ECSE 321 Introduction to Software Engineering

Hands-on Tutorials

McGill University

Table of Contents

- HTML version
- PDF version

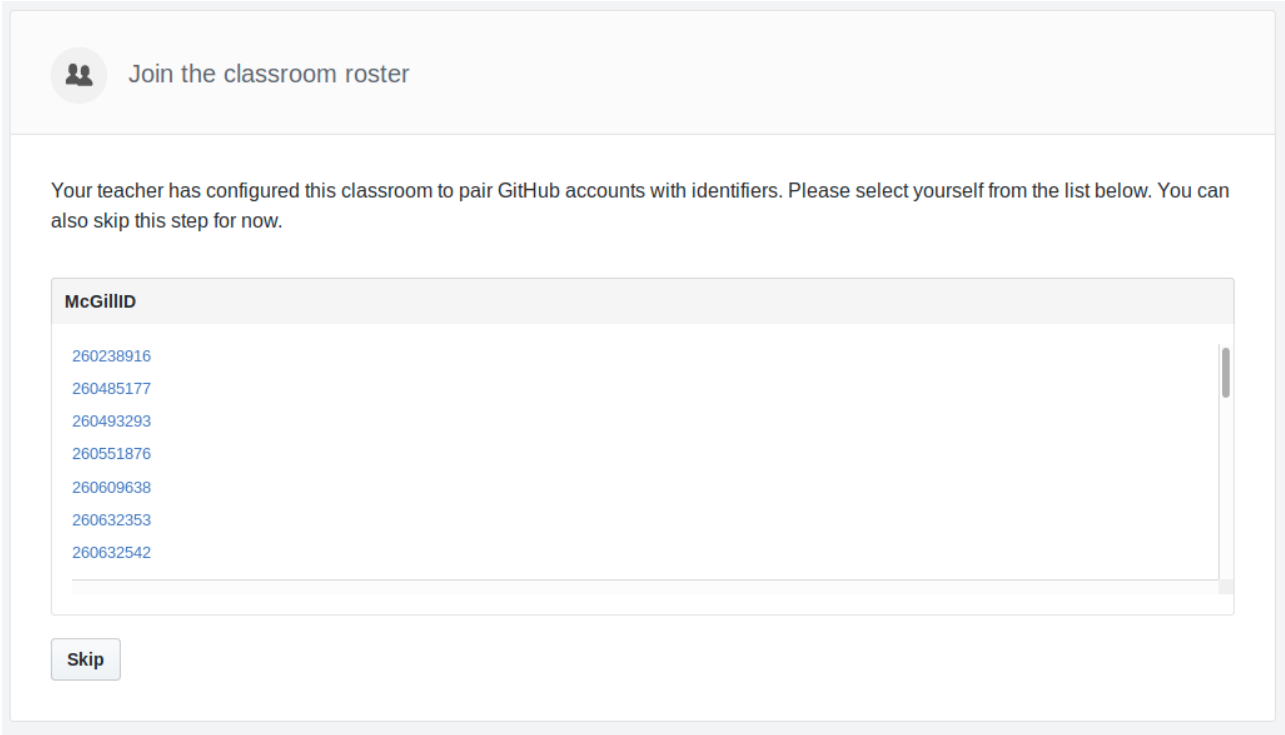
Sections of the tutorial will continuously be published at this web page.

1. Preliminaries

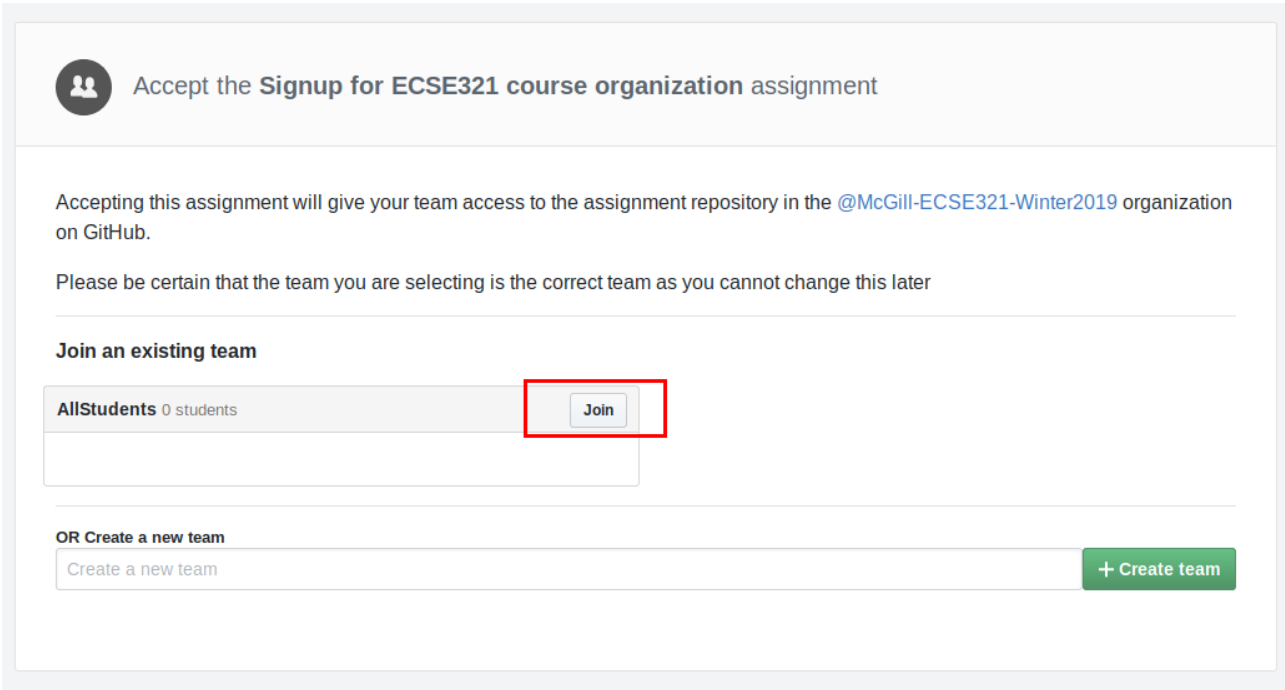
1.1. Getting Started

Steps for signing up for GitHub classroom:

1. Log in/Register on GitHub.
2. Open link https://classroom.github.com/g/o9gWNZis
3. Select your McGill ID from the list



4. Join team *All students*

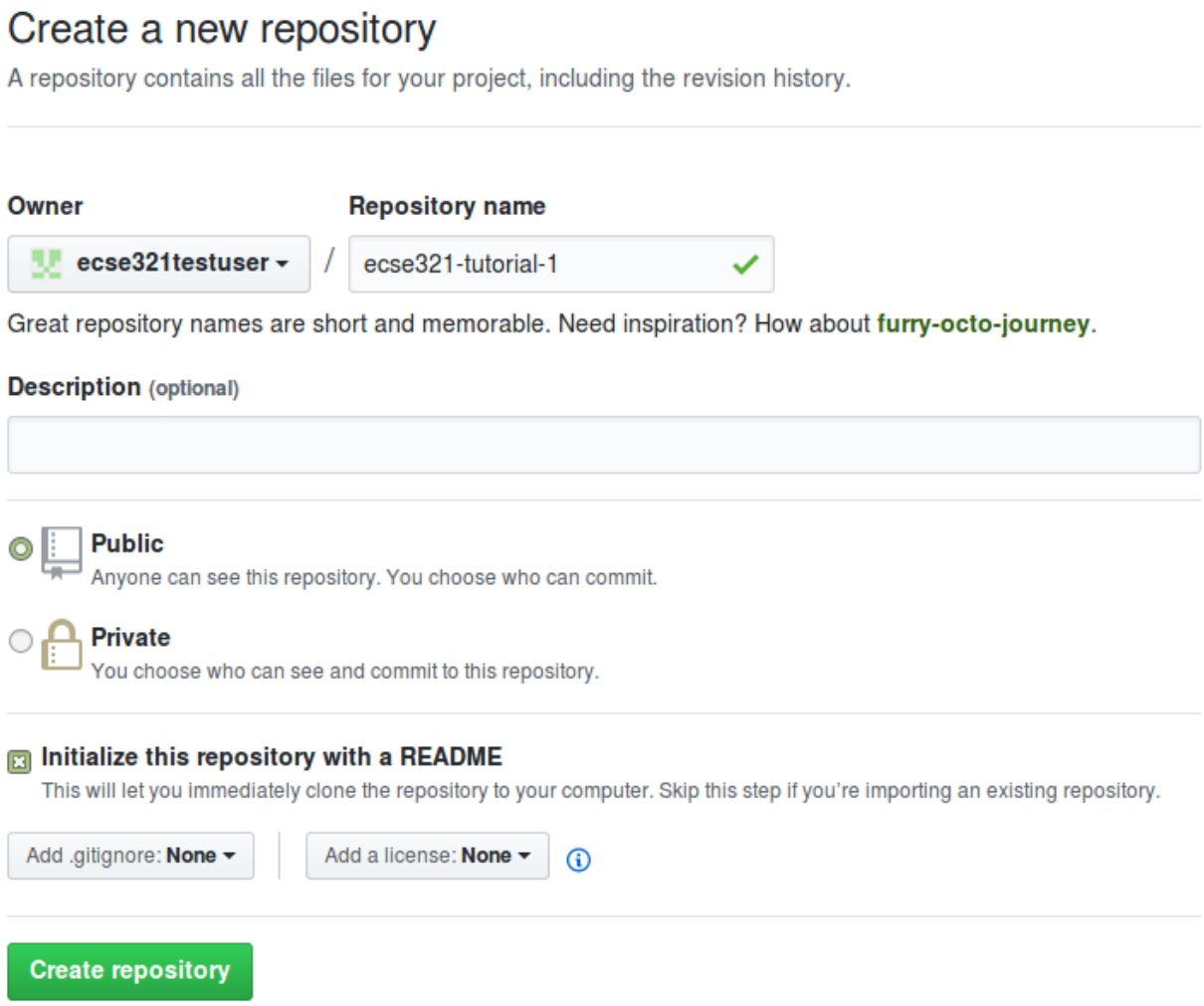


1.2. Project Management Tools for Agile Development

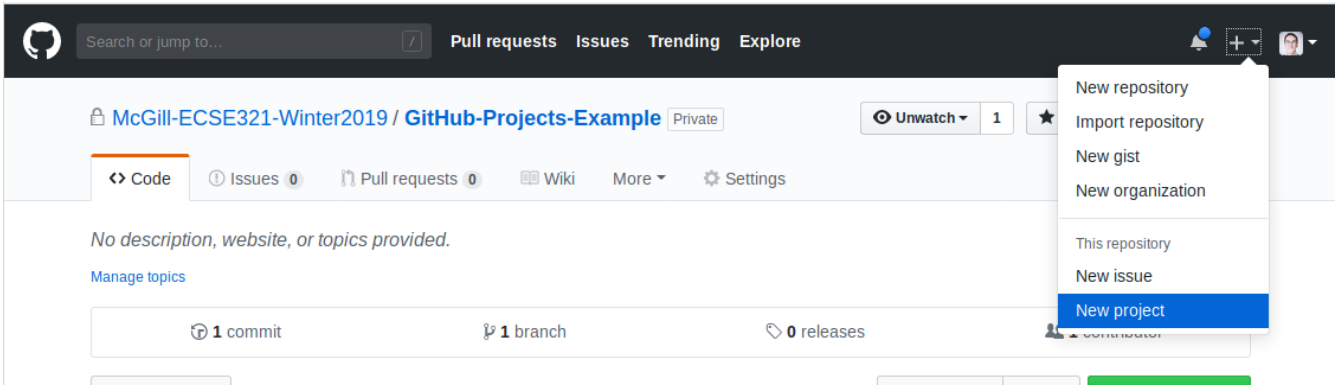
1.2.1. GitHub Projects

First, we create a new repository under everyone's own account to demonstrate the basic features of "GitHub Projects".

