SWE - 261P : Software Testing and Debugging

Project Report : Jenkins

Members:

1) Tanisha Arun Agarwal. UCI ID : 35968197

2) Dheeraj Kumar Mohan Kumar. UCI ID : 11369792

3) Neha Pradeep Patil. UCI ID : 61609148

TABLE OF CONTENTS

1. Introduction:

Jenkins is a Java-based open source automation server. It offers more than 1,800 plugins to help automating almost anything, allowing people to spend their time on tasks that machines can't perform. Jenkins is a tool that is primarily used to automate repetitive processes, saving time, and improving the development workflow. Jenkins is commonly used for:

- Building projects
- Running tests to detect bugs and other issues as soon as they are introduced
- Static code analysis
- Deployment

2. Part 1 : Building the Project:

Clone the forked repository into your local system after forking the repository on GitHub. Install the necessary development tools, namely, JDK 11 or 17, Apache Maven 3.8.1 or above, IDE that supports Maven projects, and Node.js. [2]

Run the following command in the terminal to build jenkins:

```
mvn -am -pl war,bom -DskipTests -Dspotbugs.skip clean install
```

To launch a development instance, after the above command, run:

```
mvn -pl war jetty:run
```

To build the frontend assets: On one terminal, start a development server that will not process frontend assets:

```
mvn -pl war jetty:run -Dskip.yarn
```

On another terminal, move to the war folder and start a webpack dev server:

```
cd war; yarn start
```

We can also create our own plugin and run that particular plugin using the following commands,

To create a plugin run:

```
mvn -U archetype:generate -Dfilter="io.jenkins.archetypes:"
```

The Maven HPI Plugin is used to build and package Jenkins plugins. It also provides a convenient way to run a Jenkins instance with your plugin:

```
mvn hpi:run
```

A Jenkins instance will be created and accessible at http://localhost:8080/jenkins/. Wait for the subsequent console output before launching a web browser to examine the plugin's functionality.

3. Part 2: Test framework Used

The testing framework is Junits. The tests that are currently present in the project can be run without requiring any further preparation. Every project feature is tested in the corresponding test classes, and the function of the test begins with the annotation Test, which is immediately executable from the IDE. Since the goals of Jenkins are to develop, analyze, and deploy the project, all of these features are tested in the tests. Separate classes are present in the test folder to test the functionality of the project's corresponding classes. For instance,

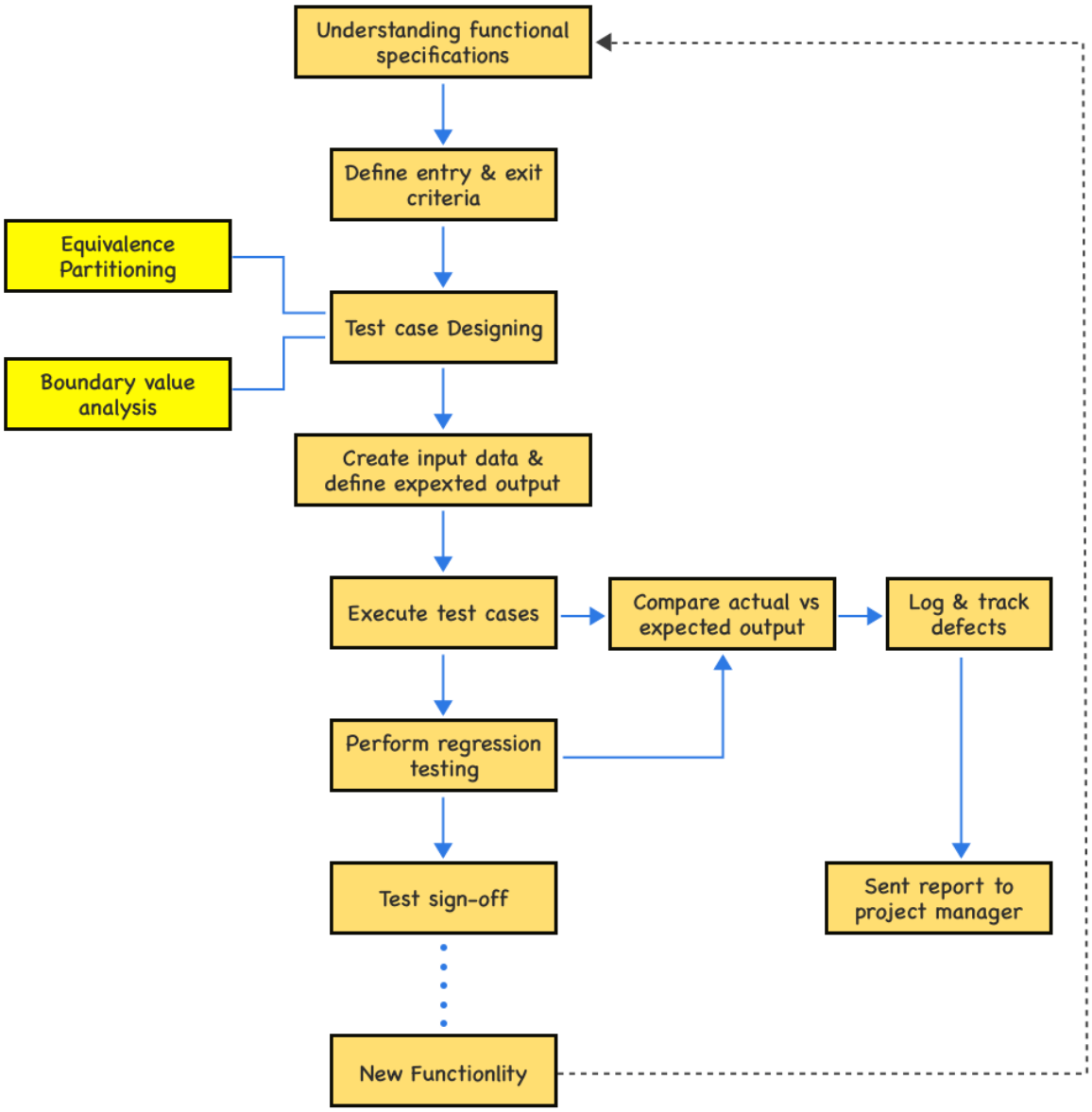
In this test class, ControllerExecutorsAgentsTest, is written to test the functionalities like to test if the agents is activated or not after calling functions, to deactivate by calling the function, setNumExecutores(0) or to check if it is active by creating the slave.Likewise, the other functionalities of the project are tested.

4. Part 3 : Functional Testing and Partitioning

3.1 Functional Testing :

The desired behavior of a program is described in a functional specification, which is different from the program itself. The functional specification is the most crucial source of information for creating tests, regardless of its formality or informality. Functional testing refers to a group of procedures used to create test case requirements from program specifications.
What test cases should I use to exercise my program? is a question that functional testing, or more specifically functional test case design, aims to address by solely taking into account a program's specification and not its design or implementation structure. Functional testing is sometimes known as specification-based or "black-box" testing since it is focused on program requirements rather than the internals of the code.
Process of deriving tests from the requirement specifications.

- The standard method for creating test cases can start with the requirements specification, continue through every stage of design, and end with implementation.
- Basis of verification - builds evidence that the implementation conforms to its specification.
- Effective at finding some classes of faults that elude code-based techniques. i.e., incorrect outcomes and missing functionality.



- *3.1 (i) The main steps of a systematic approach to functional program testing.*

3.2 Partitioning

Functional testing is based on the idea of partitioning.

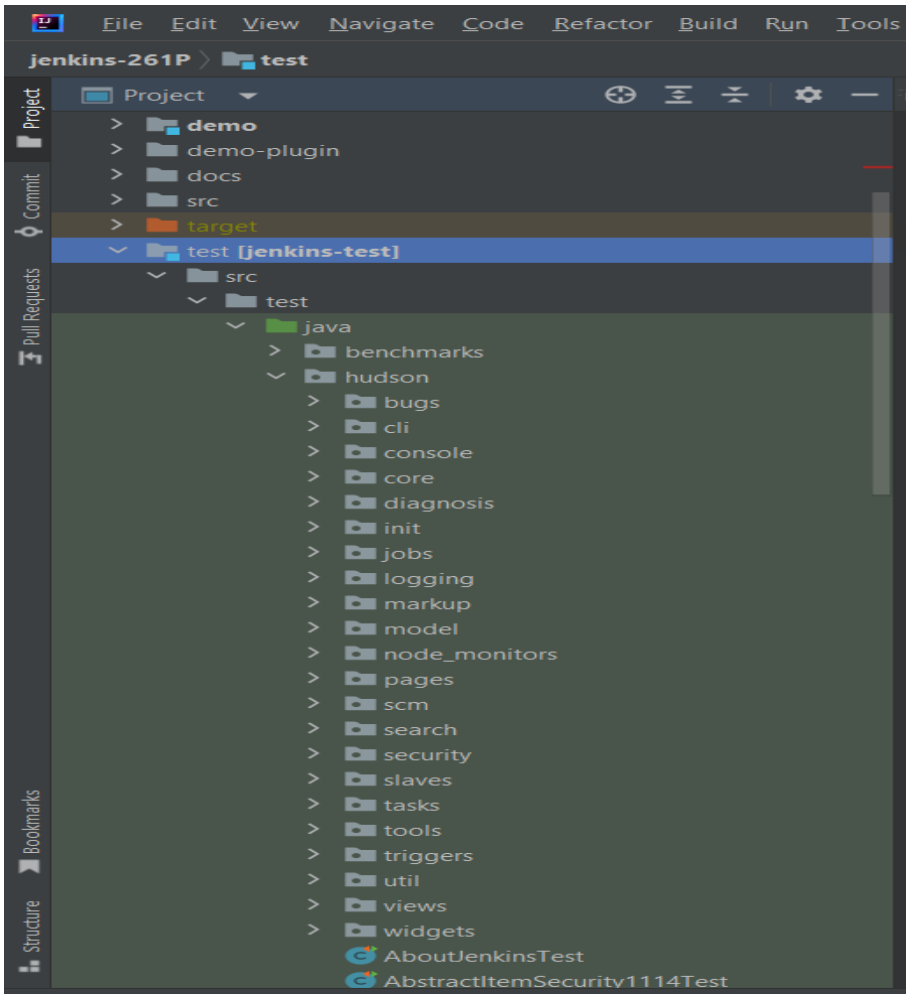
- ○ You can't test individual requirements in isolation.
- ○ Instead, we need to partition the specification and software into features that can be tested.

With few exceptions, the number of possible test cases for a particular program is unfathomably massive; it is so large that it can be practically regarded as infinite. For instance, even a

straightforward function with two 32-bit integer input arguments has valid inputs. Budgets and timelines, in contrast to input spaces, are finite, therefore any feasible testing method must pick an incredibly small subset of the entire input space.
A partition testing method is one that divides the unlimited pool of potential test cases into a limited number of classes with the intention of selecting one or more test cases from each class. Functional testing, or specification-based partition testing, is the term used when partitions are selected based on information in the specification rather than the design or implementation. Partition the outputs of a feature into the possible outcomes and the inputs, by what outcomes they cause (or other potential groupings).

3.3 Partition in the Current System

- Currently in the project the Source code is divided into two major parts Jenkins Core and Cli.
- Each of them are further subdivided into multiple classes each written for a particular functionality.
- The testing is also done in the same way to find possible input spaces for each functionality.



Model of Jenkins is an interesting feature, the whole model is further divided into labels, listeners, queues and abstract features.
One such feature is User, here the whole user representation is done. New user creation, delete user, update user, get user details and other functionality is handled in this class.

The tests are written in UserTest.java file, We can see that the input space can be divided into validating users as existing and non-existent users. The testing of this feature is formed on certain requirements.

As we know functional testing is modeled based on requirements
We can see the existing Test file following the same. Questions

- Does the user exist?
- What if the user does not exist?Can we add a new User?
- Can we Update his/her information if it is already present?
- Can the user be deleted? Etc..

Are the basis of the UserTest.java.

The entire user interface (UI), which includes the user name, user avatar (display photos), and fetching default avatar, is partitioned.
Also, extremely simple and tiny sample sizes are used for functionality like testing the getUserData. Checking for case sensitivity in user names, making sure user names are unique, checking the security realm for this, making sure users don't share email addresses, etc.

We are further partitioning the test scenarios to check for null email and null user names. Can the system validate null cases?
Can we add a user with name null or email null. Fetching user details by parsing null as user id, invalid user etc.

1. **testGetUserInvalidName() ->** This method checks if the invalid user name can be updated and compared with existing user records with the same name, they should not have the same ID.

```
no usages  new *
@Test
public void testGetUserInvalidName() throws Exception {
    {
        User user = User.get( idOrFullName: "%xJU", create: false,Collections.emptyMap());
        assertNull( message: "User %xJU should not be created.", user);
    }
    j.jenkins.reload();
    {
        User user2 = User.get("John Smith");
        user2.setFullName("Alice Smith");
        assertEquals( message: "Users should have same id even after editing the name",  expected: "John Smith", user2.getId());
        User user4 = User.get( idOrFullName: "Marie",  create: false, Collections.EMPTY_MAP);
        assertNull( message: "User should not be created because Marie does not exists.", user4);
    }
}
```

2. **testGetAndGetAllForEmail() ->**
   This method checks if we are able to create a user by giving only an email address. The checks are done for handling the create user boolean values true and false. If the create user boolean is false, the system should not create a new user based on the given email and vice-versa.

```
no usages  new *
@Test
public void testGetAndGetAllForEmail() {
    {
        User user = User.get( idOrFullName: "john.smith@test.org",  create: false, Collections.emptyMap());
        assertNull( message: "User john.smith@test.org should not be created.", user);
        assertFalse( message: "Jenkins should not contain user john.smith@test.org.", User.getAll().contains(user));
    }
    {
        User user2 = User.get( idOrFullName: "john.rambo.smith@test.org",  create: true, Collections.emptyMap());
        assertNotNull( message: "User with email john.rambo.smith@test.org should be created.", user2);
        assertTrue( message: "Jenkins should contain user john.rambo.smith@test.org", User.getAll().contains(user2));
    }
    {
        // checking if it can get the existing user
        User user = User.get( idOrFullName: "john.rambo.smith@test.org",  create: false, Collections.emptyMap());
        assertNotNull( message: "User with email john.rambo.smith@test.org should be created.", user);
        assertTrue( message: "Jenkins should contain user john.rambo.smith@test.org", User.getAll().contains(user));
    }

}
```

5. Part 4 : Finite State Machines

A finite state machine, often known as an FSM, is an abstract illustration of the behavior that some systems display. It is common practice to define sequences of interactions between a system and its surroundings using finite state machines. State-machine requirements can be employed in the creation of an oracle that determines if each observed behavior is correct as well as to direct the choice of tests.

Application requirements are used to create an FSM. An FSM could be used, for instance, to simulate a network protocol. Not all criteria for an application are given by an FSM. An FSM cannot specify, for instance, real-time needs or performance requirements.

4.1 Requirements Specification or Design Specification

An FSM could be used as a design artifact that directs how an application should be developed or as a specification of the desired behavior.
– The role assigned to an FSM depends on whether it is a part of the requirements specification or of the design specification.

4.2 Finite State Machines With Output

There are 2 FSM widely used

- Mealy Machine (due to G. H. Mealy -1955 publication) – Outputs correspond to transitions between states.
- Moore Machine (due to E. F. Moore -1956 publication) – Outputs are determined only by the states.

4.3 Test Generations Using FSM

Although there are numerous methods for producing test cases from finite state machines, the most are variations on the fundamental tactic of examining each state change. Consider that each transition is essentially a specification of a precondition and postcondition, for example, a transition from state to state on stimulus means "if the system is in state S and receives stimulus I then after reacting it will be in state T." This is one way to understand this fundamental strategy.

Each of these precondition, postcondition combinations needs to be tested because a flawed system could violate any of them. If system states are directly observable or must be inferred from stimulus/response sequences, whether the state machine specification is complete as given

or includes additional, implicit transitions, and whether the size of the (possibly augmented) state machine is modest or very large are all factors that affect the specifics of the approach taken.

Transition coverage, which demands that each transition be traveled through at least once, is a fundamental condition for producing test cases from finite state machines. Frequently, collections of state sequences or transition sequences are provided as test case specifications for transition coverage. The "relevant" states can all be represented by a state machine model, which is adequate. Single state path, single transition path, and boundary interior loop coverage are among the coverage requirements intended to handle history sensitivity.

single state path coverage - The single state path coverage criterion mandates the exercise of every path that only crosses a state once.
single transition path coverage - This criterion mandates that each path that only occasionally crosses a transition be used.
Each different state machine loop must be executed a minimum, an intermediate, and a maximum number of times in order to satisfy the border interior loop coverage condition.

6. Part 5 : Feature for implementing FSM

Create Job : Creates a new job by reading stdin as a configuration XML file.

```java
@Extension
public class CreateJobCommand extends CLICommand {
    @Override
    public String getShortDescription() {
        return Messages.CreateJobCommand_ShortDescription();
    }

    @Argument(metaVar = "NAME", usage = "Name of the job to create", required = true)
    public String name;

    @Override
    protected int run() throws Exception {
        Jenkins h = Jenkins.get();

        if (h.getItemByFullName(name) != null) {
            throw new IllegalStateException("Job '" + name + "' already exists");
        }

        ModifiableTopLevelItemGroup ig = h;
        int i = name.lastIndexOf('/');
        if (i > 0) {
            String group = name.substring(0, i);
            Item item = h.getItemByFullName(group);
            if (item == null) {
                throw new IllegalArgumentException("Unknown ItemGroup " + group);
            }

            if (item instanceof ModifiableTopLevelItemGroup) {
                ig = (ModifiableTopLevelItemGroup) item;
            } else {
                throw new IllegalStateException("Can't create job from CLI in " + group);
            }
            name = name.substring(i + 1);
        }

        Jenkins.checkGoodName(name);
        ig.createProjectFromXML(name, stdin);
        return 0;
    }
}
```

5.1 : Use case definitions

Entry

State 1 : It takes the Job name as the input argument.(Name of the job to create). This is a required parameter. If found null throw error / return to previous state.
State 2 : Once the required parameter is available, validate the parameters. Get the Jenkins current state and set of functionalities. Current Jenkins state would be current jobs that are already present, the jobs running currently, user details and info etc.,
State 3 : Now check if there is no job already using the same name, if so throw an error / return to previous state 1.
State 4 : create an instance of management Interface.
State 5 : get the last Index of job name after '/'.
State 6 : check if index is greater than 0. If true go to State 7 else go to State 13.
State 7 : get the last substring till that i^{th} value and assign to a new string.
State 8 : Now use that name and get the item description, full names are like path names, where each name of {Item} is combined by '/'. This returns null or not an instance of the given type.
State 9 : If the returned value is null, throw error/ return to State1.
State 10 : Check if this is modifiable.
State 11 : If so, create an instance of modifiable or throw error/ return to State1.
State 12 : Now the name is altered to begin from the i^{th}+1 value.
State 13 : Now validate the name by parsing it to "*checkGoodName(name)" ,* function.
This function checks if, ***the given name is suitable as a name for a view or Job etc.,***
this throws error/failure if the given name is not good.
Which sets the current state of the system to State 1.
State 14 : Create a project from the given XML name and the standard inputs sent.
Exit

5.2 : State Table Representation

No.	Use Cases	True	False	Current state	Next state
1	Get the user Inputs.	1	0	S0	S1
		0	1	S0	S0
2	Get current instance of Jenkins	1	0	S1	S2
		0	1	S1	S0
3	Validate no current job has same name	1	0	S2	S3
		0	1	S2	S0
4	Create interface instance	1	0	S3	S4
5	Get required string index to validate	1	0	S4	S5
6	Check if index > 0	1	0	S5	S6
		0	1	S5	S12
7	Get last substring, assign it as new string	1	0	S6	S7
8	Get the description using current name string	1	0	S7	S8
9	Validate description for null values	1	0	S8	S9
		0	1	S8	S0
10	Check if this instance is modifiable	1	0	S9	S10
		0	1	S9	S0
11	Create a modifiable Instance	1	0	S10	S11
12	Alter name String	1	0	S11	S12
13	Validate name for length and other conditions	1	0	S12	S13
		0	1	S12	S0
14	Create a new Jenkins job with a given name.	1	0	S13	exit

7. Part 6. FSM Diagram

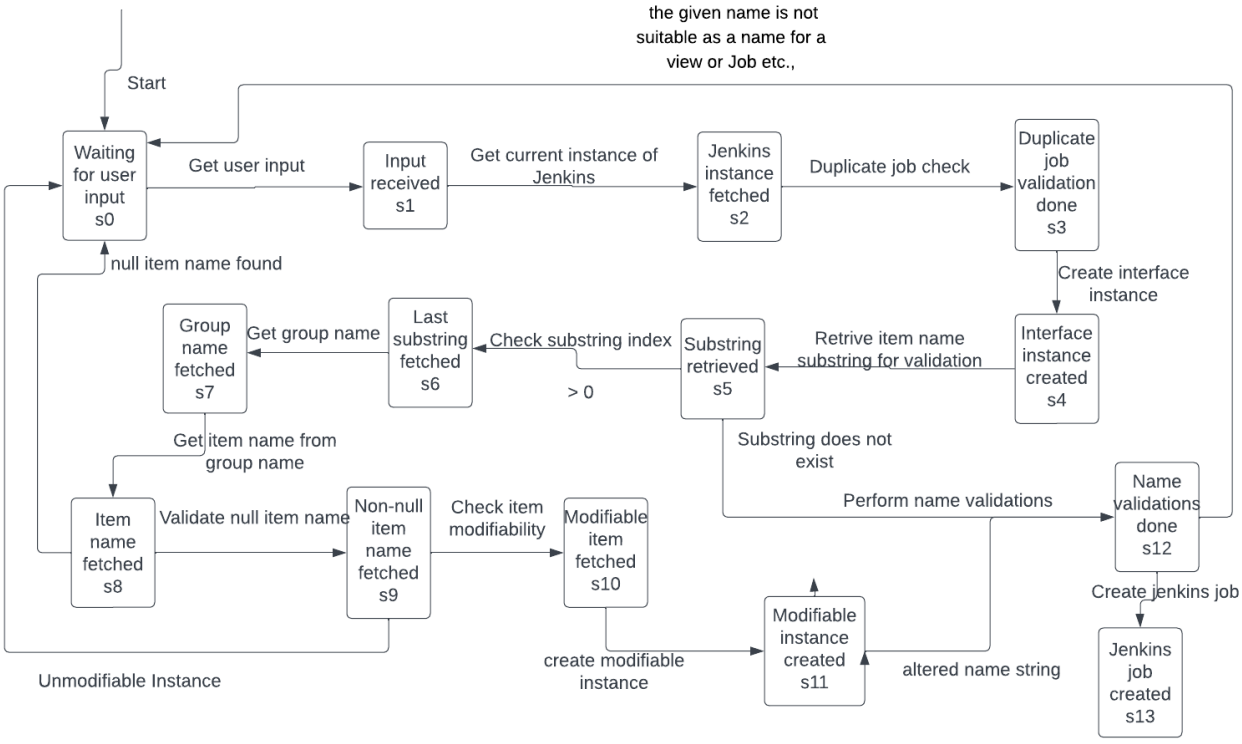


Figure : FSM diagram for the create new job functionality in Jenkins

8. Part 7. JUnit Test Coverage

Following are the 4 test cases that have been added to test the Functional State Model:

The link for the github commit is:
https://github.com/DheerajKumar19/jenkins-261P/commit/591f5227428618a9c9fe0a5fdcc4a61b6ba92c10

The test cases are added in */test/src/test/java/hudson/cli/CreateJobCommandTest.java*

i) **cannotCreateDuplicateJob** - This test case tests the transition between S2 - S3. If a job exists with the name given by the user, then throw an error saying that the job already exists.

```
patil-neha-3
@Test public void cannotCreateDuplicateJob() {
    CLICommand cmd = new CreateJobCommand();
    CLICommandInvoker invoker = new CLICommandInvoker(r, cmd);
    assertThat(r.jenkins.getItems(), Matchers.hasSize(0));

    assertThat(invoker.withStdin(new ByteArrayInputStream("<project/>".getBytes(StandardCharsets.UTF_8))).invokeWithArgs("job1"), succeededSilently());
    assertThat(r.jenkins.getItems(), Matchers.hasSize(1));

    CLICommandInvoker.Result result = invoker.withStdin(new ByteArrayInputStream("<project/>".getBytes(StandardCharsets.UTF_8))).invokeWithArgs("job1");
    assertThat(result.stderr(), containsString( substring: "already exists"));
    assertThat(result, failedWith( expectedCode: 4));
    assertThat(r.jenkins.getItems(), Matchers.hasSize(1));
}
```

ii) **cannotCreateJobInUnknownItemGroup** - This test case tests the transition between S8 - S9. If the item name before the trailing / is empty or invalid, the system throws an error.

```
patil-neha-3
@Test public void cannotCreateJobInUnknownItemGroup() {
    CLICommand cmd = new CreateJobCommand();
    CLICommandInvoker invoker = new CLICommandInvoker(r, cmd);
    assertThat(r.jenkins.getItems(), Matchers.hasSize(0));

    CLICommandInvoker.Result result = invoker.withStdin(new ByteArrayInputStream("<project/>".getBytes(StandardCharsets.UTF_8))).invokeWithArgs("job1/");
    assertThat(result.stderr(), containsString( substring: "Unknown ItemGroup"));
    assertThat(result, failedWith( expectedCode: 3));
    assertThat(r.jenkins.getItems(), Matchers.hasSize(0));
}
```

iii) **cannotCreateJobWithEmptyName** - This test case tests the transition between S11 - S12. The user cannot create a job with an empty name. The system will throw an error saying no name found.

iv) **cannotCreateJobWithUnsafeCharInName** - This test case tests the transition between S11 - S12. The user cannot create a job with a special character in it. The system will throw an error saying unsafe characters found.

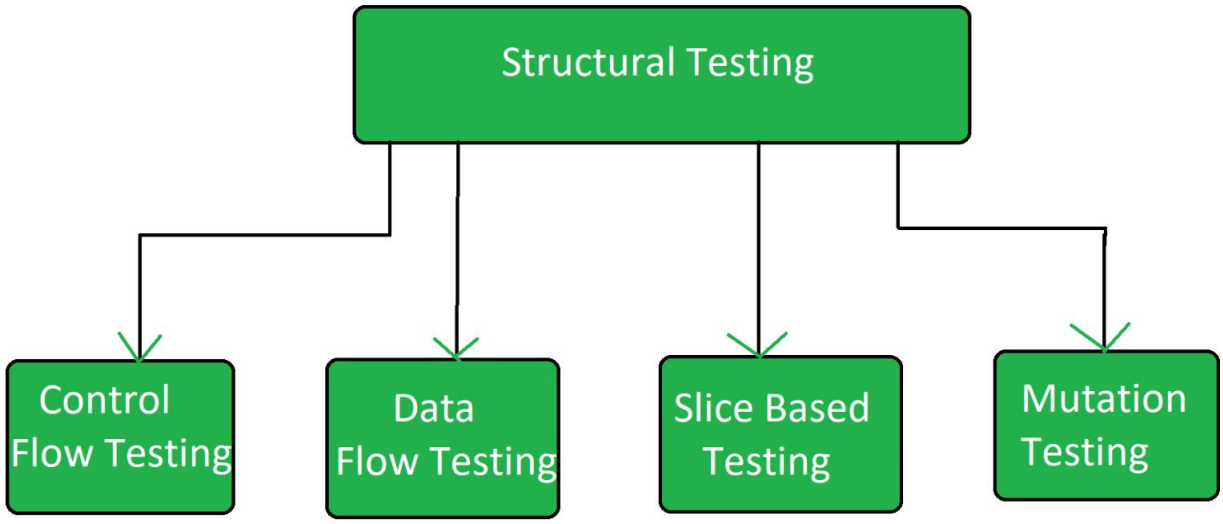
9. Part 8 : Structural Testing

Code structure is tested through a form of testing called structural testing. It is also referred to as the Glass Box or White Box examination. Since this kind of checking necessitates understanding of the code, developers handle it most frequently. Instead of focusing on the system's utility, it is more interested in how it accomplishes its tasks. It gives the testing more scope. For instance, we may need to verify the trigger condition for a particular error message in an application, but there may be several triggers for that error message. While testing the SRS requirements, it is possible to miss one. However, since structural testing seeks to cover all the nodes and paths in the structure of code, it is most likely that the trigger will be covered by this testing.[9]

In addition to functional testing, it is useful. This method enables the analysis of test cases created in accordance with system specifications, followed by the addition of additional test cases to broaden the coverage. It can be applied at various stages, including functional testing, unit testing, component testing, integration testing, and so forth. It aids in conducting in-depth software testing. The majority of the structural testing is automatic.

8.1 Types of Structural Testing:

There are 4 types of Structural Testing:



Control Flow Testing:

A form of structural testing called control flow testing makes use of the control flow of the program as a model. For this kind of testing, the complete software's code, design, and structure must be known. Developers frequently use this kind of testing to check their own code and

execution. This technique is used to evaluate the code's reasoning in order to produce the desired outcome.

Data Flow Testing:

It explores the irrational things that can happen to data using the control flow graph. The associations between values and variables are the foundation for the identification of data flow anomalies. using values without initialization Variables that are initialized are not used once.

Slice Based Testing:

Weiser and Gallagher first suggested it for software maintenance. It is helpful for software upkeep, functional cohesion measurement, and program comprehension. It separates the program into various slices and tests each slice, which has a significant impact on the complete software.

Mutation Testing:

Software testing, or mutation testing, is a technique used to create new software tests and assess the effectiveness of those that already exist. Small-scale program modification is linked to mutation testing. It concentrates on assisting the tester in creating efficient tests or identifying holes in the test information used to create the program.

8.2 Important terms to understand in structural testing

Control Flow Graph: It is a graphical representation of the program depicting all the paths that may be transverse during the execution. It contains

- **Basic Block or Node:** It is a sequence of statements such that control can only have one entry point and can exit the block only when all the statements have been executed.
- **Edge**:It shows the control flow.
- **Adequacy Criterion:** This term refers to the coverage provided by a test suite.

8.3 Structural Testing Techniques:

- **Statement Coverage**:

Its goal is to test every assertion in the program. To guarantee complete covering, the Adequacy Criteria value ought to be 1. Although it is an effective way to evaluate each component in terms of statements, it is a poor method for testing the control flow. For instance, numerous nodes can connect to one node, but since the goal is to only cover the statements, those conditions are not covered.

- **Branch Coverage:**

Testing for decision coverage is another name for it. It attempts to test each branch from a decision point at least once, as well as all the branches or edges in the test suite. It offers a fix for the issue with Statement coverage. Although Branch Testing offers more scope than Statement Testing, it is not without flaws. As it only encompasses that branch once, it does not offer adequate protection against the various circumstances that can lead from node 1 to node 2.

- **Condition Coverage:**

It aims to test individual conditions with possible different combinations of Boolean input for the expression.It is modification of Decision coverage but it provides better coverage and the problem discussed under Branch coverage can be resolved here.

- **Path Coverage:**

It aims to test the different path from entry to the exit of the program, which is a combination of different decisions taken in the sequence.The paths can be too many to be considered for testing, for example, a loop can go on and on. It can be avoided using cyclomatic complexity, which helps in finding out the redundant test cases.

8.4 Advantages of Structural Testing:

- Provides a more thorough testing of the software.
- Helps find defects at an early stage.
- Helpsin eliminating dead code.
- Not time consuming as it is mostly automated.

8.5 Disadvantages of Structural Testing:

- Requires knowledge of the code.
- Requires training in the tool used for testing
- It is expensive.

