1 Introduction 3
2 Requirements 5
3 Getting Started 7
  3.1 Install Loom .......................................................... 7
  3.2 Adjust build configuration ....................................... 7
  3.3 Run build .............................................................. 7
  3.4 More examples ....................................................... 8
4 Installing and Updating 9
  4.1 Automated installation ........................................... 9
  4.2 Source Control Management ..................................... 9
  4.3 Behind the scenes .................................................. 10
  4.4 Manual installation ................................................ 10
5 Configuration 11
  5.1 Conventions .......................................................... 11
  5.2 Module configuration .............................................. 11
  5.3 Environment .......................................................... 12
6 Building with Loom 15
  6.1 Plugins, tasks products and goals ................................ 15
  6.2 Examples .............................................................. 15
7 Plugin checkstyle 17
  7.1 Products ............................................................... 17
  7.2 Minimum example .................................................. 17
  7.3 More advanced example ......................................... 17
  7.4 Settings ............................................................... 18
  7.5 Notes ................................................................. 18
8 Plugin eclipse 19
  8.1 Products ............................................................... 19
  8.2 Settings ............................................................... 19
9 Plugin idea 21
  9.1 Products ............................................................... 21
Loom is a modern build tool for Java based projects. It focuses on simplicity, conventions, ease of use and performance. It uses a very simple, declarative configuration (YAML) and convention over configuration to reduce the configuration to a bare minimum.
Loom requires **JDK 9** to build and run but it can build projects targeting **Java 6-9** through JDKs cross-compile functionality.
3.1 Install Loom

Change to your project directory and enter:

```
curl -s https://loom.builders/installer.sh | sh
```

For more details, read the Installing and Updating section.

3.2 Adjust build configuration

Very basic projects (without any external dependencies) don’t need a configuration at all.

A typical configuration with a few external dependencies might look like this:

```
plugins:
  - junit
compileDependencies:
  - com.google.guava:guava:23.0
testDependencies:
  - org.junit.jupiter:junit-jupiter-engine:5.0.3
```

For more details, read the Configuration section.

3.3 Run build

To start a build using Loom, enter:
For more details, read the Building with Loom section.

### 3.4 More examples

If you want to see more examples on how projects are built using Loom, visit our GitHub loom-examples project.
4.1 Automated installation

To install the most recent version of Loom, change to your project directory and enter:

```bash
curl -s https://loom.builders/installer.sh | sh
```

*The same command is also used to update an existing installation.*

If no errors were reported, Loom is installed to your system / project.

If you want to install a specific (older) version of loom instead (e.g. 1.0.0), enter:

```bash
curl -s https://loom.builders/installer.sh | sh -s 1.0.0
```

4.2 Source Control Management

If you’re using a SCM system like Git, you probably want to add the *Loom Launcher* and the *Loom Installer* to your repository to ensure reproducible builds for your project:

```bash
git add loom loom.cmd loom-installer
```

Furthermore, you probably want to exclude some directories from being committed. Example .gitignore file:

```bash
.loom
/build/

# In case .jar files are globally ignored
!/loom-installer.jar
```
4.3 Behind the scenes

To guarantee reproducible builds while maintaining the size of files added to your project at a minimum, the Loom installation is split into two parts: a project wide one (the Loom Installer and the Loom Launcher) and a system wide one (the Loom Library).

After a successful installer.sh run, the following files were created:

- **In your project directory:**
  - `loom-installer/loom-installer.jar` The Loom Installer – a tiny program that installs Loom on systems where it is missing. It is executed by the Loom Launcher if necessary.
  - `loom-installer/loom-installer.properties` Configuration of the Loom Installer – e.g. the download URL of Loom and its version.
  - `loom` The Loom Launcher for Unix
  - `loom.cmd` The Loom Launcher for Windows

- **In your system:**
  - The Loom Library is installed to a user specific directory on your machine, that is dependent on your operating system:
    - On Unix: `~/.loom/
    - On Windows: `%LOCALAPPDATA%\Loom\Loom` (where `%LOCALAPPDATA%` is `C:\Users\{username}\AppData\Local` by default)

  You can override these operating system specific defaults by setting the `LOOM_USER_HOME` environment variable to a directory of your choice.

