Software Quality

Jakub Daněk (Yoso Czech s.r.o.)
jakub.danek@yoso.fi

Based on lectures for KIV/OKS by Pavel Herout (KIV ZČU)

Software Quality

Software quality can be measured in:

- **Number of user complaints (does not do what I need)**
  - → good development process, testing
- **Number of bugs found in production**
  - → different testing

- **Maintenance costs**
  - → How to find source of problems as soon as possible: debugging, logs

- **Further development cost**
  - We have talked about this a lot in previous lectures, good architecture saves money
- **And many more…**

Focus Areas (1)

- **Validation**
  - Goal: Show that application does what customer needs
  - Best achieved by good development process and communication with customer
    - → out of our scope
- **Verification**
  - Goal: Show that application does what specification says

Focus Areas (2)

- **Testing**
  - Goal: find errors, try to break the application
- **Debugging**
  - Goal: find cause of error state
  - Relatively easy during development (support of debug tools, ability to modify the code)
  - Relatively difficult in production (no debugger, real data, “must not break stuff”

Focus Areas (3) - Web Applications

- Very wide set of applications with own specifics
- Web Applications change quite often
- Commonly a multi-threaded environment
- Potentially may face higher load than expected (e.g. DoS attacks)

Verification

- **Importance grows with system size and pace of development**
  - Ensuring system works correctly is more important than having a chance of randomly finding an error
  - → repeatability
  - → done at every step of development cycle
  - → automation

Verification

- **Complex processes and systems are difficult to cover with tests**
  - → decomposition
  - → unit-tests

- **Unit tests**
  - Test only small portion of code (code → written by developers)
  - Ensure that individual parts of your code behave according to specification
  - Should be independent on each other and rest of the application

Unit tests

- **Ideally one unit-test suite tests one class/file of your application code**

- **All public methods are tested for:**

  - Correct input → method does what it should do

  - Corner cases → method properly correct input which is outside of normal operational values
    - Example: sort method needs to work correctly with array of length: 0, 1, 2, N

Unit tests

- **All public methods are tested for:**
  - Correct handling of wrong input
    - Example: authentication method has to pass on correct credentials and fail on incorrect credentials, both should be tested
  - Error input – method needs to handle error input based on its specification
    - Example: List<User> findByIds(@Nullable List<Integer> ids) should not fail when ids is null
      - Should return empty list instead

Unit tests

- **→ multiple tests per single method/function**

- **Test coverage should go over 90% of your application logic code (e.g. managers, entity methods)**

  - Helps during refactoring
  - Helps when adding new functions
  - Helps when changing behaviour of existing functions
  - First you change/write new tests → then you modify methods/functions so that the tests pass

Unit tests - Independence

- **In order to achieve high test coverage, unit-tests have to independent on each other**
  - Otherwise you create a data-/dependency-management hell
- **Two approaches to reach independence when testing your classes**
  - Stubs
  - Mocks

Unit tests - Independence

- **Stubs**
  - Special implementations of your dependencies which are used only in tests
    - e.g. for your UserDao interface you create an implementation which returns fixed set of data/ stores inserts in memory and does not touch persistent storage at all
  - Simple to understand
  - As application and number of test cases grow, becomes another maintenance nightmare

Unit tests - Independence

- **Mocks**
  - Instances of your dependencies which can be configured per test-case
    - What should a method return/throw when called with a particular parameter

  Example:

```
@Test
public void generateId() throws Exception {
    Record record = new Record("33");
    String date = dateFormat.format(new Date());

    //dao is the mocked object
    when(dao.getLastNumber(record.getService(), date)).thenReturn(9);
    String expected = record.getService() + date + "00010";

    String id = idService.generateId(record);
    assertEquals(expected, id);
}
```

Unit tests - Summary

- **Ensure application code behaves according to specification**
- **Instead of writing complicated tests of a whole, you test small parts**
- **Not enough on their own**
  - Do not test integration with external systems (e.g. database)
  - Do not test the whole system
  - But: relatively cheap comparing to other test types

Functional Tests

- **Test your functional requirements against running (whole) system**

- **Typically follow your use-cases, scenarios**

- **In simpler cases can serve as integration tests**
  - By testing functionality you test that your integration with external systems works (db, other web-services)

Functional Tests

- **Test scenarios – step-by-step specification of what tester should do**
  - Including input data
  - Used by testers during manual testing
  - → testers do not need to have strong technical skills
  - shoud not be written by developers
    - Developers suffer from detailed knowledge of the system – create tests based on how system works, not how users are going to use it

