approval Documentation

Release 0.3

nikolavp

August 09, 2016

Contents

Approval provides a powerful toolkit of ways to test the behavior of critical components so you can prevent problems **in your production environment.**

CHAPTER 1

What can it be used for?

Approval can be used for verifying objects that require more than a simple assert. The idea is that you sometimes just want to verify a particular result at the end and then start implementation refactoring. I like to call it "I will know the right result when I see it". Usecases for this might be:

- performance improvements to the implementation while preserving the current system output
- just verifying RESTful response results, be it JSON, XML, HTML whatever
- people use it for TDD but instead of providing the result upfront you just implement the simple possible thing, verify the result and then start improving the implementation.

1.1 Getting Started

***Getting Started* will guide you through the process of testing your classes with approval testing. Don't worry if you are used to normal testing with assertions you will get up to speed in minutes.**

1.1.1 Setting Up Maven

Just add the `approval` library as a dependency:

```
<dependencies>
    <dependency>
        <groupId>com.github.nikolavp</groupId>
        <artifactId>approval-core</artifactId>
        <version>${approval.version}</version>
    </dependency>
</dependencies>
```

Warning: Make sure you have a `approval.version` property declared in your POM with the current version, which is 0.3.

1.1.2 How is approval testing different

There are many sources from which you can learn about approval testing(just google it) but basically the process is the following:

1. you already have a working implementation of the thing you want to test
2. you run it and get the result the first time

3. the result will be shown to you in your preferred tool(this can be configured)
4. you either approve the result in which case it is recored(saved) and the test pass or you disapprove it in which case the test fails
5. the recorded result is then used on further test runs to make sure that there are no regressions in your code(i.e. you broke something and the result is not the same).
6. Of course sometimes you want to change the way something behaves so if the result is not the same we will prompt you with difference between the new result and the last recorded again in your preferred tool.

1.1.3 Approvals utility

This is the main starting point of the library. If you want to just approve a primitive object or arrays of primitive object then you are ready to go. The following will start the approval process for a `String` that `MyCoolThing` (our class under test) generated and use `src/test/resources/approval/string.verified` for recording/saving the results:

```
    @Test
    public void testMyCoolThingReturnsProperString() {
        String result = MyCoolThing.getComplexMultilineString();
        Approvals.verify(result, Paths.get("src", "resources", "approval", "result.txt"));
    }
```

1.1.4 Approval class

This is the main object for starting the `approval` process. Basically it is used like this:

```
    @Test
    public void testMyCoolThingReturnsProperStringControlled() {
        String string = MyCoolThing.getComplexMultilineString();
        Approval<String> approver = Approval.of(String.class)
                .withReporter(Reporters.console())
                .build();
        approver.verify(string, Paths.get("src", "resources", "approval", "string.verified"));
    }
```

note how this is different from *Approvals utility* - we are building a custom `Approval` object which allows us to control and change the whole approval process. Look at *Reporter class* and *Converter* for more info.

Note: Approval object are thread safe so you are allowed to declare them as static variables and reuse them in all your tests. In the example above if we have more testing methods we can only declare the `Approval` object once as a static variable in the *Test* class

---

1.1.5 Reporter class

Reporters(in lack of better name) are used to prompt the user for approving the result that was given to the *`Approval`* object. There is a *`withReporter`* method on *`ApprovalBuilder`* that allows you to use a custom reporter. We provide some ready to use reporters in the following classes:

- *`Reporters`* - this factory class contains cross platform programs/reporters. Here you will find things like *`gvim`*, *`console`*.

- *WindowsReporters* - this factory class contains Windows programs/reporters. Here you will find things like *notepadPlusPlus*, *beyondCompare*, *tortoiseText*.
- *MacOSReporters* - this factory class contains MacOS programs/reporters. Here you will find things like *diffMerge*, *ksdiff*, etc.

Note: Sadly I am unable to properly test the windows and macOS reporters because I mostly have access to Linux machines. If you find a problem, blame it on me.

1.1.6 Converter

Converters are objects that are responsible for serializing objects to raw form(currently `byte[]`). This interface allows you to create a custom converter for your custom objects and reuse the approval process in the library. We have converters for all primitive types, String and their array variants. Of course providing a converter for your custom object is dead easy. Let's say you have a custom entity class that you are going to use for verifications in your tests:

```
package com.github.approval.example;

public class Entity {

    private String name;
    private int age;

    public Entity(String name, int age) {
        this.age = age;
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

}
```

Here is a possible simple converter for the class:

```
package com.github.approval.example;

import com.github.approval.converters.Converter;

import javax.annotation.Nonnull;
import java.nio.charset.StandardCharsets;

public class EntityConverter implements Converter<Entity> {
    @Nonnull
    @Override
    public byte[] getRawForm(Entity value) {
        return ("Entity is:\n" +
                "age = " + value.getAge() + "\n" +
                "name = " + value.getName() + "\n").getBytes(StandardCharsets.UTF_8);
    }
}
```

now let's say we execute a simple test

```
        Entity entity = new Entity("Nikola", 30);
        Approval<Entity> approver = Approval.of(Entity.class)
                .withReporter(Reporters.console())
                .withConveter(new EntityConverter())
                .build();
        approver.verify(entity, Paths.get("src/test/resources/approval/example/entity.verified"));
    }
}
```

we will get the following output in the console(because we are using the console reporter)

> Entity is:
> age = 30
> name = Nikola

1.1.7 Path Mapper

Path mapper are used to abstract the way in which the final path file that contains the verification result is built. You are not required to use them but if you want to add structure to the your approval files you will at some point find the need for them. Let's see an example:

You have the following class containing two verifications:

```
package com.github.approval.example;

import com.github.approval.Approval;
import com.github.approval.reporters.Reporters;
import org.junit.Test;

import java.nio.file.Paths;

public class PathMappersExample {
    private static final Approval<String> APPROVER = Approval.of(String.class)
            .withReporter(Reporters.console())
            .build();

    @Test
    public void shoulProperlyTestString() throws Exception {
        APPROVER.verify("First string test", Paths.get("src", "test", "resources", "approvals", "firs
    }

    @Test
    public void shoulProperlyTestStringSecond() throws Exception {
        APPROVER.verify("Second string test", Paths.get("src", "test", "resources", "approvals", "sec
    }
}
```

now if you want to add another approval test you will need to write the same destination directory for the approval path again. You can of course write a private static method that does the mapping for you but we can do better with PathMappers:

```
package com.github.approval.example;

import com.github.approval.Approval;
import com.github.approval.pathmappers.ParentPathMapper;
import com.github.approval.reporters.Reporters;
```

```
import org.junit.Test;

import java.nio.file.Paths;

public class PathMappersExampleImproved {
    private static final Approval<String> APPROVER = Approval.of(String.class)
            .withReporter(Reporters.console())
            .withPathMapper(new ParentPathMapper<String>(Paths.get("src", "test", "resources", "appro
            .build();

    @Test
    public void shoulProperlyTestString() throws Exception {
        APPROVER.verify("First string test", Paths.get("first-test.txt"));
    }

    @Test
    public void shoulProperlyTestStringSecond() throws Exception {
        APPROVER.verify("Second string test", Paths.get("second-test.txt"));
    }
}
```

we abstracted the common parent directory with the help of the *ParentPathMapper* class. We provide other path mapper as part of the library that you can use:

- *JunitPathMapper*

1.2 User Manual

1.2.1 Simple example of the library

Let's try to test the simplest example possible:

```
package com.github.approval.example;

public class SimpleExample {
    public static String generateHtml(String pageTitle) {
        return String.format(
                "<!DOCTYPE html>\n" +
                "<html lang=\"en\">\n" +
                "<head>\n" +
                "    <title>%s</title>\n" +
                "<meta charset=\"utf-8\"/>\n" +
                "<link href=\"css/myscript.css\"\n" +
                "      rel=\"stylesheet\"/>\n" +
                "<script src=\"scripts/myscript.js\">\n" +
                "</script>\n" +
                "</head>\n" +
                "<body>\n" +
                "...\n" +
                "</body>\n" +
                "</html>", pageTitle);
    }
}
```

now this class is not rocket science and if we want to test **generateHtml()**, we would write something like the following in JUnit:

```java
package com.github.approval.example;

import org.junit.Test;

import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertThat;

public class SimpleExampleTest {

    @Test
    public void shouldReturnSomethingToTestOut() throws Exception {
        //arrange
        String title = "myTitle";
        String expected = "<!DOCTYPE html>\n" +
                "<html lang=\"en\">\n" +
                "<head>\n" +
                "   <title>" + title +"</title>\n" +
                "<meta charset=\"utf-8\"/>\n" +
                "<link href=\"css/myscript.css\"\n" +
                "      rel=\"stylesheet\"/>\n" +
                "<script src=\"scripts/myscript.js\">\n" +
                "</script>\n" +
                "</head>\n" +
                "<body>\n" +
                "...\n" +
                "</body>\n" +
                "</html>";

        //act
        String actual = SimpleExample.generateHtml(title);

        //assert
        assertThat(actual, equalTo(expected));
    }
}
```

this is quite terse and short. Can we do better? Actually because we support strings out of the box, approval is a lot shorter

```java
package com.github.approval.example;

import com.github.approval.Approvals;
import org.junit.Test;

import java.nio.file.Paths;

public class SimpleExampleApprovalTest {

    @Test
    public void shouldReturnSomethingToTestOut() throws Exception {
        //assign
        String title = "myTitle";

        //act
        String actual = SimpleExample.generateHtml(title);

        //verify
        Approvals.verify(actual, Paths.get("src/test/resources/test.txt"));
```

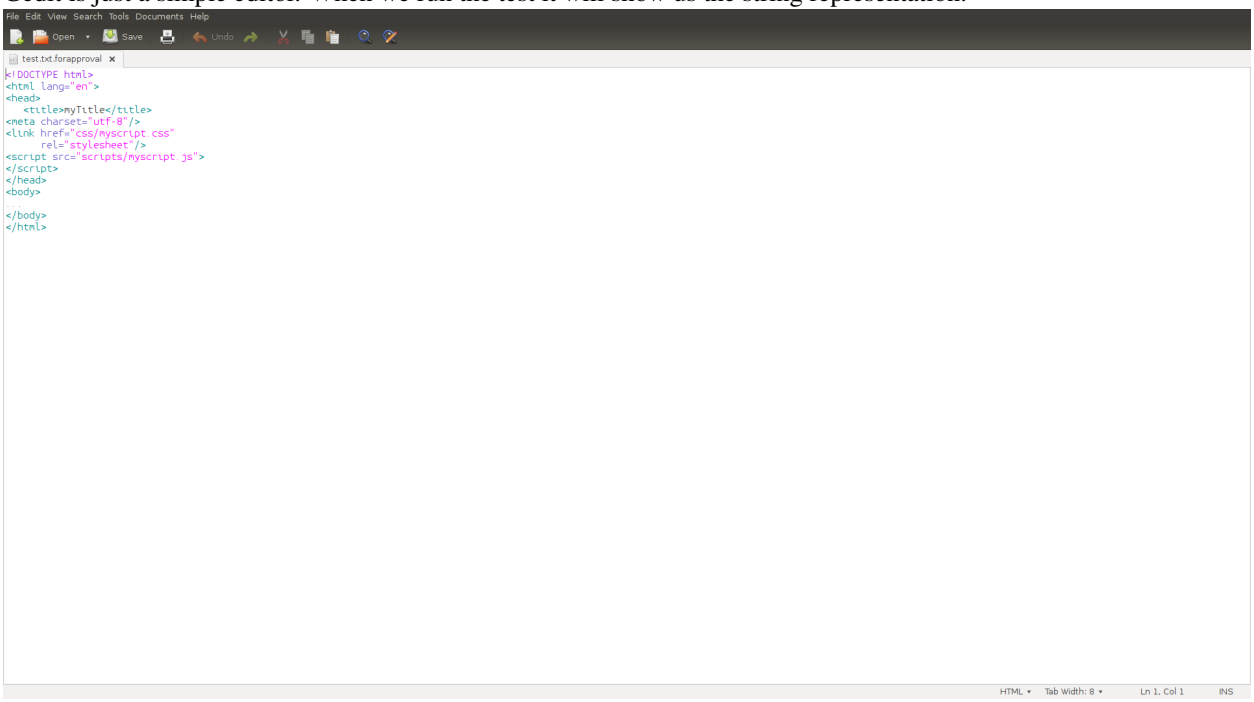
```
        }
}
```

when the latter is executed you will be prompted in your tool of choice to verify the result from **generateHtml()**. Verifying the result will vary from your tool of choice because some of them allow you to control the resulting file and others just show you what was the verification object.

To see it in action we will look at two possible reporters:

Gedit

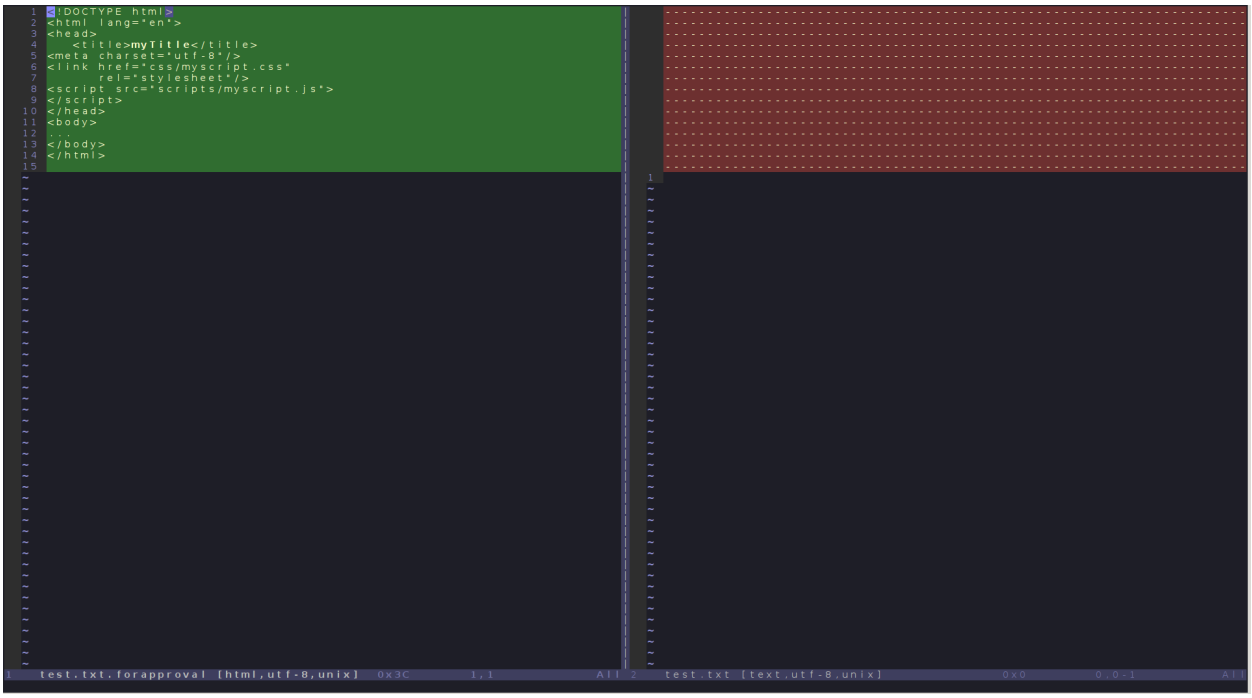
Gedit is just a simple editor. When we run the test it will show us the string representation:



as you can see this is the string representation of the result opened in gedit. If we close gedit we will prompted by a confirm window which will ask us if we approve the result or it is not OK. On not OK the test will fail with an `AssertionError` and otherwise the test will pass and will continue to pass until the returned value from **generateHtml()** changes.

