CONTINUOUS INTEGRATION
LAB EXERCISES

LECTURER: ROB FRAMPTON

In this lab you will setup some tools which help with *continuous integration*, in this case for developing a statically-generated website called `FamousQuotes`. The tools include version control, automated building, automated testing, a CI server, and automated deployment. The website is a Java project and is available on Moodle. **Download it and open the folder in Visual Studio Code**. You will also need Java and Git installed on your machine.

You can download the Java Development Kit (JDK) from:

https://www.oracle.com/uk/java/technologies/javase-downloads.html

You can download git from: https://git-scm.com/

You can download Visual Studio Code from: https://code.visualstudio.com/download

Note: Microsoft's *Visual Studio Code* is a different product to *Visual Studio*.

Exercise 1: Continuous Integration and Deployment

Exercise 1.1: Git repository setup

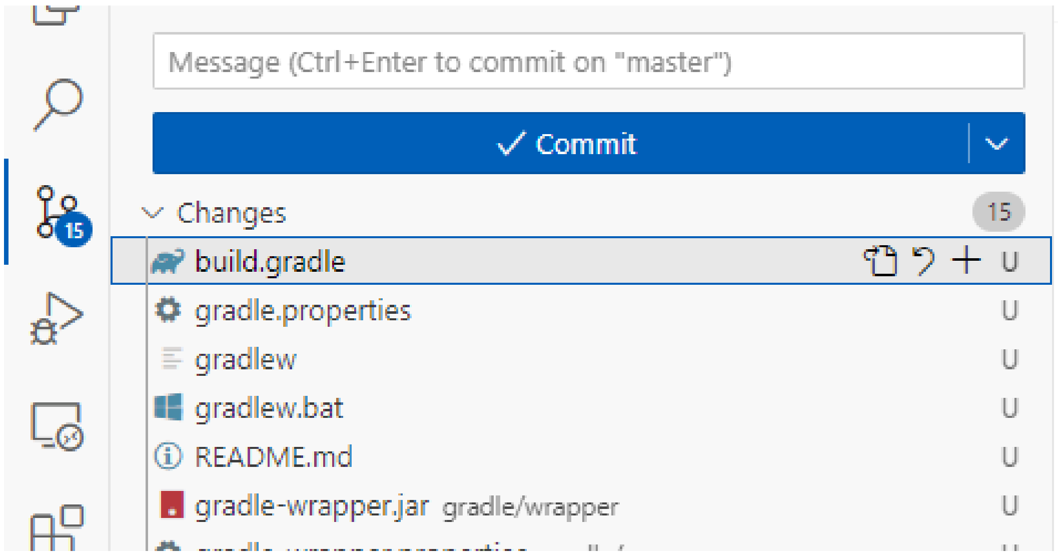
The `FamousQuotes` website is a Java project. Let's assume that this project is being worked on by two developers. We wish to use source control so that they can work independently from a single code base, and so that they can roll back any changes they have made.

- Open the `FamousQuotes` project in VS Code.
- If not already open, click **"View -> Terminal"** to make a terminal appear in the lower pane.

In the terminal window, **type the following command to initialise a new git repository:**

```
git init
```

Now select the "Source Control" icon on the far left hand side of the screen. You should see something like this:



This tells us that these files have changes which have not yet been committed to source control, which makes sense as we have only just created the repository. We want to commit these files as the initial commit. To do this, we *stage* the files - which means they are ready to be committed – and then commit.

You can either **use the command line:**

- **Type: `git add *` in the terminal to stage the new files**
- **Type: `git commit -m "Initial commit"` to make the commit**

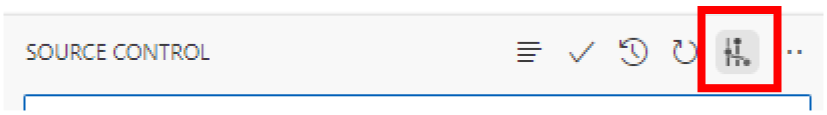
Or **use the GUI:**

- **Click on the plus icon next to "Changes" (which only appears when you hover over it), to stage all files**
- **Type "Initial commit" into the Message box and click the "Commit" button**

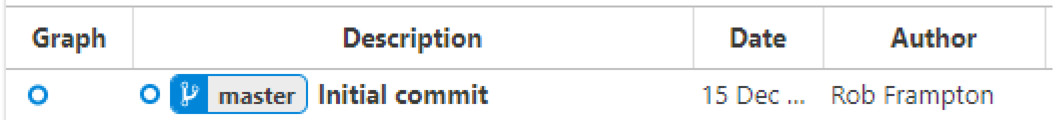
Now, whatever changes we make to the code, we can always revert back to this version if we want to.

