YAF Users Guid

1. Source codes

YAF Core: {sourcedir}core/src/main/

YAF Modules: {sourceCode}modules/

2. Overview

The goal of this document is to provide comprehensive reference documentation for qa engineers and module authors.

This document is also available as a PDF download.

2.1. What is YAF

YAF - Yet Another Framework, is a modular test automation framework that helps QA engineers to solve common problems in automation and ...

2.2. Requirements

The only requirement is Java 1.8+

2.3. Navigation

2.3.1. Common

1. Get in-depth information about how the framework works.
2. Listen to a wide variety of events about the status of your tests.
3. Get information about your test meta info from test execution context.
4. Use helpful utilities.

2.3.2. WEB

1. Get started with First Web Test guide
2. Enreach your tests learning how to implement automation best practices using YAF using First Web Test Extended guide
3. Extending Page objects with conditional pages.
4. Test data parametrization via POJO classes with data manager.
5. Create advanced execution configuration to your tests.

2.3.3. API

1. Add API support (including RestAssured).
2. Using YAF data providers.

2.4. Getting Help

Email us ....

2.5. Sample projects

1. Project 1
2. Project 2
3. Project 3

3. Writing first web application test

This guide will introduce you to the process of writing your first web application test.

1. Before implementing test case it is required to configure the build-tool. Both pom.xml(Maven) and build.gradle(Gradle) files content parts are present there. For quick start TestNg runner is used.
    a. Add external repository to your `pom.xml`:

▼ *Maven*

```
    <repositories>
        <repository>
            <repository>
                <id>central</id>
                <url>https://repo.maven.apache.org/maven2</url>
            </repository>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
            <id>coherent-yaf</id>
            <name>yaf-libs-release</name>
            <url>https://coherentsolutions.jfrog.io/artifactory/yaf-
java</url>
        </repository>
    </repositories>
```

▼ *Gradle*

```
    repositories {
        mavenCentral()
```

```
        maven {
            url = uri('https://coherentsolutions.jfrog.io/artifactory/yaf-
java')
        }
    }
```

b. Add dependencies. For current guide purposes following yaf modules are used: **core**(common framework functionality), **yaf_testng** and **yaf-web**.

▼ *Maven*

```
    <properties>
        <!-- other properties -->
        <yaf.version>1.3.4</yaf.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.coherentsolutions.yaf</groupId>
            <artifactId>core</artifactId>
            <version>${yaf.version}</version>
        </dependency>

        <dependency>
            <groupId>com.coherentsolutions.yaf</groupId>
            <artifactId>yaf-testng</artifactId>
            <version>${yaf.version}</version>
        </dependency>

        <dependency>
            <groupId>com.coherentsolutions.yaf</groupId>
            <artifactId>yaf-web</artifactId>
            <version>${yaf.version}</version>
        </dependency>
    </dependencies>
```

▼ *Gradle*

```
ext {
    yaf_version = "1.1.1"
}

dependencies {
    implementation("com.coherentsolutions.yaf:core:${yaf.version}")
    implementation("com.coherentsolutions.yaf:yaf-testng:${yaf.version}")
    implementation("com.coherentsolutions.yaf:yaf-web:${yaf_version}")
}
```

TIP ${yaf.version} means framework version is stored in variable and reused. It is

recommended to handy and transparent initial configuration and migration to new version

2. Create your first test in proper tests folder.

```
public class FirstTest extends YafTestNgTest { ①

    @Test
    @YafTest②
    public void sampleTest(){
        WebDriver driver = (WebDriver) getWebDriver().getDriver();
        driver.get("https://www.coherentsolutions.com/");
        // standard element interaction goes here
        // driver.findElement(...
    }
}
```

IMPORTANT It is required for test class to extend `YafTestNgTest`<1> and methods to be marked by YafTest<2> annotation to use yaf features.

3. YAF is Spring based framework. And although there is no need to add any spring dependencies to you project, you still suggested to follow spring style of development. For current example it is obligated to define a source of bean definitions class:

IMPORTANT Application configuration file must be placed in the project root folder

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(basePackages = {"com.your.base.package"}) ①
public class FirstConfig extends YafConfig { ②
    public FirstConfig(ApplicationArguments args) {
        super(args);
    }
}
```

NOTE ComponentScan annotation define a package to scan for components from, no values provided means current package will be chosen. These three nnotaions could be replaced by SpringBootApplication annnotation

4. Execute this test using one of described ways

NOTE No additional configuration, default browser will be the latest `Chrome`. For more information about configuration abilities read Yaf Confg Manager

Described steps should be enough to implement the simplest web application test case using YAF. The following recommended step is to extend existing test. See here

include::subdir/firstWebTestExtended.adoc

4. YAF Core features

4.1. Drivers

In Yaf Selenium WebDriver is represented by `DriverHolder`. In its turn they are stored inside DriversStore under corresponding Yaf Device key. DriverHolder contains additional driver information that helps operate drivers. One on the usefull driver's characteristic is scope. There are four of these:

1. EXECUTION - for common/default driver, only one per execution exist. Used in case no configurations is used
2. SUITE - for suite specific driver, stored in suit name - driverHolder map, is there is no driver for corresponding suite EXECUTION scope driver is used
3. CLASS - is assigned to the driver in case parallelization by classes
4. METHOD - is assigned to the driver in case parallelization by medhods

4.2. Context

In addition to the standard Spring context, there are two extra custom contexts in YAF.

The context's tasks are:

1. To store all information about the test execution process;
2. To be the basis for decision-making about the type of object that needs to be injected.

The framework has two custom contexts:

1. `ExecutionContext` - a global test execution context.
2. `TestExecutionContext` - a local test execution context.

4.2.1. ExecutionContext

ExecutionContext contains global information and launch parameters, also manages drivers with **Execution** and **Suite** scopes.

This context is created when the application starts and is cleared after the application quits.

4.2.2. TestExecutionContext

TestExecutionContext manages drivers with **Test** and **Class** scopes, and contains information about the current test, such as:

- test name;
- suite name;

- current environment;
- start time
- current additional config
- a `TestInfo` object with a lot of additional information about the test.

This context is created when the test starts (TestStartEvent) and is cleared when the test ends.

4.2.3. How is it used

These contexts are accessible inside BaseYafTest, Component and Utils classes and its inheritor classes Contexts are participate in the one of the first command trying to start your first test (link:firstWebTest.adoc):