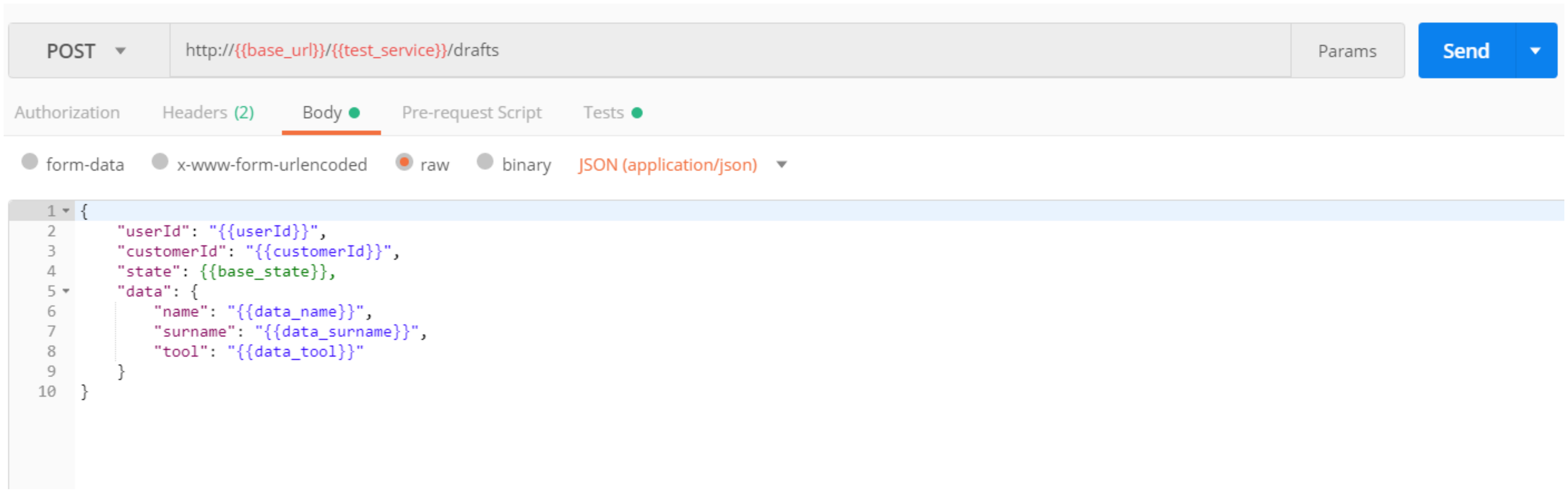
Functional Tests

- **Automated tests**
  - Scripted test scenarios
  - Cheaper in long-term, do not make mistakes
  - But can be expensive to write and maintain
    - User interfaces tend to change a lot

Functional Tests - Automated

- **Easier for web-service API**
  - Standard integration tests – you send a request and get a response
  - Validate response codes, check response body is what you expect
  - Modern WS API specification languages support basic test code generation

