Introduction

With our extensive REST API, which is free when you install Test Management for Jira, you can access testing data from Jira and carry out a variety of useful tasks, such as seamlessly migrating from another tool by importing your data, integrating with automated testing tools, and more.

By using the API, you can publish automated test-execution results to Test Management for Jira and use our powerful reporting to view metrics on your entire testing process. The API is HTTP-based and can be used from any framework, programming language, or tool.

Submitting data to Test Management for Jira through the API is easy with POST/PUT/GET/DELETE requests. For Test Management for Jira API documentation, click [here](#).

Automate Tasks

Saving time and effort by automating tasks can lead to great improvements through the development cycle, and testing is no exception. Test Management for Jira provides functionality that enables you to automate tasks that take considerable time to complete manually.

Using the REST API, you can automate tasks such as creating a test cycle from test cases that are associated to a list of Jira issues, creating folders for every sprint of all your Jira projects, or simply creating new test cases from a list of Jira issues. Many other tasks can be automated by writing your own scripts and utilizing the REST API.

Learn more about Test Management for Jira’s API by reviewing the [API documentation](#). This section contains a few useful automation script examples that use Test Management for Jira’s REST API.

Create a Test Cycle with Test Cases Associated to a List of Issues

This script first retrieves the Jira issue specified in the JQL provided, followed by any test cases associated with the issue. A test cycle is then created with the set of test cases included.

```javascript
// Create test cycles from issues

const fetch = require('node-fetch');

class TestCaseCreator {
    constructor(jiraSettings, jql, projectKey) {
        this._jiraSettings = jiraSettings;
        this._jql = jql;
        this._projectKey = projectKey;
        this._authString = 'Basic ' + Buffer.from(this._jiraSettings.user + ':' + this._jiraSettings.password).toString('base64');
    }

    async createTestCases() {
        const issues = await this._searchIssues(this._jql);
        const linkedTestCaseKeys = await this._getLinkedTestCases(issues.map(issue => issue.key));
        await this._createTestCycle('Cycle from issues', linkedTestCaseKeys, this._projectKey);
    }
}
```
Create a Folder for Each Sprint Available in the Available Board

This script first retrieves the board for your Jira project, followed by any sprints associated with the board. It then creates a test cycle folder for each of the sprints.

```javascript
async function run() {
  // Your settings
  const settings = {
    'url': '<<your-jira-url>>',
    'user': '<<your-user>>',
    'password': '<<your-password>>'
  };

  const jql = '<<Some JQL e.g. project = PROJ>>';
  const projectKey = '<<A project key e.g. PROJ>>';

  await new TestCaseCreator(settings, jql, projectKey).createTestCases();
}

run();
```

// Create test cycle folders from sprints

const fetch = require('node-fetch');