```
public class FirstTest extends YafTestNgTest { ①

    @Test
    @YafTest
    public void sampleTest(){
        WebDriver driver = (WebDriver) getWebDriver().getDriver(); ①
        WebDriver firefoxDriver = (WebDriver) getWebDriver("ff").getDriver(); ②
    }
}
```

NOTE	<1> getWebDriver()/getMobileDriver()/getDesktopDriver() of BaseYafTest return a Driver Holder instance of corresponding type Web/Mobile/Desktop. It applies to testExecutionContext and then to executionContext driver store for the required driver, created new and add it to testContext if existing one was not found. In <2> case to type filter device name(one of devices that are defined for current environment, see) will be added

By YAF getDriver methods are used during pages and components class files initializing to assign to each of them a driverHolder for further usage orchestrating app's pages and components

4.3. Config Manager

Executing tests with complex configurations can be quite a challenge. For example, you need to run tests against/on multiple browsers, mobile devices, different languages, etc. Such a task requires a lot of effort and sometimes can be confusing. YAF config manager will help to cope with such confusing cases.

The main objects of the config manager:

- Device - device on which the tests are executed, it can be a web browser, a mobile device, a desktop;
- Configuration - a typed object that represents an additional configuration of the execution, such as locale or user for instance;

- Environment - a set of devices and configurations for single test execution, for example, you need to test something in a browser and on a mobile device, with a specific locale and user;
- Execution - the set of environments where you need to execute the tests and parallel settings of the execution;
- Farm - a farm of devices, cloud provider, ... - provider of devices for tests execution.

IMPORTANT	The configuration manager supports many sources of configurations, in the basic package it reads configuration from JSON files. These readers will also be used in the examples.

4.3.1. How to

How to use

To use the configuration manager, you should:

1. Import yaf-exec-config-json dependency to your pom.xml/build.gradle

▼ *Maven*

```
        <dependency>
            <groupId>com.coherentsolutions.yaf</groupId>
            <artifactId>yaf-exec-config-json</artifactId>
            <version>${yaf.version}</version>
        </dependency>
```

▼ *Gradle*

```
dependencies {
    implementation("com.coherentsolutions.yaf:yaf-exec-config-
json:${yaf.version}")
}
```

2. In the folder with test resources create a directory structure

```
- execute
  |- config
  |- data
  |- device
  |- env
  |- farm
  |- xxx_config_file.json
```

3. For base configuration at least 2 files should be created

NOTE	configuration manager supports both JSON files and JSON5

*Device File**

File with a random name in the directory `device`

```
{
  "chrome": {
    "type": "web",
    "browser": "chrome",
    "args": [
      "--start-maximized",
      "--disable-browser-side-navigation",
    ],
    "capabilities": {
      "pageLoadStrategy": "normal"
    }
  },
  "ff": {
    "type": "web",
    "browser": "ff",
    "args": [
      "--start-maximized",
      "--disable-browser-side-navigation"
    ],
    "capabilities": {
      "pageLoadStrategy": "normal"
    }
  }
}
```

For specific device syntax see BrowserDevice, MobileDevice, ...

General configuration File

Is a file in execute derictory. In this file, you should to fill in some sections (* - required):

- name* - configuration name;
- envs* - in this block you should specify the list of environments and determine which devices will belong to the particular environment;
- envsNames - if you want to put the configuration of the environments into separate files for further reuse, specify the names of the environments in this block (the files should be placed in the **env** folder);
- configs - if you want to additionally parameterize your run (not only by devices) and specify, for example, a list of user roles, you should enter them in this section as a key/value;
- configNames - the same logic as with the list of environments: you specify the names of configurations (files must be placed in the **config** folder);
- farmNames - if your configuration implies tests execution on farms, you should specify them in this list.
- setup- if you want to decrease tests run execution time use setup block to configure parallel

execution.

```
{
  "name": "smoke",
  "envs": {
    "set1": {
      "devices": [
        "chrome",
        "ff"
      ]
    },
    "set2": {
      "devices": [
        "chrome",
      ]
    }
  }
}
```

If you desire to reuse your env define a file with a random name in the directory `env`

```
{
  "env1": {
    "devices": [
      "chrome",
      "ff"
    ]
  },
  "env2": {
    "devices": [
      "chrome"
    ]
  }
}
```

Then your general configuration file will look like this

```
{
  "name": "smoke",
  "envsNames": [
    "env1",
    "env2"
  ]
}
```

4. Finally define your configuration file name as environment variable with envSettings key running your tests. Otherwise, default_env (chrome, latest version) is used. For example, envSettings=f_config_file

Additional config

Providing extra configuration

```
{
  "name": "smoke",
  "configs": [
    {
      "user": "admin",
      "locale": "ru"
    },
    {
      "user": "admin",
      "locale": "eng"
    }
  ],
  "envsNames": [
    "env1",
    "env2"
  ]
}
```

NOTE	User and locale current values could be accessible in TestExecutionContext's params. For current example it will store DefaultYafUser with 'admin' username and password under 'user' key and java Locale object with provided language under 'locale' key. To see details of implementation and to create other config like user and locale see

If you desire to reuse config the same process as in case of environments works:

File in `configs` folder:

```
{
  "ruUser": {
    "user": "user1",
    "locale": "ru"
  },
  "enUser": {
    "user": "user1",
    "locale": "en"
  }
}
```

And general config file:

```
{
  "name": "smoke",
  "configNames": [
```

```
    "ruUser",
    "enUser"
  ],
  "envsNames": [
    "env1",
    "env2"
  ]
}
```

Farms configuration file

```
{
  bs: {
    url: "http://",
    user: "xxx",
    key: "yyyy",
    supportedTypes: [
      "web"
    ]
  },
  local: {
    supportedTypes: [
      "web",
      "mobile"
    ]
  }
}
```

IMPORTANT	Farms are matched to devices by type, i.e. if a farm is configured as a web farm, all web devices will run on this farm. At the same time if there are 2 farms for a mobile device, but one of them is purely mobile, and the other is both mobile and web, then purely mobile farm will be used by priority.

Parallelization config

Parallelization config consists of TestNG suite attributes like parallel modes and threadCount. .suiteThreadCount is used if several environments are defined and multiple suites is about to be run. Define it to run that suits in parallel.