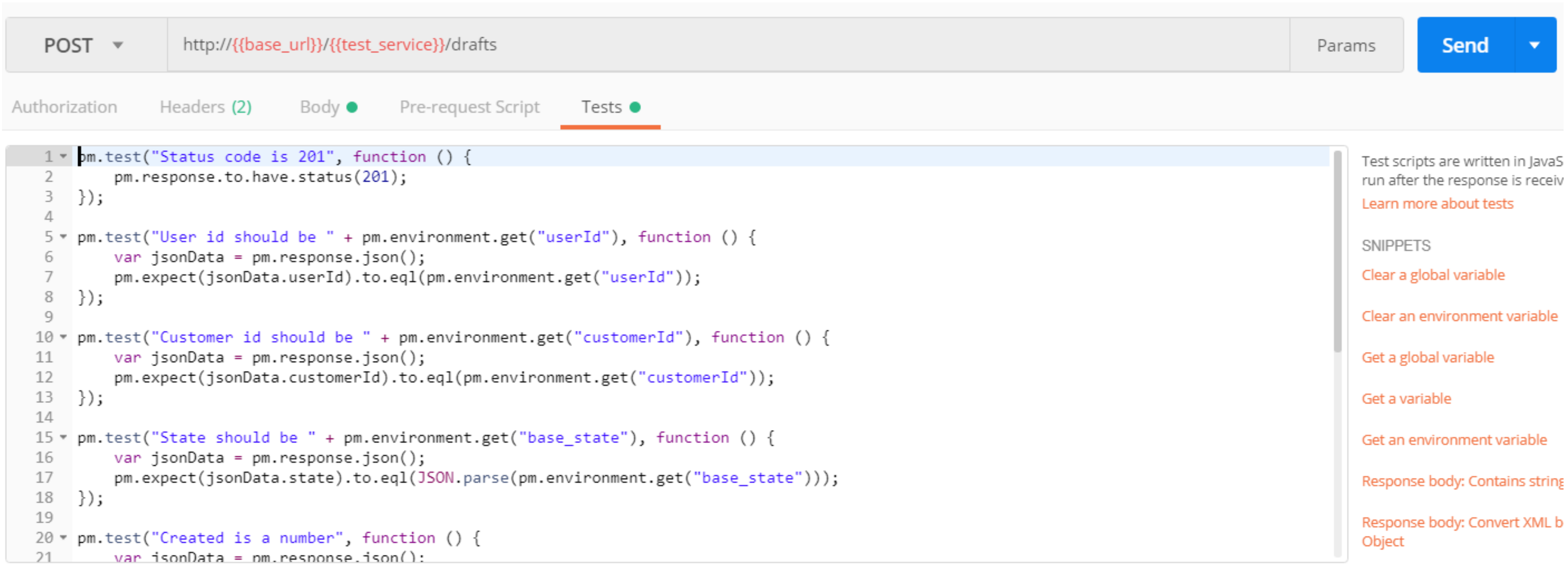
Functional Tests - Automated



http://{{base_url}}/{{test_service}}/drafts

```
{
    "userId": "{{userId}}",
    "customerId": "{{customerId}}",
    "state": {{base_state}},
    "data": {
        "name": "{{data_name}}",
        "surname": "{{data_surname}}",
        "tool": "{{data_tool}}"
    }
}
```

Functional Tests - Automated



```
pm.test("Status code is 201", function () {
    pm.response.to.have.status(201);
});

pm.test("User id should be " + pm.environment.get("userId"), function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.userId).to.eql(pm.environment.get("userId"));
});

pm.test("Customer id should be " + pm.environment.get("customerId"), function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.customerId).to.eql(pm.environment.get("customerId"));
});

pm.test("State should be " + pm.environment.get("base_state"), function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.state).to.eql(JSON.parse(pm.environment.get("base_state")));
});

pm.test("Created is a number", function () {
    var jsonData = pm.response.json();
```

Test scripts are written in JavaS
run after the response is receiv
Learn more about tests

SNIPPETS
Clear a global variable
Clear an environment variable
Get a global variable
Get a variable
Get an environment variable
Response body: Contains string
Response body: Convert XML b
Object

Functional Tests - Automated

- **A bit of challenge for WebUI**
- **Modern testing frameworks know how to simulate browser**