class TestCycleFolderCreator {

  constructor(jiraSettings, projectKey) {
```
this._jiraSettings = jiraSettings;
this._projectKey = projectKey;
this._authString = 'Basic ' + Buffer.from(this._jiraSettings.user + ':' + this._jiraSettings.password).toString('base64');
}

async createFolders() {
  const board = await this._getBoardForProject(this._projectKey);
  const sprints = await this._getSprintsForBoard(board);
  for(let sprint of sprints) {
    await this._createTestCycleFolder(sprint.name, this._projectKey)
  }
}

async _getBoardForProject(projectKey) {
  const url = encodeURI(this._jiraSettings.url + '/rest/agile/1.0/board?projectKeyOrId=' + projectKey);
  const response = await fetch(url, {headers: {'Authorization': this._authString}});
  if(response.status !== 200) throw 'Error retrieving boards for project: ' + projectKey;
  let searchResults = await response.json();
  return searchResults.values[0];
}

async _getSprintsForBoard(board) {
  const url = encodeURI(this._jiraSettings.url + '/rest/agile/1.0/board/' + board.id + '/sprint');
  const response = await fetch(url, {headers: {'Authorization': this._authString}});
  if(response.status !== 200) throw 'Error retrieving sprints for board: ' + board.name;
  let searchResults = await response.json();
  return searchResults.values;
}

async _createTestCycleFolder(name, projectKey) {
  const reqHeadersObj = this._buildRequest({'
    'projectKey': projectKey,
    'name': name,
    'type': 'TEST_RUN'
  });
  const url = encodeURI(this._jiraSettings.url + '/rest/atm/1.0/folder');
  const response = await fetch(url, reqHeadersObj);
  if(response.status !== 201) throw 'Error creating test cycle folder: ' + name;
  console.log('Created test cycle folder: ' + name);
}

_buildRequest(body) {
  return {
    method: 'POST',
    body: JSON.stringify(body),
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json',
      'Authorization': this._authString
    }
  };
}

async function run() {
  // Your settings
  const settings = {
    'url': '<<your-jira-url>>',
    'user': '<<your-user>>',
    'password': '<<your-password>>'
  };
  const projectKey = '<<A project key e.g. PROJ>>';
  await new TestCycleFolderCreator(settings, projectKey).createFolders();
}

run();
Create New Test Cases from a List of Issues

This script first retrieves the Jira issues specified in the JQL provided, and then it creates a test case for each of these issues, using the Summary field values as the test case names.
const fetch = require('node-fetch');

class TestCaseCreator {
    constructor(jiraSettings, jql, projectKey) {
        this._jiraSettings = jiraSettings;
        this._jql = jql;
        this._projectKey = projectKey;
        this._authString = 'Basic ' + Buffer.from(this._jiraSettings.user + ':' + this._jiraSettings.password).toString('base64');
    }

    async createTestCases() {
        const issues = await this._searchIssues(this._jql);
        for (let issue of issues) {
            await this._createTestCase(issue.fields.summary, issue.key, this._projectKey);
        }
    }

    async _searchIssues(jql) {
        const url = encodeURI(this._jiraSettings.url + '/rest/api/2/search?jql=' + jql);
        const response = await fetch(url, {headers: {'Authorization': this._authString}});
        if (response.status !== 200) throw 'Error searching for issues: ' + jql;
        let searchResults = await response.json();
        return searchResults.issues;
    }

    async _createTestCase(name, issueLink, projectKey) {
        const request = this._buildRequest({
            name: name,
            projectKey: projectKey,
            issueLinks: [issueLink]
        });
        const url = encodeURI(this._jiraSettings.url + '/rest/atm/1.0/testcase');
        const response = await fetch(url, request);
        if (response.status !== 201) throw 'Error creating test case: ' + name;
        const jsonResponse = await response.json();
        console.log('Test case created: ' + jsonResponse.key + ' - ' + name);
    }

    _buildRequest(body) {
        return {
            method: 'POST',
            body: JSON.stringify(body),
            headers: {
                'Accept': 'application/json',
                'Content-Type': 'application/json',
                'Authorization': this._authString
            }
        };
    }
}

async function run() {
    // Your settings
    const settings = {
        'url': '<<your-jira-url>>',
        'user': '<<your-user>>',
        'password': '<<your-password>>'
    };

    const jql = '<<Some JQL e.g. project = PROJ>>';
    const projectKey = '<<A project key e.g. PROJ>>';
    await new TestCaseCreator(settings, jql, projectKey).createTestCases();
}

run();
Migration

When migrating complex sets of data from a different test management tool or Excel document, Test Management for Jira’s self-service import options may not meet your specific needs. In this event, we have a REST API with an extensive list of endpoints for tasks such as creating new test results and retrieving test cases. This allows you to seamlessly import your data into Test Management for Jira by pulling data from a file source, making any relevant changes, and then using the API to push the data to Test Management for Jira.

The following sections present sample migration scripts that use Test Management for Jira’s REST API.

Migrate Test Cases with Custom Fields

This script takes a list of test cases with custom fields and creates new test cases in Test Management for Jira that include the custom fields.

---

Custom fields must be created in Test Management for Jira prior to running the migration script.

---

```javascript
// Migrate test cases with custom fields

const fetch = require('node-fetch');

class TestCaseMigrator {
  constructor(jiraSettings, testCasesToMigrate) {
    this._jiraSettings = jiraSettings;
    this._testCasesToMigrate = testCasesToMigrate;
    this._authString = 'Basic ' + Buffer.from(this._jiraSettings.user + ':' + this._jiraSettings.password).toString('base64');
  }

  async migrateTestCases() {
    for(let testCase of this._testCasesToMigrate) {
      await this._createTestCase(testCase);
    }
  }

  async _createTestCase(testCase) {
    const request = this._buildRequest(testCase);
    const url = encodeURI(this._jiraSettings.url + '/rest/atm/1.0/testcase');
    const response = await fetch(url, request);
    if(response.status !== 201) throw 'Error creating test case: ' + testCase.name;
    const jsonResponse = await response.json();
    console.log('Test case created: ' + jsonResponse.key + ' - ' + testCase.name);
  }

  _buildRequest(body) {
    return {
      method: 'POST',
      body: JSON.stringify(body),
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': this._authString
      }
    }
  }

  async function run() {
    // Your settings
    const settings = {
      'url': '<<your-jira-url>>',
      'user': '<<your-user>>',
      'password': '<<your-password>>'
    };
    const projectKey = '<<A project key e.g. PROJ>>';

    // These test cases could have been read from any source
    // and transformed to have this exact format. Custom fields
    // need to exist prior to running this script
    const testCasesToMigrate = [
```
Migrate Test Case with Attachments

This script takes a list of test cases with attachments and creates new test cases in Test Management for Jira with the attachments included.