```
{
  "name": "smoke",
  "envsNames": [
    "env1", "env2"
  ],
  "setup": {
    "suiteThreadsCount": 1,
    "threadsCount": 1,
    "parallelMode": "CLASSES"
  }
```

```
}
```

How it works

The upper-level process goes as follows:

1. When tests are invoked, the defined configuration General configuration File is read and an execution configuration `ExecutionConfiguration` is generated.

   Then, in `ExecutionConfiguration` an array of runs of all environments against all configurations is generated. For example, we have 2 users (administrator, manager) and 2 browsers (Chrome and FF), as a result of this formation, tests will be executed 4 times:

   - Chrome Admin
   - Chrome Manager
   - FF Admin
   - FF Manager

     And this principle can be replicated for any type of configuration.

2. The original test suite is modified to fit the configuration (more in TestNG)
3. When running a particular test, based on the configuration in step 2, we get an `Environment` object, which contains a list of devices to run, as well as a set of configurations. Then, we go through the list of configurations, find a corresponding resolver (see next section) and apply this configuration to the test execution context.

How to extend

To add additional configuration (similar to users and locales), create an implementation of the `ConfigurationResolver` class and in the `applyConfiguration` method implement the necessary usage of the configuration.

For example, as implemented in `LocaleConfigurationResolver`

```
@Service
public class LocaleConfigurationResolver extends ConfigurationResolver {

    @Override
    public String getType() {
        return "locale";
    }

    @Override
    public void applyConfiguration(String value, TestExecutionContext
testExecutionContext) {
        try {
            testExecutionContext.addPram(Consts.CTX_LOCALE, new Locale(value));
        } catch (Exception e) {
```

```
            log.error("Unable to set locale " + value, e);
        }
    }
}
```

If your resolver requires reading additional data as user is, you need to implement class `ConfigurationDataReader`.

This implementation is used to transform loaded data into strictly typed data. Due to the fact that the reading of this configuration occurs before the initialization of the Spring context, another mechanism is used to provide dynamic resolvers behavior, and you need to register the created resolver using the mechanism SPI

Here is an example of how this mechanism works by example. (This is how `UserConfigurationResolver` works).

In the configuration files we use only aliases, for example:

```
 "user": "admin",
```

However, additional data is needed to build a complete full-featured user object. This data is stored in the `execute\data\%resolverType%` directory (in case of using json file storage for configuration), it is read by the basic configuration reader and then a transformation method is applied to it from the corresponding data reader, and it transforms Map<String, Object> into an object with typed values.

Thus, the `UserDataReader` may look like this:

```
public class UserDataReader extends JsonConfigurationDataReader<DefaultYafUser> {
    @Override
    public String getType() {
        return "user";
    }

    @Override
    public Map<String, DefaultYafUser> transformData(Map<String, Object> dataMap) {
        return mapper.convertValue(dataMap, new TypeReference<Map<String,
DefaultYafUser>>() {
        });
    }
}
```

As conversion to DefaultYafUser is used json data must have the same structure as the class:

```
{
  "admin": {
    "username": "admin1",
    "password": "qwRNTdh"
```

```
  }
}
```

4.3.2. Config reading

The framework allows you to store configurations in various forms, and for this it is necessary to implement the `BaseExecutionReader`.Basically, the framework implements reading from JSON. (!TBD link)

4.4. Data Management

To facilitate working with structured data, the framework provides a special mechanism that allows you to inject POJO objects into the test depending on the execution context.

The basic implementation is based on storing data files in JSON format, but it can be reconfigured.

4.4.1. How to

How to use

1. First of all data is a separate YAF module that isn't accessible by default and additional dependence are required to be added

   ▼ *Maven*

   ```
   <dependency>
       <groupId>com.coherentsolutions.yaf</groupId>
       <artifactId>yaf-data</artifactId>
       <version>${yaf.version}</version>
   </dependency>
   ```

   ▼ *Gradle*

   ```
   dependencies {
       implementation("com.coherentsolutions.yaf:yaf-data:${yaf.version}")
   }
   ```

2. Configuration (Optional)

   If you want to use a different data format than JSON, or use a different folder, specify this in the `yaf.data⋯.` properties in YAF configuration file.

3. Create a json file in `data` folder resource or other if `dataFolder` property was defined on step 1
4. In the test class, declare a POJO object for the required data.
5. Annotate it with `@YafData` specifying the required file name. After creating an instance of the test class, this object will be filled with data from the specified file(step 2).

IMPORTANT	Please note that the initialization (filling) of the data object happens once when creating an instance of the test class, if you need to have a new data generated for each test - define `alwaysNew` YafData property to true.

NOTE	All data that comes into this mechanism is processed by the template engine (MustacheTemplateService) and the localizer (L10nService).

Faker

To generate unique data Java Faker could be used. To write dynamic data files the first thing to know is the calculated parts are inside double curly braces. There are some Faker usage cases in the following example, more could be found here Java Faker

```
{
  "website": "https://{{faker.internet.url}}",
  "domain": "{{faker.internet.domainName}}",
  "firstName": "{{faker.name.firstName}}",
  "lastName": "{{faker.name.lastName}}",
  "jobTitle": "{{faker.job.position}}",
  "isActive": "{{faker.bool}}",
  "companyName": "{{faker.company.name}}",
  "startDate": "{{faker.date.past(100, java.util.concurrent.TimeUnit.DAYS)}}"
}
```

Let's pay attention at `startDate` property. After this file processed `startDate` POJO class's property will happen to stay as it was. To fix the problem faker's methods that expected to get at least one argument should be used in a little bit another way:

```
@Component
public class CustomFaker implements CustomTemplateFunctions {

    public String past() {
        return String.valueOf(new Faker().date().past(100, TimeUnit.DAYS));
    }

    @Override
    public String getKeyName() { ①
        return "date";
    }
}
```

NOTE	<1> The method returns the prefix name to use it to apply to new CustomTemplateFunctions methods. Here could be oe or more methods implementing custom functionality that extends Faker's one and wrapping Faker's method that expects arguments to be passed

In json file it will look like this:

```
{
...
  "startDate": "{{date.past()}}"
}
```

How to extend

This mechanism can be extended by implementing your own data reader, for example, XmlDataReader or ApiDataReader. You can see the implementation of the reader in the example of `JsonDataReader` (!TBD link). And it is also necessary to create an additional annotation that "overrides" `@YafData` and specify the created reader there.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE })
@Inherited
@Documented
@YafData(reader = ApiDataReader.class)
public @interface ApiYafData {
    String url();
}
```

NOTE "overrides" here means mark "child" ApiYafData annotation by "parent" YafData defining custom Reader class as reader attribute or other attributes if necessary. Also, here is a custom data annotation attribute url installed of YafData's fileName as source of data here is a api resource, not file

```
@Component ①
public class ApiDataReader extends BaseFileReader { ②

    @Autowired
    RestAssuredYafRequest request;

    @Override
    public Object readData(YafData yafData, Class<?> objType, TestExecutionContext
executionContext) {
        String url = objType.getAnnotation(ApiYafData.class).url(); ③
        return request.anonymousReq("GET", url).jsonPath().getObject(objType);
    }
}
```

NOTE Any reader is required to extend BaseFileReader class <2> and be a spring component <1>. readData signature includes YafData annotation and ApiYafData custom attributes are not accessible. To solve this problem reflection under objType(the object type the data annotation was applied to) is used in <3> line.

