ECSE 321 Introduction to Software Engineering

Hands-on Tutorials

McGill University

Table of Contents

- HTML version
- PDF version

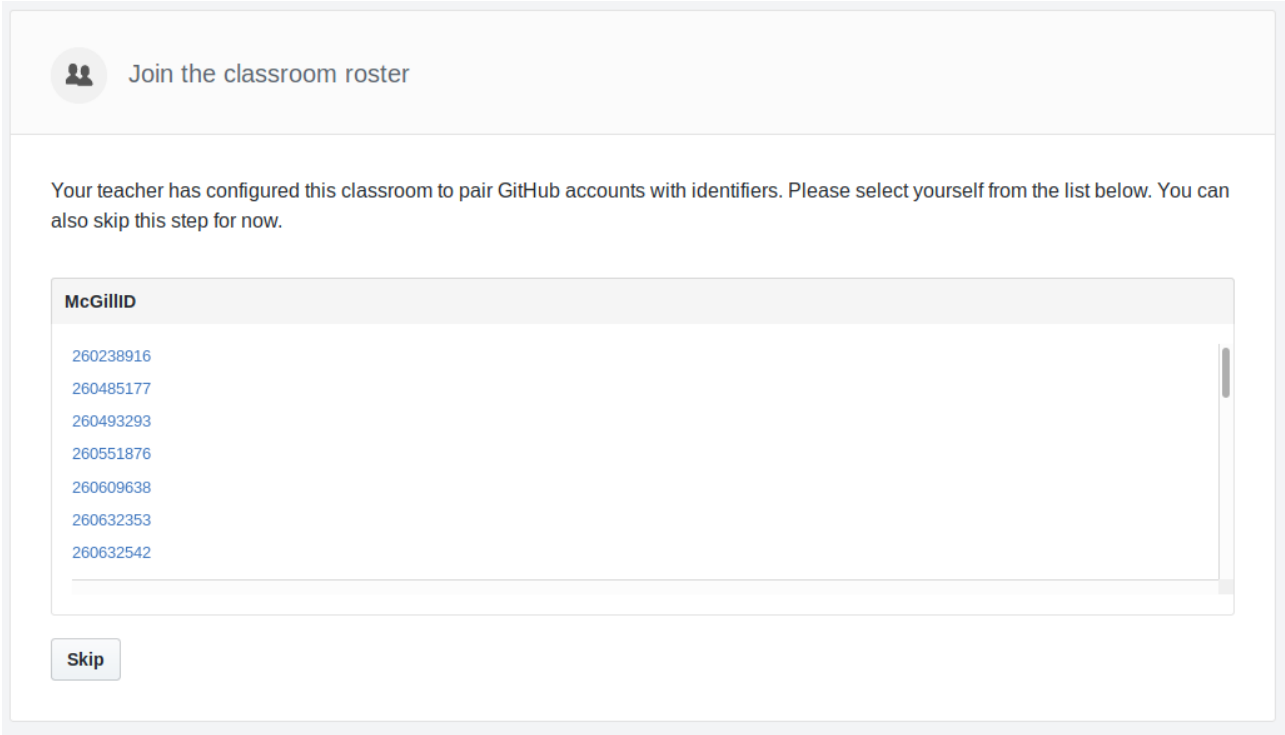
Sections of the tutorial will continuously be published at this web page.