  - They use real browser core’s
  - You can run the same tests against multiple browsers
- **But the application must be written with such testing in mind**

Functional Tests - Automated

- **But the application must be written with such testing in mind**

  - You need to be able to locate HTML element you want to test
    - id, tag attribute, Xpath
  - Xpath tends to change with all sorts of page modifications
  - ID is great, it is unique, but…
    - Frameworks that generate HTML code tend to auto-generate IDs → for each request the IDs differ

Functional Tests - Automated

- **But the application must be written with such testing in mind**
  - Solution: custom attribute
  - When writing our HTML, we add custom attribute uiTest=”unique id” to all elements we need to test
  - Seems like extra work, but in our experience this is the most maintainable way of writing testable html front-end

Functional Tests - Selenium

- **Set of test tools for implementing browser tests**
  - Same test can be run against multiple browsers, ideally without changes

- **Possibility to record tests**
- **Or write them in editor**
- **Or write them in IDE as Java classes**

- **Basically industry standard for WebUI tests**

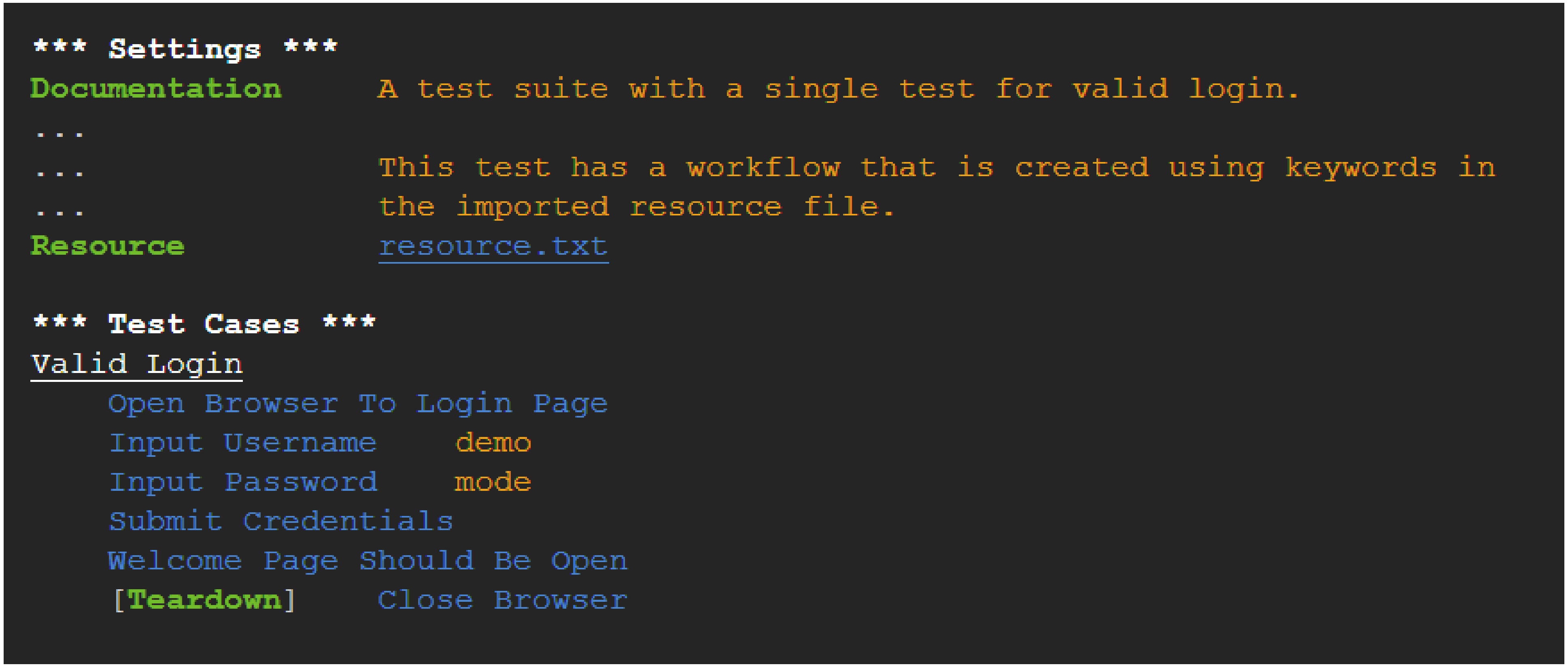
Functional Tests – Selenium Example

```
 @Test
public void testById() {
  driver.get(baseUrl + "Prevodnik");
  WebElement we = driver.findElement(By.id("cisloVstup"));
  assertEquals("", we.getAttribute("value"));
}
```