TIP	BaseFileReader class contains templates processing methods

How it works

The implementation of this approach is based on the mechanism of post-processing of the fields: On test is starting YAF checks all the field to be processable by any of registered processors or post-processors. If suitable processor is found the field and/or its value will be handled and changed

The `DataFieldPostProcessor` is used to process data fields. In case the field is annotated with a reader, it is used to read the data and initialize the field the following way:

1. Processor get a reader from the data annotation and execute it if necessary
2. Reader get data(json data for JsonDataReader) by the file name from file and convert the data on his way
3. The file name is searched in the following places (in descending order of priority):
    a. In the annotation `@YafData.fileName'
    b. In the annotation `@YafTest.dataFile` .
    c. In the annotation `@YafTest.testName` .
    d. Using the test method name
4. The data is also processed by template service (mustache) applying Faker and CustomTemplateFunctions

4.5. API

To simplify the work with API requests, several items have been implemented in YAF:

1. Special abstraction level of API client.
2. Authorization provider mechanism.

4.5.1. API Client abstraction level

A special abstraction level of API client allows replacing API client implementation. This level operates such classes as `YafRequest`, `AuthProvider`, `YafApiUser`, thereby allowing the user to override any of the components on this level.

The basic `YafRequest` contract has a support for 4 main HTTP methods, both anonymous and authorized.

4.5.2. Implementation

YAF provide a RestAssured implementation (the guide RestAssured sample). To write own implementation (Java HTTP Client or other framework) it is required to implement your own YafRequest

How to use

1. Implement YafRequest. YafRequest is a generic type so two types should be defined implementing it. T type is a type representing request as configuration before it is sent, R is a type represents a response (should be mutable, so it will be called as request config further). The following YafRequest interface methods are required to implement defining you own api implementation:
    - T anonymousReq() - get default request config without auth applied
    - T req() - get request config with default user auth applied. the source of default user could be taken from config, properties or constructor/builder
    - T req(YafApiUser user) - get request config with provided user auth applied. YafApiUser here is an interface and before using this method in tests it is useful to implement it as a POJO with required for auth info (usually login/password)
    - YafRequest addHeader(String key, String value)
    - YafRequest queryParam(String key, String value)
    - YafRequest queryParams(Map params)
    - ApiProperties getProps() - return api properties
    - R anonymousReq(String method, String url, Object... body) - execute request with no auth applied. Do not forget to use headers and queryParams set before send the request (valid for every req method)
    - R req(String method, String url, Object... body) - execute request with default user auth applied
    - R req(YafApiUser user, String method, String url, Object... body) - execute request with provided user auth applied

    For following examples java.net.http is used:

```
public class HttpYafRequest implements YafRequest<HttpRequest.Builder,
HttpResponse<String>> {

    AuthProvider authProvider;①

    @Autowired
    ApiProperties properties;②

    // builder or constructor

    // abstract methods implementation

    @PostConstruct ③
    public void init() {
        reqSpec.uri(new URI(properties.getBaseUrl()));
        //...
    }
```

```
}
```

IMPORTANT	<1> To perform request required authorization using user YafApiUser AuthProvider is used. On the YafRequest initialization authProvider entity must be wierd with its yaf request entity using setRequest method of AuthProvider to set authorization headers to request config when your yaf request implementation suggest to make authorized requests to api. That also means one thing waste to remember setting provider entity from outside (for example through constructor/builder): new provider entity should be used for each yaf request entity.
IMPORTANT	<2> To read api properties from application properties ApiProperties or child type field is required. The main nuance the Autowired annotation is used to organize a Spring easy access to properties defined in properties files via using @ConfigurationProperties on a @Bean Method defining a YafRequest bean. Pay attention that these properties are set after constructor invocation, so in case of using properties for initialization(for ex, set some common or base url/header/queryParams, logging) it is recommended to perform it inside bean @PostConstruct method or afterPropertiesSet method of InitializingBean interface <3>

2. AuthProvider

   The main role of AuthProvider is management of token-user pair, execution of authorization requests, tracking the token's lifetime and "signing" the requests.

   How does it work? There are three stages AuthProvider is responsible for. auth method aimed to manage tokens: provide actual token if exists for the user or init the request to api for the new one (getAuthToken method) and finally trigger apply auth stage (applyAuth method) where token is applied on request config in defined way. Creating your own AuthProvider you are obligated to provide your own getAuthToken and applyAuth methods.

```
public class HttpAuthProvider extends AuthProvider<HttpRequest.Builder> {

    @Override
    protected AuthToken getAuthToken(YafApiUser user) throws YafApiRequestException
{
        HttpResponse<String> response = ((HttpResponse<String>)
request.anonymousReq(
                    "post", "/api/auth/password", user));
        if(response.statusCode() != 200) {
            throw new YafApiRequestException("Faild to get token: status: " +
response.statusCode());
        }
        String userToken = response.body();
        //convert token to AuthToken
```

```
        AuthToken token = new AuthToken();
        token.setToken(userToken);
        return token;
    }

    @Override
    protected void applyAuth(AuthToken token, HttpRequest.Builder req) {
        req.header(AUTHORIZATION, BEARER + " " + token.getToken());
    }
}
```

Also, in your config class define a bean for each API

```
    @Bean("one")
    @Primary
    @ConfigurationProperties(prefix="yaf.api.first-api")
    public HttpYafRequest one() { ①
        //...
    }

    @Bean("two")
    @ConfigurationProperties(prefix="yaf.api.second-api")
    public HttpYafRequest two() {
        //...
    }
```

To use this requests define dependence in your test class

```
//for first API
    @Autowired
    YafRequest yafRequest;

//or
    @Autowired
    YafRequest one;

//or
    @Autowired
    @Qualifier("one")
    YafRequest one;

///------------------

//for second API

    @Autowired
    YafRequest two;

//or
```

```
    @Autowired
    @Qualifier("two")
    YafRequest two;
```

3. In your `application.properties` file add props for several APIs like below (using additional name after `api`, unique for each API). The example shows how to define more than one api

```
yaf.api.first-api.props.baseUrl=https://xxxx
yaf.api.second-api.props.baseUrl=https://yyyy
①
yaf.api.first-api.props.{apiPropertiesClassFieldName}={value}
yaf.api.second-api.props.baseUrl.{apiPropertiesClassMapFieldName}.key=value
```

> **NOTE** <1> You can specify all options you need from ApiProperties class.

4. Extending APIs properties (optional)

   Create class `XXXProperties` and extend it from `ApiProperties` class, adding your custom properties as fields. Mark this class with `@Primary` annotation.

```
@Primary
@Data
@EqualsAndHashCode(callSuper = true)
@Configuration
public class SApiProperties extends ApiProperties {

    String extKey;
}
```

Use it in your YafRequest class as it showed above just changing properties field type. As well, it is possible to use in AuthProvider or somewhere else with `cast`:

```
public class SAuthProvider extends AuthProvider {

    @Override
    protected AuthToken getAuthToken(ApiUser user) throws YafApiRequestException {
        SApiProperties properties = (SApiProperties) request.getProps();
        properties.getExtKey();

    }
}
```

4.6. Conditions

To provide a flexible test configuration with the necessary beans, a special mechanism was

designed. If you mark a certain bean with a specialized annotation, the necessary implementation will be selected at the moment of its injection, depending on the test execution context.