8.6 Tools Required:

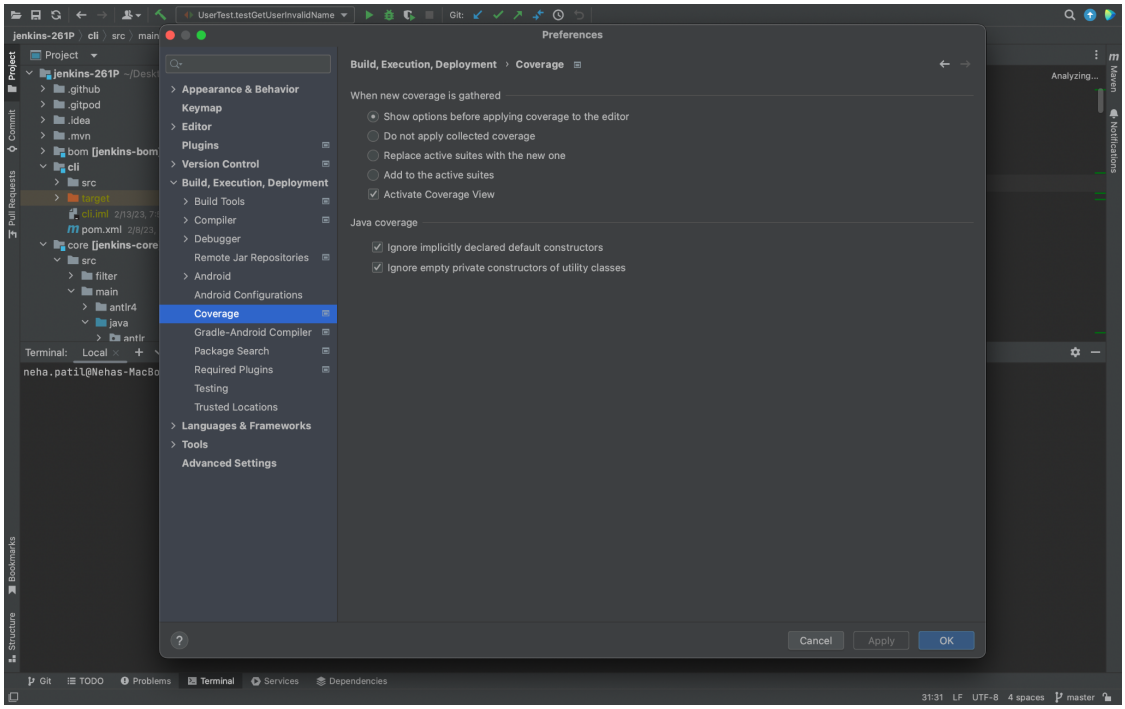
The available test instruments that are utilized for Structural Testing are:

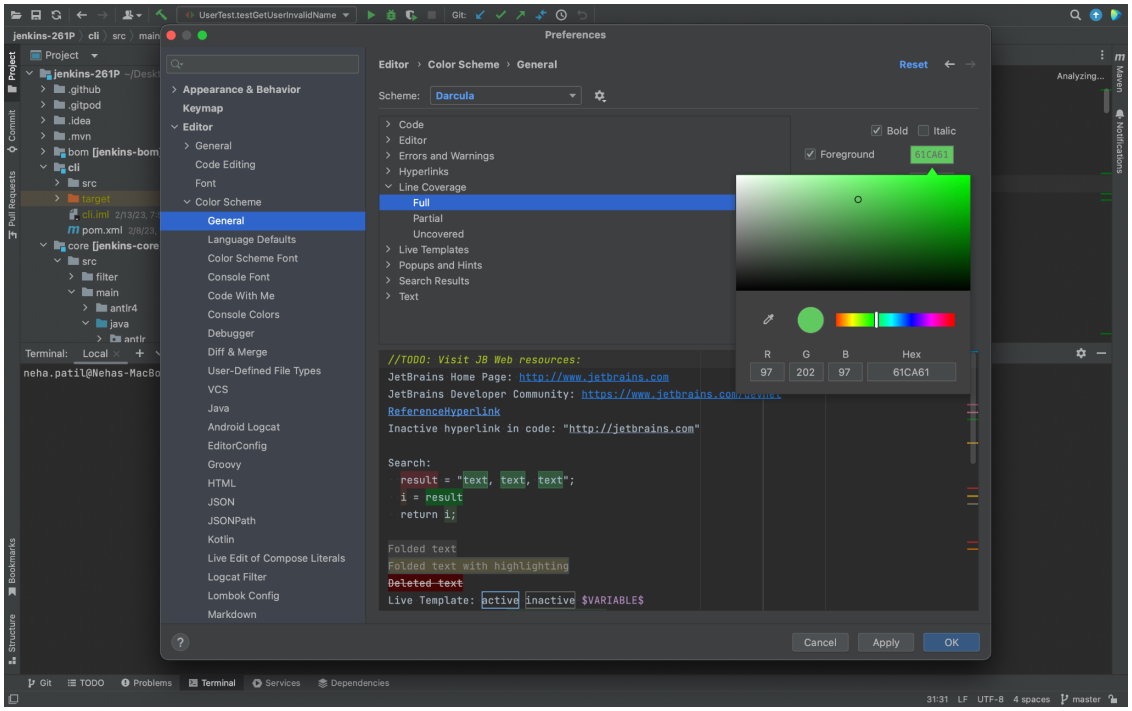
1. JBehave
2. Cucumber
3. Junit
4. Cfix

10. Part 9 : Running a Test coverage tool

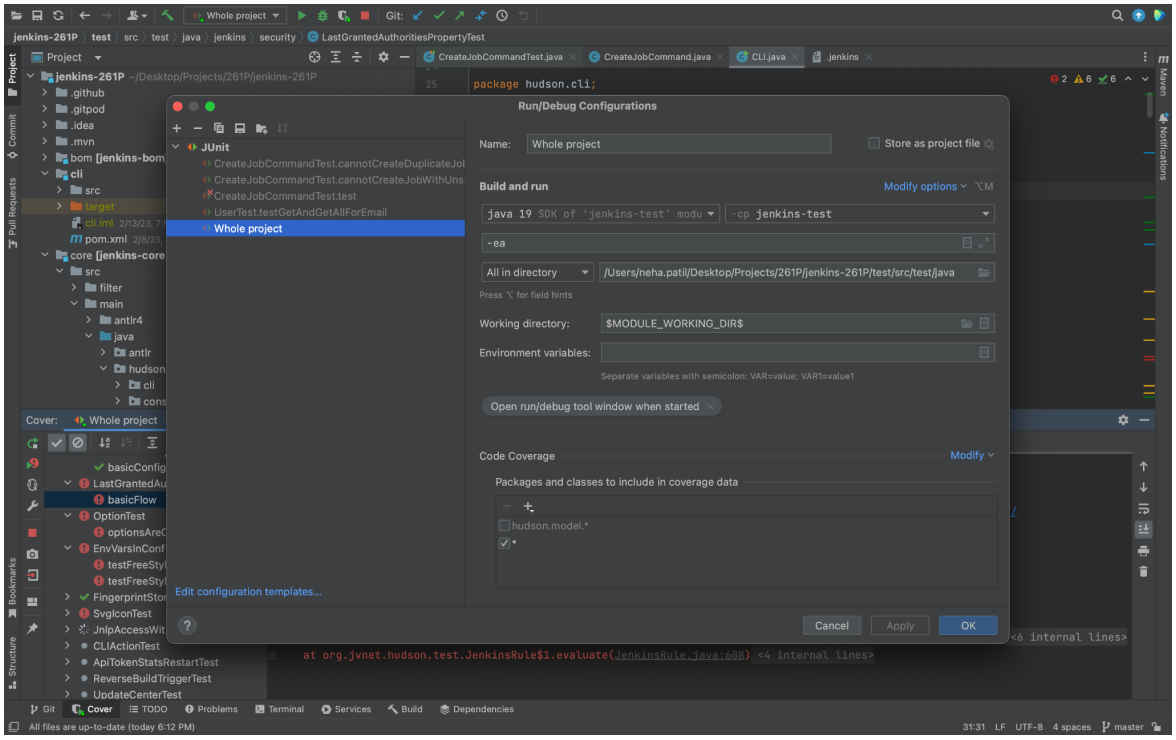
We are using IntelliJ Idea's coverage plugin - IntelliJ IDEA code coverage runner. Setting up this is simple, follow the below screenshots.

9.1 Configuring coverage tool



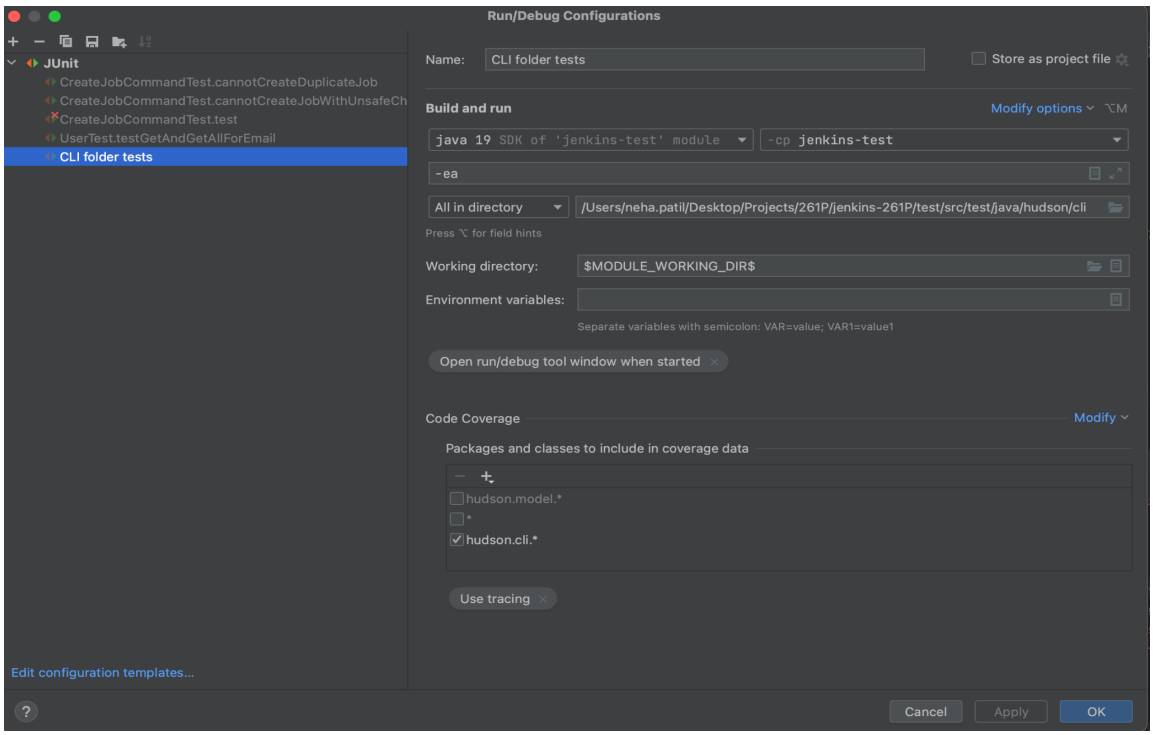


Configuration to run tests on the whole project:

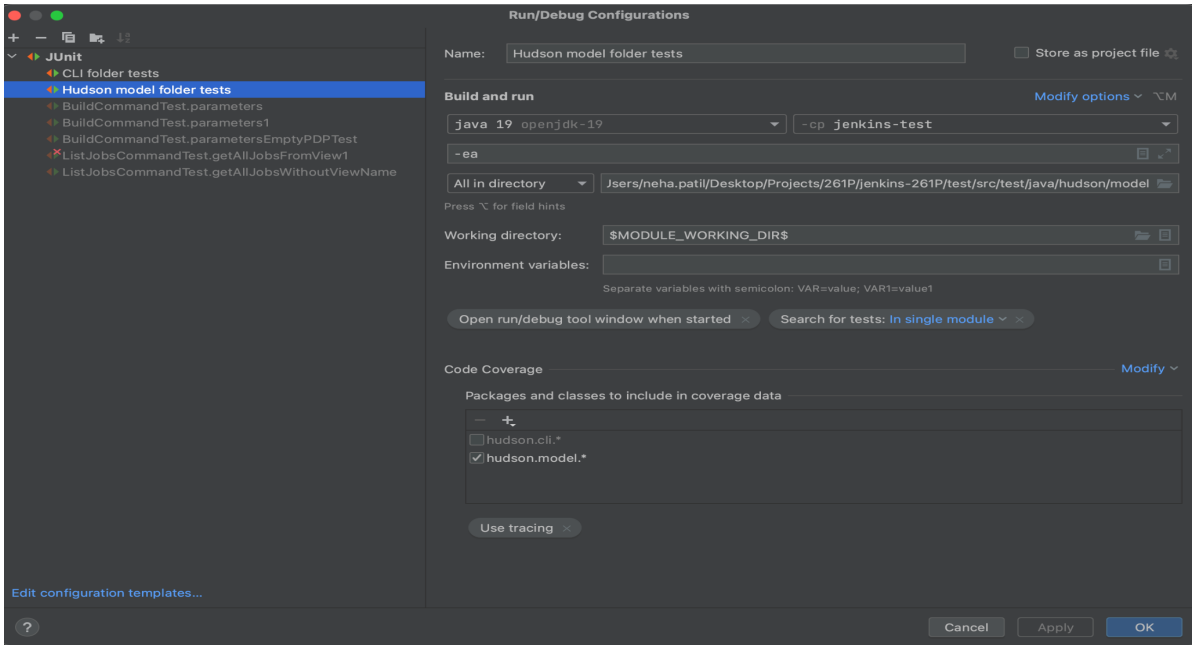


There are a total of approximately 9000 tests in the Jenkins project. To narrow down the scope for this project, we would be focusing on the tests in the CLI folder and the Model folder.

Configuration to run tests on the CLI folder:



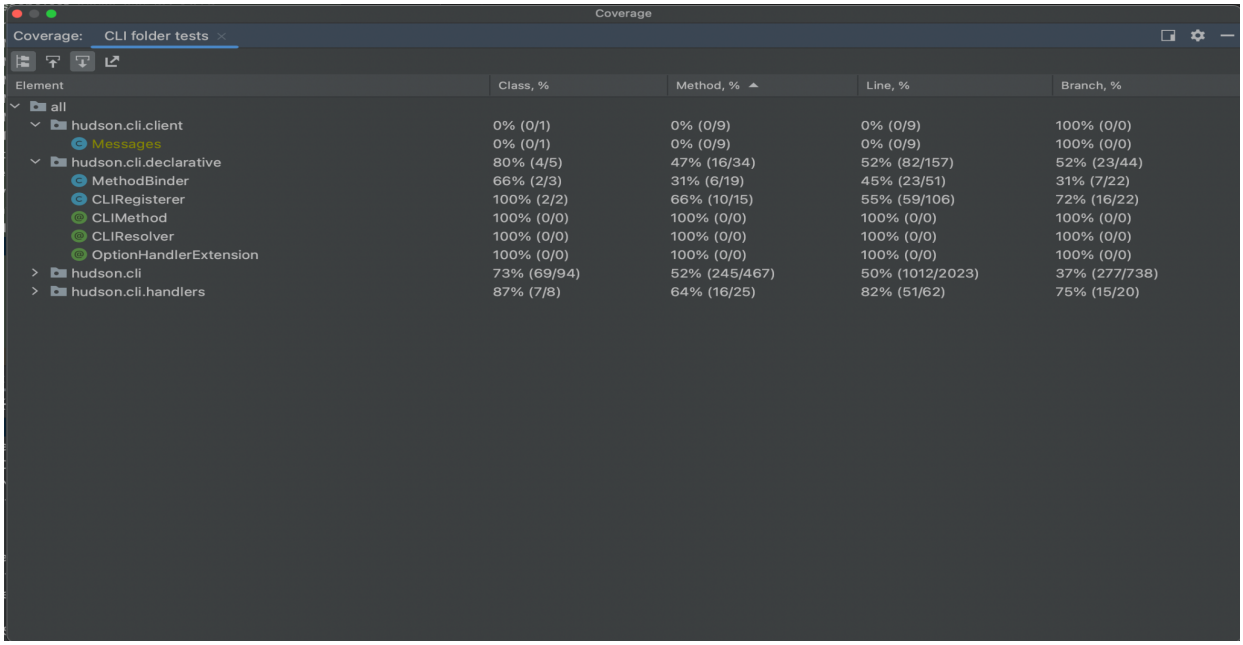
Configuration to run tests on the Model folder:

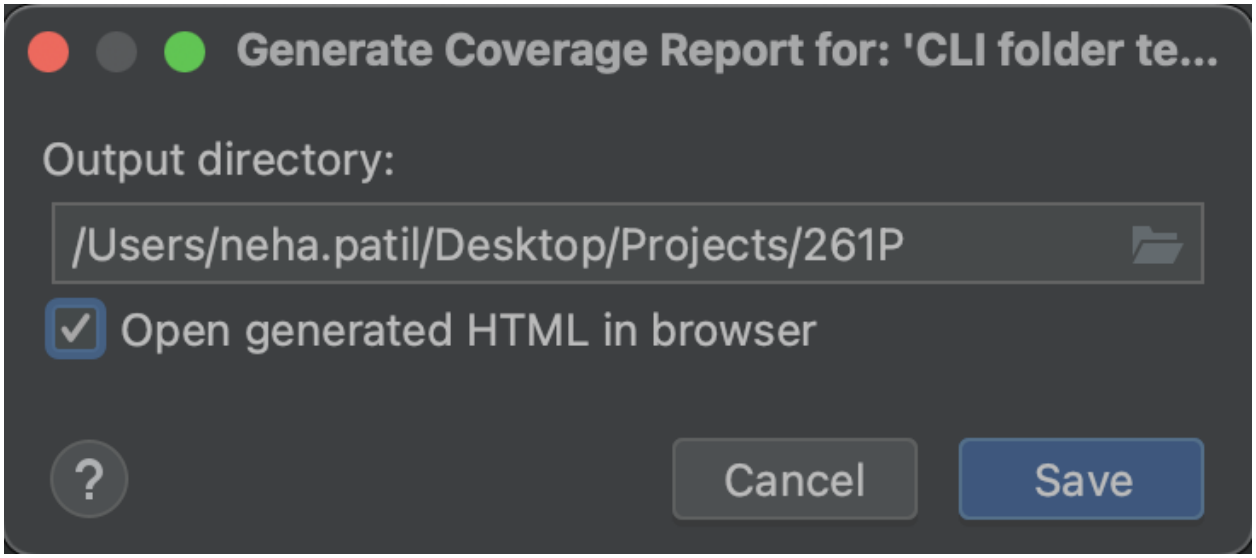


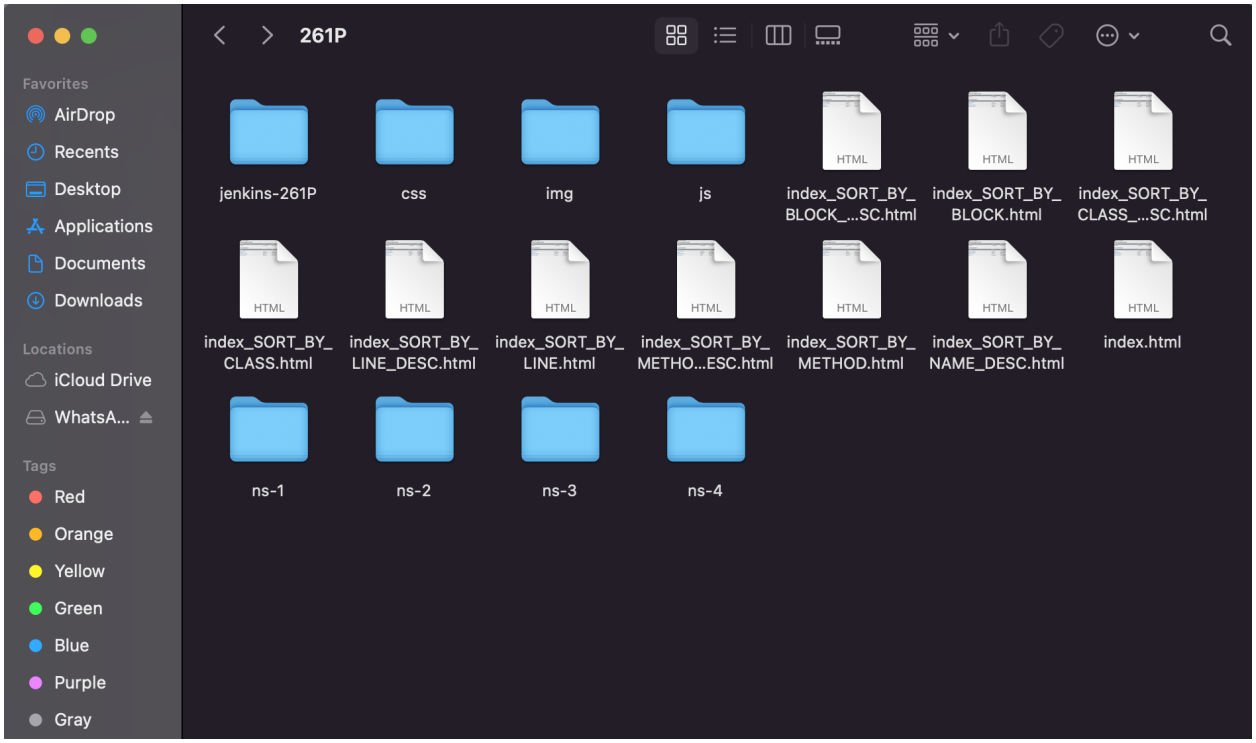
IntelliJ by default gives the line, method and class coverage in the coverage report. In order to get the details on branch coverage, we need to enable the **Use tracing** option in our Run Configuration.

9.2 Generating coverage report:

IntelliJ lets us generate the coverage report and view it in the IDE. There is also an option to download the to local storage and view it in the browser.

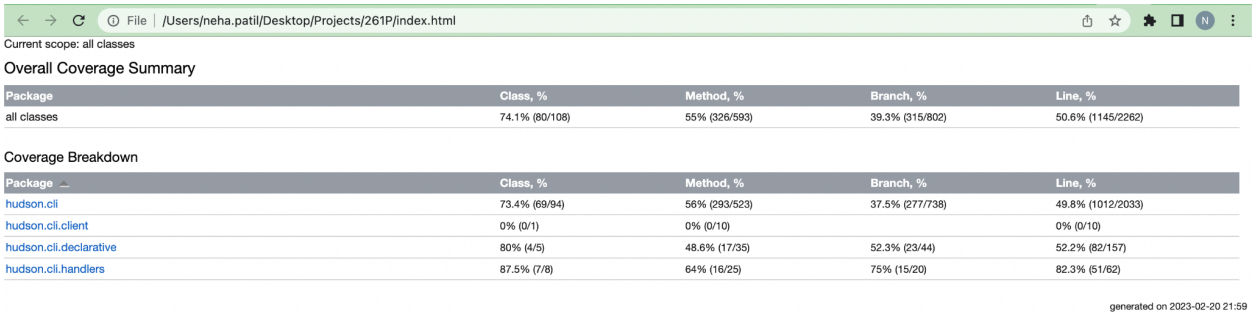



Generate Coverage Report for: 'CLI folder te...
Output directory:
/Users/neha.patil/Desktop/Projects/261P
Open generated HTML in browser
Cancel
Save


261P
Favorites
AirDrop
Recents
Desktop
Applications
Documents
Downloads
Locations
iCloud Drive
WhatsA...
Tags
Red
Orange
Yellow
Green
Blue
Purple
Gray
jenkins-261P
css
img
js
index_SORT_BY_BLOCK_...SC.html
index_SORT_BY_BLOCK.html
index_SORT_BY_CLASS_...SC.html
index_SORT_BY_CLASS.html
index_SORT_BY_LINE_DESC.html
index_SORT_BY_LINE.html
index_SORT_BY_METHO...ESC.html
index_SORT_BY_METHOD.html
index_SORT_BY_NAME_DESC.html
index.html
ns-1
ns-2
ns-3
ns-4

9.3 Current Test Coverage of CLI and Model folders

Current coverage for the CLI folder:



File | /Users/neha.patil/Desktop/Projects/261P/index.html

Current scope: all classes

Overall Coverage Summary

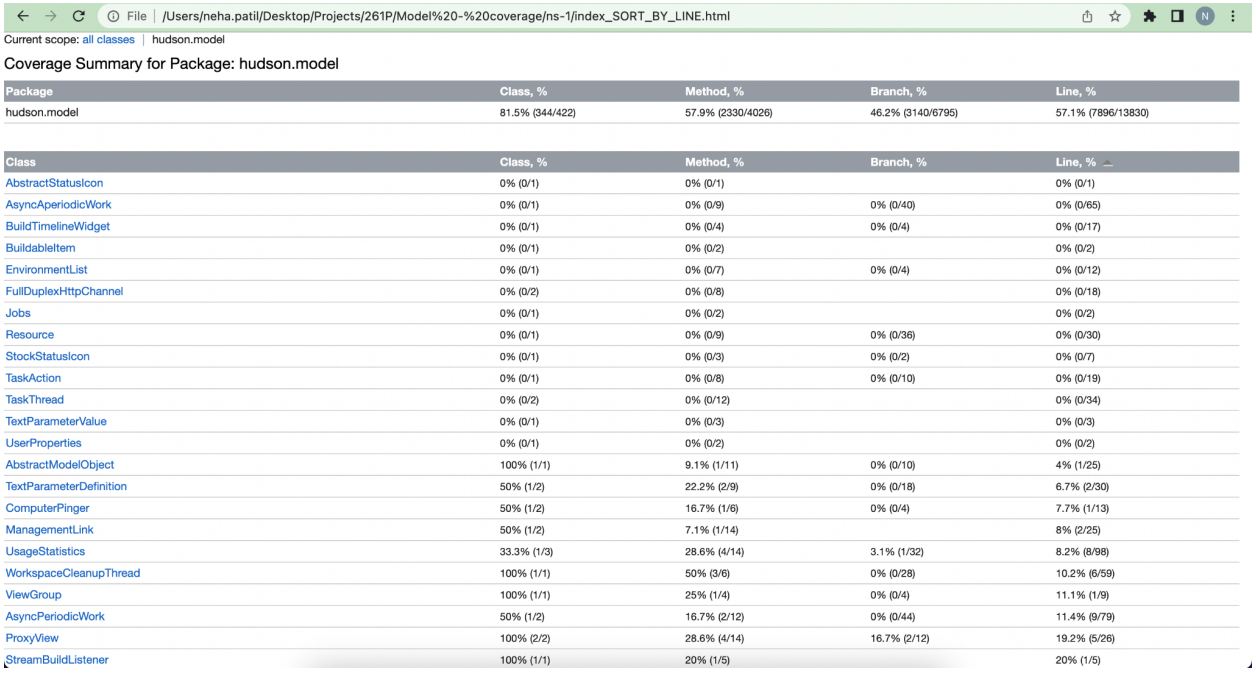
Package	Class, %	Method, %	Branch, %	Line, %
all classes	74.1% (80/108)	55% (326/593)	39.3% (315/802)	50.6% (1145/2262)

Coverage Breakdown

Package	Class, %	Method, %	Branch, %	Line, %
hudson.cli	73.4% (69/94)	56% (293/523)	37.5% (277/738)	49.8% (1012/2033)
hudson.cli.client	0% (0/1)	0% (0/10)		0% (0/10)
hudson.cli.declarative	80% (4/5)	48.6% (17/35)	52.3% (23/44)	52.2% (82/157)
hudson.cli.handlers	87.5% (7/8)	64% (16/25)	75% (15/20)	82.3% (51/62)

generated on 2023-02-20 21:59

Current coverage for the model folder:



File | /Users/neha.patil/Desktop/Projects/261P/Model%20-%20coverage/ns-1/index_SORT_BY_LINE.html

Current scope: all classes | hudson.model

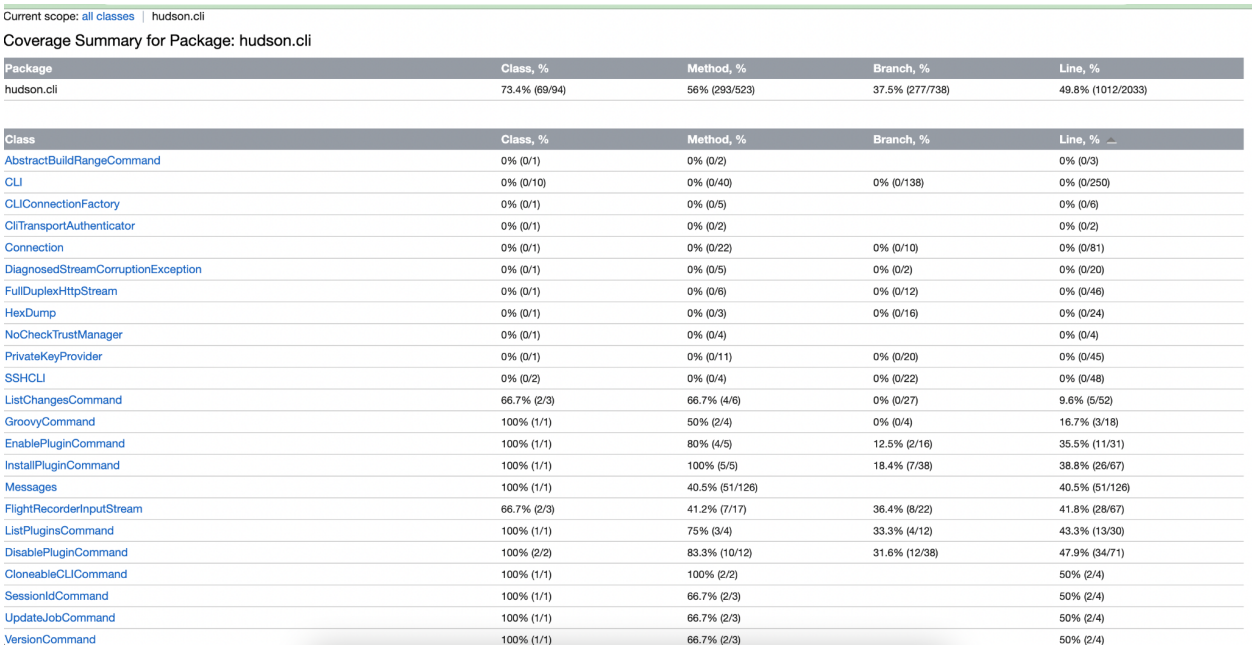
Coverage Summary for Package: hudson.model

Package	Class, %	Method, %	Branch, %	Line, %
hudson.model	81.5% (344/422)	57.9% (2330/4026)	46.2% (3140/6795)	57.1% (7896/13830)

Class	Class, %	Method, %	Branch, %	Line, %
AbstractStatusIcon	0% (0/1)	0% (0/1)		0% (0/1)
AsyncAperiodicWork	0% (0/1)	0% (0/9)	0% (0/40)	0% (0/65)
BuildTimelineWidget	0% (0/1)	0% (0/4)	0% (0/4)	0% (0/17)
BuildableItem	0% (0/1)	0% (0/2)		0% (0/2)
EnvironmentList	0% (0/1)	0% (0/7)	0% (0/4)	0% (0/12)
FullDuplexHttpChannel	0% (0/2)	0% (0/8)		0% (0/18)
Jobs	0% (0/1)	0% (0/2)		0% (0/2)
Resource	0% (0/1)	0% (0/9)	0% (0/36)	0% (0/30)
StockStatusIcon	0% (0/1)	0% (0/3)	0% (0/2)	0% (0/7)
TaskAction	0% (0/1)	0% (0/8)	0% (0/10)	0% (0/19)
TaskThread	0% (0/2)	0% (0/12)		0% (0/34)
TextParameterValue	0% (0/1)	0% (0/3)		0% (0/3)
UserProperties	0% (0/1)	0% (0/2)		0% (0/2)
AbstractModelObject	100% (1/1)	9.1% (1/11)	0% (0/10)	4% (1/25)
TextParameterDefinition	50% (1/2)	22.2% (2/9)	0% (0/18)	6.7% (2/30)
ComputerPinger	50% (1/2)	16.7% (1/6)	0% (0/4)	7.7% (1/13)
ManagementLink	50% (1/2)	7.1% (1/14)		8% (2/25)
UsageStatistics	33.3% (1/3)	28.6% (4/14)	3.1% (1/32)	8.2% (8/98)
WorkspaceCleanupThread	100% (1/1)	50% (3/6)	0% (0/28)	10.2% (6/59)
ViewGroup	100% (1/1)	25% (1/4)	0% (0/4)	11.1% (1/9)
AsyncPeriodicWork	50% (1/2)	16.7% (2/12)	0% (0/44)	11.4% (9/79)
ProxyView	100% (2/2)	28.6% (4/14)	16.7% (2/12)	19.2% (5/26)
StreamBuildListener	100% (1/1)	20% (1/5)		20% (1/5)

9.4 Highlighting some parts uncovered code by the existing test suite:

These are some of the classes with the lowest line coverage in the hudson.cli folder:



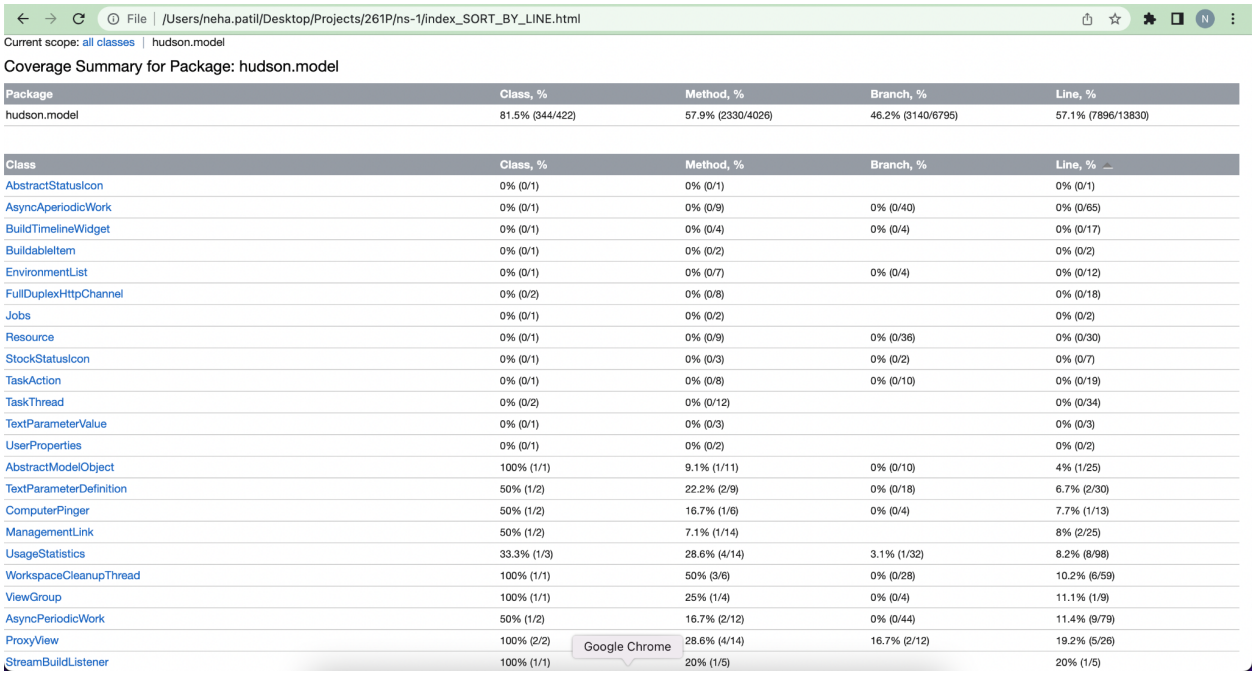
Current scope: all classes | hudson.cli

Coverage Summary for Package: hudson.cli

Package	Class, %	Method, %	Branch, %	Line, %
hudson.cli	73.4% (69/94)	56% (293/523)	37.5% (277/738)	49.8% (1012/2033)