Graph view

It is often useful to visualise branches and their commits as a graph. **Click "Extensions" on the far left side, and search for and install the Git Graph extension**. After installing, you will see a new graph icon appear back on the Source Control pane:



Clicking it allows us to see the full commit history, including the commit message, the date, and the author.



Graph	Description	Date	Author
○	○ master Initial commit	15 Dec ...	Rob Frampton

Exercise 1.2: Automated build and test with Gradle

For ease, the `FamousQuotes` project already contains the Gradle wrapper scripts which enable you to run Gradle without having it installed on your machine. **Run the following command in the terminal window**:

```
gradlew
```

Note: this command depends on your command line shell; if it does not work, try:

```
sh gradlew
```

You should see some output from Gradle which ends with "Build Successful".

Although Gradle is now installed, we have not configured it for our project. We need to complete the `build.gradle` file for the project to work. **Open `build.gradle`**.

First, we must tell Gradle that this is a Java project, and give it the name of the main class where execution starts. Add the following lines to the file:

```
apply plugin: 'java'
apply plugin: 'application'

mainClassName = 'mmu.quotes.Main'
```

Also, we need to tell Gradle that this project depends on external libraries. There are two libraries used by this project:

- SnakeYAML, which is a library for reading YAML files, and is used to read the content for the statically generated site
- JUnit, the Java test runner for the unit tests.

We can tell Gradle about these dependencies, and also where to find them, by adding the following lines to `build.gradle`:

```
repositories {
    mavenCentral()
}
```

```
dependencies {
    testImplementation "junit:junit:4.12"
    implementation 'org.yaml:snakeyaml:2.2'
}
```

This is all we need for Gradle to build a simple Java program. To build the program, **run the command**:

```
gradlew build
```

The first time you run Gradle, this may take a few minutes. Gradle should build your project and display "Build Successful". You may notice that Gradle creates a folder called `build`, in which you can see the compiled class.

Note: If you receive a message saying that `javac` could not be found, you should ensure that:

- The Java Development Kit (JDK) is installed on your system
- Your PATH environment variable includes the location of the JDK

Note: If you get a message saying that Gradle was unable to find a JDK installation, you may need to set a variable in the `gradle.properties` file. The variable should something like this:

```
org.gradle.java.home=C:\\Program Files\\Java\\Your-Jdk-Folder
```

The project also has unit tests. They were automatically executed when we ran the Gradle build, but they can also be run explicitly:

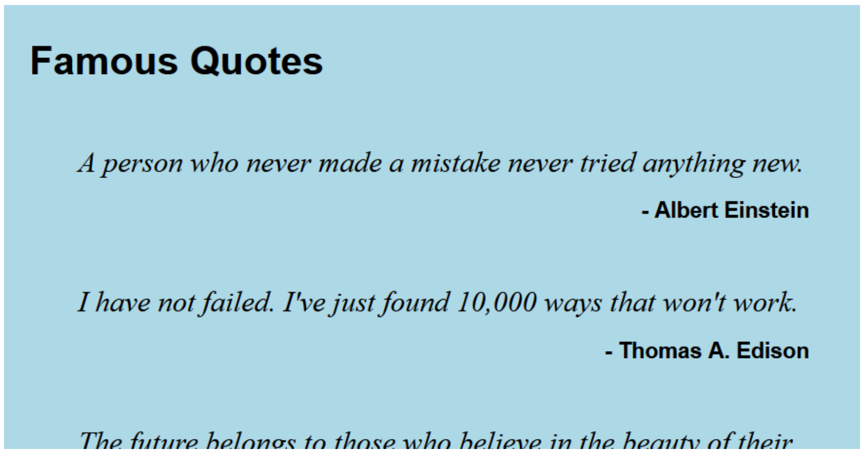
```
gradlew test
```

Now that the build was successful, we can run the project.

Run the project with the following command:

```
gradlew run
```

This project is a simple static site generator which will create a website in a folder called `dist`. If it was successful, you should now see a folder called `dist` with a file called `index.html`. Opening the file in your web browser should look like this:


Famous Quotes

A person who never made a mistake never tried anything new.