For example, you have several implementations of a page (PageObject) depending on the screen resolution. In the test class, the basic implementation of this page is declared, different inheritors ("child-objects") are labeled with special annotations. When a page is initialized, depending on the conditions described in the annotations and the test context, a necessary implementation will be injected.

4.6.1. How to

How to use

In specific framework modules, there is a standard set of conditional annotations, such as `@Android`, `@Chrome`, and so on.

To use them, you need to:

1. Create an abstract core page (or any other yaf bean), such as `LoginPage`, which is inherited from `Page` or `Component` or `Utils` or any of their inheritors.
2. Create its inheritors, which will define the conditions of their creation.
3. Annotate the inheritors with necessary annotations.

You can also use core annotation `@YafCondition` and specify the necessary parameters in it.

If you need more than standard annotations, you can create your own annotation, the inheritor of YafCondition, by specifying necessary parameters in it.

```
public @interface YafCondition {

    DeviceType deviceType() default DeviceType.OTHER;

    OS os() default OS.OTHER;

    MobileOS mobileOs() default MobileOS.OTHER;

    String mobileOsVersion() default Strings.EMPTY;

    Browser browser() default Browser.OTHER;

    String browserVersion() default Strings.EMPTY;

    String appVersion() default Strings.EMPTY;

    String osVersion() default Strings.EMPTY;

    int width() default 0;

    int height() default 0;
```

```
    boolean simulator() default false;
}
```

How to extend

As mentioned above, you can create your own annotations. In case you are not going to simply "fix" some annotation parameters, as for example **browser** = **chrome**, you will need your own implementation of `ConditionMather`. An example can be seen here - `BaseConditionMatcher` *(!!ссылка на класс)*

How it works

At the time the field is processed by `YafContextFieldPostProcessor`, information about the implementations (inheritors) of the given type is analyzed and if there are several ones, the `ConditionMatchService` is used to calculate points (score) on how well the given implementation matches the environment of the current test. The scoring is based on the comparison between the given properties in the condition annotation and the devices for which the test is performed.

4.7. YAF reporting

YAF contains basic reporting functionality. This implementation is a basic one, and we suggest extending it or using a production one.

4.7.1. How to

How to use

1. Add a report module dependency

▼ *Maven*

```
<dependency>
            <groupId>com.coherentsolutions.yaf</groupId>
            <artifactId>yaf-report</artifactId>
            <version>${yaf.version}</version>
</dependency>
```

▼ *Gradle*

```
dependencies {
    implementation("com.coherentsolutions.yaf:yaf-report:${yaf.version}")
}
```

2. Enable the reporting service `yaf.report.enabled=true` in YAF configuration file.
3. Configure it if necessary based on available options (`YafReportProperties`):
    - enable report logging (by default, it is shown in the console);

- change/create a new report template (using the Velocity templating tool).
- specify the path to the report template or default one will be used;

NOTE	There is one more property in `YafReportProperties` contextName that is a name of variable in template to map the result report to. Pay attention on its value creating your own report

```
yaf.report.enabled=true
yaf.report.logReport=true
yaf.report.template="templates/report.vm";
```

How to extend

There are two options to modify the reporter:

1. Changing the report template (see example `templates/executionReport.vm`).
2. Change the algorithm of the report processing. You can not just output it to the console, but, for instance, form a pdf document and send it via email. For this, you need to subscribe to the event `ExecutionReportReadyEvent` and add aditional processing logic.

```
@Service
@Slf4j
public class SendEmailPdfReportService {

    @EventListener
    public void sendPdfReport(ExecutionReportReadyEvent event) {
        PdfDocument pdf = pdfService.generate(event.getReport());
        emailService.sendPdf(getSubEmailList(), pdf);
    }

    ...
}
```

How it works

The `ReportService` class listens to events as the tests are running, aggregates them in the `ExecutionReport` object and when the `ExecutionFinishEvent` event occurs, this class completes the `ExecutionReport`, prints it (outputs to the console) using the `PrintReportService` and also generates the `ExecutionReportReadyEvent` event, which can be further processed by custom handlers.

4.8. Slack

One such handler implemented by YAF is an event processor that sends a report to a Slack

4.8.1. How to use

1. To use the functionality it is required to add slack module dependency (the module is already includes report module):

▼ *Maven*

```
<dependency>
            <groupId>com.coherentsolutions.yaf</groupId>
            <artifactId>yaf-slack</artifactId>
            <version>${yaf.version}</version>
</dependency>
```

▼ *Gradle*

```
dependencies {
    implementation("com.coherentsolutions.yaf:yaf-slack:${yaf.version}")
}
```

2. Continue adding Slack properties in properties file. There are two ways how to get access to the channel to send the report to. Every require to create an App for Slack Workspace:
   - using apiToken and channelId, see how to get permissions
   - usinga webhook, see how to create a Webhook for the channel

```
yaf.report.enabled=true
# yaf.report. ...
yaf.slack.enabled=true
yaf.slack.sendReport=true

yaf.slack.apiToken=xoxb-111-111-xxxxxxxxx
yaf.slack.channelId=XX1XX111X
# or
yaf.slack.webhookUrl=https://hooks.slack.com/services/T00000000/B00000000/XXXXXX
XXXXXXXXXXXXXXXXX
```

NOTE	apiToken is a bot(starts with `xoxb-`) or user(starts with `xoxp-`) token. channelId is the ID of the Slack channel where the message will be posted, could be found at the bottom of the channel 'About' tab in slack application

IMPORTANT	yaf.report.enabled=true is required or else ExecutionReport won't be collected and `ExecutionReportReadyEvent` event won't be published

4.9. Utils

There are numerous utilities in YAF that are designed to make test development easier.

4.9.1. Localization

The standard Spring localization mechanism (!TBD link) and the main class that implements it - `L10nService` - are used to provide localization in YAF. This service can use resources both with an explicitly specified locale and with a locale set in `TestExecutionContext` (accordingly, the localization may be test-dependent).

This mechanism is also automatically used in the templating engine.

4.9.2. Templates engine

For dynamic processing of strings (files), the framework has a templating mechanism (class `TemplateService`). Two template engines are supported:

1. Mustache - used by default, for localization and elsewhere.
2. Velocity - additional implementation, used to generate the final test execution report.

4.9.3. Store

In YAF two types of additional storage are implemented:

1. `FileStore` (!TBD link) - a wrapper over the usual file storage. For example, it is used to store downloaded versions of mobile apps.
2. `YafDbRepository` (!TBD link) - Spring boot repository on top of RocksDB (key-value store). Objects that are stored in this DB should be Serializable. To use this store just @Autowire `YafDbRepository` and use provided methods. Default DB name is `tmpdb`, you can change it in properties `yaf.tempdb.name=⋯`.

