Jira and Xray cloud setup

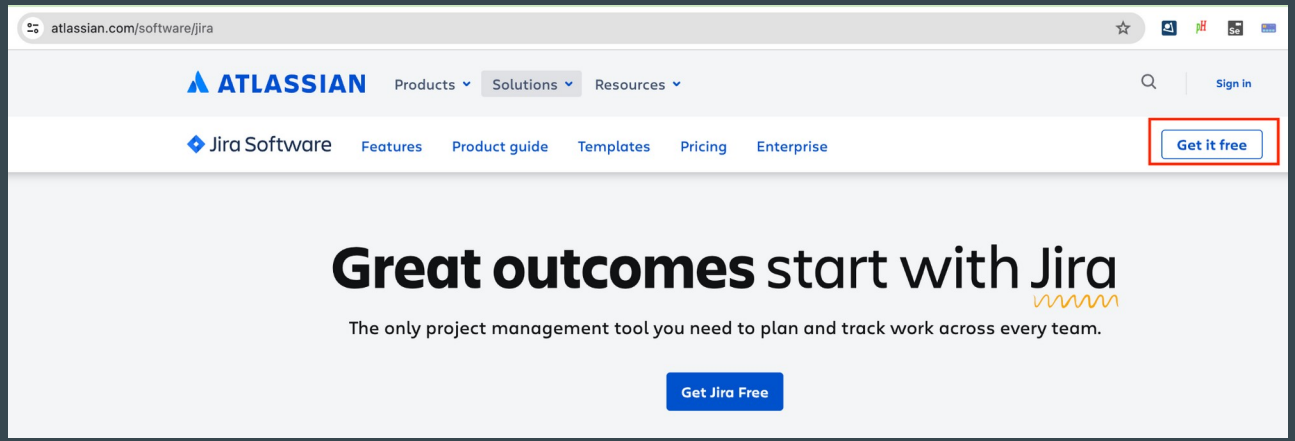
...

Quick setup instructions of a Jira Software + Xray cloud instance

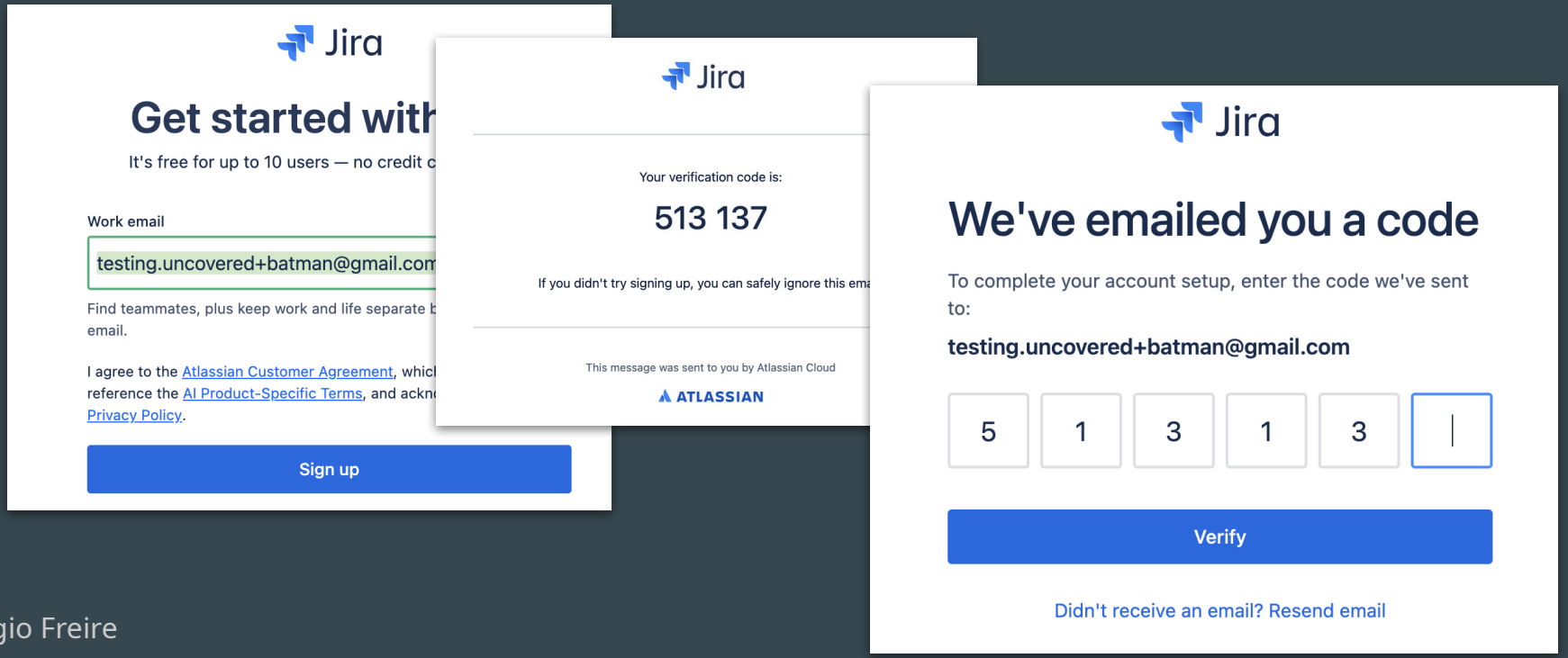
Sergio Freire

Jira cloud setup

https://www.atlassian.com/software/jira



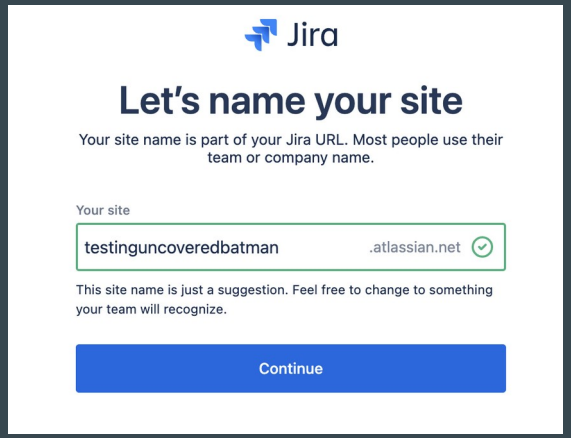
Jira cloud signup

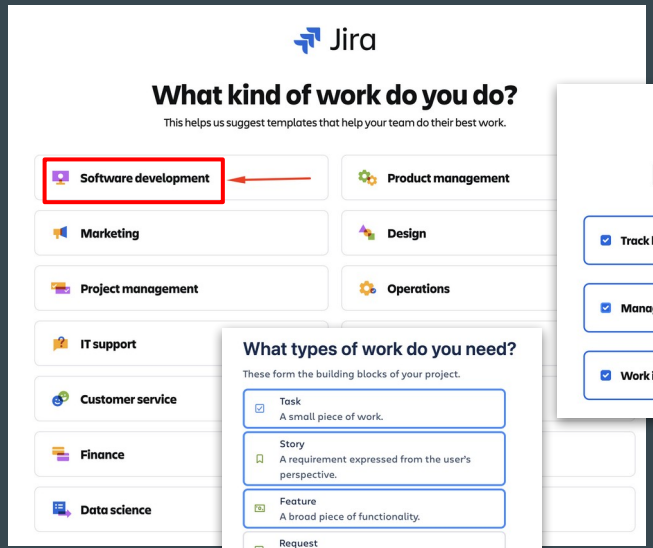


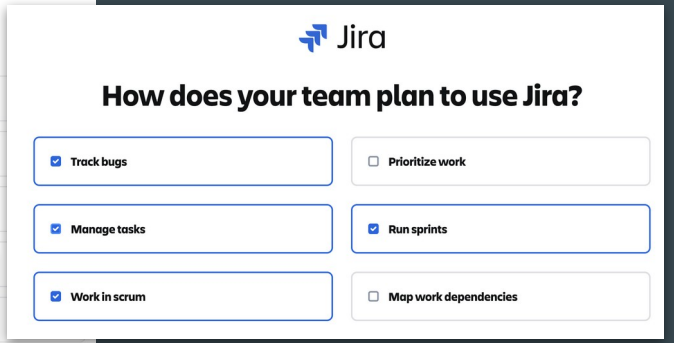
Sergio Freire

Jira cloud signup

Take note of your Jira Cloud instance URL. You'll be asked some questions that you may skip.



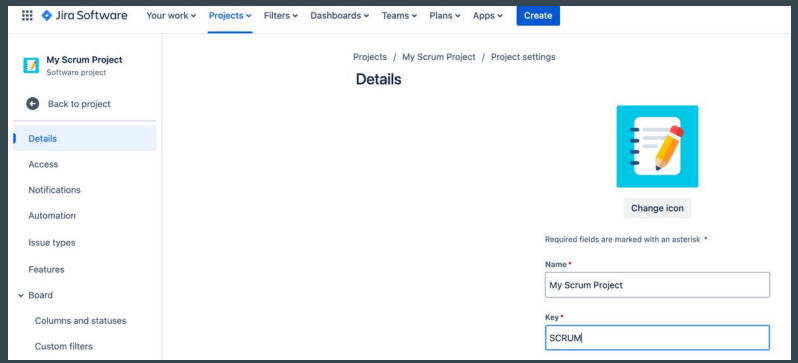




Sergio Freire

Project creation during signup

During the Jira cloud instance signup, you'll be asked to create your first project; this will create a *team-managed* project named "My Scrum Project", having the key "SCRUM"; therefore, all issues on that project will have keys like SCRUM-<int> (e.g., "SCRUM-1"). You can change the project name and key on the project settings panel.

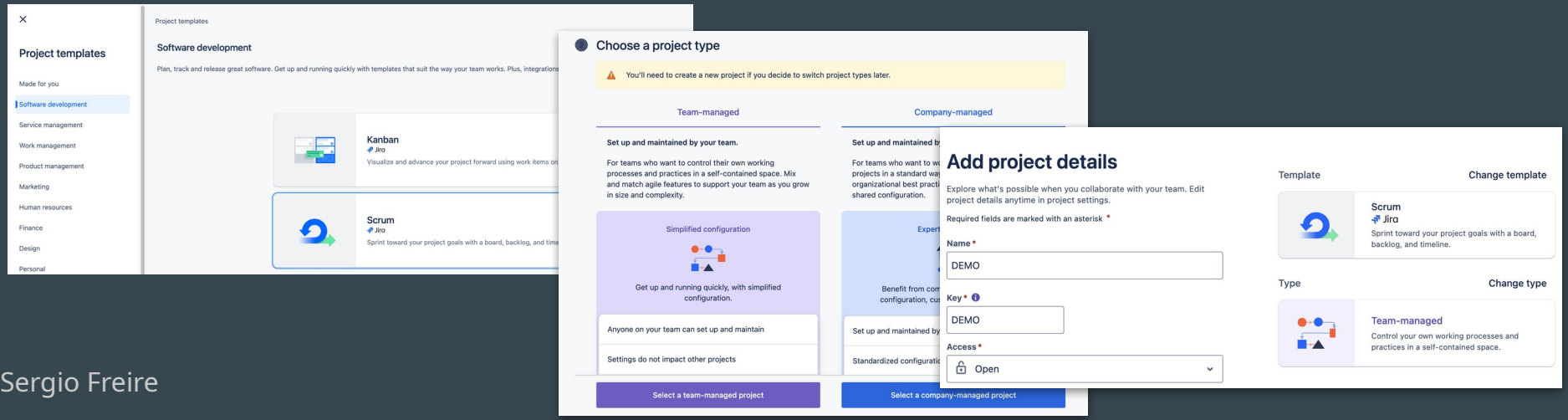


Creation of projects

You can also create your project later on, from the top Projects menu.

Have in mind though that there are 2 main project types in Jira Cloud:

- **Team-managed** (simpler but not so powerful; good enough for us)
- Company-managed (more complex and more powerful)

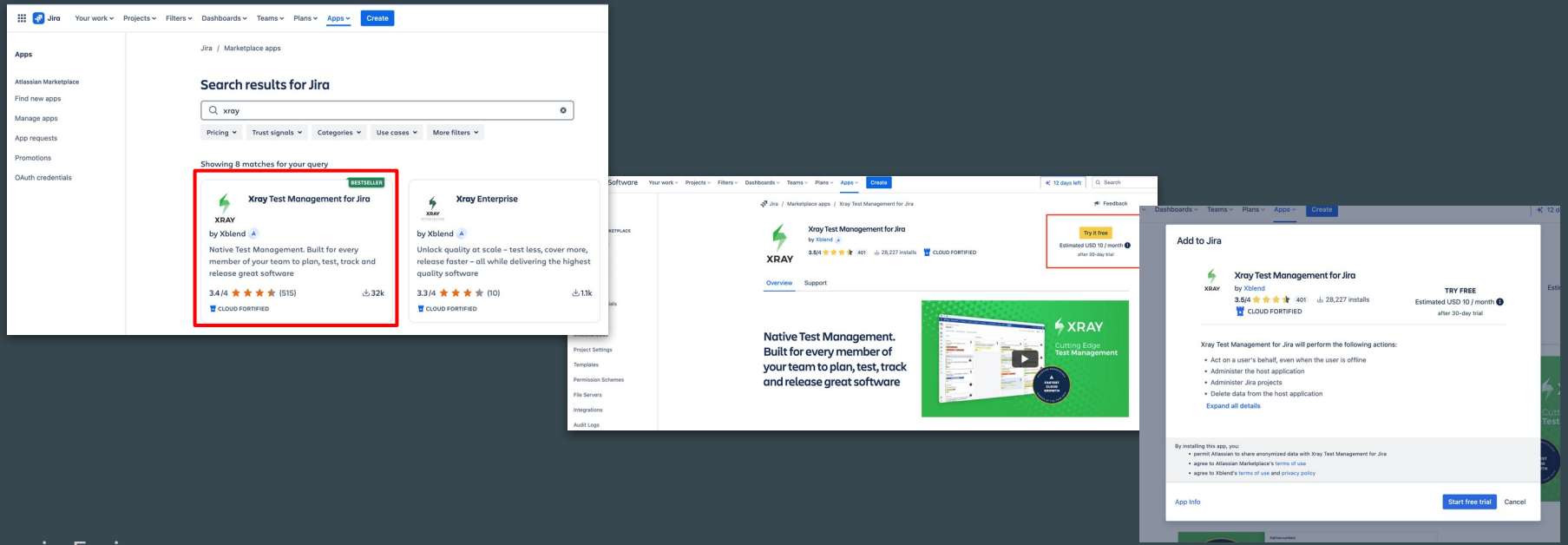


Xray Installation

Sergio Freire

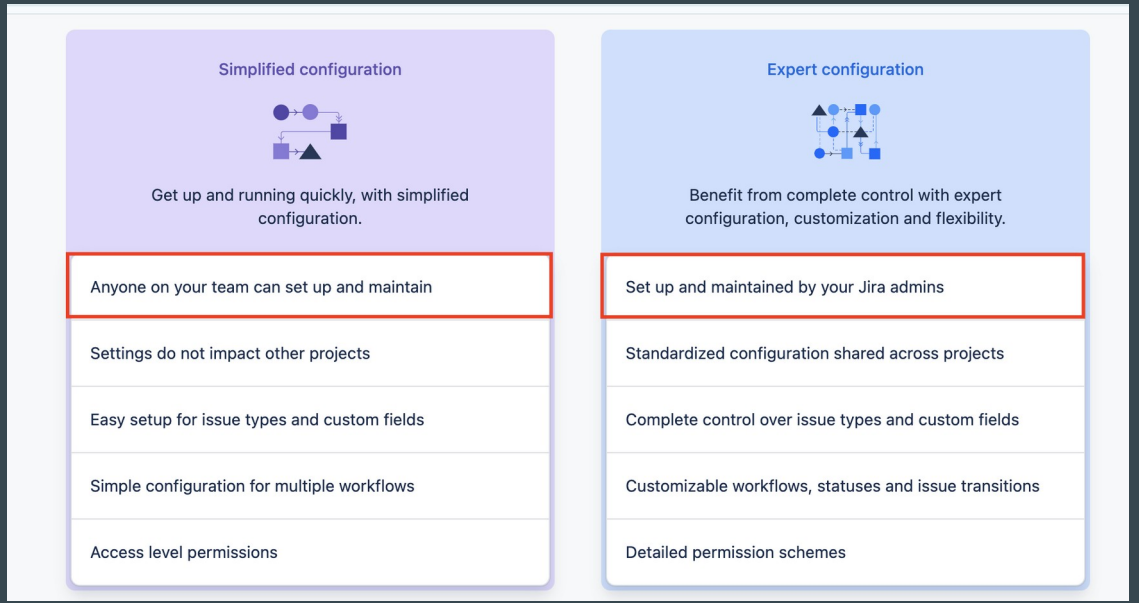
Installation of Xray

Installation is quite straightforward; go to Apps > Explore more apps > Xray and then “Try it free”. You need to use the account that created the Jira



Enabling Xray on Jira projects

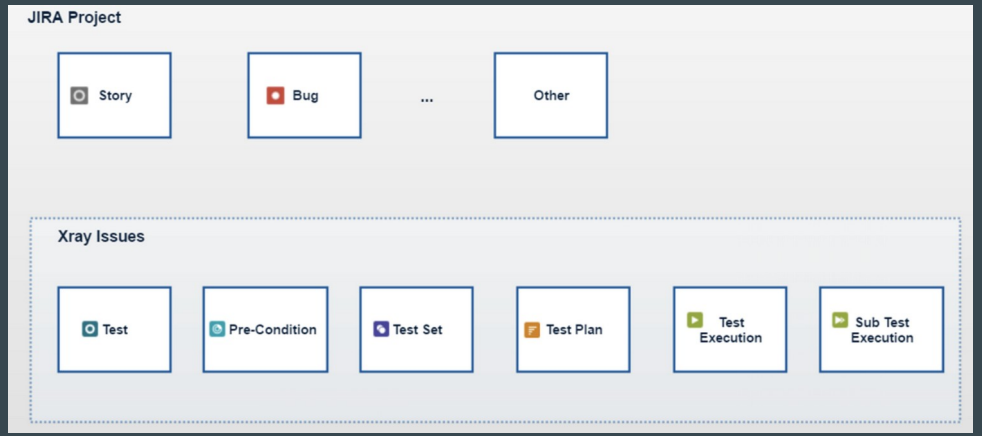
- Note: there are 2 main project types in Jira Cloud:
  - Team-managed
  - Company-managed
- We can enable Xray in any of these project types but the configuration steps are slightly different



Organization strategy: “All-in-One”

A single project to manage your “Requirements” (stories, epics), Defects (bugs), and Test related issues and also have all your Test Executions.

The idea is to use one project to manage everything related to it, including testing.



Enabling Xray on team managed projects

Sergio Freire

Setting a team-managed project in a nutshell

Create or use an existing team-managed project. You can use the project you created during the Jira signup.

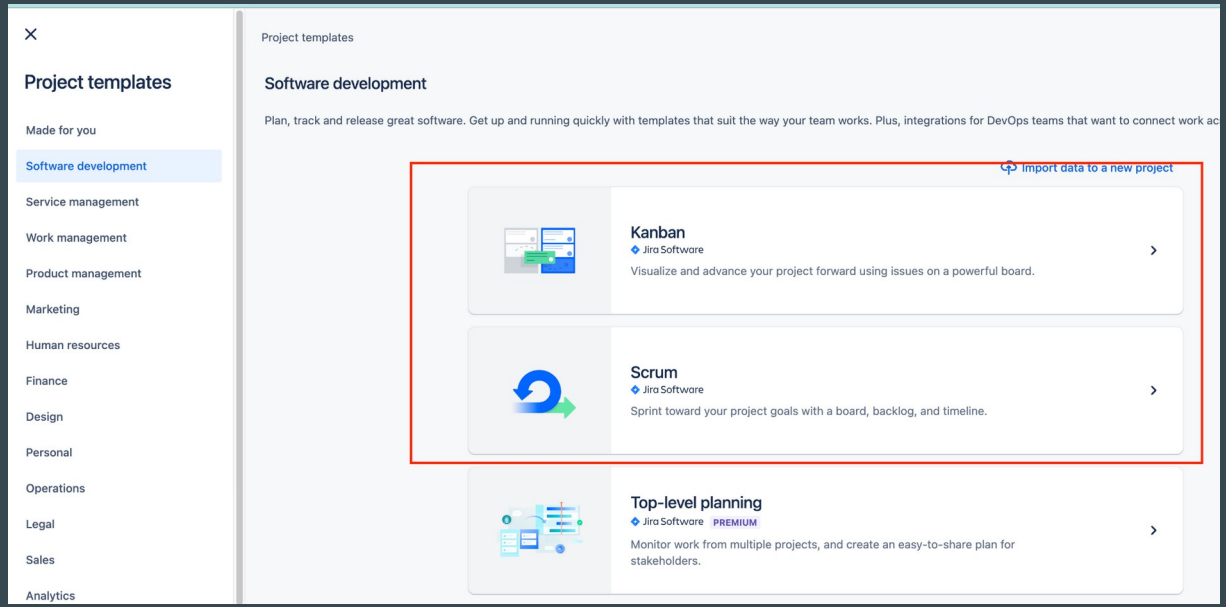
Then:

1. Add (i.e., create) issue types to the project and map them to the Xray concepts
    a. Precondition, Test, Test Set, Test Execution, Test Plan
2. Define which items you consider to be “requirements” like
3. Define which items you consider to be “defects” like

That’s it.