Class	Class, %	Method, %	Branch, %	Line, %
AbstractBuildRangeCommand	0% (0/1)	0% (0/2)		0% (0/3)
CLI	0% (0/10)	0% (0/40)	0% (0/138)	0% (0/250)
CLIConnectionFactory	0% (0/1)	0% (0/5)		0% (0/6)
CliTransportAuthenticator	0% (0/1)	0% (0/2)		0% (0/2)
Connection	0% (0/1)	0% (0/22)	0% (0/10)	0% (0/81)
DiagnosedStreamCorruptionException	0% (0/1)	0% (0/5)	0% (0/2)	0% (0/20)
FullDuplexHttpStream	0% (0/1)	0% (0/6)	0% (0/12)	0% (0/46)
HexDump	0% (0/1)	0% (0/3)	0% (0/16)	0% (0/24)
NoCheckTrustManager	0% (0/1)	0% (0/4)		0% (0/4)
PrivateKeyProvider	0% (0/1)	0% (0/11)	0% (0/20)	0% (0/45)
SSHCLI	0% (0/2)	0% (0/4)	0% (0/22)	0% (0/48)
ListChangesCommand	66.7% (2/3)	66.7% (4/6)	0% (0/27)	9.6% (5/52)
GroovyCommand	100% (1/1)	50% (2/4)	0% (0/4)	16.7% (3/18)
EnablePluginCommand	100% (1/1)	80% (4/5)	12.5% (2/16)	35.5% (11/31)
InstallPluginCommand	100% (1/1)	100% (5/5)	18.4% (7/38)	38.8% (26/67)
Messages	100% (1/1)	40.5% (51/126)		40.5% (51/126)
FlightRecorderInputStream	66.7% (2/3)	41.2% (7/17)	36.4% (8/22)	41.8% (28/67)
ListPluginsCommand	100% (1/1)	75% (3/4)	33.3% (4/12)	43.3% (13/30)
DisablePluginCommand	100% (2/2)	83.3% (10/12)	31.6% (12/38)	47.9% (34/71)
CloneableCLICommand	100% (1/1)	100% (2/2)		50% (2/4)
SessionIdCommand	100% (1/1)	66.7% (2/3)		50% (2/4)
UpdateJobCommand	100% (1/1)	66.7% (2/3)		50% (2/4)
VersionCommand	100% (1/1)	66.7% (2/3)		50% (2/4)

These are some of the classes with the lowest line coverage in the hudson.model folder:



File | /Users/neha.patil/Desktop/Projects/261P/ns-1/index_SORT_BY_LINE.html

Current scope: all classes | hudson.model

Coverage Summary for Package: hudson.model

Package	Class, %	Method, %	Branch, %	Line, %
hudson.model	81.5% (344/422)	57.9% (2330/4026)	46.2% (3140/6795)	57.1% (7896/13830)

Class	Class, %	Method, %	Branch, %	Line, %
AbstractStatusIcon	0% (0/1)	0% (0/1)		0% (0/1)
AsyncAperiodicWork	0% (0/1)	0% (0/9)	0% (0/40)	0% (0/65)
BuildTimelineWidget	0% (0/1)	0% (0/4)	0% (0/4)	0% (0/17)
BuildableItem	0% (0/1)	0% (0/2)		0% (0/2)
EnvironmentList	0% (0/1)	0% (0/7)	0% (0/4)	0% (0/12)
FullDuplexHttpChannel	0% (0/2)	0% (0/8)		0% (0/18)
Jobs	0% (0/1)	0% (0/2)		0% (0/2)
Resource	0% (0/1)	0% (0/9)	0% (0/36)	0% (0/30)
StockStatusIcon	0% (0/1)	0% (0/3)	0% (0/2)	0% (0/7)
TaskAction	0% (0/1)	0% (0/8)	0% (0/10)	0% (0/19)
TaskThread	0% (0/2)	0% (0/12)		0% (0/34)
TextParameterValue	0% (0/1)	0% (0/3)		0% (0/3)
UserProperties	0% (0/1)	0% (0/2)		0% (0/2)
AbstractModelObject	100% (1/1)	9.1% (1/11)	0% (0/10)	4% (1/25)
TextParameterDefinition	50% (1/2)	22.2% (2/9)	0% (0/18)	6.7% (2/30)
ComputerPinger	50% (1/2)	16.7% (1/6)	0% (0/4)	7.7% (1/13)
ManagementLink	50% (1/2)	7.1% (1/14)		8% (2/25)
UsageStatistics	33.3% (1/3)	28.6% (4/14)	3.1% (1/32)	8.2% (8/98)
WorkspaceCleanupThread	100% (1/1)	50% (3/6)	0% (0/28)	10.2% (6/59)
ViewGroup	100% (1/1)	25% (1/4)	0% (0/4)	11.1% (1/9)
AsyncPeriodicWork	50% (1/2)	16.7% (2/12)	0% (0/44)	11.4% (9/79)
ProxyView	100% (2/2)	28.6% (4/14)	16.7% (2/12)	19.2% (5/26)
StreamBuildListener	100% (1/1)	20% (1/5)		20% (1/5)

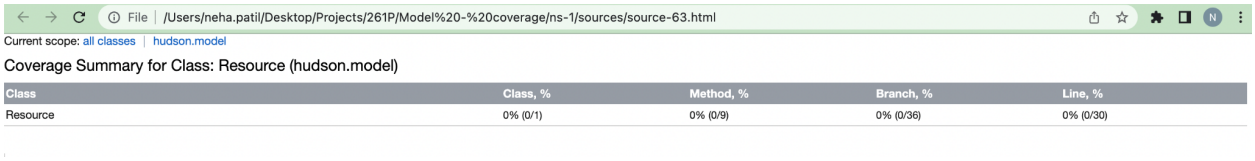
Google Chrome

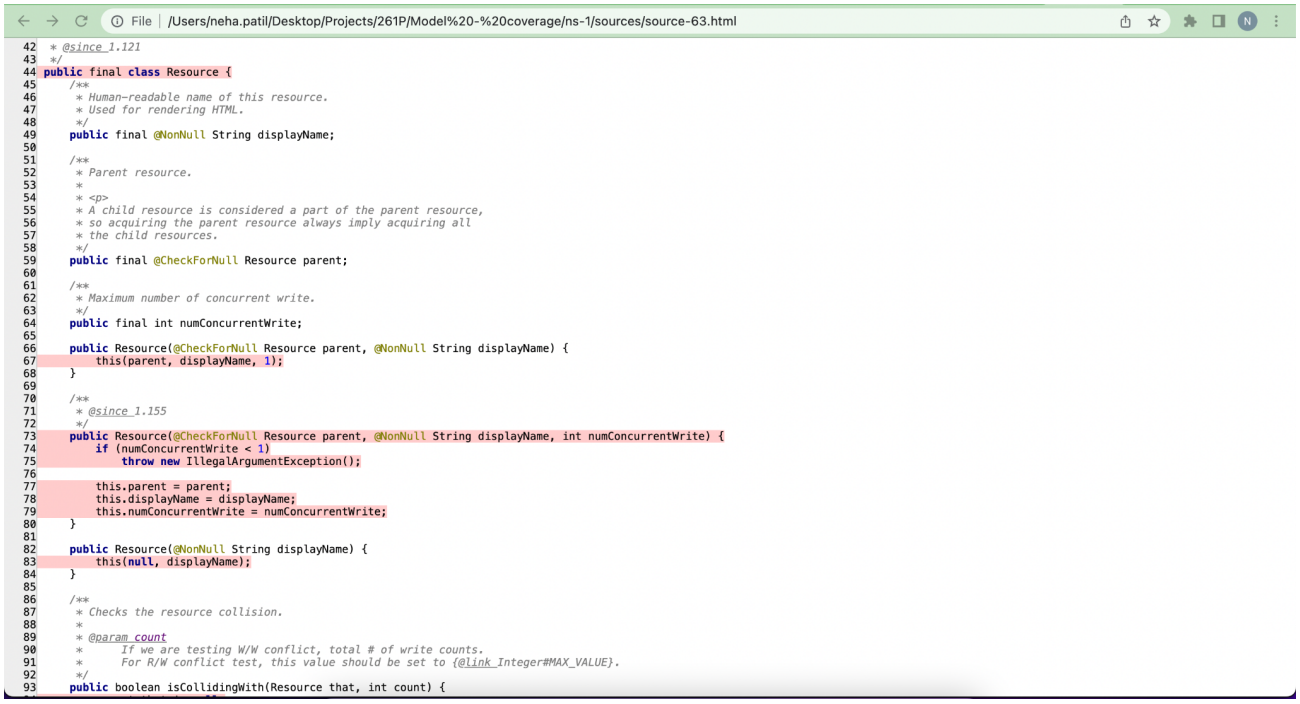
Part 10 :Test coverage Improvement

The newly added Junits can be found in the commit_link.
The test coverage report can be found here: CLI_Folder , Model_folder.

10.1 Resource.java File

Before: The Resource.java file does not have any test cases covering it i.e 0% coverage.
The ResourceList.java file has 22/45 lines covered.





File | /Users/neha.patil/Desktop/Projects/261P/Model%20-%20coverage/ns-1/sources/source-63.html

```
84      }
85
86      /**
87       * Checks the resource collision.
88       *
89       * @param count
90       *      If we are testing W/W conflict, total # of write counts.
91       *      For R/W conflict test, this value should be set to {@link Integer#MAX_VALUE}.
92       */
93      public boolean isCollidingWith(Resource that, int count) {
94          assert that != null;
95          for (Resource r = that; r != null; r = r.parent)
96              if (this.equals(r) && r.numConcurrentWrite < count)
97                  return true;
98          for (Resource r = this; r != null; r = r.parent)
99              if (that.equals(r) && r.numConcurrentWrite < count)
100                 return true;
101         return false;
102     }
103
104     @Override
105     public boolean equals(Object o) {
106         if (this == o) return true;
107         if (o == null || getClass() != o.getClass()) return false;
108         Resource that = (Resource) o;
109
110         return displayName.equals(that.displayName) && eq(this.parent, that.parent);
111     }
112
113     private static boolean eq(Object lhs, Object rhs) {
114         if (lhs == rhs)     return true;
115         if (lhs == null || rhs == null)  return false;
116         return lhs.equals(rhs);
117     }
118
119     @Override
120     public int hashCode() {
121         return displayName.hashCode();
122     }
123
124     @Override
125     public String toString() {
126         StringBuilder buf = new StringBuilder();
127         if (parent != null)
128             buf.append(parent).append('/');
129         buf.append(displayName).append('(').append(numConcurrentWrite).append(')');
130         return buf.toString();
131     }
132 }
```

generated on 2023-02-21 02:

File | /Users/neha.patil/Desktop/Projects/261P/Model%20-%20coverage/ns-1/sources/source-66.html

Current scope: all classes | hudson.model

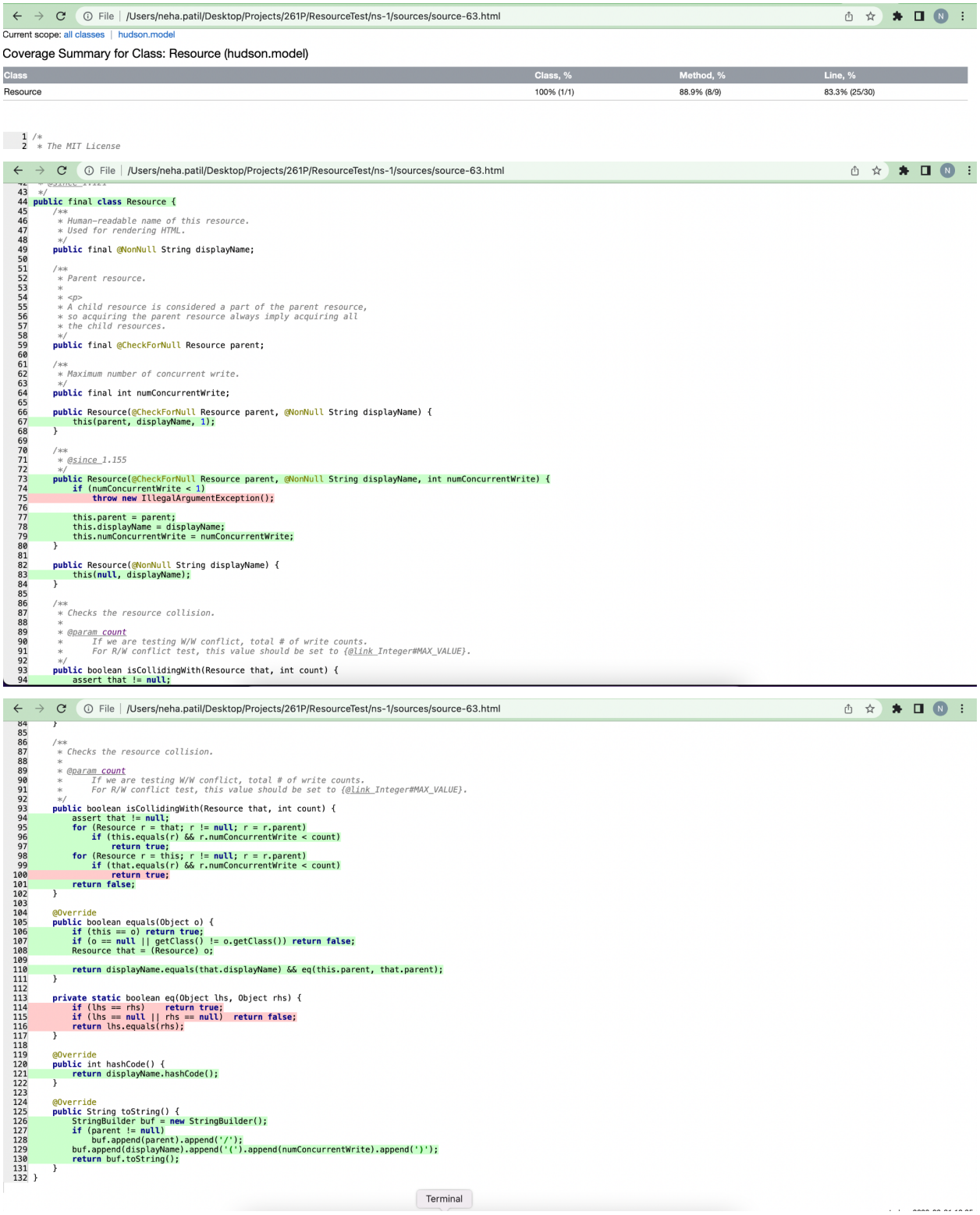
Coverage Summary for Class: ResourceList (hudson.model)

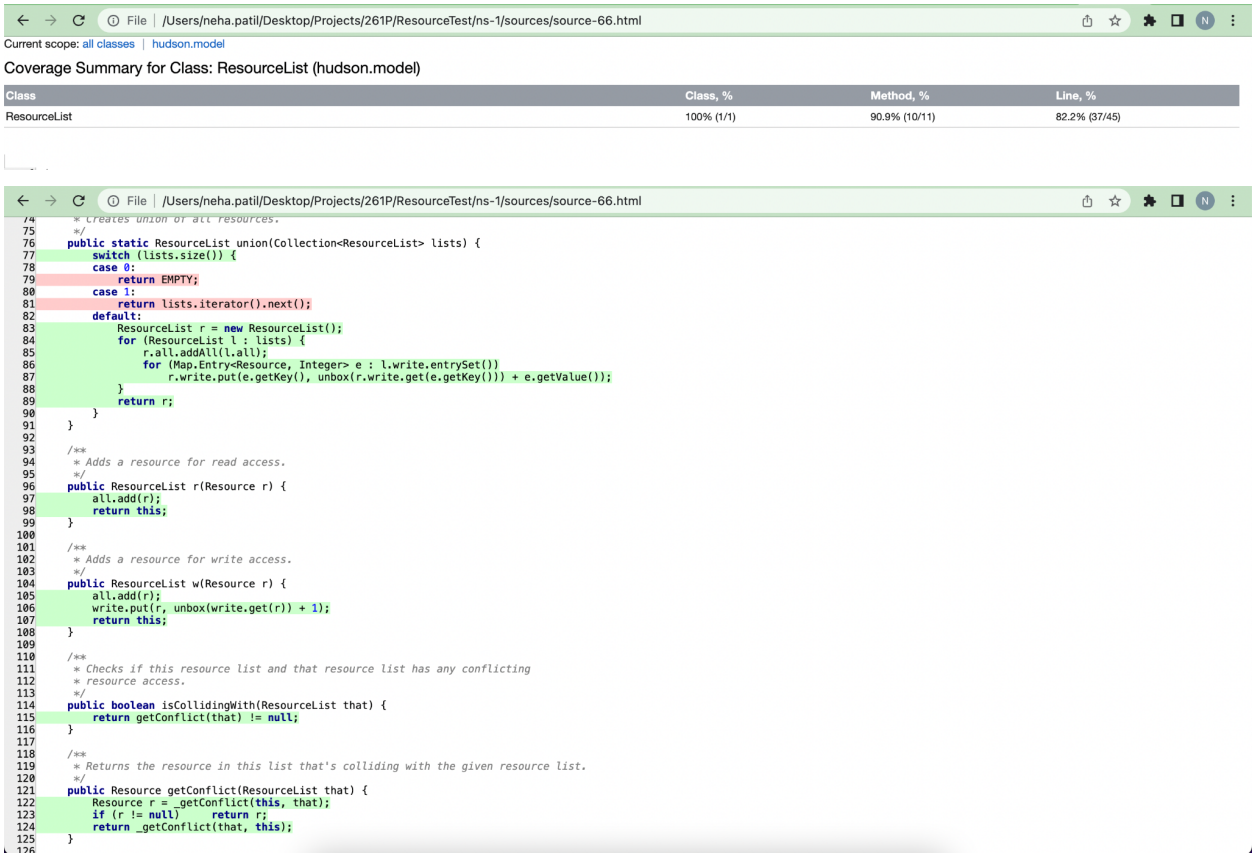
Class	Class, %	Method, %	Branch, %	Line, %
ResourceList	100% (1/1)	63.6% (7/11)	33.3% (8/24)	48.9% (22/45)

File | /Users/neha.patil/Desktop/Projects/261P/Model%20-%20coverage/ns-1/sources/source-66.html

```
86                  for (Map.Entry<Resource, Integer> e : l.write.entrySet())
87                      r.write.put(e.getKey(), unbox(r.write.get(e.getKey())) + e.getValue());
88              }
89              return r;
90          }
91      }
92
93      /**
94       * Adds a resource for read access.
95       */
96      public ResourceList r(Resource r) {
97          all.add(r);
98          return this;
99      }
100
101     /**
102      * Adds a resource for write access.
103      */
104     public ResourceList w(Resource r) {
105         all.add(r);
106         write.put(r, unbox(write.get(r)) + 1);
107         return this;
108     }
109
110     /**
111      * Checks if this resource list and that resource list has any conflicting
112      * resource access.
113      */
114     public boolean isCollidingWith(ResourceList that) {
115         return getConflict(that) != null;
116     }
117
118     /**
119      * Returns the resource in this list that's colliding with the given resource list.
120      */
121     public Resource getConflict(ResourceList that) {
122         Resource r = _getConflict(this, that);
123         if (r != null)      return r;
124         return _getConflict(that, this);
125     }
126
127     private Resource _getConflict(ResourceList lhs, ResourceList rhs) {
128         for (Map.Entry<Resource, Integer> r : lhs.write.entrySet()) {
129             for (Resource l : rhs.all) {
130                 Integer v = rhs.write.get(l);
131                 if (v != null) // this is write/write conflict.
132                     v += r.getValue();
133                 else // Otherwise set it to a very large value, since it's read/write conflict
134                     v = MAX_INT;
135                 if (r.getKey().isCollidingWith(l, unbox(v))) {
136                     LOGGER.info("Collision with " + r + " and " + l);
137                     return r.getKey();
```

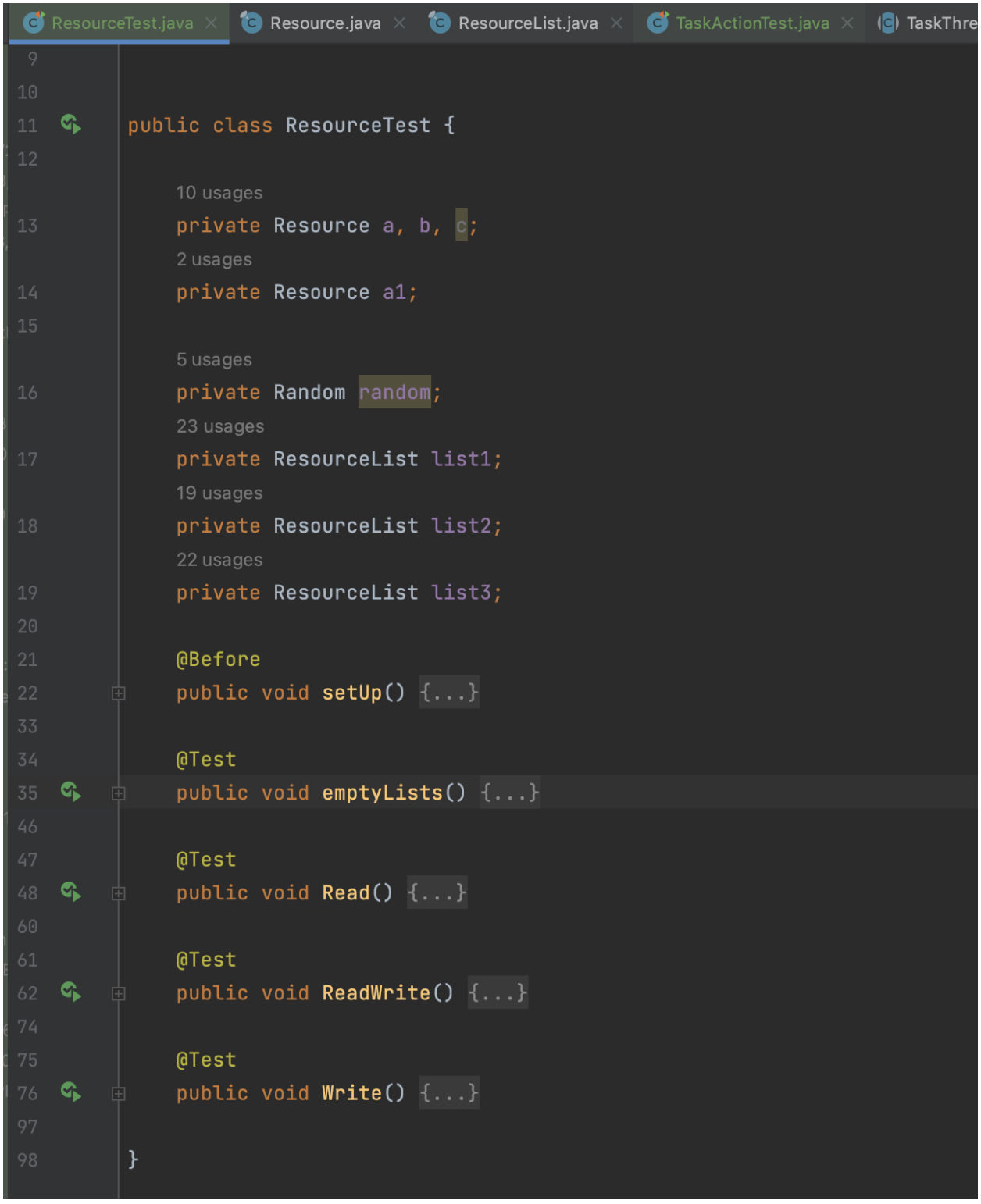
After: Test coverage improved by 25 lines in Resource.java and additional 15 lines in ResourceList.java file.





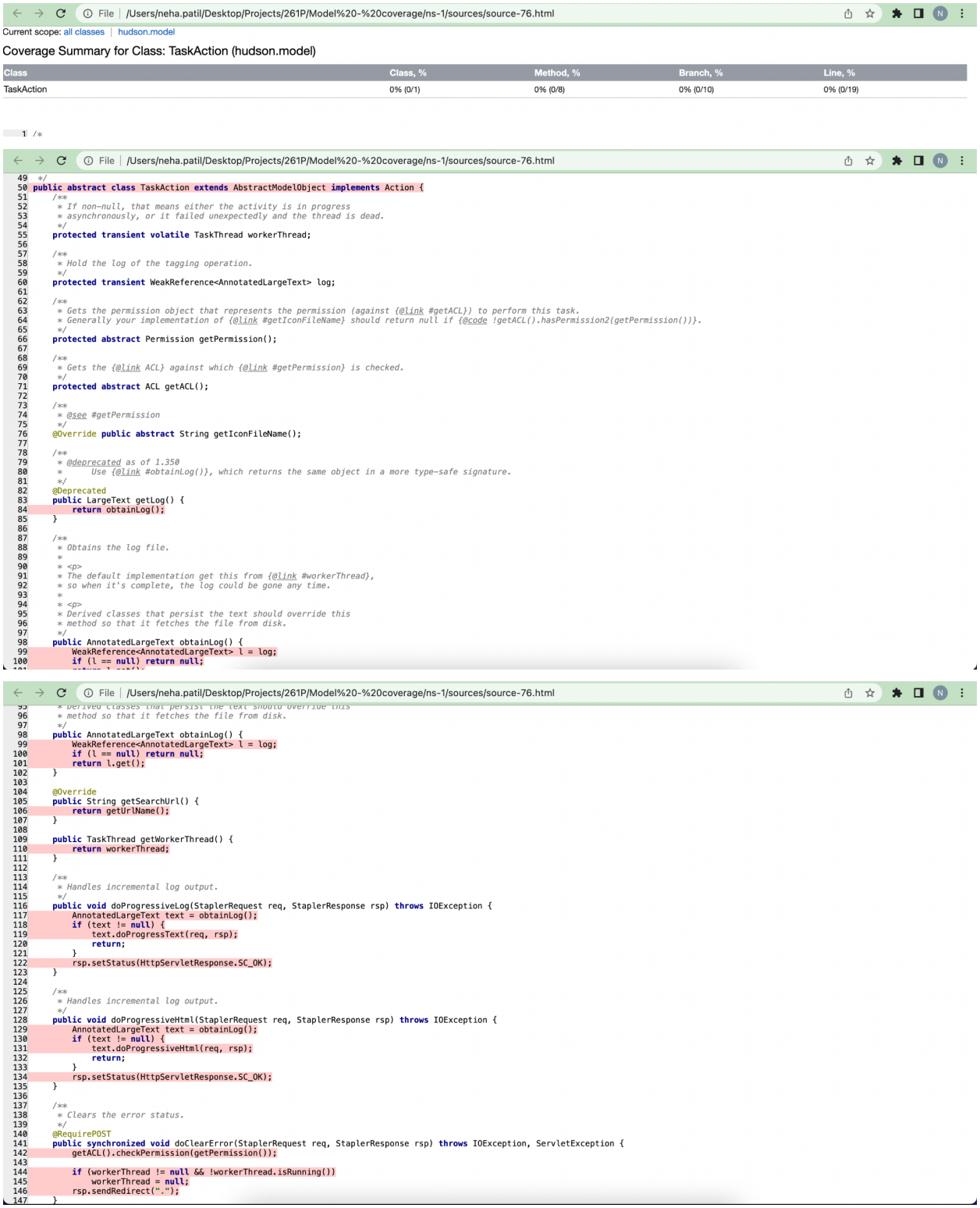
Explanation:
The items that a Queue utilizes while it is active are represented by Resources. To enable basic mutual exclusion/locks, this is used in queues. The same Resource won't be used by two queued tasks at the same moment. The ResourceTest test suite is testing the mutual exclusion principle in case multiple queued tasks attempt to access the same resource at the same time.

JUnit Added:


ResourceTest.java
Resource.java
ResourceList.java
TaskActionTest.java
TaskThre
9
10
11 public class ResourceTest {
12
10 usages
13 private Resource a, b, c;
2 usages
14 private Resource a1;
15
5 usages
16 private Random random;
23 usages
17 private ResourceList list1;
19 usages
18 private ResourceList list2;
22 usages
19 private ResourceList list3;
20
21 @Before
22 public void setUp() {...}
33
34 @Test
35 public void emptyLists() {...}
46
47 @Test
48 public void Read() {...}
60
61 @Test
62 public void ReadWrite() {...}
74
75 @Test
76 public void Write() {...}
97
98 }

10.2 TaskAction.java and TaskThread.java File

Before: TaskAction.java and TaskThread.java files have 0 lines covered. TaskListener has 8/14 lines covered.



File | /Users/neha.patil/Desktop/Projects/261P/Model%20-%20coverage/ns-1/sources/source-76.html

Current scope: all classes | hudson.model

Coverage Summary for Class: TaskAction (hudson.model)

Class	Class, %	Method, %	Branch, %	Line, %
TaskAction	0% (0/1)	0% (0/8)	0% (0/10)	0% (0/19)

```
1  /*
49 */
50 public abstract class TaskAction extends AbstractModelObject implements Action {
51     /**
52      * If non-null, that means either the activity is in progress
53      * asynchronously, or it failed unexpectedly and the thread is dead.
54      */
55     protected transient volatile TaskThread workerThread;
56
57     /**
58      * Hold the log of the tagging operation.
59      */
60     protected transient WeakReference<AnnotatedLargeText> log;
61
62     /**
63      * Gets the permission object that represents the permission (against {@link #getACL}) to perform this task.
64      * Generally your implementation of {@link #getIconFileName} should return null if {@code !getACL().hasPermission2(getPermission())}.
65      */
66     protected abstract Permission getPermission();
67
68     /**
69      * Gets the {@link ACL} against which {@link #getPermission} is checked.
70      */
71     protected abstract ACL getACL();
72
73     /**
74      * @see #getPermission
75      */
76     @Override public abstract String getIconFileName();
77
78     /**
79      * @deprecated as of 1.350
80      *      Use {@link #obtainLog()}, which returns the same object in a more type-safe signature.
81      */
82     @Deprecated
83     public LargeText getLog() {
84         return obtainLog();
85     }
86
87     /**
88      * Obtains the log file.
89      *
90      * <p>
91      * The default implementation get this from {@link #workerThread},
92      * so when it's complete, the log could be gone any time.
93      *
94      * <p>
95      * Derived classes that persist the text should override this
96      * method so that it fetches the file from disk.
97      */
98     public AnnotatedLargeText obtainLog() {
99         WeakReference<AnnotatedLargeText> l = log;
100        if (l == null) return null;
```

File | /Users/neha.patil/Desktop/Projects/261P/Model%20-%20coverage/ns-1/sources/source-76.html

```
96      * method so that it fetches the file from disk.
97      */
98     public AnnotatedLargeText obtainLog() {
99         WeakReference<AnnotatedLargeText> l = log;
100        if (l == null) return null;
101        return l.get();
102    }
103
104    @Override
105    public String getSearchUrl() {
106        return getUrlName();
107    }
108
109    public TaskThread getWorkerThread() {
110        return workerThread;
111    }
112
113    /**
114     * Handles incremental log output.
115     */
116    public void doProgressiveLog(StaplerRequest req, StaplerResponse rsp) throws IOException {
117        AnnotatedLargeText text = obtainLog();
118        if (text != null) {
119            text.doProgressText(req, rsp);
120            return;
121        }
122        rsp.setStatus(HttpServletResponse.SC_OK);
123    }
124
125    /**
126     * Handles incremental log output.
127     */
128    public void doProgressiveHtml(StaplerRequest req, StaplerResponse rsp) throws IOException {
129        AnnotatedLargeText text = obtainLog();
130        if (text != null) {
131            text.doProgressiveHtml(req, rsp);
132            return;
133        }
134        rsp.setStatus(HttpServletResponse.SC_OK);
135    }
136
137    /**
138     * Clears the error status.
139     */
140    @RequirePOST
141    public synchronized void doClearError(StaplerRequest req, StaplerResponse rsp) throws IOException, ServletException {
142        getACL().checkPermission(getPermission());
143
144        if (workerThread != null && !workerThread.isRunning())
145            workerThread = null;
146        rsp.sendRedirect(".");
147    }
```

File | /Users/neha.patil/Desktop/Projects/261P/Model%20-%20coverage/ns-1/sources/source-78.html

Current scope: all classes | hudson.model

Coverage Summary for Class: TaskThread (hudson.model)

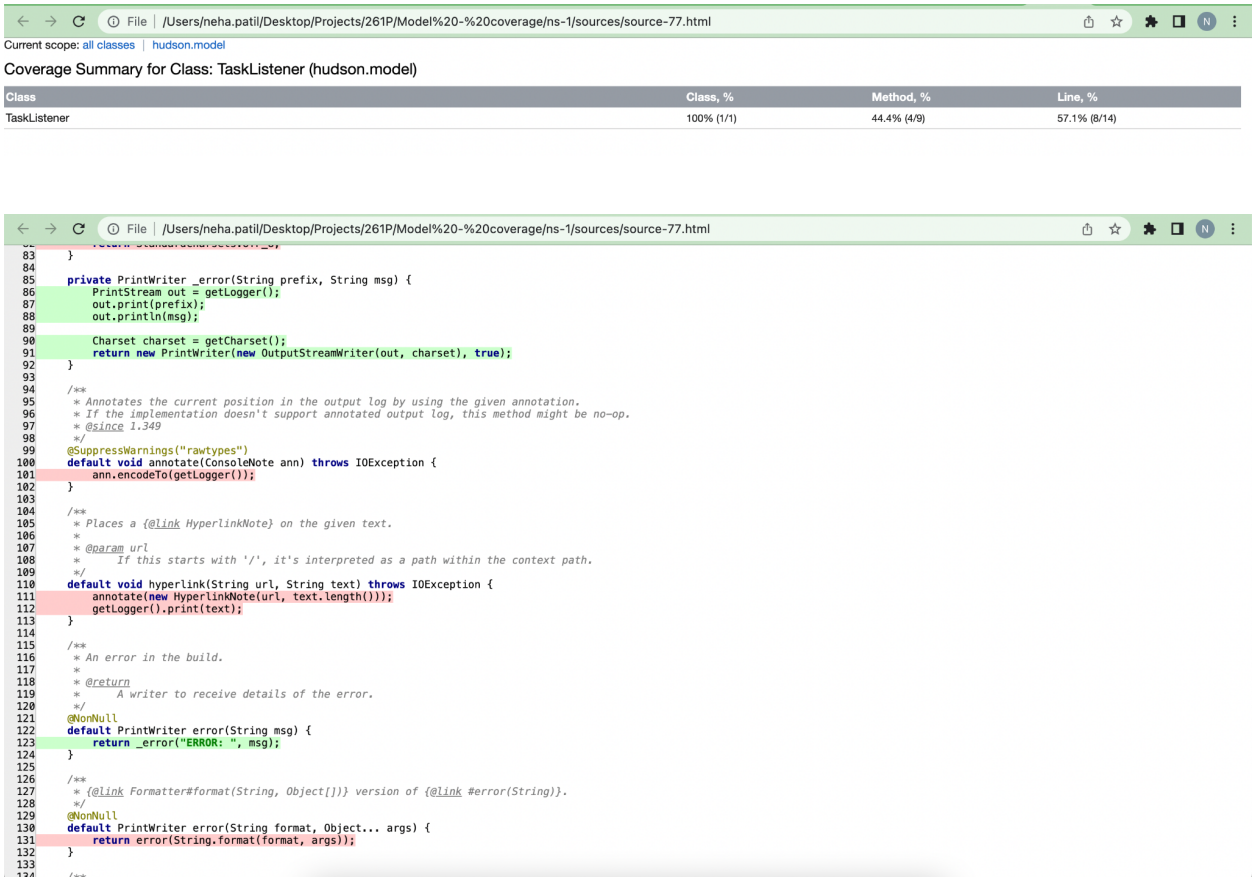
Class	Method, %	Line, %
TaskThread	0% (0/7)	0% (0/24)
TaskThread$ListenerAndText	0% (0/5)	0% (0/10)
Total	0% (0/12)	0% (0/34)

