Gradle assignment

During this assignment, we are going to get familiar with the build automation tool Gradle. A simple Java image web crawler will be implemented with the help of available free open-source dependencies and plugins.

Gradle is an open-source build automation tool designed to support multiple languages, like C, C++, Java, Scala, Kotlin and others. It uses Groovy (or Kotlin) based Domain Specific Language (DSL) over traditional XML build files.

If you don't have Gradle installed, please complete this assignment first.

Difficulty: ☆☆☆

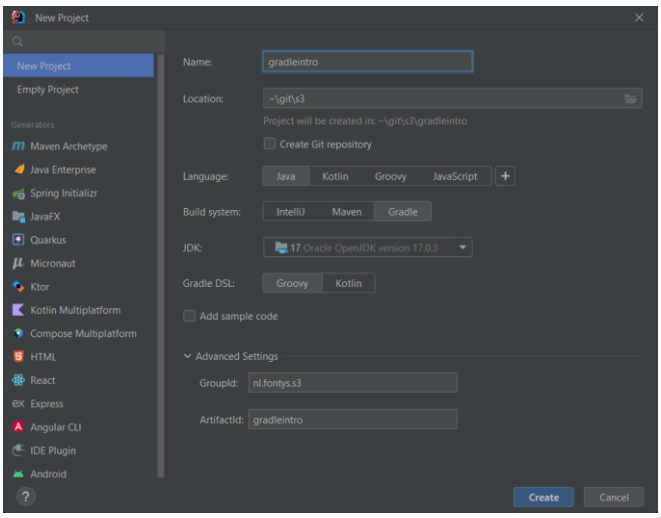
Learning objectives:

- I can setup a Gradle project using Intellij
- I can name all the Gradle parts of the project
- I can name all the Gradle parts of the build file
- I can add new dependencies to the Gradle project
- I can add new plugins to the Gradle project

Estimated time required: 90 minutes

1) Setting up a Gradle Project

Let's create a new Java/Gradle project in IntelliJ:

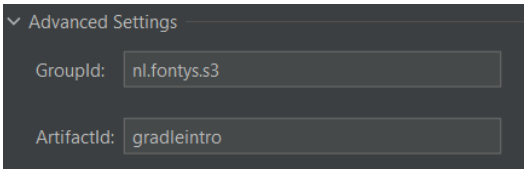


Make sure to select 'New Project' and not one of the generators. Let's explain the options here, and especially the build system option, inside the yellow box.

- **Name**: This is going to be the name of your project.
- **Location**: This is where the project is going to be located. Choose your preferred option.

- **Language**: This is the language of choice. Pick JAVA here.
- **Build system**: This is the build automation tool of choice. Make sure to pick Gradle here. Now when you picked Gradle, you will see another option arise:
- **Grade DSL**: This is the language used to configure Gradle. (DSL → Domain specific language). Make sure to pick Groovy here.
- **JDK**: Make sure to pick the latest JAVA LTS Development Kit (>= JDK 17).

Now if we look closely, you might have spotted a collapsed 'Advanced' section in this menu. Let us take a closer look at that as well, by unfolding it:



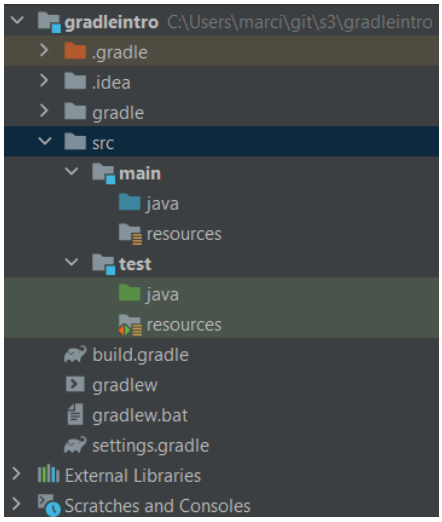
- **GroupId**: This is the 'family-name' or 'group name' of your products/software, which you can set to your own liking. Usually something like '<country>.<company>.<product group>'. But this can be literally anything, just make sure it is unique, to avoid clashes.
- **ArtifactId**: This is the name of your current project. IntelliJ automatically fills it in once the name of the project is informed.