4.4 Manual installation

If you want to install Loom manually (discouraged), follow these steps:

- Change to your project directory
- Create a directory named `loom-installer`
- Create a file named `loom-installer/loom-installer.properties`, which points to the Loom Library URL (example: `distributionUrl=https://loom.builders/loom-1.0.0.zip`)
- Download the Loom Installer to a file named `loom-installer/loom-installer.jar`. You can download it from `https://loom.builders/loom-installer-1.0.0.jar`
- Launch the Loom Installer by entering `java -jar loom-installer/loom-installer.jar`
The build configuration for projects built with Loom is located in a file called `module.yml`. For multi-module builds, every module may have such a file.

## 5.1 Conventions

For a better understanding of how the configuration works and what can be done, it is important to first understand the conventions of Loom.

### 5.1.1 Directory layout

Loom knows two kinds of directory layouts:

**Single module** Single module projects have their sources in `src/main/java/`, their resources in `src/main/resources/`, their test sources in `src/test/java/` and their test resources in `src/test/resources/`.

**Multi module** Multi module projects have a directory `modules/` and a subdirectory per module beneath that `modules/` directory. In the module subdirectories, the structure is as with single modules (starting with a `src/` directory).

These two directory layouts are mutually exclusive and cannot be used together.

Every module might have a configuration, called `module.yml`. For single modules, this file is located in the root directory of your project. For multi modules, this file is located in each module subdirectory beneath the `modules/` directory.

## 5.2 Module configuration

A `module.yml` configuration has up to five sections.
plugins A list of plugins to activate for the module. Some plugins in Loom are active by default: eclipse, idea and java.

settings A list of global or plugin specific settings. Plugin specific settings starts with the plugin name followed by a dot as a prefix (e.g. java.mainClassName for the setting mainClassName of the java plugin). Global settings have no prefix.

moduleCompileDependencies A list of modules (of the same project) this module depends on. This is only available within multi module projects.

compileDependencies A list of compile time dependencies.

testDependencies A list of dependencies, required when running tests.

Example:

Listing 1: module.yml

```yaml
plugins:
  - checkstyle
settings:
  moduleName: builders.loom.example.app
  javaPlatformVersion: 8
  java.mainClassName: builders.loom.example.app.Main
  checkstyle.configLocation: config/checkstyle/checkstyle.xml
moduleCompileDependencies:
  - builders.loom.example.api
compileDependencies:
  - com.google.guava:guava:21.0
testDependencies:
  - org.junit.jupiter:junit-jupiter-engine:5.0.3
```

5.2.1 Global settings

moduleName By default, the module name of a module in a multi module project is the name of the module subdirectory. In a single module setup, the default name is unnamed. The recommended way to define/overwrite this, is to create a module-info.java file in each module. If you do not want to create this file, you can specify the name of the module using this setting.

javaPlatformVersion By default, Loom compiles your sources with Java 9 as the target Java version. If you want to lower the required Java version of your application, you can specify 6, 7 or 8 and Loom will use JDKs cross-compile functionality to support older Java releases. Please note, that Loom itself always requires JDK 9 to run.

5.3 Environment

Some behaviour of Loom can be controlled by environment variables. This is a comprehensive overview:

LOOM_EXIT_CONSOLE Used by the Loom Launcher for Windows (loom.cmd) to control the script termination behaviour in case of an error (default: exit /B; setting LOOM_EXIT_CONSOLE to a non empty-value causes the script to exit just with exit).

LOOM_OPTS Specify parameters (like system properties or memory switches) that are passed to the Java executable when launching Loom.