```javascript
// Migrate test cases with attachment
const fetch = require('node-fetch');
const fs = require('fs');
const FormData = require('form-data');

class TestCaseMigrator {
  constructor(jiraSettings, testCasesToMigrate) {
    this._jiraSettings = jiraSettings;
    this._testCasesToMigrate = testCasesToMigrate;
    this._authString = 'Basic ' + Buffer.from(this._jiraSettings.user + ':' + this._jiraSettings.password).toString('base64');
  }

  async migrateTestCases() {
    for(let testCase of this._testCasesToMigrate) {
      const filePath = testCase.attachment;
      delete(testCase.attachment);
      const testCaseKey = await this._createTestCase(testCase);
      await this._uploadAttachment(testCaseKey, filePath);
    }
  }

  async _createTestCase(testCase) {
    const request = this._buildRequest(testCase);
    const url = encodeURI(this._jiraSettings.url + '/rest/atm/1.0/testcase');
    const response = await fetch(url, request);
    const jsonResponse = await response.json();
    if(response.status !== 201) throw 'Error creating test case: ' + testCase.name;
    console.log('Test case created: ' + jsonResponse.key + ' - ' + testCase.name);
    return jsonResponse.key;
  }

  async _uploadAttachment(testCaseKey, filePath) {
    const formData = new FormData();
    formData.append('file', fs.createReadStream(filePath));
    const request = {
      method: 'POST',
      body: formData,
      headers: formData.getHeaders()
    };
    // Here you would handle the request to upload the attachment
    };

  async new TestCaseMigrator(settings, testCasesToMigrate).migrateTestCases();
} run();
```
// Migrate Test-Execution Results

This script takes a list of test cases and creates a new test case in Test Management for Jira for each one, prior to creating any executions related to any test case.

// Migrate test cases with test executions

const fetch = require('node-fetch');

class TestCaseMigrator {
  constructor(jiraSettings, testCasesToMigrate) {
    this._jiraSettings = jiraSettings;
    this._testCasesToMigrate = testCasesToMigrate;
    this._authString = 'Basic ' + Buffer.from(this._jiraSettings.user + ':' + this._jiraSettings.password).toString('base64');
  }

  async migrateTestCases() {
    for (let testCase of this._testCasesToMigrate) {
      // Your settings
      const settings = {
        'url': '<<your-jira-url>>',
        'user': '<<your-user>>',
        'password': '<<your-password>>'
      };
      const projectKey = '<<A project key e.g. PROJ>>';

      // These test cases could have been read from any source and transformed to have this exact format.
      const testCasesToMigrate = [{
        'projectKey': projectKey,
        'name': 'Test case with attachment',
        'attachment': './aDocument.pdf'
      }, {
        'projectKey': projectKey,
        'name': 'Another test case with attachment',
        'attachment': './anImage.png'
      }];

      await new TestCaseMigrator(settings, testCasesToMigrate).migrateTestCases();
    }
  }
}
un();
async _createTestCase(testCase) {
    const executions = testCase.executions;
    delete(testCase.executions);
    const request = this._buildRequest(testCase);
    const url = encodeURI(this._jiraSettings.url + '/rest/atm/1.0/testcase');
    const response = await fetch(url, request);
    if(response.status !== 201) throw 'Error creating test case: ' + testCase.name;
    const jsonResponse = await response.json();
    console.log('Test case created: ' + jsonResponse.key + ' - ' + testCase.name);
    await this._createTestExecutions(testCase.projectKey, jsonResponse.key, executions);
}

async _createTestExecutions(projectKey, testCaseKey, executions) {
    for(let execution of executions) {
        execution.projectKey = projectKey;
        execution.testCaseKey = testCaseKey;
        const request = this._buildRequest(execution);
        const url = encodeURI(this._jiraSettings.url + '/rest/atm/1.0/testresult');
        const response = await fetch(url, request);
        if(response.status !== 201) throw 'Error creating test execution: ' + testCaseKey;
        console.log('Test execution created: ' + testCaseKey);
    }
}

_buildRequest(body) {
    return {
        method: 'POST',
        body: JSON.stringify(body),
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json',
            'Authorization': this._authString
        }
    };
}

async function run() {
    // Your settings
    const settings = {
        'url': '<<your-jira-url>>',
        'user': '<<your-user>>',
        'password': '<<your-password>>'
    };
    const projectKey = '<<A project key e.g. PROJ>>';

    // These test cases could have been read from any source
    // and transformed to have this exact format.
    const testCasesToMigrate = [{
        'projectKey': projectKey,
        'name': 'Test case with executions',
        'executions': [{
            'status': 'Pass',
            'environment': 'Firefox',
            'executionTime': 180000,
            'executionDate': '2018-12-13T15:22:00-0300',
        }, {
            'status': 'Fail',
            'environment': 'Chrome',
            'executionTime': 365000,
            'executionDate': '2018-12-13T18:11:00-0300',
        }]
    }, {
        'projectKey': projectKey,
        'name': 'Test case with simpler executions',
        'executions': [{
            'status': 'Pass'
        }]
    }];
Test Automation

Automated testing has become an essential part of the testing process, freeing up time for the team and reducing the rate of errors that occur in manual testing. Coding automated tests that run either locally or during builds using continuous-integration (CI) tools gives you a quick and standardized understanding of whether your code is safe to release. This is why we have invested effort to improve our integrations with CI tools such as Jenkins and Bamboo and automated testing in general.

Test Management for Jira offers a range of features that simplify the integration process. We provide a plugin for Jenkins that allows you to both pull BDD test cases from Jira and report the results back to Jira. Our full-featured REST API can be used for a more custom approach to this. We currently support Cucumber and JUnit tests.

Using these features and functionalities removes the need to create custom scripts to post data to Test Management for Jira and makes the process much more simple. The sections below offer workflow examples for the development project, CI server, and Test Management for Jira.

REST API Endpoints

Whichever tool you use for automated testing, we have a public REST API with an extensive set of endpoints that can be used to create custom scripts and enable full integration.

The following image demonstrates the flow of actions from your CI tool, which outputs the results from automated tests as part of a build. Data is pulled from a results file, manipulated if necessary, and then pushed to Test Management for Jira.

With results published back to Test Management for Jira using our REST API, you can view all your testing data in one place.

The script below provides an example of how to create a test cycle (and then test executions as part of this cycle) in Test Management for Jira, using the REST API.