When everything is set, and you picked the names for your project, click the Create button, and let us dive into the project. Make sure to wait a couple of seconds/minutes for IntelliJ to finish up everything.

If done correctly, your Gradle project should have been set up.

2) Exploring the Gradle Project

Now that we have the project in place, let's explore the its structure. Look at the following:



src folder

The *src* folder is where you are going to place your code. With the main code inside the *src/main/java* folder, and the testing code inside the *src/test/java* folder.

Auto-generated folders

There are a couple of auto-generated folders which we should NOT TOUCH:

- *.gradle*: actual gradle integration, managed by Gradle itself.
- *.idea*: IntelliJ specific folder.
- *gradle*: this is the folder containing the Gradle Wrapper.

Note: Folder *gradle* can and should be committed to git. It contains the gradle wrapper jar and a properties file which has the Gradle version to be used in the project. It's wise the all members of a team use the same version of Gradle, and the wrapper is a convenient way of enforcing this.

File build.gradle

Now open the following file: *build.gradle* (inside the root folder of the project).

This is the project's build configuration file. You should have the following:

```
plugins {
    id 'java'
}

group 'nl.fontys.s3'
version '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'
}

test {
    useJUnitPlatform()
}
```

plugins section:
Here you can add all the plugins that are necessary for your Gradle project. A plug-in extends a project by providing tasks and configuration. For instance, this is the documentation for the 'java' plugin.

group:
This is the group id, which you set during the creation of your project.

version:
This is the version of your project. It's only important if you're publishing Java libraries to be used in other projects.

repositories:
Now this is an interesting one. You see that it uses Maven central repository. We are using Gradle right? That is right, however, it (re)uses the already existing repositories of Maven, to be able to resolve/download any of the dependencies you want/need, which we will discuss later.

dependencies:
Here you can locate all dependencies your project is using. As you can see, a couple of them are already set, being the JUnit ones.

Dependencies can have different scopes. The main ones to be aware of:

- *implementation*: dependency is available during code compilation and runtime
- *compileOnly*: dependency is available during code compilation only
- *runtimeOnly*: dependency is available during application execution only
- *testImplementation*: dependency is available during test-code compilation and test executions
- testRuntimeOnly: dependency is available during test executions only

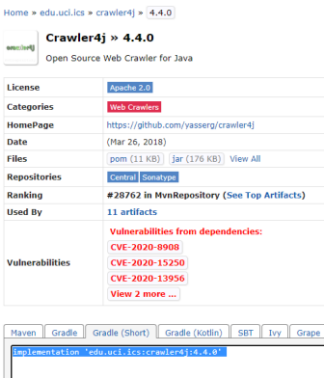
test:
This section configures the "test" task. In this case, we instruct Gradle to use the JUnit test platform to run the tests.

3) Adding dependencies

Let's say we want to write a simple program to do web crawling.

Then after some googling you find out that there's crawler4j Java library available in Maven and you want to give it a try. Note: From the Github repo, this project is pretty outdated, so wouldn't be a good option for a production project. That said, let's give it a try.

Searching for "crawler4j" in https://mvnrepository.com/ you can find the dependency, it's latest available version and also the Gradle way of declaring it.



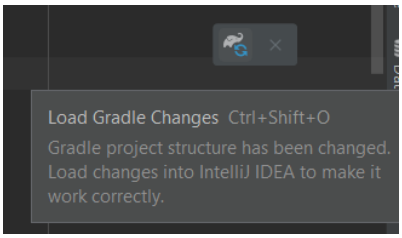
Let's copy the "Gradle (Short)" version of the dependency into our project's *build.gradle* dependencies. In practice there's also another dependency needed for the project to work correctly. Both dependencies are highlighted below:

```
plugins {
    id 'java'
}

group 'nl.fontys.s3'
version '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'edu.uci.ics:crawler4j:4.4.0'
    implementation 'com.sleepycat:je:18.3.12'
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'
}

test {
    useJUnitPlatform()
}
```

Formatted: Font: 10.5 pt, Complex Script Font: 10.5 pt

Formatted: Font: 10.5 pt, Font color: Accent 4, Complex Script Font: 10.5 pt

The Gradle icon with a refresh symbol will appear whenever you change *build.gradle* file. Always click it so IntelliJ loads the Gradle changes (i.e., it will download those dependencies --- the jar files --- into your project folder).