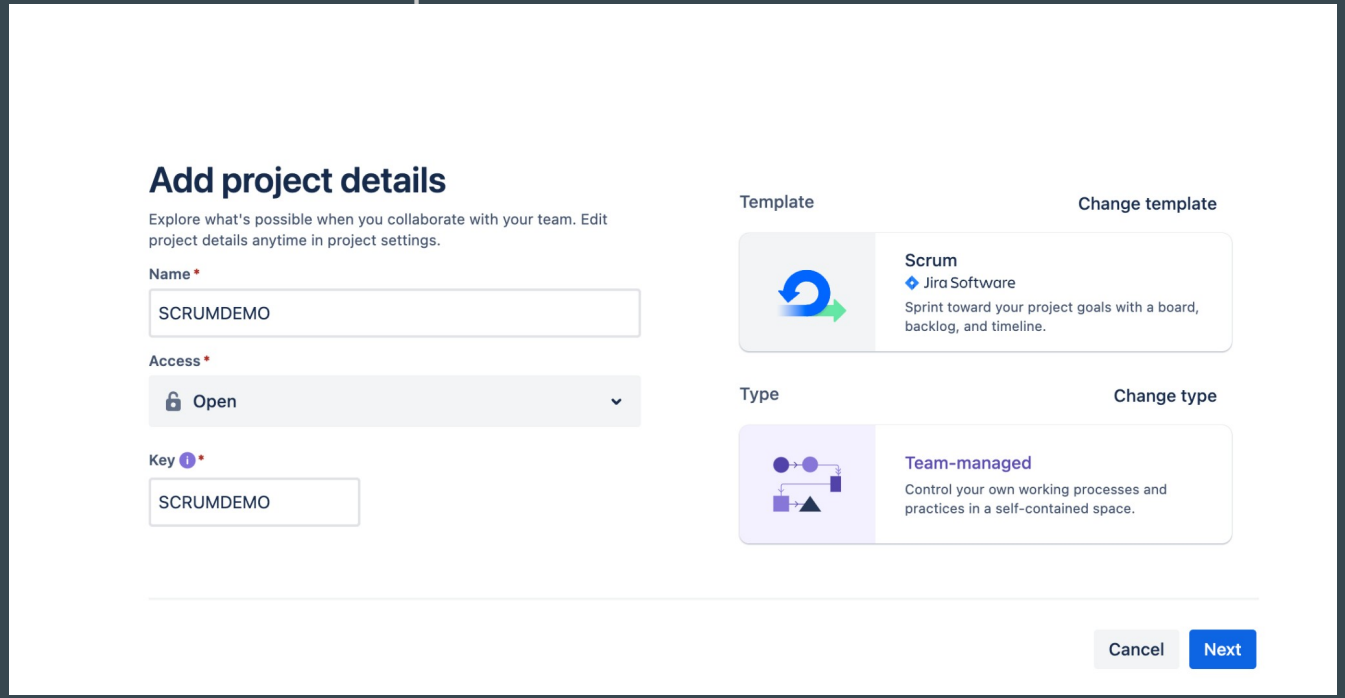
Create project (unless you have created one already)

- Create a Jira project using a Kanban or Scrum template



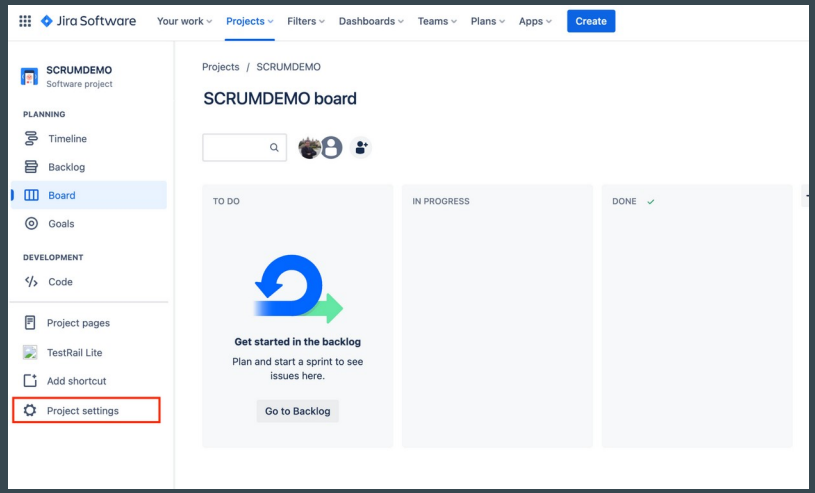
Create project (unless you have created one already)

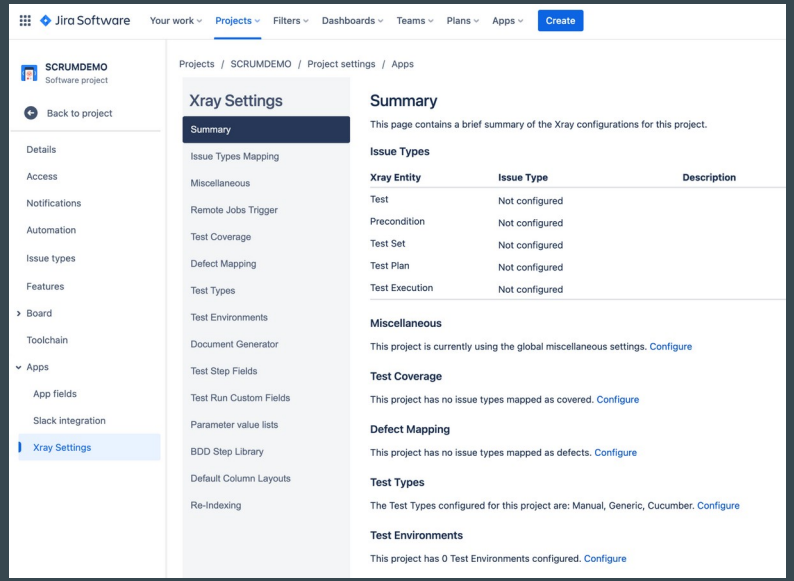
- We'll use the Scrum template



Xray project setup

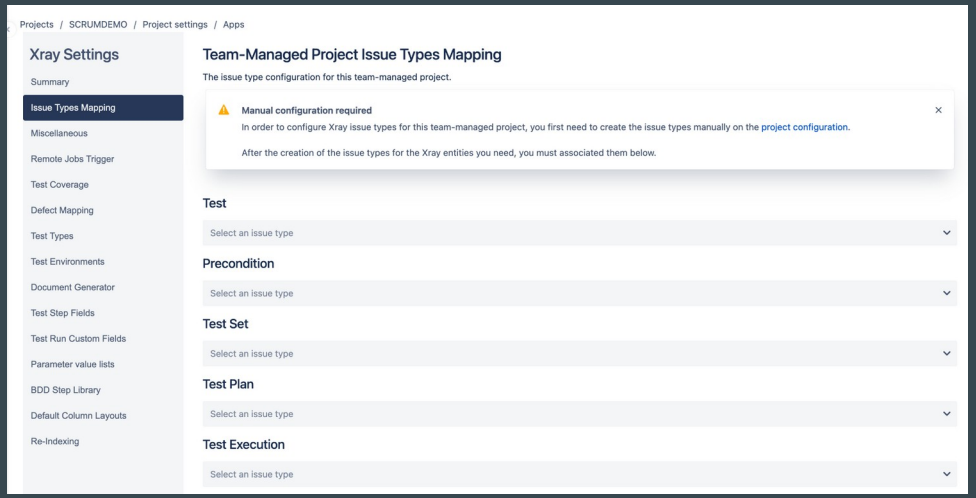
- Go to **Project settings > Apps > Xray Settings**
- There are a few configurations to perform

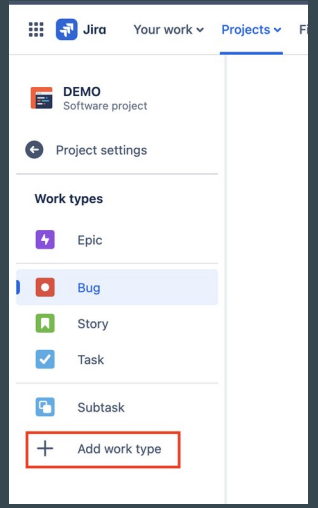




Xray project setup

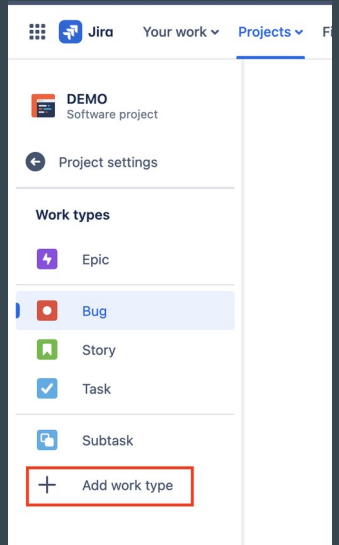
- Go to **Issue Types Mapping** to configure the issue types
- Then go back to **Project settings > Issue Types** using the “project configuration” link to create the issue types used by Xray

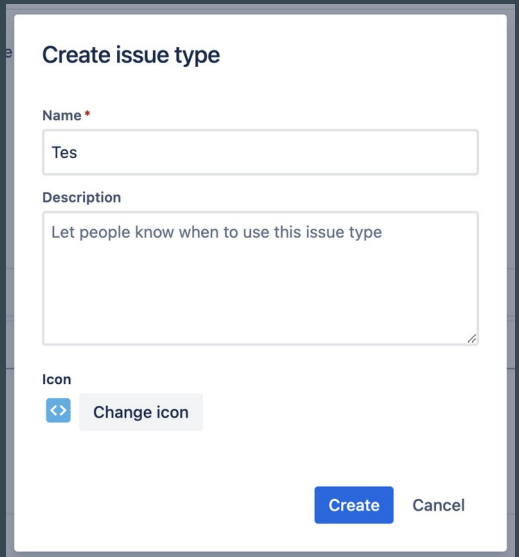




Xray project setup: creation of issue types

- Create the following issue types (also called “work types”) with these names:
  - Precondition, Test, Test Set, Test Plan, and Test Execution
- Optionally: configure the related icons (download them here)



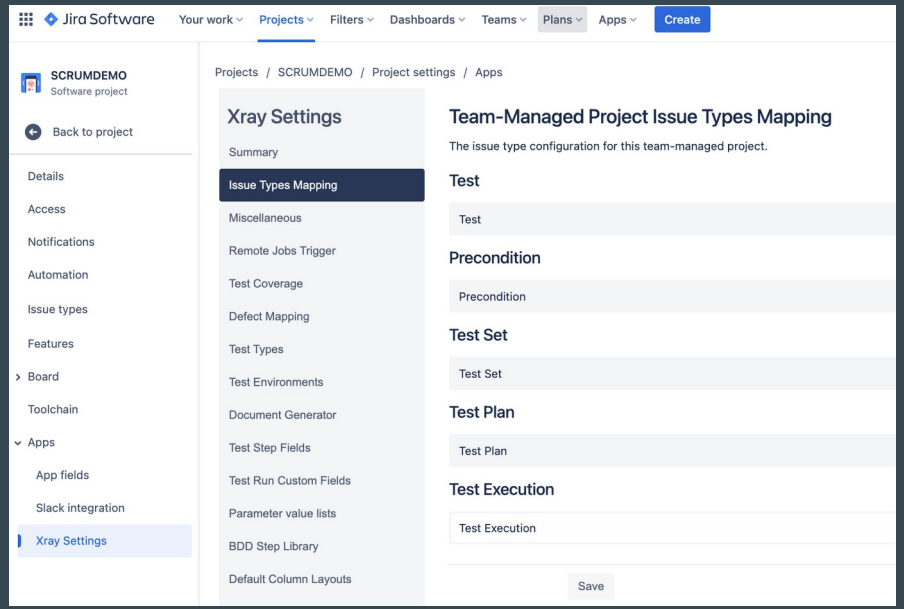


Xray project setup: associate the issue types

This step is for Xray to be aware of the issue types we want to use for its own entities; we could have created “Test Case”, “Test List” issue types (for example) and associate them with the Test and Test Set concept of Xray.

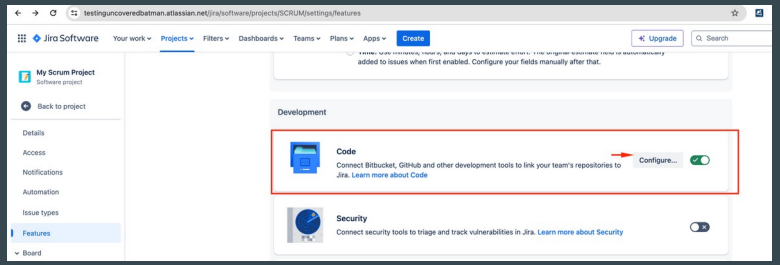
However, in our case, we’ll keep it simple, using issue types with the same names of the expected entities:

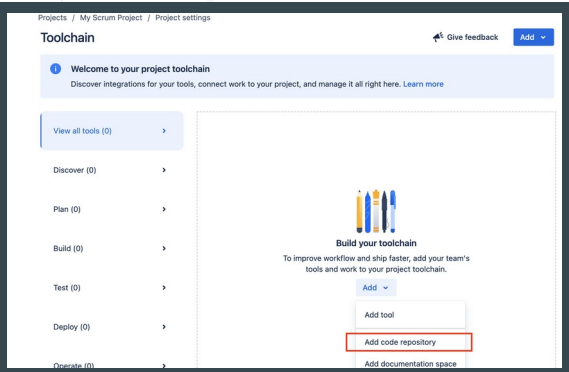
- Test > Test
- Precondition > Precondition
- Test Set > Test Set
- Test Plan > Test Plan
- Test Execution > Test Execution

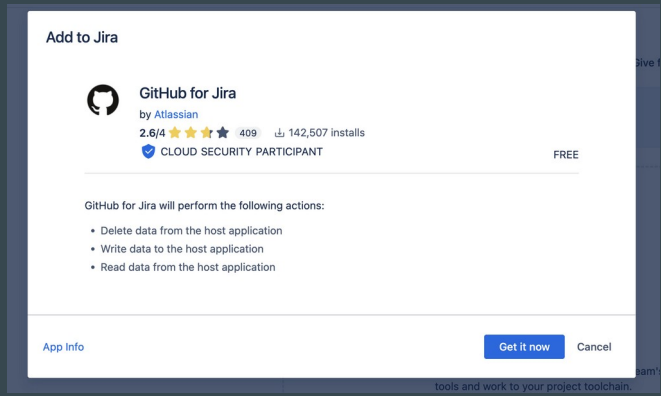


Nice-2-Have

In project settings > **Features**, enable Code and add the code repository tool (e.g., GitHub).



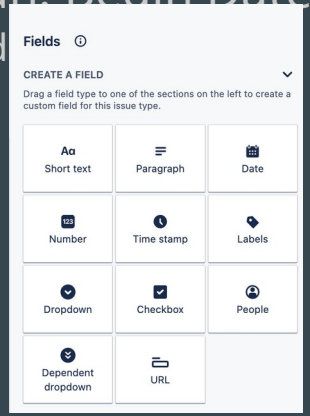


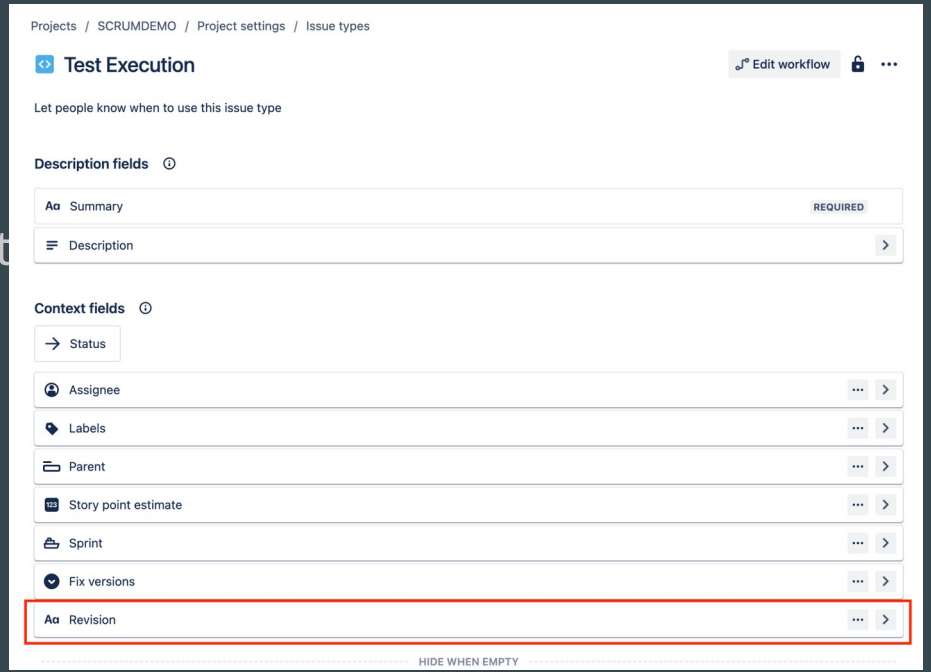


OPTIONAL: Associate some custom fields to Xray issue types

On **Project settings > Issue types,** for the following issue types add the fields

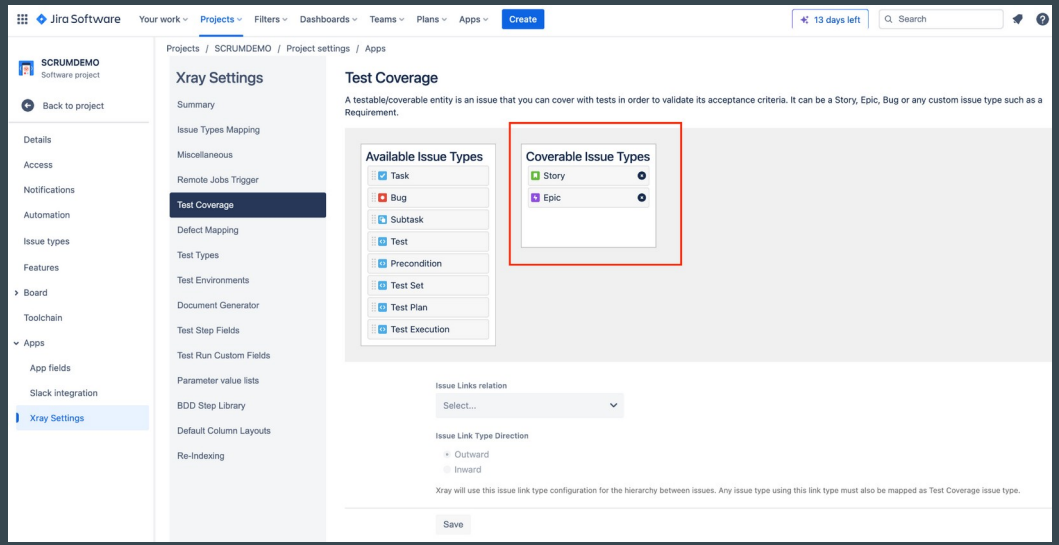
- Test Execution: Revision
  - As short text
- Test Plan: Begin Date and End Dat
  - As d