Functional Tests - Selenium

- Tests are sometimes tedious to write and difficult to maintain
- Yet still a very useful tool to automate UI-based tests
- Keep in mind the application must be written with such testing in mind
- But it can be even better...

Functional Tests – Robot Framework



```
*** Settings ***
Documentation     A test suite with a single test for valid login.
...
...               This test has a workflow that is created using keywords in
...               the imported resource file.
Resource          resource.txt

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser
```

Src: http://robotframework.org

Functional Testing – Robot Framework

Robot framework can be used to automate more than Selenium tests (but we will focus on those here)

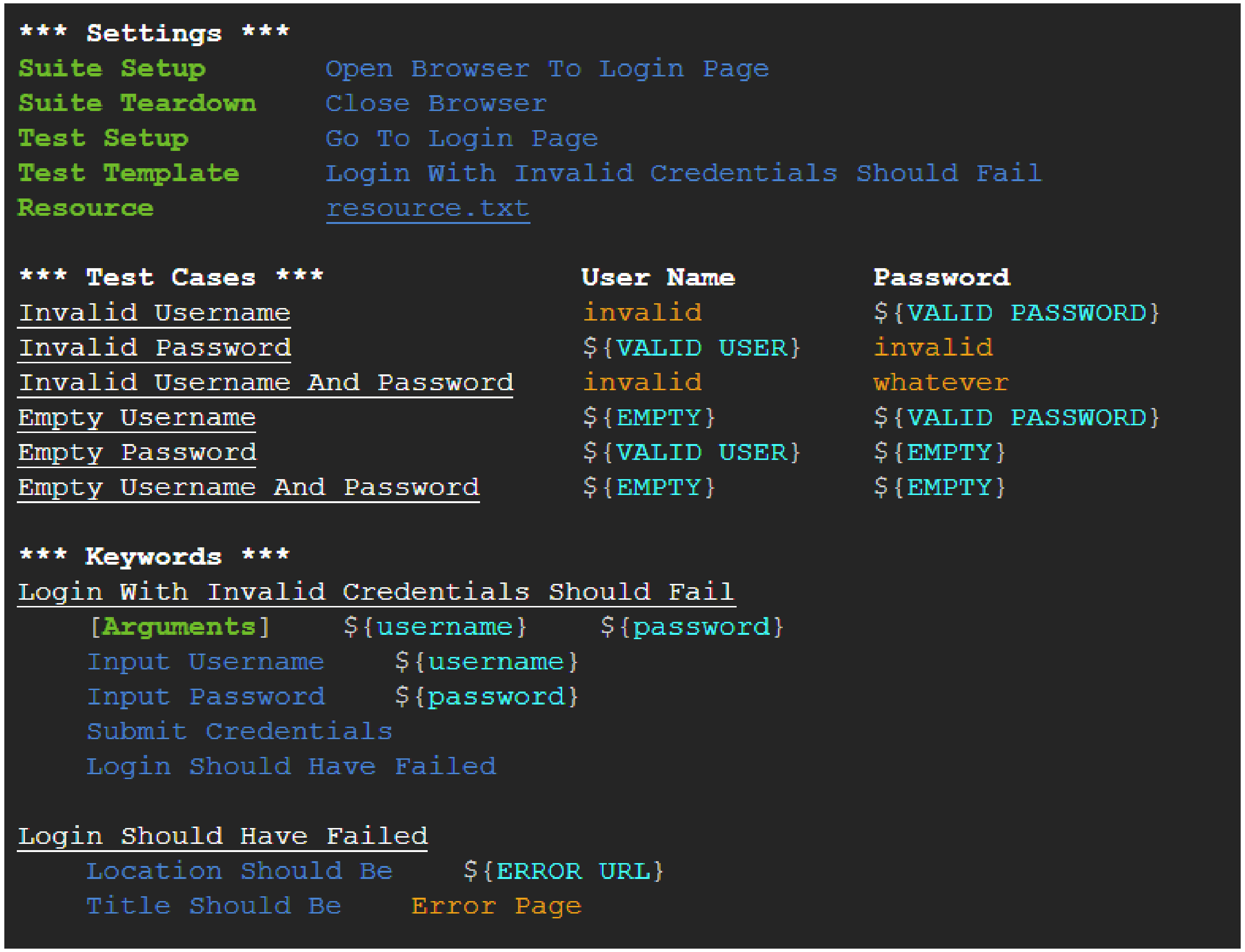
Uses key-words as function calls

→ easy to read by anyone

Key-words are implemented in code (Java, Python), you can even write your own

For WebUI testing, Selenium is the underlying library

Robot Framework – Data Driven



```
*** Settings ***
Suite Setup       Open Browser To Login Page
Suite Teardown    Close Browser
Test Setup        Go To Login Page
Test Template     Login With Invalid Credentials Should Fail
Resource          resource.txt

*** Test Cases ***               User Name        Password
Invalid Username                 invalid          ${VALID PASSWORD}
Invalid Password                 ${VALID USER}    invalid
Invalid Username And Password    invalid          whatever
Empty Username                   ${EMPTY}         ${VALID PASSWORD}
Empty Password                   ${VALID USER}    ${EMPTY}
Empty Username And Password      ${EMPTY}         ${EMPTY}

*** Keywords ***
Login With Invalid Credentials Should Fail
    [Arguments]    ${username}    ${password}
    Input Username    ${username}
    Input Password    ${password}
    Submit Credentials
    Login Should Have Failed

Login Should Have Failed
    Location Should Be    ${ERROR URL}
    Title Should Be    Error Page
```

Src: http://robotframework.org

Other types of tests

- **Integration tests**
  - Verify that connection and message exchange between two systems works as expected
    - Necessary when you integrate two or more systems (e.g. your invoicing application and accounting application)
    - Such integrations are often not triggered by user action, but run periodically → other types of tests wont cover them

Other types of tests

- **Performance tests**
  - Several goals:
    - Ensure application has sufficient performance under real load
    - Check application’s behaviour under high load (e.g. does it recover after it stops responding?)
    - Ensure new versions of application do not introduce performance hit
  - Should be run against production-like environment

Other types of tests

- **Smoke tests**
  - Full functional/integration test suite can take quite some time to run
  - Smoke tests are a basic subset of the whole test suite which ensure basic functionality works
  - Use: ensure new deployment has been successful and the application is responsive
    - Common in production, where you do not want to run other types of tests

Other types of tests

- **Usability tests**
  - Tests how easy is your UI to navigate for users without any prior experience with your system (or introduced changes)
  - You ask a set of users to perform fixed set of tasks
  - Gives you a warning if users are unable to easily perform the task
  - Gives you insight on how users tend to use your system

Debugging

Main techniques

- **Helper prints into console**
  - Easy, but messy
- **System of break-points in IDE/Code debug tools**
  - Great for development debugging
- **Application logs**
  - Necessity for production-like deployment debugging

Code Debugging (in IDE)

