapper mapper = new ObjectMapper();

    // convert user to JSON
    String json = mapper.writeValueAsString(stu);

    // send POST request to create user
    mvc.perform(post( urlTemplate: "/api/students/save")
                .contentType(MediaType.APPLICATION_JSON)
                .content(json))
        .andExpect(status().isOk());

    // verify that user was indeed created
    mvc.perform(get( urlTemplate: "/api/students/getOne/{id}", ...uriVars: 4))
        .andExpect(status().isOk())
        .andExpect(jsonPath( expression: "$.name", is(stu.getName())))
        .andExpect(jsonPath( expression: "$.email", is(stu.getEmail())));
}
```



Executing Tests

Next Lecture

- JVM