1. Preliminaries

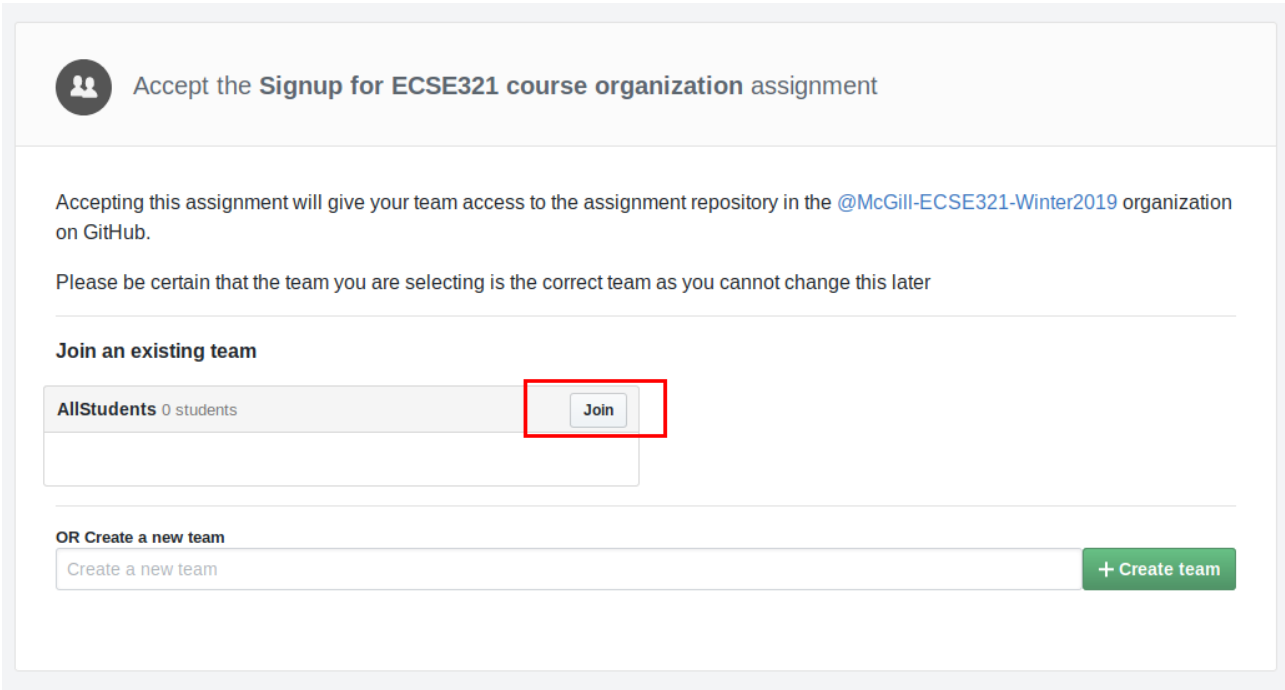
1.1. Getting Started

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*

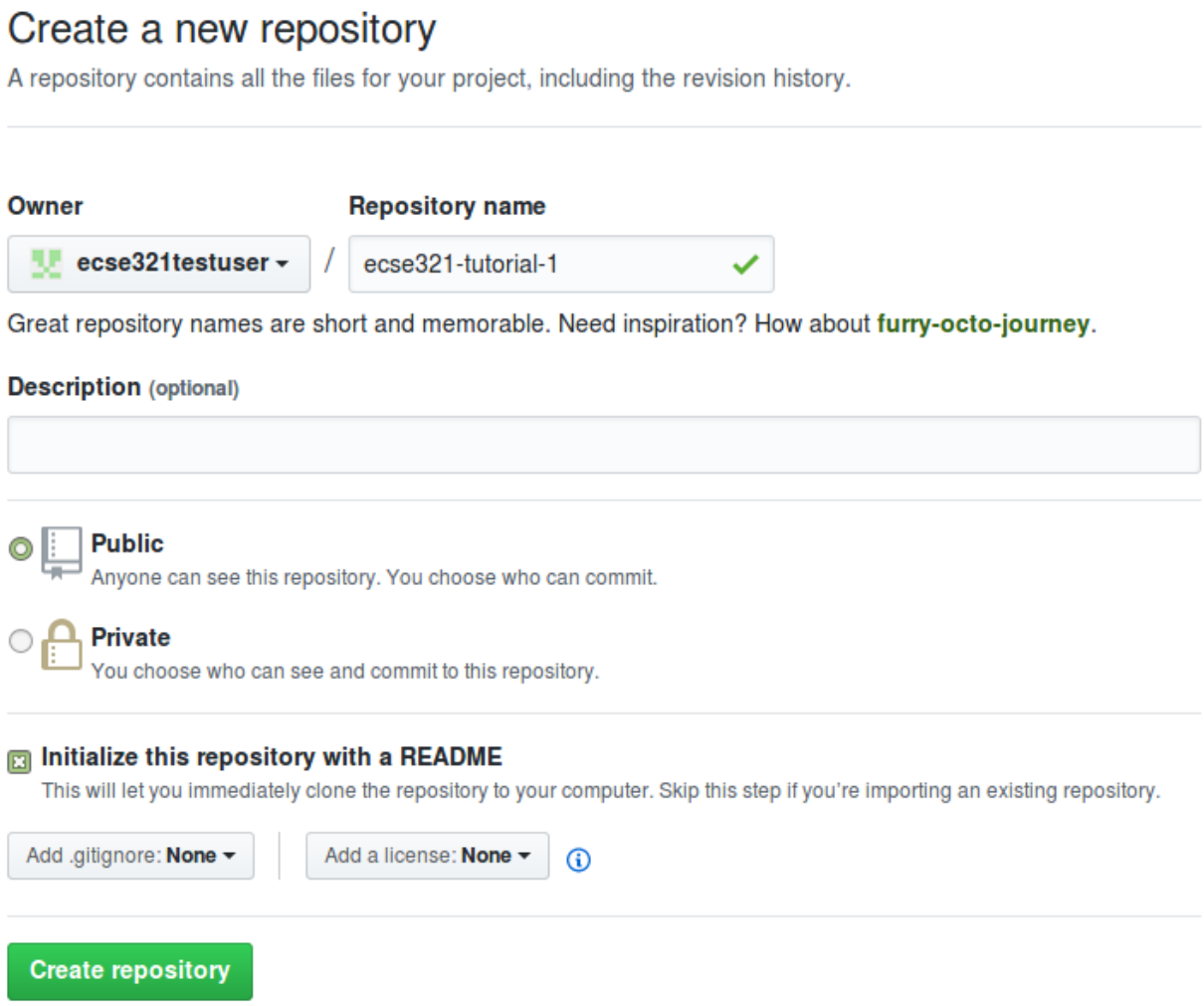


1.2. Project Management Tools for Agile Development