After loading, the dependencies are available in the project.

4) Using dependencies

Time for actually using crawler4j in the project. The use-case will be that we need to download all the images from a website. The crawler should keep running until there's no new page left to visit or the process is stopped.

Create a *nl.fontys.s3* package in *src/main/java* (right-click the folder -> New -> Package).

Then copy and paste the three classes below inside it.

ImageCrawler:

```
package nl.fontys.s3;

import edu.uci.ics.crawler4j.crawler.Page;
import edu.uci.ics.crawler4j.crawler.WebCrawler;
import edu.uci.ics.crawler4j.parser.BinaryParseData;
import edu.uci.ics.crawler4j.url.WebURL;
```

```java
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Collections;
import java.util.List;
import java.util.UUID;
import java.util.regex.Pattern;

public class ImageCrawler extends WebCrawler {

    private static final Pattern exclusionPatterns = Pattern.compile(
            ".*(\\.(css|js|mid|mp2|mp3|mp4|wav|avi|mov|mpeg|ram|m4v|pdf" +
                    "|rm|smil|wmv|swf|wma|zip|rar|gz))$");

    private static final Pattern imgPatterns =
Pattern.compile(".*(\\.(bmp|gif|jpe?g|png|tiff?))$");

    private final File storageFolder;
    private final List<String> crawlDomains;

    public ImageCrawler(File storageFolder, List<String> crawlDomains) {
        this.storageFolder = storageFolder;
        this.crawlDomains = Collections.unmodifiableList(crawlDomains);
    }

    @Override
    public boolean shouldVisit(Page referringPage, WebURL url) {
        String href = url.getURL().toLowerCase();
        if (exclusionPatterns.matcher(href).matches()) {
            return false;
        }

        if (imgPatterns.matcher(href).matches()) {
            return true;
        }

        for (String domain : crawlDomains) {
            if (href.startsWith(domain)) {
                return true;
            }
        }
        return false;
    }

    @Override
    public void visit(Page page) {
        String url = page.getWebURL().getURL();

        if (!imgPatterns.matcher(url).matches() ||
                !(page.getParseData() instanceof BinaryParseData)) {
            return;
        }

        String filename = getUniqueFileName(url);
        try {
            Files.write(Path.of(filename), page.getContentData());
            WebCrawler.logger.info("Stored: {}", url);
        } catch (IOException e) {
            WebCrawler.logger.error("Failed to write file: {}", filename, e);
        }
    }

    private String getUniqueFileName(String url) {
        String extension = url.substring(url.lastIndexOf('.'));
        String hashedName = UUID.randomUUID() + extension;

        return storageFolder.getAbsolutePath() + '/' + hashedName;
    }

}
```

ImageCrawlerRunner:

```java
package nl.fontys.s3;

import edu.uci.ics.crawler4j.crawler.CrawlConfig;
import edu.uci.ics.crawler4j.crawler.CrawlController;
import edu.uci.ics.crawler4j.fetcher.PageFetcher;
import edu.uci.ics.crawler4j.robotstxt.RobotstxtConfig;
import edu.uci.ics.crawler4j.robotstxt.RobotstxtServer;

import java.io.File;
import java.util.List;

public class ImageCrawlerRunner {
    private final CrawlController crawler;
    private final String storageFolder;
    private final int numberCrawlers;
    private final List<String> crawlDomains;

    public ImageCrawlerRunner(List<String> crawlDomains, String storageFolder, int
numberCrawlers) {
        this.crawlDomains = crawlDomains;
        this.crawler = createCrawler(crawlDomains);
        this.storageFolder = storageFolder;
        this.numberCrawlers = numberCrawlers;
    }

    public void run() {
        CrawlController.WebCrawlerFactory<ImageCrawler> factory =
createCrawlerFactory(crawlDomains);
        this.crawler.start(factory, this.numberCrawlers);
    }

    private CrawlController createCrawler(List<String> crawlDomains) {
        CrawlConfig config = new CrawlConfig();
        createFolderIfNeeded(getTmpFolder());
        config.setCrawlStorageFolder(getTmpFolder());
        config.setIncludeBinaryContentInCrawling(true);

        PageFetcher pageFetcher = new PageFetcher(config);
        RobotstxtConfig robotstxtConfig = new RobotstxtConfig();
        RobotstxtServer robotstxtServer = new RobotstxtServer(robotstxtConfig, pageFetcher);
        try {
            CrawlController controller = new CrawlController(config, pageFetcher,
robotstxtServer);
            for (String domain : crawlDomains) {
                controller.addSeed(domain);
            }
            return controller;
        } catch (Exception e) {
            throw new RuntimeException("Error creating crawler.", e);
        }
    }

    private CrawlController.WebCrawlerFactory<ImageCrawler> createCrawlerFactory(List<String>
crawlDomains) {
        File storageFolder = new File(getResultsFolder());
        createFolderIfNeeded(getResultsFolder());
        return () -> new ImageCrawler(storageFolder, crawlDomains);
    }

    private String getResultsFolder() {
        return this.storageFolder + "/results";
    }

    private String getTmpFolder() {
        return this.storageFolder + "/tmp";
    }

    private void createFolderIfNeeded(String folderPath) {
        File folder = new File(folderPath);
        if (!folder.exists()) {
            boolean created = folder.mkdirs();
            if (!created) {
                throw new RuntimeException("Error creating folder: " + folderPath);
            }
        }
    }
}
```