LOOM_REPOSITORY_URL Overrides the default repository URL https://repo.maven.apache.org/maven2/.
LOOM_USER_HOME  Overrides the default location where the Loom Library is installed. The default is system specific. For more details, read the Installing and Updating section.
Loom has a very simple command line interface. The usage is:

```
./loom [option...] [product|goal...]
```

Just enter `./loom -h` so see the full options reference.

### 6.1 Plugins, tasks products and goals

`./loom -p` prints all products and goals available with your current configuration.

To understand what products and goals are, you should get the full picture:

- **product**: A product is the smallest unit in the Loom build chain. It might be a file (e.g. a .jar file), a directory (e.g. compiled classes) or something else.
- **goal**: A goal is a list of dependent products and goals. It doesn’t add any functionality / products by itself.
- **task**: Every task in Loom is responsible for providing (building/producing) a product.
- **plugin**: A plugin registers one or more tasks to the Loom build chain. A plugin also defines the dependency between products and goals.

You may also want to use `loom -p dot` to generate a .dot file, containing all the dependencies between the products and goals. This file can be turned into a graphical representation (e.g. .png) by using Graphviz.

### 6.2 Examples

#### 6.2.1 Default build

`./loom build` requests the `build` goal.
6.2.2 Build just some products

./loom jar javadocJar requests the products jar and javadocJar.

6.2.3 Clean before

./loom -c build requests the build goal after removing the directories .loom and build.

6.2.4 Build without caches

./loom -n build requests the build goal without writing any caches (or reading existing ones).
The Checkstyle plugin provides the functionality of Checkstyle to Loom. See the Checkstyle website for a comprehensive documentation of how to write custom configurations.

Loom uses Checkstyle version 8.2.

7.1 Products

This plugin configures two products: checkstyleMainReport and checkstyleTestReport to provide reports for sources in src/main/java/ and src/test/java/ respectively.

These products are registered for the check goal (which is part of the build goal).

7.2 Minimum example

Listing 1: module.yml

```yaml
plugins:
  - checkstyle
```

By default, the checkstyle plugin reads its configuration from config/checkstyle/checkstyle.xml. This directory/file has to be located in the project directory (or in each of your module directories for multi module projects). This default can be changed via the setting configLocation.

7.3 More advanced example
In this example, a custom location `../config/checkstyle/checkstyle.xml` will be used. This is handy if you want to have one checkstyle configurations for multiple modules.

Checkstyle itself comes with two built-in checks: `/google_checks.xml` and `/sun_checks.xml` (note the leading `/` for built-in check configurations). Both are part of the official Checkstyle package and are maintained by the Checkstyle Team.

### 7.4 Settings

**configLocation** Path to a file containing the Checkstyle configuration. By default, `config/checkstyle/checkstyle.xml` is used. The path is relative to the Loom module directory, so each module requires its own configuration. If you want a checkstyle configuration per project, you can set `../config/checkstyle/checkstyle.xml` for example.

### 7.5 Notes

- This plugin handles the cache configuration of Checkstyle automatically. Do **not** include a `cacheFile` property within your `checkstyle.xml` – otherwise Loom’s CLI options to disable caching (`--no-cache` or `-n`) wouldn’t work as expected.
This plugin creates all necessary files for Eclipse IDE for Java Developers to work on your project.

8.1 Products

This plugin configures two products:

**eclipse** Creates and maintains the files `.project`, `.classpath` and `.settings/org.eclipse.jdt.core.prefs`. One set of project files will be created per module.

**Note**: Project files (.project and `**/*.prefs`) will be patched to meet the minimum requirements of an Eclipse IDE project. Existing configuration (natures/buildSpecs) will not be changed by the task.

**cleanEclipse** Removes the files created/patched by `eclipse` product: `.project`, `.classpath`, `.settings/**`

8.2 Settings

This plugin has no settings.
This plugin creates all necessary files for IntelliJ IDEA to work on your project.