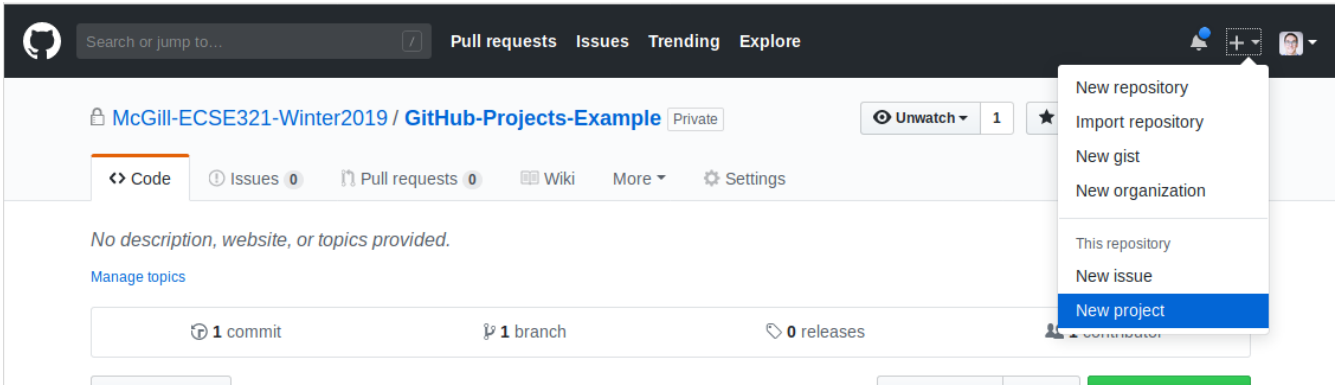
1.2.1. GitHub Projects

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

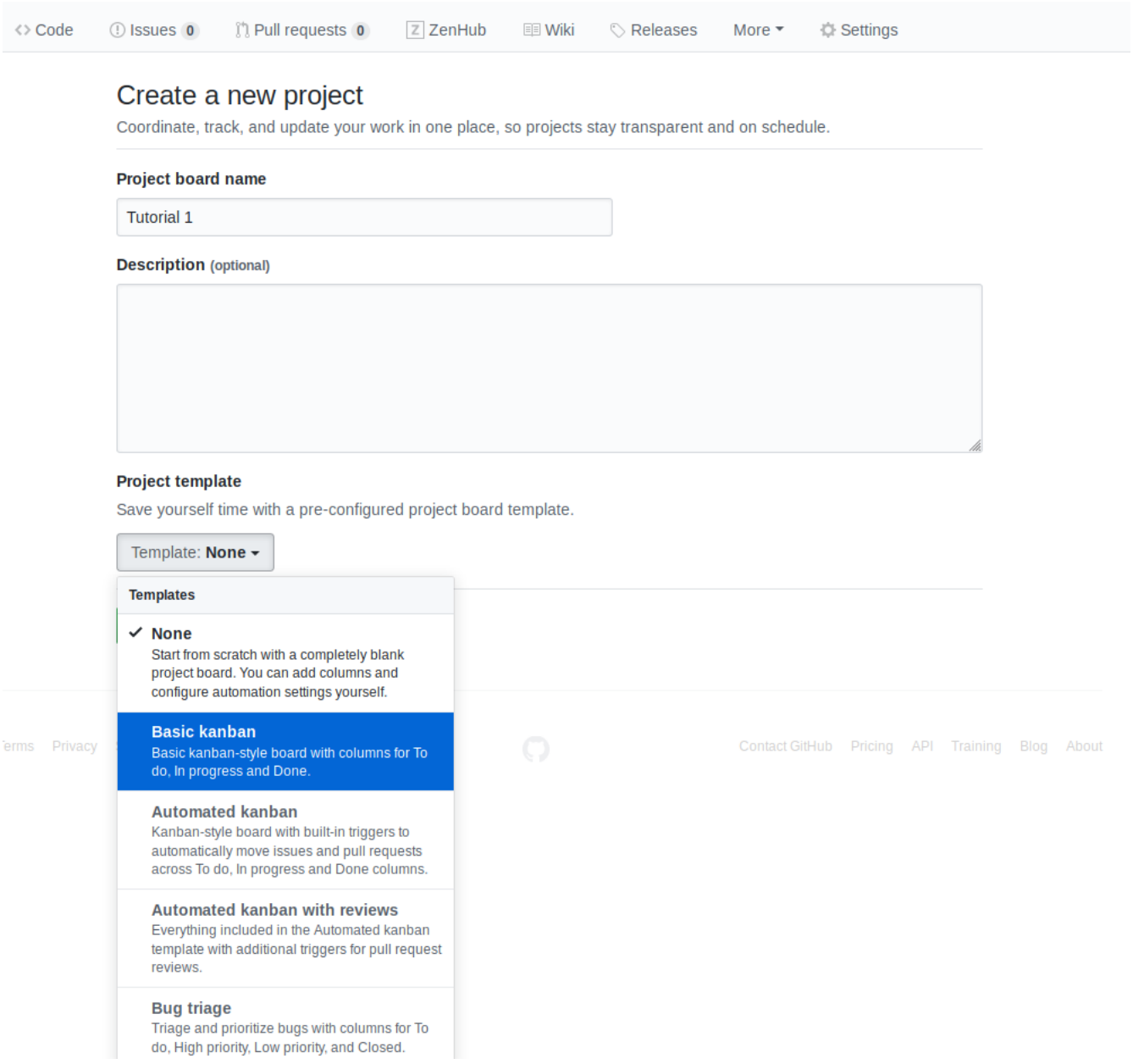
1. Visit https://github.com/ then click on *New repository* (green buttom on the right).
2. Set your user as the owner of the repository.
3. Give a name for the repository (e.g., ecse321-tutorial-1), leave it *public*, then check *Initialize this repository with a README*. Click on *Create repository*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