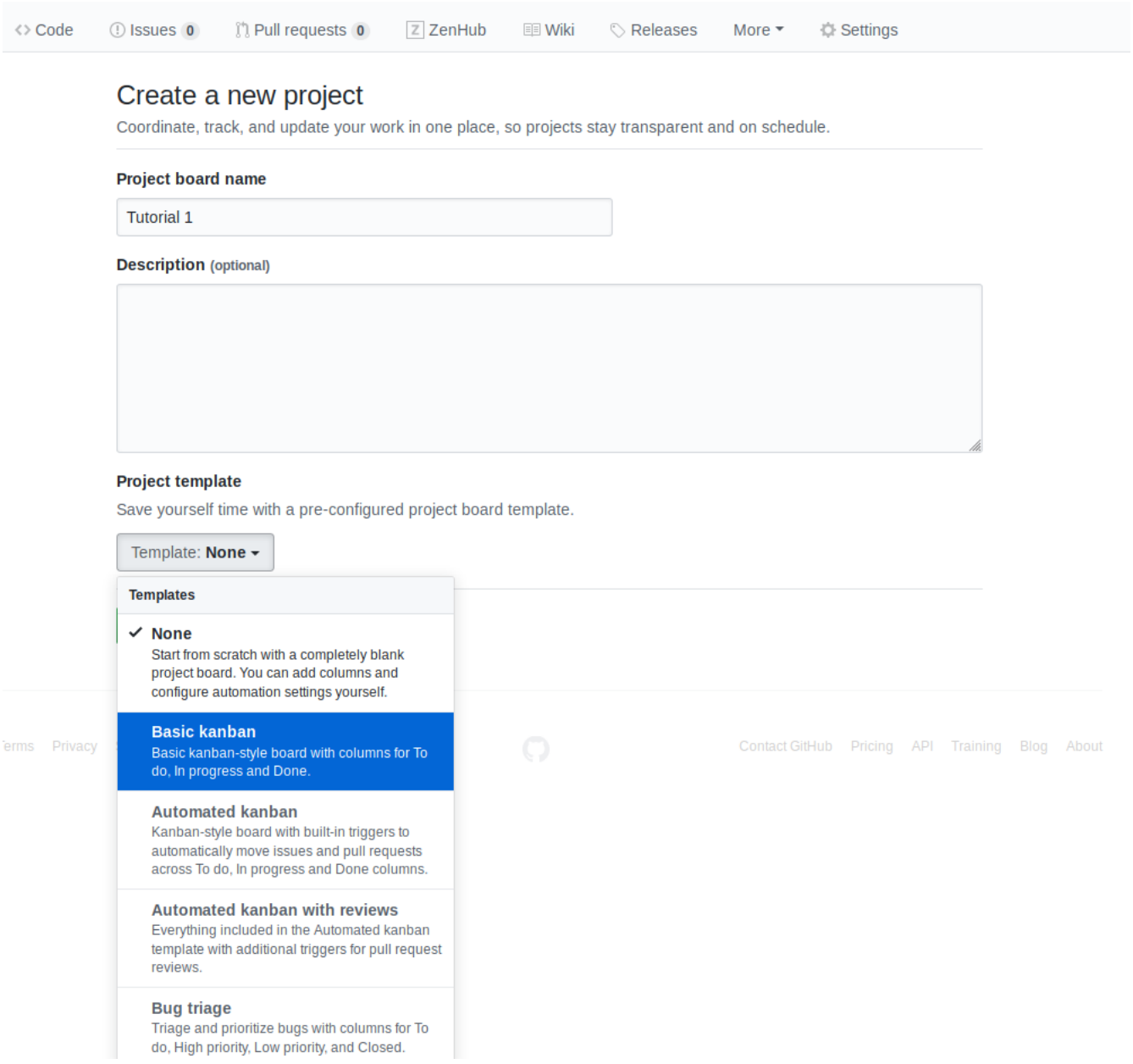
1. Visit https://github.com/ then click on *New repository* (green buttom on the right).
2. Set your user as the owner of the repository.
3. Give a name for the repository (e.g., ecse321-tutorial-1), leave it *public*, then check *Initialize this repository with a README*. Click on *Create repository* afterwards. At this point the remote repository is ready to use.



Once the repository is ready, associate a new GitHub Project and see how their features work. Create a project:

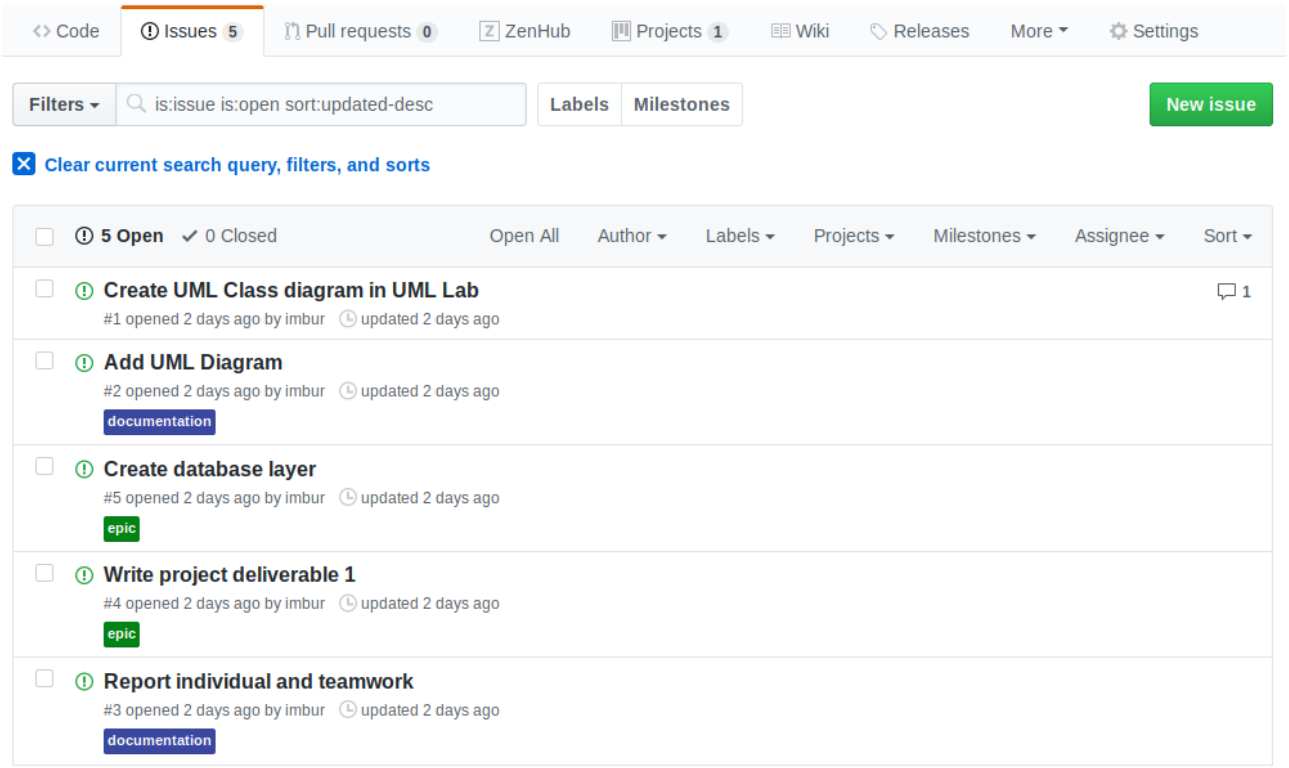


Select Basic Kanban project style:

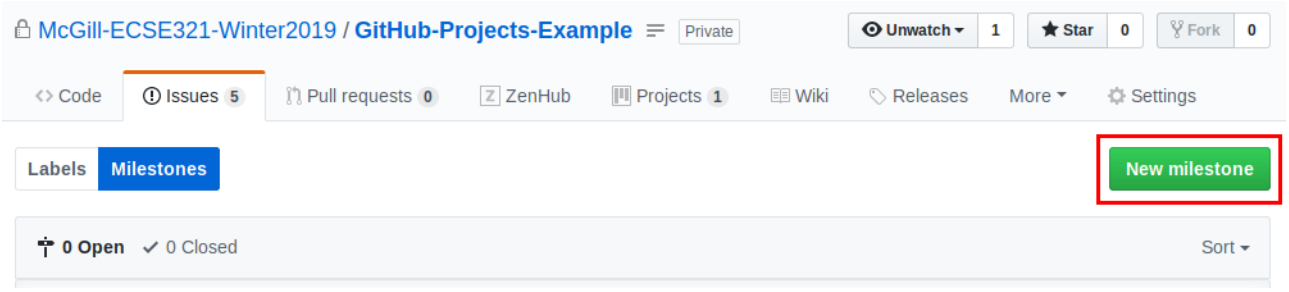


Tasks to complete:

1. Create a few issues to outline the tasks for the first deliverable. Assign them appropriate labels and add yourself as the assignee!



2. Create a milestone for the issues.



3. Create cards from the issues on the project board.
4. See how GitHub track the project progress as you move the cards from the different columns.

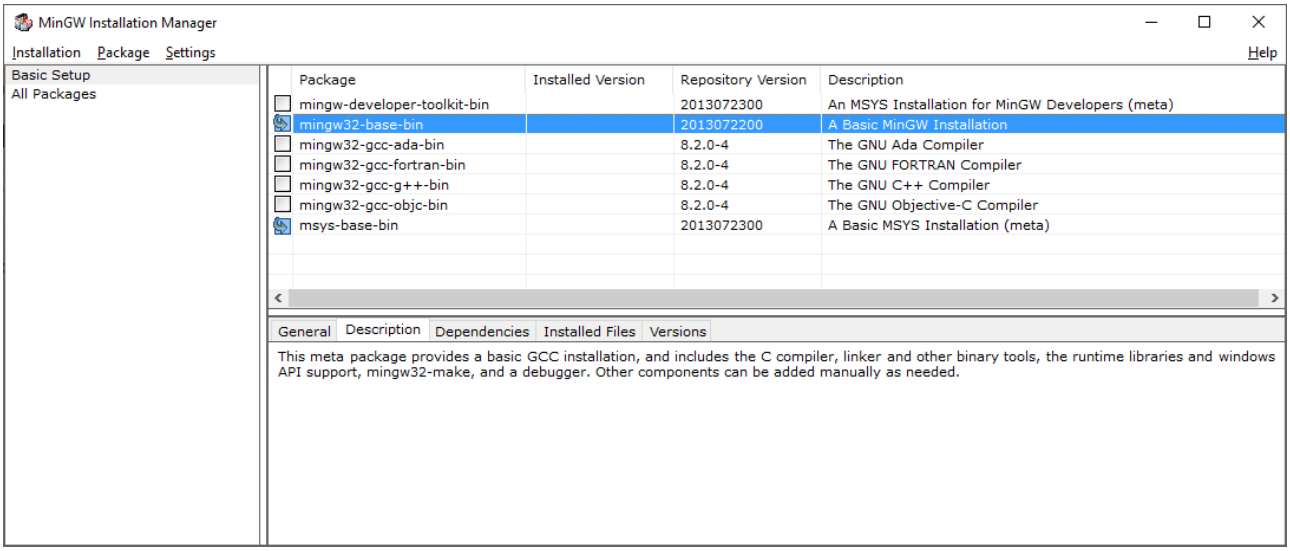
1.3. Command Line Basics

This section shows a few handy terminal commands.

1.3.1. Windows prerequisites

This step can be skipped if you are using MacOS or Linux. However, if you are using Windows, you need to have a terminal that supports the execution of basic Linux commands. Such programs are Git Bash or MinGW, for example. You can find below a few helper steps to get MinGW running on your system.