- Albert Einstein

I have not failed. I've just found 10,000 ways that won't work.

- Thomas A. Edison

The future belongs to those who believe in the beauty of their

.gitignore

You may notice that, after running the project, there are many new files which are listed as "Changes" in the Source Control pane. These new files are Gradle packages, compiled Java classes, and the actual output of our static site generator. Since they are all build artifacts and not source code, **none of these files should be committed to source control**.

However, it is annoying to have them appear under changes, and may lead us to accidentally commit them in the future. So, we will tell Git to ignore this type of file.

- **Create a new file called `.gitignore`**
- **Tell Git to ignore all files in the** `.gradle`**,** `build`, `bin` **and** `dist` **folders by adding the following lines:**

```
.gradle/
build/
bin/
dist/
```

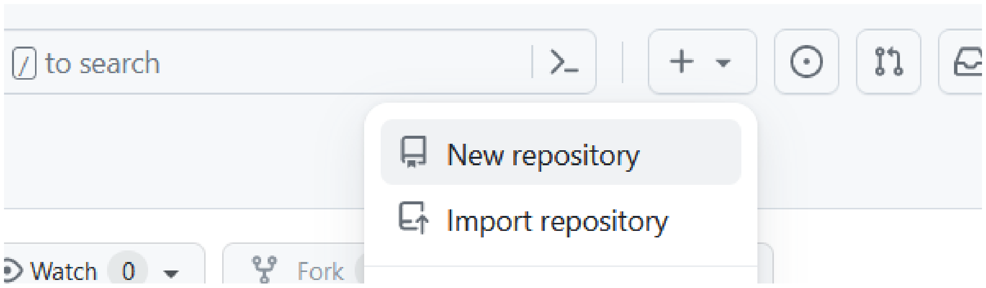
- **Save the file.** Most of the files under "changes" will now disappear.
- **Commit `.gitignore` and `build.gradle`.** You can use either method described above during our initial commit. Write a sensible commit message, such as: "Added build.gradle and .gitignore".

Exercise 1.3: Push to GitHub

For this exercise, you will need a GitHub account. If you don't have one already, create an account on github.com and sign in.

We are going to use GitHub as a remote repository, which will help us to share code with other developers. **Open github.com in your web browser and ensure you are signed in**.

From the GitHub home screen, **click the Create New icon in the top right corner, and select New Repository.**

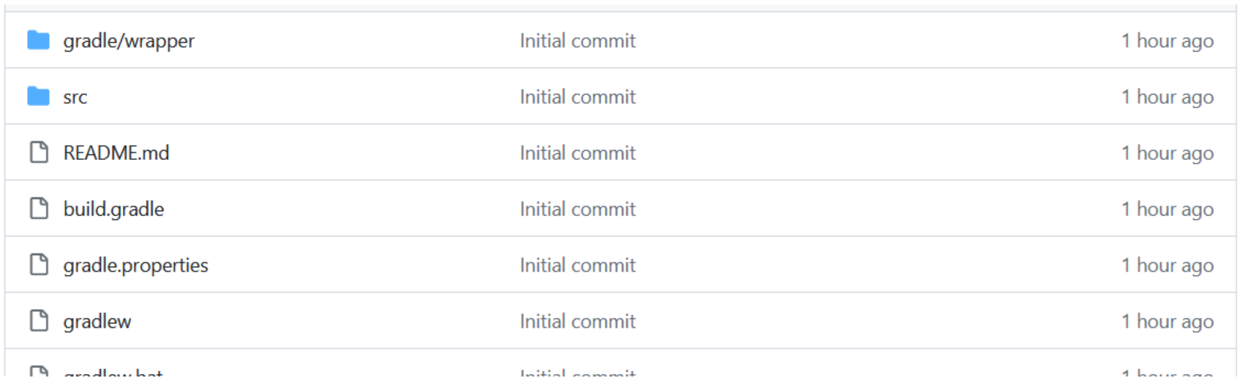


Give the repository a name, and choose "Public". **You MUST choose public for the later exercises to work. Do NOT add a README, or add a .gitignore or license**, because we will push our existing repository and adding commits here could cause problems. **Click "Create Repository".**

We will then **follow the instructions** under the heading "*…or push an existing repository from the command line*". The commands shown there will add the new GitHub repository as a "remote" destination on your local repository (i.e., a place where you can push your commits), rename the branch to `main`, and then push all the commits we have made so far up to GitHub. **Copy each of the three commands into your terminal**.