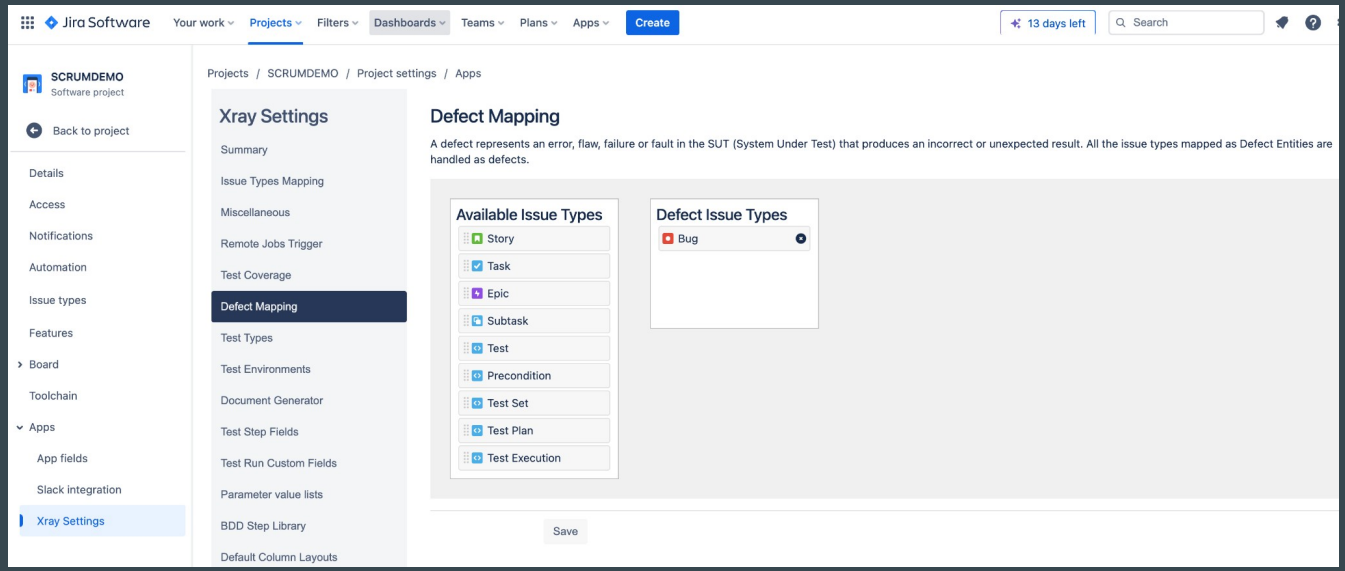
Xray: define “requirements” / issues coverable by tests

- **Coverable Issue Types**: *Which issue types do you aim to cover with tests? (e.g., Story, Epic)*



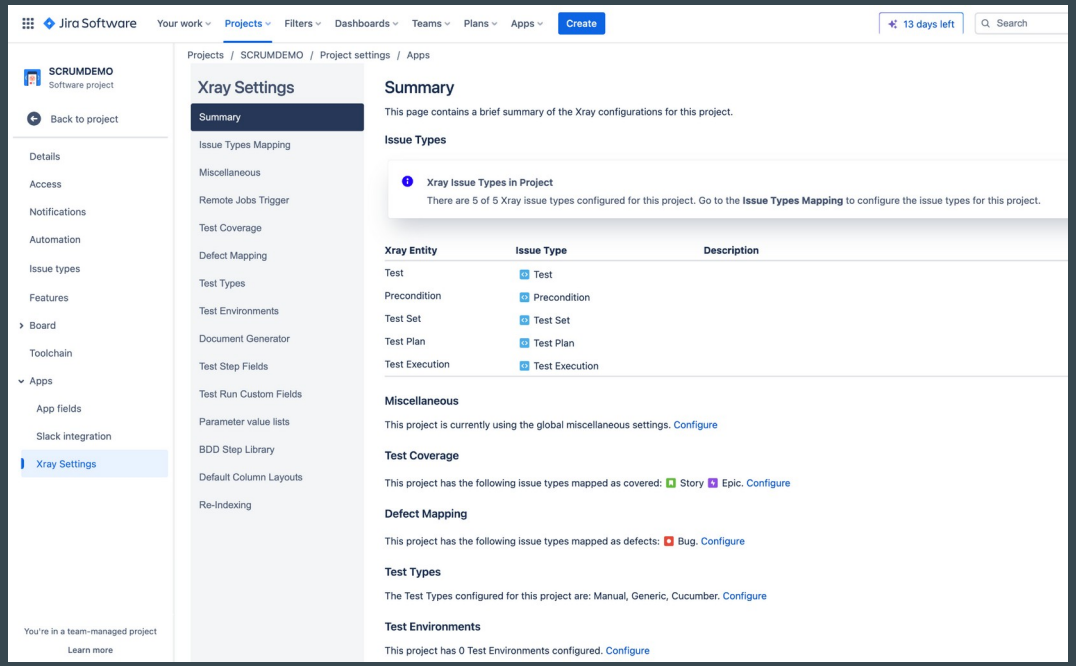
Xray: define defects

- **Defect Issue Types**: *Which issue types do you want to be handled as defects? (e.g., Bug)*



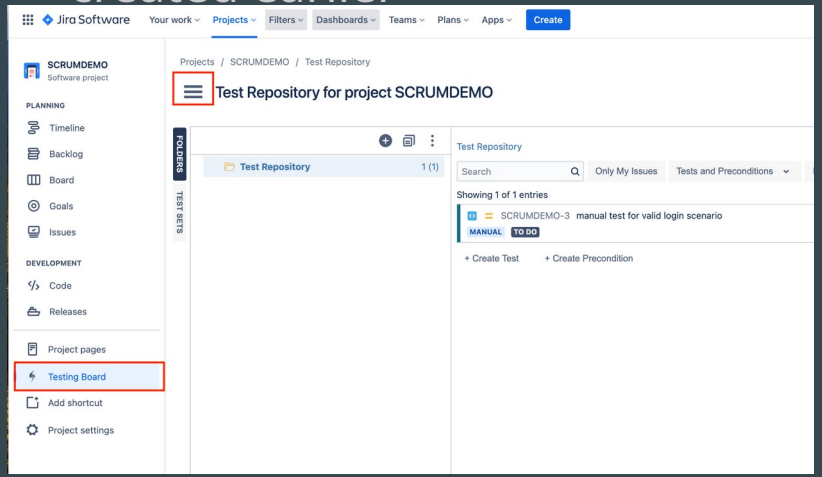
Xray: final checkup

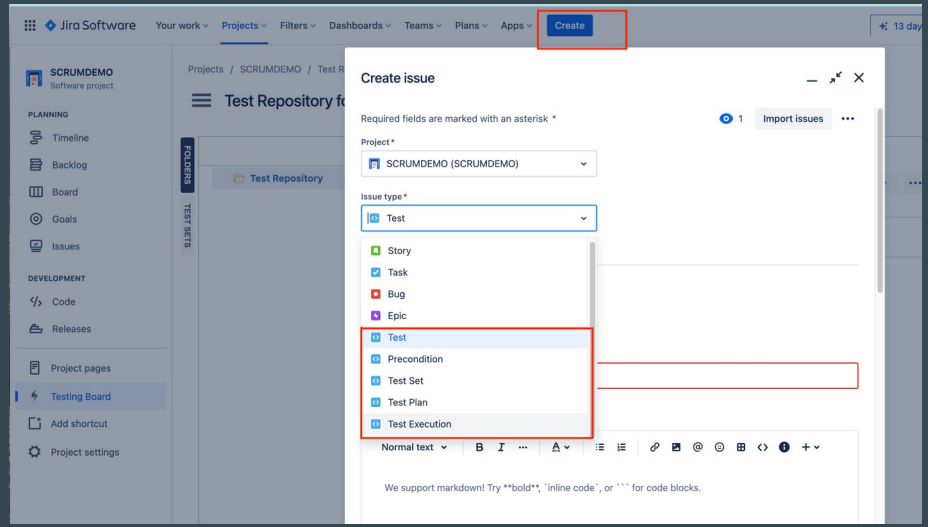
Go to **Project settings > Apps > Xray Settings > Summary.**



If all goes well...

- You should see a “Testing Board” shortcut on the side menu
- Whenever creating issues, you should see the issue types that were created earlier

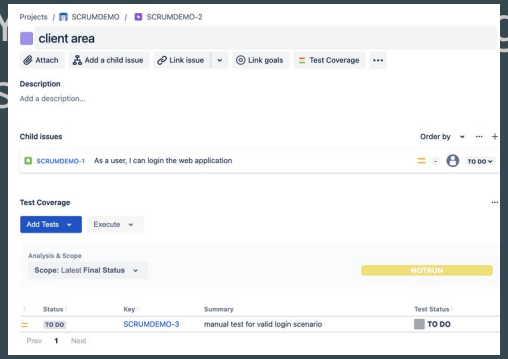


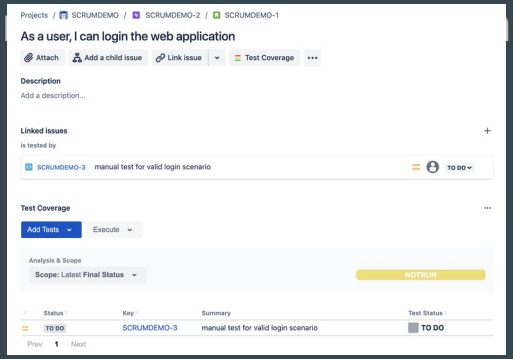


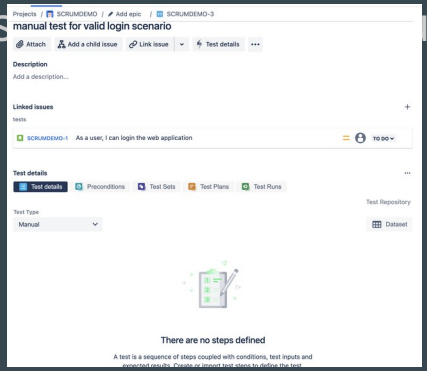
Try it out, just in case :)

1. Create project with your name in uppercase (e.g., “SCRUMDEMO”); notice the project key (e.g., “SCRUMDEMO”)
2. Create Epic “client area” and a Story “login”; associate Story to the parent Epic
3. Create a Test from the Story screen

Y ge info d als ue s

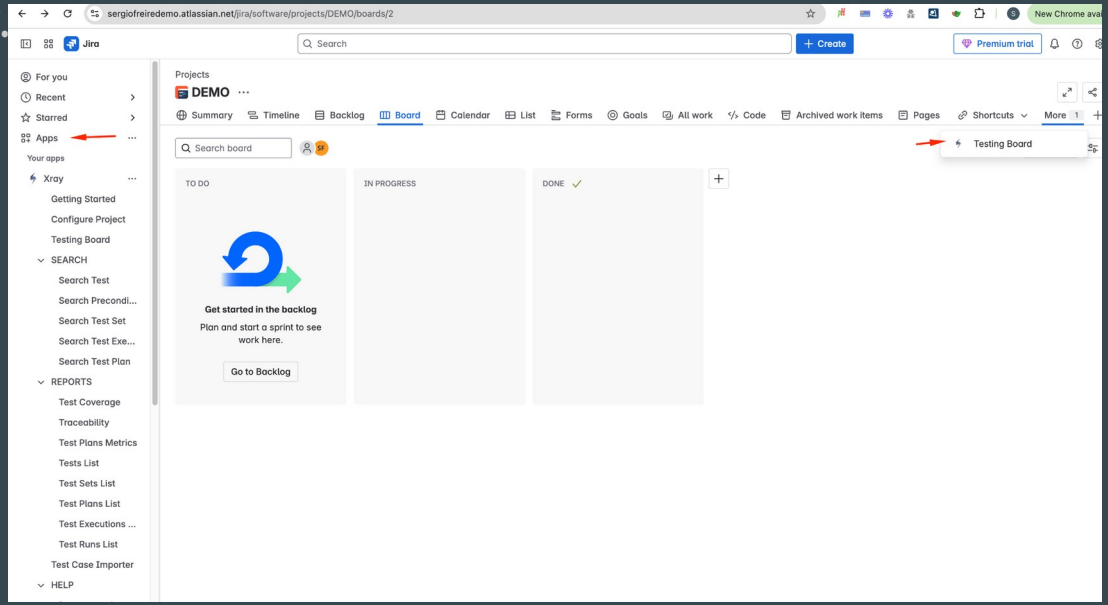






NOTE: new Jira UI coming during 2025...

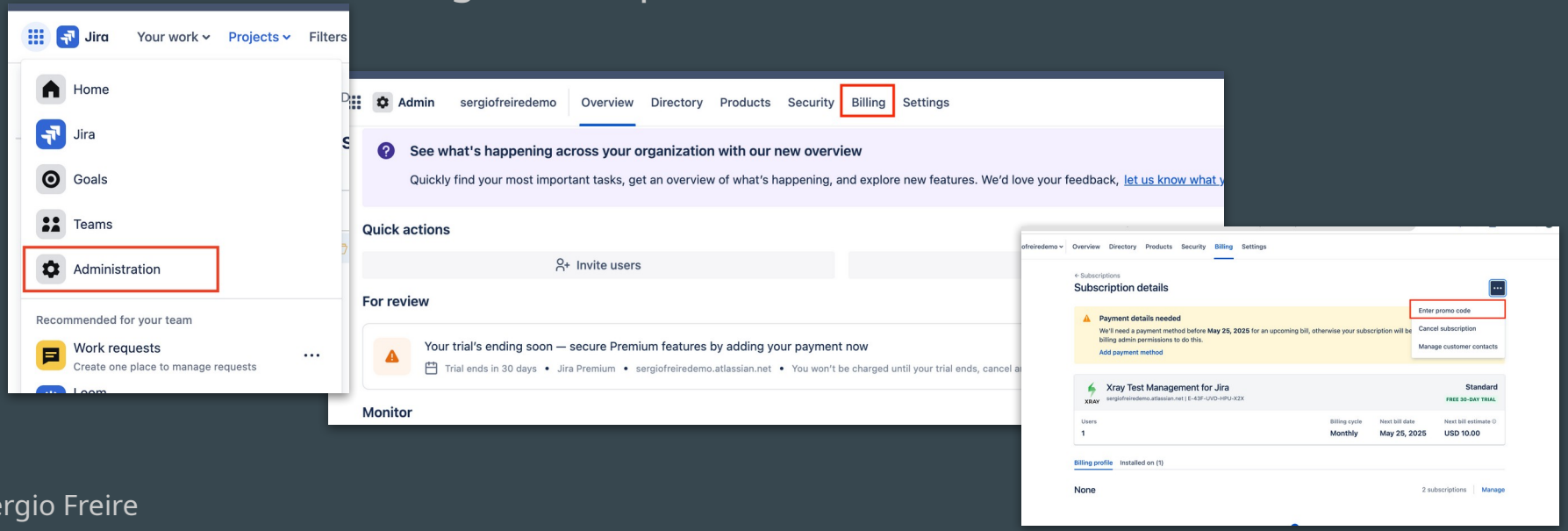
Jira is changing the UI so the “Testing Board” page from Xray will appear on a different place. On the left side menu, Xray related pages are available from within Apps > Xray.



Sergio Freire

Applying promo code

Your Xray trial ends in 1 month, so you must use the “promo code” given by your teacher. Each group can only use the promo code once! Go to Jira > Administration > Billing > Enter promo code.



Xray Entities & Concepts

...

Sergio Freire

Xray entities & concepts (a few)

Xray provides several concepts; we'll focus just on a few ones.

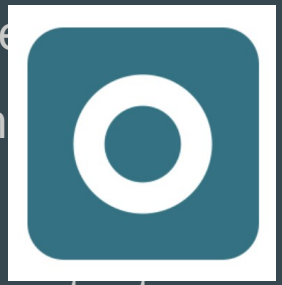
- Test
- Test Run
- Test Execution
- Test Plan
- "Requirements"
- "Defects"

All of the previous entities will be issues in Jira/Xray, except the Test Run which is

Test

An abstraction of a test idea/scenario, automated test, and thus re

It is essentially a way to verify/validate the associated requiremen

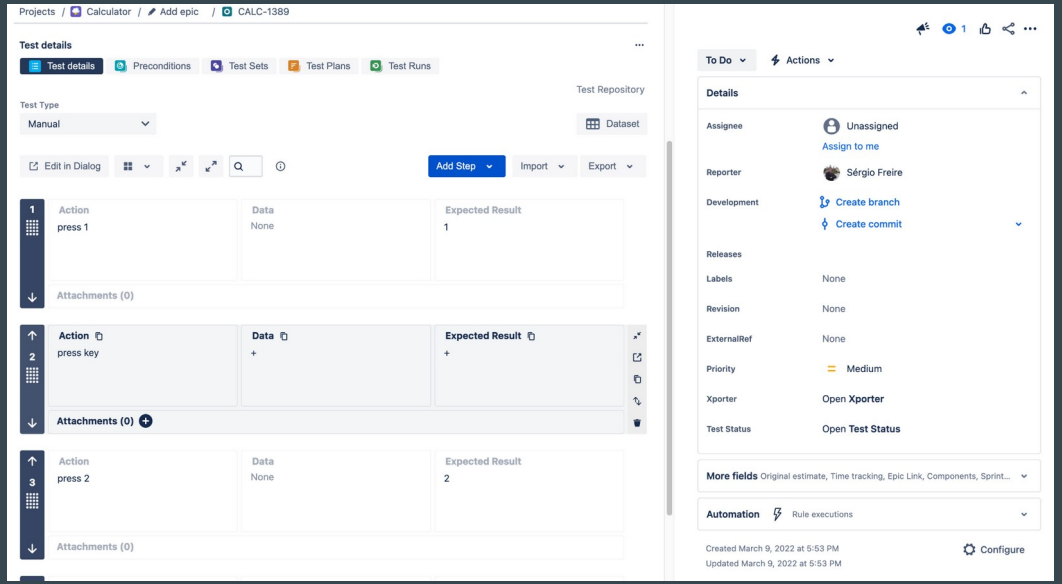


- Can be a scripted (e.g. test case, automated test) or exploratory test
- Can be specified using Gherkin (Scenario)
- Can be executed manually in Jira/Xray or through automation
- May be linked to/cover 1 (or more) requirements, using the “tests” issue link type
- Has one type: “Manual”, “Cucumber”, “Generic”

Test: Manual

A traditional test case composed by a list of steps, thus scripted.

- Each step is mainly defined by:
  - Action/step
  - Expected result

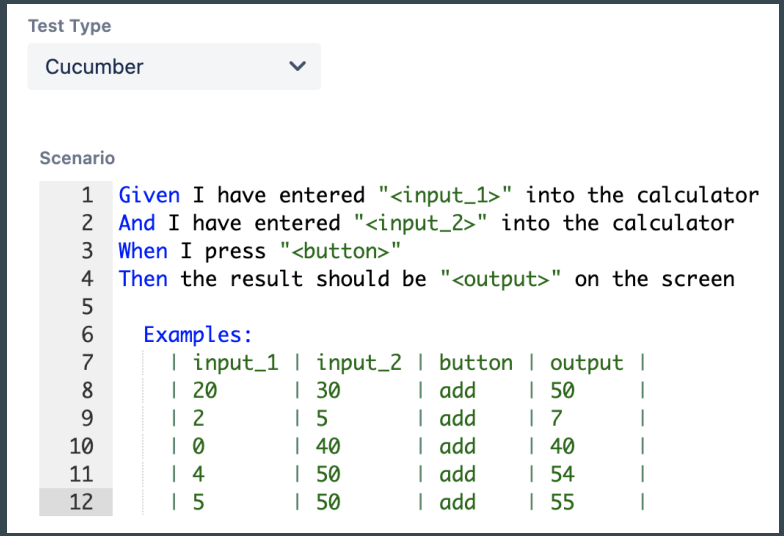


If we don’t have test automation yet, or the test scenario is hard/costly to automation, we can create a “manual” test and link it to the related Story, for example. We can then manually record its result in Xray.