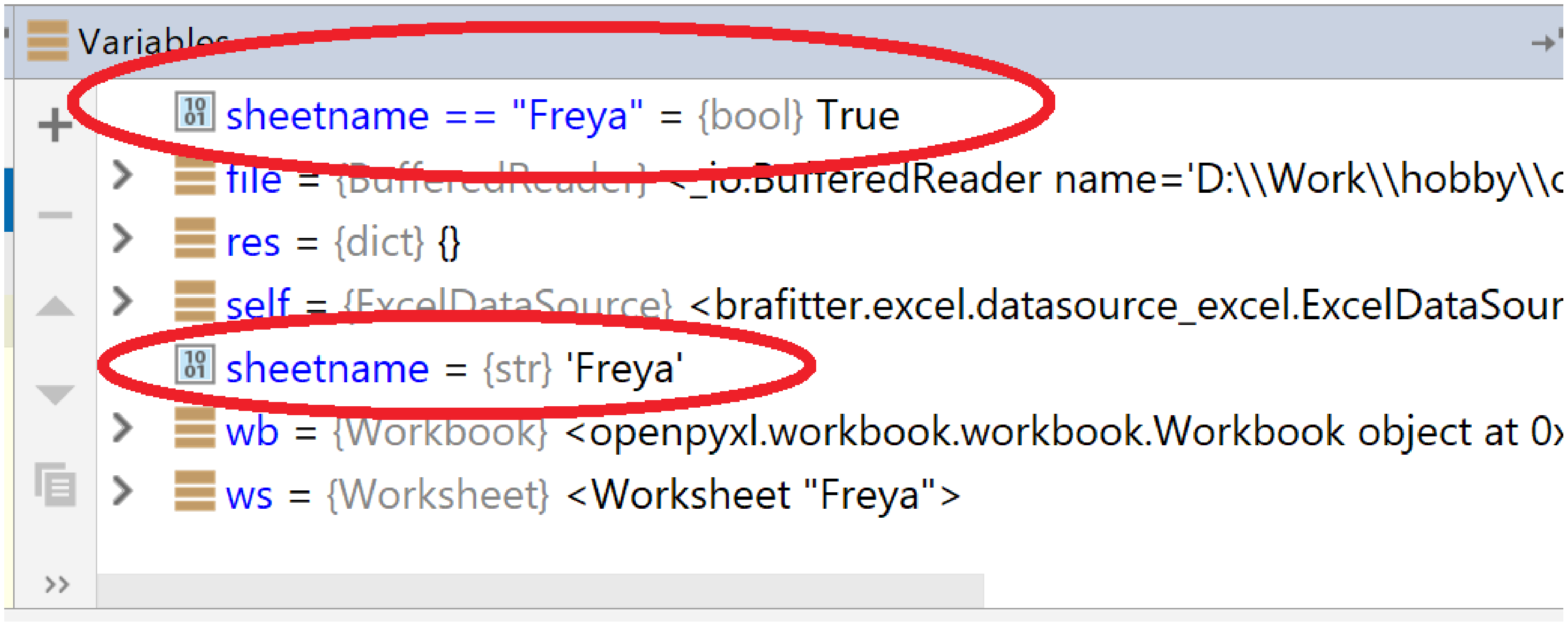
- **Basic Terms**
  - Breakpoint – point in code where you want to pause the execution
  - Watch – a variable or an expression you want watch during debugging
    - Debug tools show you its current value at every step of the execution

Code Debugging (in IDE)

- **Basic Control Flow Actions**
  - Step Over
    - Proceed one step ahead with the execution
  - Step Into
    - Step into a function/method
  - Step Out
    - Step out of current function/method
  - Continue Execution
    - Proceed with execution until next break-point or execution finish

Code Debugging - Useful tips

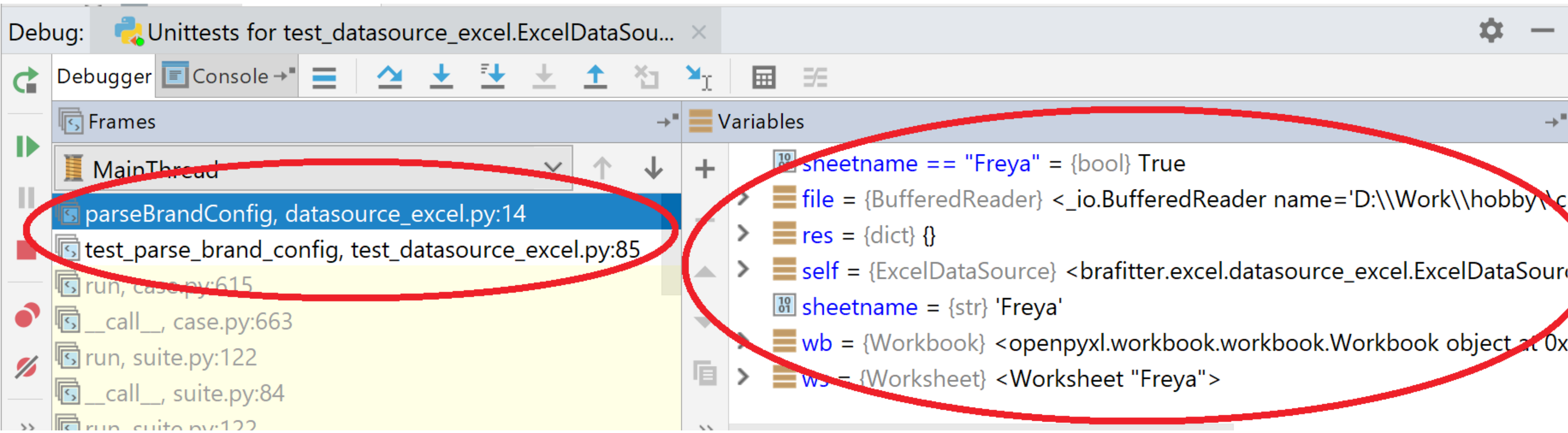
- You can watch not only variables, but also expressions



```
Variables
sheetname == "Freya" = {bool} True
file = {BufferedReader} <_io.BufferedReader name='D:\\Work\\hobby\\c
res = {dict} {}
self = {ExcelDataSource} <brafitter.excel.datasource_excel.ExcelDataSour
sheetname = {str} 'Freya'
wb = {Workbook} <openpyxl.workbook.workbook.Workbook object at 0x
ws = {Worksheet} <Worksheet "Freya">
```

Code Debugging - Useful tips

- **Most debug tools allow you to traverse stacktrace during pause**
  - Lets you check variables outside of your current method´s/function´s scope



Debug: Unittests for test_datasource_excel.ExcelDataSou...