1. Get the MinGW installer from here
2. Install it to wherever you like, the default installation folder is *C:\MinGW*
3. Once the setup finishes, open the MinGW Installation Manager
4. Select the two packages for installation as shown in the figure below



5. Click on *Installation/Apply Changes*. This will take a few moments to fetch and install the required packages.
6. You can open a terminal window by running the executable *C:\MinGW\msys\1.0\bin\bash.exe*

1.3.2. Basic file system operaions

1. Open a terminal, and try the following commands:
   - `pwd`: prints the present working directory
     Example:

     ```
     $ pwd
     /home/ecse321
     ```

   - `ls`: lists the content of a given folder
     Example:

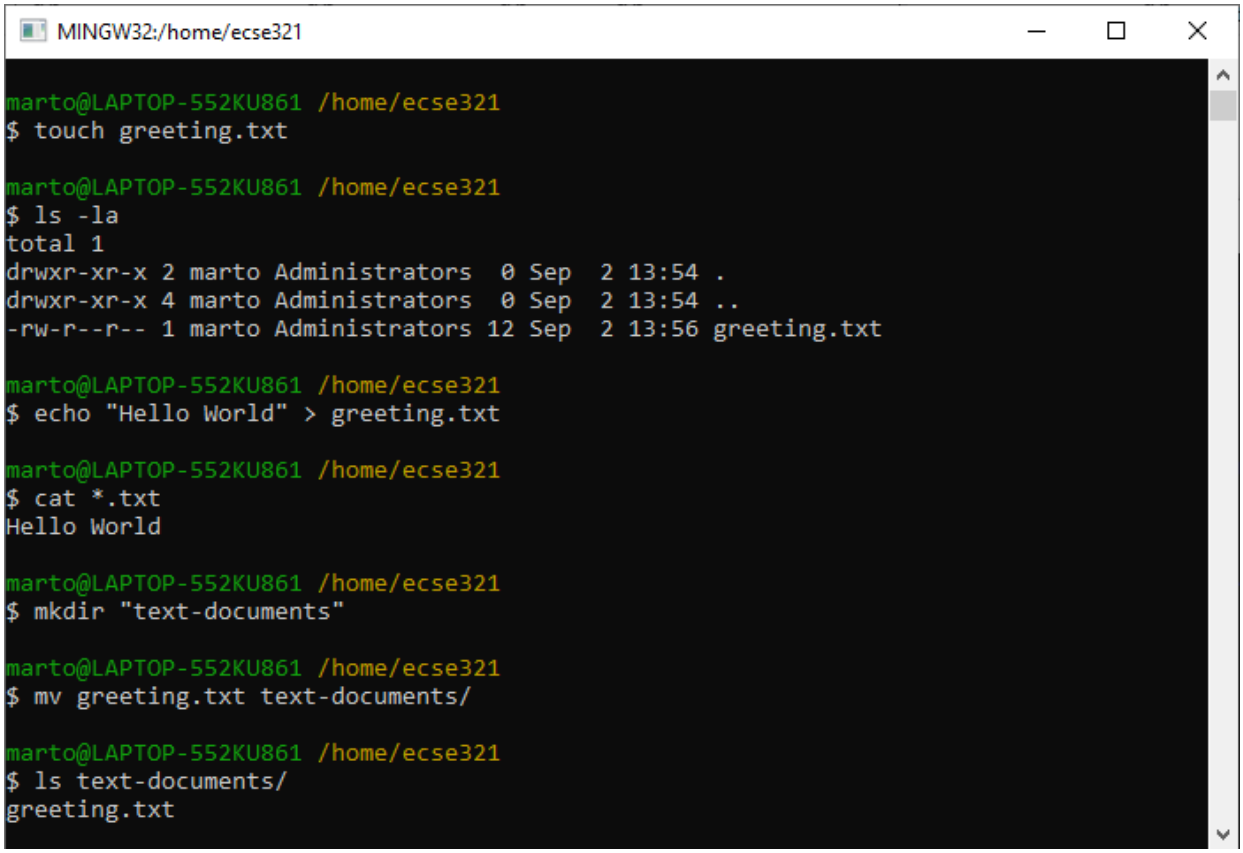
```
$ ls /home
ecse321 guest-user admin
```

- `cd`: navigates the file system
  Example:

```
$ cd ..
$ pwd
/home
$ cd ecse321
$ pwd
/home/ecse321
```

> **NOTE** The following steps will include images that illustrate the commands and their output to prevent easy copy-paste. Sorry! :)

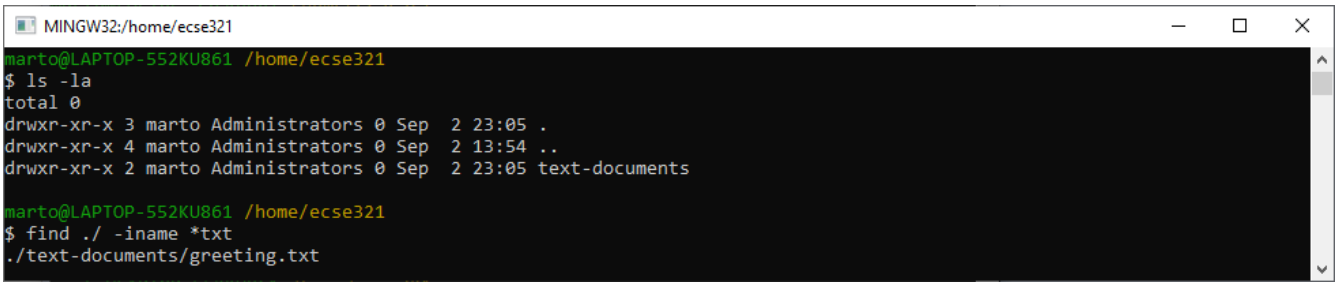
2. Creating files and reading/writing their contents
   - `touch`: creates a file
   - `mkdir`: creates a directory
   - `mv`: moves a file (or directory) from its current location to a target location
   - `echo`: prints a string
   - `cat`: prints the contents of a file
     Example:



1.3.3. Finding files

The versatile `find` command allows us to find files based on given criteria. Take look at its manual page with `man find`!

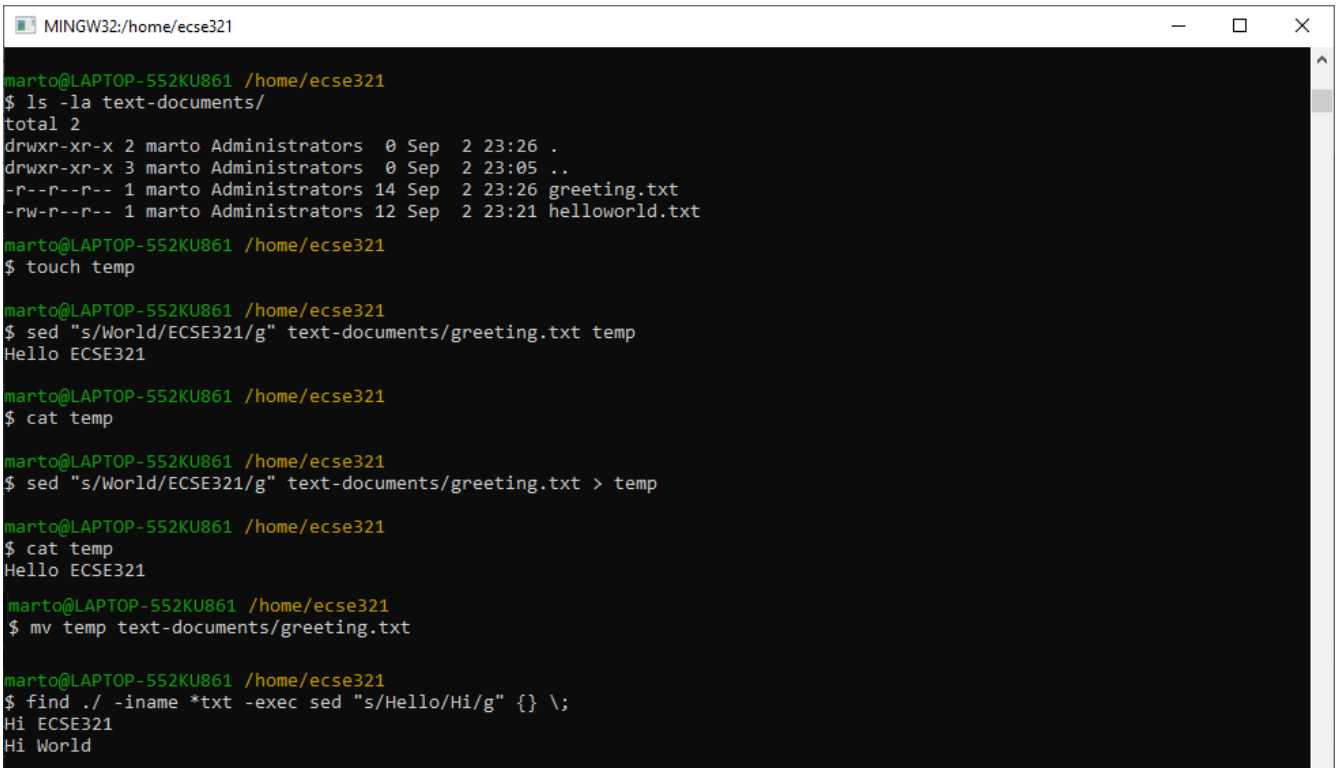
Example:



1.3.4. Batch file operations

- `sed`: stream editor; changes a given string to a replacement

Combining `find` with an additional command (e.g., `sed`) can greatly speed up your repetitive tasks. Example:



NOTE	The file *helloworld.txt* in the example is initially a copy of *greeting.txt*.

1.3.5. Some additional useful commands

- `rm`: removes a file
- `cp -r`: copies a directory recursively with its contents
- `rmdir`: remove an empty directory

- `rm -rf`: force to recursively delete a directory (or file) and all its contents
- `nano`: an easy-to-use text editor (not available by default in MinGW)
- `grep`: finds matches for a string in a given stream of characters
- `ag`: takes a string as argument and searches through the contents of files recursively to find matches of the given string (this tool is included in the *silversearcher-ag* package)

1.4. Git and GitHub

1.4.1. Installing Git

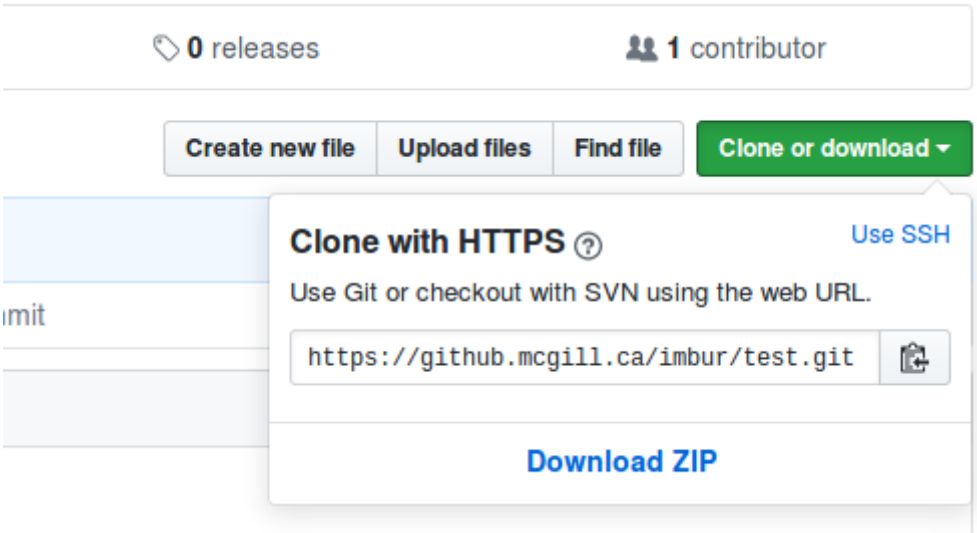
Install the Git version control system (VCS) from https://git-scm.com/downloads.