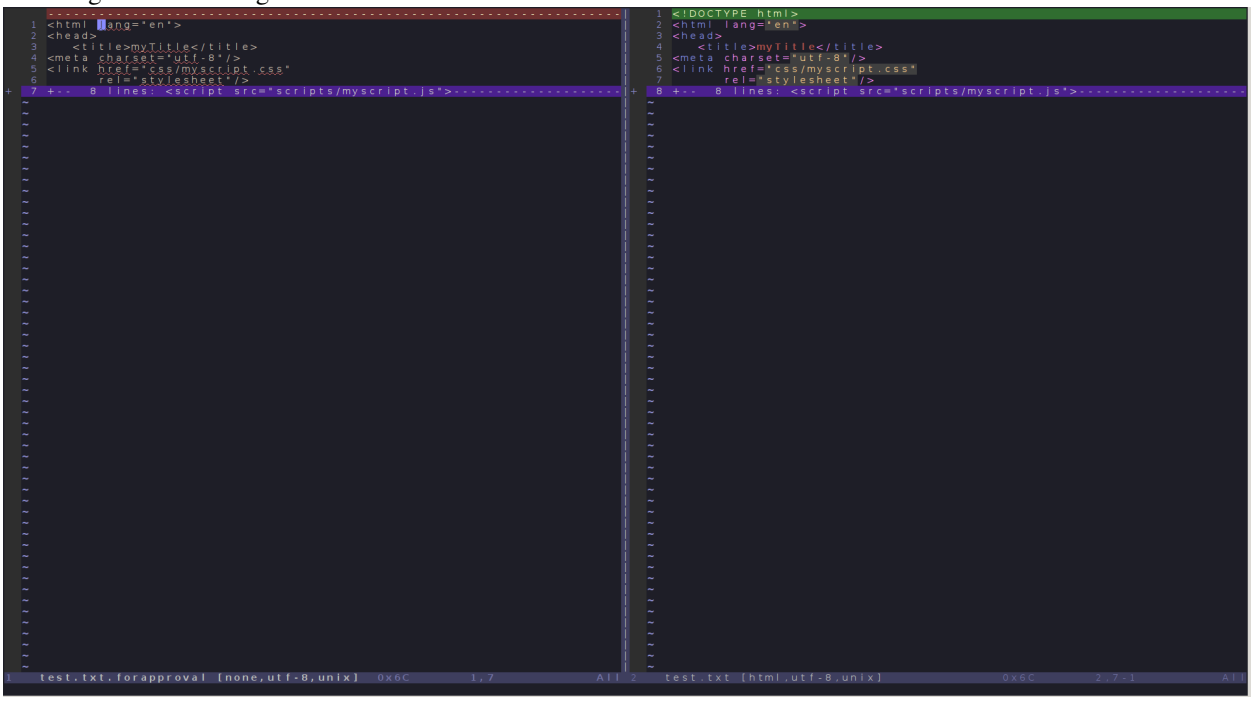
GvimDiff

Gvimdiff is much more powerful than gedit. If we decide to use it then we got the power in our hands and we can decide if we want the file or not(there will be no confirmation window). Here is how it looks like:



as you can see on the left side is the result from the test run and on the right side is what will be written for consecutive test runs. If we are ok with the result we can get everything from the left side, save the right side and exit vim. The test will now pass and will continue to pass until the returned value from **generateHtml()** changes.

Let's say someone changes the code and it no longer contains a DOCTYPE declaration. The reporter will fire up and we will get the following window:



we can approve the change or exit our tool and mark the result as invalid.

1.2.2 Converters usecase

In this section we will show you how to use a custom converter for a class that you have written yourself and want to verify. For our example we will use the excellent jackson library.

Let's say you want to verify that the following entity object returned from your restful API will be represented properly in JSON by jackson:

```
package com.github.approval.example.converters;

public class Person {
    private String name;

    private String email;

    private Person() {
        // Jackson deserialization
    }

    public Person(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    // hashCode
    // equals
    // toString etc.
}
```

we will use the following **person** as an example:

```
        final Person person = new Person("Luther Blissett", "lb@example.com");
```

Normal setup with assertions

now in normal circumstances you will now have to do the following:

- create a file representing the json serialization with the following content(for example named *person.json*):

```
/*
 * #%L
 * com.github.nikolavp:approval-core
```

```
 * %%
 * Copyright (C) 2014 - 2016 Nikolavp
 * %%
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 * #L%
 */
{"name":"Luther Blissett","email":"lb@example.com"}
```

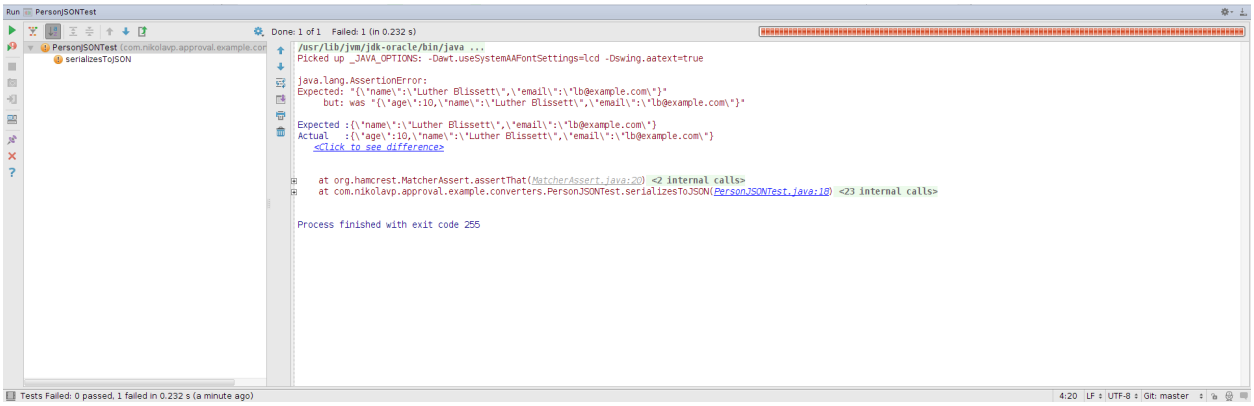
- read the content of the file
- run ObjectMapper on the person object to get the json representation
- compare the last two results and verify that they are the same

your code might be something like the following using guava

```
package com.github.approval.example.converters;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.google.common.io.Resources;
import org.junit.Test;

import static java.nio.charset.StandardCharsets.UTF_8;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertThat;

public class PersonJSONTest {
    private static final ObjectMapper MAPPER = new ObjectMapper();

    @Test
    public void serializesToJSON() throws Exception {
        final Person person = new Person("Luther Blissett", "lb@example.com");
        final String fixtureValue = Resources.toString(Resources.getResource("person.json"), UTF_8);
        assertThat(MAPPER.writeValueAsString(person), equalTo(fixtureValue));
    }
}
```

now run the tests and they should pass. Of course software is all about change so if you later change the representation of the person and add an age field, you will have to do the following:

- run the tests and see them fail with message that would say something like the following:



- now you will have to manually look at that message, see the difference between the two and change the person.json file appropriately to match the newly added age field(this is all manual work and you know what manual work leads to, right?)

Using approval

Using approval, we will first have to build a converter that converts our Person class to a string form(currently the library doesn't have a jackson converter). A simple converter like the following will work:

```
package com.github.approval.example.converters;


import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.github.approval.converters.AbstractStringConverter;

import javax.annotation.Nonnull;
import javax.annotation.Nullable;

public class JacksonConverter<T> extends AbstractStringConverter<T> {
    private static final ObjectMapper mapper = new ObjectMapper();
    @Nonnull
    @Override
    protected String getStringForm(@Nullable T value) {
        try {
            return mapper.writeValueAsString(value);
        } catch (JsonProcessingException e) {
            throw new AssertionError("Couldn't convert " + value + " to json!", e);
        }
    }
}
```

Note: This converter is super generic. It can convert any object that is accepted by jackson(hence the name of the converter)