```javascript
// Create a test cycle with test executions. The test execution results could have come from an automated test tool, for example.
const fetch = require('node-fetch');

class TestExecutionCreator {
  constructor(jiraSettings, executionResults, projectKey) {
    this._jiraSettings = jiraSettings;
    this._executionResults = executionResults;
    this._projectKey = projectKey;
    this._authString = 'Basic ' + Buffer.from(this._jiraSettings.user + ':' + this._jiraSettings.password).toString('base64');
  }

  async createExecutions() {
    const request = this._buildRequest({
      // Additional request configuration
    });
  }
}
```
const url = encodeURI(this._jiraSettings.url + '/rest/atm/1.0/testrun');
const response = await fetch(url, request);
if(response.status !== 201) throw 'Error creating test cycle.';
const jsonResponse = await response.json();
console.log('Test cycle created:' + jsonResponse.key);
}

_buildRequest(body) {
    return {
        method: 'POST',
        body: JSON.stringify(body),
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json',
            'Authorization': this._authString
        }
    };
}

async function run() {
    // Your settings
    const settings = {
        'url': '<<your-jira-url>>',
        'user': '<<your-user>>',
        'password': '<<your-password>>'
    };
    const projectKey = '<<A project key e.g. PROJ>>';

    // These test results could have been generated by an automated test tool
    const executionResults = [{
        'testCaseKey': '<<A test case key>>',
        'status': 'Pass',
        'environment': 'Firefox',
        'executionTime': 180000,
        'executionDate': '2018-12-13T15:22:00-0300',
    }, {
        'testCaseKey': '<<A test case key>>',
        'status': 'Fail',
        'environment': 'Chrome',
        'executionTime': 365000,
        'executionDate': '2018-12-13T18:11:00-0300',
    }, {
        'testCaseKey': '<<A test case key>>',
        'status': 'Pass'
    }, {
        'testCaseKey': '<<A test case key>>',
        'status': 'Fail'
    }];

    await new TestExecutionCreator(settings, executionResults, projectKey).createExecutions();
}

run();

**Jenkins Integration**

Jenkins is a continuous-integration (CI) tool that helps deploy software projects, run automated tests while building a project, and monitor results. If you choose to, you can install and configure an automated testing tool such as Cucumber and a CI tool such as Jenkins, to work with TM4J. You can then start using it by creating BDD-Gherkin test cases. Once the test cases have been signed off, they can be exported manually which generates a feature file parsable by Cucumber or pulled automatically from a Jenkins' build task. Automating the BDD workflow still requires the use of an external test automation tool which parses the standard BDD feature files and links it to an automated test code. You will need to implement the step definitions, which is the code that will execute according to the sequence of steps defined in the feature files. Once the tests have been executed during the CI pipeline, the TM4J API, used for example by the Jenkins plugin post-build task, can send the test results back into Jira and publish them in the test cases. Download the Jenkins plugin here: https://plugins.jenkins.io/tm4j-automation.
1. Open a test case, then click the Test Script tab.

2. Select BDD - Gherkin from the Type dropdown. The Gherkin editor appears.

3. Write your test case. Remember not to include Feature or Scenario.

4. Click Save when you are finished, and close the test case.
5. Select Export Feature Files (BDD - Gherkin) from the More to download the feature file for your new test case (or the selected test cases).

The zip file downloads to your computer. Developers using your company’s supported BDD tool can reference this information when writing automated tests related to the test case.