Note: if you have not used GitHub with VS Code before, after the last command, VS Code will require you to sign in with your GitHub credentials.

Now, click on your repository name in GitHub to see the repository. You should see all the files we have committed from VS Code.



gradle/wrapper	Initial commit	1 hour ago
src	Initial commit	1 hour ago
README.md	Initial commit	1 hour ago
build.gradle	Initial commit	1 hour ago
gradle.properties	Initial commit	1 hour ago
gradlew	Initial commit	1 hour ago

Exercise 1.4: Continuous Integration pipeline with GitHub Actions

A typical part of continuous integration is to have automated building and testing occur on a build server whenever code is pushed to the repository. GitHub has a feature to do this called *GitHub Actions*.

GitHub Actions works by looking for a workflow file in your repository. It expects to find this file in the `.github/workflows` directory.

In VS Code, **create the `.github/workflows` directory, and create a file inside it called `main.yml`.**

First, we tell GitHub that this workflow should run every time we push commits to our `main` branch. **Add the following lines to the file:**

```
name: FamousQuotes build

on:
  push:
    branches:
      - main
```

Now, we tell GitHub what we want to happen when the workflow runs. Add a job to the workflow by **adding these lines to the end of the file**:

```
jobs:
  build:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      pages: write
      id-token: write


    steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Build the project
      run: sh gradlew build
```

Notice that this simply checks the code out of the repository (on a virtual machine in the cloud), and then runs the same Gradle command we run locally.

Note: The “permissions” field is a more advanced feature which is required for Exercise 1.5 to work.

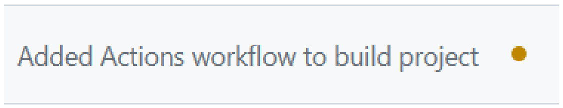
Commit this new file.

While that has committed the file to our local git repository, we won’t see anything on GitHub unless we *push* our changes. You can either **use the command line**:

```
git push
```

Or press the **Sync Changes** button in the user interface.

Look at GitHub after doing this. You may want to refresh the page if it hasn’t updated. You will notice an orange dot next to the latest commit, which indicates that an action is currently running.


Added Actions workflow to build project ●

If you click on the Actions tab, you can see more detail about the workflow. You can click through and see even more detail, include the actual command line output.

After a short while, the workflow should succeed. The orange dot will turn into green tick. We now have a working CI pipeline which will pick up build or test errors in our code very quickly after they are pushed.

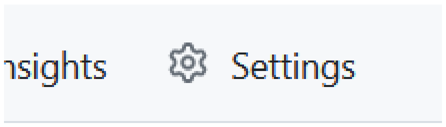
Try committed some broken code. You could try removing a semicolon from one of the `.java` files, or you could change a test assertion in `ComponentTests.java` so that the test fails. What happens when you push?

Exercise 1.5: Continuous Deployment with GitHub Actions

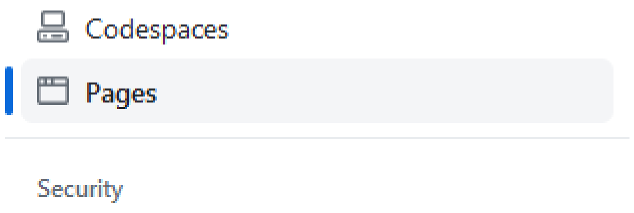
Our current workflow is useful for warning us that we have pushed broken code. However, another common practice is *continuous deployment*, in which we automatically deploy the code from the repository to a development website, so we can see the product and test it as soon as possible.

We will use a feature of GitHub called GitHub Pages, which is a free static web hosting feature.