4.9.4. YAF utils

1. YafReflectionUtils
2. YafBeanUtils (be careful, some methods of this class are cached
3. YafBy

User extended YAF utils

If a developer wants to create utilities that work as context-dependently as, for example, PageObjects (!!ссылка на пейджи), he needs to inherit this utility classes from the `Utils` class and mark them with the corresponding `@Condition` annotation.

Misc utils

1. Capabilities Utils
2. Date Utils
3. File Utils
4. Properties Utils
5. Safe Parse

5. Extras

5.1. TestNg

5.1.1. TestNg integration

This module is designed to integrate YAF and TestNG.

It includes:

1. Spring Boot tests support
2. Modification of the test run
3. Basic test when work with TestNG

Spring Boot support

Since YAF is based on the Spring Boot platform, the user needs to perform the adaptation of TestNG tests and make them Spring managed (to have the ability to work with Spring context). A special class `BaseTestNgTest` is provided for this purpose, which in turn almost fully corresponds to the default Spring adapter for TestNG, except for reporting changes (Allure TestNg)

Test suite execution

A mechanism of management of test execution configurations is implemented in YAF (!!Ссылка). To make it works correctly the user needs to make changes in the test execution mechanism: duplication and parameterization of suites according to the required tests execution configuration.

For example, if you want to run tests in 2 different browsers, `YafTransformer` will create 2 suites with different browser configurations before processing the suite file and then execute them. Read more here (!!ссылка на код трансфомер)

Base test class for TestNG

If TestNG is chosen as a runner, to ensure correct work of YAF, all tests must inherit this class which provides the following features:

1. As you know, YAF has a mechanism of its own events (!TBD link) and such events as (!!список своих ивентов) are generated in this class.
2. This class is the entry point to the global data provider (!TBD link).
3. In this class the mechanism of YAF's own test processing triggers before the test method is executed (!!ссылка на кастом бины).

5.1.2. DataProviders

Some inconveniences might arise when you use the basic TestNG data providers (**DP**), such as the complexity of creating an array of objects (Object[][]), the difficulty of DP reusing and the impossibility of configuring.

A wrapper for TestNG has been created to solve these problems. It simplifies the work with data providers, and allows you to create your own.

How to

How to use

1. Above the test method add any of the possible `@···Source` annotations
2. Parameterize it as needed.

```
@Test
@RandomIntSource(size = 5, min = 1, max = 40) ①
public void yafDataProviderIntTest(int x) {
    // test code
}

@Test
@RandomStringSource(size = 5) ②
public void yafDataProviderStringTest(String x) {
    // test code
}

@Test
@LangSource(value = {EN, RU}) ③
public void yafDataProviderLangTest(Locale x) {
    // test code
}

@Test
@FakerDataSource(value = "country.currencyCode", size = 5) ④
public void yafDataProviderFakerTest(String x) {
    // test code
}

@Test
@ApiDataSource(apiCall = CustomApiCall.class) ⑤
public void yafDataProviderApiTest(Entity x) {
    // test code
}
```

① generates an array of five integers between 1 and 40

② generates an array of five strings

③ generates an array of provided Lang

④ generates an array of five strings(or other, depends on the result type of `value` expression) calculated using expression provided in value. The expression is a Java Faker methods chain

⑤ generates an array of objects gotten via api. To Use ApiDataSource it is requited to provide ApiDataSource.ApiDataCall implementation like in example. Define the class using `apiCall`

attribute while using of the annotation

```
@Component
public class CustomApiCall implements ApiDataSource.ApiDataCall {

    @Autowired
    RestAssuredYafRequest httpYafRequest;


    @Override
    public List<Args> getData(Method method, Annotation annotation,
TestExecutionContext testExecutionContext) {
        String url = "{url}";
        List<Pet> pets = httpYafRequest.anonymousReq(YafRequest.GET,
url).jsonPath().getList("", Entity.class);
        return pets.stream().map(Args::build).toList();
    }
}
```

NOTE	The implementation of the only one method includes a request and converting a response body to Arg list. `url` is a path to resource that represents an array of entities(here it is an Entity class, the instances of this class is used are accepted as a test class parameter). See Rest Assured to know details how api call is performed

How to extend

If you need to create your own data provider, you should:

1. Create an annotation `@xxxSource`.
2. Describe there all the necessary parameterization.
3. Create a handler class (Processor).
4. Relink it with the annotation

```
@Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
@Target({METHOD})
@Source(processor = RandomIntSource.Processor.class)
public @interface RandomIntSource {
//...
```

and in the processor method

```
@Override
public Class getSupportedAnnotationClass() {
    return RandomIntSource.class;
}
```

5. Implement business logic to prepare data packed in `List<Args>` as a result

```
    @Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
    @Target({METHOD})
    @Source(processor = RandomIntSource.Processor.class) ④
    public @interface RandomIntSource { ①

        ②
        int size() default 1;
        int min() default Integer.MIN_VALUE;
        int max() default Integer.MAX_VALUE;
        boolean onlyPositive() default true;
        boolean commonRange() default true;

        ③
        @Component
        class Processor extends DataProcessor<Args> {

            @Override
            public Class getSupportedAnnotationClass() {
                return RandomIntSource.class; ④
            }

            ⑤
            @Override
            public List<Args> process(Method method, Annotation annotation,
TestExecutionContext testExecutionContext) {
                RandomIntSource source = (RandomIntSource) annotation;
                int min = source.commonRange() ? -100 : source.min();
                int max = source.commonRange() ? 100 : source.max();
                min = source.onlyPositive() ? 0 : min;
                return new Random().ints(source.size(), min, max).boxed().map(i ->
Args.build(i))
                    .collect(Collectors.toList());
            }
        }
    }
```

How it works

The concept of own DPs based on 2 mechanisms:

1. Before the start of the tests, the built-in listener `YafTransformer` runs and processes the annotations on the test methods.

   In case a method lacks @DataProvider annotation, but the number of parameters is not equal to zero, a global YAF data-provider is assigned to this method.

2. The YAF-global data provider analyzes the test method, finds the custom DP and calls it together with all the meta-information about the test method.

Apart from that, it can mix in parameters from DP and global test parameters (from suite file).

After the processor processes custom data provider, data from `List<Args>` is converted into standard `Object[][]` and returned by global data provider to the test method.

5.2. RestAssured

5.3. Rest Assured API implementation sample

The implementation of the request contract above the RestAssured library can be found in the `RestAssuredYafRequest` class. It is recommended to be familiar with common API guide first.

The input arguments of this object include:

- AuthProvider;
- Default user;
- The initial parametrization of the quest.