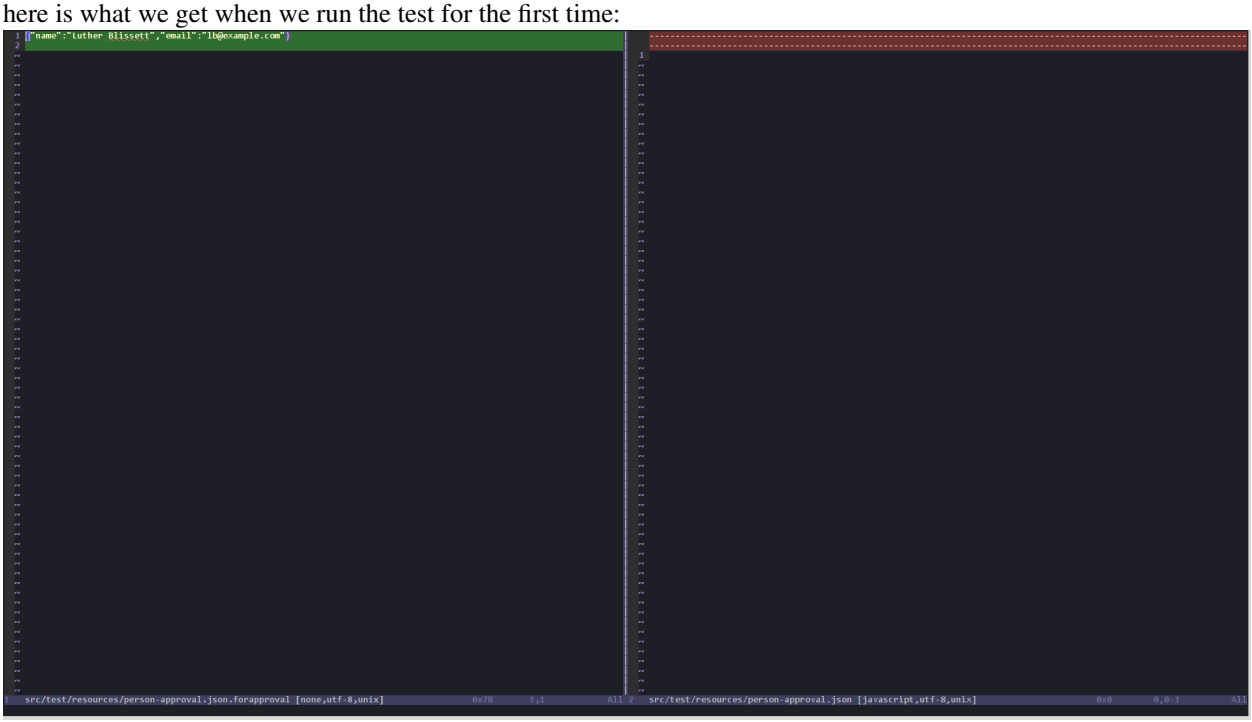
now let's write our test:

```
package com.github.approval.example.converters;

import com.github.approval.Approval;
import com.github.approval.reporters.Reporters;
import org.junit.Test;
```

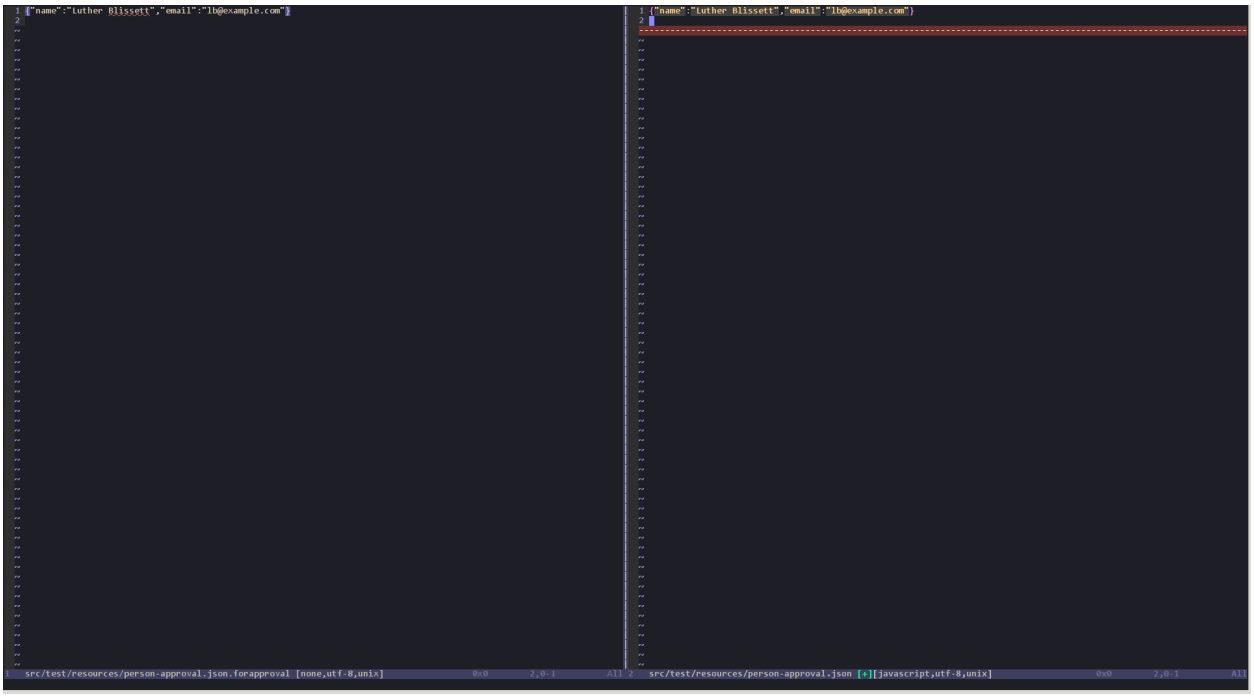
```
import java.nio.file.Paths;

public class PersonApprovalTest {
    private static final Approval<Person> APPROVER = Approval.of(Person.class)
            .withConveter(new JacksonConverter<Person>())
            .withReporter(Reporters.gvim())
            .build();

    @Test
    public void serializesToJSON() throws Exception {
        final Person person = new Person("Luther Blissett", "lb@example.com");
        APPROVER.verify(person, Paths.get("src/test/resources/person-approval.json"));
    }

}
```

Note: we didn't have to write the person.json file, it will be generated for us the first time the test is run

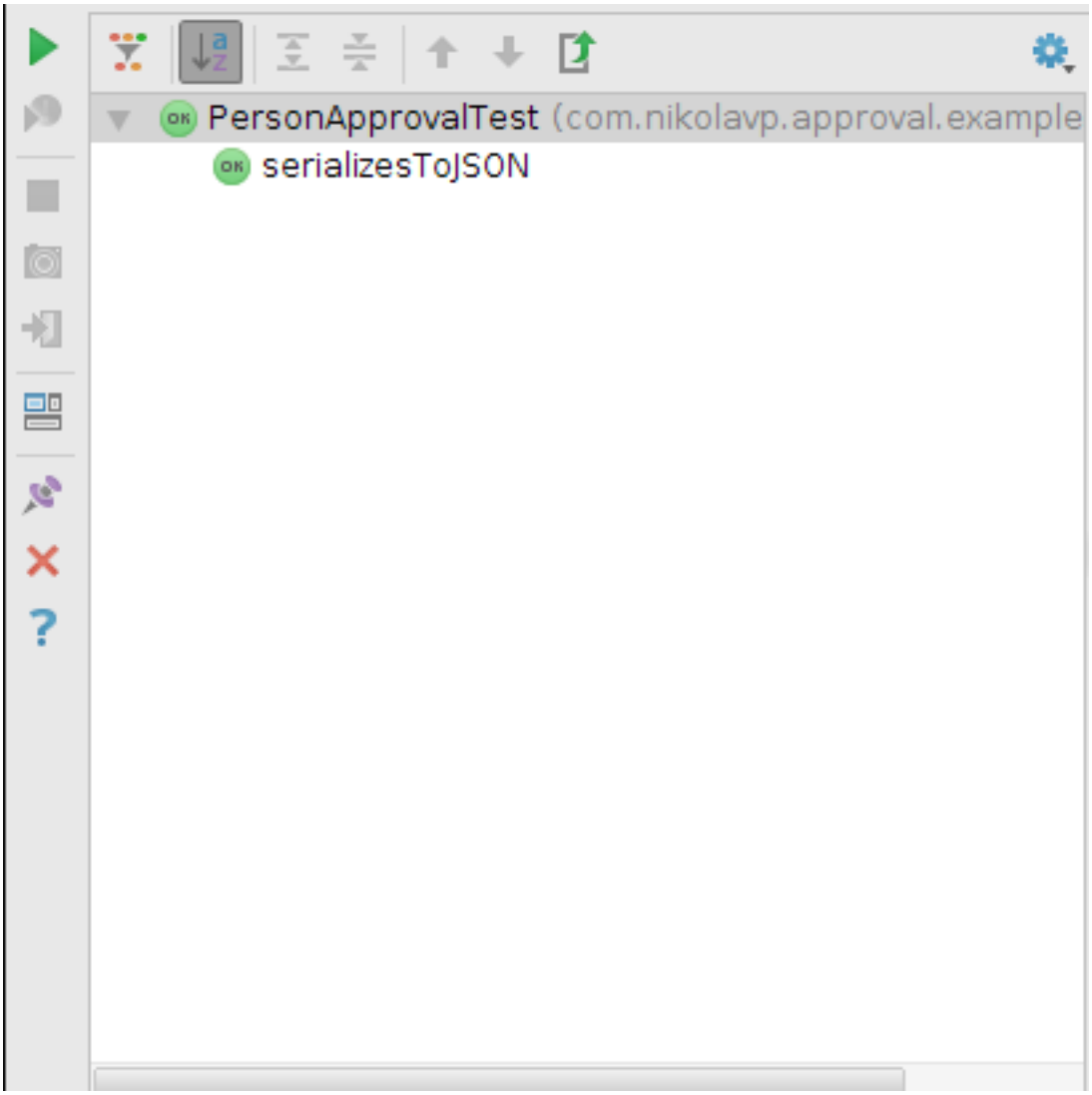
here is what we get when we run the test for the first time:



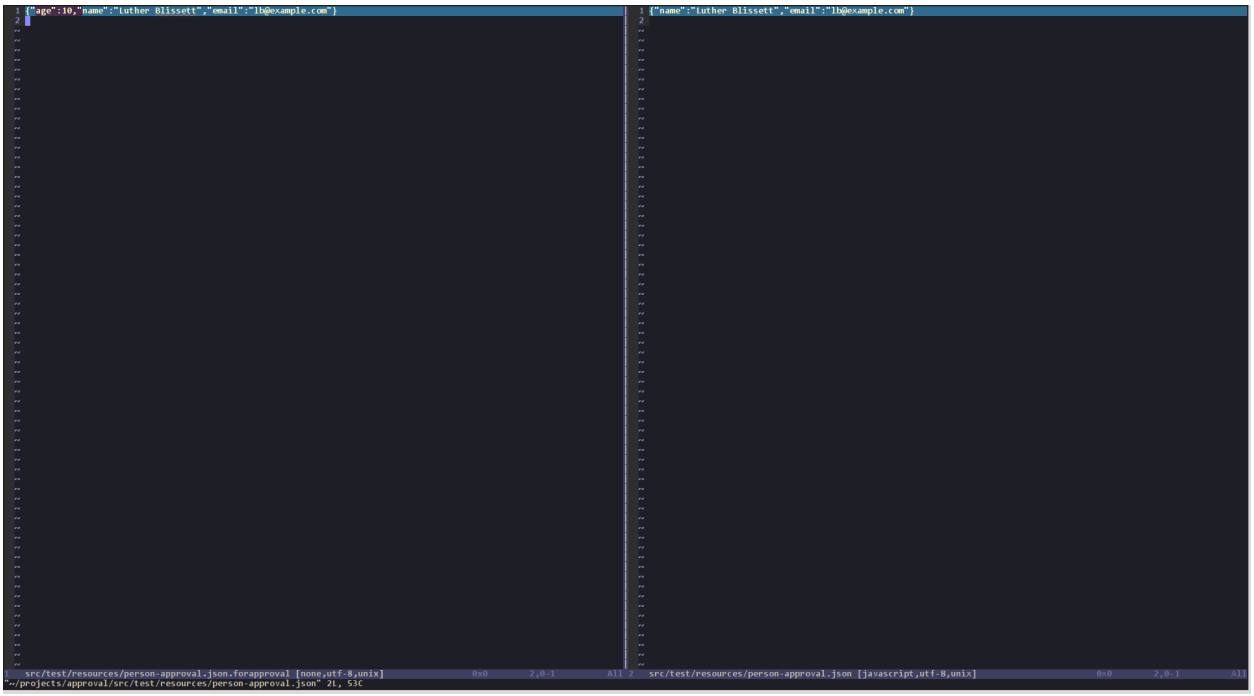
basically the library is asking us if we "approve" the result on the left side and what parts of it we want to move to the right side(most of the time all of it). In this case we want the whole left side so we get the following:



now when we save the right hand side and close our tool, we can see the green bar in our test runner(intellij in this case)



running the tests again will continue to pass. What's interesting is if we change the Person class and add an age field. We just rerun the tests but now they won't pass, we will be prompted in our tool again with something like the following:



we can "approve" the change(move the change from left to right) or we can close the tool and state that the new value is invalid; we should then fix our code.

What's interesting in the approval case is that we didn't have to manually check the new value and verify that it is valid in the console view. We got a good looking diff window that prompt us for verification.

Note: This guide is using an example from the dropwizard testing documentation

---

1.3 Adapters

We provide many adapters for external libraries that you can use in the verification process. The adapters for a particular framework are represented as submodules in our codebase so you will need to add them explicitly as dependencies.

1.3.1 Sesame

Sesame is one of the frameworks that we support out of the box. The integration allows you to verify `Graph` objects through graphviz.

1. Ok so first you will need the dot binary otherwise our reporter will fail. Go to the Graphviz, download and install graphviz.
2. Add the dependency in maven:

```
<dependencies>
    <dependency>
        <groupId>com.github.nikolavp</groupId>
        <artifactId>approval-sesame</artifactId>
        <version>${approval.version}</version>
    </dependency>
</dependencies>
```

3. Now you will be able to approve graph objects with the following:

```
        Graph graph = new TreeModel();
        // populate our graph with statements(maybe from GraphQueryResult?

        // Note: this is still thread safe...
        Approval<Graph> graphApproval = Approval.of(Graph.class)
                .withConveter(new GraphConverter())
                .withReporter(GraphReporter.getInstance())
                .build();

        // Verify the graph, change the path accordingly
        graphApproval.verify(graph, Paths.get("graph-result.dot"));
```

1.4 FAQ

1.4.1 Is there a way to delete orphaned(not used) approval files?

Yes! Each time we do the approval process, the last modified date will be set to current time for you. This allows you to delete the files with something like the following in a shell(bash/cygwin/zshell)

```
find src/test/resources/approval -type f -mtime +7 -exec rm -i {} \;
```

this will all files that are located in src/test/resources/approval and were not modified in the last 7 days. The _-i_ flag on **rm** will make the whole process interactive.

1.4.2 I am getting Illegal state exteption with "<myclass> is not a primitive type class!"?

This means that you are trying to create/use an *Approval* object that's for a non primitive type and you haven't specified a *Converter*

1.4.3 Can I use the library in android?

No. Sadly the android framework is only stubbed while you are writing/compiling your code. The only way you can run the tests is on the device/emulator which doesn't allow us to use the reporters properly.

1.5 Javadoc

1.5.1 com.nikolavp.approval

Approval

public class **Approval**<T>

The main entry point class for each approval process. This is the main service class that is doing the hard work - it calls other classes for custom logic based on the object that is approved. Created by nikolavp on 1/29/14.

Parameters

- **<T>** – the type of the object that will be approved by this *Approval*

Constructors

Approval
Approval (*Reporter reporter*, *Converter*<T> *converter*, *PathMapper*<T> *pathMapper*)
Create a new object that will be able to approve "things" for you.

Parameters

- **reporter** – a reporter that will be notified as needed for approval events
- **converter** – a converter that will be responsible for converting the type for approval to raw form
- **pathMapper** – the path mapper that will be used

Approval
Approval (*Reporter reporter*, *Converter*<T> *converter*, *PathMapper*<T> *pathMapper*, com.nikolavp.*approval.utils.FileSystemUtils fileSystemReadWriter*)
This ctor is for testing only.

Methods

getApprovalPath
public static Path **getApprovalPath** (Path *filePath*)
Get the path for approval from the original file path.

Parameters

- **filePath** – the original path to value

Returns the path for approval

getConverter
Converter<T> **getConverter** ()

getPathMapper
PathMapper<T> **getPathMapper** ()

getReporter
Reporter **getReporter** ()

of
public static <T> *ApprovalBuilder*<T> **of** (Class<T> *clazz*)
Create a new approval builder that will be able to approve objects from the specified class type.

Parameters

- **clazz** – the class object for the things you will be approving
- **<T>** – the type of the objects you will be approving

Returns an approval builder that will be able to construct an `Approval` for your objects

verify
public void **verify**(T *value*, Path *filePath*)
Verify the value that was passed in.

Parameters

- **value** – the value object to be approved
- **filePath** – the path where the value will be kept for further approval

Approval.ApprovalBuilder

public static final class **ApprovalBuilder**<T>
A builder class for approvals. This is used to conveniently build new approvals for a specific type with custom reporters, converters, etc.

Parameters

- **<T>** – the type that will be approved by the the resulting approval object

Methods

build
public *Approval*<T> **build**()
Creates a new approval with configuration/options(reporters, converters, etc) that were set for this builder.