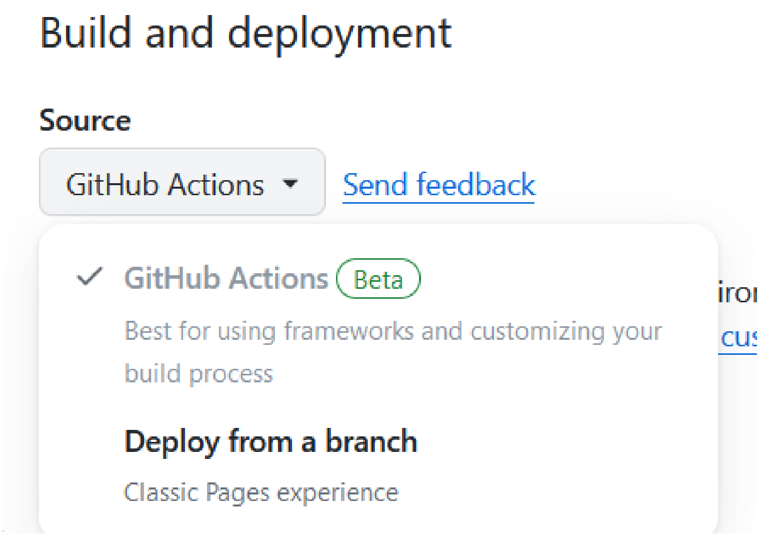
First, GitHub Pages must be enabled. **Select the Settings tab for your repository**:



And choose "Pages" from the left-side navigation.



Under "Build and deployment", "Source", select "GitHub Actions".

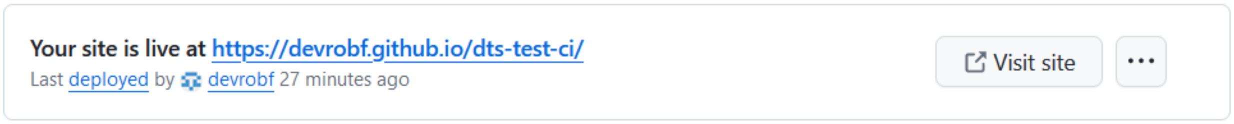


Now return to VS Code and add the following lines to the end of the workflow. These lines invoke our static site generator, which outputs the static site into the `dist` folder, and then deploys the result to the GitHub Pages page.

```
    - name: Generate the static site
      run: sh gradlew run

    - name: Setup Pages
      uses: actions/configure-pages@v4

    - name: Upload artifact
      uses: actions/upload-pages-artifact@v2
      with:
        path: './dist/'

    - name: Deploy to GitHub Pages
      id: deployment
      uses: actions/deploy-pages@v3
```

Commit and push this change.

After your workflow completes, **return to the Pages section of the Settings tab**. You should see a message like this:


Your site is live at https://devrobf.github.io/dts-test-ci/

Last deployed by devrobf 27 minutes ago

You can now visit the deployed site.

Try making a change to the website and pushing it. For example, you could open `src/resources/style.css` and change the background colour. After pushing and waiting for the workflow to complete, you should see your changes reflected on the deployed page.

Exercise 2: GitFlow

Implementing a Feature with Branching

The initial commit created the `master` branch, which is the mainline stable branch. If you completed Exercise 1.3 and pushed your repository to GitHub, this branch was renamed `main`.

For GitFlow, we need an active development branch called `develop`. **In the terminal window, use the following git command to create a branch called develop**:

```
git branch [branch-name]
```

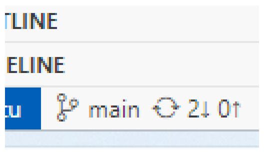
We do not commit code directly to `develop`, either. Instead, we work in a feature branch. The first feature we are going to implement is to update the styling of the webpage, so create another new branch called `feature/update-styling`.

It is conventional to always put `feature/` in front of your feature branch names. You can also use `bug/` for bug fixes.

Although we have *created* the branches, we aren't currently "working on" them – we are still on the original `main` branch. The command to checkout (i.e. switch to) a branch is:

```
git checkout [branch-name]
```

Alternatively, we can **create** and **switch** branches by clicking on the branch display in the lower-left corner of VS Code.



Note: if you create branches with the terminal and they do not appear here, you may need to click the Refresh button in the source control panel:

Using either method, **checkout the `feature/update-styling` branch**.