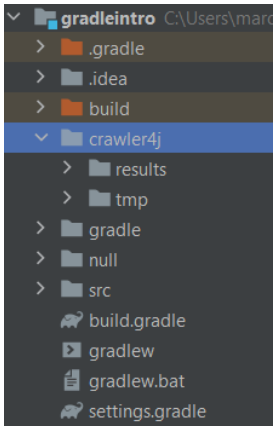
ImageCrawlerMain:

```
package nl.fontys.s3;

import java.util.List;

public class ImageCrawlerMain {
    private static final int NUMBER_OF_CRAWLERS = 2;
    private static final String PROJECT_DIR = System.getProperty("user.dir");
    private static final String CRAWLER_STORAGE_FOLDER_PATH = PROJECT_DIR + "/crawler4j/";

    public static void main(String[] args) {
        List<String> crawlDomains = List.of("https://www.wired.com/");
        ImageCrawlerRunner imageCrawlerRunner = new ImageCrawlerRunner(crawlDomains,
CRAWLER_STORAGE_FOLDER_PATH, NUMBER_OF_CRAWLERS);
        imageCrawlerRunner.run();
    }
}
```

Take your time and try understanding what this code is about. For instance, what does extends keyword mean in Java?

When you're done, then go to class *ImageCrawlerMain* and run the main method. That will start crawling site https://www.wired.com/ for images.

Then, after refreshing the Project view, a new *crawler4j* folder will appear:



And all the images will be inside *results* folder.

If you had enough images, you can stop the program execution.

5) Running the project with the Gradle wrapper

In the previous step we ran the program from IntelliJ. This time we will run it from the Gradle wrapper and practice a few more Gradle commands in the process.

In IntelliJ's terminal type command: *gradlew tasks*.

```
(base) PS C:\Users\marci\git\s3\gradleintro> gradlew tasks

> Task :tasks

------------------------------------------------------------
Tasks runnable from root project 'gradleintro'
------------------------------------------------------------

Build tasks
-----------
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
```

This will show you all available tasks in your project. Be aware and try other commands like clean, test and build.

Notice that there is no “run” task available.

Let’s add Application plugin to the project now. It’s just about adding the highlighted lines below. Everything else stays the same:

```
plugins {
    id 'java'
    id 'application'
}

application {
    mainClass = 'nl.fontys.s3.ImageCrawlerMain'
}

...
```

Formatted: Font color: Accent 4

Formatted: Font color: Accent 4

Now a new “run” task should show-up after executing *gradlew tasks*:

```
------------------------------------------------------------
Tasks runnable from root project 'gradleintro'
------------------------------------------------------------

Application tasks
-----------------
run - Runs this project as a JVM application
```

Let’s use it:

```
Terminal:  Local  +
(base) PS C:\Users\marci\git\s3\gradleintro> gradlew run
```

Images coming through! All good. Press *Ctrl+c* in the terminal when you have enough of them.

After this step you should understand better the difference between plugins and dependencies in Gradle. Good work!