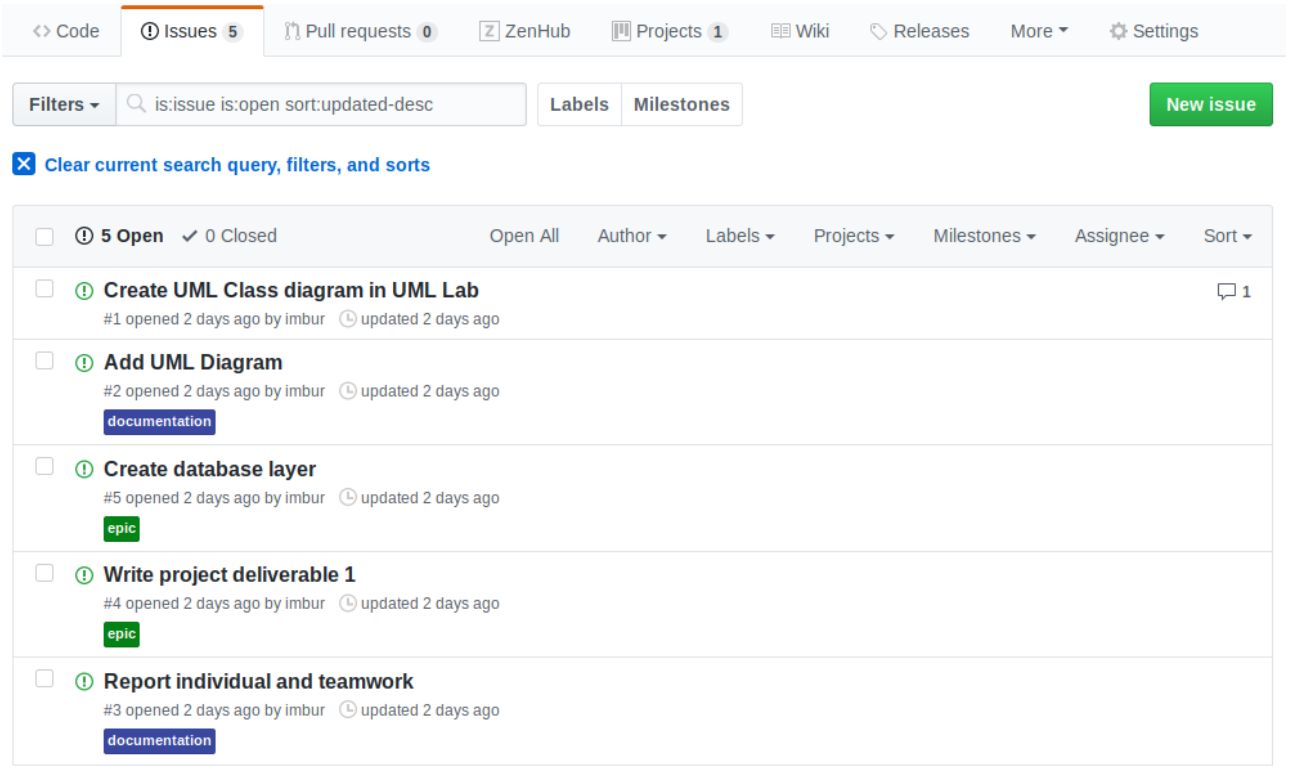


Select Basic Kanban project style:

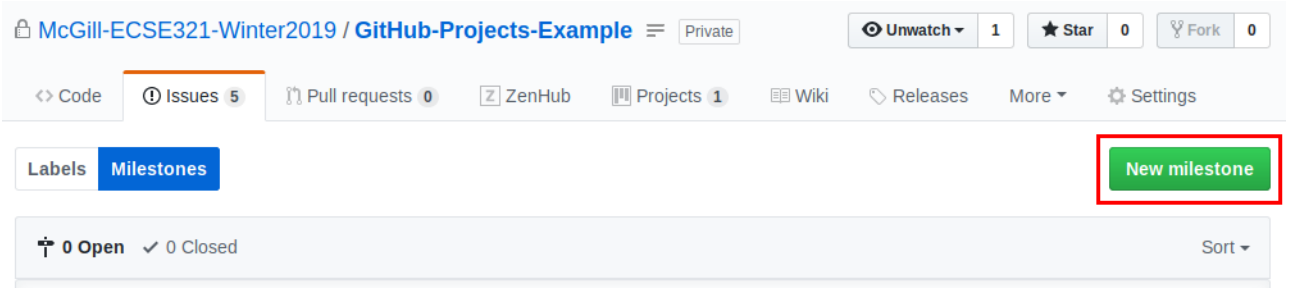


Tasks to complete:

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.

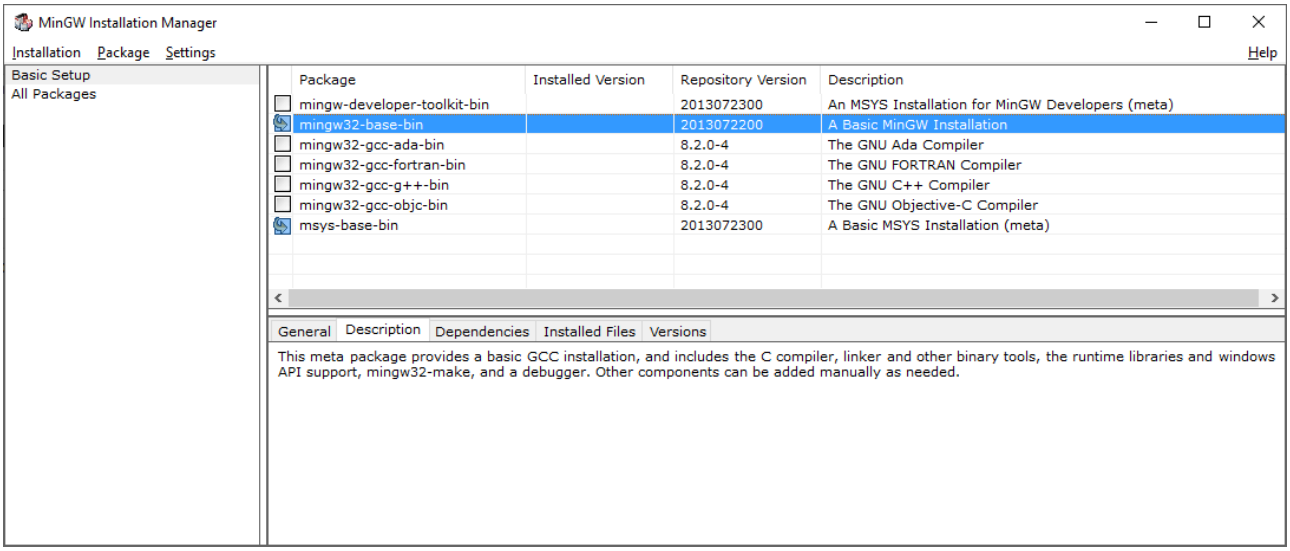
1.3. Command Line Basics

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:

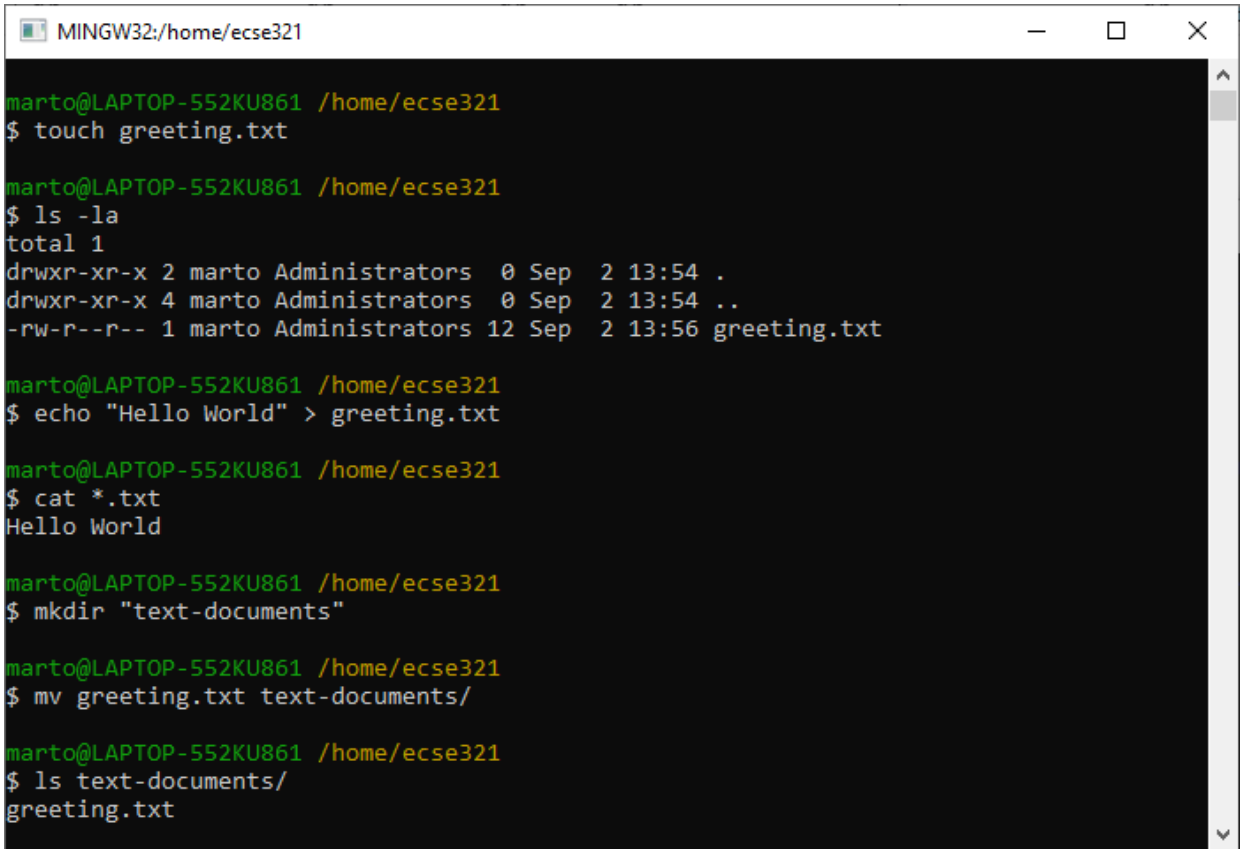
```
$ ls /home
ecse321 guest-user admin
```

- `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! :)

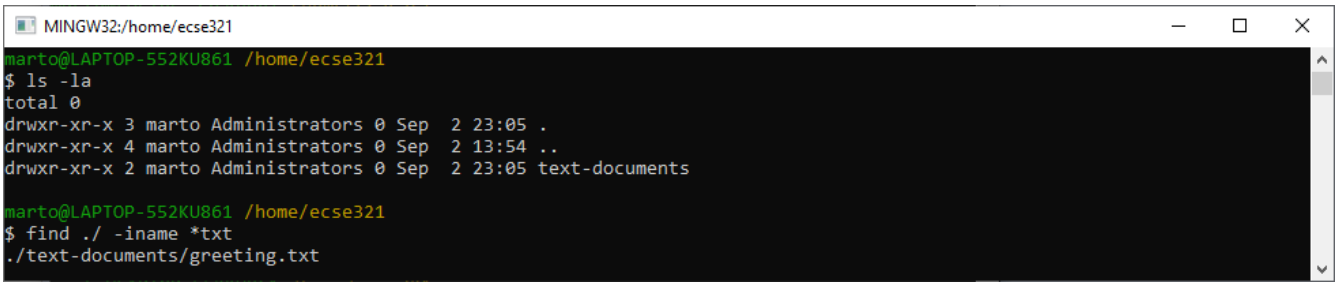
2. Creating files and reading/writing their contents
   - `touch`: creates a file
   - `mkdir`: creates a directory
   - `mv`: moves a file (or directory) from its current location to a target location
   - `echo`: prints a string
   - `cat`: prints the contents of a file
     Example:



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`!