### 9.1 Products

This plugin configures two products:

- **idea**: Creates a `.idea` directory with a couple of files and `.iml` files for all modules of your project.

- **cleanIdea**: Removes the `.idea` directory recursively and all `.iml` files within your project.

### 9.2 Settings

This plugin does currently not have any settings.
10.1 Resource filtering

The java plugin comes with resource filtering functionality. Filtering means, replacing special variable placeholders ($\{placeholder\}$) with dynamic values.

When resolving placeholders, Loom will first try to find a system property (specified via the -D switch on the CLI) with the name of the placeholder. If no system property with that name could be found, Loom will try to find an environment variable with that name.

The Loom build will fail, if a placeholder can’t be resolved. If you want to have default values instead, use the syntax $\{placeholder:default-value\}$.

The following placeholders are registered automatically:

- $\{project.version\}$ will be replaced by the version defined for the build. Don’t forget to specify a version for the build via the --release or -r parameter.

See the setting resourceFilterGlob for how to control which files should be filtered.

10.2 Settings

**mainClassName** This defines the fully qualified name of a Java class that contains a main method that should be used automatically, when launching the jar file this class is contained in. Technically, this sets the Main-Class attribute of the META-INF/MANIFEST.MF file of the jar file. For Java 9 modularized jar files, this also sets the ModuleMainClass attribute to the corresponding module-info.class file.

**resourceFilterGlob** Controls, which resource files should be filtered. A value of *.*.properties for example, ensures, that all files with a suffix of .properties within src/main/resources/ and src/test/resources/ of your module will be filtered. By default, no files will be filtered.
This plugin allows the use of JUnit within projects build with Loom. Loom uses the JUnit Platform version 1.0.3 (which is part of JUnit 5).

### 11.1 Products

This plugin configures the product `junitReport`. The product `junitReport` is registered for the check goal (which is part of the build goal).

### 11.2 Configuration

To execute Tests with JUnit 5 natively, this is your configuration:

```yaml
Listing 1: module.yml

plugins:
  - junit
testDependencies:
  - org.junit.jupiter:junit-jupiter-engine:5.0.3
```

### 11.3 Legacy configuration

If you have a project which currently depends on the API of JUnit 4 and don’t want to update your code to the new API of JUnit 5, you can configure Loom to use the JUnit Vintage dependency to test your project without any changes.
Listing 2: module.yml

```yaml
plugins:
  - junit
testDependencies:
```

11.4 Settings

This plugin has no settings.
This plugin allows Loom to access the repository offered by Apache Maven.

### 12.1 Products

The only user visible product of this Plugin is `mavenArtifact`. It installs all Jar products of your project into the local maven repository.

This allows sharing artifacts by Loom for other build tools that can access the local Maven repository.

The local maven repository is located at `~/.m2/repository/` by default. This default can be changed by the Maven specific `settings.xml` file, which is located in `${maven.home}/conf/settings.xml` or `${user.home}/.m2/settings.xml`. See Maven Settings for more details.

### 12.2 Settings

`groupId` and `artifactId` used by the `mavenArtifact` product when creating a `pom.xml` and installing to your local repository. The format is: `yourGroupId:yourArtifactId`. The also required `version` is specified through the `--release` or `-r` parameters of the Loom CLI.
Chapter 13

Plugin pmd

The PMD plugin provides the functionality of PMD to Loom. See the PMD website for a comprehensive documentation of existing RuleSets and how to create custom ones.

Loom uses PMD version 5.8.1.

13.1 Products

This plugin configures two products: pmdMainReport and pmdTestReport to provide reports for sources in src/main/java/ and src/test/java/ respectively.

These products are registered for the check goal (which is part of the build goal).