6. Installing the TM4J plugin for Jenkins: Navigate to the Manage Plugins page, and install the Test Management for Jira plugin on your Jenkins instance. You can also download the Jenkins plugin here: https://plugins.jenkins.io/tm4j-automation.

7. Click Manage Jenkins, and scroll to the Test Management for Jira section.

8. Enter the details for your Jira instance, test the connection, then click Add.

9. Select your Jenkins project, then click Configure.
10. Scroll to the Build section, select Test Management for Jira: Export BDD test cases from the Add Build Step dropdown, then fill out the following details and click Apply:

- **Jira Instance** - One of the preconfigured Jira instances. If you do not have any, please go to General Configuration and set up a new Jira instance.
- **Project Key** - Jira project key used to filter test cases.
- **Target Path** - The path where the exported feature files are to be placed.
11. Scroll to the Post-Build Actions section, select Test Management for Jira: Publish Test Results from the Add Post-Build Action dropdown, then fill out the following details:

- **Jira Instance** - One of the preconfigured Jira instances. If you do not have any, please go to General Configuration and set up a new Jira instance.
- **Project Key** - Jira project key used to filter test cases.
- **Test Framework** - The test framework where the test results are to be generated from.
- **File Path** - The path where the build places generated test result files. You can also use Ant-style patterns, such as `*/test-reports/*.json`.

12. Check Automatically Create Non-Existing Test Cases if you want to create a test case for a test where a test case key is not provided.
13. Click Apply, and then click Save.

**Bamboo Integration**

Bamboo is a continuous-integration, deployment, and delivery tool from Atlassian that integrates incredibly well with Jira. Test Management for Jira’s Bamboo integration enables you to automate tests outside Jira, run them in Bamboo during the CI/CD pipeline and have results sent back to TM4J for reporting and tracking.

**Prerequisite**

Windows users must download and install Curl and 7-zip and add both system paths to their machines. More information about how to update system path variables is here.

**Before You Test: Set Global Variables**

To initiate the TM4J integration with Bamboo, an admin must navigate to Bamboo’s *Global Variables* and set variable names and values for both `jira_user` and `jira_password`.

**Before You Test: Download Feature Files**
Prior to executing tests, Cucumber users with gherkin definitions in Test Management for Jira should create a script task and add the following code to enable feature files to be downloaded from TM4J. When you add the code, replace the `PROJECT_KEY` and `JIRA_URL` variables with your actual information. Optionally, you can replace the `TARGET_PATH`, `ZIP_FILE`, and `AUTO_CREATE_TEST_CASES` variables with your information, as well.

### Windows: Download Feature Files from TM4J

```
set USER=%bamboo_jira_user%
set PASSWORD=%bamboo_jira_password%
set BUILD_DIR=${bamboo.build.working.directory}
set TARGET_PATH="src/test/resources/features"
set ZIP_FILE="bddfile.zip"

set PROJECT_KEY="WEB"
set JIRA_URL="http://localhost:2990"

echo "Clean target path"
rm -rf %BUILD_DIR%/%TARGET_PATH%
mkdir -p %BUILD_DIR%/%TARGET_PATH%

echo "Downloading feature files"
curl -u %USER%:%PASSWORD% %JIRA_URL%/jira/rest/atm/1.0/automation/testcases?tql="testCase.projectKey='%PROJECT_KEY%'", --output %BUILD_DIR%/%TARGET_PATH%/%ZIP_FILE%

echo "Unzipping feature files"
7z e -aoa %BUILD_DIR%/%TARGET_PATH%/%ZIP_FILE% -d %BUILD_DIR%/%TARGET_PATH%

echo "Finished"
```

### Linux: Download Feature Files from TM4J

```
USER=$bamboo_jira_user
PASSWORD=$bamboo_jira_password
BUILD_DIR=${bamboo.build.working.directory}"
TARGET_PATH="src/test/resources/features"
ZIP_FILE="bddfile.zip"