Test: Cucumber

A Cucumber Scenario/Scenario Outline that provides one or more examples of an acceptance criteria.

- Write tests in a business-readable domain-specific language (Gherkin)
- Specify Cucumber Scenarios and Scenario Outlines with Gherkin syntax highlighting
- Ability to export to .feature files and execute during Continuous Integration

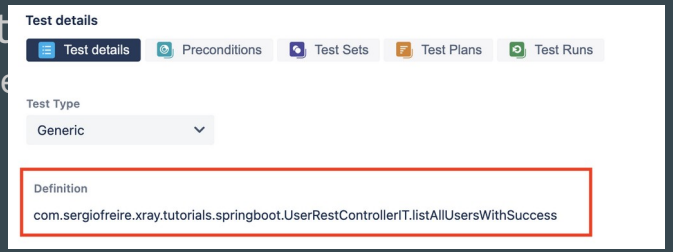


We can use Xray as the master for the Cucumber/Gherkin scenarios or we can use Git instead. The flows for fully integrating the results from automation are slightly different as shown in this tutorial.

Test: Generic

A way to abstract and have visibility of traditional automated tests or exploratory tests.

- Allows you to track the results of automated tests that are non-Cucumber (e.g. JUnit)
  - Automate tests in any framework and report results back to Xray
  - These tests are usually auto-provisioned whenever importing the results the first time
  - Definition field contains a unique ident
    <package_name>.<class_name>.<method_name



Note: Generic tests could also be used to have visibility of Cucumber results if upload JUnit XML reports. In that case we lose visibility of the individual Gherkin sentences; we would just have visibility of the whole test result.

Test Execution

A “task” for executing a group of tests on a given version of the system, on a given environment. This task will also contain the results when they’re reported. Sometimes the Test Execution is created alongside with its results (i.e., the Test Runs).

- Contains a list of Tests and their results (i.e. Test Runs)
- May be planned (especially for manual tests) or ad hoc

- May be created manually (i.e., “I want to run these tests...”);
  - Later on someone will “execute” it manually, by going to each test and record the actual results and weather the test passed or failed
- May be created during Continuous Integration
  - in this case it will already contain the results (i.e., a Test Run for each Test that was executed)

Test Run

An instance of a Test in the context of a Test Execution. A run of a Test in some scope.

- Contains the results obtained for a Test, including evidence and linked/reported defects
- For compliance, also contains a copy of the Test specification at the moment of execution
- It’s an internal entity; not a Jira issue

Test Plan

A way to define the scope for testing and track its progress;
what testing we'll be performing and its latest status.

- Tracks a group of tests and their results independently of the number of executions
- Can be used in a planned way, with its scope for testing (i.e. the Tests) defined beforehand
- Can be used in an Agile way, acting as a testing guidance result aggregator, by allowing you to create/add Tests along with their results at any time
- Test Plans may be assigned to versions, sprints and users, as they're a Jira issue
- A version or a sprint may have multiple Test Plans

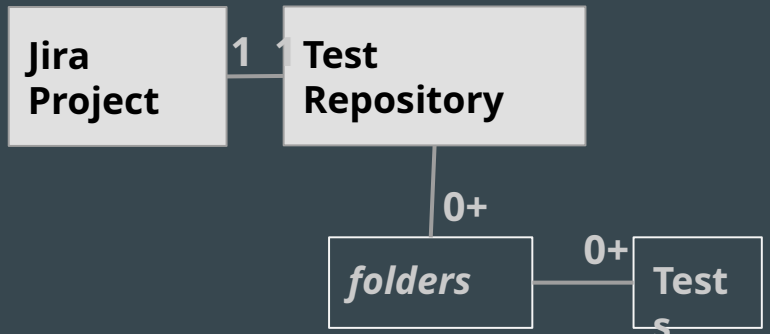
Relation between entities

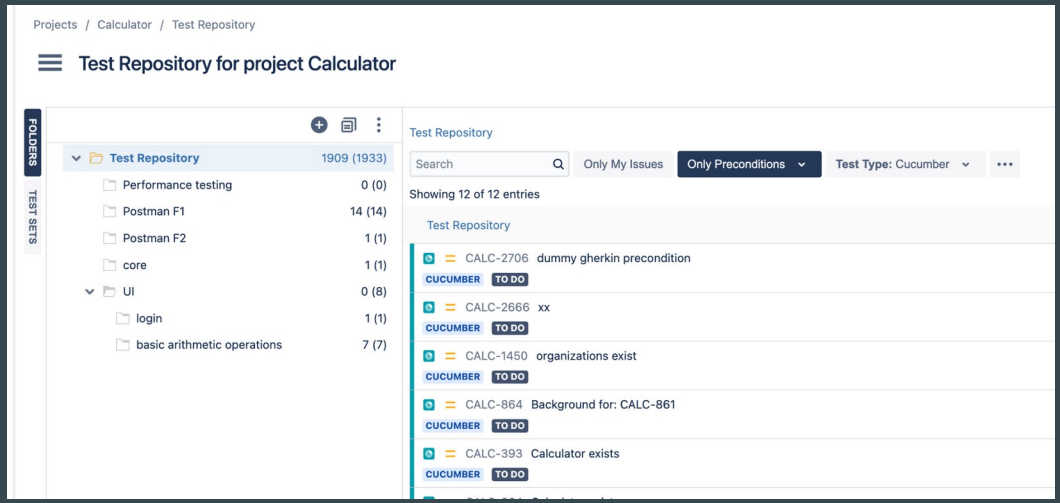
- “Requirements” are coverable by one or more Tests
- A Test Plan has a list of Tests it tracks and usually several Test Executions related to those Tests
- A Test Execution contains Test Runs, one for each Test that is on the Test Execution

Additional entities... (non issue type based)

Test Repository

- The place where we can see all Tests in a project, organized in folders
- No information about results
- Mostly useful for manual testing

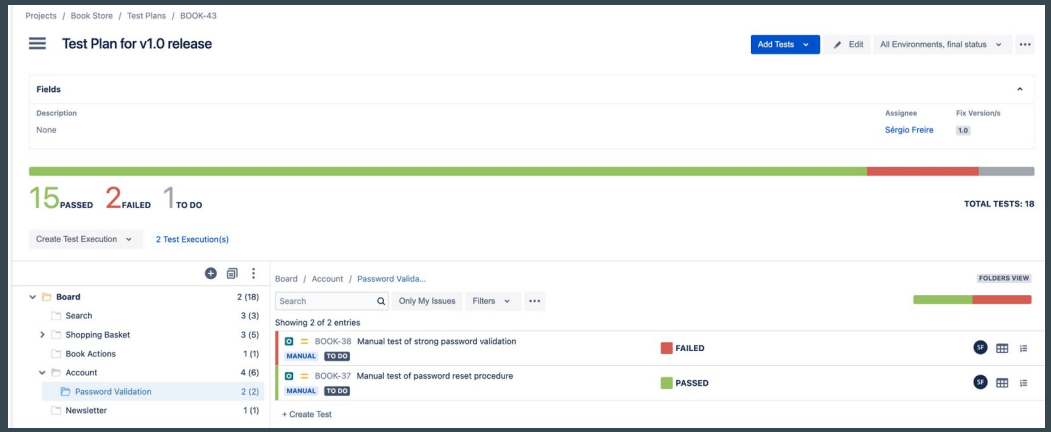


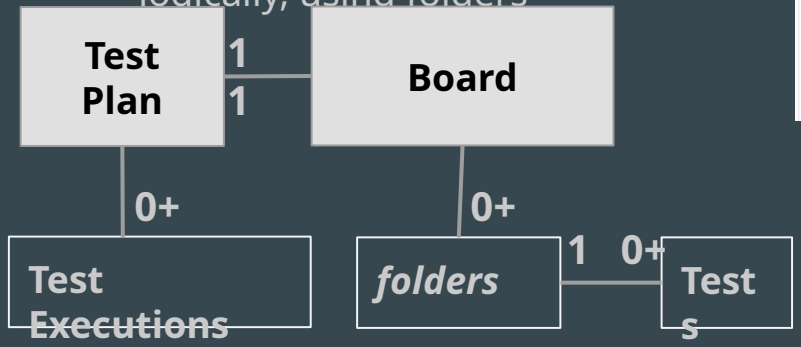


Additional entities... (non issue type based)

Test Plan' Board

- An implicit structure in each Test Plan to organize the Tests in folders, considering priorities and what makes sense from a execution perspective
- Can be useful to group the results logically, using folders





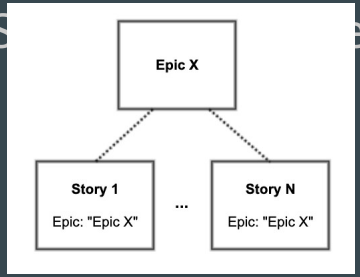
Defects (e.g., “Bug”)

A defect is something that negatively impacts quality (the value perceived to some stakeholder).

- Xray uses the “defect” word in a broader sense, i.e. some deficiency/imperfection related to the product. Usually, these are also known as bugs
- In practice, a “defect” is something we report manually during testing, or after testing whenever analyzing the test results, as a Jira issue (e.g., “Bug”)
- You can define one or more issue types to be treated as “defects”; they can include Bug or any other custom issue types you may want for that purpose

“Requirements” (e.g., Story, Epic)

- Xray uses “requirement” word in broad sense: something that the SUT must meet
- In practice, a “requirement” is any issue in Jira that can be covered (i.e., verified or validated) with Tests
- You can define the issue types to be treated as “requirements”; they can include Story, Epic, or any other custom issue type
- Xray is aware of the hierarchical relation between Epics and Stories; if you create a Test for a S ee it at the related Epic level (i.e., covering the Epic)



Coverage in Xray

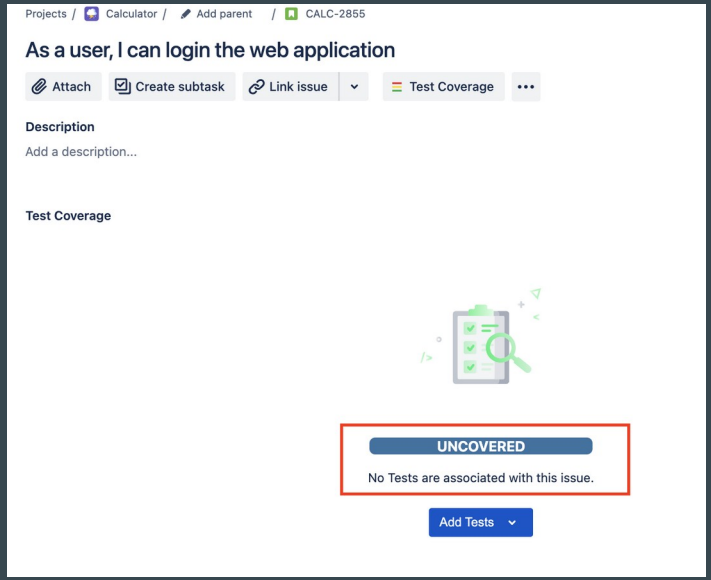
Xray provides an heuristic, called (test) coverage; sometimes people may refer to it as requirement coverage, depending on the perspective.

This is different from code coverage, which is focused on covering code.

Coverage in Xray is at higher level; it’s a way to understand if a deliverable/requirement (Story, Epic) has some tests associated and if so, if all those tests have been run and were successful.

Coverage: UNCOVERED

In Xray a “requirement” (Story, Epic) is UNCOVERED if it has no Tests linked/covering it.

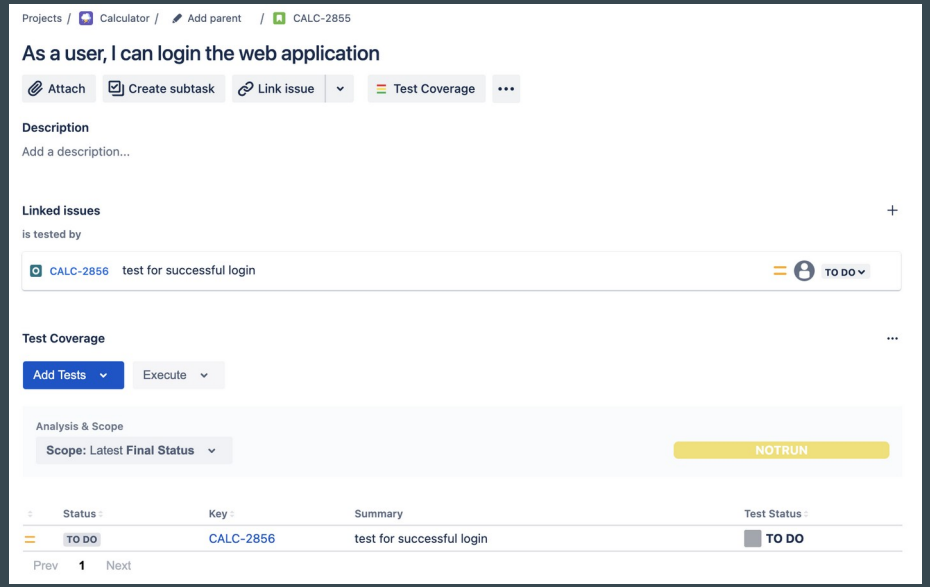


Coverage: NOTRUN

In Xray, a requirement is “NOTRUN” if:

- *any* of the linked tests need yet to be run

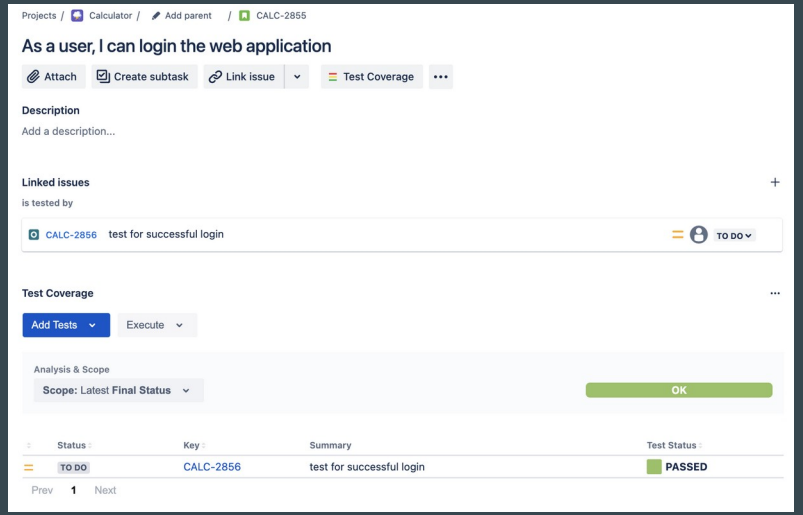
Remember: it’s an heuristic! Tests don’t cover everything and not every test (problems found during their execution) has the same importance.



Coverage: OK

In Xray, and also in some tools, a requirement is “OK” if …

- All linked tests passed, according with latest results



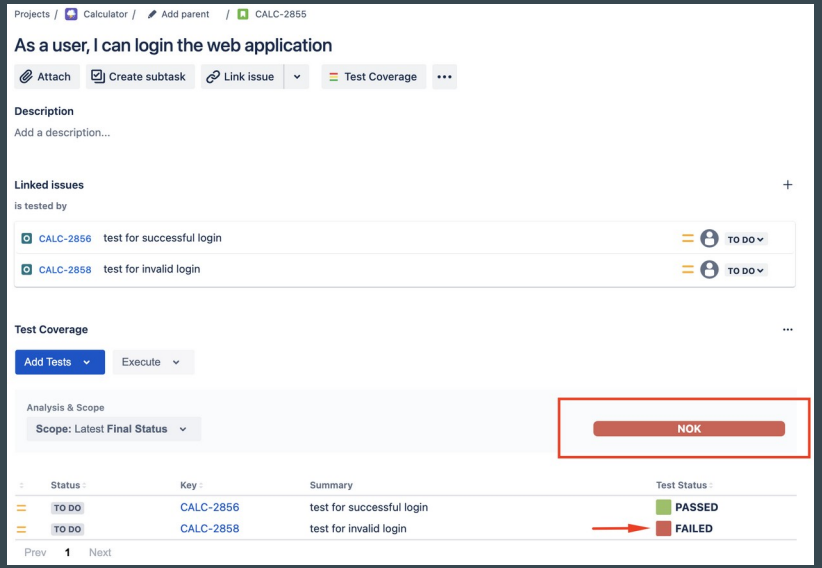
Remember: it’s an heuristic! Tests passing don’t mean we can ensure the requirement doesn’t have problems.

Coverage: NOK

In Xray, a requirement is “NOK” if …

- *any* of the linked failed, according with latest results

Remember: it’s an heuristic! Tests don’t cover everything and not every test (problems found during their execution) has the same relevance.



OTHER, OPTIONAL INFO

...

Just in case you need it...

Enabling Xray on company managed projects

Sergio Freire

Setting a company-managed project in a nutshell

Create or use an existing company-managed project and then:

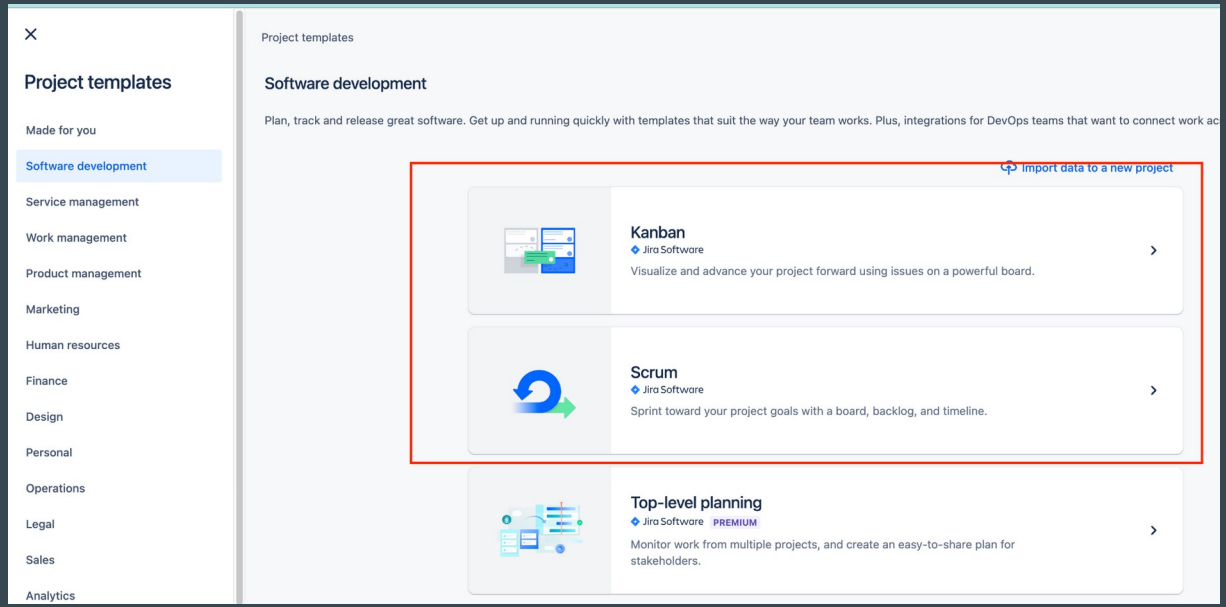
1. Add issue types to the project and map them to the Xray concepts
2. Define which items you consider to be “requirements” like
3. Define which items you consider to be “defects” like

That’s it.