IMPORTANT	do not forget to implement custom logging filter and append this logs to the test end event

5.3.1. How to use

1. To start off add corresponding module dependency to the project

▼ *Maven*

```
        <dependency>
            <groupId>com.coherentsolutions.yaf</groupId>
            <artifactId>yaf-api-restassured</artifactId>
            <version>${yaf.version}</version>
        </dependency>
```

▼ *Gradle*

```
dependencies {
    implementation("com.coherentsolutions.yaf:yaf-api-
restassured:${yaf.version}")
}
```

2. Create your own auth layer (if you need any).

```
public class SampleAuthProvider extends RestAssuredAuthProvider {

    @Override
    protected AuthToken getAuthToken(YafApiUser user) throws YafApiRequestException
{
```

```
        UserToken userToken = ((RequestSpecification)
request.anonymousReq()).body(user)
                .post("/api/auth/password")
                .as(UserToken.class);①
        //convert token to AuthToken
        AuthToken token = new AuthToken();
        token.setToken(userToken.getAccessToken());
        return token;
    }
```

NOTE <1> UserToken is a model class represents particular auth response(here it is POST "/api/auth/password") body(usually json)

3. Create a user POJO that will be used for auth.

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@JsonIgnoreProperties(ignoreUnknown = true)
@JsonInclude(JsonInclude.Include.NON_NULL)
public class UserLogin implements YafApiUser {
    @JsonProperty("login")
    private String login;

    @JsonProperty("password")
    private String password;
}
```

4. Choose an api req implementation (for now only restAssured supported). In your config (see) class declares a bean for `YafRequest` with proper implementation and auth provider (if you need any)

```
    @Bean
    public RestAssuredYafRequest req() {
        return RestAssuredYafRequest.builder()
                .withAuthProvider(new SampleAuthProvider()) ①
                .withDefaultUser(new UserLogin("xxx", "yyy")) ②
                .build();
    }
```

NOTE <1> Set AuthProvider <2> Specify default user Both are optional so current bean definition step could be skipped if your API accepts requests without authorization and RestAssuredYafRequest's anonymousReq method could be used (RestAssuredYafRequest's req method usage will throw the exception). Default user could be also skipped if user configuration defined (see)

5. Configure appropriate properties in `application.properties` file (see here)

6. @Autowire your request bean, and use it

```
    @Autowired
    RestAssuredYafRequest request; //yes, we autowire final impl, cause base class
 is generic one, and we'll think how to deal with it later)

    @Test
    public void sampleApiTest() throws YafApiRequestException {
        String roles = request.get("api/settings").asString();
        // code
    }
```

5.4. Allure

5.4.1. Allure integration

Integration with Allure allows the user to get a detailed test report quickly. The basic implementation of Allure requires some work from the developer, such as:

- adding logs to the report;
- modifying the test name in case of working with repetitive tests;
- adapting to work with own annotations, ect.

In YAF all these difficulties are solved in the `AllureService` class.

How to

How to use

To use Allure module, follow the instructions

Add allure yaf modules dependencies

▼ *Maven*

```
<dependency>
    <groupId>com.coherentsolutions.yaf</groupId>
    <artifactId>yaf-allure</artifactId>
    <version>${yaf.version}</version>
</dependency>
<dependency>
    <groupId>com.coherentsolutions.yaf</groupId>
    <artifactId>yaf-allure-testng</artifactId>
    <version>${yaf.version}</version>
</dependency>
```

▼ *Gradle*

```
dependencies {
    implementation("com.coherentsolutions.yaf:yaf-allure:${yaf.version}")
    implementation("com.coherentsolutions.yaf:yaf-allure-testng:${yaf.version}")
}
```

To add custom details to your allure report like steps, attachments and so on via annotations

▼ *Maven*

```
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.20</version>
                <configuration>
                    <argLine>

-javaagent:"${settings.localRepository}/org/aspectj/aspectjweaver/${aspectj.version}
/aspectjweaver-${aspectj.version}.jar"
                    </argLine>
                </configuration>
                <dependencies>
                    <dependency>
                        <groupId>org.aspectj</groupId>
                        <artifactId>aspectjweaver</artifactId>
                        <version>${aspectj.version}</version>
                    </dependency>
                </dependencies>
        </plugin>
```

▼ *Gradle*

```

```

In the basic configuration everything is ready. You can use both standard Allure annotations such as @Step, @Description, @TmsId, etc. and the global YAF annotation `@YafTest` - all information from them will be pasted to the report.

How to extend

The completeness of the report can be affected in 3 ways:

1. By setting special `yaf.allure⋯.` features in the framework configuration file.

   ```
   yaf.allure.enabled=true
   yaf.allure.fullLogAllTests=true
   yaf.allure.envMap.platform=linux
   ```

   Here, fullLogAllTests - if true logs will be attached to report in both success or failed test statuses

envMap - define allure environment variables, here platform is a key and linux is a value (see allure environment)

2. By changing or adding your allure report data about test during processing events such as TestStartEvent, TestFinishEvent. Use Allure class static methods to customize your report here. There are example what could be done

```
@EventListener
public void testStart(TestStartEvent testStartEvent) {
        // use Allure class methods to customize your report here, for instanse
        // add a step
        Allure.step("Test starting..." +
testStartEvent.getTestInfo().getFullTestName());
        // or an attachment
        Allure.addAttachment("test config",
testStartEvent.getTestInfo().getEnvSetup());
        // mark test case with feature label if test case maked by annotation
        if(testStartEvent.getTestInfo().getAnnotations().contains(Test.class)) {
            Allure.feature("custom");
            // or add custom label
            Allure.label("marker", "custom");
        }
}
```

3. By extending the `AllureService` class and marking own implementation as `@Primary`.

How it works

The `AllureService` class listens to events regarding the start, and the end of tests, and adds all the necessary information to the Allure context.

In Allure's lifecycle, not all information can be changed once it is added to the context, but this can be done with reflection. To do this, an additional `TestPatcher` is added to Allure's listeners, which completes the necessary cases.

5.4.2. Allure Test NG integration

Due to the fact that the framework uses a large number of wrappers and special hooks, they all end up in the base report and thus clutter it up.

How to

How to use

To exclude unnecessary methods from the report, a special annotation `@SkipReport` is used.

Example

```
    @SkipReport
```

```
    @BeforeClass
    public void skipReportBeforeClass(ITestContext testContext) {
        super.beforeClass(testContext);
    }
```

How it works

There is a special aspect in the module that intercepts the execution of methods in the standard TestNG adapter, and if a method is marked with this annotation, it is "skipped" (not included in the report).

5.5. Selenide

5.5.1. Selenide integration

To integrate YAF and Selenide, `SelenideDriverManager` class is used. Its main task is to listen to a driver creation event and, if it is a web driver, pass it to the Selenide driver repository.

Also, in case of selenide, in order for `YafLocatorFactory` (! ссылка на доку to work correctly, you must skip processing `SelenideElement` on the page.

5.6. Emails

To facilitate working with structured data, the framework provides a special mechanism that allows you to inject POJO objects into the test depending on the execution context.

The basic implementation is based on storing data files in JSON format, but it can be reconfigured.

5.6.1. How to

5.6.2. How to use

1. The first required action is to add a mail module dependency

▼ *Maven*