JIRA_URL="http://localhost:2990"
PROJECT_KEY="WEB"

echo "Clean target path"
rm -rf $BUILD_DIR/$TARGET_PATH
mkdir -p $BUILD_DIR/$TARGET_PATH

echo "Downloading feature files"
curl -u $USER:$PASSWORD $JIRA_URL/jira/rest/atm/1.0/automation/testcases?tql="testCase.projectKey='$PROJECT_KEY'", --output $BUILD_DIR/$TARGET_PATH/$ZIP_FILE

echo "Unzipping feature files"
unzip $BUILD_DIR/$TARGET_PATH/$ZIP_FILE -d $BUILD_DIR/$TARGET_PATH

echo "Finished"
```

### Send Test Results from Bamboo to TM4J

To transfer your test-execution results from Bamboo back to TM4J, you need to create a **final script task** in the job where you want to report the test results.

Depending on if you use Cucumber or JUnit, add one of the following code sets, and remember to replace the `PROJECT_KEY` and `JIRA_URL` variables with your actual information. Optionally, you can replace the `PATH_FILE`, `ZIP_FILE`, and `AUTO_CREATE_TEST_CASES` variables with your information, as well.

When executed properly, the script runs after the testing concludes. Regardless of whether the tests pass or fail, the build completes, and the test results are published to TM4J.
**Windows/Cucumber: Publish Test Results to TM4J**

```bash
set USER=%BAMBOO_jira_user%
set PASSWORD=%BAMBOO_jira_password%
set BUILD_DIR=${bamboo.build.working.directory}
set FILE_PATH="target/cucumber/*.json"
set ZIP_FILE="tm4j_result.zip"
set AUTO_CREATE_TEST_CASES="true"
set PROJECT_KEY="WEB"
set JIRA_URL="http://localhost:2990"

echo "Removing old zip files"
rm -f %BUILD_DIR%/%ZIP_FILE%

echo "Creating new zip file"
7z a %BUILD_DIR%/%ZIP_FILE% %BUILD_DIR%/%FILE_PATH%

echo "Sending zip file to TM4J"
curl --basic --user %USER%:%PASSWORD%  -F "file=@%BUILD_DIR%/%ZIP_FILE%;type=application/x-zip-compressed" %JIRA_URL%/jira/rest/atm/1.0/automation/execution/cucumber/%PROJECT_KEY%?autoCreateTestCases=%AUTO_CREATE_TEST_CASES% -v

echo "Finished"
```

**Windows/JUnit: Send Results to TM4J**

```bash
set USER=%BAMBOO_jira_user%
set PASSWORD=%BAMBOO_jira_password%
set BUILD_DIR=${bamboo.build.working.directory}
set ZIP_FILE="tm4j_result.zip"
set AUTO_CREATE_TEST_CASES="true"
set PROJECT_KEY="WEB"
set JIRA_URL="http://localhost:2990"

echo "Removing old zip files"
rm -f %BUILD_DIR%/%ZIP_FILE%

echo "Creating new zip file"
7z a %BUILD_DIR%/%ZIP_FILE% %BUILD_DIR%/tm4j_result.json

echo "Sending zip file to TM4J"
curl --basic --user %USER%:%PASSWORD%  -F "file=@%BUILD_DIR%/%ZIP_FILE%;type=application/x-zip-compressed" %JIRA_URL%/jira/rest/atm/1.0/automation/execution/%PROJECT_KEY%?autoCreateTestCases=%AUTO_CREATE_TEST_CASES% -v

echo "Finished"
```
Linux/Cucumber: Publish Test Results to TM4J

```bash
USER=$bamboo_jira_user
PASSWORD=$bamboo_jira_password
BUILD_DIR=${bamboo.build.working.directory}
PATH_FILE="target/cucumber/*.json"
ZIP_FILE="tm4j_result.zip"
AUTO_CREATE_TEST_CASES="true"
JIRA_URL="http://localhost:2990"
PROJECT_KEY="WEB"

echo "Removing old zip files"
rm -f $BUILD_DIR/$ZIP_FILE

echo "Creating new zip file"
zip $BUILD_DIR/$ZIP_FILE $BUILD_DIR/$PATH_FILE -j

echo "Sending zip file to TM4J"
curl --basic --user $USER:$PASSWORD -F "file=@$BUILD_DIR/$ZIP_FILE;type=application/x-zip-compressed" $JIRA_URL/jira/rest/atm/1.0/automation/execution/cucumber/$PROJECT_KEY?autoCreateTestCases=$AUTO_CREATE_TEST_CASES -v