File | /Users/neha.patil/Desktop/Projects/261P/Model%20-%20coverage/ns-1/sources/source-78.html

```
 */
public abstract class TaskThread extends Thread {
    /**
     * @deprecated as of Hudson 1.350
     *      Use {@link #log}. It's the same object, in a better type.
     */
    @Deprecated
    private final LargeText text;

    /**
     * Represents the output from this task thread.
     */
    private final AnnotatedLargeText<TaskAction> log;

    /**
     * Represents the interface to produce output.
     */
    private TaskListener listener;

    private final TaskAction owner;

    private volatile boolean isRunning;

    /**
     *
     * @param output
     *      Determines where the output from this task thread goes.
     */
    protected TaskThread(TaskAction owner, ListenerAndText output) {
        //FIXME this failed to compile super(owner.getBuild().toString()+' '+owner.getDisplayName());
        //Please implement more general way how to get information about action owner,
        //if you want it in the thread's name.
        super(owner.getDisplayName());
        this.owner = owner;
        this.text = this.log = output.text;
        this.listener = output.listener;
        this.isRunning = true;
    }

    public Reader readAll() throws IOException {
        // this method can be invoked from another thread.
        return text.readAll();
    }

    /**
     * Registers that this {@link TaskThread} is run for the specified
     * {@link TaskAction}. This can be explicitly called from subtypes
     * to associate a single {@link TaskThread} across multiple tag actions.
     */
    protected final void associateWith(TaskAction action) {
        action.workerThread = this;
        action.log = new WeakReference<>(log);
    }
```

File | /Users/neha.patil/Desktop/Projects/261P/Model%20-%20coverage/ns-1/sources/source-78.html

```
        final TaskListener listener;
        final AnnotatedLargeText<TaskAction> text;

        public ListenerAndText(TaskListener listener, AnnotatedLargeText<TaskAction> text) {
            this.listener = listener;
            this.text = text;
        }

        /**
         * @deprecated as of Hudson 1.350
         *      Use {@link #forMemory(TaskAction)} and pass in the calling {@link TaskAction}
         */
        @Deprecated
        public static ListenerAndText forMemory() {
            return forMemory(null);
        }

        /**
         * @deprecated as of Hudson 1.350
         *      Use {@link #forFile(File, TaskAction)} and pass in the calling {@link TaskAction}
         */
        @Deprecated
        public static ListenerAndText forFile(File f) throws IOException {
            return forFile(f, null);
        }

        /**
         * Creates one that's backed by memory.
         */
        public static ListenerAndText forMemory(TaskAction context) {
            // StringWriter is synchronized
            ByteBuffer log = new ByteBuffer();

            return new ListenerAndText(
                new StreamTaskListener(log),
                new AnnotatedLargeText<>(log, Charset.defaultCharset(), false, context)
            );
        }

        /**
         * Creates one that's backed by a file.
         */
        public static ListenerAndText forFile(File f, TaskAction context) throws IOException {
            return new ListenerAndText(
                new StreamTaskListener(f),
                new AnnotatedLargeText<>(f, Charset.defaultCharset(), false, context)
            );
        }
    }
}
```



File | /Users/neha.patil/Desktop/Projects/261P/Model%20-%20coverage/ns-1/sources/source-77.html

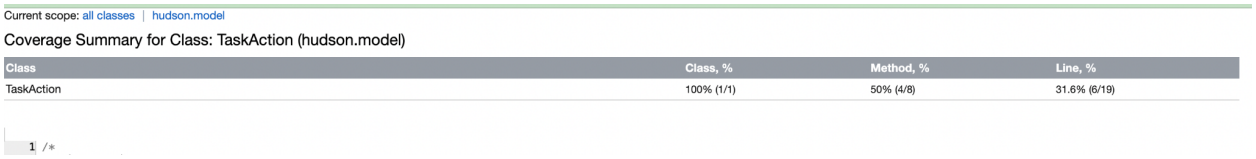
Current scope: all classes | hudson.model

Coverage Summary for Class: TaskListener (hudson.model)

Class	Class, %	Method, %	Line, %
TaskListener	100% (1/1)	44.4% (4/9)	57.1% (8/14)

```
    }

    private PrintWriter _error(String prefix, String msg) {
        PrintStream out = getLogger();
        out.print(prefix);
        out.println(msg);

        Charset charset = getCharset();
        return new PrintWriter(new OutputStreamWriter(out, charset), true);
    }

    /**
     * Annotates the current position in the output log by using the given annotation.
     * If the implementation doesn't support annotated output log, this method might be no-op.
     * @since 1.349
     */
    @SuppressWarnings("rawtypes")
    default void annotate(ConsoleNote ann) throws IOException {
        ann.encodeTo(getLogger());
    }

    /**
     * Places a {@link HyperlinkNote} on the given text.
     *
     * @param url
     *      If this starts with '/', it's interpreted as a path within the context path.
     */
    default void hyperlink(String url, String text) throws IOException {
        annotate(new HyperlinkNote(url, text.length()));
        getLogger().print(text);
    }

    /**
     * An error in the build.
     *
     * @return
     *      A writer to receive details of the error.
     */
    @NonNull
    default PrintWriter error(String msg) {
        return _error("ERROR: ", msg);
    }

    /**
     * {@link Formatter#format(String, Object[])} version of {@link #error(String)}.
     */
    @NonNull
    default PrintWriter error(String format, Object... args) {
        return error(String.format(format, args));
    }
```

After: Test coverage increased by 6 lines in TaskAction.java
23 lines in TaskThread and 3 lines in TaskListener file.



Current scope: all classes | hudson.model

Coverage Summary for Class: TaskAction (hudson.model)

Class	Class, %	Method, %	Line, %
TaskAction	100% (1/1)	50% (4/8)	31.6% (6/19)

```
1 /*
```

File | /Users/neha.patil/Desktop/Projects/261P/TaskActionTest/ns-1/sources/source-76.html

```
49  */
50  public abstract class TaskAction extends AbstractModelObject implements Action {
51      /**
52       * If non-null, that means either the activity is in progress
53       * asynchronously, or it failed unexpectedly and the thread is dead.
54       */
55      protected transient volatile TaskThread workerThread;
56
57      /**
58       * Hold the log of the tagging operation.
59       */
60      protected transient WeakReference<AnnotatedLargeText> log;
61
62      /**
63       * Gets the permission object that represents the permission (against {@link #getACL}) to perform this task.
64       * Generally your implementation of {@link #getIconFileName} should return null if {@code !getACL().hasPermission2(getPermission())}.
65       */
66      protected abstract Permission getPermission();
67
68      /**
69       * Gets the {@link ACL} against which {@link #getPermission} is checked.
70       */
71      protected abstract ACL getACL();
72
73      /**
74       * @see #getPermission
75       */
76      @Override public abstract String getIconFileName();
77
78      /**
79       * @deprecated as of 1.350
80       *      Use {@link #obtainLog()}, which returns the same object in a more type-safe signature.
81       */
82      @Deprecated
83      public LargeText getLog() {
84          return obtainLog();
85      }
86
87      /**
88       * Obtains the log file.
89       *
90       * <p>
91       * The default implementation get this from {@link #workerThread},
92       * so when it's complete, the log could be gone any time.
93       *
94       * <p>
95       * Derived classes that persist the text should override this
96       * method so that it fetches the file from disk.
97       */
98      public AnnotatedLargeText obtainLog() {
99          WeakReference<AnnotatedLargeText> l = log;
100         if (l == null) return null;
101         return l.get();
102     }
103
104     @Override
105     public String getSearchUrl() {
106         return getUrlName();
107     }
108
109     public TaskThread getWorkerThread() {
110         return workerThread;
111     }
```

File | /Users/neha.patil/Desktop/Projects/261P/TaskActionTest/ns-1/sources/source-78.html

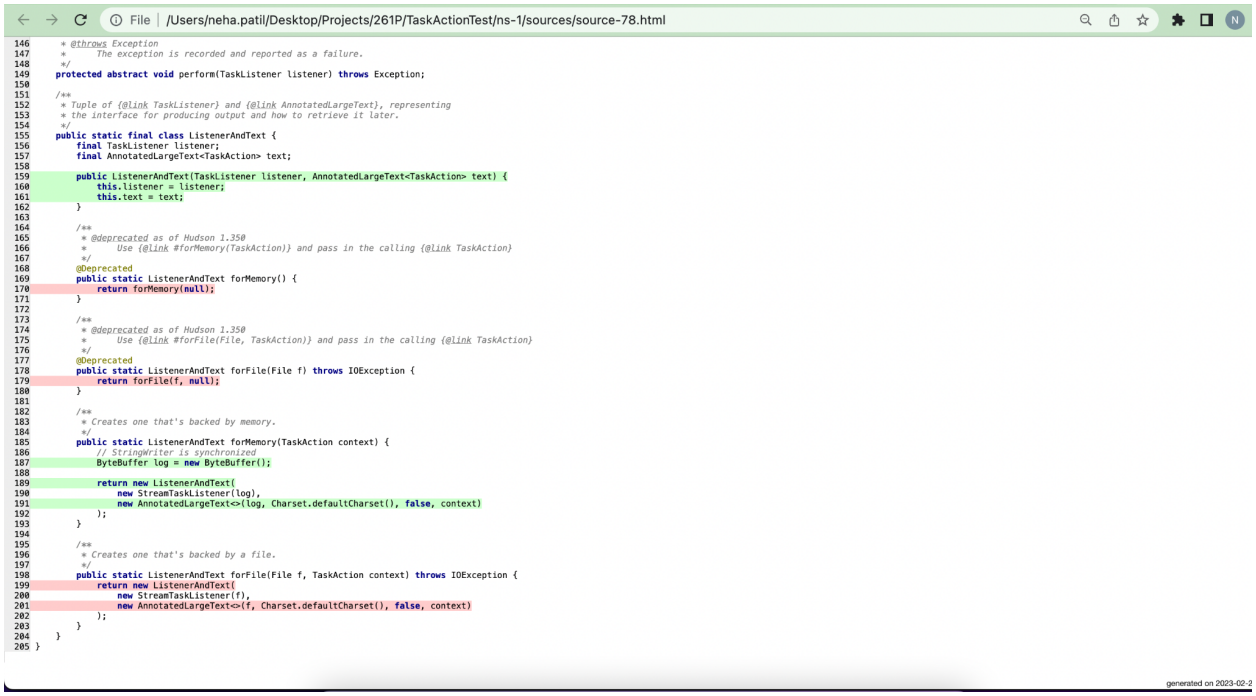
Current scope: all classes | hudson.model

Coverage Summary for Class: TaskThread (hudson.model)

Class	Method, %	Line, %
TaskThread	57.1% (4/7)	70.8% (17/24)
TaskThread$ListenerAndText	40% (2/5)	60% (6/10)
Total	50% (6/12)	67.6% (23/34)

File | /Users/neha.patil/Desktop/Projects/261P/TaskActionTest/ns-1/sources/source-78.html

```
77      protected TaskThread(TaskAction owner, ListenerAndText output) {
78          //FIXME this failed to compile super(owner.getBuild().toString()+' '+owner.getDisplayName());
79          //Please implement more general way how to get information about action owner,
80          //if you want it in the thread's name.
81          super(owner.getDisplayName());
82          this.owner = owner;
83          this.text = this.log = output.text;
84          this.listener = output.listener;
85          this.isRunning = true;
86      }
87
88      public Reader readAll() throws IOException {
89          // this method can be invoked from another thread.
90          return text.readAll();
91      }
92
93      /**
94       * Registers that this {@link TaskThread} is run for the specified
95       * {@link TaskAction}. This can be explicitly called from subtypes
96       * to associate a single {@link TaskThread} across multiple tag actions.
97       */
98      protected final void associateWith(TaskAction action) {
99          action.workerThread = this;
100         action.log = new WeakReference<>(log);
101     }
102
103     /**
104      * Starts the task execution asynchronously.
105      */
106     @Override
107     public void start() {
108         associateWith(owner);
109         super.start();
110     }
111
112     public boolean isRunning() {
113         return isRunning;
114     }
115
116     /**
117      * Determines where the output of this {@link TaskThread} goes.
118      * <p>
119      * Subclass can override this to send the output to a file, for example.
120      */
121     protected ListenerAndText createListener() throws IOException {
122         return ListenerAndText.forMemory();
123     }
124
125     @Override
126     public final void run() {
127         isRunning = true;
128         try {
129             perform(listener);
130             listener.getLogger().println("Completed");
131             owner.workerThread = null;
132         } catch (InterruptedException e) {
133             listener.getLogger().println("Aborted");
134         } catch (Exception e) {
135             Functions.printStackTrace(e, listener.getLogger());
136         } finally {
137             listener = null;
138             isRunning = false;
139         }
140         log.markAsComplete();
```



```
File | /Users/neha.patil/Desktop/Projects/261P/TaskActionTest/ns-1/sources/source-78.html

146     * @throws Exception
147     *      The exception is recorded and reported as a failure.
148     */
149    protected abstract void perform(TaskListener listener) throws Exception;
150
151    /**
152     * Tuple of {@link TaskListener} and {@link AnnotatedLargeText}, representing
153     * the interface for producing output and how to retrieve it later.
154     */
155    public static final class ListenerAndText {
156        final TaskListener listener;
157        final AnnotatedLargeText<TaskAction> text;
158
159        public ListenerAndText(TaskListener listener, AnnotatedLargeText<TaskAction> text) {
160            this.listener = listener;
161            this.text = text;
162        }
163
164        /**
165         * @deprecated as of Hudson 1.350
166         *      Use {@link #forMemory(TaskAction)} and pass in the calling {@link TaskAction}
167         */
168        @Deprecated
169        public static ListenerAndText forMemory() {
170            return forMemory(null);
171        }
172
173        /**
174         * @deprecated as of Hudson 1.350
175         *      Use {@link #forFile(File, TaskAction)} and pass in the calling {@link TaskAction}
176         */
177        @Deprecated
178        public static ListenerAndText forFile(File f) throws IOException {
179            return forFile(f, null);
180        }
181
182        /**
183         * Creates one that's backed by memory.
184         */
185        public static ListenerAndText forMemory(TaskAction context) {
186            // StringWriter is synchronized
187            ByteBuffer log = new ByteBuffer();
188
189            return new ListenerAndText(
190                new StreamTaskListener(log),
191                new AnnotatedLargeText<>(log, Charset.defaultCharset(), false, context)
192            );
193        }
194
195        /**
196         * Creates one that's backed by a file.
197         */
198        public static ListenerAndText forFile(File f, TaskAction context) throws IOException {
199            return new ListenerAndText(
200                new StreamTaskListener(f),
201                new AnnotatedLargeText<>(f, Charset.defaultCharset(), false, context)
202            );
203        }
204    }
205 }
```

generated on 2023-02-2

```
File | /Users/neha.patil/Desktop/Projects/261P/TaskActionTest/ns-1/sources/source-77.html
```

Current scope: all classes | hudson.model

Coverage Summary for Class: TaskListener (hudson.model)

Class	Class, %	Method, %	Line, %
TaskListener	100% (1/1)	33.3% (3/9)	28.6% (4/14)

```
File | /Users/neha.patil/Desktop/Projects/261P/TaskActionTest/ns-1/sources/source-77.html

75      * A charset to use for methods returning {@link PrintWriter}.
76      * Should match that used to construct {@link #getLogger}.
77      * @return by default, UTF-8
78      */
79     @Restricted(ProtectedExternally.class)
80     @NonNull
81     default Charset getCharset() {
82         return StandardCharsets.UTF_8;
83     }
84
85     private PrintWriter _error(String prefix, String msg) {
86         PrintStream out = getLogger();
87         out.print(prefix);
88         out.println(msg);
89
90         Charset charset = getCharset();
91         return new PrintWriter(new OutputStreamWriter(out, charset), true);
92     }
93
94     /**
95      * Annotates the current position in the output log by using the given annotation.
96      * If the implementation doesn't support annotated output log, this method might be no-op.
97      * @since 1.349
98      */
99     @SuppressWarnings("rawtypes")
100    default void annotate(ConsoleNote ann) throws IOException {
101        ann.encodeTo(getLogger());
102    }
103
104    /**
105     * Places a {@link HyperlinkNote} on the given text.
106     *
107     * @param url
108     *      If this starts with '/', it's interpreted as a path within the context path.
109     */
110    default void hyperlink(String url, String text) throws IOException {
111        annotate(new HyperlinkNote(url, text.length()));
112        getLogger().print(text);
113    }
114
```

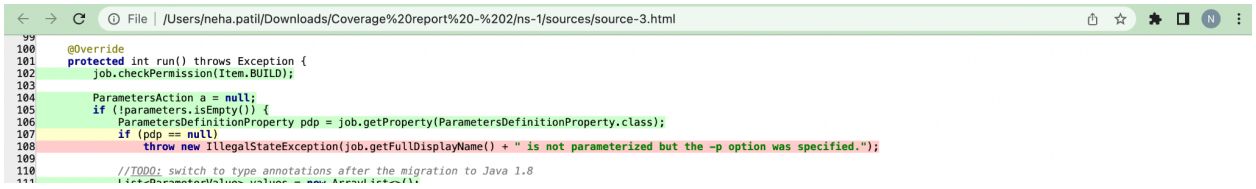
Explanation: TaskAction.java is an implementation of a partial action that initiates some work asynchronously. The class provides the fundamental collection of tools needed to carry it out. The TaskActionTest test suite tests a task that has started asynchronous and its corresponding functionality.

JUnit added:

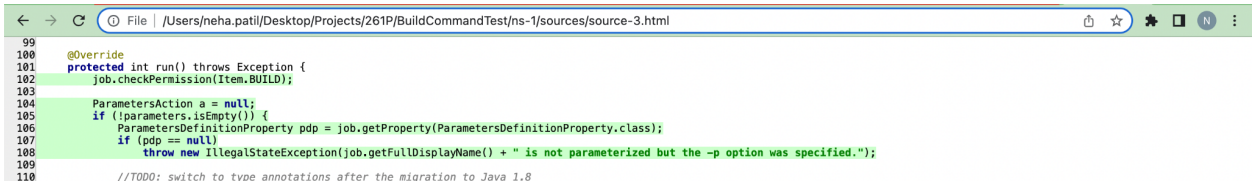
```java
import java.nio.charset.StandardCharsets;
import org.junit.Test;


public class TaskActionTest {

    1 usage
    private static class MyTestTaskThread extends TaskThread {...}

    2 usages
    private static class MyTestTaskAction extends TaskAction {...}

    @Test
    public void annotatedText() throws Exception {
        MyTestTaskAction action = new MyTestTaskAction();
        action.start();
        AnnotatedLargeText annotatedText = action.obtainLog();
        String url = action.getSearchUrl();
        TaskThread taskThread = action.getWorkerThread();

        assertEquals(url, actual: "xyz");
        assertEquals(taskThread.getName(), actual: "My Task Thread");

        while (!annotatedText.isComplete()) {
            Thread.sleep( millis: 10);
        }
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        final long length = annotatedText.writeLogTo( start: 0, os);

        assertTrue( message: "length should be longer or even 219", condition: length >= 219);
        assertTrue(os.toString(StandardCharsets.UTF_8).startsWith("a linkCompleted"));
    }
}
```

10.3 BuildCommand File

Before: The BuildCommand functionality is not tested for when parameters are not given but -p argument is specified. The lines 107 and 108 were not covered.



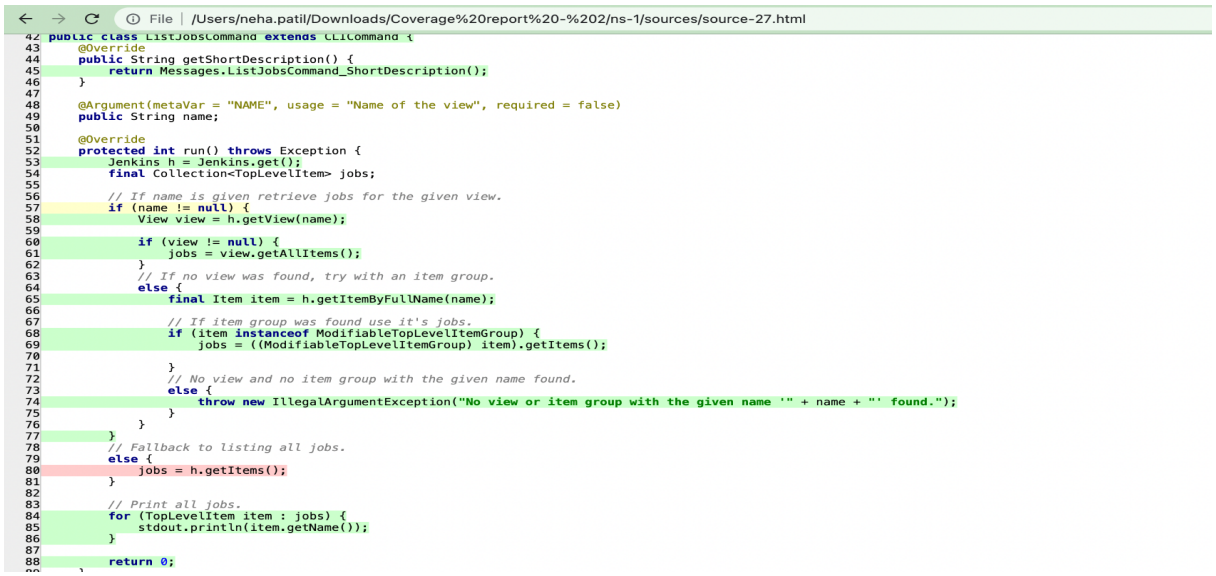
After: We added test scenarios to cover those 2 lines.



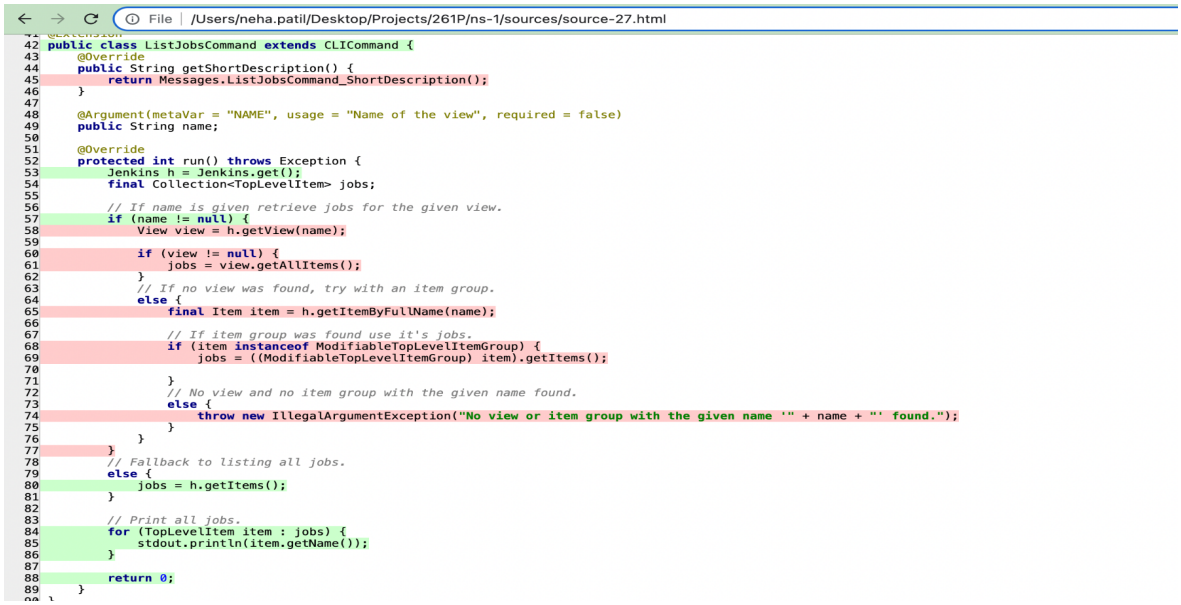
JUnitAdded:

10.4 ListJobsCommand File

Before: The fallback condition for listing jobs when no view name is specified was not tested. Else part of the code was not tested, that is line no. 80..



After: You can see below that line of code is covered.



JUnitAdded:

```
patil-neha-3
@Test public void getAllJobsWithoutViewName() throws Exception {
    MockFolder folder = j.createFolder( name: "Folder");
    MockFolder nestedFolder = folder.createProject(MockFolder.class, name: "NestedFolder");
    FreeStyleProject job = folder.createProject(FreeStyleProject.class, name: "job");
    FreeStyleProject nestedJob = nestedFolder.createProject(FreeStyleProject.class, name: "nestedJob");

    ListView view = new ListView( name: "OuterFolder");
    view.setRecurse(true);
    j.jenkins.addView(view);

    ((DirectlyModifiableView) j.jenkins.getView( name: "OuterFolder")).add(folder);
    ((DirectlyModifiableView) j.jenkins.getView( name: "OuterFolder")).add(job);

    // use invoke as opposed to invokeWithArgs to test ListJobsCommand with no view name.
    CLICommandInvoker.Result result = command.invoke();
    assertThat(result, CLICommandInvoker.Matcher.succeeded());
    assertThat(result.stdout(), containsString( substring: "Folder"));
}
```

12. Part 11: Continuous Integration

With a continuous integration method, developers add new code they've written to the code base at least once per day, integrating it more frequently during the development cycle. Each iteration of the build is subjected to automated testing in order to find integration problems earlier, when they are simpler to fix, and to help prevent issues at the release's final merging. Overall, continuous integration contributes to the build process' streamlining, producing software of greater quality and more reliably meeting delivery deadlines.

When you create a continuous integration build and a package, continuous testing can start (also known as an installable entity or packaged entity). As that bundled item enters production, it comes to an end. A testing suite is used at every stage from beginning to conclusion.[10]

When there is only one test step, testing accounts for at least 30% of continuous integration. In reality, testing makes up between 50% and 70% of continuous integration operations. Formerly, testing had to be finished manually. You can now employ automated tests, which are essential for continuous integration to succeed.

Test-driven development iteratively creates code and tests one use case at a time as part of automated testing for continuous integration in order to assure test coverage, enhance code quality, and lay the foundation for continuous delivery. If new code fails one or more of the tests created across all functional areas of the application, automated testing will let you know. Developers should run all or a subset of tests locally as it is best practice to only commit source code to version control after tests have passed for the new code changes. Effective regression testing, according to experience, can assist prevent subsequent unpleasant shocks.

11.1 The continuous integration pipeline

Continuous integration pipelines automates project pipeline stages including builds, tests, and deploys in a repeatable manner with little human involvement. By offering controls, checkpoints, and speed, an automated continuous integration pipeline is crucial to streamlining the development, testing, and deployment of any application.

11.2 Continuous integration best practices

A crucial part of DevOps is the continuous integration process, which enables organizations to combine the development and operations teams into one repository for developing, testing, deploying, and maintaining software. Some CI best practices that can assist business are the ones listed below:

1. Maintain a single source code repository: To organize and track every file needed to develop a product, use source control management. Distribution and visibility are made simpler by the consolidated code base.

2. Automate the build: Compiling, linking, and other procedures that result in the build artifacts are involved in this. Moreover, self-testing needs to be automated.
3. Use daily mainline commits: Make it mandatory for developers to commit their changes at least once per day to the main development stream. It is the responsibility of each developer to ensure that their working copy is in sync with the main development stream.
4. Test in a clone of the production environment: Try to replicate the final production environment as closely as possible in the testing environment.
5. Automate deployment: To perform builds and tests, implement various environments (development, integration, and production).

11.3 Benefits of Continuous Integration

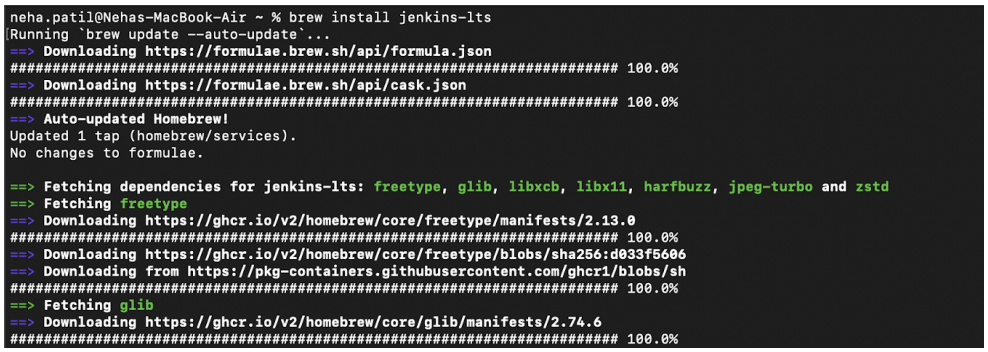
Some common benefits of the Continuous Integration are[10]:

1. Metrics that enable early error detection and resolution, often within minutes of check-in, are improved.
2. Reduces overhead across the development and deployment process
3. Enables a quick feedback mechanism on every change
4. Improved team collaboration; everyone on the team can change the code, integrate the system and quickly determine conflicts with other parts of the software.
5. Enhancing system integration lowers surprises at the conclusion of the software development lifecycle.
6. A less number of simultaneous modifications for merging and testing.
7. Decreased number of failures during system testing.
8. Testing against constantly updated systems

11.4 Sign up for TravisCI or CircleCI or GitHub Actions (or install Jenkins or other CI system)

Installing Jenkins locally:

1. Using Homebrew to install the latest LTS version of Jenkins:
   brew install jenkins-lts



```
==> Caveats
==> jenkins-lts
Note: When using launchctl the port will be 8080.

To start jenkins-lts now and restart at login:
  brew services start jenkins-lts
neha.patil@Nehas-MacBook-Air ~ % brew services start jenkins-lts
==> Downloading https://formulae.brew.sh/api/formula.json
######################################################################## 100.0%
Bootstrap failed: 5: Input/output error
Try re-running the command as root for richer errors.
Error: Failure while executing; `/bin/launchctl bootstrap gui/501 /Users/neha.patil/Library/LaunchAgents/homebrew.mxcl.jenkins-lts.plist`
 exited with 5.
neha.patil@Nehas-MacBook-Air ~ % sudo brew services start jenkins-lts
Password:
Warning: Taking root:admin ownership of some jenkins-lts paths:
  /opt/homebrew/Cellar/openjdk@17/17.0.6/libexec/openjdk.jdk/Contents/Home/bin
  /opt/homebrew/Cellar/openjdk@17/17.0.6/libexec/openjdk.jdk/Contents/Home/bin/java
  /opt/homebrew/opt/jenkins-lts
  /opt/homebrew/opt/jenkins-lts/bin
  /opt/homebrew/var/homebrew/linked/jenkins-lts
This will require manual removal of these paths using `sudo rm` on
brew upgrade/reinstall/uninstall.
Warning: jenkins-lts must be run as non-root to start at user login!
==> Successfully started `jenkins-lts` (label: homebrew.mxcl.jenkins-lts)
```

2. Now to start the jenkins service enter the following in the terminal:
   brew services start jenkins-lts

   While starting the jenkins service, we encountered an error which stated:
   Error: Failure while executing; `/bin/launchctl bootstrap gui/501 /Users/neha.patil/Library/LaunchAgents/homebrew.mxcl.jenkins-lts.plist` exited with 5.
   To fix it, we modified the service start command to run as root user
   sudo brew services start jenkins-lts

3. Jenkins service, by default, runs on port 8080. We can change the default port by modifying the *~/Library/LaunchAgents/homebrew.mxcl.jenkins-lts.plist* file. In our case we have changed the port to 8888 as another service was up and running on port 8080.

homebrew.mxcl.jenkins-lts.plist

opt > homebrew > Cellar > jenkins-lts > 2.375.3 > homebrew.mxcl.jenkins-lts.plist

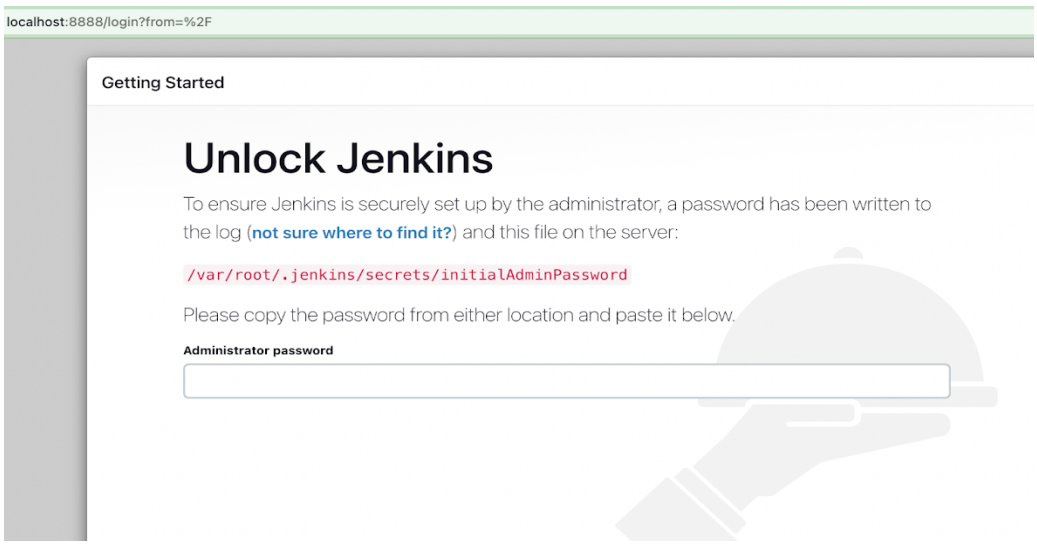
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
3  <plist version="1.0">
4  <dict>
5      <key>Label</key>
6      <string>homebrew.mxcl.jenkins-lts</string>
7      <key>LimitLoadToSessionType</key>
8      <array>
9          <string>Aqua</string>
10         <string>Background</string>
11         <string>LoginWindow</string>
12         <string>StandardIO</string>
13         <string>System</string>
14     </array>
15     <key>ProgramArguments</key>
16     <array>
17         <string>/opt/homebrew/opt/openjdk@17/bin/java</string>
18         <string>-Dmail.smtp.starttls.enable=true</string>
19         <string>-jar</string>
20         <string>/opt/homebrew/opt/jenkins-lts/libexec/jenkins.war</string>
21         <string>--httpListenAddress=127.0.0.1</string>
22         <string>--httpPort=8888</string>
23     </array>
```

4. The next step is to unlock Jenkins by entering the initial admin password that can be found in the ~/.jenkins/secrets/initialAdminPassword file.

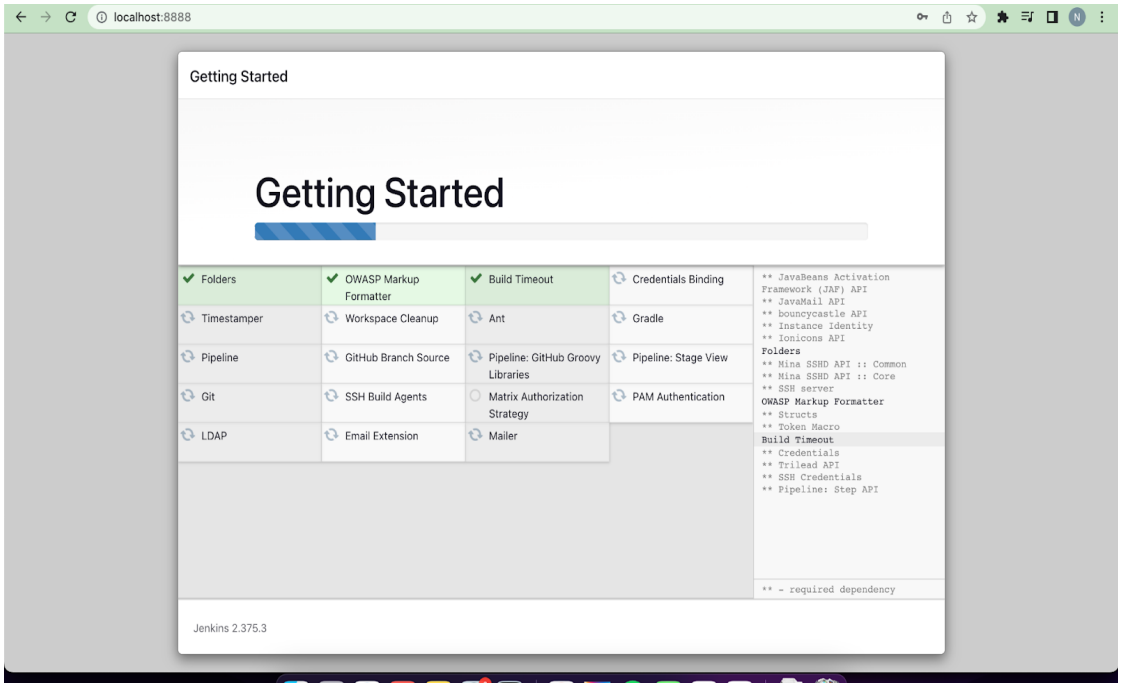
   We encountered a problem with this step as the .jenkins folder was not created on the local system. After some google searches, we found a work around for this. The command sudo jenkins-lts helped fix this.