Returns a new approval for the specified type with custom configuration if any

withConveter
public *ApprovalBuilder*<T> **withConveter**(*Converter*<T> *converterToBeUsed*)
Set the converter that will be used when building new approvals with this builder.

Parameters

- **converterToBeUsed** – the converter that will be used from the approval that will be built

Returns the same builder for chaining

See also: *Converter*

withPathMapper
public *ApprovalBuilder*<T> **withPathMapper**(*PathMapper*<T> *pathMapperToBeUsed*)
Set a path mapper that will be used when building the path for approval results.

Parameters

- **pathMapperToBeUsed** – the path mapper

Returns the same builder for chaining

withReporter
public *ApprovalBuilder*<T> **withReporter**(*Reporter reporterToBeUsed*)
Set the reporter that will be used when building new approvals with this builder.

Parameters

- **reporterToBeUsed** – the reporter that will be used from the approval that will be built

Returns the same builder for chaninig

See also: *Reporter*

Approvals

public final class **Approvals**

Approvals for primitive types. This is a convenient static utility class that is the first thing to try when you want to use the library. If you happen to be lucky and need to verify only primitive types or array of primitive types then we got you covered.

User: nikolavp (Nikola Petrov) Date: 07/04/14 Time: 11:38

Methods

verify

public static void **verify**(int[] *ints*, Path *path*)

An overload for verifying int arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **ints** – the int array that needs to be verified
- **path** – the path in which to store the approval file

verify

public static void **verify**(byte[] *bytes*, Path *path*)

An overload for verifying byte arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **bytes** – the byte array that needs to be verified
- **path** – the path in which to store the approval file

verify

public static void **verify**(short[] *shorts*, Path *path*)

An overload for verifying short arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **shorts** – the short array that needs to be verified
- **path** – the path in which to store the approval file

verify

public static void **verify**(long[] *longs*, Path *path*)

An overload for verifying long arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **longs** – the long array that needs to be verified

- **path** – the path in which to store the approval file

verify
public static void **verify**(float[] *floats*, Path *path*)
An overload for verifying float arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **floats** – the float array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(double[] *doubles*, Path *path*)
An overload for verifying double arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **doubles** – the double array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(boolean[] *booleans*, Path *path*)
An overload for verifying boolean arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **booleans** – the boolean array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(char[] *chars*, Path *path*)
An overload for verifying char arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **chars** – the char array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(String[] *strings*, Path *path*)
An overload for verifying string arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **strings** – the string array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(byte *value*, Path *path*)
An overload for verifying a single byte value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the byte that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(short *value*, Path *path*)
An overload for verifying a single short value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the short that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(int *value*, Path *path*)
An overload for verifying a single int value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the int that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(long *value*, Path *path*)
An overload for verifying a single long value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the long that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(float *value*, Path *path*)
An overload for verifying a single float value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the float that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(double *value*, Path *path*)
An overload for verifying a single double value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the double that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(boolean *value*, Path *path*)
An overload for verifying a single boolean value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the boolean that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(char *value*, Path *path*)
An overload for verifying a single char value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the char that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(String *value*, Path *path*)
An overload for verifying a single String value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the String that needs to be verified
- **path** – the path in which to store the approval file

FullPathMapper

public interface **FullPathMapper**<T>
A mapper that unlike *PathMapper* doesn't resolve the approval file path based on a given sub path but only needs the value. Of course there are possible implementations that don'n even need the value like *com.nikolavp.approval.pathmappers.JunitPathMapper*.

Parameters

- **<T>** – the value that will be approved

See also: *PathMapper*

Methods

getApprovalPath
Path **getApprovalPath**(T *value*)
Get the full approval path based on the value.

Parameters

- **value** – the value that will be approved and for which the approval path will be built

Returns a Path for the given value

PathMapper

public interface **PathMapper**<T>
An interface representing objects that will return an appropriate path for the approval process. Most of the times those are used because you don't want to repeat yourself with the same parent path in `com.nikolavp.approval.Approval.verify(Object,java.nio.file.Path)` for the path argument. This will map your approval results file from the value for approval and a possible sub path.

Parameters

- **<T>** – the value that will be approved

See also: *com.nikolavp.approval.pathmappers.ParentPathMapper*

Methods

getPath
Path **getPath**(T *value*, Path *approvalFilePath*)
Gets the path for the approval result based on the value that we want to approve and a sub path for that.

Parameters

- **value** – the value that will be approved
- **approvalFilePath** – a name/subpath for the approval. This will be the path that was passed to *Approval.verify(Object,java.nio.file.Path)*

Returns the full path for the approval result

Pre

public final class **Pre**
Pre conditions exceptions.

Methods

notNull
public static void **notNull**(Object *value*, String *name*)
Verify that a value is not null.

Parameters

- **value** – the value to verify
- **name** – the name of the value that will be used in the exception message.

Reporter

public interface **Reporter**
Created by nikolavp on 1/30/14.

Methods

approveNew
void **approveNew**(byte[] *value*, File *fileForApproval*, File *fileForVerification*)
Called by an *com.nikolavp.approval.Approval* object when a value for verification is produced but no old.

Parameters

- **value** – the new value that came from the verification
- **fileForApproval** – the approval file(this contains the value that was passed in)
- **fileForVerification** – the file for the this new approval value @return true if the new value is approved and false otherwise

canApprove
boolean **canApprove**(File *fileForApproval*)
A method to check if this reporter is supported for the following file type or environment! Reporters are different for different platforms and file types and this in conjuction with `com.nikolavp.approval.reporters.Reporters.firstWorking` will allow you to plug different reporters for different environments(CI, Windows, Linux, MacOS, etc).

Parameters

- **fileForApproval** – the file that we want to approve

Returns true if we can approve the file and false otherwise

notTheSame
void **notTheSame**(byte[] *oldValue*, File *fileForVerification*, byte[] *newValue*, File *fileForApproval*)
Called by an *com.nikolavp.approval.Approval* object when values don't match in the approval process.

Parameters

- **oldValue** – the old value that was found in fileForVerification from old runs
- **newValue** – the new value that was passed for verification
- **fileForVerification** – the file for this approval value
- **fileForApproval** – the file for the new content

1.5.2 com.nikolavp.approval.converters

AbstractConverter

public abstract class **AbstractConverter**<T> implements *Converter*<T>
An abstract class for the Converter interface. All external converters are advised to subclass this class.

Parameters

- **<T>** – the type you want to convert

AbstractStringConverter

public abstract class **AbstractStringConverter**<T> extends *AbstractConverter*<T>

A convenient abstract converter to handle object approvals on string representable objects.

Parameters

- **<T>** – the type you want to convert

Methods

getRawForm

public final byte[] **getRawForm**(T *value*)

getStringForm

protected abstract String **getStringForm**(T *value*)

Gets the string representation of the type object. This representation will be written in the files you are going to then use in the approval process.

Parameters

- **value** – the object that you want to convert

Returns the string representation of the object

ArrayConverter

public class **ArrayConverter**<T> extends *AbstractStringConverter*<T[]>

An array converter that uses another converter for it's items. This allows this converter to be composed with another one and allow you to convert your types even if they are in an array. User: nikolavp Date: 20/03/14 Time: 19:34

Parameters

- **<T>** – The type of the items in the list that this converter accepts

Constructors

ArrayConverter

public **ArrayConverter**(*Converter*<T> *typeConverter*)

Creates an array converter that will use the other converter for it's items and just make array structure human readable.

Parameters

- **typeConverter** – the converters for the items in the array

Methods

getStringForm

protected String **getStringForm**(T[] *values*)

Converter

public interface **Converter**<T>

A converter interface. Converters are the objects in the approval system that convert your object to their raw form that can be written to the files. Note that the raw form is not always a string representation of the object. If for example your object is an image. User: nikolavp Date: 28/02/14 Time: 14:47

Parameters

- **<T>** – the type you are going to convert to raw form

Methods

getRawForm

byte[] **getRawForm**(T *value*)

Gets the raw representation of the type object. This representation will be written in the files you are going to then use in the approval process.

Parameters

- **value** – the object that you want to convert

Returns the raw representation of the object

Converters

public final class **Converters**

Converters for primitive types. Most of these just call toString on the passed object and then get the raw representation of the string result. . User: nikolavp Date: 28/02/14 Time: 17:25

Fields

BOOLEAN

public static final *Converter*<Boolean> **BOOLEAN**

A converter for the primitive or wrapper boolean object.