1.4.2. Creating a remote git repository on GitHub

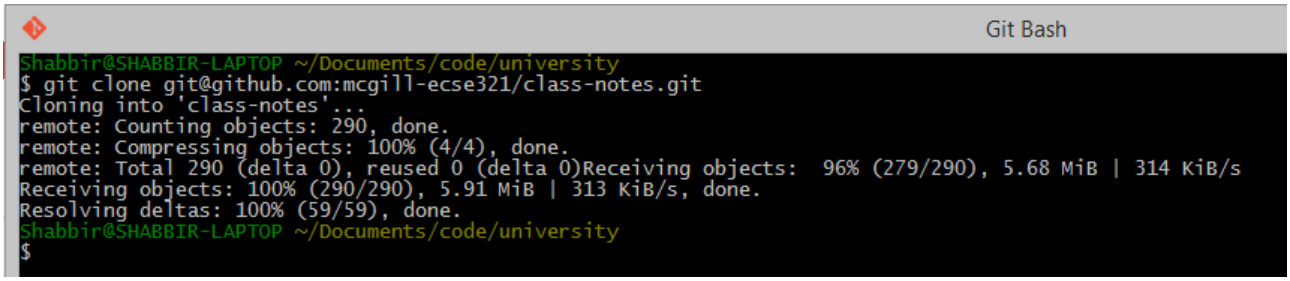
1. Go to https://github.com/new
2. Set *test* as the name of the repository
3. Check the checkbox *Initialize this repository with a README*
4. Click on create repository

1.4.3. Cloning to a local repository

1. Open up a terminal (Git bash on Windows).
2. Navigate to the designated target directory (it is typical to use the `git` folder within the home directory for storing Git repositories, e.g., `cd /home/username/git`).
3. Using a Git client, clone this newly created *test* repository to your computer. First, get the repository URL (use HTTPS for now).



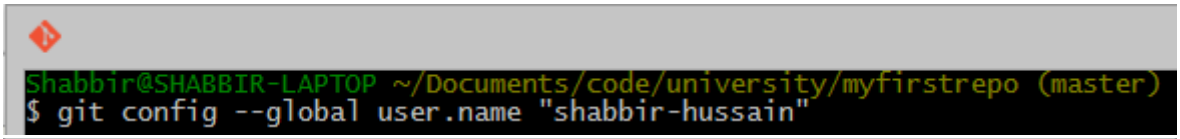
Then, issue `git clone https://url/of/the/repository.git`
You should get an output similar to this:

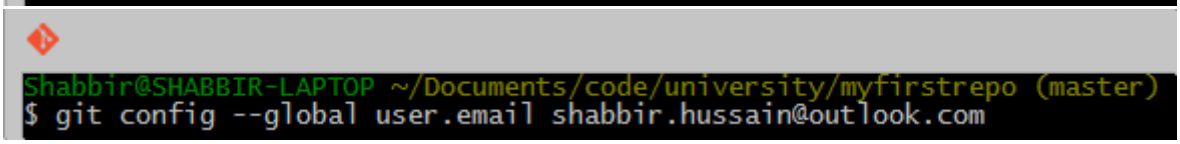


4. Verify the contents of the *working copy* of the repository by `ls -la ./test`. The *.git* folder holds version information and history for the repository, while the *README.md* is an auto-generated text file by GitHub.

1.4.4. Git basics

1. Open up a terminal and configure username and email address. These are needed to identify the author of the different changes.

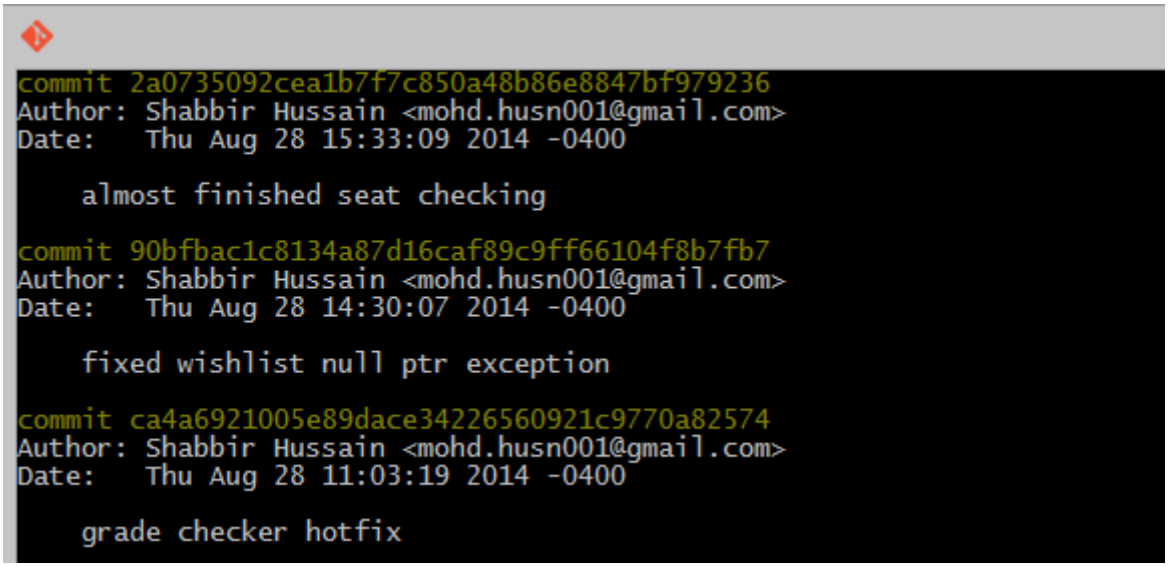




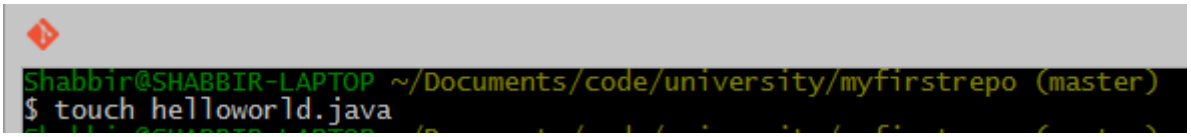
Glossary — Part 1:

   - **Git** is your version control software
   - **GitHub** hosts your repositories
   - A **repository** is a collection of files and their history
   - A **commit** is a saved state of the repository

2. Enter the working directory, then check the history by issuing `git log`. Example output:



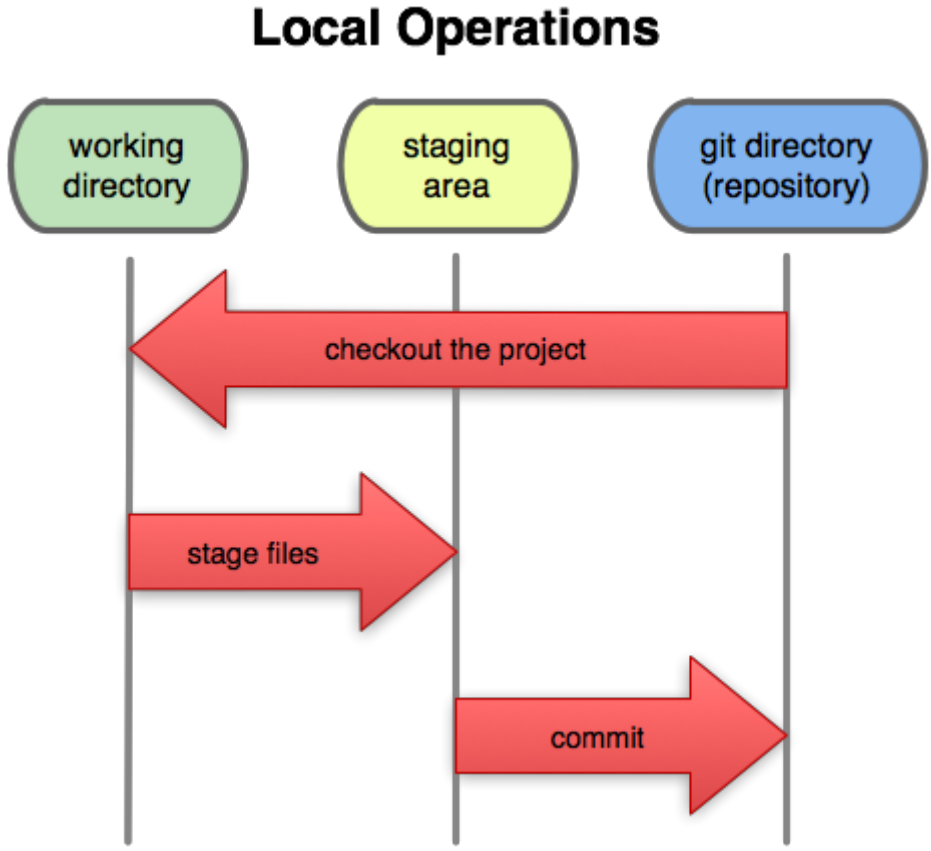
3. Adding and commiting a file: use the `git add` and `git commit` commands.



```
Shabbir@SHABBIR-LAPTOP ~/Documents/code/university/myfirstrepo (master)
$ git add helloworld.java
```

```
Shabbir@SHABBIR-LAPTOP ~/Documents/code/university/myfirstrepo (master)
$ git commit -m 'added hello world file to the project'
[master (root-commit) f4a1ddc] added hello world file to the project
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 helloworld.java
```

The effect of these commands are explained on the figure below:



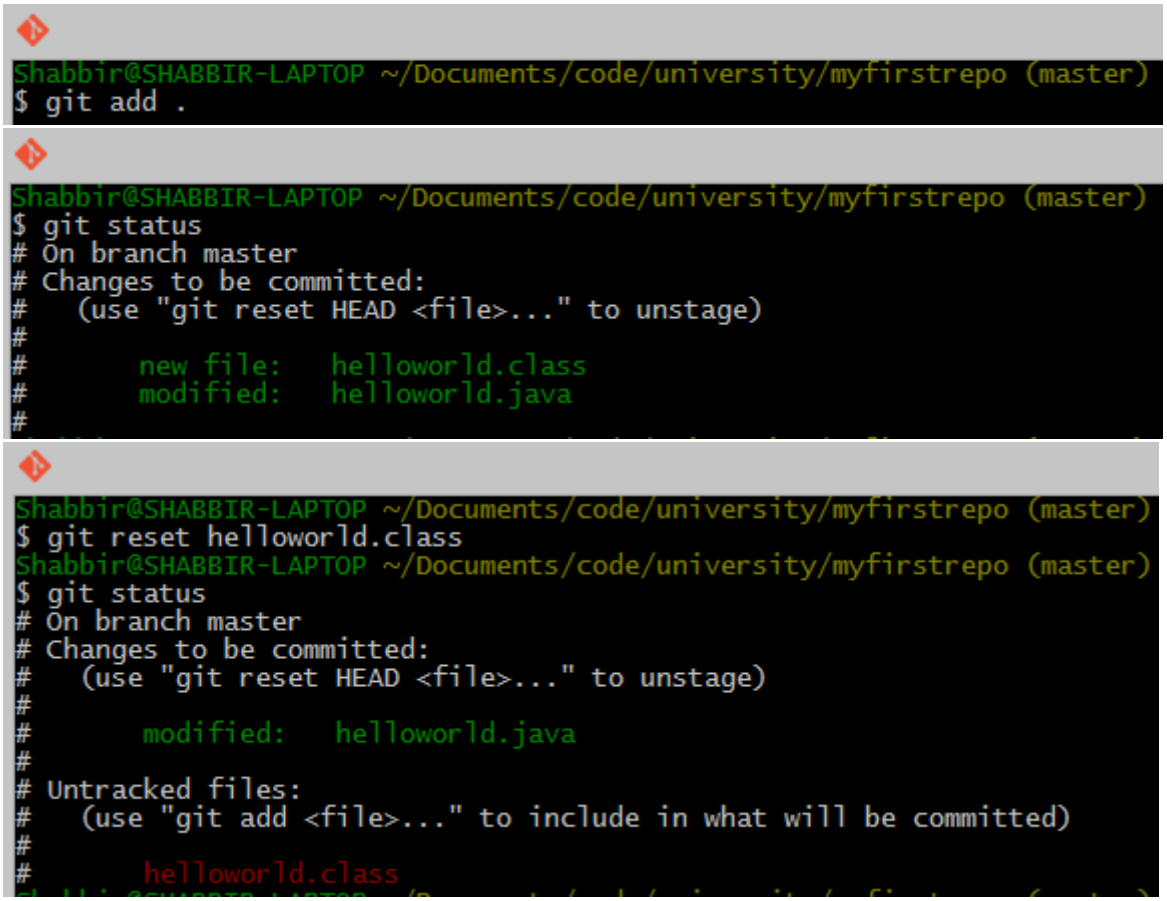
Glossary — Part 2:

- **Working Directory**: files being worked on right now
- **Staging area**: files ready to be committed
- **Repository**: A collection of commits

4. Checking current status is done with `git status`.

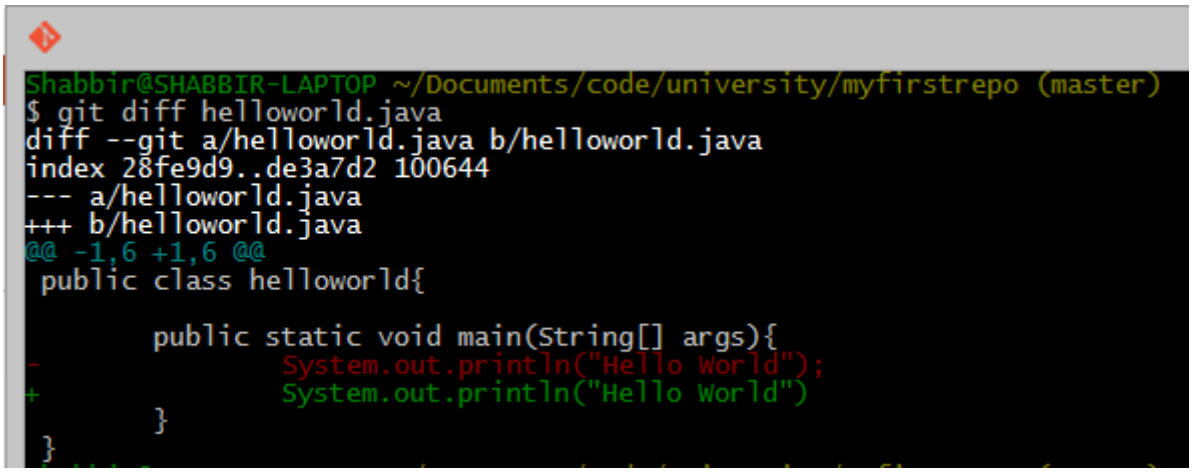
```
Shabbir@SHABBIR-LAPTOP ~/Documents/code/university/myfirstrepo (master)
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   helloworld.java
#
no changes added to commit (use "git add" and/or "git commit -a")
```

5. Staging and unstaging files: use `git add` to add and `git reset` to remove files from the staging area.

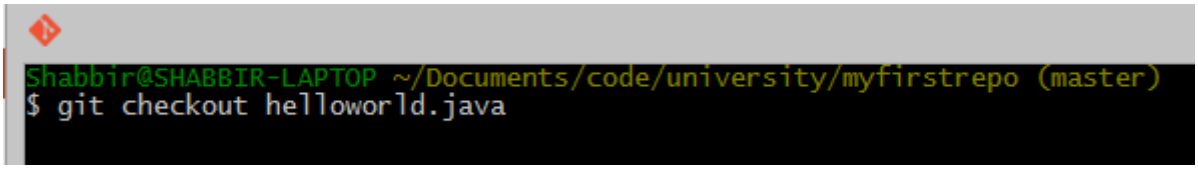


CAUTION	Only staged files will be included in the next commit.

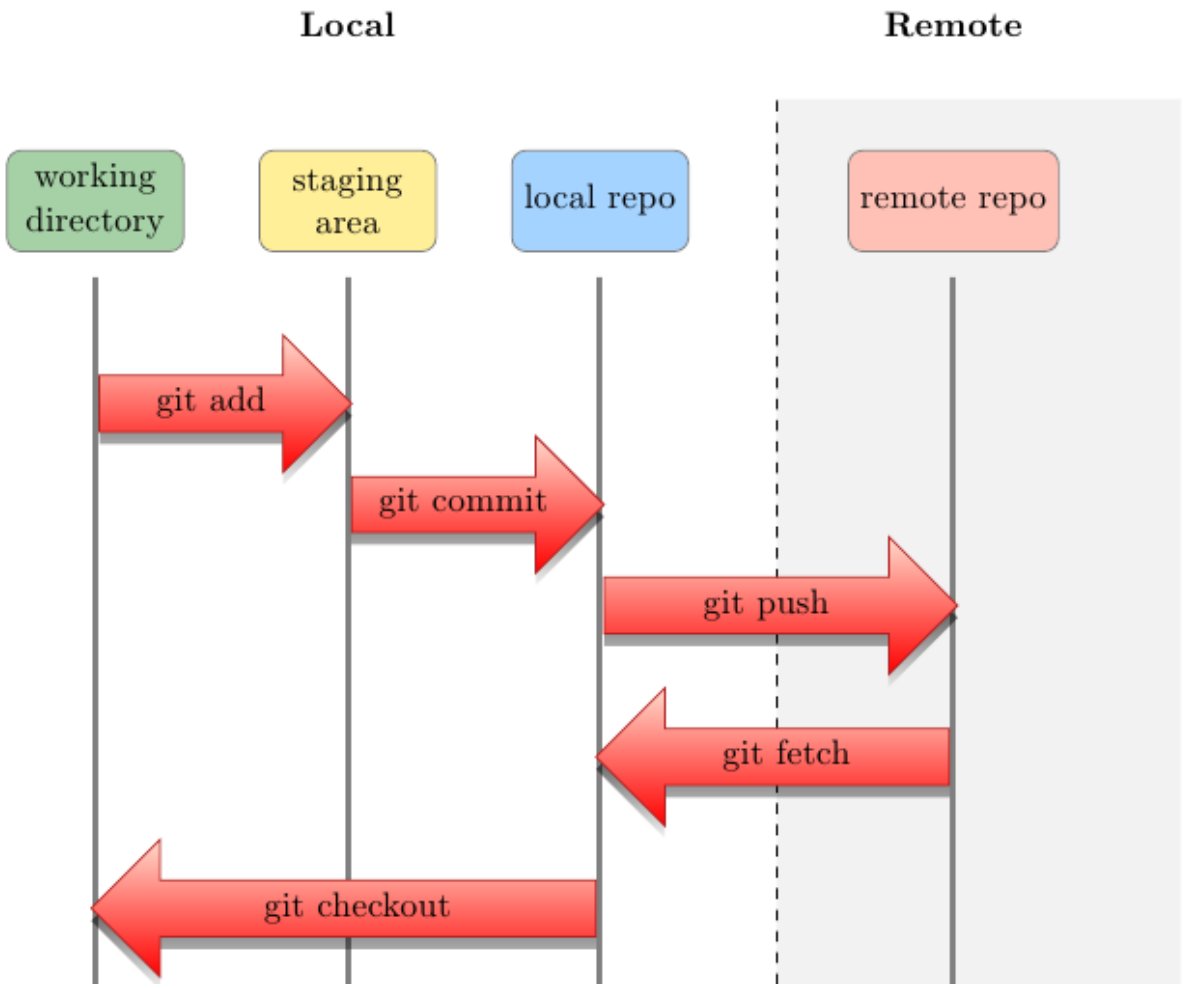
6. To display detailed changes in unstaged files use `git diff`, while use `git diff --staged` to show changes within files staged for commit.



7. Reverting to a previous version is done using `git checkout`.

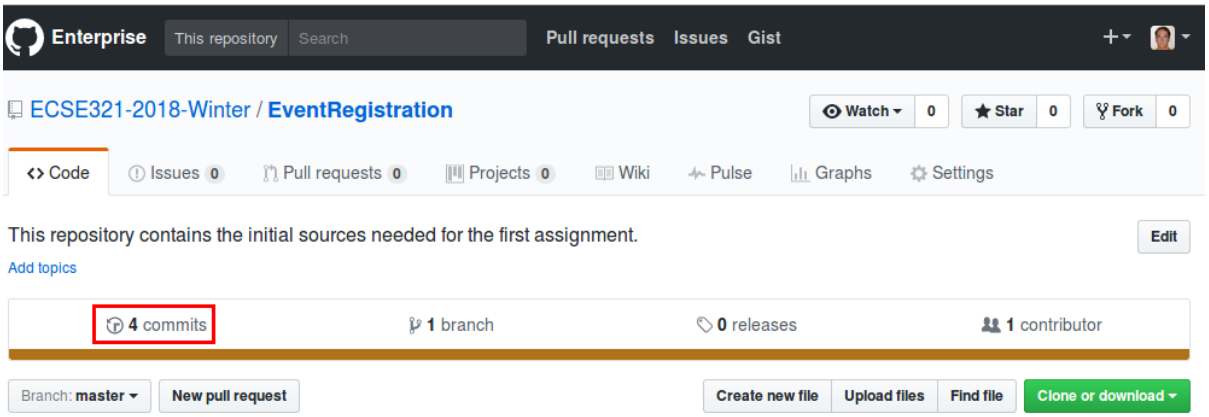


8. The commands `git pull` (or the `git fetch` + `git rebase` combination) and `git push` are used to synchronize local and remote repositories.

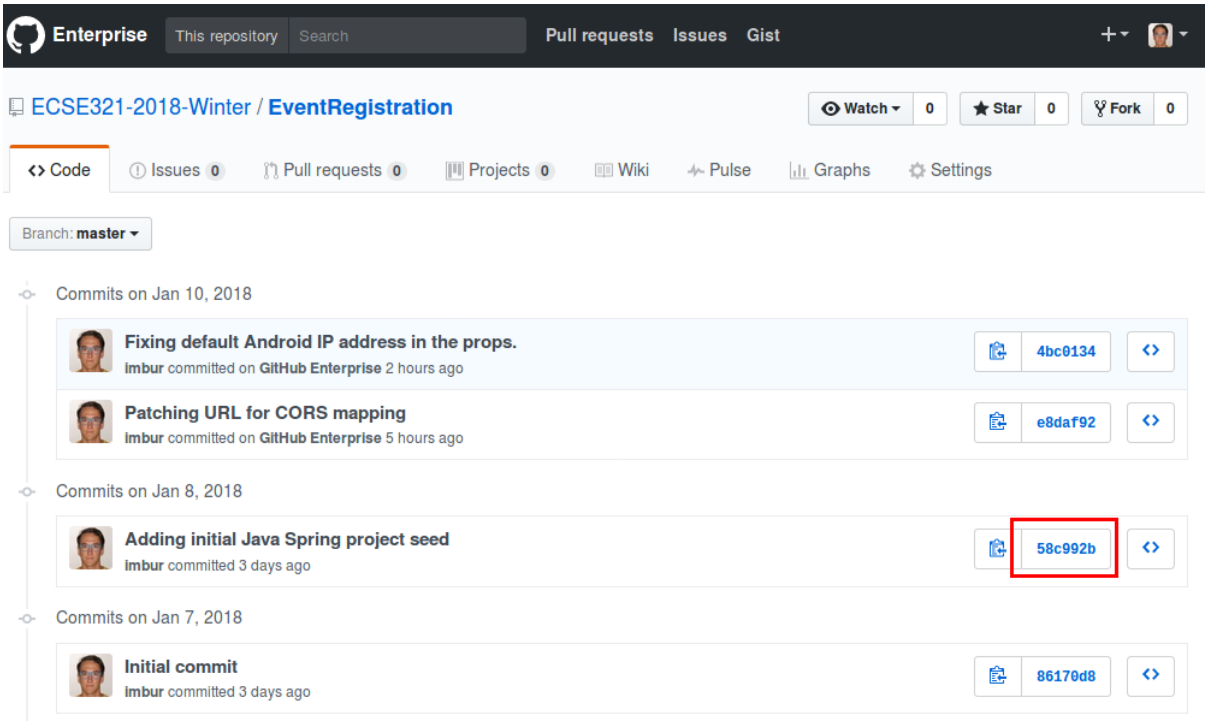


1.4.5. Browsing commit history on GitHub

1. You can browse pushed commits in the remote repository online using GitHub. You can select the *commits* menu for a repository.