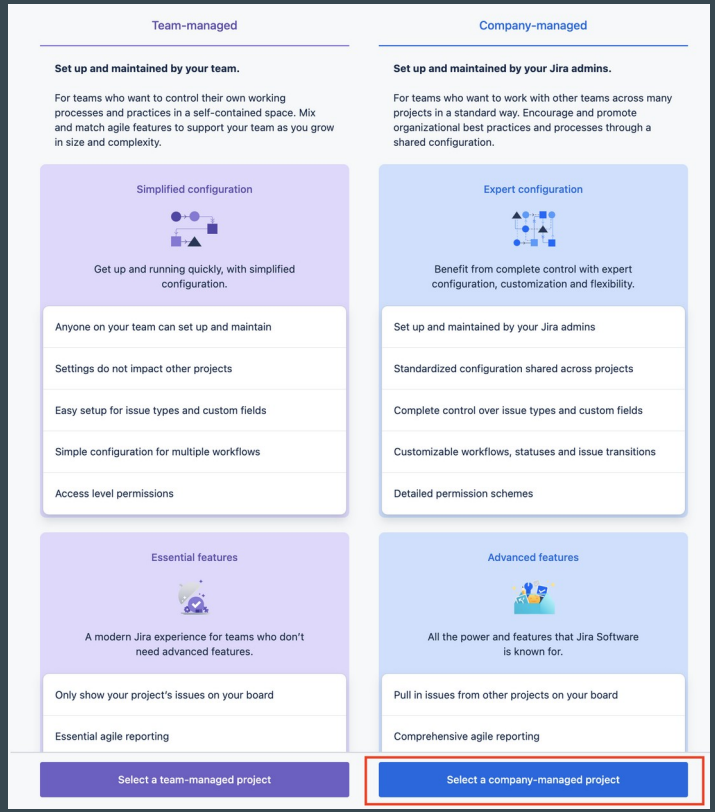
There are additional settings that we can fine tune ahead.

Create project

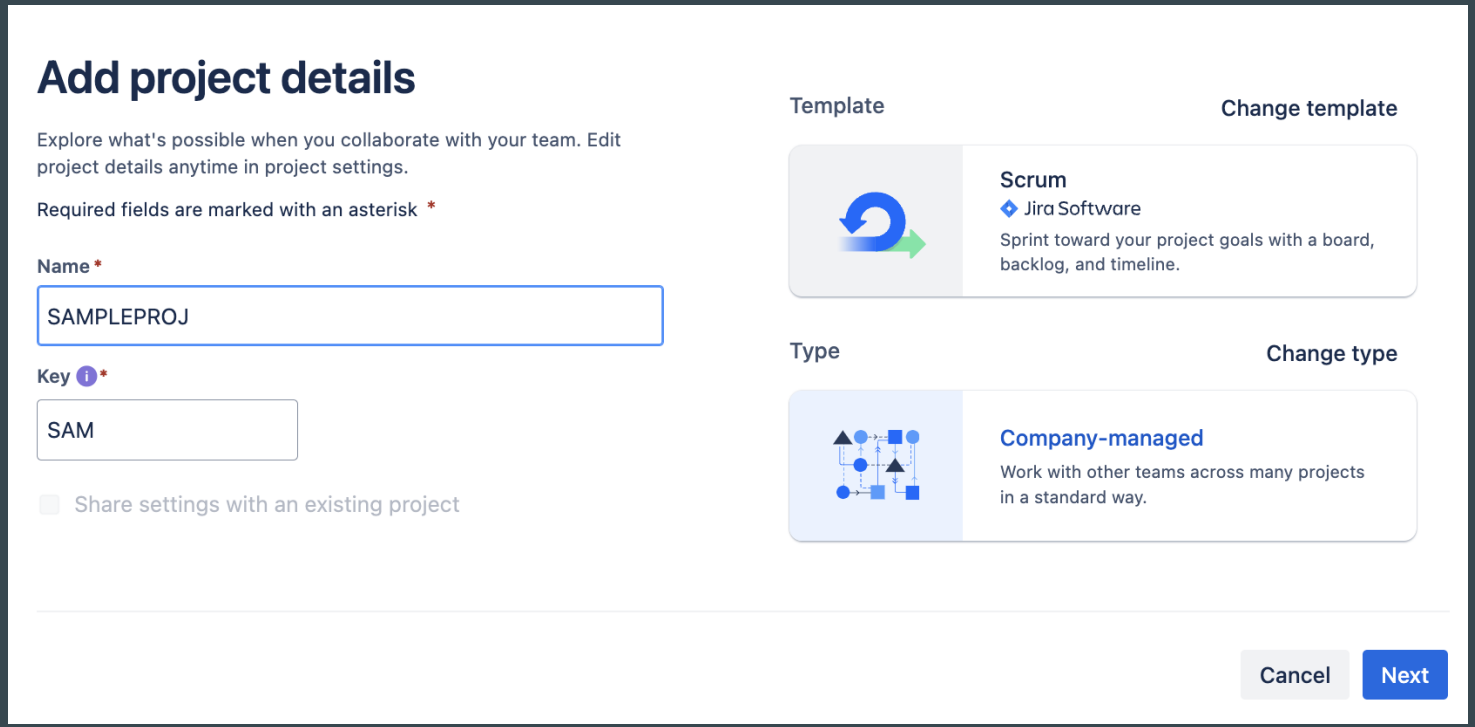
- Create a Jira project using a Kanban or Scrum template



Select type of project

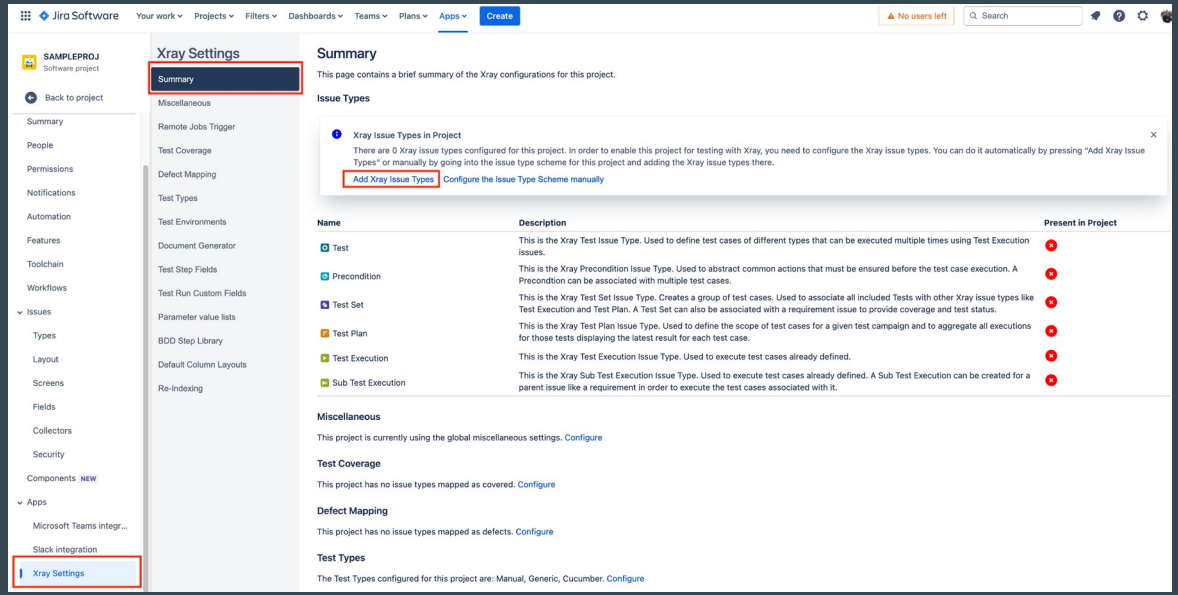


Add project details



Xray project setup

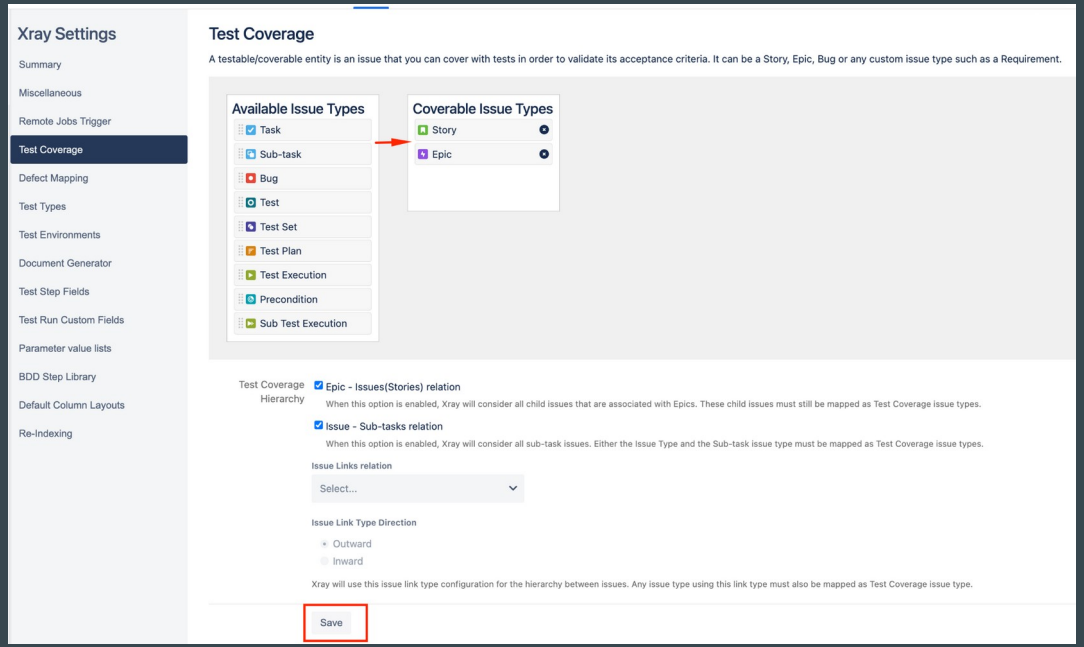
- Go to **Project settings > Apps > Xray Settings**
- There are several configurations to perform



Xray: define “requirements” / the coverage

- **Coverable Issue Types**: *Which issue types do you aim to cover with tests?*

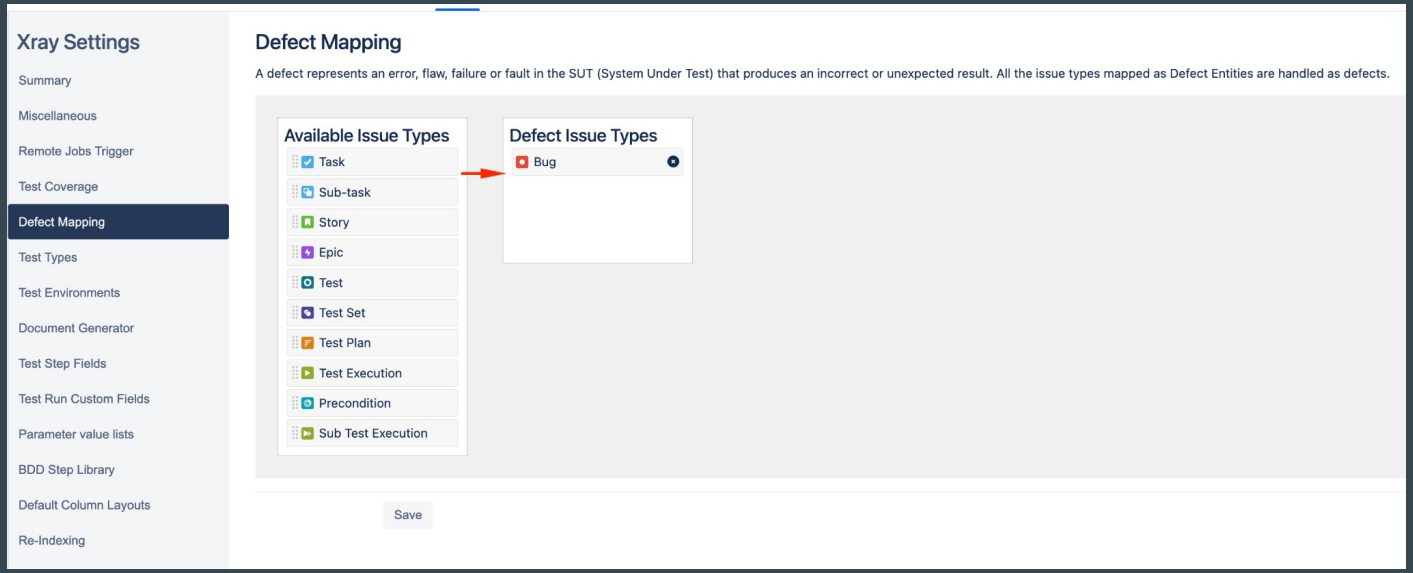
  (e.g., Story, Epic)



Xray: define defects

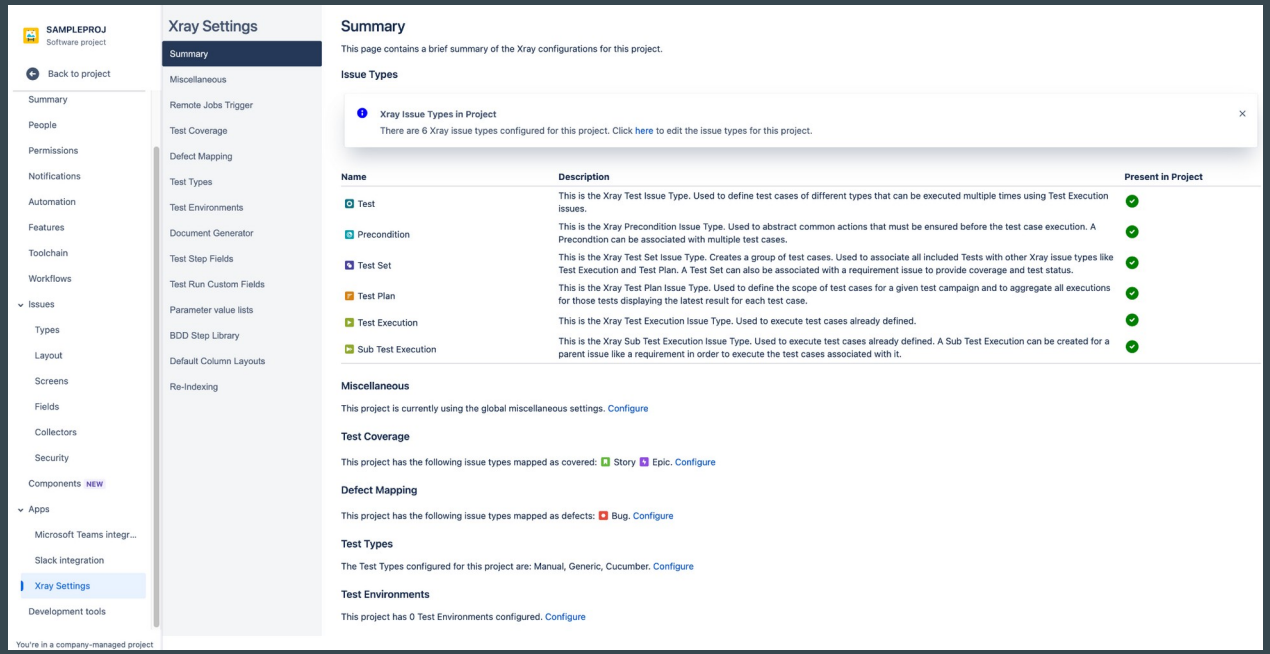
- **Defect Issue Types**: *Which issue types do you want to be handled as defects?*

  (e.g., Bug)



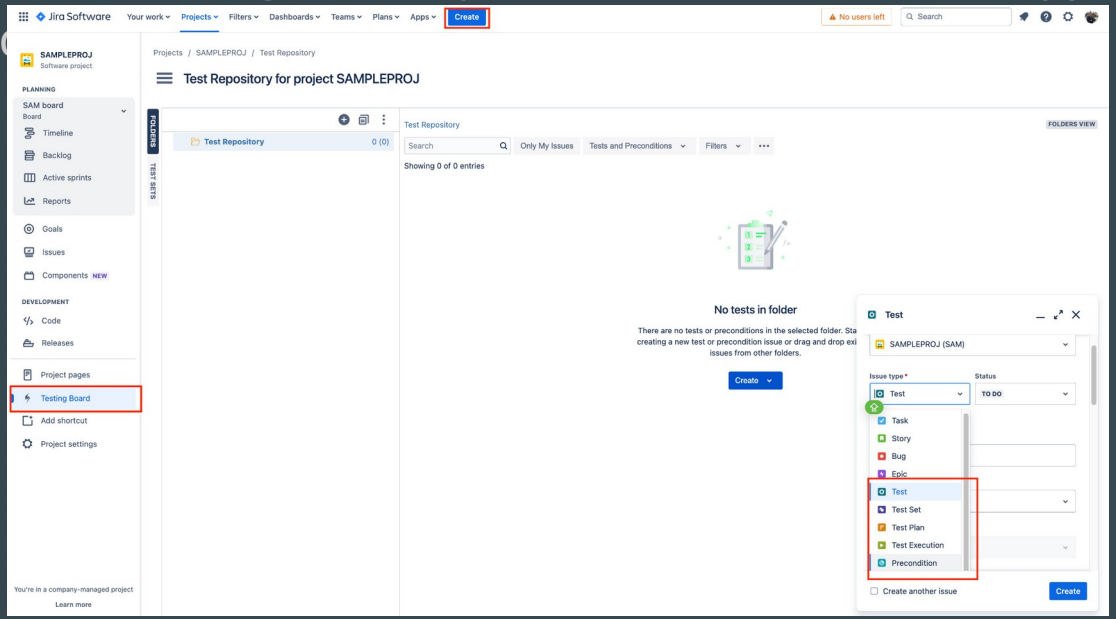
Xray: final checkup

Go to **Project settings > Apps > Xray Settings > Summary.**



If all goes well...

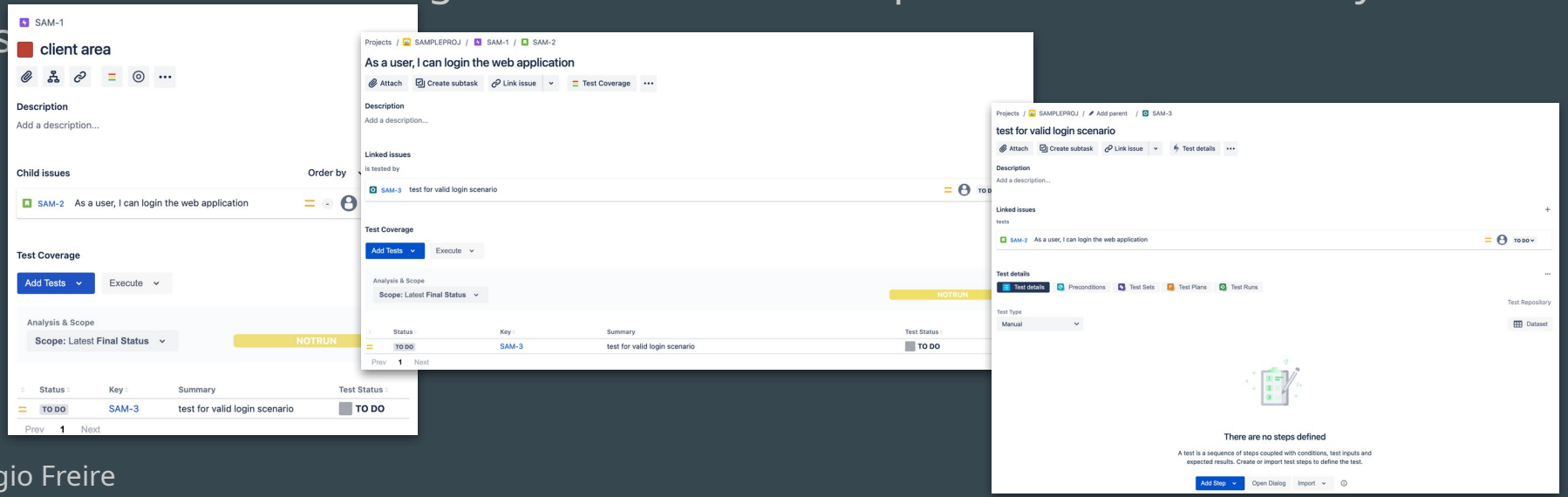
- You should see a “Testing Board” shortcut on the side menu
- Whenever creating issues, you should see the issue types that were create



DEMO / Try it out, just in case :)