```
neha.patil@Nehas-MacBook-Air ~ % sudo jenkins-lts
Running from: /opt/homebrew/Cellar/jenkins-lts/2.375.3/libexec/jenkins.war
webroot: $user.home/.jenkins
2023-02-27 04:20:01.317+0000 [id=1]     INFO    winstone.Logger#logInternal: Beginning extraction from war file
2023-02-27 04:20:01.349+0000 [id=1]     WARNING o.e.j.s.handler.ContextHandler#setContextPath: Empty contextPath
2023-02-27 04:20:01.378+0000 [id=1]     INFO    org.eclipse.jetty.server.Server#doStart: jetty-10.0.12; built: 2022-09-14T01:54:40.076Z; git: 408d0139887e27a57b
d52e2d92a36731a7e88; jvm 17.0.6+0
2023-02-27 04:20:01.517+0000 [id=1]     INFO    o.e.j.w.StandardDescriptorProcessor#visitServlet: NO JSP Support for /, did not find org.eclipse.jetty.jsp.Jetty
Servlet
2023-02-27 04:20:01.542+0000 [id=1]     INFO    o.e.j.s.s.DefaultSessionIdManager#doStart: Session workerName=node0
2023-02-27 04:20:01.727+0000 [id=1]     INFO    hudson.WebAppMain#contextInitialized: Jenkins home directory: /var/root/.jenkins found at: $user.home/.jenkins
2023-02-27 04:20:02.361+0000 [id=1]     INFO    o.e.j.s.handler.ContextHandler#doStart: Started w.@2af616d3{Jenkins v2.375.3,/,file:///private/var/root/.jenkins
r/,AVAILABLE}{/var/root/.jenkins/war}
2023-02-27 04:20:02.371+0000 [id=1]     INFO    o.e.j.server.AbstractConnector#doStart: Started ServerConnector@7dc222ae{HTTP/1.1, (http/1.1)}{0.0.0.0:8080}
2023-02-27 04:20:02.375+0000 [id=1]     INFO    org.eclipse.jetty.server.Server#doStart: Started Server@4df828d7{STARTING}[10.0.12,sto=0] @1293ms
2023-02-27 04:20:02.376+0000 [id=28]    INFO    winstone.Logger#logInternal: Winstone Servlet Engine running: controlPort=disabled
2023-02-27 04:20:02.472+0000 [id=35]    INFO    jenkins.InitReactorRunner$1#onAttained: Started initialization
2023-02-27 04:20:02.501+0000 [id=48]    INFO    jenkins.InitReactorRunner$1#onAttained: Listed all plugins
2023-02-27 04:20:02.844+0000 [id=44]    INFO    jenkins.InitReactorRunner$1#onAttained: Prepared all plugins
2023-02-27 04:20:02.845+0000 [id=36]    INFO    jenkins.InitReactorRunner$1#onAttained: Started all plugins
2023-02-27 04:20:02.848+0000 [id=37]    INFO    jenkins.InitReactorRunner$1#onAttained: Augmented all extensions
2023-02-27 04:20:02.975+0000 [id=41]    INFO    jenkins.InitReactorRunner$1#onAttained: System config loaded
2023-02-27 04:20:02.975+0000 [id=41]    INFO    jenkins.InitReactorRunner$1#onAttained: System config adapted
2023-02-27 04:20:02.975+0000 [id=40]    INFO    jenkins.InitReactorRunner$1#onAttained: Loaded all jobs
2023-02-27 04:20:02.976+0000 [id=46]    INFO    jenkins.InitReactorRunner$1#onAttained: Configuration for all jobs updated
2023-02-27 04:20:03.038+0000 [id=37]    INFO    jenkins.install.SetupWizard#init:

*************************************************************
*************************************************************
*************************************************************

Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:

753e3c6cb5fa4ac29c1c9abb8600db97

This may also be found at: /var/root/.jenkins/secrets/initialAdminPassword

*************************************************************
*************************************************************
*************************************************************

2023-02-27 04:20:07.327+0000 [id=37]    INFO    jenkins.InitReactorRunner$1#onAttained: Completed initialization
2023-02-27 04:20:07.354+0000 [id=27]    INFO    hudson.lifecycle.Lifecycle#onReady: Jenkins is fully up and running
```

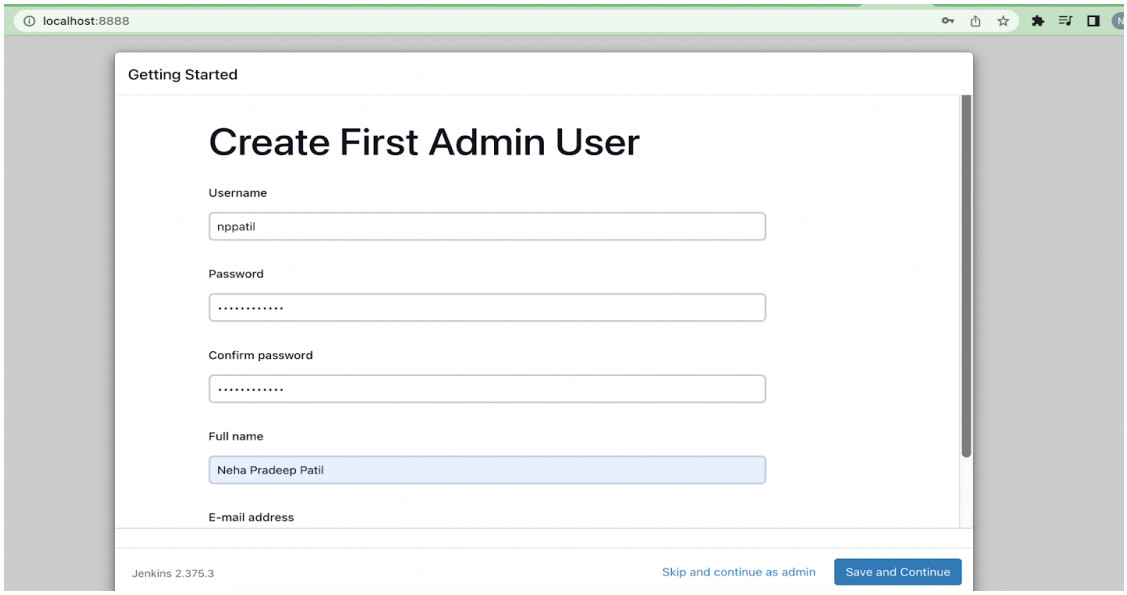
5. We can then enter the initial admin password at localhost:8888 and get started with the initial setup.



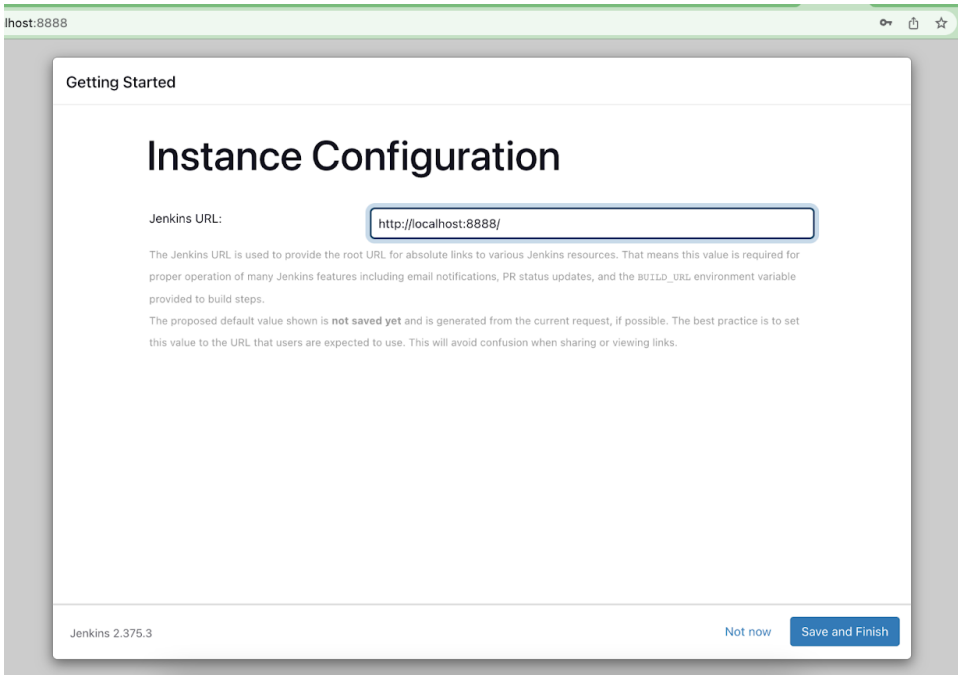
6. Install the necessary plugins



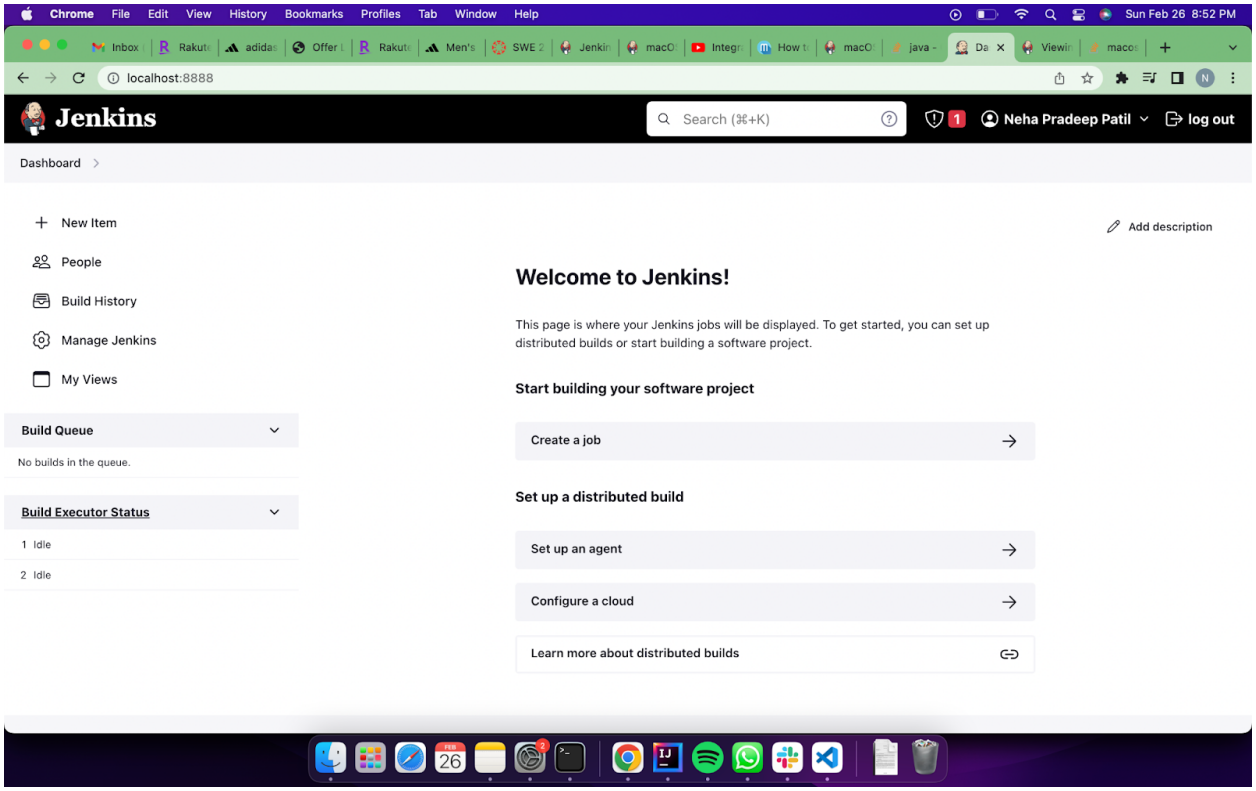
7. Create the first admin user



8. Configure the instance

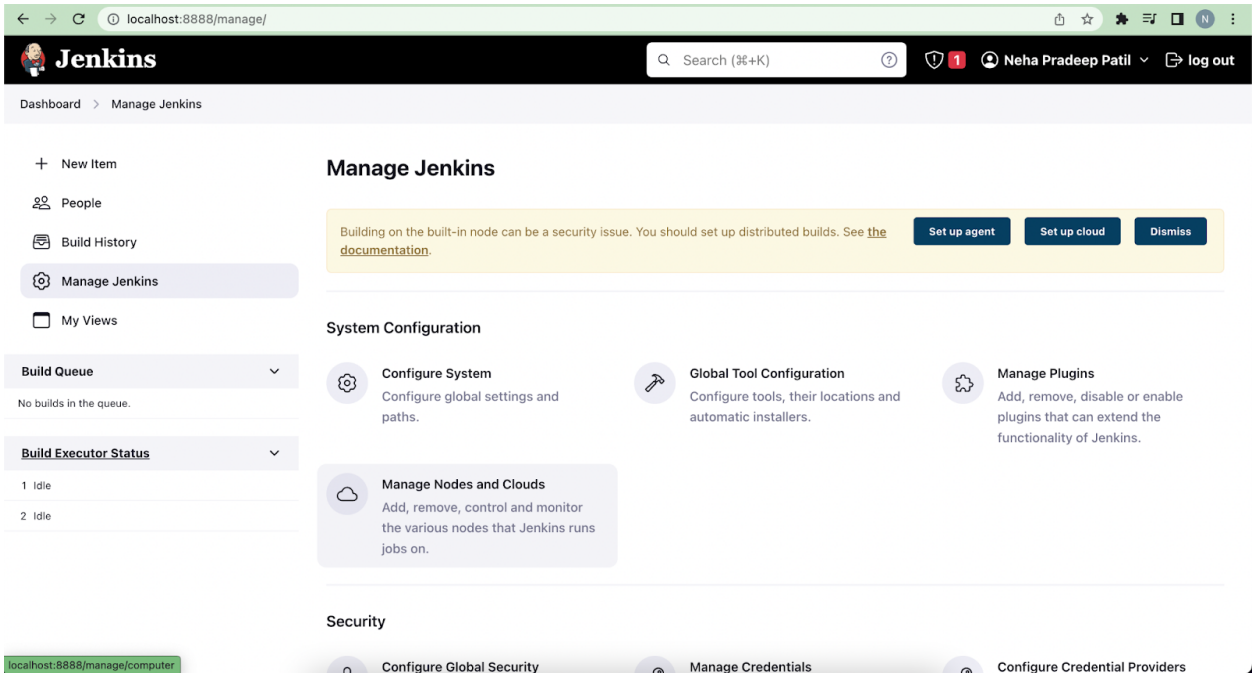


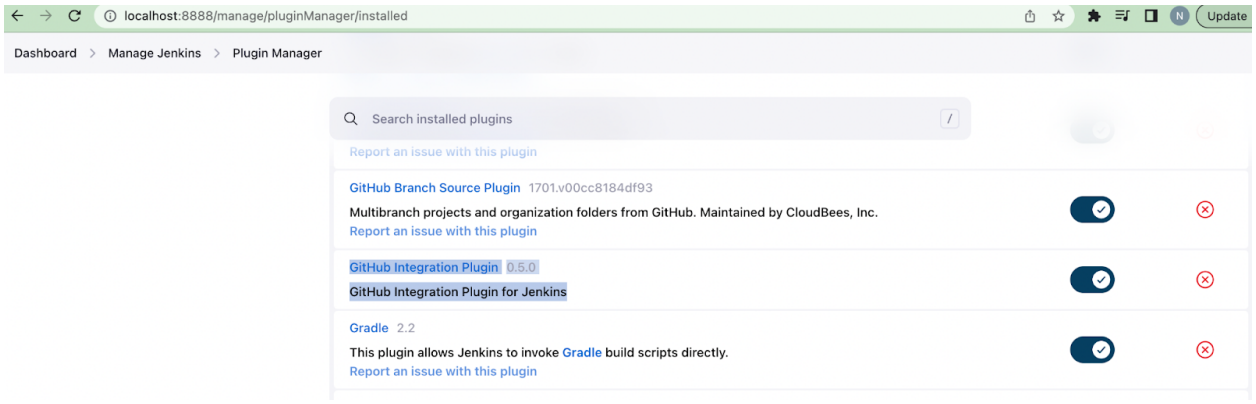
9. This is the homepage for the locally setup Jenkins service



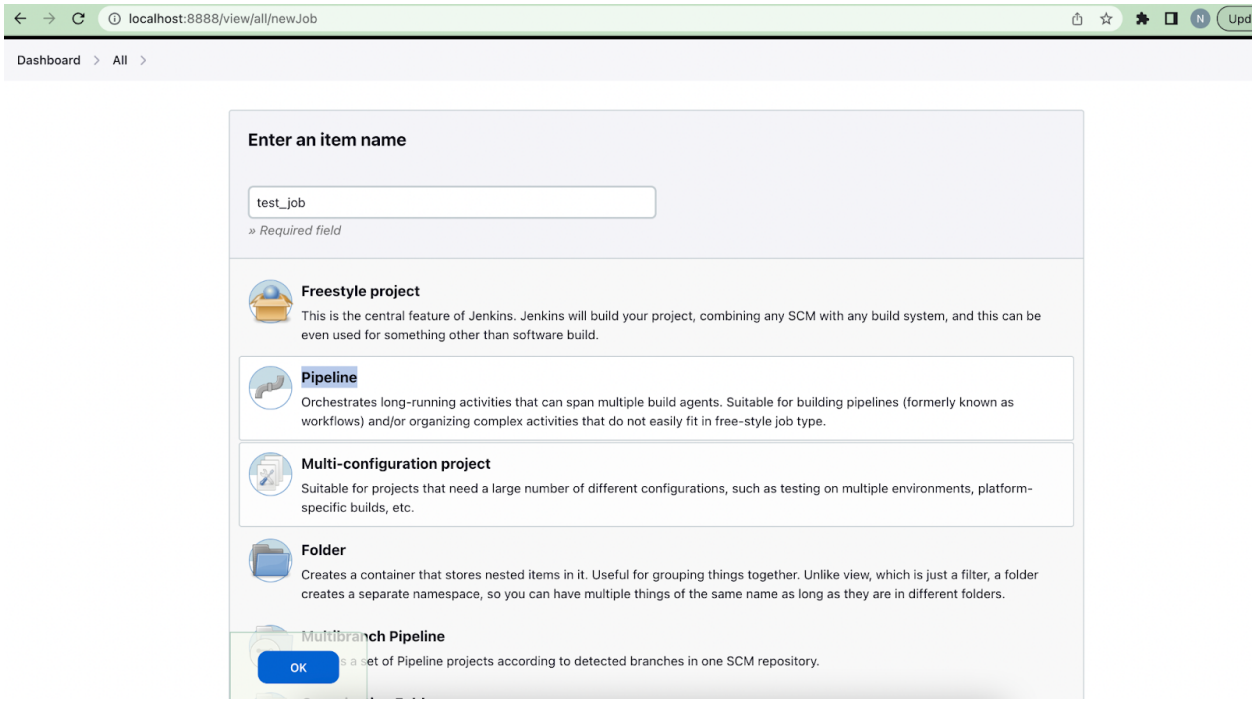
11.5 Integrating the chosen project (Jenkins) with a CI system (Jenkins)

1. Install the Github Integration Plugin which can be found in the ‘Manage Jenkins’ tab

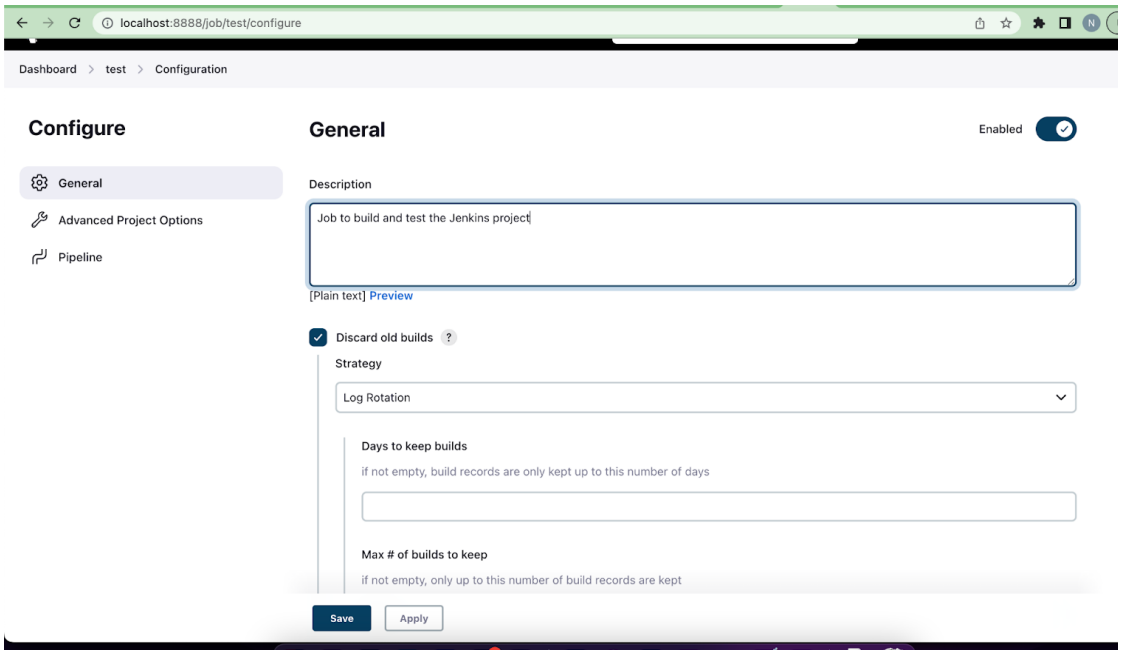




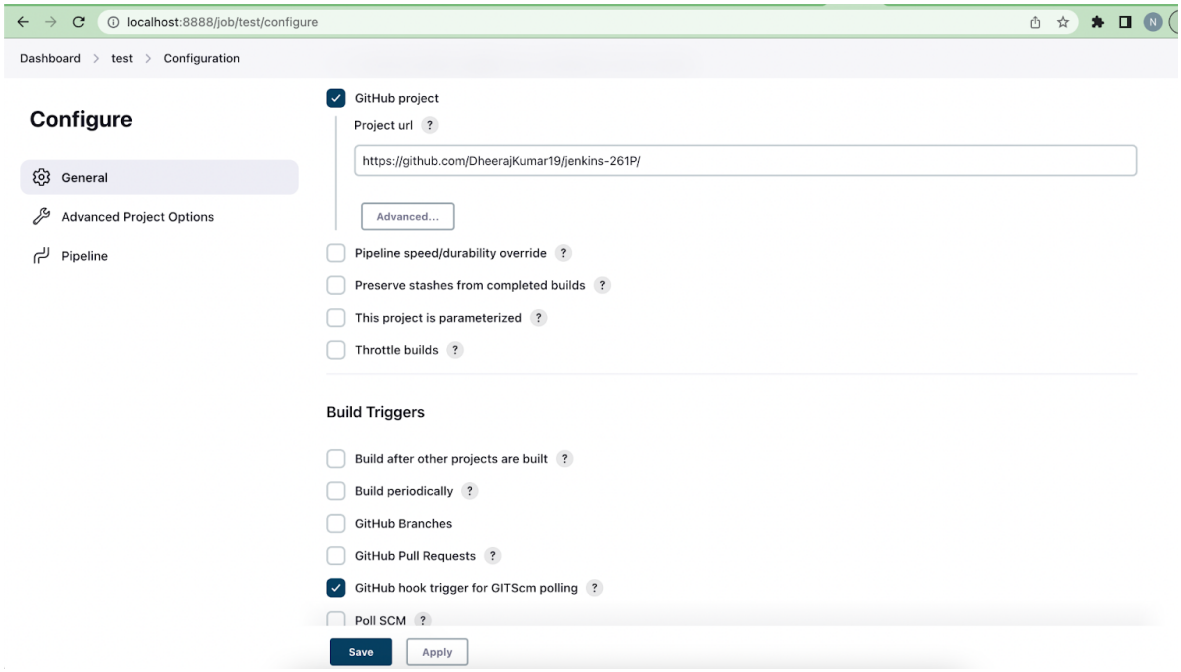
2. Create Jenkins job
   Enter a name for the jenkins job and select ‘Pipeline’ from the drop down.



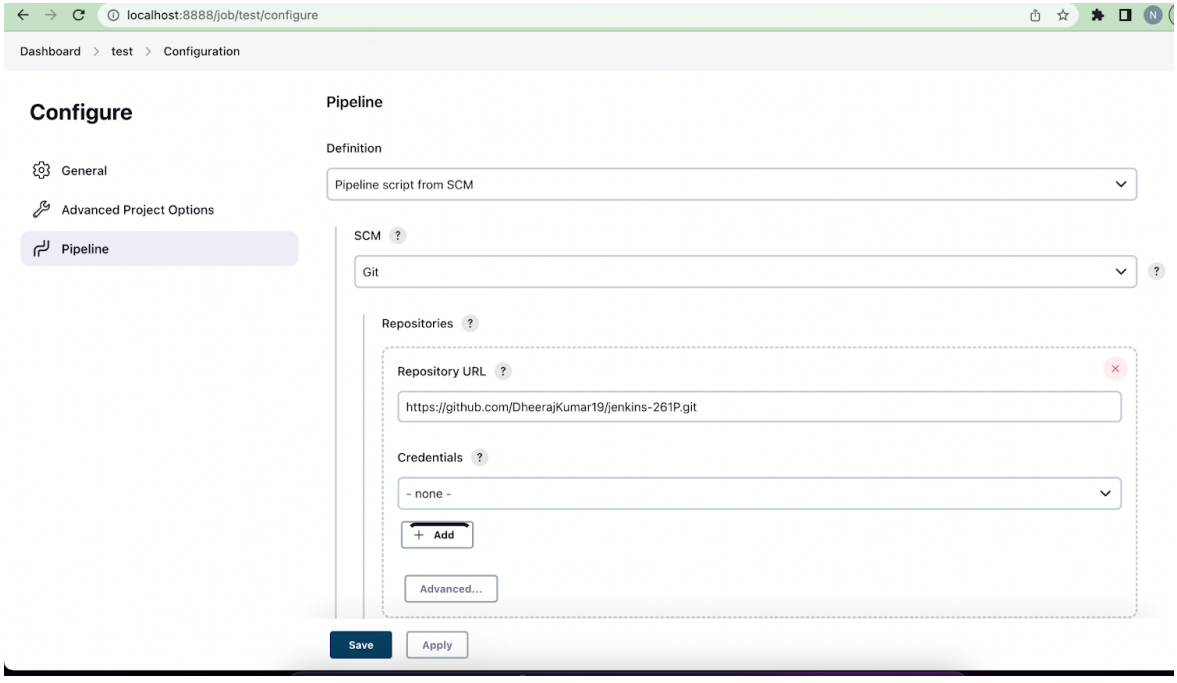
3. Configure the pipeline by giving appropriate description



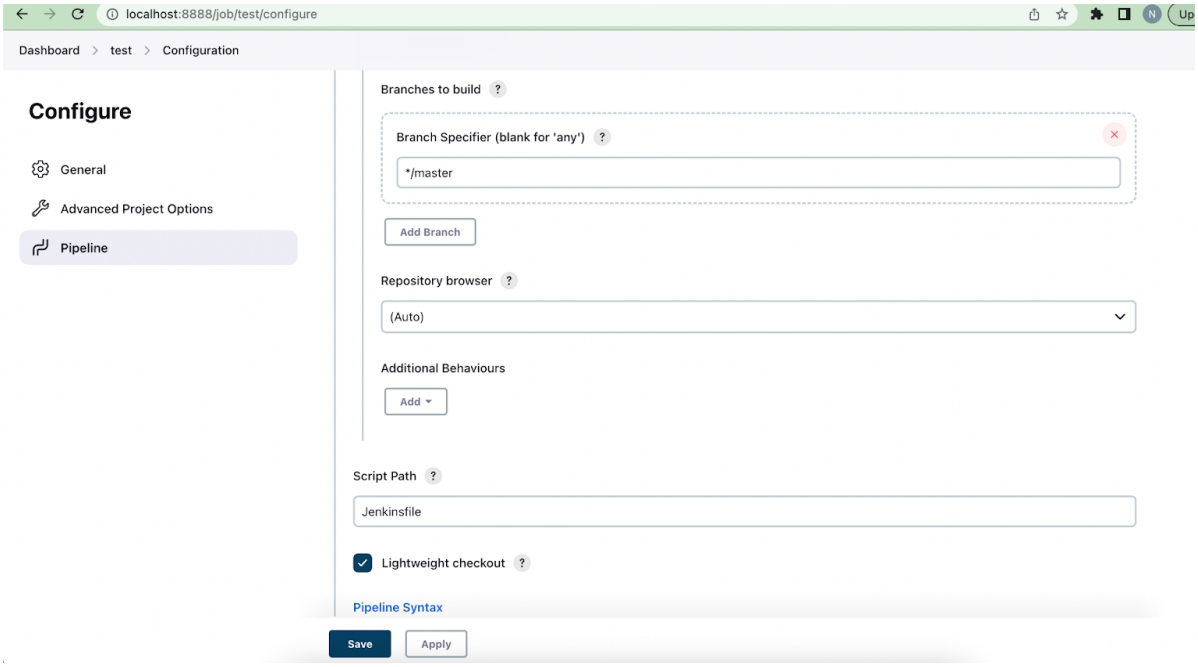
4. Add the github project URL and build triggers.



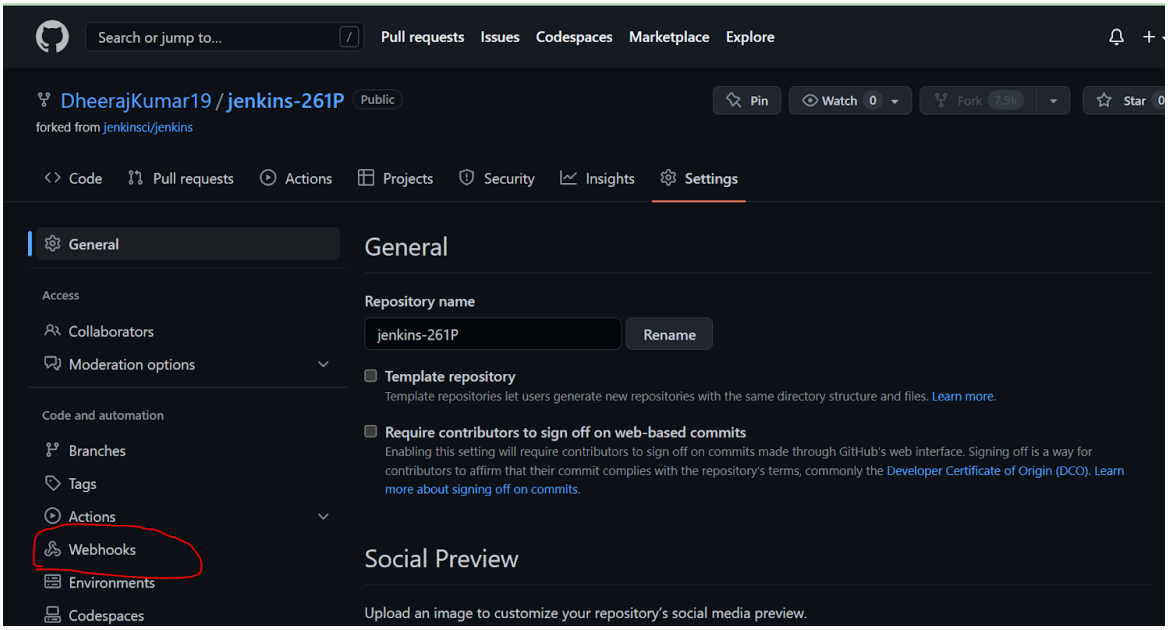
5. Configure the pipeline definition by selecting the ‘Pipeline script from SCM option. Enter the repo link

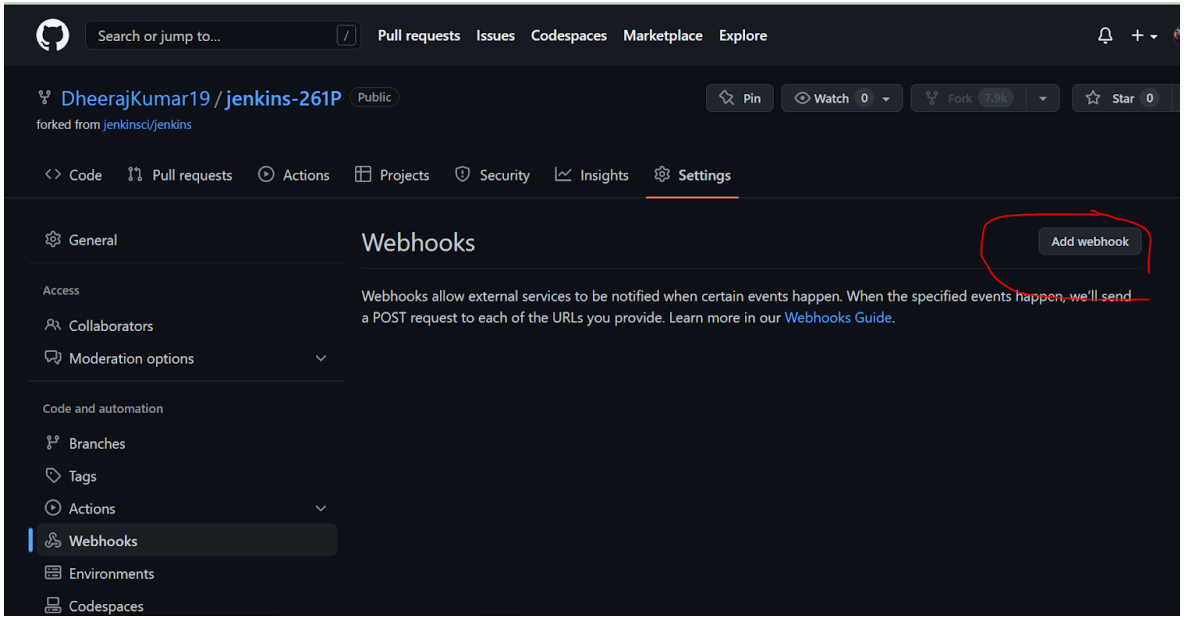


6. Enter the script path. Jenkinsfile is the file which specifies the groovy script to build the pipeline.

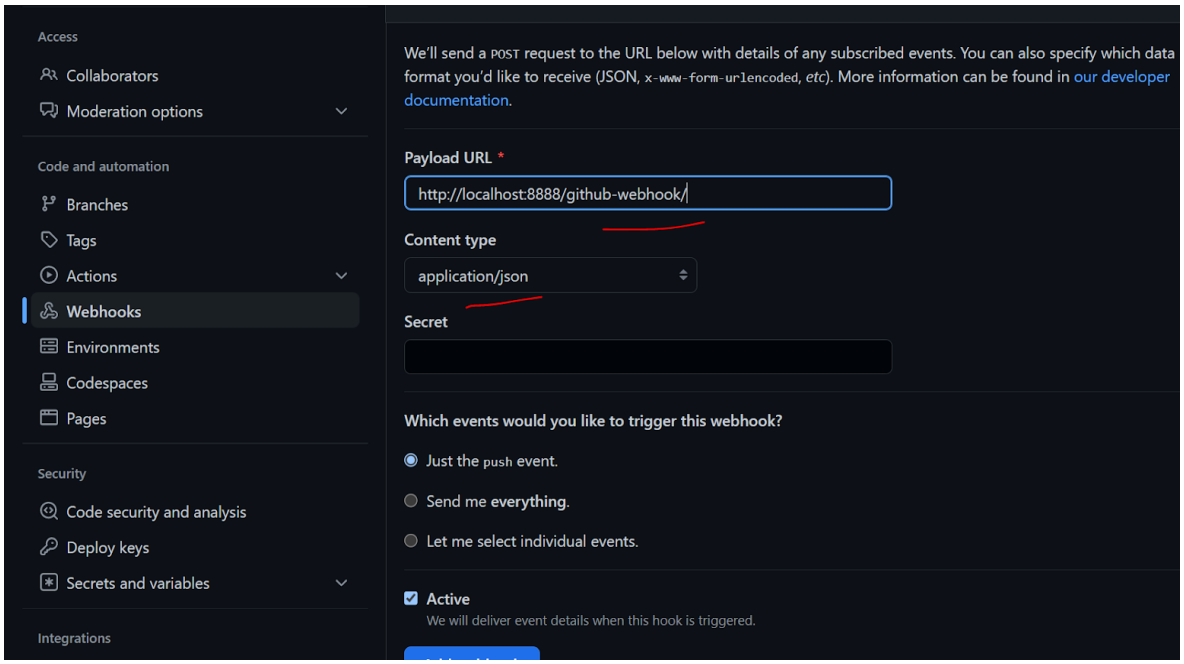


7. Add a webhook which will trigger the build after each commit.
   Enter the repository name to add the webhook on.

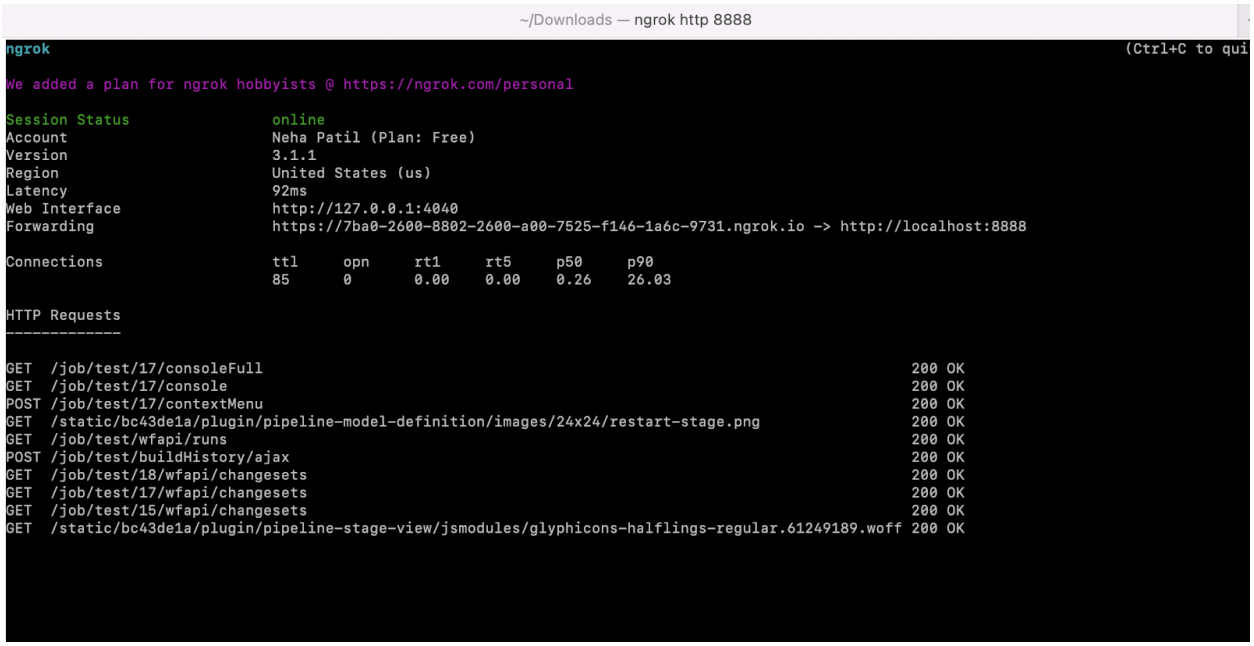




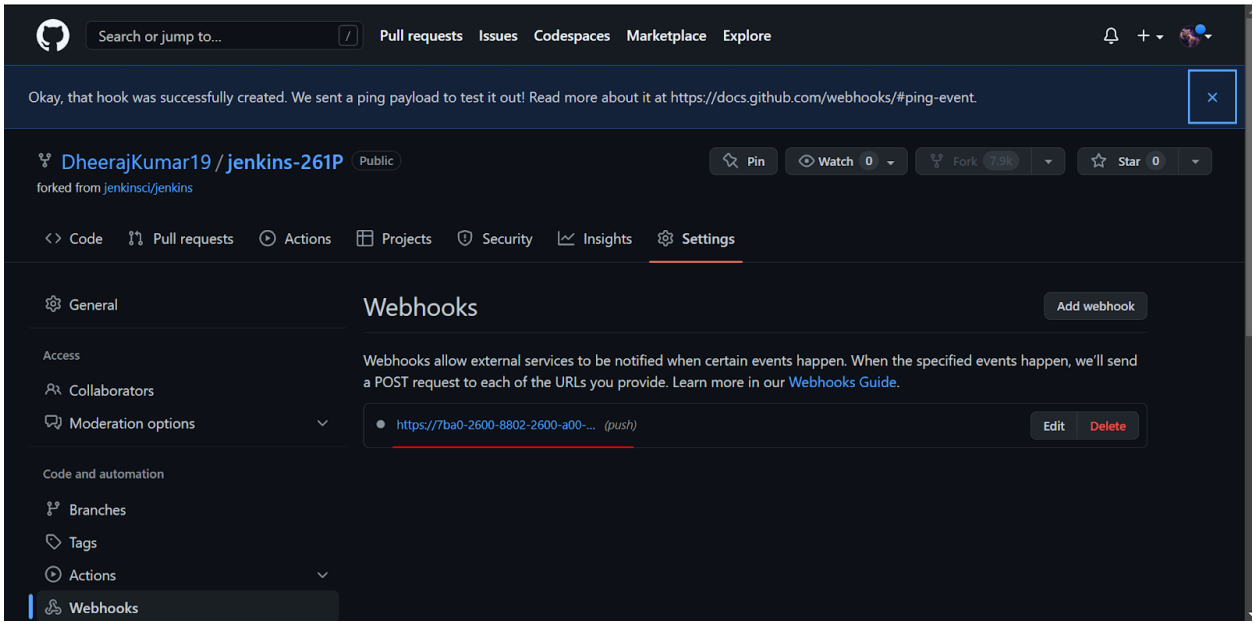
8. Enter the payload URL.



As we have setup jenkins locally, we entered the localhost URL but this gave an error as Github expects the URL to be hosted on public domain. To overcome this, we installed ngrok, which forwarded the private localhost address to a public domain.

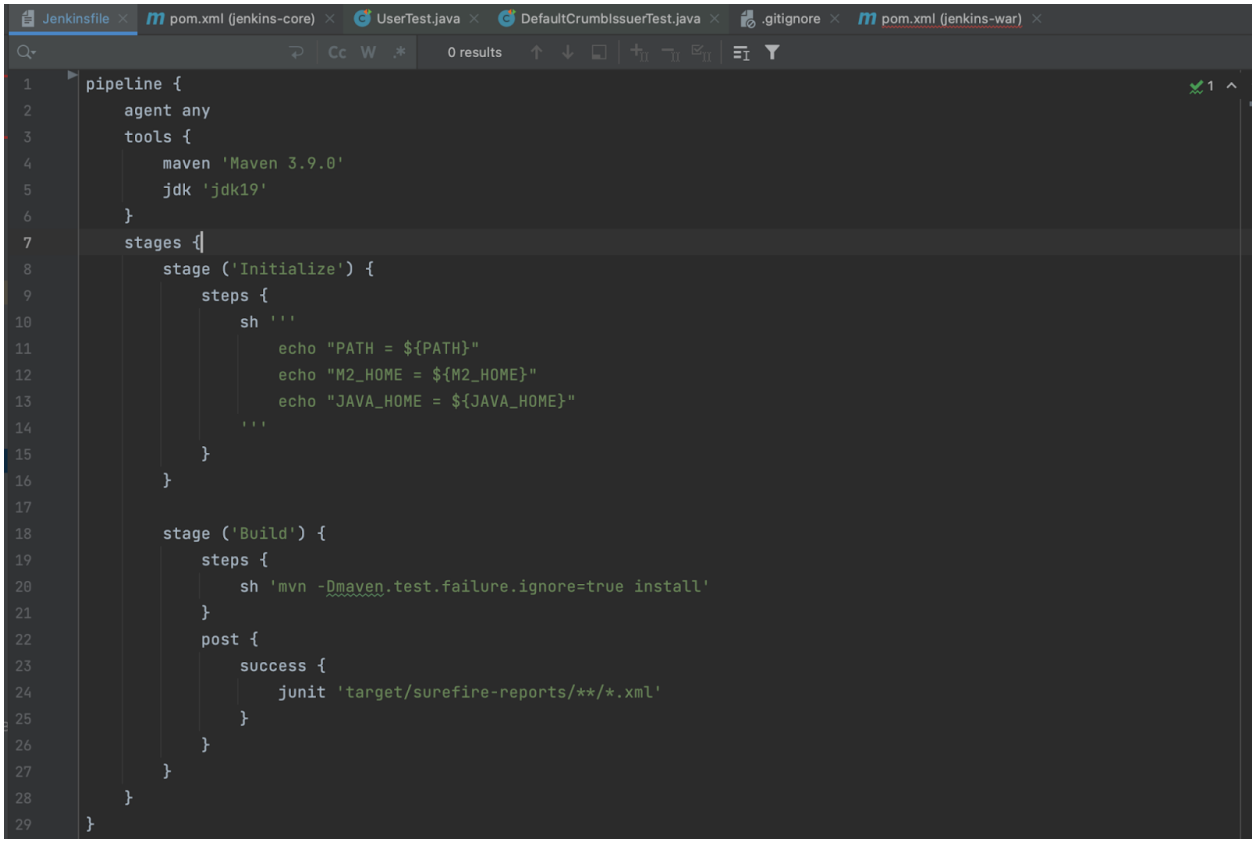


Ngrok gives us the public IP which can be added as a payload URL for the webhook



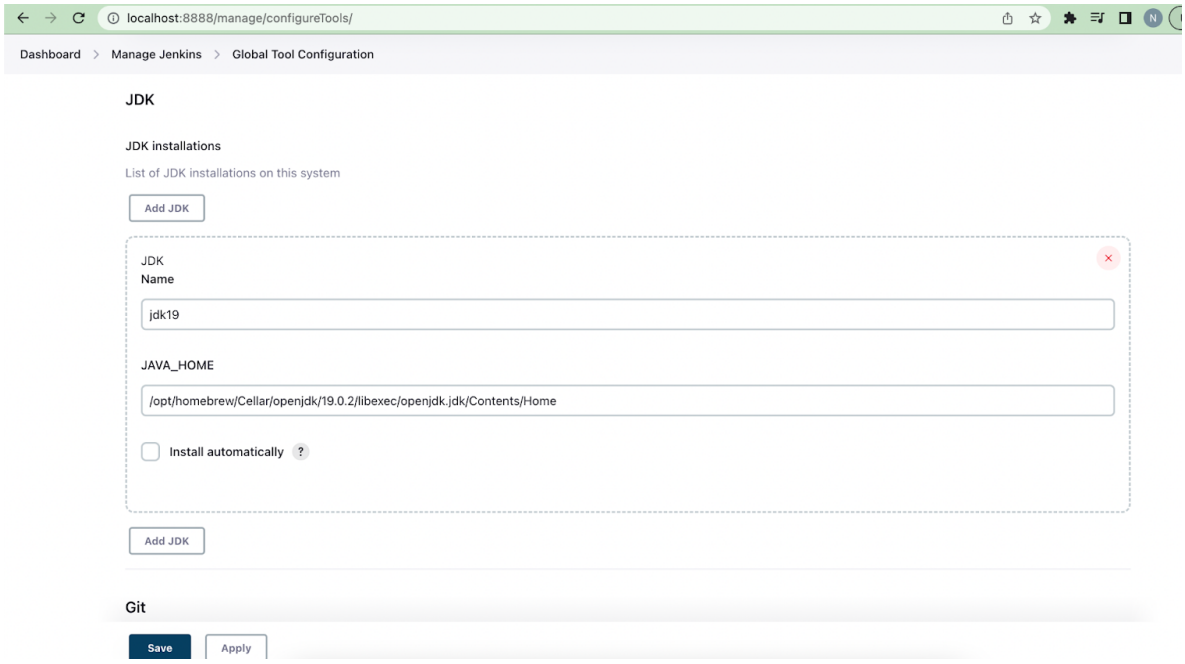
Create a configuration file (e.g., .travis.yml, or similar for the CI system that you chose) to build and test your project.

1. Add a Jenkinsfile to your project.
   Commit
   https://github.com/DheerajKumar19/jenkins-261P/commit/9853e605caef3ed065f1d79095e70b2481410c38

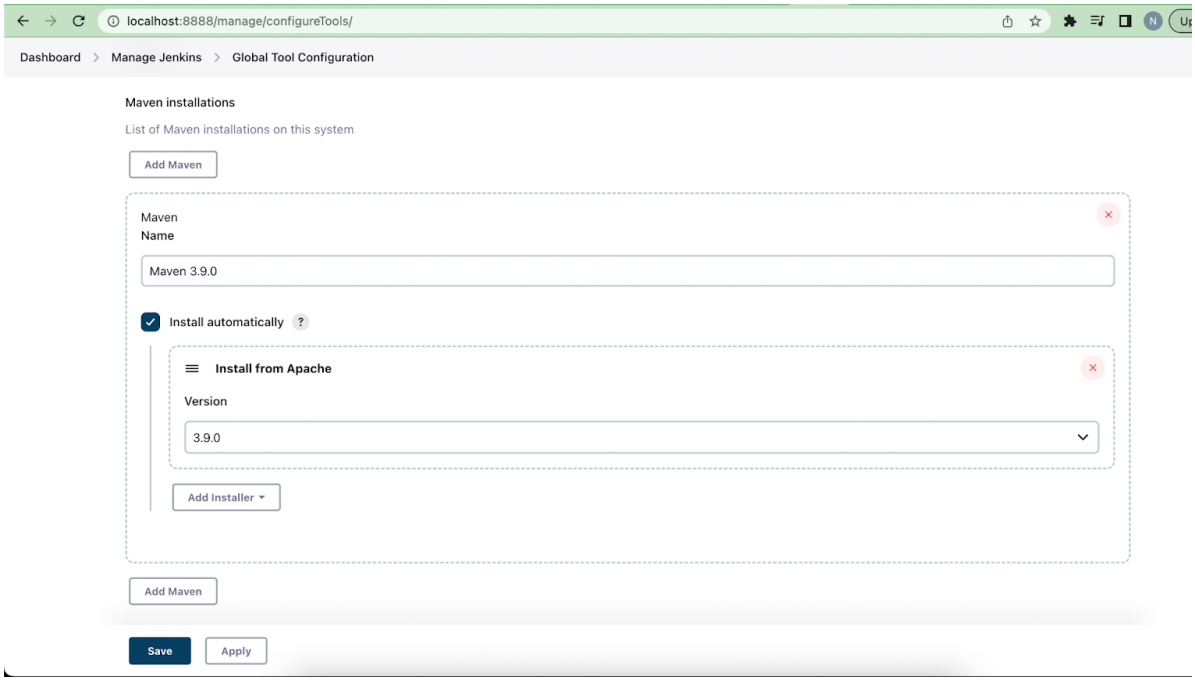


2. Add the required tools to your global tools configuration in your local jenkins setup. In our case, we need to add jdk and maven to our tools configuration.

JDK:



MAVEN:



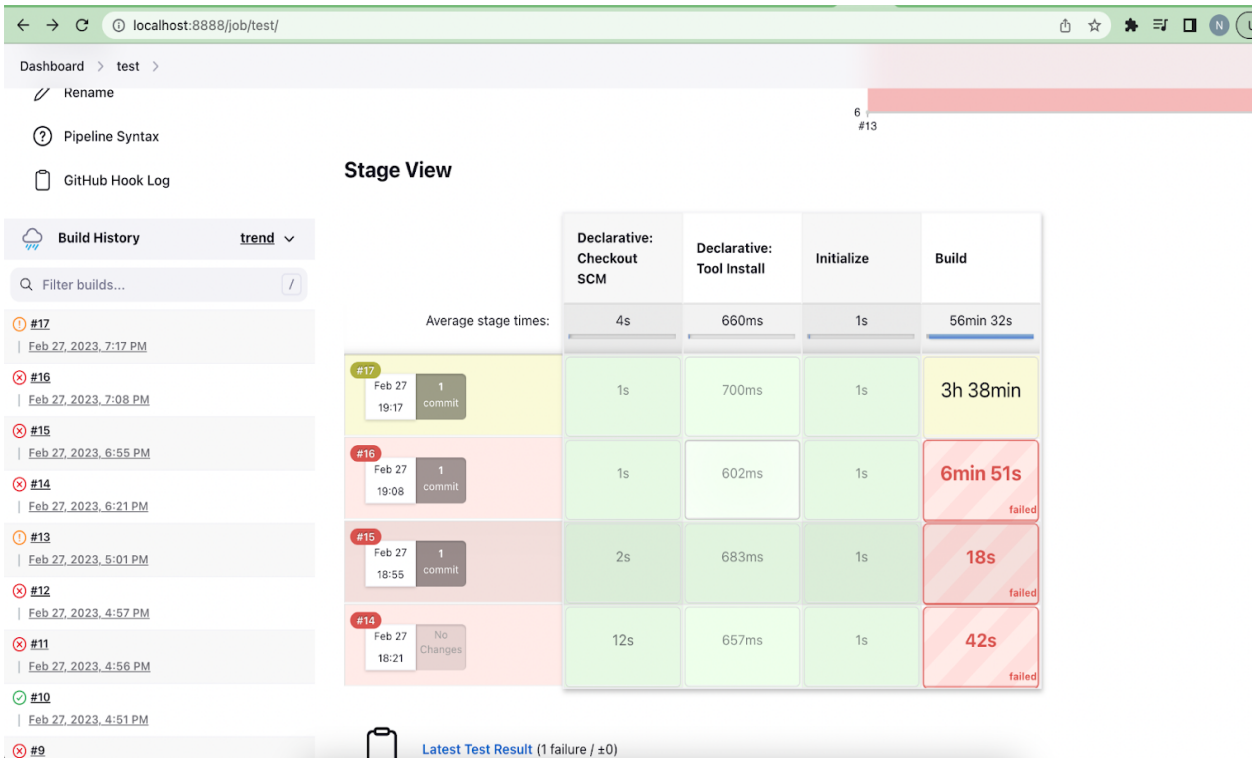
3. The build stage for maven is actually the **build + test** stage combined