13.2 Settings

- minimumPriority
  Rules with a lower priority will not be used. Possible priorities are: LOW (default), MEDIUM_LOW, MEDIUM, MEDIUM_HIGH and HIGH

- ruleSets
  A comma separated list of RuleSets. By default, only rulesets/java/basic.xml is configured – which is embedded in PMD. If you specify a RuleSet which is not embedded in PMD, the specified path has to be relative to the module path for which the configuration is made. See the PMD RuleSet index and PMD RuleSet files for a full reference of all available RuleSets.
The SpotBugs plugin provides the functionality of SpotBugs to Loom.

From the SpotBugs website: *SpotBugs is the spiritual successor of FindBugs carrying on from the point where it left off with support of its community.*

Loom uses SpotBugs version 3.1.0, FbContrib version 7.0.5.sb, FindSecBugs version 1.7.1.

### 14.1 Products

This plugin configures two products: `spotbugsMainReport` and `spotbugsTestReport` to provide reports for compiled main and test classes respectively.

These products are registered for the `check` goal (which is part of the `build` goal).

### 14.2 Settings

- **customPlugins** Additional SpotBugs plugins can be activated by listing them comma separated. Currently available plugins are: FbContrib (see [FbContrib website](#)) and FindSecBugs (see [FindSecBugs website](#)).
- **effort** The effort level SpotBugs should use. Possible levels are: min, default (default), max. Higher values may find more bugs, but also have more cpu and memory consumption (and thus may take longer).
- **excludeFilterFiles** Filename of a filter configuration (blacklist). Example: `config/spotbugs/exclude-filters.xml`. See SpotBugs Filter file for details on how to create such a filter.
- **excludeTests** Boolean flag to allow the exclusion of test classes. As it is *not* recommended, the default is `false`. Set it to `true` to have standard Maven behaviour.
- **includeFilterFiles** Filename of a filter configuration (whitelist). Example: `config/spotbugs/include-filters.xml`. See SpotBugs Filter file for details on how to create such a filter.
- **reporter** The report output format. Possible values are: html (default) and xml.
**reportLevel** Defines the minimum priority a bug has to have to get reported. Possible levels are: LOW, NORMAL (default), HIGH.
To build a Spring Boot application, first create a `module.yml` configuration:

```yaml
plugins:
  - springboot

settings:
  springboot.version: 1.5.4.RELEASE

compileDependencies:
  - org.springframework.boot:spring-boot-starter-web:1.5.4.RELEASE
```

### 15.1 Alternative 1 - Build and run Fat-JAR

The first, any simplest way to build/launch a Spring Boot application is the Fat-JAR approach, where a single jar containing all other dependencies is created and launched.

```bash
./loom build springBootFatJarApplication
java -jar build/products/unnamed/springboot-fatjar/unnamed-fatjar.jar
```

### 15.2 Alternative 2 - Build application for Docker container

The second approach is, to create a Docker image for your application and start a Docker container with that image. First, create a file called `Dockerfile`:

Listing 1: Dockerfile

```dockerfile
FROM openjdk:9-slim
EXPOSE 8080
```

(continues on next page)
WORKDIR /app
COPY build/products/unnamed/springboot/META-INF /app/META-INF
COPY build/products/unnamed/springboot/org /app/org
COPY build/products/unnamed/springboot/BOOT-INF/lib /app/BOOT-INF/lib
COPY build/products/unnamed/springboot/BOOT-INF/classes /app/BOOT-INF/classes

springframework.boot.loader.JarLauncher"]

Then, build the application, the docker image and launch a container:

```
./loom build
docker build -t my-spring-boot-app .
docker run -ti my-spring-boot-app
```

### 15.3 Settings

**version**  Use this setting to specify the version of the Spring Boot Loader to use. Spring Boot 1.5.x releases are known to work.
CHAPTER 16

Continuous Integration

16.1 Travis CI

To build (and test) a project built with Loom using Travis CI, add this to your `.travis.yml` file:

```yaml
language: java
jdk:
  - oraclejdk9
script: ./loom -n build
cache:
  directories:
    - $HOME/.loom/library/
    - $HOME/.loom/repository/
```

For more information about Travis CI visit their online documentation.