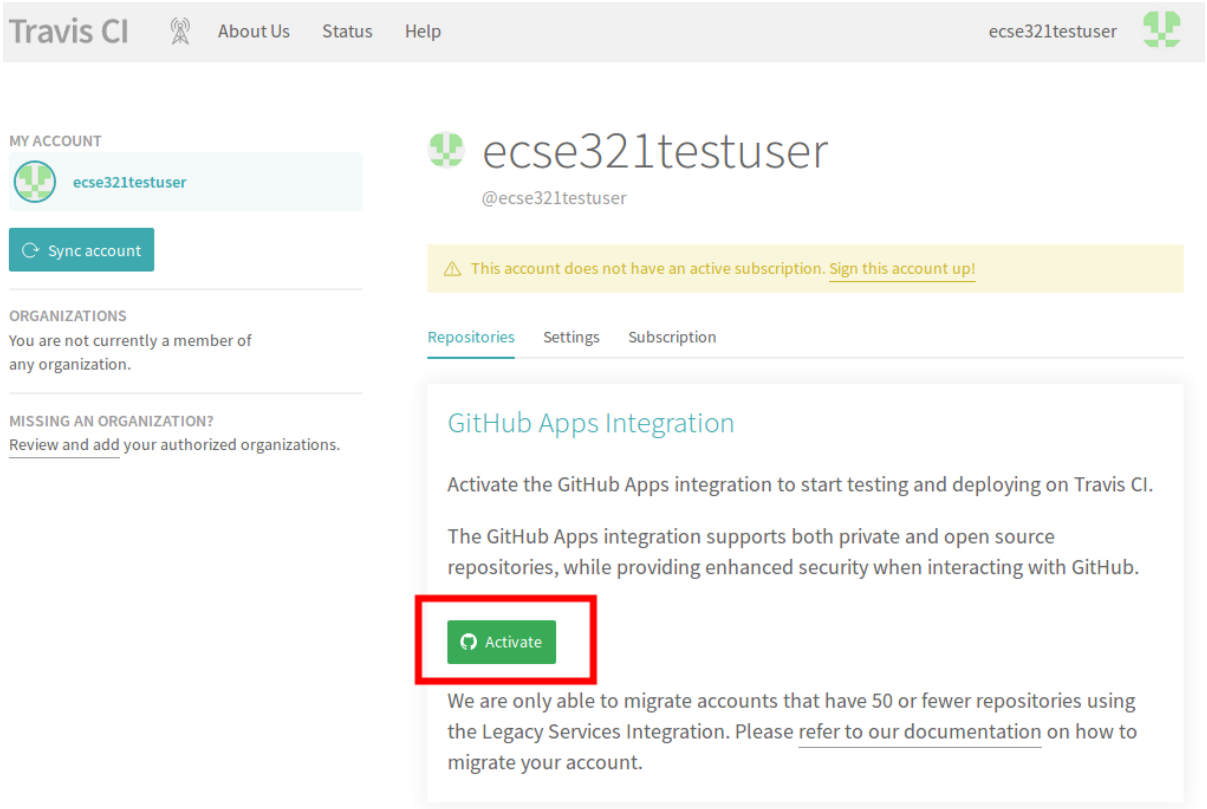
To get a link for a specific commit, click on the button with the first few characters of the hash of the commit.



The source for most of the images in the Git documentation: https://github.com/shabbir-hussain/ecse321tutorials/blob/master/01-githubTutorial1.pptx

1.5. Travis CI

1. Go to https://travis-ci.com/, click on Sign up with GitHub.
2. Click on the green authorize button at the bottom of the page.
3. Activate Travis-CI on your GitHub account



4. Select the repositories you want to build with Travis (make sure to include your repository that you created for this tutorial). You can modify this setting anytime later as well.
5. In your working copy of your repository, create a default Gradle java project.
   - Make sure you have Gradle installed (`gradle --version`).
   - Issue `gradle init --type java-library`
   - Add a `.gitignore` to ignore generated resources by Git:

     ```
     .gradle/
     build/
     ```

   - Make sure your application is compiling by running `gradle build`
6. Create a file called `.travis.yml`:

   ```
   language: java
   script:
   - gradle build
   ```

7. Commit and push your work. If everything is set up correctly, the build should trigger and

Travis should run your build using Gradle.

1.6. Gradle: A Build Framework

1.6.1. Example Gradle application

This section focuses on writing a Gradle (https://gradle.org/) build script that builds a single Gradle project referred to as *Computation*. The source code and tests for a Java application is available here: Computation.zip (src and tst folders). It is your job to create a folder called *Computation*, move sources and tests into that folder, and produce the Gradle build script *build.gradle* within this folder to automate the software build process for this project.

First, open a terminal, and ensure you have the newes version of Gradle (ver. 5.0+) installed with `gradle --version`.

Follow the steps below and add the snippets listed here to `build.gradle`, one after the other:

1. Create the following folder structure and a new *build.gradle* (empty) file within the *Computation* folder:

```
Computation
├─── build.gradle
└─── src
    ├─── main
    │    └─── java
    │         ├─── application
    │         │    └─── CompApp.java
    │         ├─── computation
    │         │    └─── Computation.java
    │         └─── view
    │              └─── ComputationPage.java
    └─── test
         └─── java
              └─── computation
                   ├─── AllTests.java
                   ├─── ComputationTestAddSubstract.java
                   └─── ComputationTestDivideMultiply.java
```

1. Add the `java` and the `application` plugins to the build configuration script *build.gradle*.

```
apply plugin: 'java'
// This plugin has a predefined 'run' task that we can reuse to use Gradle to
execute our application
apply plugin: 'application'
```

2. Add JUnit libraries to the `dependencies` section.

```
repositories {
    mavenCentral()
}
dependencies {
    testImplementation "junit:junit:4.12"
}
```

3. Add and describe a new task `compile(type: JavaCompile)` to specify all source files (both application and test) and set the *build/bin* as destination dir to put all compiled class files in.

```
task compile(type: JavaCompile) {
  classpath = sourceSets.main.compileClasspath
  classpath += sourceSets.test.runtimeClasspath
  sourceSets.test.java.outputDir = file('build/bin')
  sourceSets.main.java.outputDir = file('build/bin')
}
```

NOTE One can specify source sets and their variables the following way:

```
/*
 * specifying sourceSets is not necessary in this case, since
 * we are applying the default folder structure assumed by Gradle
 */
sourceSets {
  main {
    java { srcDir 'src/main/java' }
  }
  test {
    java { srcDir 'src/test/java'}
  }
}
```

4. Specify the main class and run the application.

```
mainClassName='application.CompApp'
```

In the command line issue `gradle run`

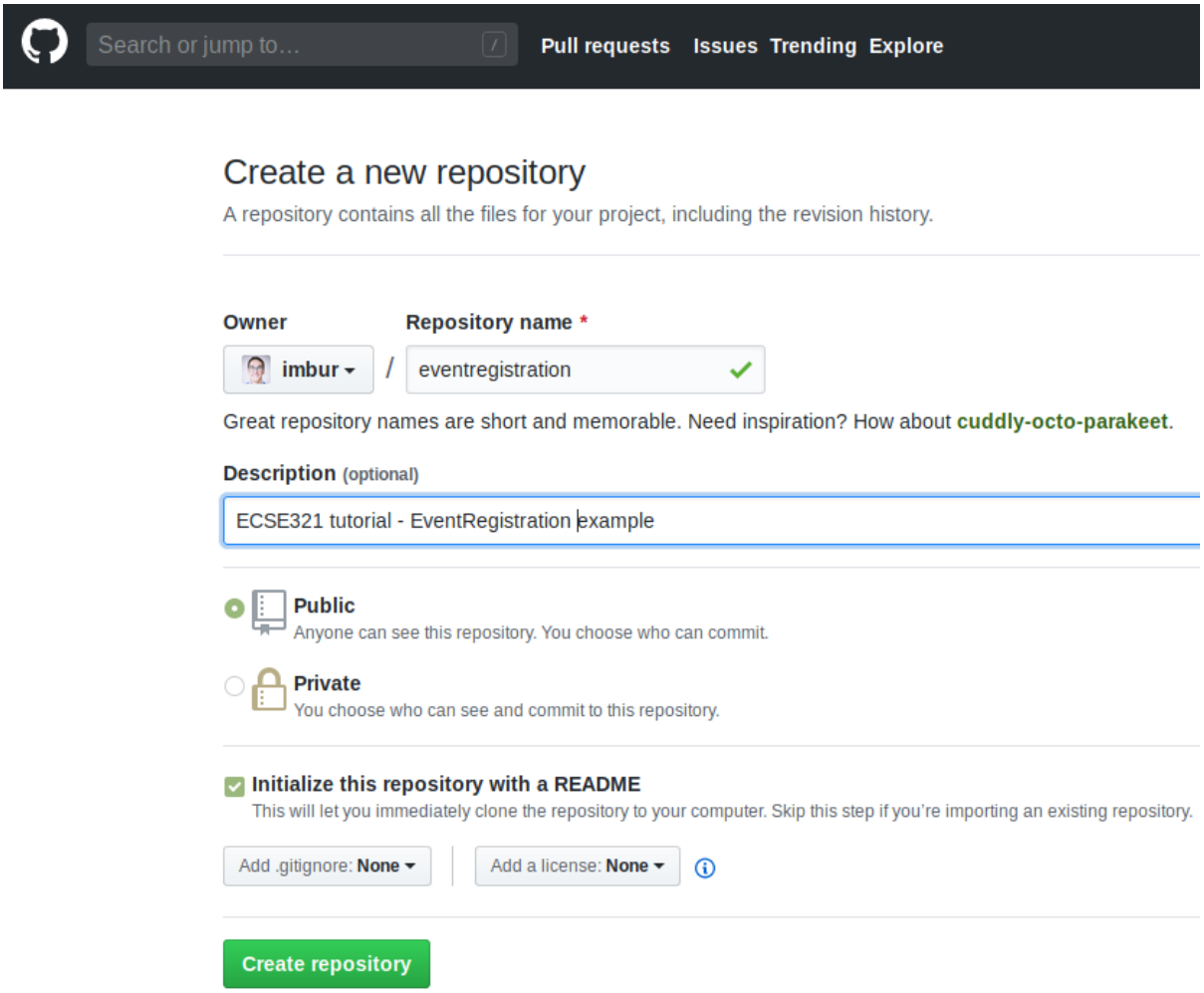
5. Describe the `jar` Gradle task (defined by the `java` plugin) to produce an executable jar file into `distributable/`.

```
jar {
  destinationDir=file('distributable')
  manifest {
    // It is smart to reuse the name of the main class variable instead of
hardcoding it
    attributes "Main-Class": "$mainClassName"
  }
}
```

NOTE The `settings.gradle` and its usage is to be shown later.

1.6.2. Setting up a Spring/Spring Boot backend app with Gradle

1. Install the Spring Boot CLI
2. Create a new repository under your account on GitHub for an example application that we are going to develop throughout the semester. Name the repository **eventregistration**. See more on the specification of the application functionality later.