Implementing a Feature

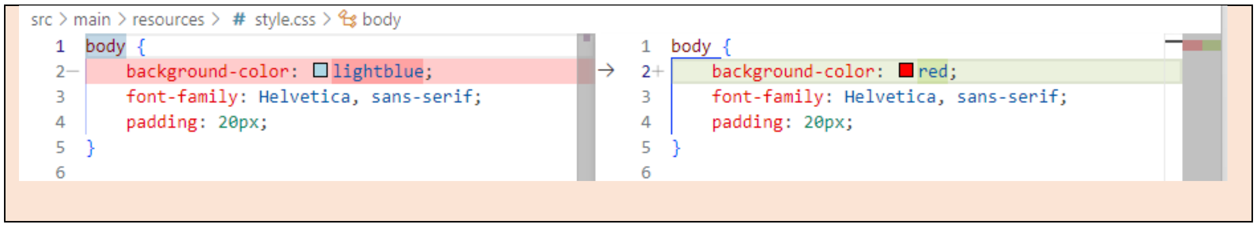
Now that we are on our feature branch, we can start implementing. We will just make very basic commits for demonstration purposes, but in reality these are likely to be more substantial.

- **Open `src/main/resources/style.css` and change the background colour to red.**
- **Stage and commit this change.**
- **Now, under `.quote-author`, add the line:**

```
text-transform: uppercase;
```

- **Stage and commit this change.**

Note: when you click on a file in the source control view, it will show you the changes you made since the last commit. This can be very useful for checking what you are committing:



src > main > resources > # style.css > body

```
body {
    background-color: lightblue;
    font-family: Helvetica, sans-serif;
    padding: 20px;
}
```

```
body {
    background-color: red;
    font-family: Helvetica, sans-serif;
    padding: 20px;
}
```

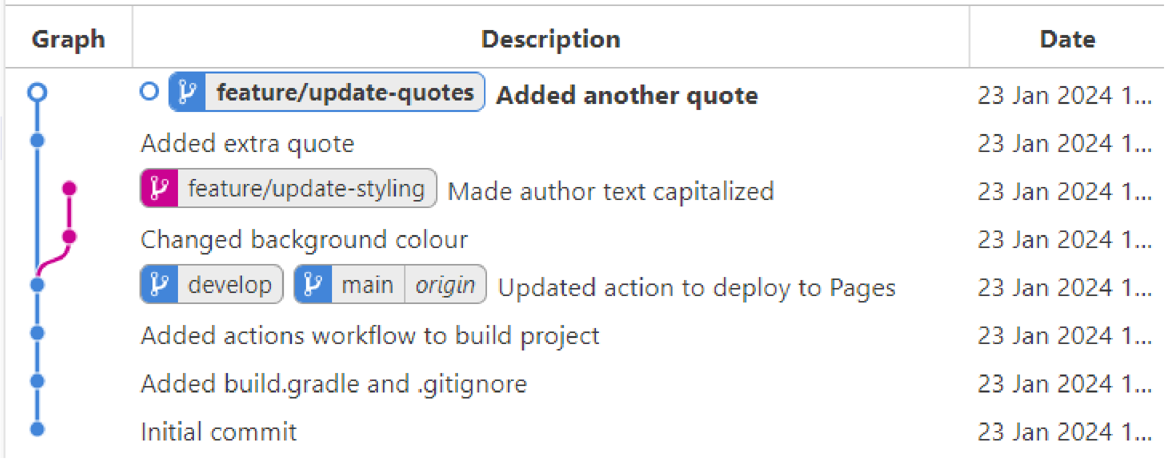
Implementing a Different Feature

While this feature is being implemented, it might be the case than *another* developer is implementing a *different* feature on another branch. This exercise will simulate this scenario.

Implement a new feature. You should:

- **Checkout the develop branch.** Notice the changes you made will disappear, since that code only exists in its feature branch.
- **Create a new branch called `feature/update-quotes`.**
- **Open `src/main/resources/quotes.yaml`.**
- **Add a new quote, and commit this change.**
- **Add another new quote, and commit this change.**

To help visualise what we have done so far, have a look at the graph view. You can see how the `develop` branch has split into `feature/update-styling` and `feature/update-quotes`.



Graph	Description	Date
	feature/update-quotes **Added another quote**	23 Jan 2024 1...
	Added extra quote	23 Jan 2024 1...
	feature/update-styling Made author text capitalized	23 Jan 2024 1...
	Changed background colour	23 Jan 2024 1...
	develop main *origin* Updated action to deploy to Pages	23 Jan 2024 1...
	Added actions workflow to build project	23 Jan 2024 1...
	Added build.gradle and .gitignore	23 Jan 2024 1...
	Initial commit	23 Jan 2024 1...