Debugger Console

Frames

MainThread

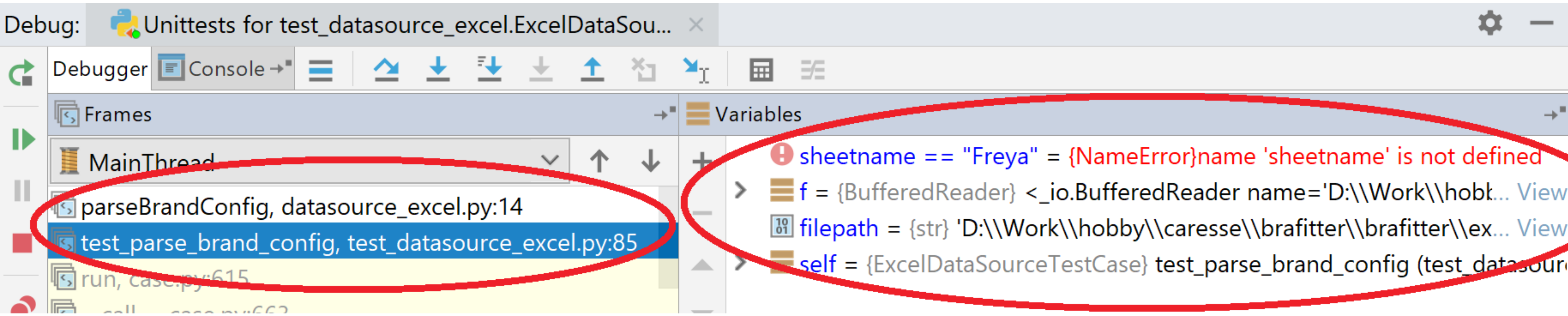
```
parseBrandConfig, datasource_excel.py:14
test_parse_brand_config, test_datasource_excel.py:85
run, case.py:615
__call__, case.py:663
run, suite.py:122
__call__, suite.py:84
run, suite.py:122
```

Variables

```
sheetname == "Freya" = {bool} True
file = {BufferedReader} <_io.BufferedReader name='D:\\Work\\hobby\\c
res = {dict} {}
self = {ExcelDataSource} <brafitter.excel.datasource_excel.ExcelDataSour
sheetname = {str} 'Freya'
wb = {Workbook} <openpyxl.workbook.workbook.Workbook object at 0x
ws = {Worksheet} <Worksheet "Freya">
```

Code Debugging - Useful tips

- **Most debug tools allow you to traverse stacktrace during pause**
  - Lets you check variables outside of your current method´s/function´s scope



```
parseBrandConfig, datasource_excel.py:14
test_parse_brand_config, test_datasource_excel.py:85
run, case.py:615
```

```
sheetname == "Freya" = {NameError}name 'sheetname' is not defined
f = {BufferedReader} <_io.BufferedReader name='D:\\Work\\hobl... View
filepath = {str} 'D:\\Work\\hobby\\caresse\\brafitter\\brafitter\\ex... View
self = {ExcelDataSourceTestCase} test_parse_brand_config (test_datasour
```

Code Debugging - Useful tips

- **Debugger shows type/structure information**
  - Great help in dynamically typed/untyped languages
- **You can also debug remote servers**
  - Must be explicitly enabled when deploying
  - Security flaw → use only in controlled environments
- **All major browsers contain JavaScript debugger**

Code Debugging - Summary

- **Great tool during development**
  - Use it!
- **Security flaw during deployments**
  - Do not use it!

Debugging - Deployment

- **Finding bugs in production is difficult**
  - Can’t break service
  - Can’t damage data
  - Can’t attach debugger

- **But bugs still find their way into production**
  - Despite our good test coverage
  - → good logs help a lot

Types of Logs

- Application logs – messages describing what is going on in the application

```
08:58:03.021 [http-nio-8080-exec-1] INFO  c.y.p.h.c.s.impl.DefaultDraftService -
Draft metadata successfully saved.
```

- Access logs – all calls made to the application from the outside world (HTTP requests in our case most of the time)

```
127.0.0.1 - - [14/Lis/2018:08:58:03 +0100] "POST /tt/drafts HTTP/1.1" 201 163
```

- Audit logs – tracing what particular user is doing

Application logs

Multiple levels:

- Info – each successfully finished action should be logged
- Warn – Application did not end-up in error state, but something is not as it should be and we should be notified of it
- Error – All error situations must be logged
- Debug – detailed information of what is going on in the applicaton – messages include data structures etc. Imagine you do not have a debugger.
- Trace – should describe every step within app (start and end of method call should be logged, all conditional executions, etc.)