3. Clone it somewhere on your disk. We assume you cloned it to *~/git/eventregistration*.
4. Navigate to that folder in the terminal: `cd ~/git/eventregistration`.
5. Create a project for the backend application using Spring Boot CLI in this repository.

```
spring init \
 --build=gradle \
 --java-version=1.8 \
 --package=ca.mcgill.ecse321.eventregistration \
 --name=EventRegistration \
 --dependencies=web,data-jpa,postgresql \
 EventRegistration-Backend
```

NOTE	Backslashes in this snippet indicate linebreaks in this one liner command typed in the terminal. You can select and copy-paste this snippet as-is.

6. Navigate to the *EventRegistration-Backend* folder

7. For future use, locate the *application.properties* file in the *src/* folder and add the following content:

```
server.port=${PORT:8080}

spring.jpa.properties.hibernate.temp.use_jdbc_metadata_defaults = false
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
```

NOTE	Source: https://vkuzel.com/spring-boot-jpa-hibernate-atomikos-postgresql-exception

NOTE	It may be the case that the `PostgreSQLDialect` needs to be changed for certain database instances (e.g., to `PostgreSQL9Dialect`).

8. Locate the Java file containing the main application class (`EventRegistrationApplication.java`) and add the following content

```
package ca.mcgill.ecse321.eventregistration;

import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
@SpringBootApplication
public class EventRegistrationApplication {

  public static void main(String[] args) {
    SpringApplication.run(EventRegistrationApplication.class, args);
  }

  @RequestMapping("/")
  public String greeting(){
    return "Hello world!";
  }

}
```

9. Verify that it builds with `gradle build -xtest`.
10. Commit and push the files of the new Spring project.

```
git add .
git status #verify the files that are staged for commit
git commit -m "Initial commit of the backend application"
git push
```

1.7. Heroku

1.7.1. Preparations

1. Sign up/log in on Heroku by visiting https://www.heroku.com/.
2. Install the command line client for Heroku: Heroku CLI

> **NOTE** The Travis client might also be useful at later stages of the course, you can install it from here: Travis CLI

3. Log in to Heroku CLI by opening a terminal an typing: `heroku login`.

1.7.2. Creating a Heroku app

We are creating a Heroku application and deploying the *Hello world!* Spring example. Additionally, the steps below will make it possible to store multiple different applications in the same git repository and deploy them individually to Heroku. Steps will be shown through the example EventRegistration application, and should be adapted in the course project.

> **NOTE** All actions described here for configuring Heroku applications using the Heroku CLI could also be done via the web UI.

1. Once you are logged in with the Heroku-CLI, create a new Heroku application: in the root of the git repository of your repository (assumed to be *~/git/eventregistration*), issue `heroku create eventregistration-backend-<UNIQUE_ID> -n` to create an application named "eventregistration-backend-<UNIQUE_ID>".

> **NOTE** In Heroku, the application name should be unique Heroku-wise, that is, each application in Heroku's system should have a unique name. If you don't provide a name parameter for the command, Heroku will randomly generate one.

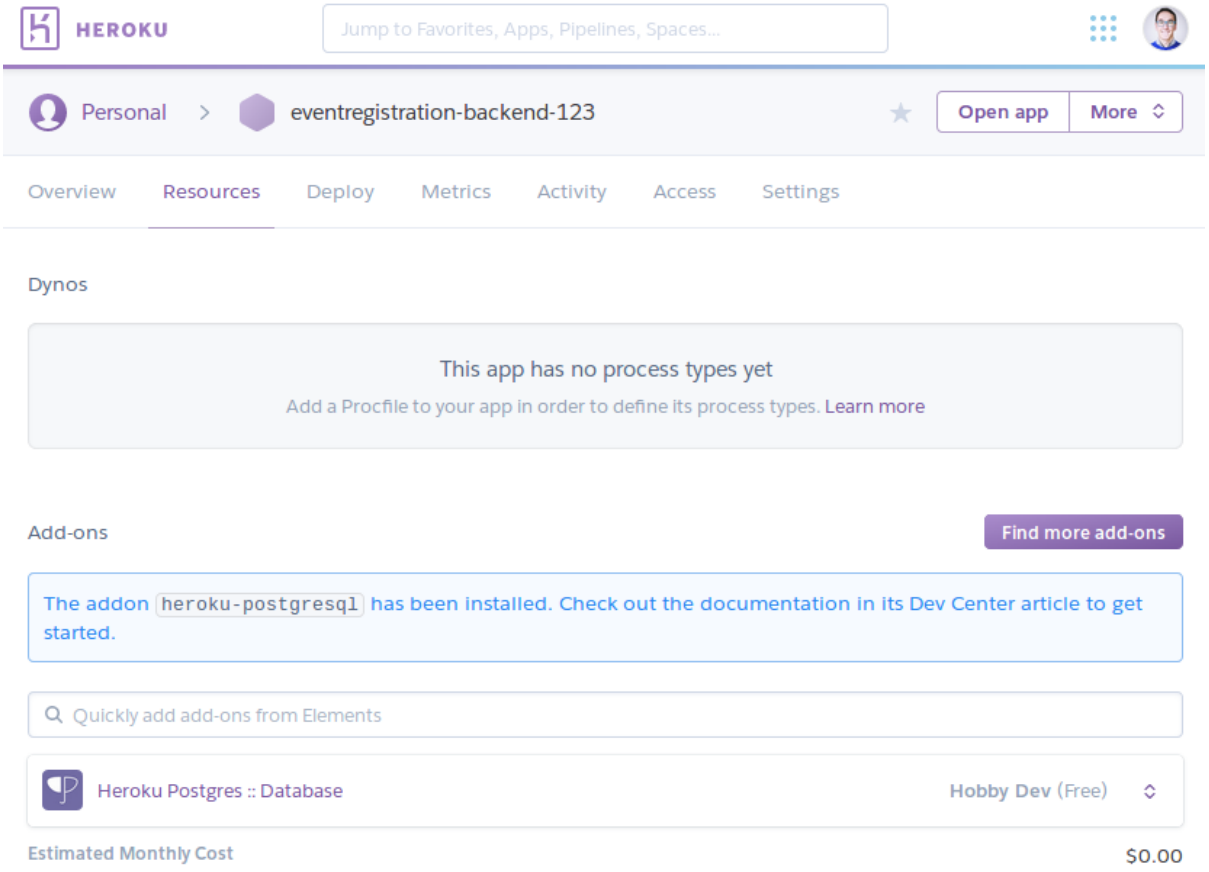
2. Add the multi procfile and Gradle buildpacks to the app.

```
heroku buildpacks:add -a eventregistration-backend-<UNIQUE_ID>
https://github.com/heroku/heroku-buildpack-multi-procfile
heroku buildpacks:add -a eventregistration-backend-<UNIQUE_ID> heroku/gradle
```

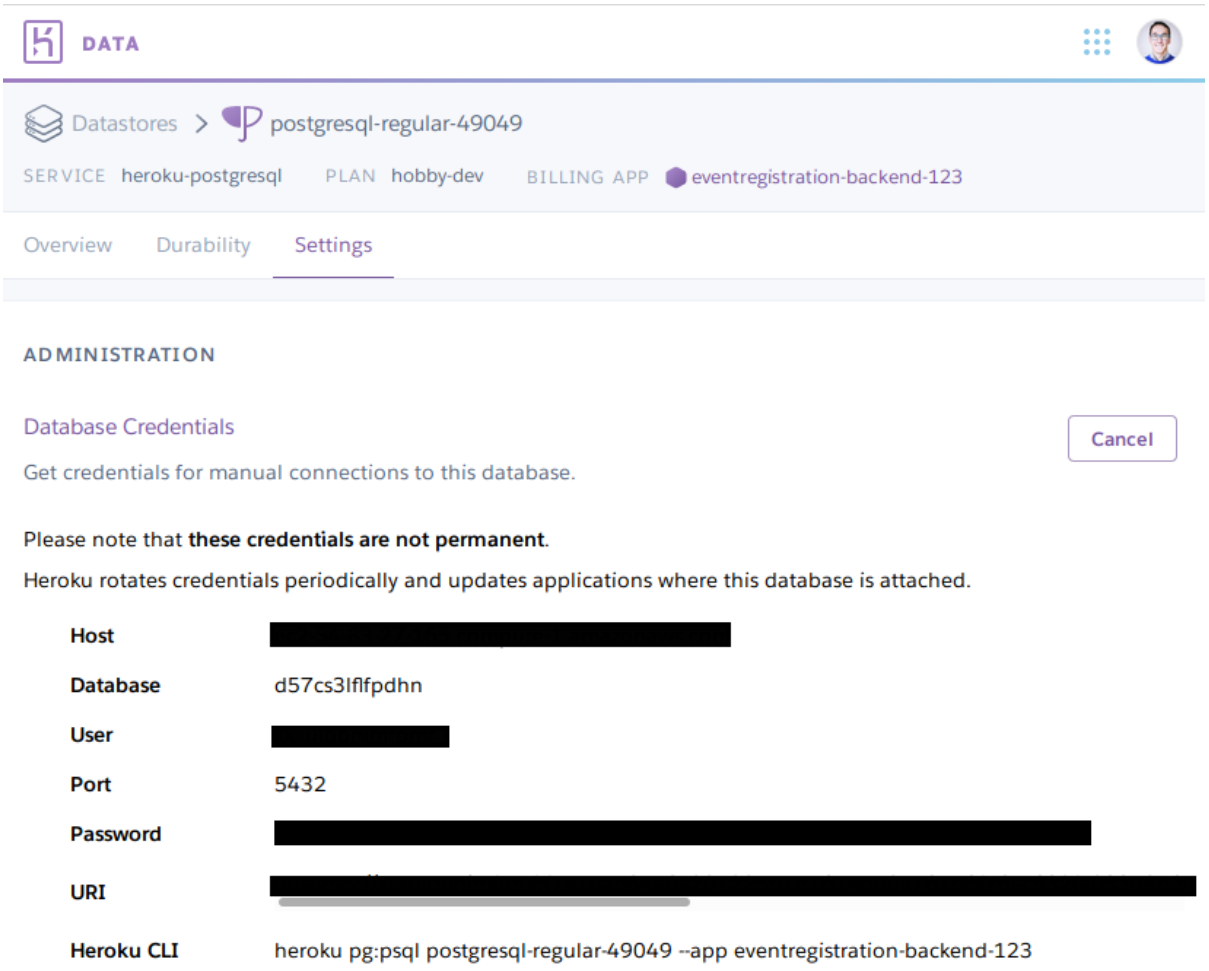
> **CAUTION** Order is important.

1.7.3. Adding a database to the application

1. Open the Heroku applications web page and go to *Resources*, then add the Heroku Postgres add-on.



2. Click the entry for Postgres within the list of add-ons, then go to *Settings*. You can see the database credentials there.



NOTE	The credentials are periodically updated and changed by Heroku, so make sure that you are using the actual credentials when manually connecting to the database. (E.g., during manual testing.)