Merging Features into Develop

Now that we have finished the new quotes feature, we wish to merge it in to develop. The command to merge a branch *in* to the *current branch* is:

```
git merge [branch-name]
```

First: **check out `develop`, and merge in `feature/update-quotes`**.

Now, let's pretend the developer of the updated styling feature also decides they are finished. Before they can merge in to `develop`, *they must merge `develop` into their feature branch* in case `develop` has moved on (which it has) and there are conflicts.

Check out `feature/update-styling`, and merge `develop` in.

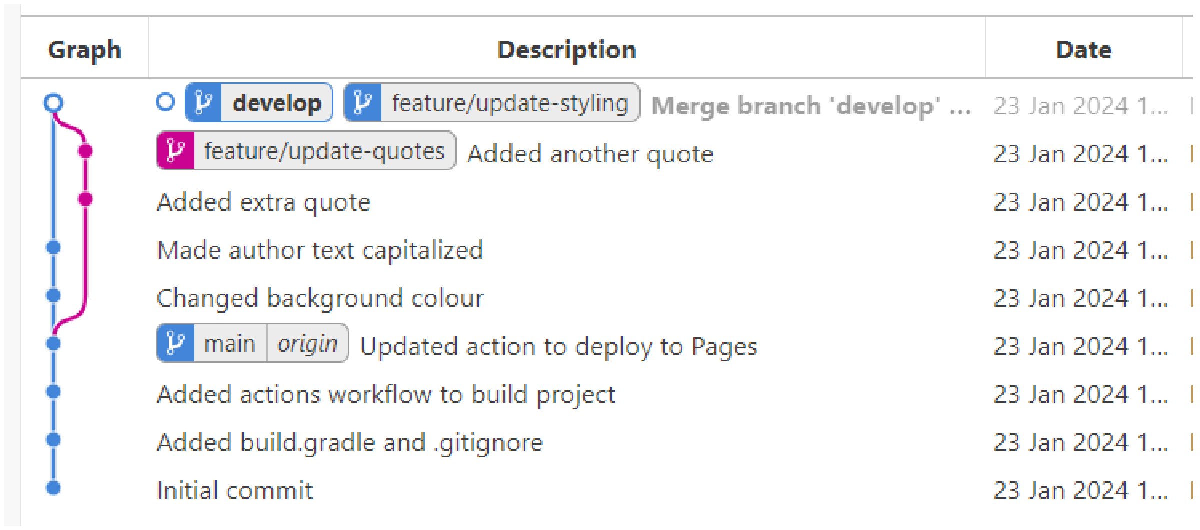
You should find that git is safely able to merge, since the changes were in different files. If we had modified the same files on different branches, this is the point where we would have to merge them – safely on our feature branches, where we can test we didn't break anything.

Finally, we can merge our feature in to `develop`.

Checkout `develop` and merge in `feature/update-styling`.

The result should be a project with both the new styling and the new quotes. Notice how the graph now shows how the codebase split into two branches, before merging back into one.

Note: the graph only shows two branches rather than three, because we did not make any commits directly to develop, so there was no reason for the graph to show a third branch



Graph	Description	Date
	develop feature/update-styling Merge branch 'develop' ...	23 Jan 2024 1...
	feature/update-quotes Added another quote	23 Jan 2024 1...
	Added extra quote	23 Jan 2024 1...
	Made author text capitalized	23 Jan 2024 1...
	Changed background colour	23 Jan 2024 1...
	main origin Updated action to deploy to Pages	23 Jan 2024 1...
	Added actions workflow to build project	23 Jan 2024 1...
	Added build.gradle and .gitignore	23 Jan 2024 1...
	Initial commit	23 Jan 2024 1...

Exercise 3: Feature flags

In this exercise, we will change the static site generator to read from a JSON file instead of a YAML file. Using a Trunk-Based Development philosophy, we want to avoid making one large commit with all our changes. Instead, we will use a **feature flag** to enable and disable our new functionality, so that we can commit straight to main incrementally.

Note: You will need to have completed Exercise 1 to build the project. If you haven't, you can download the completed code from Moodle and work from there.

Add necessary abstractions

First, we will use standard object-oriented principles to make it easy to change the implementation of our quotes reader.