BOOLEAN_ARRAY

public static final *Converter*<boolean[]> **BOOLEAN_ARRAY**

A converter for the primitive boolean arrays.

BYTE

public static final *Converter*<Byte> **BYTE**

A converter for the primitive or wrapper byte types.

BYTE_ARRAY

public static final *Converter*<byte[]> **BYTE_ARRAY**

A converter for the primitive byte arrays.

CHAR

public static final *Converter*<Character> **CHAR**

A converter for the primitive or wrapper char object.

CHAR_ARRAY
public static final *Converter*<char[]> **CHAR_ARRAY**
A converter for the primitive char arrays.

DOUBLE
public static final *Converter*<Double> **DOUBLE**
A converter for the primitive or wrapper double object.

DOUBLE_ARRAY
public static final *Converter*<double[]> **DOUBLE_ARRAY**
A converter for the primitive double arrays.

FLOAT
public static final *Converter*<Float> **FLOAT**
A converter for the primitive or wrapper float object.

FLOAT_ARRAY
public static final *Converter*<float[]> **FLOAT_ARRAY**
A converter for the primitive float arrays.

INTEGER
public static final *Converter*<Integer> **INTEGER**
A converter for the primitive or wrapper int object.

INTEGER_ARRAY
public static final *Converter*<int[]> **INTEGER_ARRAY**
A converter for the primitive int arrays.

LONG
public static final *Converter*<Long> **LONG**
A converter for the primitive or wrapper long object.

LONG_ARRAY
public static final *Converter*<long[]> **LONG_ARRAY**
A converter for the primitive long arrays.

SHORT
public static final *Converter*<Short> **SHORT**
A converter for the primitive or wrapper short object.

SHORT_ARRAY
public static final *Converter*<short[]> **SHORT_ARRAY**
A converter for the primitive short arrays.

STRING
public static final *Converter*<String> **STRING**
A converter for the String object.

STRING_ARRAY
public static final *Converter*<String[]> **STRING_ARRAY**
A converter for an array of strings.

Methods

of
static <T> *Converter*<T> **of**()

ofArray
static <T> *Converter*<T> **ofArray**()

DefaultConverter

public class **DefaultConverter** implements *Converter*<byte[]>
Just a simple converter for byte array primitives. We might want to move this into *Converters*. User: nikolavp Date: 28/02/14 Time: 14:54

Methods

getRawForm
public byte[] **getRawForm**(byte[] *value*)

ListConverter

public class **ListConverter**<T> extends *AbstractStringConverter*<List<T>>
A list converter that uses another converter for it's items. This allows this converter to be composed with another one and allow you to convert your types even if they are in a list. User: nikolavp Date: 28/02/14 Time: 17:47

Parameters

- **<T>** – The type of the items in the list that this converter accepts

Constructors

ListConverter
public **ListConverter**(*Converter*<T> *typeConverter*)
Creates a list converter that will use the other converter for it's items and just make list structure human readable.

Parameters

- **typeConverter** – the converters for the items

Methods

getStringForm
protected String **getStringForm**(List<T> *values*)

ReflectiveBeanConverter

public class **ReflectiveBeanConverter**<T> extends *AbstractStringConverter*<T>

A converter that accepts a bean object and uses reflection to introspect the fields of the bean and builds a raw form of them. Note that the fields must have a human readable string representation for this converter to work properly. User: nikolavp Date: 28/02/14 Time: 15:12

Parameters

- **<T>** – the type of objects you want convert to it's raw form

Methods

getStringForm
protected String **getStringForm**(T *value*)

1.5.3 com.nikolavp.approval.pathmappers

JunitPathMapper

public class **JunitPathMapper** implements TestRule, *PathMapper*, *FullPathMapper*

A path mapper that have to be declared as a `org.junit.Rule` and will use the standard junit mechanisms to put your approval results in $package-name-with-slashes/$classname/$methodname.

This class can be used as a *`com.nikolavp.approval.PathMapper`* in which case it will put your approval results in that directory or you can use it as a *`com.nikolavp.approval.FullPathMapper`* in which case your approval result for the **single virifacion** will be put in a file with that path. In the latter case you will have to make sure that there aren't two approvals for a single test method.

Constructors

JunitPathMapper
public **JunitPathMapper**(Path *parentPath*)

A parent path in which you want to put your approvals if any.

Parameters

- **parentPath** – the parent path

Methods

apply
public Statement **apply**(Statement *base*, Description *description*)

getApprovalPath
public Path **getApprovalPath**(Object *value*)

getCurrentTestPath
Path **getCurrentTestPath**()

getPath
public Path **getPath**(Object *value*, Path *approvalFilePath*)

ParentPathMapper

public class **ParentPathMapper**<T> implements *PathMapper*<T>

A path mapper that will put all approvals in a common parent path. Let's say you want to put all your approval results in **src/test/resources/approval**(we assume a common maven directory structure) then you can use this mapper as follows:

```
Approval approval = Approval.of(String.class)
 .withPathMapper(new ParentPathMapper(Paths.get("src/test/resources/approval")))
 .build();
```

now the following call

```
approval.verify("Some cool string value", Paths.get("some_cool_value.txt");
```

will put the approved value in the file **src/test/resources/approval/some_cool_value.txt**

Parameters

- **<T>** – the value that will be approved

Constructors

ParentPathMapper
public **ParentPathMapper**(Path *parentPath*)

Creates a parent path mapper that puts approvals in the given parent path.

Parameters

- **parentPath** – the parent path for all approvals

Methods

getPath
public Path **getPath**(T *value*, Path *approvalFilePath*)

1.5.4 com.nikolavp.approval.reporters

AbstractReporter

public abstract class **AbstractReporter** implements *Reporter*

An abstract class for the Reporter interface. All external reporters are advised to subclass this class.

ExecutableDifferenceReporter

public class **ExecutableDifferenceReporter** implements *Reporter*

A reporter that will shell out to an executable that is presented on the user's machine to verify the test output. Note that the approval command and the difference commands can be the same.

•approval command will be used for the first approval

- the difference command will be used when there is already a verified file but it is not the same as the value from the user

Constructors

ExecutableDifferenceReporter
public **`ExecutableDifferenceReporter`**(String *approvalCommand*, String *diffCommand*)
Main constructor for the executable reporter.

Parameters

- **`approvalCommand`** – the approval command
- **`diffCommand`** – the difference command

Methods

approveNew
public void **`approveNew`**(byte[] *value*, File *approvalDestination*, File *fileForVerification*)

buildApproveNewCommand
protected String[] **`buildApproveNewCommand`**(File *approvalDestination*, File *fileForVerification*)

buildCommandline
static List<String> **`buildCommandline`**(String... *cmdParts*)

buildNotTheSameCommand
protected String[] **`buildNotTheSameCommand`**(File *fileForVerification*, File *fileForApproval*)

canApprove
public boolean **`canApprove`**(File *fileForApproval*)

getApprovalCommand
protected String **`getApprovalCommand`**()

getDiffCommand
protected String **`getDiffCommand`**()

notTheSame
public void **`notTheSame`**(byte[] *oldValue*, File *fileForVerification*, byte[] *newValue*, File *fileForApproval*)

runProcess
public static Process **`runProcess`**(String... *cmdParts*)
Execute a command with the following arguments.

Parameters

- **`cmdParts`** – the command parts

Throws

- **IOException** – if there were any I/O errors

Returns the process for the command that was started

startProcess
Process **startProcess**(String... *cmdParts*)

FirstWorkingReporter

class **FirstWorkingReporter** implements *Reporter*
A reporter that will compose other reporters and use the first one that can approve the objects for verification as per `com.nikolavp.approval.Reporter.canApprove(java.io.File)`.

Constructors

FirstWorkingReporter
FirstWorkingReporter(*Reporter*... *others*)

Methods

approveNew
public void **approveNew**(byte[] *value*, File *fileForApproval*, File *fileForVerification*)

canApprove
public boolean **canApprove**(File *fileForApproval*)

notTheSame
public void **notTheSame**(byte[] *oldValue*, File *fileForVerification*, byte[] *newValue*, File *fileForApproval*)

MacOSReporters

public final class **MacOSReporters**
Reporters that use macOS specific binaries, i.e. not cross platform programs.

If you are looking for something cross platform like gvim, emacs, you better look in *com.nikolavp.approval.reporters.Reporters*.