Access Logs

- **All http requests logs, including:**
  - RequestId, SessionId
  - Request hostname and path
  - Request parameters
  - Request headers
  - Cookies

Audit Logs

- **When any action is executed (read or update, does not matter)**
  - What action
  - Who (which user)
  - When
  - Possible context, if we know it

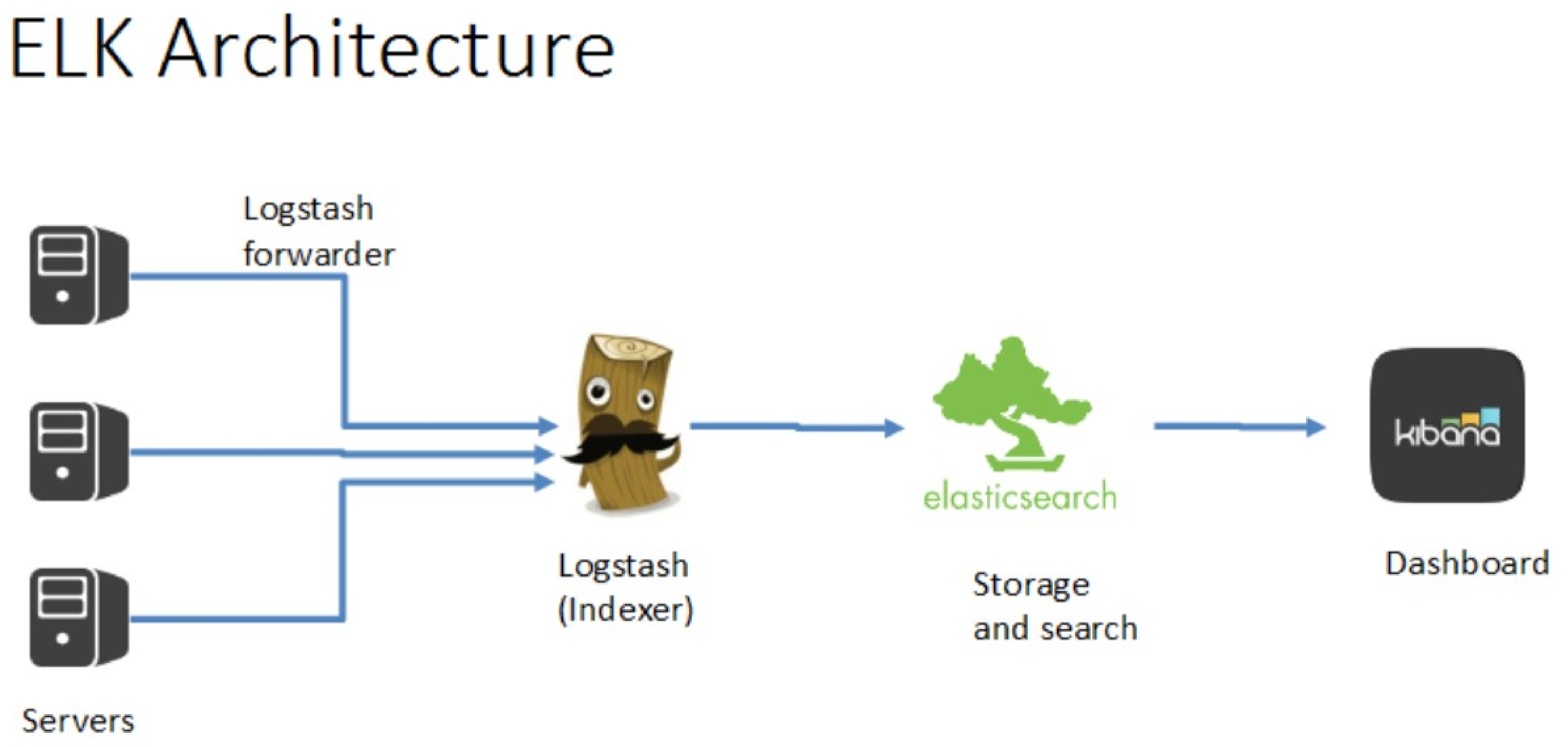
Implementation

- **All major languages have mature logging support (core or via libraries) with out-of-the-box support for**
  - Formatting
  - Message Levels
  - Context information
  - Filtering by source
  - Various storage options
    - Console, file, web-service, message queue, database

Modern Logging

- **Cloud storage for logs with processing(logstash) indexing (elasticsearch) and good presentation (kibana)**
  - Same logging libraries, but logs sent to cloud via message queue
  - Easy to aggregate all your logs in one place
  - Bad for audit logs – does not guarantee delivery
- **Audit logs can be stored in database**
  - And cloud storage can then process and index logs from the database to provide advanced search functions

Modern Logging



Src: http://bingoarun.github.io/openstack-log-analysis-elk.html

Logs

- Each application should have them
- Necessary when you want to know what happened in production
- With proper use of log levels do not present a performance issue for application