1. Create Epic and a Story; associate Story to the parent Epic
2. Create a Test from the Story screen

You should see coverage information on the Epic and also on the Story issue



Sergio Freire

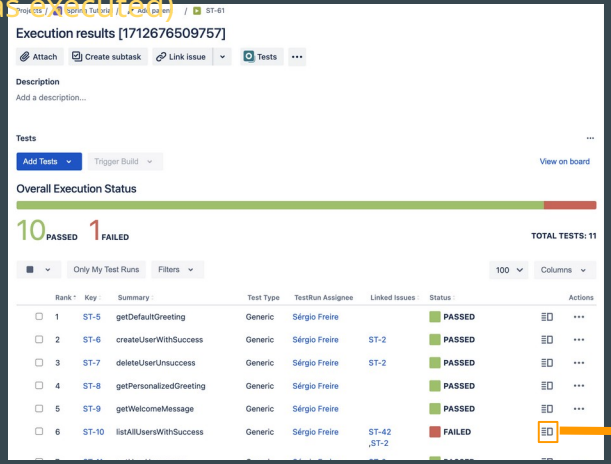
Integrating Xray with Test Automation

Sergio Freire

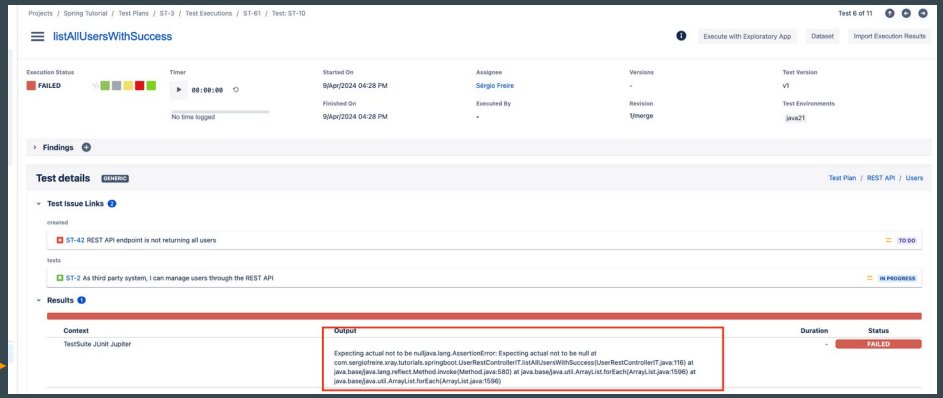
Integration with Jira and Xray

Goal 1: Track (i.e., have visibility of) test automation results in Jira, using Xray

Test Execution
(i.e., a batch of test results, composed of multiple Test Runs, one per each Test that was executed)



Details of a Test Run
(i.e., result of this Test in the scope of the Test Execution)



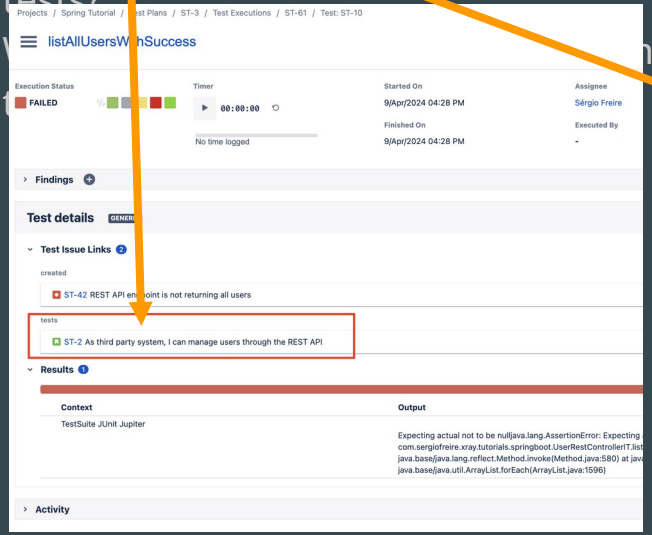
Integration with Jira and Xray

Goal 2: Track coverage related with test automation

a. What “requirements” are covered by automated tests?

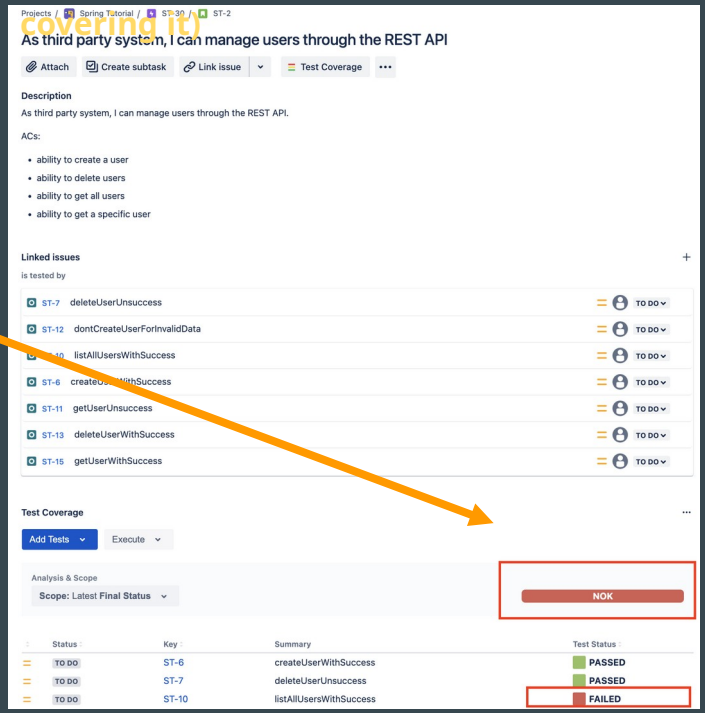
b. W… …ments” given …

Test Run details screen



Story (or Epic) issue screen

(in this case it is shown as NOK because the latest result failed for one of the Tests covering it)



What test automation frameworks/tools are supported?

You can use any language for developing your automated tests.

Xray processes the reports generated by your test runner, which is usually part of the automation framework.

Your automated scripts can be developed in any language (or tool) if you generate a supported test report (see here).

Many test runners and tools can generate reports in the ”standard” JUnit XML format, which Xray is able to process. Xray also processes a custom, enhanced JUnit XML report containing additional information tailored for it (e.g., issue key of covered Story issue, comments, evidence).

What’s the flow?

There are 2 main flows:

1. Common flow (i.e., based on JUnit XML reports)
2. Cucumber specific flows
    a. To be able to have full integration with Cucumber, it requires some additional step; however, we can follow the “common flow” to keep it simple and if having visibility of overall status of test result is enough; more info ahead

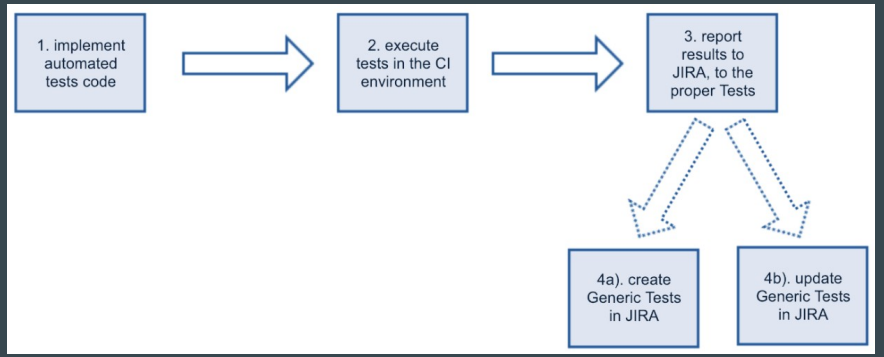
Common flow for having visibility of test automation results

1. Implement test automation code (e.g., in Java + JUnit)
    a. Use whatever IDE, language and framework; store it in the SCVS (e.g., git)
2. Run the tests, usually in CI/CD
    a. Using Maven, for example
    b. Produce a compatible test report; most test runners can output JUnit XML reports as a last resource
3. Push results to Xray
    a. From the pipeline (or even from your local machine, invoking Xray's REST API)

Common flow for JUnit, TestNG and similar test results

In Xray we don’t need to do anything beforehand; we just import the results to it.

- Tests are auto-provisioned the first time results are imported; from them onwards, Tests are reused (just new Test Runs will be created for these Tests)

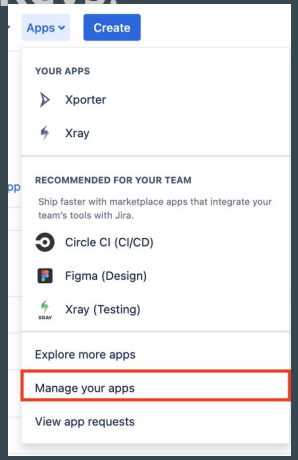


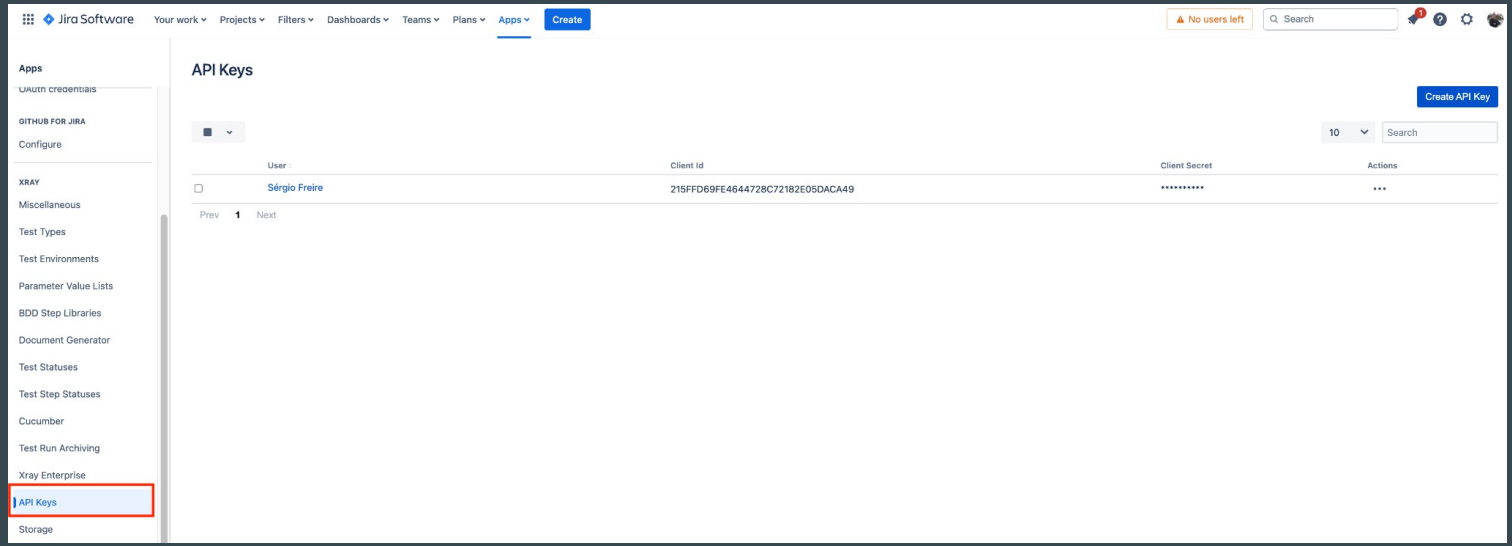
Optionally (depends on framework),

- We can link tests to existing Story issues (i.e., cover them)
- We can report results against existing Tests

Before importing results

Create an “API key” (i.e., a pair of *client id* and *client secret*) to be used on Xray API authentication requests. Go to **Apps > Manage your apps > Xray > Api Keys**.





Submitting test results

To submit test automation results to Xray/Jira we have several options:

a. Using a plugin for CI/CD tools, like the free xray-action GitHub action (see a tutorial)
b. Using a Maven plugin: xray-maven-plugin (
   example of a GH workflow in a sample project)
c. Invoking Xray's REST API endpoints for importing results directly

Submitting test results through the REST API (using “curl”)

To submit results to Xray using its REST API, we need to obtain a token, based on the client id/secret, and then use it on the request that sends the test results to Xray on Jira.

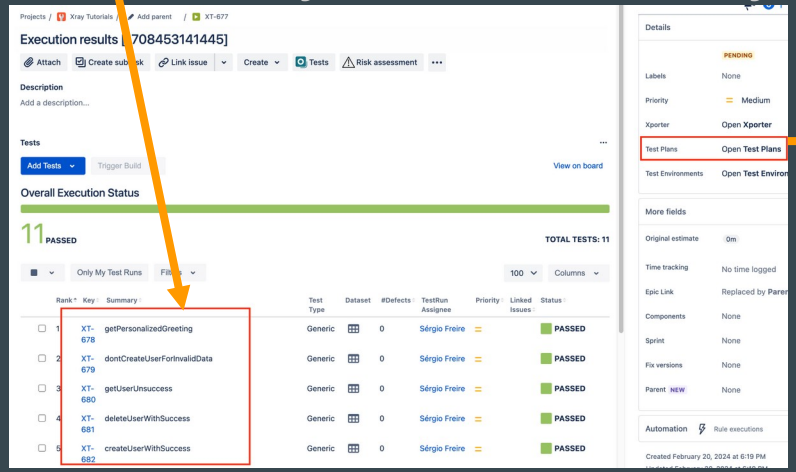
We need to specify the target Jira project, by its key. Optionally, we can also specify a Test Plan by its issue key; this Test Plan needs to exist beforehand (create it manually in Jira). Test Plan is used to group the results from multiple “builds”/iterations.

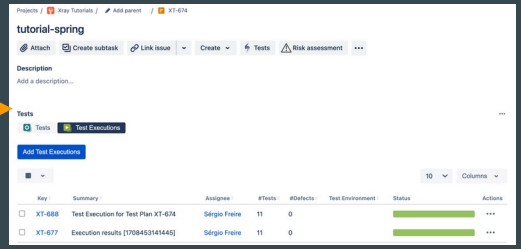
```
token=$(curl -H "Content-Type: application/json" -X POST --data '{ "client_id": "<CLIENT_ID>","client_secret": "<CLIENT_SECRET>" }'  https://xray.cloud.getxray.app/api/v2/authenticate)

curl -H "Content-Type: text/xml" -X POST -H "Authorization: Bearer $token"  --data @"reports/TEST-junit-jupiter.xml" https://xray.cloud.getxray.app/api/v2/import/execution/junit?projectKey=XPTO&testPlanKey=XPTO-123
```

Note: this way of submitting results may be mostly useful for debugging purposes.

What happens in Jira side?

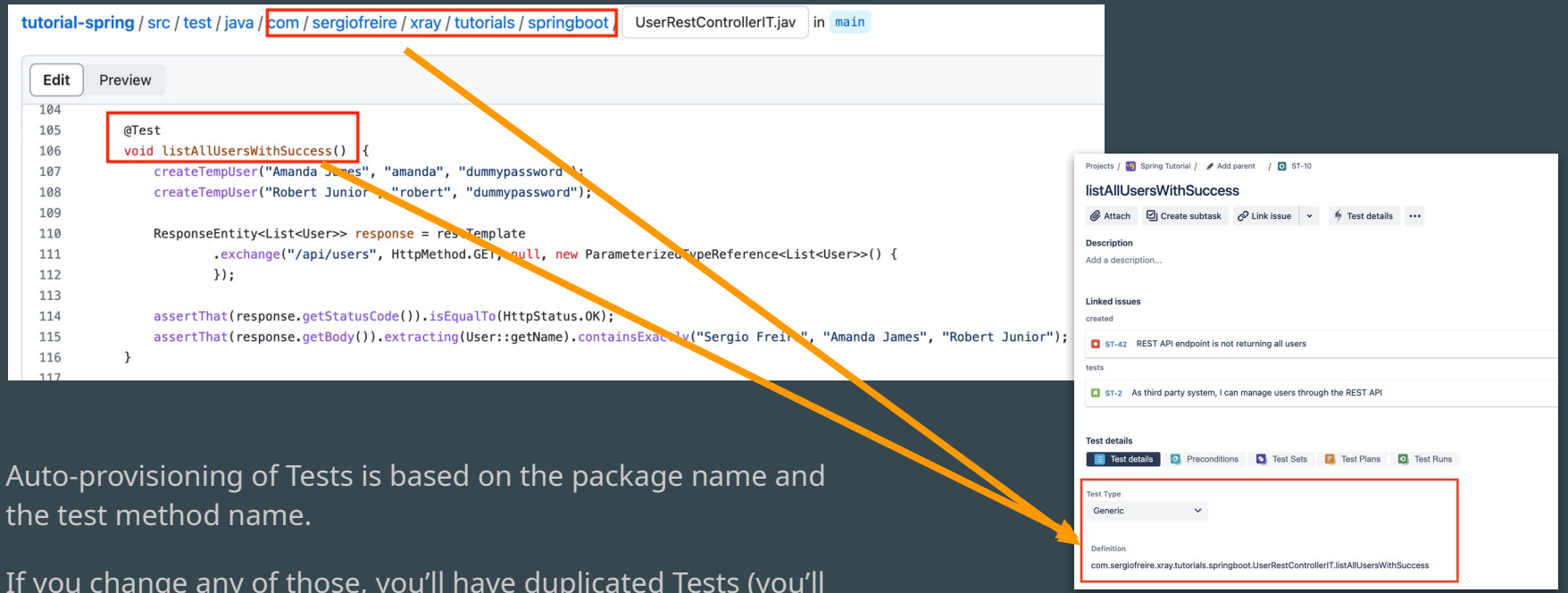
- A Test Execution issue is created on the target Jira project, containing results for the tests contained in the submitted test report file
- Test issues will be created as abstraction of the original test methods; these are only created unless they don't exist yet





If you associated the results to a Test Plan, on it you can see the all the previous results (i.e., Test Executions).

Auto-provisioning of Tests: how does it work?



Auto-provisioning of Tests is based on the package name and the test method name.

If you change any of those, you'll have duplicated Tests (you'll need to delete them in Xray).

Linking to user stories

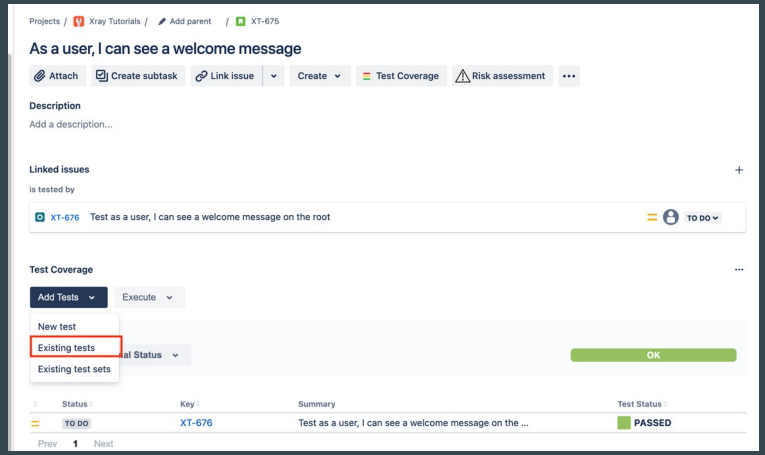
You’re writing the test automation code; how do you link it to the user stories, to let Xray know that you’re covering that user story with that test?

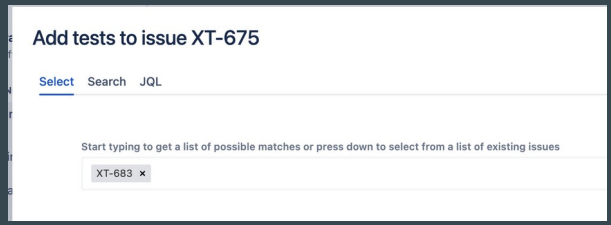
There are 2 options:

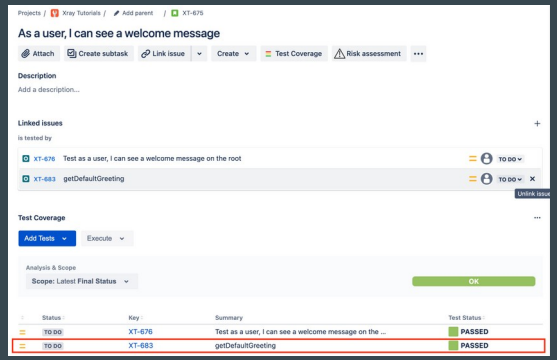
1. Use the standard JUnit XML report produced by surefire or failsafe, upload it as usual, and then link each Test to the Story manually (you’ll need to do this for each Test, just once)
2. Use a JUnit 5 extension “xray-junit-extensions”, that provides additionally features

Option 1: manually linking Tests to the user stories

After you imported results at least once, go the the Story issue, and choose **Add Tests > Existing tests**. After this, the Test will appear under the Coverage section.