```
mvn -Dmaven.test.failure.ignore=true install
```

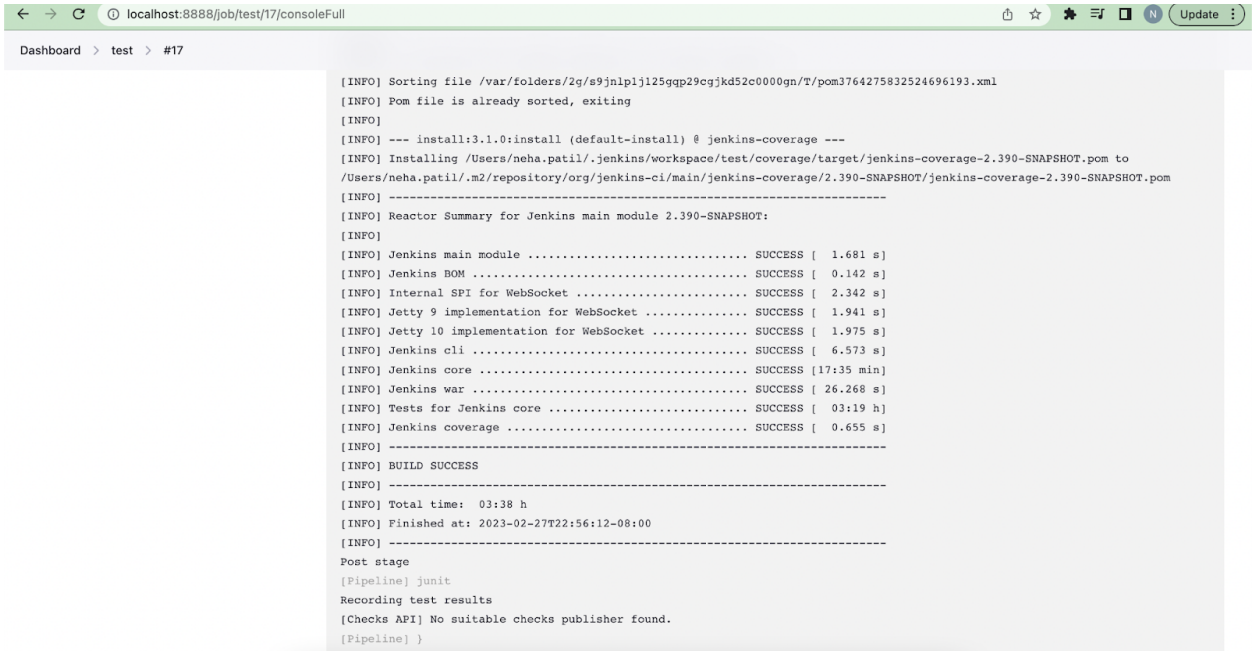
Commit the changes and verify that it builds your system and runs the test cases. Document with screenshots and description of how it builds and any issues that you run into.

1. After committing the Jenkinsfile changes, the build gets auto triggered. We ran into a couple of issues initially. The following errors were encountered:
    a. [ERROR] Rule 4: org.apache.maven.plugins.enforcer.RequireUpperBoundDeps failed with message: Failed while enforcing RequireUpperBoundDeps.
    Fix:https://github.com/DheerajKumar19/jenkins-261P/commit/68d2f066708a87fc70cad5e7cf394c47a4cceda8
    b. [ERROR] Failed to execute goal com.diffplug.spotless:spotless-maven-plugin:2.31.0:check (default) on project jenkins-core:
    Fix:https://github.com/DheerajKumar19/jenkins-261P/commit/37d03783dc9f39adf1a12e4486c8c3c1d68c6912

    After fixing all the issues, the build was successful:



2. Successfully built the jenkins project.



13. Part 12: Testable Design

Why is testability in my design important?
It makes sense to ask the question. When creating software, one develops the ability to consider the goals of the system as well as the outcomes for the user.
Nevertheless, testing against our program represents a different kind of user. That user has exacting requirements for our software, but they are all logically derived from testability. Such requests may have numerous effects on the design of our software, most of which are positive.

Each logical piece of code (loops, ifs, switches, etc.) in a testable design needs to be simple and quick to create a unit test against, one that exhibits certain properties:

1. Runs fast.
2. Independent tests may be performed before or after any other test, separately or as a part of a collection of tests.
3. No extra configuration is necessary.
4. Gives a reliable pass/fail outcome.

12.1 Design goals for testability

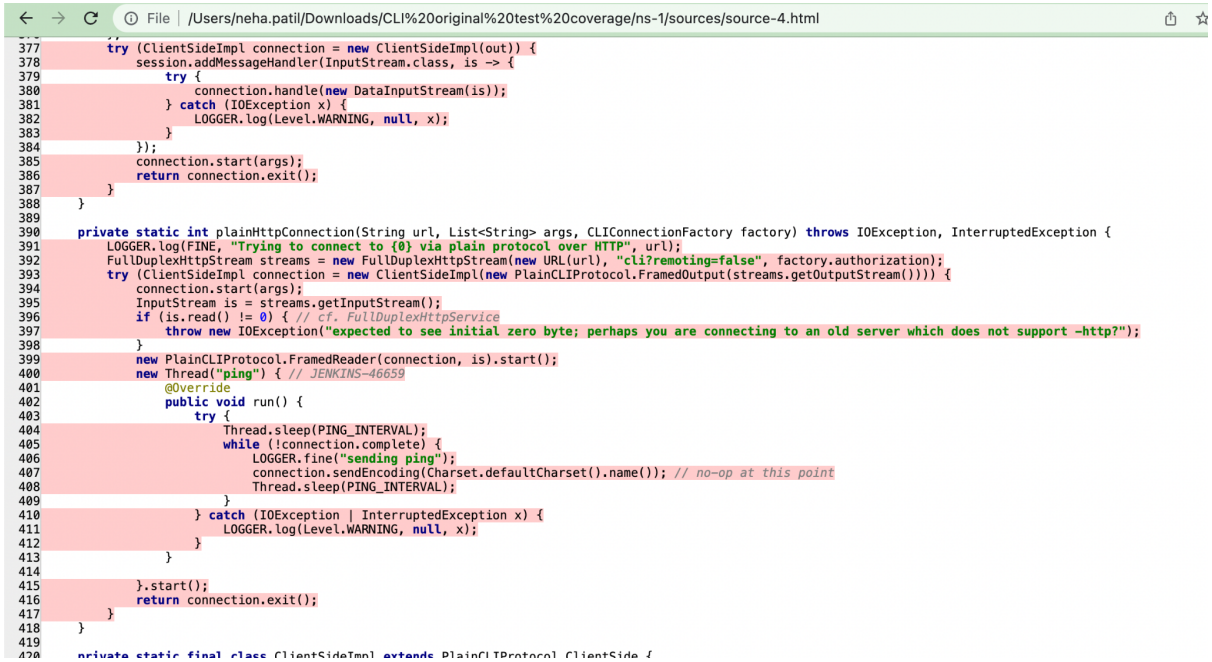
Code is substantially more testable as a result of various design considerations.The majority of the tips offered here concern allowing the code to have seams, or locations that allow for substitute functionality without altering the original class.
(Martin's "Principles of OOD[OOD]" mentions the Open-Closed Principle, which is frequently discussed in relation to seams.)

Design principles	Benefit(s)
Make all methods virtual by default.	This enables users to test by overriding the methods of a derived class. Through overriding, it is possible to modify behavior or stop calling an external dependency
Design with interfaces in mind.	This provides a chance to use polymorphism to replace system dependencies by creating our own stubs or mocks.
Make classes non sealed by default.	If the class is sealed, nothing virtual can be overridden (final in Java).
Do not instantiate concrete classes inside	This enables users from being forced to use a

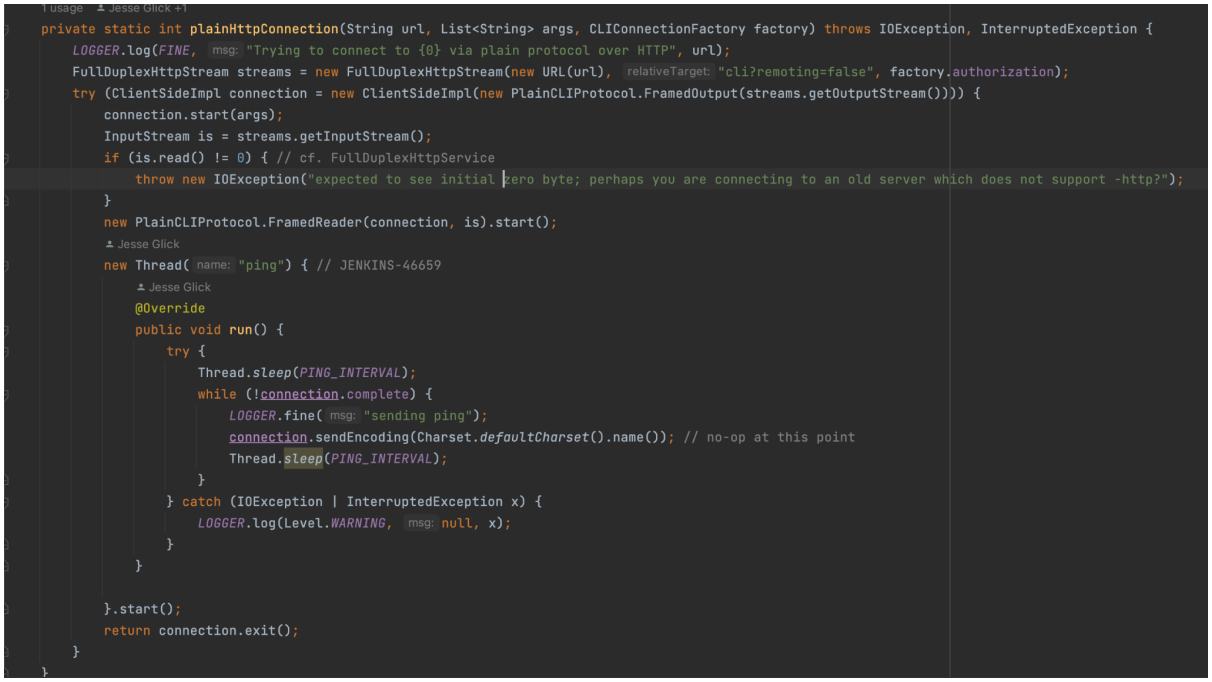
logical methods. Don't directly construct instances of classes; instead, obtain them through factories, helper methods, inversion of control frameworks like Unity, or other sources.	class's internal production instance and lets us offer up their own fictitious instances of classes to methods that demand them.
Don't invoke static methods directly. Favour instance methods over static methods that call statics afterwards.	By overriding instance methods, one can prevent calls to static methods. (Static methods cannot be overridden.)
Distinguish singleton holders from singleton logic.	Provide a way to replace a singleton's instance with a new one in order to inject a stub singleton or reset it.

Find an example in the code of an implementation that would make a certain kind of testing difficult or impossible. Document that code in your report and describe what would be prevented with that code. Describe how you would advise to fix that code. Implement a new version of that code with the newer design. (This can be as a dummy method that does not break the existing code).

As per the coverage report generated in this previous assignment, the following code is not covered by any test cases:




```
377         try (ClientSideImpl connection = new ClientSideImpl(out)) {
378             session.addMessageHandler(InputStream.class, is -> {
379                 try {
380                     connection.handle(new DataInputStream(is));
381                 } catch (IOException x) {
382                     LOGGER.log(Level.WARNING, null, x);
383                 }
384             });
385             connection.start(args);
386             return connection.exit();
387         }
388     }
389
390     private static int plainHttpConnection(String url, List<String> args, CLIConnectionFactory factory) throws IOException, InterruptedException {
391         LOGGER.log(FINE, "Trying to connect to {0} via plain protocol over HTTP", url);
392         FullDuplexHttpStream streams = new FullDuplexHttpStream(new URL(url), "cli?remoting=false", factory.authorization);
393         try (ClientSideImpl connection = new ClientSideImpl(new PlainCLIProtocol.FramedOutput(streams.getOutputStream()))) {
394             connection.start(args);
395             InputStream is = streams.getInputStream();
396             if (is.read() != 0) { // cf. FullDuplexHttpService
397                 throw new IOException("expected to see initial zero byte; perhaps you are connecting to an old server which does not support -http?");
398             }
399             new PlainCLIProtocol.FramedReader(connection, is).start();
400             new Thread("ping") { // JENKINS-46659
401                 @Override
402                 public void run() {
403                     try {
404                         Thread.sleep(PING_INTERVAL);
405                         while (!connection.complete) {
406                             LOGGER.fine("sending ping");
407                             connection.sendEncoding(Charset.defaultCharset().name()); // no-op at this point
408                             Thread.sleep(PING_INTERVAL);
409                         }
410                     } catch (IOException | InterruptedException x) {
411                         LOGGER.log(Level.WARNING, null, x);
412                     }
413                 }
414
415             }.start();
416             return connection.exit();
417         }
418     }
419
420     private static final class ClientSideImpl extends PlainCLIProtocol.ClientSide {
```



```
1 usage   Jesse Glick +1
private static int plainHttpConnection(String url, List<String> args, CLIConnectionFactory factory) throws IOException, InterruptedException {
    LOGGER.log(FINE, msg: "Trying to connect to {0} via plain protocol over HTTP", url);
    FullDuplexHttpStream streams = new FullDuplexHttpStream(new URL(url), relativeTarget: "cli?remoting=false", factory.authorization);
    try (ClientSideImpl connection = new ClientSideImpl(new PlainCLIProtocol.FramedOutput(streams.getOutputStream()))) {
        connection.start(args);
        InputStream is = streams.getInputStream();
        if (is.read() != 0) { // cf. FullDuplexHttpService
            throw new IOException("expected to see initial zero byte; perhaps you are connecting to an old server which does not support -http?");
        }
        new PlainCLIProtocol.FramedReader(connection, is).start();
        Jesse Glick
        new Thread( name: "ping") { // JENKINS-46659
            Jesse Glick
            @Override
            public void run() {
                try {
                    Thread.sleep(PING_INTERVAL);
                    while (!connection.complete) {
                        LOGGER.fine( msg: "sending ping");
                        connection.sendEncoding(Charset.defaultCharset().name()); // no-op at this point
                        Thread.sleep(PING_INTERVAL);
                    }
                } catch (IOException | InterruptedException x) {
                    LOGGER.log(Level.WARNING, msg: null, x);
                }
            }

        }.start();
        return connection.exit();
    }
}
```

This method has a lot of internal dependencies and side effects, such as:

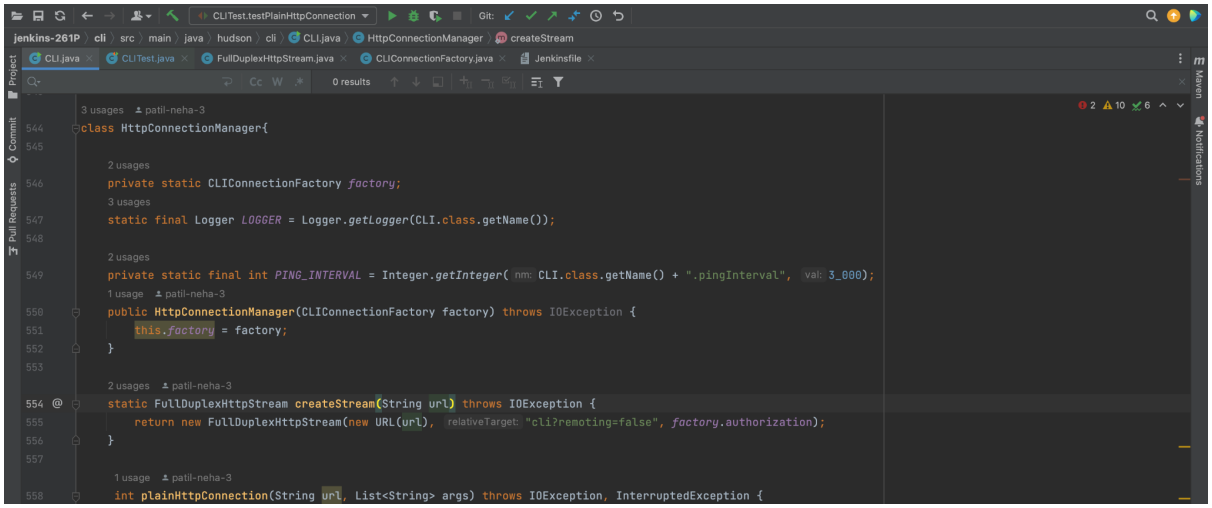
- creating and starting a thread
- making network connections
- reading from input and output streams
- tight coupling because of the “new” keyword

This makes it difficult to isolate and test individual parts of the code, and to write test cases that accurately reflect the behavior of the method in different scenarios. Test cases would have the additional overload of instantiating the objects of ‘FullDuplexHttpStream’ and ‘ClientSideImpl’.

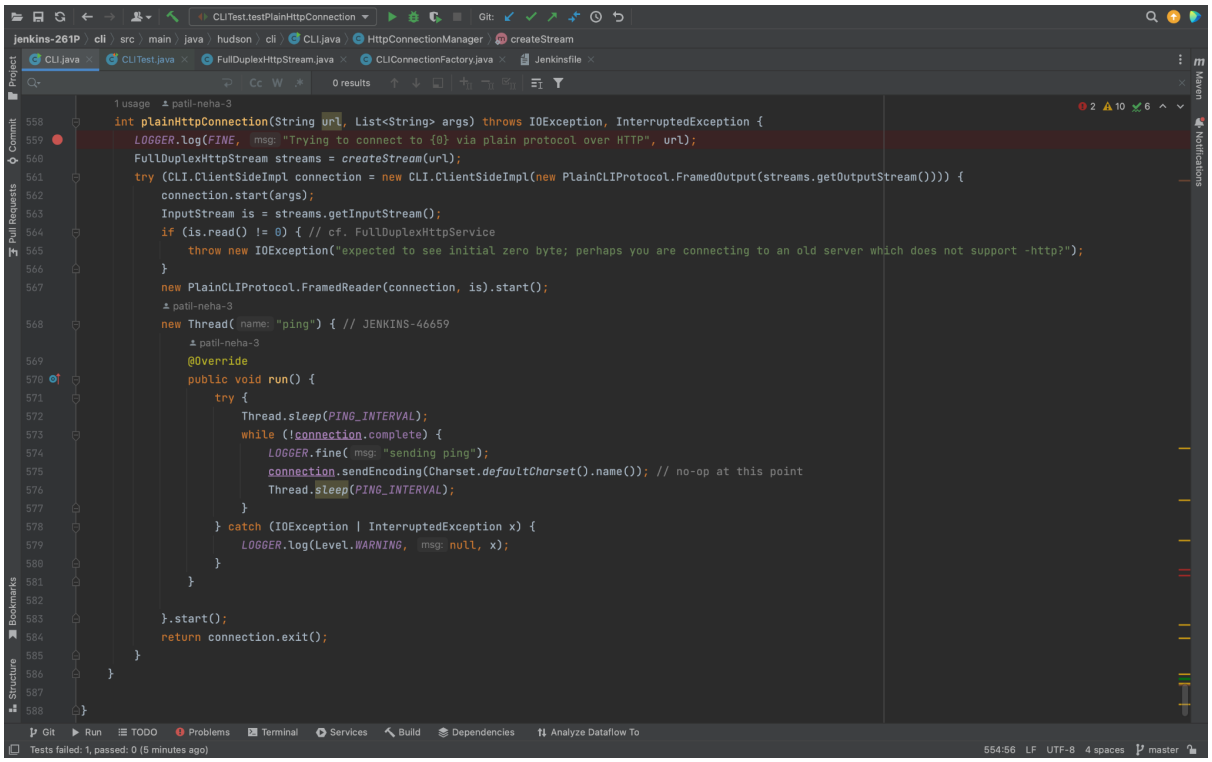
To make the code more testable, we can do the following:

- Extract the code that performs the network connection and data transfer into a separate class or method that can be easily mocked or stubbed for testing purposes.
- Pass in any necessary dependencies as parameters rather than creating them internally, so they can be more easily mocked or stubbed.
- Refactor the method to reduce the number of side effects and make it more focused on a single responsibility, such as making a network connection.

Newer Design:
Commit: Commit_Link



```
class HttpConnectionManager{

    private static CLIConnectionFactory factory;
    static final Logger LOGGER = Logger.getLogger(CLI.class.getName());

    private static final int PING_INTERVAL = Integer.getInteger( nm: CLI.class.getName() + ".pingInterval",  val: 3_000);
    public HttpConnectionManager(CLIConnectionFactory factory) throws IOException {
        this.factory = factory;
    }

    static FullDuplexHttpStream createStream(String url) throws IOException {
        return new FullDuplexHttpStream(new URL(url),  relativeTarget: "cli?remoting=false", factory.authorization);
    }

    int plainHttpConnection(String url, List<String> args) throws IOException, InterruptedException {
```



```
    int plainHttpConnection(String url, List<String> args) throws IOException, InterruptedException {
        LOGGER.log(FINE,  msg: "Trying to connect to {0} via plain protocol over HTTP", url);
        FullDuplexHttpStream streams = createStream(url);
        try (CLI.ClientSideImpl connection = new CLI.ClientSideImpl(new PlainCLIProtocol.FramedOutput(streams.getOutputStream()))) {
            connection.start(args);
            InputStream is = streams.getInputStream();
            if (is.read() != 0) { // cf. FullDuplexHttpService
                throw new IOException("expected to see initial zero byte; perhaps you are connecting to an old server which does not support -http?");
            }
            new PlainCLIProtocol.FramedReader(connection, is).start();
            new Thread( name: "ping") { // JENKINS-46659
                @Override
                public void run() {
                    try {
                        Thread.sleep(PING_INTERVAL);
                        while (!connection.complete) {
                            LOGGER.fine( msg: "sending ping");
                            connection.sendEncoding(Charset.defaultCharset().name()); // no-op at this point
                            Thread.sleep(PING_INTERVAL);
                        }
                    } catch (IOException | InterruptedException x) {
                        LOGGER.log(Level.WARNING,  msg: null, x);
                    }
                }

            }.start();
            return connection.exit();
        }
    }
}
```

- Create a class called HttpConnectionManager that injects an object of CLIConnectionFactory.
- Extract the code creating a FullHttpDuplexStream into a separate method which can be mocked.
- We have altered the tight dependency in the code by restructuring the design of the code.

Test Case:

```
public class CLITest {

    patil-neha-3 *
    @Test
    public void testPlainHttpConnection() throws IOException, InterruptedException {
        // Create mock dependencies
        CLIConnectionFactory mockFactory = mock(CLIConnectionFactory.class);
        CLI.ClientSideImpl mockConnection = mock(CLI.ClientSideImpl.class);
        FullDuplexHttpStream mockStreams = mock(FullDuplexHttpStream.class);
        InputStream mockInputStream = mock(InputStream.class);

        // Configure mock objects
        mockFactory.authorization= "nppatil";
        when(mockStreams.getOutputStream()).thenReturn(mock(OutputStream.class));
        when(mockStreams.getInputStream()).thenReturn(mockInputStream);
        when(mockConnection.exit()).thenReturn( t: 0);
        when(mockInputStream.read()).thenReturn( t: 0);

        // Create test object
        HttpConnectionManager connectionManager = new HttpConnectionManager(mockFactory);
        when(HttpConnectionManager.createStream( url: "https://jenkins.io/")).thenReturn(mockStreams);
        // Call method under test
        int result = connectionManager.plainHttpConnection( url: "https://jenkins.io/", Arrays.asList("help"));

//        // Verify results
        assertEquals( expected: 0, result);
        verify(mockStreams).getOutputStream();
        verify(mockStreams).getInputStream();
        verify(mockConnection).start(Arrays.asList("help"));
        verify(mockConnection).sendEncoding(anyString());
        verify(mockConnection).exit();
        verify(mockInputStream).read();
    }
}
```

In this test case, we're using the Mockito library to create mock objects for the CLIConnectionFactory, ClientSideImpl, FullDuplexHttpStream, and InputStream classes. We're then configuring these mock objects to return the expected data or behavior when they're called. We create an instance of the HttpConnectionManager class, and call its plainHttpConnection method with a mocked URL and list of arguments. We then verify that the method returns the expected result, and that the mock objects were called with the expected parameters and at the expected times.

This test case verifies that the plainHttpConnection method is working correctly, and that it's properly interacting with its dependencies.

It's also much more testable than the original implementation, since we're able to isolate and test individual parts of the code without needing to perform complex setup or teardown operations.

12.4 Mocking and its Utilities

Mocking is used in software testing and development to simulate the behavior of real objects to the test in the system by creating objects. It is basically used to isolate a part of the system from the system which contains its dependencies so that the focus is only on the object or the component that is being tested while being unaffected by the other dependencies and other components. Different programming languages provide different mockito frameworks such as Java uses Mockito, .NET uses NUnit.Mocks and unittest.mock is used by Python. With the help of mocking we can achieve speed by which we need not depend or spend time on other classes and their methods and just test the functionality of the particular object or module to be tested. This helps in solving bugs and fixing the issues quickly. It also helps in achieving modularity in the code and making the tests more reliable and efficient. This is also very useful in full stack applications, when the frontend is dependent on the backend component of the software. So in case when the frontend part is not completed, backend functionalities can still be tested using Mocks, this saves a lot of time of the testing.

Utilities of Mocking:

1. **Isolation**: Mocking helps to isolate the module being tested from its external dependencies, which makes it easier to find bugs and solves the issues while not being affected by the external dependencies and prevents regression. This makes it possible to reduce the false positives because of external dependencies or components. This makes it possible to really focus on the module efficiently.
2. **Speed**: It helps to achieve speed for testing. For example, if its a full stack application, the backend functionalities can be tested without waiting for the frontend component to be completed and vice versa. Even the application that requires the population of the database system, can be testing fast using mocks by not actually populating the database.
3. **Flexibility**: By using Mocks, we can mock the real world objects and test the edge cases which would be difficult using normal tests. This provides flexibility for tests with more range of values and test coverage can also be improved. Flexibility is there just to change the parts that we can control or change.
4. **Debugging**: Using mocks, debugging gets improved as mocking frameworks usually provide logs with detailed information about where the issue or bug has occurred which helps to get to the root cause easily.

12.5 Feature that could be mocked using Mockito

Here we are mocking a function **createValue()** which can be found inside the class **PasswordParameterDefinition.** Below is the code coverage that we generated from the previous assignment.

Coverage Summary for Class: PasswordParameterDefinition (hudson.model)

Class	Method, %	Branch, %	Line, %
PasswordParameterDefinition	54.5% (6/11)	11.1% (2/18)	34.3% (12/35)

Here we use Mockito to mock the **StaplerRequest** and **JSONObject** objects in this code snippet, and then test the behavior of the **createValue** method.