1.7.4. Extending the build for the Heroku deployment environment

1. Before deploying, a top level *build.gradle* and *settings.gradle* need to be created in the root of the repository (i.e., in *~/git/eventregistration*)
   build.gradle:

   ```
   task stage () {
       dependsOn ':EventRegistration-Backend:assemble'
   }
   ```

   settings.gradle:

   ```
   include ':EventRegistration-Backend'
   ```

2. Generate the Gradle wrapper with the newest Gradle version

   ```
   gradle wrapper --gradle-version 5.6.2
   ```

3. Create a *.gitignore* file for the *.gradle* folder:
   .gitignore:

   ```
   .gradle/
   ```

4. Add all new files to git

   ```
   git add .
   git status #make sure that files in .gradle/ are not added
   ```

   Expected output for `git status`:

```
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitignore
    new file:   build.gradle
    new file:   gradle/wrapper/gradle-wrapper.jar
    new file:   gradle/wrapper/gradle-wrapper.properties
    new file:   gradlew
    new file:   gradlew.bat
    new file:   settings.gradle
```

Commit changes:

```
git commit -m "Adding Gradle wrapper"
```

1.7.5. Supply application-specific setting for Heroku

1. Within the *EventRegistration-Backend* folder, create a file called *Procfile* (**not** Procfile.txt, name it **exactly** Procfile) with the content:

```
web: java -jar EventRegistration-Backend/build/libs/EventRegistration-Backend-
0.0.1-SNAPSHOT.jar
```

2. Add the Procfile to a new commit
3. Configure the multi-procfile buildpack to find the Procfile:

```
heroku config:add PROCFILE=EventRegistration-Backend/Procfile --app
eventregistration-backend-<UNIQUE_ID>
```

1.7.6. Deploying the app

1. Obtain and copy the *Heroku Git URL*

```
heroku git:remote --app eventregistration-backend-<UNIQUE_ID> --remote backend-
heroku
```

Output:

```
set git remote backend-heroku to https://git.heroku.com/eventregistration-backend-
<UNIQUE_ID>.git
```

2. Verify that the `backend-heroku` remote is successfully added besides `origin` with `git remote -v`. Output:

```
backend-heroku  https://git.heroku.com/eventregistration-backend-123.git (fetch)
backend-heroku  https://git.heroku.com/eventregistration-backend-123.git (push)
origin  git@github.com:imbur/eventregistration.git (fetch)
origin  git@github.com:imbur/eventregistration.git (push)
```

3. Deploy your application with

```
git push backend-heroku master
```

> **NOTE** If it fails to build, make sure you try understanding the output. Typical issue: buildpacks are not added/are not in the right order.

4. Visit the link provided in the build output. It may take some time (even 30-60 seconds) for the server to answer the first HTTP request, so be patient!
5. Save your work to the GitHub repository, too: `git push origin master`
   Final layout of the files (only two directory levels are shown and hidden items are suppressed):

```
~/git/eventregistration
├──── build.gradle
├──── EventRegistration-Backend
│     ├──── build
│     │     ├──── classes
│     │     ├──── libs
│     │     ├──── resources
│     │     └──── tmp
│     ├──── build.gradle
│     ├──── gradle
│     │     └──── wrapper
│     ├──── gradlew
│     ├──── gradlew.bat
│     ├──── Procfile
│     ├──── settings.gradle
│     └──── src
│         ├──── main
│         └──── test
├──── gradle
│     └──── wrapper
│         ├──── gradle-wrapper.jar
│         └──── gradle-wrapper.properties
├──── gradlew
├──── gradlew.bat
├──── README.md
└──── settings.gradle
```

1.8. Domain modeling and code generation

1.8.1. Installing UML Lab

Go to the download page of UML Lab and install it on your machine. To activate it, use the licence key shared in the MyCourses announcement.

NOTE	By the time of the fourth tutorial, we may not have the license key ready. You should be able to work with a 30-day trial version in this case and activate your license later.

1.8.2. UML Lab project setup

NOTE	Once you start UML Lab, there are some useful tutorials that help you learn about the features of the modeling tool. Furthermore, there is an introduction on how to use and configure UML Lab among the resources of Rice University.

1. Create a new UML Lab Java project with the name `ca.mcgill.ecse321.eventregistration.model` with the default project settings.
2. Within the project, create a linked folder (Select Project → New Folder → Click Advanced Button → select "Link to alternate location (linked folder)" option) that points to the *src/main/java* folder of your `Eventregistration-Backend` project. Name the folder as *src-gen*. It will be used as the target for generating model code.
   [Linking folder]

CAUTION	Links to folders will not be versioned, so each team member needs to set this link individually after cloning the project.

3. Open the *ca.mcgill.ecse321.eventregistration.model.umlcd* diagram file by double clicking it. It is an empty diagram by default.
4. Click on the empty diagram editor canvas and open the *properties view* and configure code generation path.
   [Codegen output folder]
5. In the same *Properties view*, apply the *Direct* > *JPA1* code style.
   [Code style]

1.8.3. Domain modeling exercise: the Event Registration System

1. Using the *Palette* on the left hand side of the class diagram editor, create the following package structure and the `Person` class, and connect them with the *Containment* line. Once you save the diagram, the code should be generated to the *src-gen* folder (left part of the figure below).
   [Class Diagram Packages]

NOTE	If you disabled the automatic code generation on file save action, then you need to do *right click the diagram → generate code* manually.

2. Study the generated `Person` class in the `ca/mcgill/ecse321/eventregistration/model` package (folder)!
3. In the upcoming steps, we will use the `java.sql.Time` and `java.sql.Date` data types from the Java Runtime Library, so we need to add them to the model as datatypes.
   [Adding data types]
4. Extend the diagram by adding more classes and association and composition relations as shown below. Pay extra attention to the navigability and multiplicity of the references.
   [Class Diagram With Classes]
5. Select attributes to be primary keys (`Person`: id is `name`, `Event`: id is `name`, `Registration`: id is `id`)
   [Selecting primary keys]

NOTE	Verify the generated code:remove any `@OneToOne` annotations from getters associated with `Date` and `Time` from the `Event` class.

6. Create an extra `int` attribute for the `RegistrationManager` as well and set it as the ID (similarly to the other three classes).

CAUTION	If you forget to supply an ID to **any of your entities**, Hibernate will throw an exception and you application will fail to start.

7. Share the modeling project to git. You can use the command line git client or EGit.
   [umllab share project]

1.9. Setting up a Spring-based Backend

You can download the Spring Tools Suite IDE from here.

1.9.1. Running the Backend Application from Eclipse

1. Import the **EventRegistration-Backend** Spring Boot project as a Gradle project from *File* > *Import...* > *Gradle* > *Existing Gradle project* using the default settings. Select the previously generated Spring project folder as the root of the project.
   [Gradle project import in Eclipse]
2. Ignore the bin folder.
   [sts ignore bin]
3. Find the `EventRegistrationApplication.java` source file, then right click and select *Run As* > *Spring Boot App*. The application will fail to start, since the database is not yet configured, but this action will create an initial run configuration. Example console output (fragment):

   ```
   [...]
   ***************************
   APPLICATION FAILED TO START
   ***************************

   Description:

   Failed to configure a DataSource: 'url' attribute is not specified and no embedded
   datasource could be configured.

   Reason: Failed to determine a suitable driver class
   [...]
   ```

4. Obtain the database URL to access the database remotely, e.g., by opening up a terminal and running: `heroku run echo \$JDBC_DATABASE_URL --app=<YOUR_BACKEND_APP_NAME>`.
5. In Eclipse, open the *EventRegistration-Backend - EventregistrationApplication* run configuration page and add an environment variable called `SPRING_DATASOURCE_URL` with the value obtained in the previous step.
   [Adding env var to run config]
6. Add the `spring.jpa.hibernate.ddl-auto=create` to `application.properties`. The database content along with the tables this way will be deleted (as necessary) then re-created each time your application starts.

IMPORTANT	In production, the value of this property should be `none` (instead of `create`). Possible values are `none`, `create`, `validate`, and `update`.

7. If needed: troubleshooting:
   - If you get an error message saying something similar to `createClob() is not yet implemented`, then you can try setting the

`spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true` variable in your `application.properties`. It could be a workaround a workaround for an issue with Postgres.

- Sometimes environment variables don't work with Spring apps. In this case you can set the `spring.datasource.url`, the `spring.datasource.username`, and the `spring.datasource.password` variables in the application properties as an alternative to setting the `SPRING_DATASOURCE_URL` environment variable.
- Make sure no other apps are running on localhost:8080. You can test it by opening the browser and entering `localhost:8080` as the address.

1.9.2. Spring Transactions

1. **Verify** the contents of the `EventRegistrationApplication` class:

```
package ca.mcgill.ecse321.eventregistration;

import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
@SpringBootApplication
public class EventRegistrationApplication {

    public static void main(String[] args) {
        SpringApplication.run(EventRegistrationApplication.class, args);
    }

    @RequestMapping("/")
    public String greeting() {
        return "Hello world!";
    }
}
```

2. Create a new package in `src/main/java` and name it `ca.mcgill.ecse321.eventregistration.dao`.
3. Create the `EventRegistrationRepository` class within this new package

```
package ca.mcgill.ecse321.eventregistration.dao;

import java.sql.Date;
import java.sql.Time;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;

import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import ca.mcgill.ecse321.eventregistration.model.Person;
import ca.mcgill.ecse321.eventregistration.model.Event;

@Repository
public class EventRegistrationRepository {

    @Autowired
    EntityManager entityManager;

    @Transactional
    public Person createPerson(String name) {
        Person p = new Person();
        p.setName(name);
        entityManager.persist(p);
        return p;
    }

    @Transactional
    public Person getPerson(String name) {
        Person p = entityManager.find(Person.class, name);
        return p;
    }

    @Transactional
    public Event createEvent(String name, Date date, Time startTime, Time endTime)
{
        Event e = new Event();
        e.setName(name);
        e.setDate(date);
        e.setStartTime(startTime);
        e.setEndTime(endTime);
        entityManager.persist(e);
        return e;
    }

    @Transactional
    public Event getEvent(String name) {
        Event e = entityManager.find(Event.class, name);
        return e;
    }

}
```

4. Add a new method that gets all events before a specified date (`deadline`). Use a typed query created from an SQL command:

```
@Transactional
public List<Event> getEventsBeforeADeadline(Date deadline) {
    TypedQuery<Event> q = entityManager.createQuery("select e from Event e where
e.date < :deadline",Event.class);
    q.setParameter("deadline", deadline);
    List<Event> resultList = q.getResultList();
    return resultList;
}
```

NOTE

To try the methods, you can create a JUnit test under *src/test/java*. Currently the methods in `EventRegistrationRepository` directly access the objects stored in the database via the `EntityManager` instance and these methods should implement both database operations and service business logic (including input validation — which we omitted in this part). In later sections, however, we will see how we can easily separate the database access and the service business logic in Spring applications.

1.9.3. Debugging: connecting to the database using a client

There are cases when a developer wants to know the contents of the database. In this case, a database client program can be used to access the database schema and table contents. Here are the general steps to access the Postgres database provided by Heroku:

1. Obtain the database URL to access the database remotely, e.g., by opening up a terminal and running: `heroku run echo \$JDBC_DATABASE_URL --app=<YOUR_BACKEND_APP_NAME>`.
2. The returned value follows the format that holds all main important parameters that are needed for accessing the database server:

   ```
   jdbc:postgresql://<HOST>:<PORT>/<DATABASE_NAME>?user=<USERNAME>&password=<PASSWORD>
   &sslmode=require
   ```

   These parameters are:

   - Database host: the URL for the server
   - Port: the por on which the DB server is listening
   - Database name: the first section after the URL
   - Username: the first parameter value in the provided URL
   - Password: the second parameter value in the provided URL
3. With these parameters you can use any Postgres client you prefer to connect to the database. Here is an example for such a connection from Linux using `postgres-client`:

```
$> psql postgresql://ec2-54-243-223-245.compute-
1.amazonaws.com:5432/d4412g60aaboa7?user=hdjnflfirvkmmr
Password:
psql (10.6 (Ubuntu 10.6-0ubuntu0.18.04.1))
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256,
compression: off)
Type "help" for help.

d4412g60aaboa7=> \dt
                          List of relations
 Schema |                Name                | Type  |     Owner
--------+------------------------------------+-------+---------------
 public | event                              | table | hdjnflfirvkmmr
 public | person                             | table | hdjnflfirvkmmr
 public | registration                       | table | hdjnflfirvkmmr
 public | registration_manager               | table | hdjnflfirvkmmr
 public | registration_manager_events        | table | hdjnflfirvkmmr
 public | registration_manager_persons       | table | hdjnflfirvkmmr
 public | registration_manager_registrations | table | hdjnflfirvkmmr
(7 rows)

d4412g60aaboa7=> select * from event ;
 name |    date    | end_time | start_time
------+------------+----------+------------
 e1   | 3899-10-09 | 12:00:00 | 10:00:00
(1 row)

d4412g60aaboa7=> \q
$>
```