Methods

diffMerge
public static *Reporter* **diffMerge**()
A reporter that calls diffmerge to show you the results.

Returns a reporter that calls diffmerge

ksdiff
public static *Reporter* **ksdiff**()
A reporter that calls ksdiff to show you the results.

Returns a reporter that calls ksdiff

p4merge
public static *Reporter* **p4merge**()
A reporter that calls p4merge to show you the results.

Returns a reporter that calls p4merge

tkdiff
public static *Reporter* **tkdiff**()
A reporter that calls tkdiff to show you the results.

Returns a reporter that calls tkdiff

Reporters

public final class **Reporters**
Created with IntelliJ IDEA. User: nikolavp Date: 10/02/14 Time: 15:10 To change this template use File | Settings | File Templates.

Methods

console
public static *Reporter* **console**()
Creates a simple reporter that will print/report approvals to the console. This reporter will use convenient command line tools under the hood to only report the changes in finds. This is perfect for batch modes or when you run your build in a CI server

Returns a reporter that uses console unix tools under the hood

fileLauncher
public static *Reporter* **fileLauncher**()
A reporter that launches the file under test. This is useful if you for example are generating an spreadsheet and want to verify it.

Returns a reporter that launches the file

firstWorking
public static *Reporter* **firstWorking**(*Reporter... others*)
Get a reporter that will use the first working reporter as per `com.nikolavp.approval.Reporter.canApprove` for the reporting.

Parameters

- **others** – an array/list of reporters that will be used

Returns the newly created composite reporter

gedit
public static *Reporter* **gedit**()
Creates a reporter that uses gedit under the hood for approving.

Returns a reporter that uses gedit under the hood

gvim
public static *Reporter* **gvim**()

Creates a convenient gvim reporter. This one will use gvimdiff for difference detection and gvim for approving new files. The proper way to exit vim and not approve the new changes is with ":cq" - just have that in mind!

Returns a reporter that uses vim under the hood

imageMagick
public static *Reporter* **imageMagick**()

A reporter that compares images. Currently this uses imagemagick for comparison. If you only want to view the new image on first approval and when there is a difference, then you better use the `fileLauncher()` reporter which will do this for you.

Returns the reporter that uses ImagemMagick for comparison

SwingInteractiveReporter

public class **SwingInteractiveReporter** implements *Reporter*

A reporter that can wrap another reporter and give you a prompt to approve the new result value.

This is useful for reporters that cannot give you a clear way to create the result file. If for example you are using a reporter that only shows you the resulting value but you cannot move it to the proper result file for the approval.

If you say OK/YES on the prompt then the result will be written to the proper file for the next approval time.

Constructors

SwingInteractiveReporter
SwingInteractiveReporter(*Reporter other*, *FileSystemUtils fileSystemReadWriter*)

Methods

approveNew
public void **approveNew**(byte[] *value*, File *fileForApproval*, File *fileForVerification*)

canApprove
public boolean **canApprove**(File *fileForApproval*)

isHeadless
boolean **isHeadless**()

notTheSame
public void **notTheSame**(byte[] *oldValue*, File *fileForVerification*, byte[] *newValue*, File *fileForApproval*)

promptUser
int **promptUser**()

wrap
public static *SwingInteractiveReporter* **wrap**(*Reporter reporter*)
Wrap another reporter.

Parameters

- **reporter** – the other reporter

Returns a new reporter that call the other reporter and then propmts the user

WindowsReporters

public final class **WindowsReporters**
Reporters that use windows specific binaries, i.e. the programs that are used are not cross platform.

If you are looking for something cross platform like gvim, emacs, you better look in *com.nikolavp.approval.reporters.Reporters*.

Methods

beyondCompare
public static *Reporter* **beyondCompare**()
A reporter that calls Beyond Compare 3 to show you the results.

Returns a reporter that calls beyond compare

notepadPlusPlus
public static *Reporter* **notepadPlusPlus**()
A reporter that calls notepad++ to show you the results.

Returns a reporter that calls notepad++

tortoiseImage
public static *Reporter* **tortoiseImage**()
A reporter that calls TortoiseIDiff to show you the results.

Returns a reporter that calls tortoise image difference tool

tortoiseText
public static *Reporter* **tortoiseText**()
A reporter that calls TortoiseMerge to show you the results.

Returns a reporter that calls tortoise difference tool for text

winMerge
public static *Reporter* **winMerge**()
A reporter that calls WinMerge to show you the results.

Returns a reporter that calls win merge

WindowsReporters.WindowsExecutableReporter

public static class **WindowsExecutableReporter** extends *ExecutableDifferenceReporter*
Windows executable reporters should use this class instead of the more general ExecutableDifferenceReporter.

Constructors

WindowsExecutableReporter
public **`WindowsExecutableReporter`**(String *approvalCommand*, String *diffCommand*)
Main constructor for the executable reporter.

Parameters

- **`approvalCommand`** – the approval command
- **`diffCommand`** – the difference command

Methods

canApprove
public boolean **`canApprove`**(File *fileForApproval*)

1.5.5 com.nikolavp.approval.utils

CrossPlatformCommand

public abstract class **`CrossPlatformCommand`**<T>
A command that when run will execute the proper method for the specified operating system. This is especially useful when you are trying to create for handling different platforms. Here is an example usage of the class:

```
final Boolean result = new CrossPlatformCommand<Boolean>() {
    &#064;Override protected Boolean onWindows() {
    //do your logic for windows
    }

    &#064;Override protected Boolean onUnix() {
    //do your logic for unix
    }

    &#064;Override protected Boolean onMac() {
    //do your logic for mac
    }

    &#064;Override protected Boolean onSolaris() {
    //do your logic for solaris
    }
}.execute();
```

Parameters

- **`<T>`** – the result from the command.

Methods

execute
public T **`execute`**()
Main method that should be executed. This will return the proper result depending on your platform.

Returns the result

isMac
public static boolean **isMac**()
Check if the current OS is MacOS.

Returns true if macOS and false otherwise

isSolaris
public static boolean **isSolaris**()
Check if the current OS is Solaris.

Returns true if solaris and false otherwise

isUnix
public static boolean **isUnix**()
Check if the current OS is some sort of unix.

Returns true if unix and false otherwise

isWindows
public static boolean **isWindows**()
Check if the current OS is windows.

Returns true if windows and false otherwise

onMac
protected T **onMac**()
What to execute on macOS.

Returns the result

onSolaris
protected T **onSolaris**()
What to execute on solaris.

Returns the result

onUnix
protected abstract T **onUnix**()
What to execute on windows.

Returns the result

onWindows
protected abstract T **onWindows**()
What to execute on windows.

Returns the result

setOS
static void **setOS**(String *newOs*)

DefaultFileSystemUtils

public class **DefaultFileSystemUtils** implements com.nikolavp.*approval.utils.FileSystemUtils*
A default implementation for *com.nikolavp.approval.utils.FileSystemUtils*. This one just delegates to methods in Files. User: nikolavp Date: 27/02/14 Time: 12:26

Methods

createDirectories
public void **createDirectories** (File *directory*)

move
public void **move** (Path *path*, Path *filePath*)

readFully
public byte[] **readFully** (Path *path*)

touch
public void **touch** (Path *pathToCreate*)

write
public void **write** (Path *path*, byte[] *value*)

FileSystemUtils

public interface **FileSystemUtils**
This class is mostly used for indirection in the tests. We just don't like static utility classes. Created by ontotext on 2/2/14.

Methods

createDirectories
void **createDirectories** (File *directory*)
Create a directory and their parent directories as needed.

Parameters

- **directory** – the directory to create

Throws

- **IOException** – if there was an error while creating the directories

move
void **move** (Path *path*, Path *filePath*)
Move a path to another path.

Parameters

- **path** – the source
- **filePath** – the destination

Throws

- **IOException** – if there was an error while moving the paths

readFully

byte[] **readFully**(Path *path*)

Read the specified path as byte array.

Parameters

- **path** – the path to read

Throws

- **IOException** – if there was an error while reading the content

Returns the path content

touch

void **touch**(Path *pathToCreate*)

Creates the specified path with empty content.

Parameters

- **pathToCreate** – the path to create

Throws

- **IOException** – if there was error while creating the path

write

void **write**(Path *path*, byte[] *value*)

Write the byte value to the specified path.

Parameters

- **path** – the path
- **value** – the value

Throws

- **IOException** – if there was an error while writing the content

A

B

C

D

E

F

G