```
public PasswordParameterValue createValue(StaplerRequest req, JSONObject jo) {
PasswordParameterValue value = req.bindJSON(PasswordParameterValue.class, jo);
if (value.getValue().getPlainText().equals(DEFAULT_VALUE)) {
value = new PasswordParameterValue(getName(), getDefaultValue());
}
value.setDescription(getDescription());
return value;
}
```

We are writing a test method that configures the mock **StaplerRequest** and **JSONObject** objects so that, when their methods are called, they return the expected values, and then it checks to see that the mock objects' methods actually called the expected methods. The **createValue** method's ability to return the anticipated value is then tested.

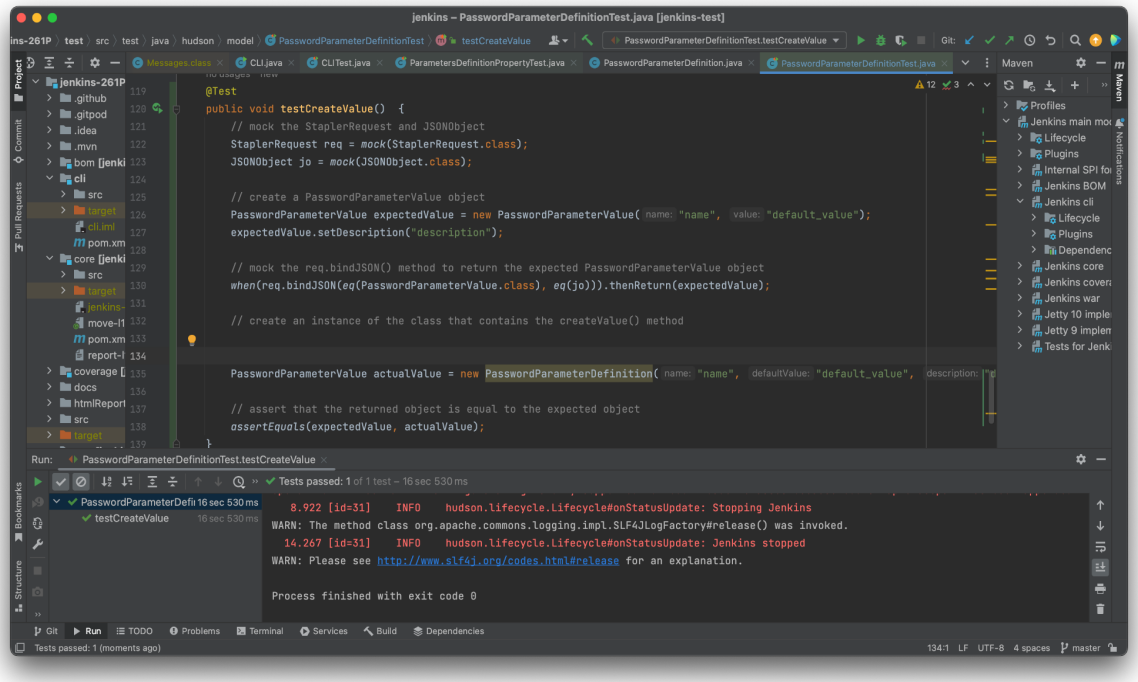
Why do we prefer to do a mock on this function even when we can test it by writing a Junit test?

In developing a JUnit test for the above code, you can run into the following restrictions:

1. **Dependency Injection**: **StaplerRequest** and **JSONObject** objects are prerequisites for the **createValue** function. These items must be injected during the test.
2. **Default Value**: If the password value matches the default value, the **createValue** function produces a new **PasswordParameterValue** object with that value. We must ensure that the test includes both scenarios (when the password value is the default value and when it is not).
3. **Parameter Validation**: The **PasswordParameterValue** object given into the **createValue** method is assumed to have a non-null value. We must ensure that this scenario is covered by the test.
4. **Object Mutability**: The **value** object is altered by the **createValue** method. We must ensure that the test validates the proper modification of the object.
5. **Side Effects**: The **value** object is altered as a byproduct of the **createValue** method.We must make sure that the test confirms that this side effect is applied correctly.
6. **Exception Handling**: The **createValue** method could raise exceptions (for example, if the JSON object is not well-formed). We must ensure that the test includes these scenarios.

How may mocking be helpful in this situation?
By enabling us to isolate the method and test it independently from external systems, mocking can assist in the testing of the **createValue** function.
To test how the function responds to different cases, such as a valid or incorrect input, we may, for instance, mimic the **StaplerRequest** and **JSONObject** objects and specify their behavior.
Mocking can also be used to check that specific actions or behaviors take place when a method is being executed, such as checking that the getDescription method is invoked and that the returned value is utilized to change the description of the value object.

12.6 Mock test for createValue function:



The following mock test is added to the repo via this commit.

14. Part 13: Static Analyzers

13.1 First, describe the goals, purposes, and use of static analysis tools (i.e., static analyzers).

Software testing goals: The primary goals of software testing is to improve software quality, reduce maintenance costs and to increase productivity. To reduce the risk of security vulnerabilities and software bugs, static analysis tools are used to search defects in code before it is executed. The source code flaws can easily be identified and to make sure that the software is up to the mark of the specified standards prior to release. They provide automated solutions which are useful for detecting flaws and for monitoring code quality in the development lifecycle.

Purpose: The purpose of the static analysis tool is to improve the performance issues, security vulnerabilities and incorrect behavior by recognizing defects in software code. The purpose is mainly to make sure that the software uses best practices and coding standards. The static analyzer tool can identify a lot of defects including null pointer dereferences, security vulnerabilities, race conditions, buffer overflows and memory leaks.
Uses: Static analysis tools are used for code security and for improving code quality. They are used in code reviews and during the build process to automate the build process.

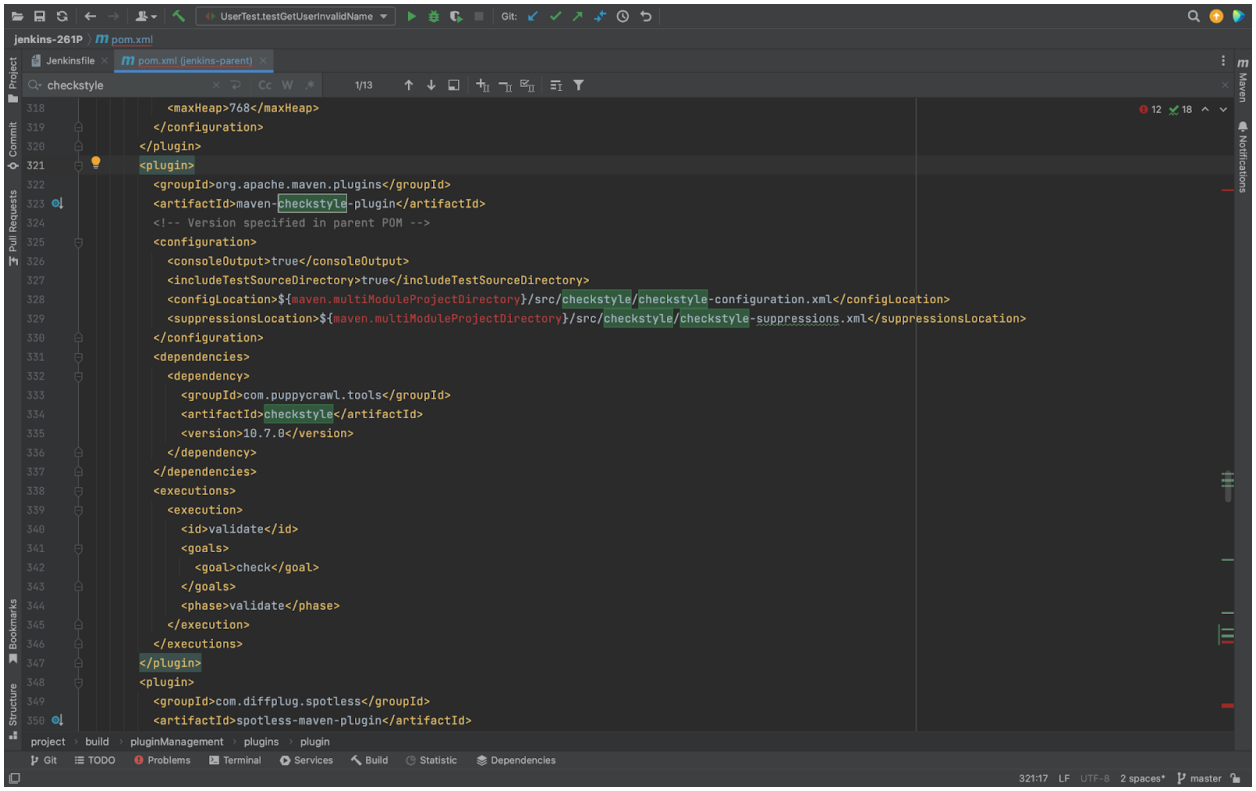
1. Maintain code uniformity: When several developers write the software code it becomes very essential to maintain the uniformity in the code which includes the way the function names, class names and variables are written, including the location of the declaration, underscore, capital letters, etc. This makes the code readable so that even in the future the developers can read the code easily.
2. Code Documentation: Documentation is very important for the development and to read the code for developers. Static Code analysis tools enables the documentation for classes, methods, so that it is easier to understand.
3. Security Issues: Static analysis can identify the security issues and sanitation of the code. It helps prevent security attacks.
4. Client side analysis: Static code analysis can also be used for various javascript frameworks like JSHint and EESLint.

13.2 Use two different static analyzers on your project (on the whole or on the same subset of the code for each tool).

We have chosen Checkstyle and SpotBugs as the static analyzers for Jenkins project\\

1. Checkstyle:

Checkstyle is already integrated with the Jenkins project as a maven plugin

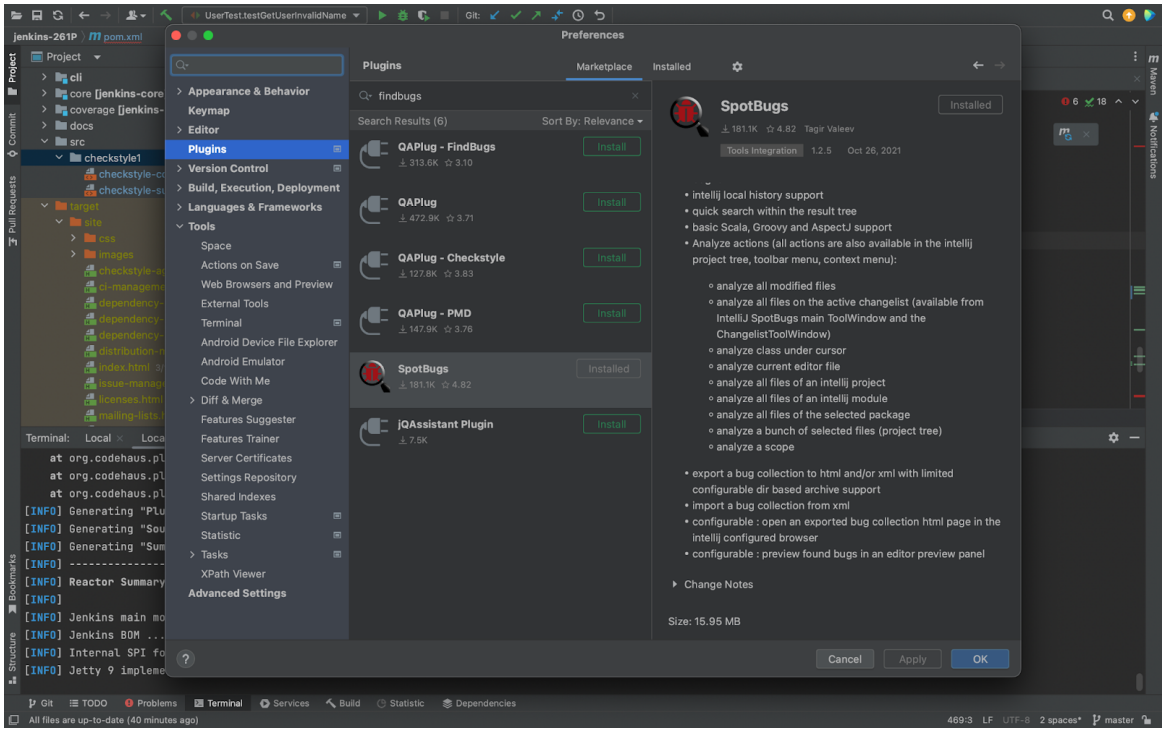


This plugin is added in the build section of the pom.xml file. This causes the style check to happen at the build time and if the code does not conform to the the styling standards mentioned in the checkstyle configuration file, the build will fail. Jenkins follows a custom styling configuration specified in the checkstyle-configuration.xml file that is present in the src/checkstyle folder.

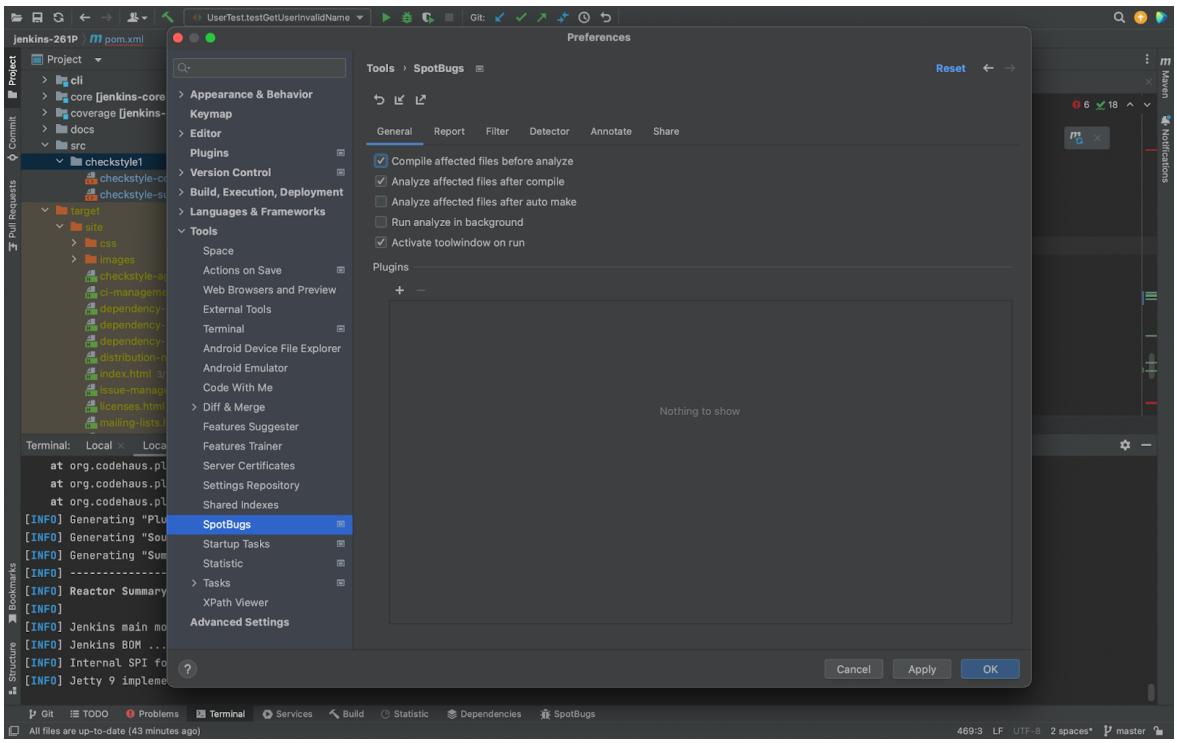
For the sake of this assignment, we need to generate a checkstyle report. Thus we need to add this plugin as part of the reporting section in the pom.xml file. We are also changing the configuration to use a styling standard such as a Sun-style check as opposed to the custom style. When no configuration file is specified, the default config file is sun_checks.xml. To generate the static analysis report, run the following "mvn site". The report file will be generated and stored in the target/site folder. The name of the report will be checkstyle-aggregate.html. This can be opened in the web browser for further analysis.

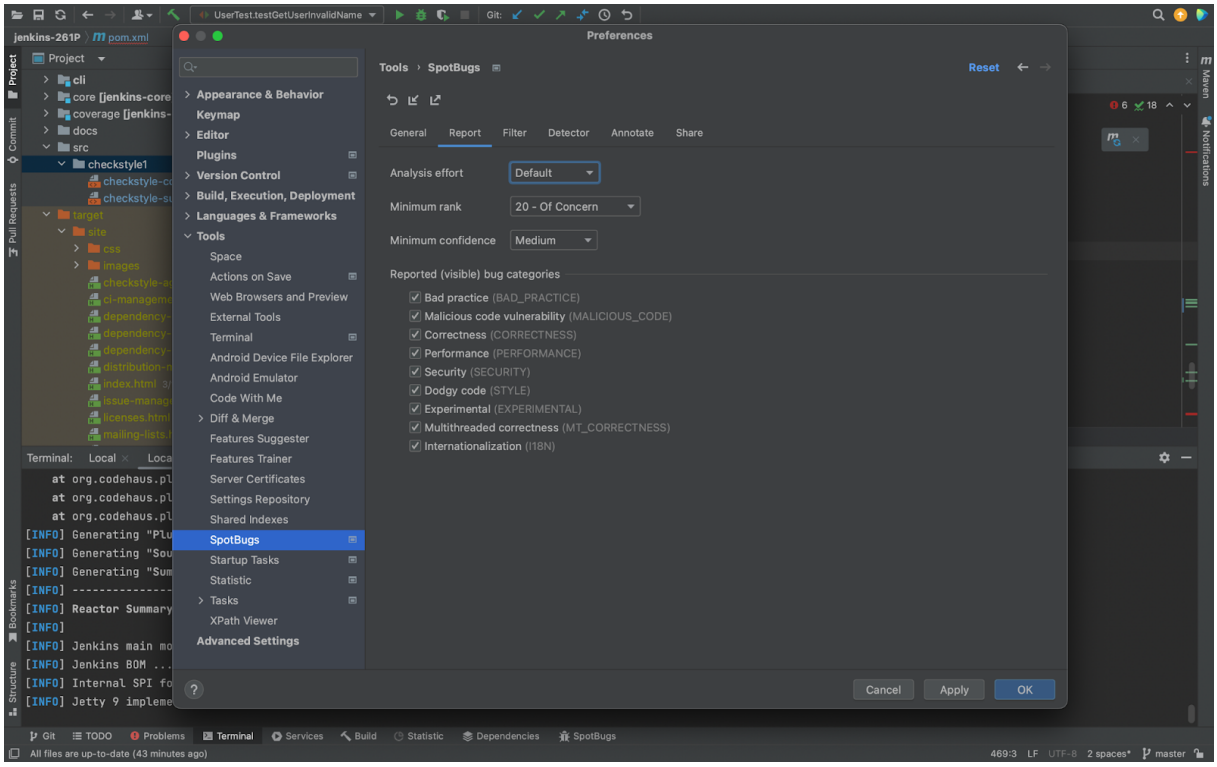
2. Spotbugs

Install Spotbugs plugin from the IntelliJ plugin marketplace.



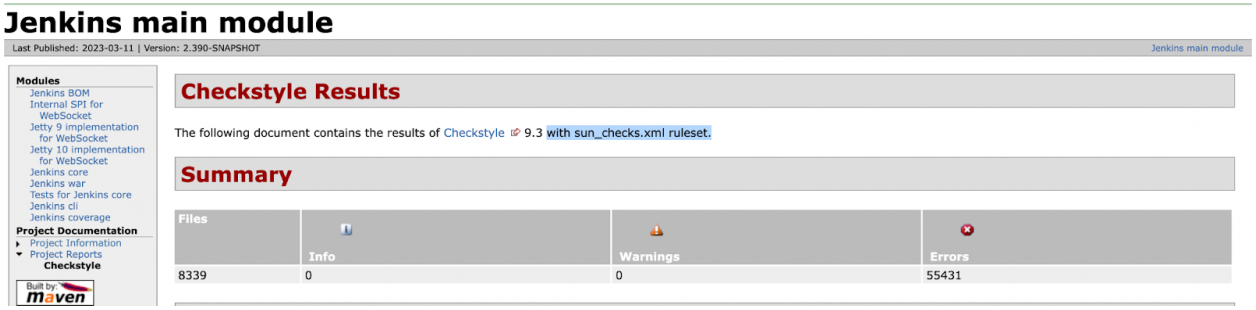
Navigate to SpotBugs in the settings pane in IntelliJ and customize the static analysis tool.





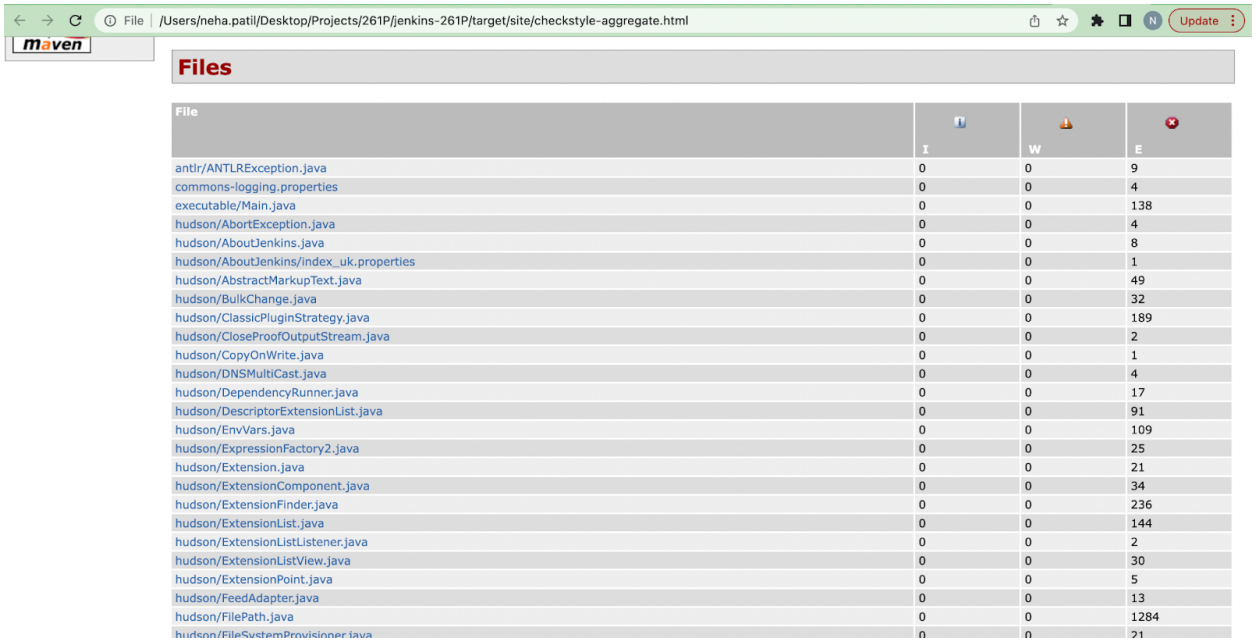
13.3 Show the aggregated numbers for the results. First separate out the results for each tool.

Checkstyle: For checkstyle, we are opening the results in the web browser. Checkstyle has detected 55431 errors in the entire project based on sun_checks.xml ruleset



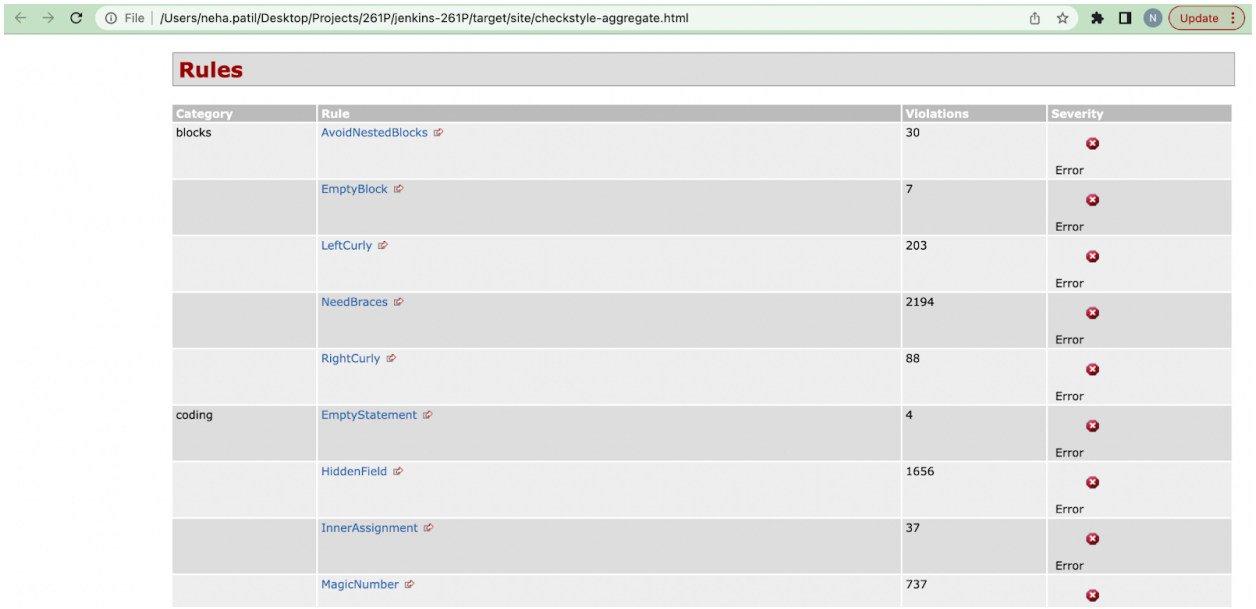
The checkstyle report is mainly divided into 3 sections:

Files: The list of files in which the violations have occurred is provided to us in this part of the report. Additionally, it displays the numbers of violations and their severity degrees.



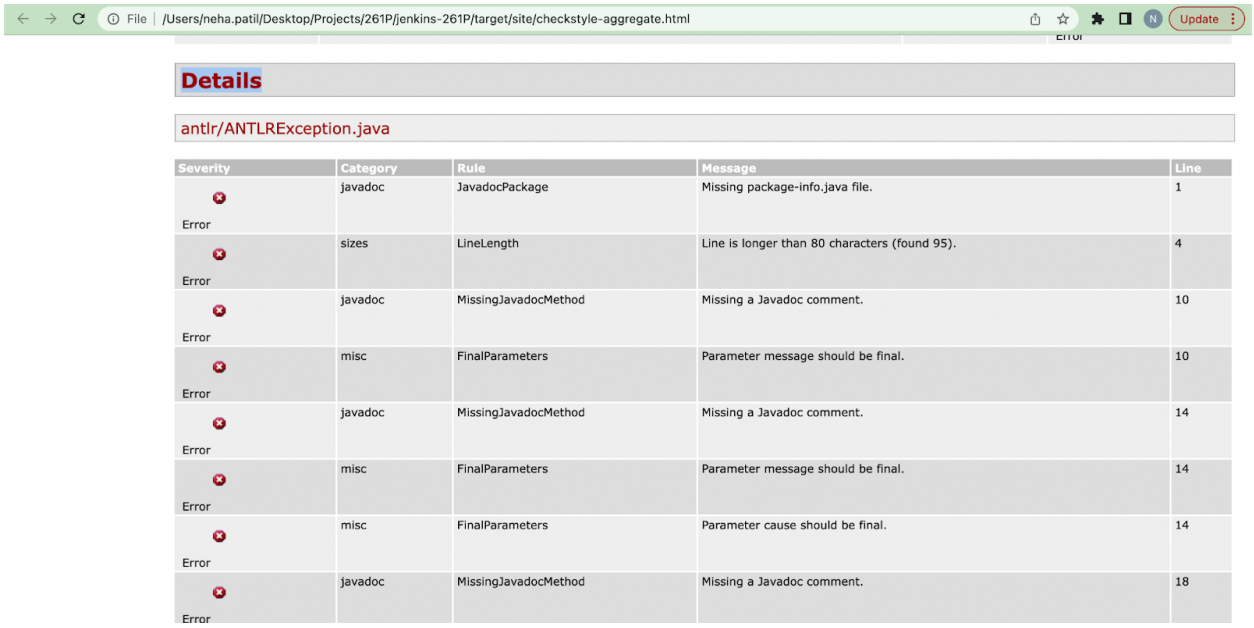
File	I	W	E
antlr/ANTLRException.java	0	0	9
commons-logging.properties	0	0	4
executable/Main.java	0	0	138
hudson/AbortException.java	0	0	4
hudson/AboutJenkins.java	0	0	8
hudson/AboutJenkins/index_uk.properties	0	0	1
hudson/AbstractMarkupText.java	0	0	49
hudson/BulkChange.java	0	0	32
hudson/ClassicPluginStrategy.java	0	0	189
hudson/CloseProofOutputStream.java	0	0	2
hudson/CopyOnWrite.java	0	0	1
hudson/DNSMultiCast.java	0	0	4
hudson/DependencyRunner.java	0	0	17
hudson/DescriptorExtensionList.java	0	0	91
hudson/EnvVars.java	0	0	109
hudson/ExpressionFactory2.java	0	0	25
hudson/Extension.java	0	0	21
hudson/ExtensionComponent.java	0	0	34
hudson/ExtensionFinder.java	0	0	236
hudson/ExtensionList.java	0	0	144
hudson/ExtensionListListener.java	0	0	2
hudson/ExtensionListView.java	0	0	30
hudson/ExtensionPoint.java	0	0	5
hudson/FeedAdapter.java	0	0	13
hudson/FilePath.java	0	0	1284
hudson/FileSystemProvisioner.java	0	0	21

Rules: In this section of the report, we are given a summary of the rules that were used to look for violations. The category of the rules, the quantity of violations, and the seriousness of those violations are all displayed.



Category	Rule	Violations	Severity
blocks	AvoidNestedBlocks	30	Error
	EmptyBlock	7	Error
	LeftCurly	203	Error
	NeedBraces	2194	Error
	RightCurly	88	Error
coding	EmptyStatement	4	Error
	HiddenField	1656	Error
	InnerAssignment	37	Error
	MagicNumber	737	

Details: The report's details section gives us information about the violations that actually took place. The information is given down to the line number level.



File | /Users/neha.patil/Desktop/Projects/261P/jenkins-261P/target/site/checkstyle-aggregate.html

Update

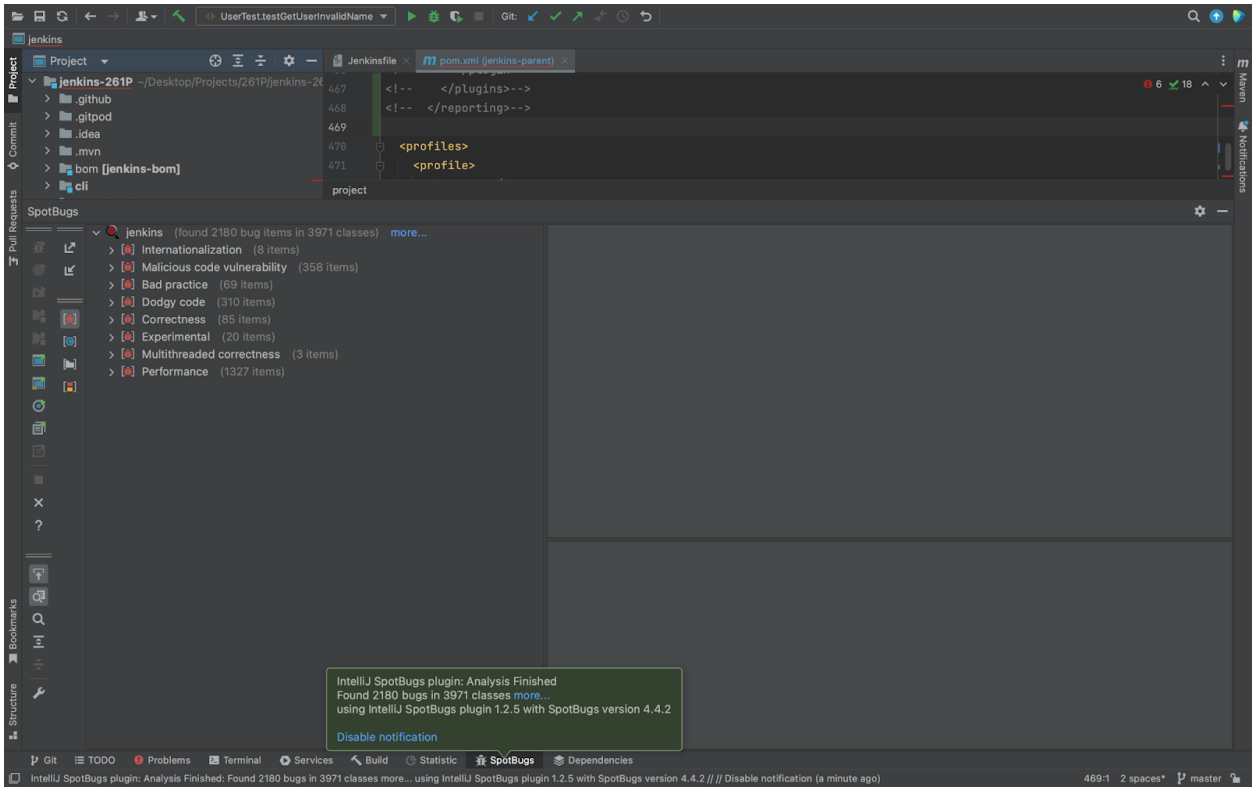
Details

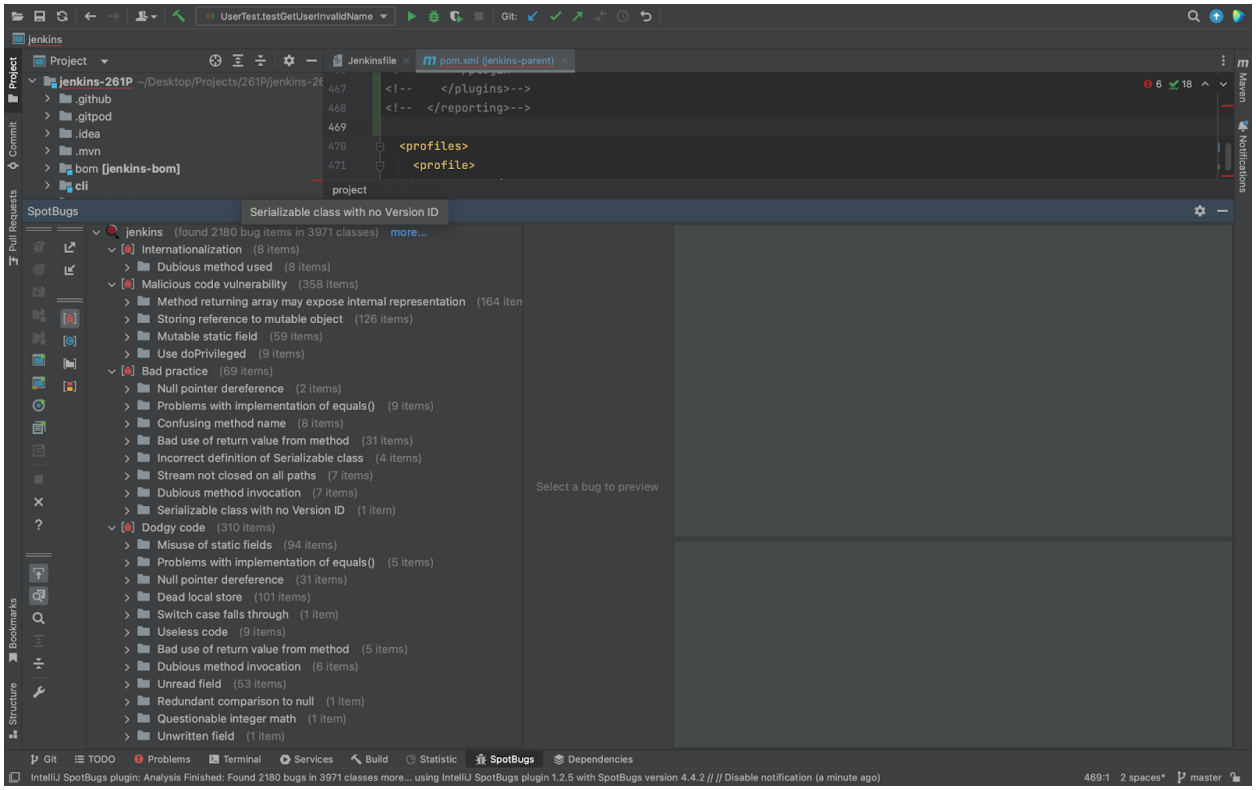
antlr/ANTLRException.java

Severity	Category	Rule	Message	Line
Error	javadoc	JavadocPackage	Missing package-info.java file.	1
Error	sizes	LineLength	Line is longer than 80 characters (found 95).	4
Error	javadoc	MissingJavadocMethod	Missing a Javadoc comment.	10
Error	misc	FinalParameters	Parameter message should be final.	10
Error	javadoc	MissingJavadocMethod	Missing a Javadoc comment.	14
Error	misc	FinalParameters	Parameter message should be final.	14
Error	misc	FinalParameters	Parameter cause should be final.	14
Error	javadoc	MissingJavadocMethod	Missing a Javadoc comment.	18

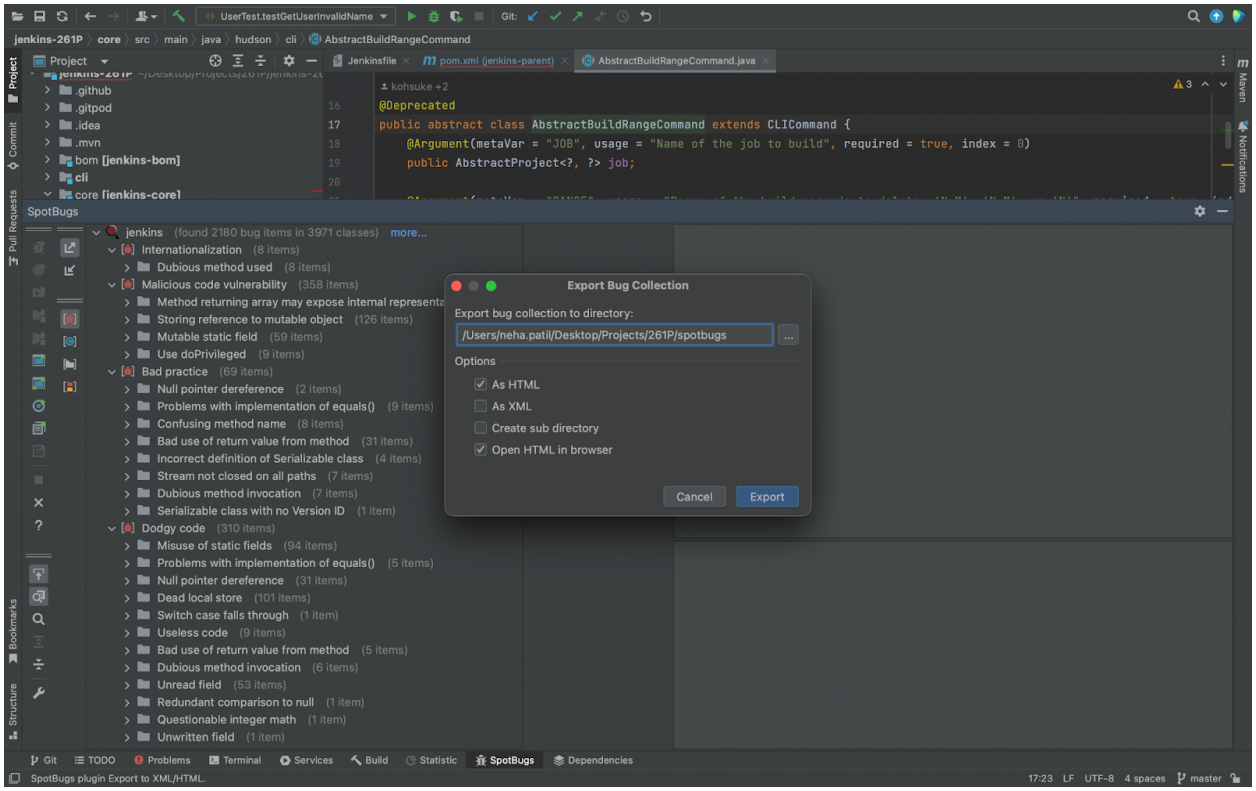
SpotBugs: We are browsing SpotBugs analysis in the IDE itself.

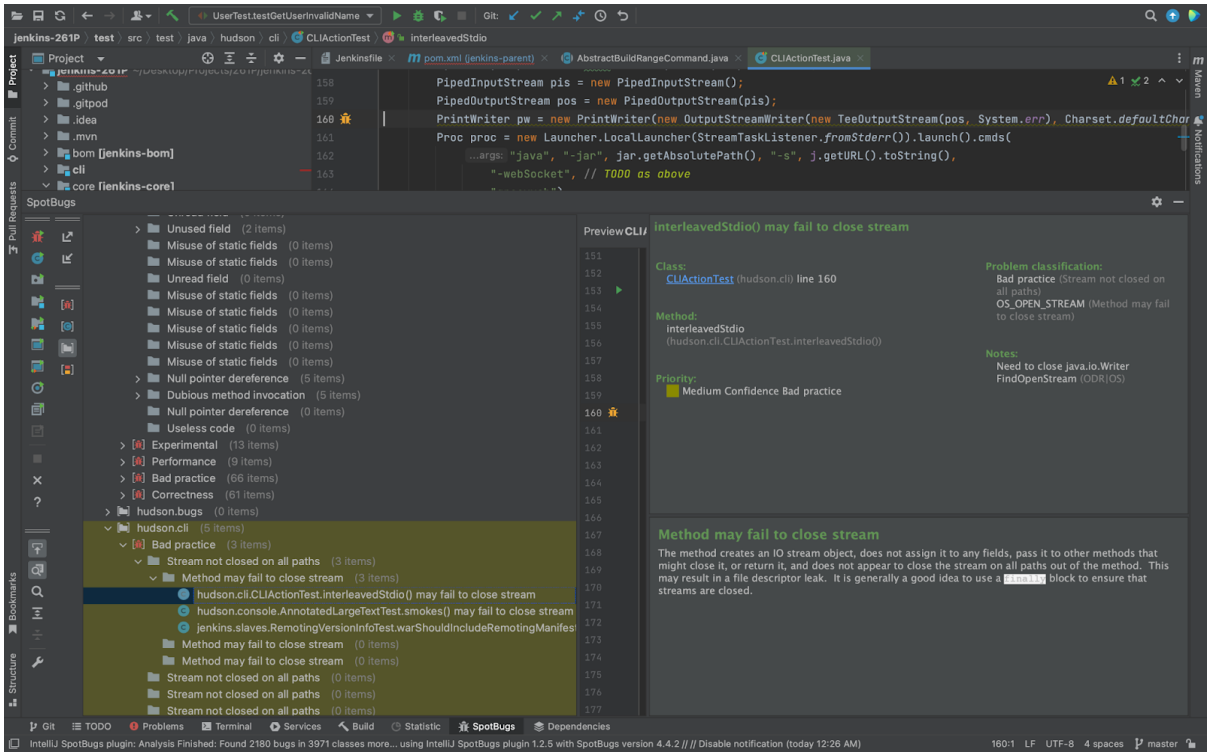
We can now see a SpotBugs pane at the bottom in the IDE. SpotBug has identified **2180** bugs in the entire project. SpotBugs has grouped the errors by the type into 8 types as mentioned below





This report can be exported in the form of an xml or html file to a desired location on the system for convenient browsing.





13.4 For each tool, dive into the warnings that they identify and describe in detail a few. Describe how they are or aren't actual problems in the code.

Checkstyle report analysis:

Error category	Type	Description	Is it a problem in code?
sizes	LineLength	Line is longer than 80 characters	This is not a problem in the code as it is only a styling convention specified for cleaner looking code.
javadoc	JavadocVariable	Missing a Javadoc comment.	This is not a problem in the code as it is only a styling convention specified for more readable code.
blocks	NeedBraces	'if' construct must use '{}'s.	This can be an error in the code if the block has more than 1 statement to execute.

			Otherwise we can skip the braces
design	DesignForExtension	Class 'BuildCommand' looks like designed for extension (can be subclassed), but the method 'run' does not have javadoc that explains how to do that safely. If class is not designed for extension consider making the class 'BuildCommand' final or making the method 'run' static/final/abstract/empty, or adding allowed annotation for the method.	This is currently not a problem in the code but can cause problems if the BuildCommand class is not extended appropriately.
design	FinalClass	Class CLI should be declared as final.	This is not a bug in the code but can lead to an error if this class is accessed by members outside the package.
naming	ParameterName	Name '_args' must match pattern '^[a-z][a-zA-Z0-9]*$'.	This is not a bug in the code but it is a styling convention that is good to follow.
imports	UnusedImports	Unused import - hudson.cli.declarative.CLIMethod.	This is not a bug instead it is redundant code.
coding	InnerAssignment	Inner assignments should be avoided.	This is not a bug in the code but it is a coding convention that is good to follow.
coding	HiddenField	'out' hides a field.	This is not a bug but redundant code.

SpotBugs report analysis:

Error category	Type	Description	Is it a problem in code?
Bad practice	Stream not closed on all paths	interleavedStdio() may fail to close stream The method creates an IO stream object, does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. It is generally a good idea to use a finally block to ensure that streams are closed.	This is a problem in the code as not closing the stream could cause a file descriptor leak
Dodgy code	Useless code	Useless object stored in variable baos of method testOnlyOneEnvVar() Our analysis shows that this object is useless. It's created and modified, but its value never goes outside of the method or produces any side-effects. Either there is a mistake and the object was intended to be used or it can be removed.	This is not a problem in the code but creates redundancy in the codebase.
Malicious code vulnerability	Mutable static field	NodeTest.notTake should be package protected A mutable static field could be changed by malicious code or by accident. The field could be made package protected to avoid this vulnerability.	This is a potential bug in the code which could lead to runtime errors.
Malicious code vulnerability	Mutable static field	AtomicFileWriterTest.tmp isn't final but should be. This static field is public but not final, and could be changed by malicious code or by accident from another package. The field could be made final to avoid this	This is a potential bug in the code which could lead to runtime errors.

		vulnerability.	
Experimental	Logger Problem	Changes to logger could be lost OpenJDK introduces a potential incompatibility. In particular, the java.util.logging.Logger behavior has changed. Instead of using strong references, it now uses weak references internally. That's a reasonable change, but unfortunately some code relies on the old behavior - when changing logger configuration, it simply drops the logger reference. That means that the garbage collector is free to reclaim that memory, which means that the logger configuration is lost	This is a bug as logs could be lost.
Performance	Inner class should be made static	Should ProjectTest$ActionImpl be a _static_ inner class? This class is an inner class, but does not use its embedded reference to the object which created it. This reference makes the instances of the class larger, and may keep the reference to the creator object alive longer than necessary. If possible, the class should be made static.	This is a potential bug in the code which could lead to runtime errors.
Correctness	Redundant comparison to null	Nullcheck of mocked at line 229 of value previously dereferenced A value is checked here to see whether it is null, but this value can't be null because it was previously dereferenced and if it were null a null pointer exception would have occurred at the earlier dereference. Essentially, this code and the previous dereference disagree as to whether this value is allowed to be null. Either the check is redundant or the previous dereference is erroneous.	This is not a problem in the code but creates redundancy in the codebase.
International	Reliance on	Found reliance on default encoding:	This is a potential

ization	default encoding	new String(byte[]) Found a call to a method which will perform a byte to String (or String to byte) conversion, and will assume that the default platform encoding is suitable. This will cause the application behavior to vary between platforms. Use an alternative API and specify a charset name or Charset object explicitly.	bug in the code which could lead to runtime errors.

13.5 Difference between analysis provided by Checkstyle and SpotBugs

Popular static analysis tools Checkstyle and SpotBugs each have their own strengths and capabilities.

Analysis done by SpotBugs, potentially detects a bug or defects in code like null pointer dereferences, deadlocks, and infinite loops. On the other hand, Checkstyle is a tool that verifies Java code in accordance with a list of coding standards or conventions. It looks for formatting, naming, and indentation errors as well as stylistic problems. Checkstyle is adaptable and may be combined with build tools like Maven and Ant to enforce particular coding standards. Whereas SpotBugs analyzes the code and produces reports of any potential problems using a set of predefined rules.

As we can go over the report we find that analysis done by CheckStyle is more on coding conventions. It does not say anything about the deadlocks, potential bugs. It talks more on the naming conventions, unused imports, missing braces etc., Whereas if we look at the report found by SpotBugs we can find ‘Redundant comparison to null’ which talks about the null reference found in the code. SpotBugs’ analysis also highlights the potential security issues that may arise by the use of java.util.logging.Logger. Also on the analysis we can find the AtomicFileWriterTest.tmp whose value can be changed by others but in actuality it would be best practice to declare it as final.

In conclusion, while both tools are used for static analysis of Java code, Checkstyle concentrates on enforcing coding standards and conventions whereas SpotBugs focuses on identifying potential problems and defects.

15. References

1. Project referenced : https://github.com/jenkinsci/jenkins
2. Build procedure : https://github.com/jenkinsci/jenkins/blob/master/CONTRIBUTING.md
3. Developer tools: https://www.jenkins.io/doc/developer/tutorial/
4. Junits : https://www.vogella.com/tutorials/JUnit/article.html
5. https://greg4cr.github.io/courses/spring18csce747/Lectures/Spring18-Lecture4FunctionalTesting.pdf
6. https://www-users.cse.umn.edu/~heimdahl/csci5802-spring02/readings/book-ch13-functional.pdf
7. Test Document: https://www.jenkins.io/doc/developer/testing/
8. https://www.jetbrains.com/help/idea/code-coverage.html
9. https://www.geeksforgeeks.org/structural-software-testing/
10. https://www.cloudbees.com/continuous-delivery/continuous-integration
11. https://www.scaledagileframework.com/design-for-testability-a-vital-aspect-of-the-system-architect-role-in-safe/
12. https://www.geeksforgeeks.org/design-for-testability-dft-in-software-testing/
13. https://www.geeksforgeeks.org/software-engineering-mock-introduction/
14. https://www.telerik.com/products/mocking/unit-testing.aspx
15. https://www.baeldung.com/checkstyle-java
16. https://www.baeldung.com/intro-to-findbugs
17. https://spotbugs.readthedocs.io/en/stable/links.html
18. https://somindagamage.medium.com/how-to-configure-checkstyle-and-findbugs-plugins-to-intellij-idea-38148aad2387
19. https://www.securecodewarrior.com/article/what-is-static-analysis#:~:text=Static%20Analysis%20is%20the%20automated,Security%20vulnerabilities.
20. https://www.perforce.com/blog/sca/what-static-analysis
21. https://checkstyle.sourceforge.io/config_annotation.html