We will write a generic interface which both our YAML and JSON quote readers will use. **Create the file `src/main/java/mmu/quotes/QuoteReader.java`**, and **write the following interface**:

```java
package mmu.quotes;

import java.io.FileNotFoundException;
import java.util.List;
import java.util.Map;

public interface QuoteReader {
    List<Map<String, Object>> readQuotes() throws FileNotFoundException;
}
```

Then **open `src/main/java/mmu/quotes/YamlQuoteReader.java`**, and **change the class definition** so that it implements this new interface:

```java
public class YamlQuoteReader implements QuoteReader {
```

Now, we can use our generic `QuoteReader` interface anywhere that needs to read quotes. Go to `src/main/java/mmu/quotes/Main.java` and **change the declaration of the reader variable** to use our new interface type instead:

```java
QuoteReader reader = new YamlQuoteReader("src/main/resources/quotes.yaml");
```

Commit this change to the main branch. We want our fellow developers to get this small change as soon as possible to avoid conflicts.

Add the feature flag

Next, we must implement a feature flag which enables or disables our new functionality. We will leave the feature turned off in main, but we can turn it on for our own development.

We will use a command line argument to switch the feature on or off. **Update `Main.java`** so that the `reader` is now created like this:

```
QuoteReader reader = null;
if (args.length > 0 && args[0].equals("json")) {
    throw new UnsupportedOperationException(
        "TODO: implement JSON reader");
}
else {
    reader = new YamlQuoteReader("src/main/resources/quotes.yaml");
}
```

This code checks to see if the command line argument "`json`" has been passed to the program. **Test that it works:**

- First, **run the program normally**, which should be successful:

  ```
  gradlew run
  ```

- Now, **run with the `json` argument** like this, which should show our error message:

  ```
  gradlew run --args="json"
  ```

Because the switch is off by default, the code is not broken, even though we haven't implemented our feature yet. This means you should **commit the code**, so that other developers receive the change as soon as possible.

Implement the feature

Now we can start implementing our new feature. First, our JSON feature will need a library called GSON. In `build.gradle`, add the following line in the dependencies section:

```
implementation 'com.google.code.gson:gson:2.10.1'
```

This won't break the code for other developers, so **commit this change**.

Next, we need to implement the new `QuoteReader` which reads from JSON files. Create the file `src/main/java/mmu/quotes/JsonQuoteReader.java`, and write the following code:

```
package mmu.quotes;

import java.util.*;
import java.io.*;
import com.google.gson.Gson;
```

```
import com.google.gson.reflect.TypeToken;

public class JsonQuoteReader implements QuoteReader {
    private String filePath;

    public JsonQuoteReader(String filePath) {
        this.filePath = filePath;
    }

    public List<Map<String, Object>> readQuotes() throws FileNotFoundException
{
        Gson gson = new Gson();
        FileReader fileReader = new FileReader(filePath);
        return gson.fromJson(fileReader,
            new TypeToken<List<Map<String, Object>>>() {}.getType());
    }
}
```

Let's integrate this by replacing the exception we wrote in Main.java with code which creates our new `JsonQuoteReader`, so it now looks like this:

```
if (args.length > 0 && args[0].equals("json")) {
    reader = new JsonQuoteReader("src/main/resources/quotes.json");
}
else {
   reader = new YamlQuoteReader("src/main/resources/quotes.yaml");
}
```

Now let's **commit these changes**.

You may notice that we've made a mistake – the code wants to read from a JSON file, but that file doesn't exist! This is a good example of *why feature flags are useful* – our code is broken, but because the flag is *off* for all the other developers, we haven't broken their code. This is different from GitFlow, where we keep all our commits in a different branch.

Let's fix the issue. Create the file `src/main/resources/quotes.json`, and add some quotes:

```
[
  {
    "author": "Albert Einstein",
    "quote": "Imagination is more important than knowledge."
  },
  {
    "author": "Thomas A. Edison",
    "quote": "The best way to predict the future is to create it."
  }
]
```

Test the program:

- First, **run with the `json` feature flag switched on**, as before. Your program should succeed, and the output in `dist/index.html` should contain quotes from the JSON file.
- Next, **run without the `json` feature flag**. Your program should succeed, and the output in `dist/index.html` should contain quotes from the YAML file.

Since this is working, we can **commit the changes**.

Remove the switch

Finally, we want to change our code to make the new functionality the default. This is simple; we just remove the switch in Main.java, just leaving the line which creates the JSON reader:

```
reader = new JsonQuoteReader("src/main/resources/quotes.json");
```

Test that everything still works, and **commit** this change to main.