Note: If we wish to remove/dissociate in terms of coverage, we need to remove the proper issue link in the “Linked Issues”.

Option 2: linking Tests to the user stories right from the test code

Use the xray-junit-extensions which provides the ability to link a test to an existing “requirement” (e.g., Story, Epic) in Jira, using the **`@Requirement`** annotation.

```
import com.getxray.xray.junit.customjunitxml.annotations.Requirement;
…

    @Test
    @Requirement("ST-2")  // issue key of related Story issue covered by this test
    void listAllUsersWithSuccess()  {
        createTempUser("Amanda James", "amanda", "dummypassword");
        createTempUser("Robert Junior", "robert", "dummypassword");

        ResponseEntity<List<User>> response = restTemplate
                .exchange("/api/users", HttpMethod.GET, null, new ParameterizedTypeReference<List<User>>() {
                });

        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(response.getBody()).extracting(User::getName).containsExactly("Sergio Freire", "Amanda James", "Robert
Junior");
    }
```

Ultimately, this Maven plugin will generate an enhanced JUnit XML like report (alternative to the the surefire/failsafe XML reports) with additional information that Xray can take advantage of. It doesn’t communicate with Jira or Xray at all!

Option 2: setting up the “xray-junit-extensions”

Add the following dependency to your pom.xml file.

```
<dependencies>
      <dependency>
            <groupId>app.getxray</groupId>
            <artifactId>xray-junit-extensions</artifactId>
            <version>0.8.0</version>
            <scope>test</scope>
      </dependency>

…
      <dependency>
            <groupId>app.getxray</groupId>
            <artifactId>xray-maven-plugin</artifactId>
            <version>0.7.5</version>
      </dependency>
</dependencies>
```

Add the optional *xray-maven-plugin* dependency, if you want to push the results to Xray directly from Maven using a specific task.

Option 2: setting up the “xray-junit-extensions”

In order to generate the enhanced, customized JUnit XML report we need to register a specific Test Execution Listener: `EnhancedLegacyXmlReportGeneratingListener`.

- Create a file `src/test/META-INF/services/org.junit.platform.launcher.TestExecutionListener` with the following content:

```
app.getxray.xray.junit.customjunitxml.EnhancedLegacyXmlReportGeneratingListener
```

- By default, the report will be generated under the directory target/ (./target/TEST-junit-jupiter.xml)

More info on the xray-junit-extensions GitHub project.

Tutorials

Check these resources:

- Tutorial showcasing Spring and integration with Xray
- Sample project on GitHub showcasing testing Spring apps, fully integrated with Xray (using the xray-junit-extensions to provide specific annotations and to generate an enhanced JUnit XML report with this information that Xray can process)

What about Cucumber?

Cucumber and similar frameworks are a bit different; Cucumber and other Gherkin-based frameworks have specification (based on Gherkin) and the underlying implementation (code).

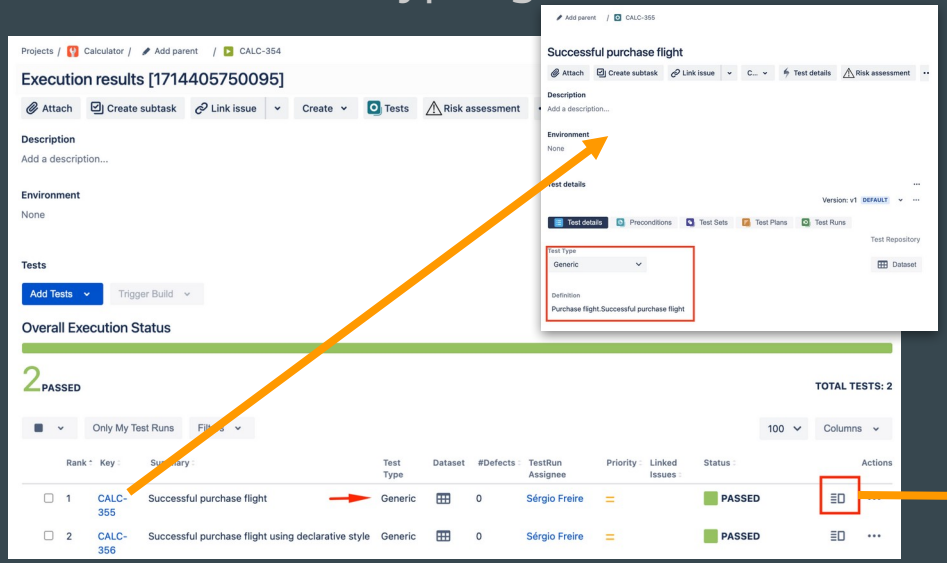
We have two options:

I. Treat them as typical automated tests (as mentioned earlier for JUnit), where we just aim to have overall visibility of the test results in Jira
   A. Generate JUnit XML test report (has less detail, but easier to implement)
      1. Xray will process it as described earlier for standard JUnit tests; in Xray we'll have overall information about the test result but we won't have Gherkin steps based information
      2. **This is simpler to implement in CI as we just need to push the JUnit XML report(s)**

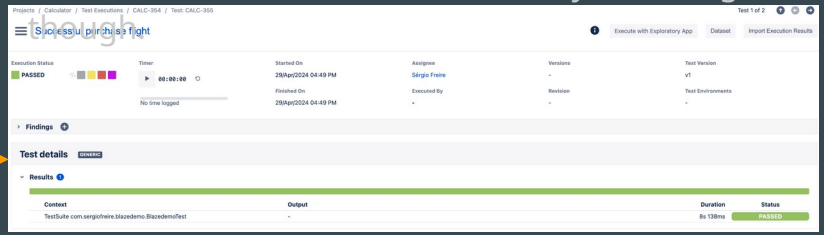
II. Handle Cucumber specifics
   A. Generate a Cucumber JSON test report (has more detail, but more work for setup)
      1. This report contains Gherkin statement level information; however, we cannot simply import the results
      2. We need to have Cucumber' Scenarios as Test issues in Xray; we decide the master for editing the Scenarios and then implement the proper flow as consequence of that (see tutorial)

1. Handling Cucumber similar to regular JUnit tests

We can upload the surefire/failsafe JUnit XML tests, as usual; Xray will create Test entities (of type “generic”), if needed, one per each Cucumber Scenario.



The Test Run (the result report for the Test) just has overall status and elapsed time; there is no Gherkin statement level information; this may be enough though



2. Handling Cucumber specifics

Scenarios (and Backgrounds) must exist in Xray, so that you can report results against them; these cannot be created automatically during the import of test results.

Where are you going to make the Gherkin specification of the Cucumber (test) Scenarios?

Specification may be managed/edited with Xray or not; use one of these approaches (not both):

A. Xray as the master for editing Scenarios/Backgrounds (as Test and Precondition issues, respectively)
B. VCS (e.g. Git) as the master to store Scenarios/Backgrounds, while the edition is done by some external IDE (e.g. Eclipse, IntelliJ IDEA, Sublime, VSCode, etc)

In sum, we need to decide (A or B) where we will edit the Cucumber related tests; that will affect the flow we need to implement.

Cucumber: generating the JSON report

To be able to show Gherkin statement/step level information in Xray, we need to generate a Cucumber JSON report.

We can specify that we want the Cucumber JSON report right from the maven CLI, or using the @ConfigurationParameter annotation on the test class.

```
 mvn test -Dtest=pt.ua.sergiofreire.blazedemo.BlazedemoTest
-Dcucumber.plugin="json:target/report.json" …
```

```
import static io.cucumber.junit.platform.engine.Constants.GLUE_PROPERTY_NAME;

import org.junit.platform.suite.api.ConfigurationParameter;
import org.junit.platform.suite.api.IncludeEngines;
import org.junit.platform.suite.api.SelectClasspathResource;
import org.junit.platform.suite.api.Suite;

@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("pt/ua/sergiofreire/blazedemo")
@ConfigurationParameter(key = GLUE_PROPERTY_NAME, value = "pt.ua.sergiofreire.blazedemo")
@ConfigurationParameter(key = Constants.PLUGIN_PROPERTY_NAME,value = "pretty, json:target/report.json")
public class BlazedemoTest {
}
```

Warning: Cucumber with JUnit 4

If you see code similar to this, then it's for JUnit 4 and not JUnit 5.

```
package webdemo;

import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(plugin = {"pretty"})
public class RunCucumberTest {

}
```

Cucumber related tutorials

Check these:

- Tutorial detailing the 2 possible flows related with Cucumber and Xray integration
- GH repo using Cucumber + Selenium, supporting the 2 flows

Reporting

Sergio Freire

The different ways of reporting

Xray provides several ways of reporting:

- Built-in, tailored reports (more info) => mostly for your awareness
  - Test Coverage and Traceability Reports are 2 of the most important ones, but there's more

- Gadgets to be used on standard Jira dashboards (more info)
  - The way to share information in and between teams, in near real-time

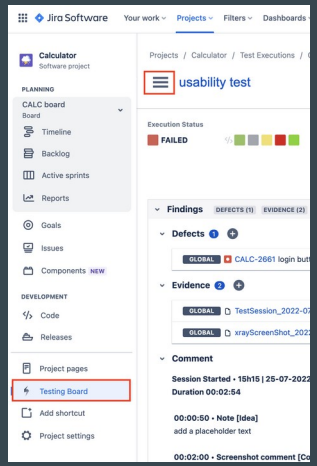
Some built-in reports are also available as gadgets but not all of them. Teams usually use a mix of approaches, depending on their needs.

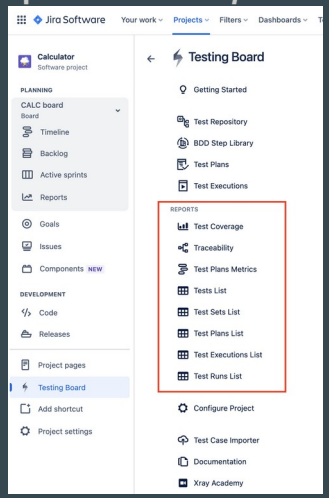
Built-in Reports
(a few examples)

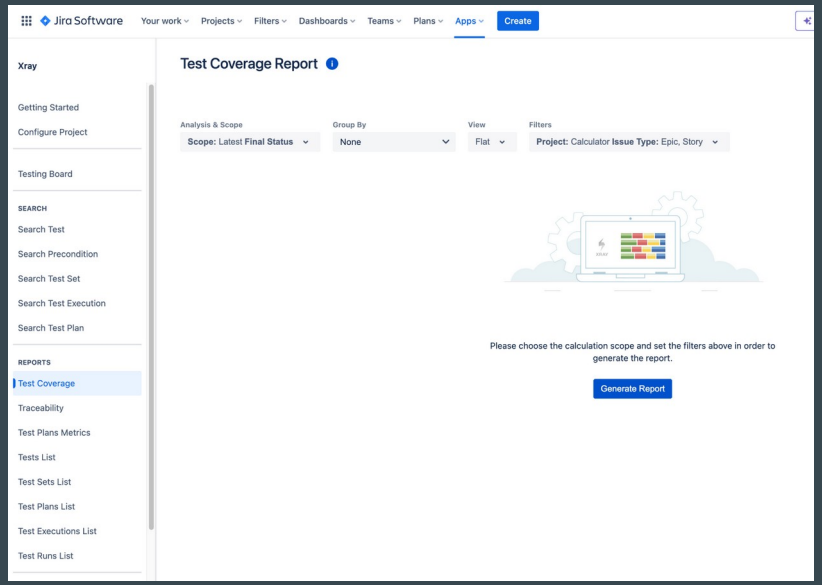
Accessing built-in reports

Built-in reports can be accessed from several places.

- **Testing Board > (hamburger menu) > *report***
- Apps > Xray > Reports > *report*



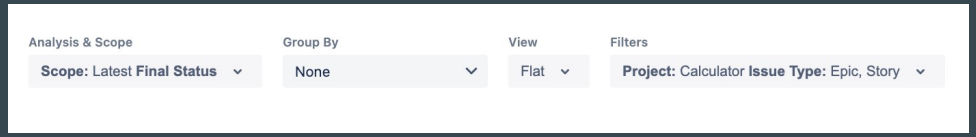




Typical actions/inputs on built-in reports

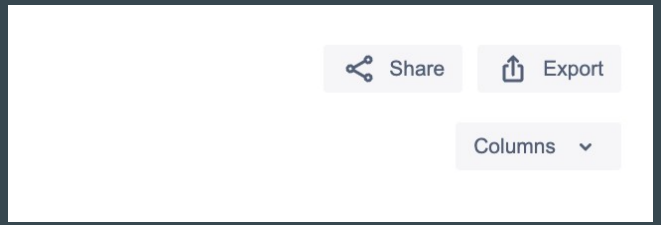
Left side

- Analysis & Scope
  - How to analyze the given data
- Filters
  - What data to use as source data
- Group by
- View



Right side

- Export (to CSV)
- Share
  - Create a unique URL that can be shared
- Columns / Show
  - show/hide some information



Built-in report: Traceability

Provides traceability from “requirements” up to defects.

Traceability works in the forward way:

Parent requirement > sub requirement(s) > Test(s) > Test Run(s) > Defect(s)

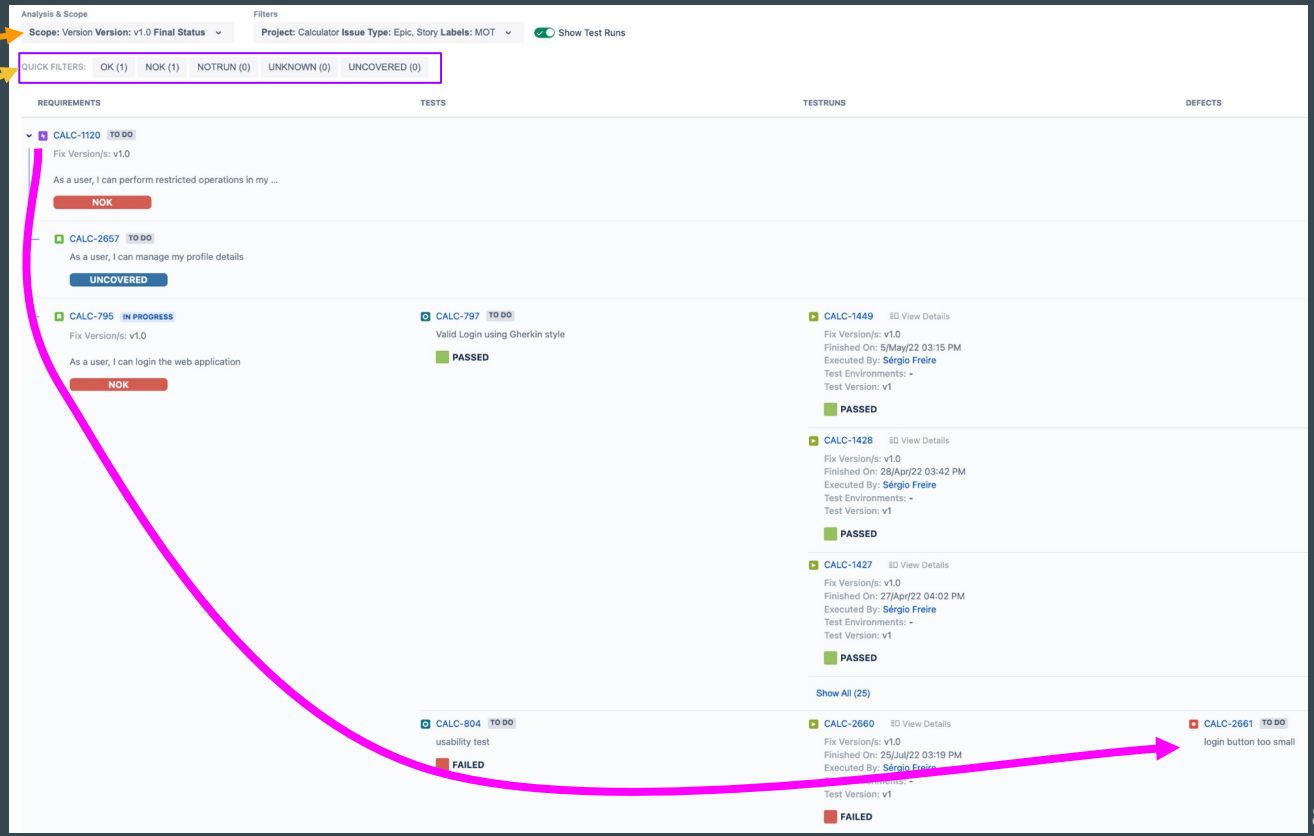
Epic > Stories > Tests > Test Runs > Bugs

Built-in report: Traceability

It's dynamic: depends on scope (i.e., the version and environment we want to analyze it on)

Quick filters for requirements based on their coverage status on the selected "scope" (e.g., version and/or environment; or even just using a Test Plan)

- *Show me just the requirements having problems…*
- *Show me just the uncovered issues…*



Built-in report: Test Coverage

Provides a birds-eye overview of the (coverage) status of your deliverables (e.g., stories, epics) based on their latest testing results.

It answers this simple question: **What are the impacts of my testing?**

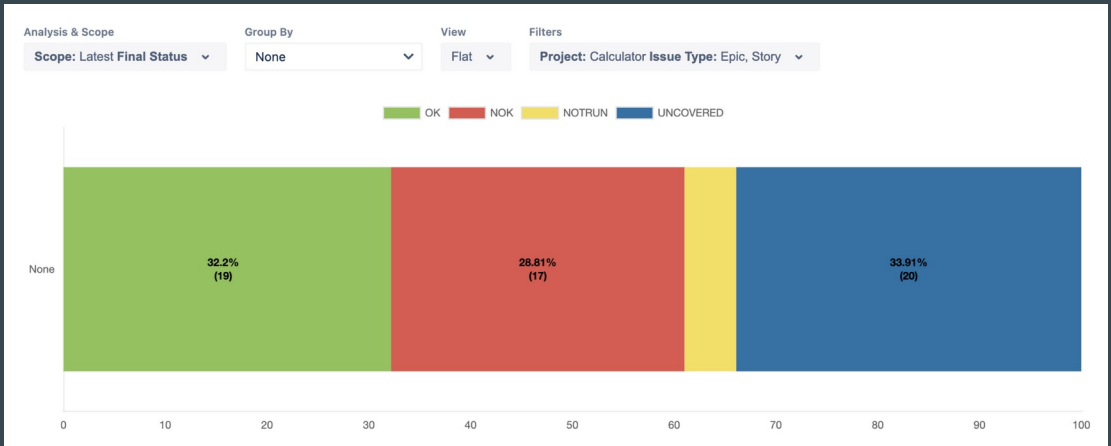
But also,

- *How are my stories on version 2.0 of the SUT?*
- *Are my highest priority stories covered by tests?*
- *How are the requirements related to some component?*
- *Which is the test completeness of certain requirements?*
- *...*

Built-in report: Test Coverage

It’s dynamic: depends on scope (i.e., the version and environment we want to analyze it on), considering the latest results obtained on it

- Output will most likely be different on two different versions



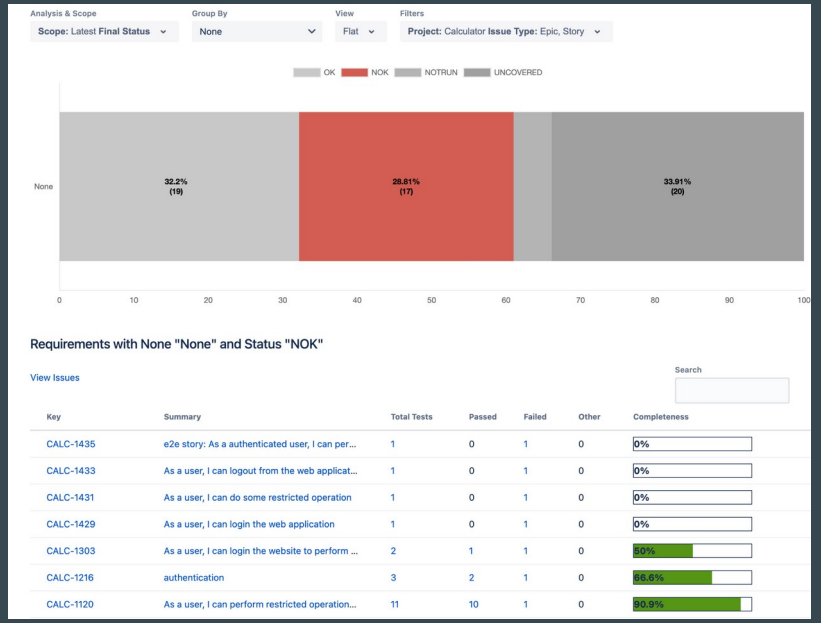
Status meaning:

- OK: covered, and all related tests are passing
- NOK: covered, but one of the related tests is failing
- NOTRUN: covered, but some of the related tests are yet to be run
- UNCOVERED: has not tests

Built-in report: Test Coverage

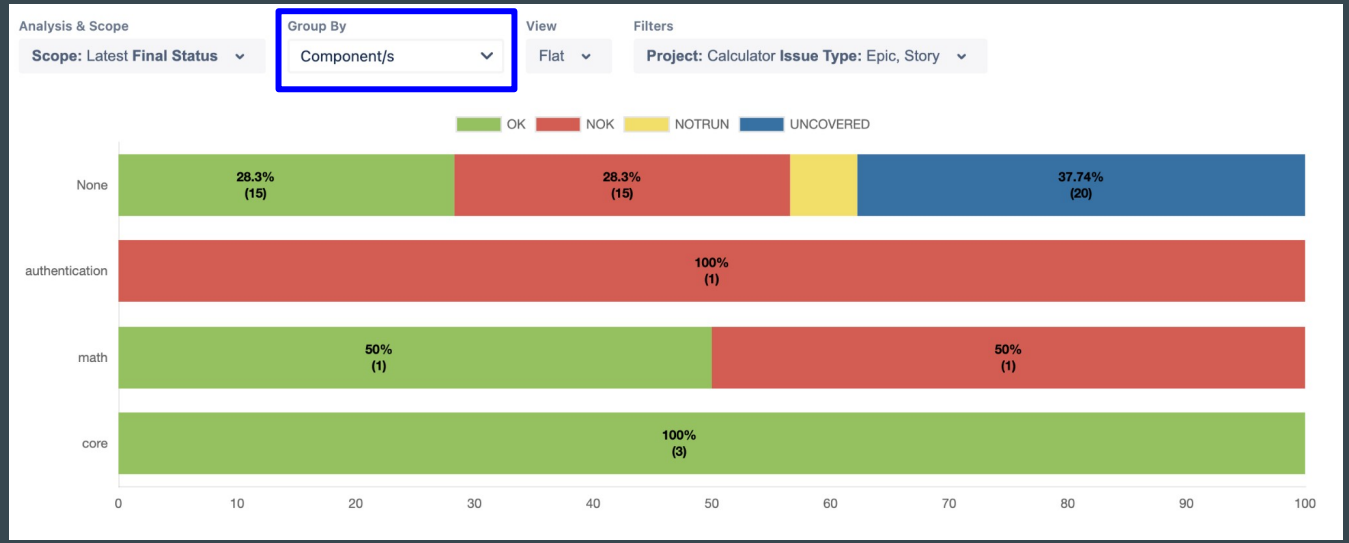
Drill-down, by clicking on a bar, to see:

- List of related “requirements”
- Test completeness of each requirement
- Total tests that cover each requirement



Built-in report: Test Coverage

Group by component, priority, or other compatible field on the “requirements”.



Jira dashboards with Xray Gadgets

Quality related information: quick considerations

What do we want to answer?

Who is going to look at it?

Is that information understandable and actionable by the team?

Quality related information

Focus on: **testing results at high-level**

- *What were the testing results, namely from test automation ran in the pipeline?*

- How to achieve it? In different ways depending on what we aim to show...
  - "Test Executions List" gadget, if we want to see each "batch" of test results (i.e., Test Execution issues), usually coming from a specific build on the pipeline
    - Here we see each build result
  - "Test Plans List" gadget, if we want to see the consolidated results from multiple "batch" of test results (i.e., from multiple builds on the pipeline)
    - If you create a Test Plan manually in Jira, and then report the results back to it, the Test Plan will show the consolidated results and on each test you can track the runs you obtained for it
    - Here we see the consolidated results from multiple builds
  - "Overall Test Results" gadget, if you want to show how are your tests, in a pie-chart, grouped by their latest result

Quality related information

Focus on: **“Requirements” / epics & user stories**

- *How are our deliverables given the latest testing results? Are they ready to be released?*

- How to achieve it?
  - “Overall Test Coverage” gadget, for an overall perspective at high-level
  - “Requirements List” gadget, for a detailed, per user story/epic, information

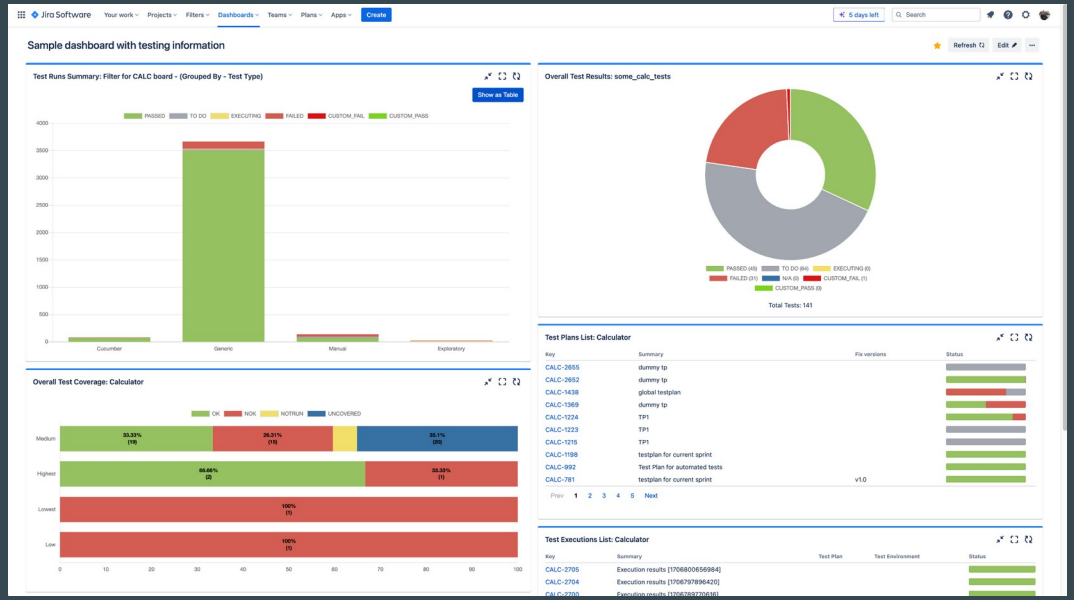
In this case we’re focused on the impacts of our testing. What does it tell about the quality of our deliverables?

Jira dashboards

The typical way to make shareadle, near real-time reports, composed of multiple gadgets. Go to **Dashboards** top menu shortcut and create a new dashboard.

In a dashboard, you can use:

- Jira default gadgets
- Xray gadgets
- Gadgets from other apps



Xray gadgets

Listings:

- Requirements List
- Tests List
- Test Sets List
- Test Executions List
- Test Plans List
- Test Runs List

Aggregated or Calculated:

- Overall Test Coverage
- Overall Test Results
- Test Runs Summary
- Tests by Test Type

Xray gadgets: how source data is specified

Depends on gadget, but source data for gadgets can include:

- Saved filter
- Project
- Issue picker

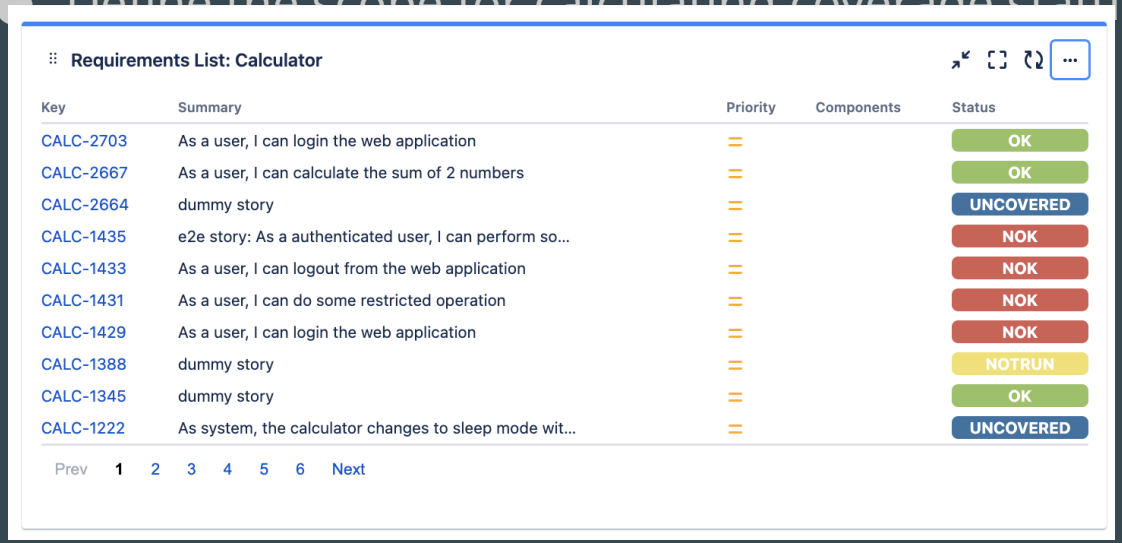
And eventually...

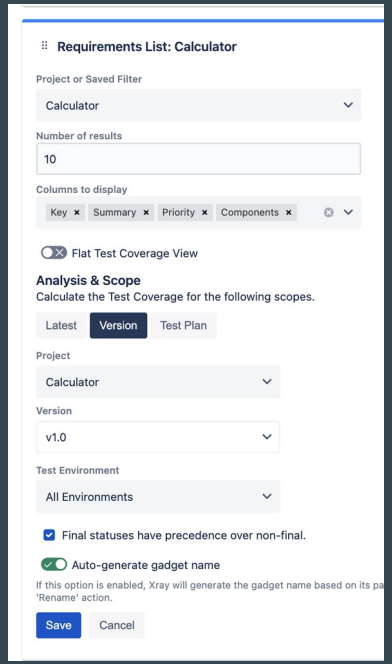
- Test Runs related info (e.g., status, executed by, dates)
- Tests related info (e.g., priority, component)
- ...

Xray gadgets: Requirements List

Show list of requirements, some attributes, and the coverage status for some scope.

- Define the scope for calculating coverage status

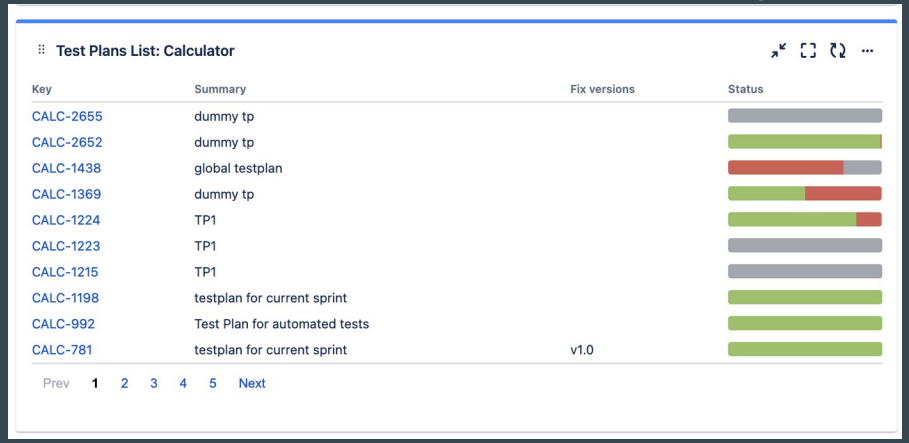


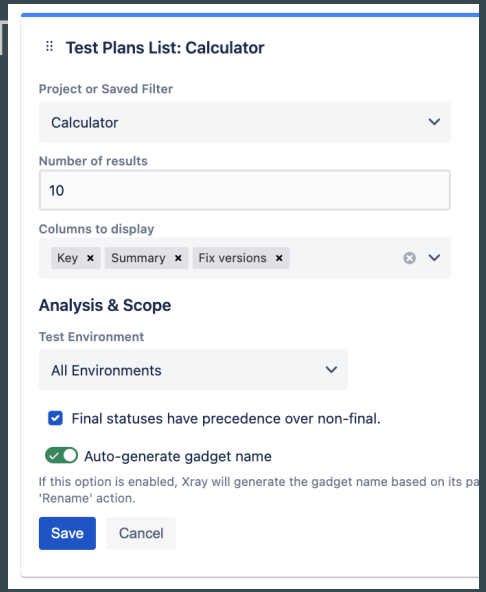


Xray gadgets: Test Plans List

Show a list of Test Executions, some attributes, and overall progress.

- Track consolidated progress considering the results on all Test Environments or just progress on a specific T
- Similar to the Test Plans List Report

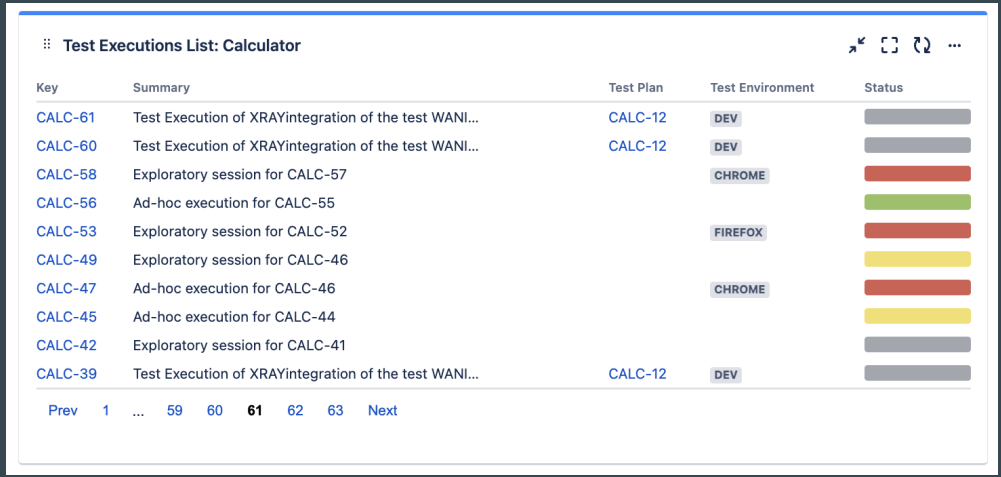




Xray gadgets: Test Executions List

Show a list of Test Executions, some attributes, and overall progress.

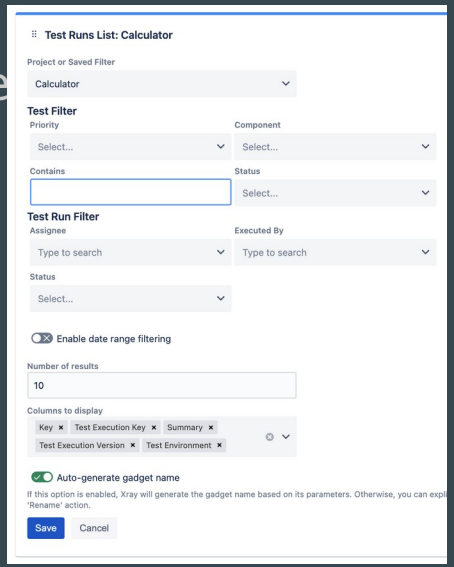
- Track in which Test Plan, Test Env., Fix Version, Revision they were performed
- Similar to



Xray gadgets: Test Runs List

Show a list of Test Executions, some attributes, and overall progress.

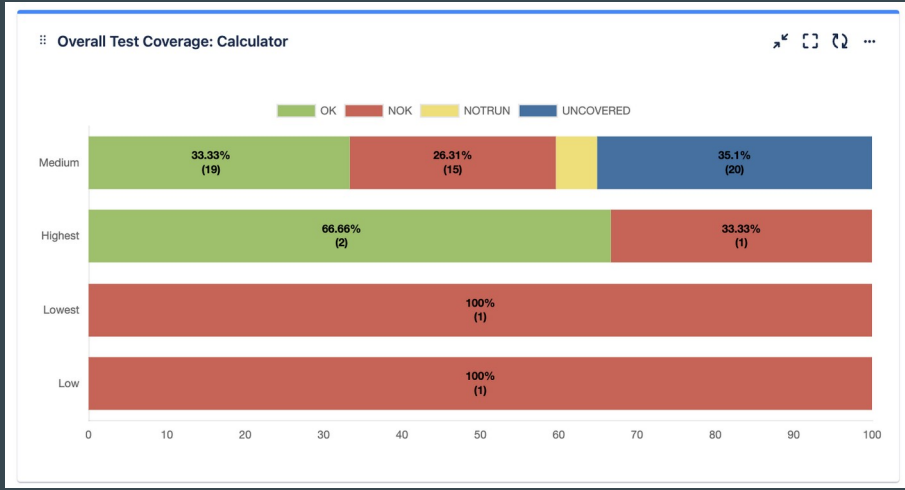
- Track in which Test Environment, Fix Version, Revision they were performed
- Tr… …uted by, e…



Xray gadgets: Overall Test Coverage

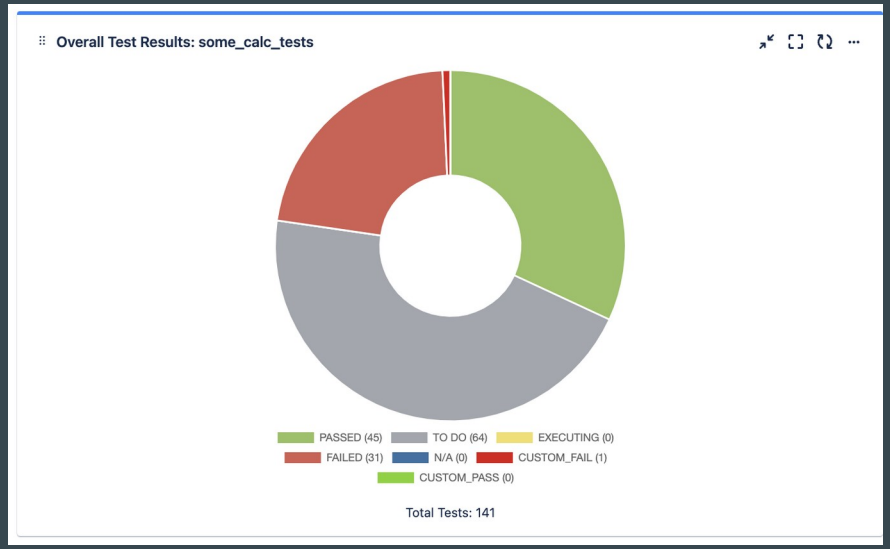
Provides a birds-eye overview of the (coverage) status of your deliverables (e.g., stories, epics) based on their latest testing results.

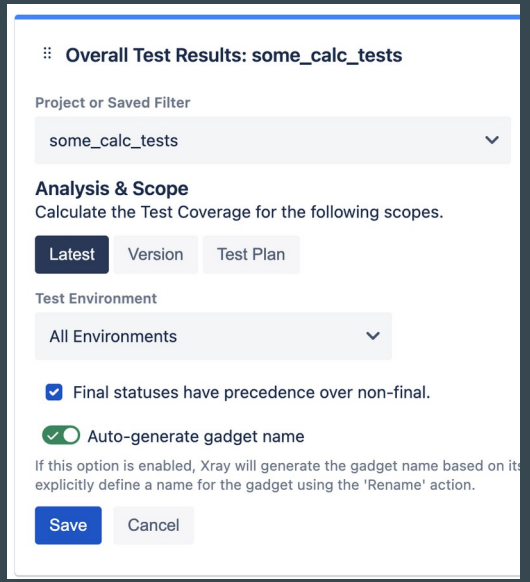
- Similar to the Test Coverage Report



Xray gadgets: Overall Test Results

Shows how the given Tests are in some “scope” (e.g., version and/or Test Env.).





Example of dashboard