echo "Finished"
```

Linux/JUnit: Publish test results to TM4J

```bash
USER=$bamboo_jira_user
PASSWORD=$bamboo_jira_password
BUILD_DIR=${bamboo.build.working.directory}
ZIP_FILE="tm4j_result.zip"
AUTO_CREATE_TEST_CASES="true"
PROJECT_KEY="WEB"
JIRA_URL="http://localhost:2990"

echo "Removing old zip files"
rm -f $BUILD_DIR/$ZIP_FILE

echo "Sending zip file to TM4J"
zip $BUILD_DIR/$ZIP_FILE $BUILD_DIR/tm4j_result.json -j

echo "Sending a zip to TM4J"
curl --basic --user $USER:$PASSWORD -F "file=@$BUILD_DIR/$ZIP_FILE;type=application/x-zip-compressed" $JIRA_URL/jira/rest/atm/1.0/automation/execution/$PROJECT_KEY?autoCreateTestCases=$AUTO_CREATE_TEST_CASES -v

echo "Finished"
```

### Applying Scripts in Bamboo

This task describes how to apply scripts in Bamboo for the TM4J integration.

1. Navigate to your Bamboo project.
2. Select the plan you want to configure.
3. On the Actions menu in the upper-right corner, click Configurations.
4. On the Tasks tab, click Add Task.
5. Search for and select the Script task type.
6. Update the field values in the *Script Configuration* section, as shown. Task descriptions will vary.

For Windows:

For Linux:
7. Paste the script into the Script Body.

8. Drag the script task into the desired position.

To transfer your test-execution results from Bamboo back to TM4J, you need to drag the script into the Final Tasks section.
Adding Test Management for Jira annotations

If you are using Cucumber, setup could not be more simple. Add Test Management for Jira annotations to your Cucumber tests, and use our pre-made script to send the results to Jira. The integration is based on the JSON output files generated by Cucumber, so this output option needs to be configured properly, as follows.

1. Configure the JSON output format for your automated tests on the TestRunner class. Here we’ve added JUnit and Cucumber output formats:

```java
@RunWith(Cucumber.class)
@CucumberOptions(
    features = "src/test/resources/calculatorFeatures",
    glue="com/adaptavist/tm4j/cucumberExample/stepDefinition",
    plugin = {"junit:target/cucumber/result.xml", "json:target/cucumber/calculator.json"}
)
```

2. Annotate your automated tests with the @TestCaseKey tag. The @TestCase(key = "JQA-T1") annotation maps the test method to an existing test case in Test Management for Jira by matching its key. The @TestCase(name = "Sum Two Numbers") annotation adds a name to the test case. This maps this test method to an existing test case in Test Management for Jira by matching its name. If the test case doesn’t exist in Test Management for Jira, then a new one can be automatically created using this name when the results are uploaded.

Scenarios can be mapped to test cases in Test Management for Jira by using the test-case key or name as a scenario tag. For example:

```java
@TestCaseKey=ABC-T123.
```

To Annotate a Cucumber Test

Feature: Calculator

```
@TestCaseKey=ABC-T123
Scenario: Add a number
    Given a calculator I just turned on
    And I add 4 and 4
    Then the result is 8
```
public class CalculatorSumTest {

    @Test
    @TestCase(key = "JQA-T1")
    public void sumTwoNumbersAndPass() {
        Calculator calculator = new Calculator();
        assertEquals(5, calculator.sum(3, 2));
    }

    @Test
    @TestCase(key = "JQA-T2")
    public void sumTwoNumbersAndFail() {
        Calculator calculator = new Calculator();
        assertNotEquals(2, calculator.sum(1, 2));
    }

    public void notMappedToTestCaseAndPass() {
        Calculator calculator = new Calculator();
        assertEquals(3, calculator.sum(1, 2));
    }

    @Test
    @TestCase(name = "Mapped to a Test Case Name and Pass")
    public void mappedToATestCaseNameAndPass() {
        Calculator calculator = new Calculator();
        assertEquals(4, calculator.sum(2, 2));
    }
}

Now you can run the tests with Maven test, and the Test Management for Jira test execution results file will generate in the same execution folder.

JUnit Integration

If you use JUnit for your testing, complete the following steps.

1. Add the dependency to get access to the JUnit listener and the annotations to your pom.xml file.

   ```xml
   <dependencies>
   <dependency>
       <groupId>com.adaptavist</groupId>
       <artifactId>Adaptavist Test Management for Jira-junit-integration</artifactId>
       <version>1.0.0</version>
       <scope>test</scope>
   </dependency>
   </dependencies>
   