```
<dependency>
    <groupId>com.coherentsolutions.yaf</groupId>
    <artifactId>yaf-mail</artifactId>
    <version>${yaf.version}</version>
</dependency>
```

▼ *Gradle*

```
dependencies {
    implementation("com.coherentsolutions.yaf:yaf-mail:${yaf.version}")
}
```

2. For both sending or receiving emails it will require to have a mail account to retrieve emails from in case of testing or to send messages from in case of test events notification sending. The account credentials must be provided in properties (yaf.mail.user-email and yaf.mail.password)

IMPORTANT	Some additional account settings could be required that depends on Mail Client is used, for instance, to generate an app password or enable SMTP/POP3/IMAP

```
yaf.mail.enabled=true
yaf.mail.user-email=xxxx@gmail.com
yaf.mail.password=xxxx
```

Farther steps are depends on YAF mail using purposes

How to receive

1. To connect to a mail folder, and it is required to have the host to connect to in yaf.mail.store-host property. The host value depends on store protocol(IMAP by default) and email client:

	SMTP	IMAP	POP3
Gmail	smtp.gmail.com	imap.gmail.com	pop.gmail.com
Outlook/Office 365	smtp.office365.com	outlook.office365.com	outlook.office365.com
Yahoo Mail	smtp.mail.yahoo.com	imap.mail.yahoo.com	pop.mail.yahoo.com
iCloud Mail	smtp.mail.me.com	imap.mail.me.com	pop.mail.me.com

```
yaf.mail.enabled=true
yaf.mail.user-email=xxxx@gmail.com
yaf.mail.password=xxxx
yaf.mail.store-protocol=pop3
yaf.mail.store-host=pop.gmail.com
yaf.mail.folder=inbox
```

NOTE	yaf.mail.folder could be omitted as 'inbox' is a default value or customized, for instance, changed to 'trash', 'sent', 'archive' and so on (the names could differ depends on mail client) or any other folder's name created previously

1. The next step is email(s) retrieving. There are several options that EmailService provides:
    - readSingleEmail gets the most recent email or the most recent email that corresponds condition if it is provided
    - readEmails gets the first 10 emails by default if other settings was not provided. Emails could be filtered by condition as well
    - waitForEmails wait for a new emails according provided settings(required). Conditions are optional. There is method implementation that accepts additional Runnable method

argument and starts waiting as soon as provided operation is performed. It allows to catch an email that expected to be received as result of any operation

Let's elaborate what's mentioned above settings and conditions mean:

EmailQuerySettings allows to override settings from application profile for particular request(s) like account's creds or target folder, pagination<1> and timing settings<2>

EmailQueryCondition(see more in How does in work section) is an interface and several commonly used implementations with the ait to collect only emails that match one or more conditions. There are subject and recipient conditions usages in the example below. There are AfterDateEmailQueryCondition and EmailNumberMoreQueryCondition conditions as well and obviously the custom one could be added

```
public static void main(String[] args) {

    EmailService service = new JavaEmailService();
    Email email = service.readSingleEmail();
    List<Email> emails = service.readEmails();

    EmailQuerySettings paginatingSettings = new EmailQuerySettings();
    paginatingSettings.setPage(2);
    paginatingSettings.setPageSize(20); ①
    List<Email> welcomeEmails = service.readEmails(paginatingSettings, new
SubjectEmailQueryCondition("Welcome"));

    EmailQuerySettings waitForSettings = new EmailQuerySettings();
    waitForSettings.setInitialDelay(1000);
    waitForSettings.setTimeOut(3000);
    waitForSettings.setPooling(200);②
    service.waitForEmails(waitForSettings, new
RecipientQueryCondition("recipient@gmail.com"));
}
```

2. Finally, it comes to email's parsing. EmailService offer to parse email's body as a string or html document, extract attachments. EmailService parse multipart messages inside, but it is possible to get list of parts separately<1>

```
public static void main(String[] args) {
    EmailService service = new JavaEmailService();
    Email email = service.readSingleEmail();
    String body = service.getMessageData(email);
    Document html = service.getMessageDataHtml(email);
    List<MultiPartData> multipart = service.getMessageDataMultipart(email); ①
    List<Attachment> attachments = service.getAttachments(email);
}
```

3. Customizing(optional). Described above EmailQuerySettings could be defined globally to define

custom timing and pagination settings for requests where EmailQuerySettings are not provided.

```
yaf.mail.pooling-time = 2000L
yaf.mail.delay-time = 0L;
yaf.mail.timeout = 60000L;

yaf.mail.timeout.default-page = 1;
yaf.mail.timeout.default-page-size = 10;
```

NOTE	The example contains values that are default for presentation

How to send

1. As previously start from connection config. See table in How to receive section SMTP column to find host value

```
yaf.mail.enabled=true
yaf.mail.user-email=xxxx@gmail.com
yaf.mail.password=xxxx
yaf.mail.transport-protocol="smtp";
yaf.mail.transport-host="smtp.gmail.com";
yaf.mail.transport-port = 587;
```

NOTE	transport-protocol and transport-port could be omitted as SMTP protocol is used by default as the most commonly used for email sending purposes

5.6.3. How does it work

YAF mail module use Jakarta Mail under the hood. Using it as a base the module:

- init session and connect to the mail store caching the connections using Spring Cache by user email and folder. For ex, connection to the 'inbox' folder under 'test@gmail.com' will be created ones and reused before is reset. In turn cache is cleared every 30 minutes that is configurable in application properties:

  ```
  yaf.mail.store-t-t-l=600000
  ```

IMPORTANT	value should be provided in milliseconds

- converts jakarta mail Message to more readable Email object and back. Particularly, mail body from specific multipart of different types(TEXT, HTML, PNG) to user familial string/html document
- provides clear and expandable way to filter emails to retrieve from the folder - EmailQueryCondition. To create a custom one it is required to implement the described interface<1> providing a condition as Predicate<3>, dynamic param(s) for the condition are

defined as private class variables<2> and provided while instance creation

```
public class SenderEmailQueryCondition implements EmailQueryCondition<Email> { ①

    private String sender; ②

    public SubjectEmailQueryCondition(String sender) {
        this.sender = sender;
    }

    @Override
    public Predicate<Email> buildPredicate() { ③
        return o -> o.getFrom().contains(sender);
    }


    @Override
    public String getApiCondition() {
        throw new NotImplementedException("Not implemented yet!");
    }
}
```

6. Q&A

Question 1

Sample Answer 1

Question 2

Sample Answer 2