   <repositories>
   <repository>
       <id>adaptavist-repository</id>
       <name>Adaptavist Repository</name>
       <url>https://nexus.adaptavist.com/content/repositories/external-snapshots</url>
   </repository>
   </repositories>
   
   This dependency is not currently deployed on Maven Central Repository, so please note the custom repository added to the pom.xml file:

   <repositories>
   <repository>
       <id>adaptavist-repository</id>
       <name>Adaptavist Repository</name>
       <url>https://nexus.adaptavist.com/content/repositories/external-snapshots</url>
   </repository>
   </repositories>
   
2. Register the Test Management for Jira JUnit Listener. This listener generates the correct JSON output file needed for uploading to Test Management for Jira.
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.0</version>
      <configuration>
        <properties>
          <property>
            <name>listener</name>
            <value>com.adaptavist.Adaptavist Test Management for Jira.junit.ExecutionListener</value>
          </property>
        </properties>
      </configuration>
    </plugin>
  </plugins>
</build>

This enables you to annotate your JUnit tests with a Test Management for Jira key or name, which then generates a custom report when you run the tests.

To send these results to Jira, include a pre-made script (see the following section). Alternatively, if you use our Jenkins plugin, you can create a task to do this for you. A task and a script is also required if using other CI tools.

**Using Test Management for Jira’s Annotations with Other CI Tools/Running Locally**

If you are using Cucumber, setup could not be more simple. Add Test Management for Jira annotations to your Cucumber tests, and use our pre-made script to send the results to Jira. The integration is based on the JSON output files generated by Cucumber, so this output option needs to be configured properly, as follows.

1. Configure the JSON output format for your automated tests on the TestRunner class. Here we’ve added JUnit and Cucumber output formats:
2. Annotate your automated tests with the @TestCaseKey tag.
The @TestCase(key = "JQA-T1") annotation maps the test method to an existing test case in Test Management for Jira by matching its key. The
@TestCase(name = "Sum Two Numbers") annotation adds a name to the test case. This maps this test method to an existing test case in Test
Management for Jira by matching its name. If the test case doesn’t exist in Test Management for Jira, then a new one can be automatically created using
this name when the results are uploaded.

Scenarios can be mapped to test cases in Test Management for Jira by using the test-case key or name as a scenario tag. For example:

@TestCaseKey=ABC-T123.

To Annotate a Cucumber Test

Feature: Calculator

@TestCaseKey=ABC-T123
Scenario: Add a number
    Given a calculator I just turned on
    And I add 4 and 4
    Then the result is 8

To Annotate a JUnit Test

public class CalculatorSumTest {
    @Test
    @TestCase(key = "JQA-T1")
    public void sumTwoNumbersAndPass() {
        Calculator calculator = new Calculator();
        assertEquals(5, calculator.sum(3, 2));
    }

    @Test
    @TestCase(key = "JQA-T2")
    public void sumTwoNumbersAndFail() {
        Calculator calculator = new Calculator();
        assertNotEquals(2, calculator.sum(1, 2));
    }

    @Test
    public void notMappedToTestCaseAndPass() {
        Calculator calculator = new Calculator();
        assertEquals(3, calculator.sum(1, 2));
    }

    @Test
    @TestCase(name = "Mapped to a Test Case Name and Pass")
    public void mappedToATestCaseNameAndPass() {
        Calculator calculator = new Calculator();
        assertEquals(4, calculator.sum(2, 2));
    }
}

Now you can run the tests with Maven test, and the Test Management for Jira test execution results file will generate in the same execution folder.

Test Management for Jira Scripts for Local Testing/Other CI Tools
Add the following scripts to your project if you want to run automated tests locally.

For running automated tests with other CI tools, add this script as a task/script within the tool itself to run after the build.

```bash
#!/bin/bash
```

When using JUnit with the custom output file provided by the Test Management for Jira JUnit Listener:

```bash
zip output_results.zip tm4j_result.json
curl --basic --user user:pass -F "file=@output_results.zip"
(your-jira-base-url)/rest/atm/1.0/automation/execution/{projectKey}?autoCreateTestCases=false
```

When using Cucumber:

```bash
zip output_results.zip target/cucumber/*
curl --basic --user user:pass -F "file=@output_results.zip"
(your-jira-base-url)/rest/atm/1.0/automation/execution/cucumber/{projectKey}?autoCreateTestCases=false
```

Next Steps

- Start your free 30-day trial now.

Have a question for us?

- Please submit a support request. We’re happy to help!