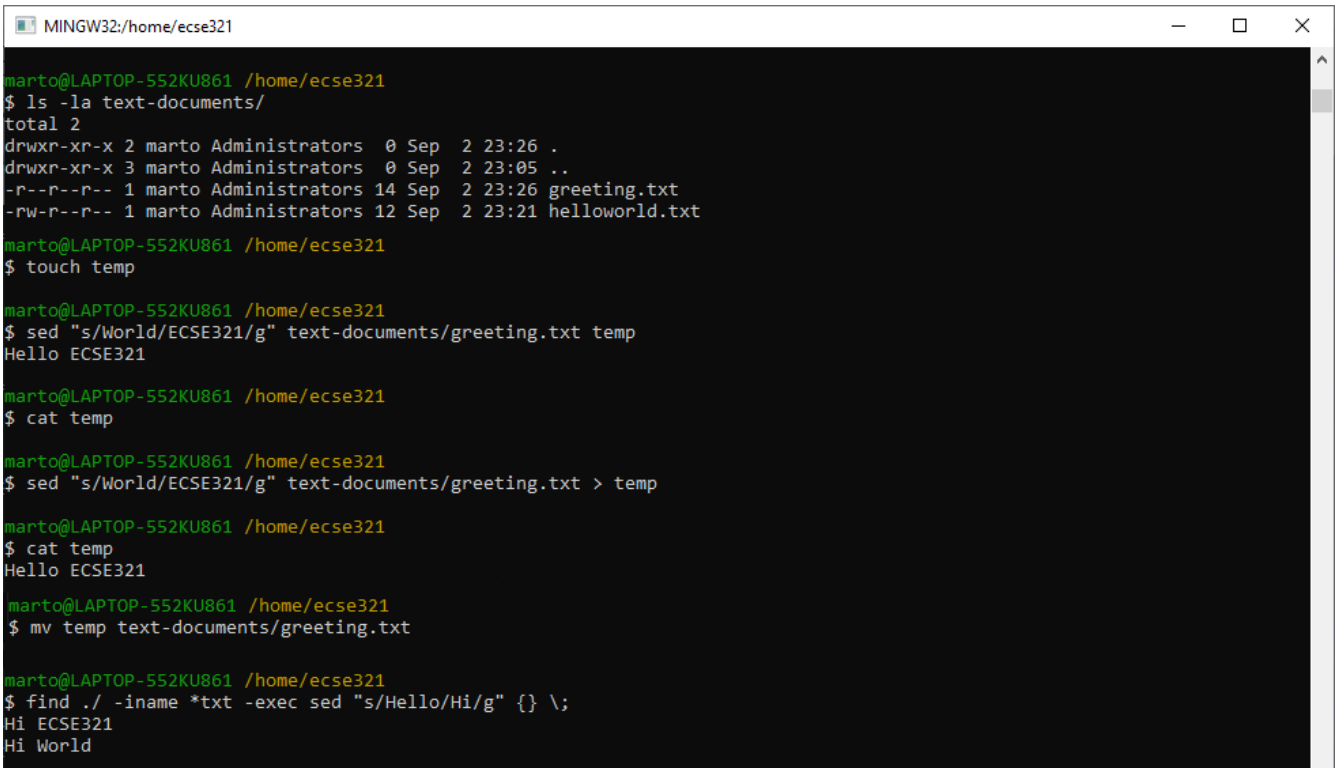
Example:



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

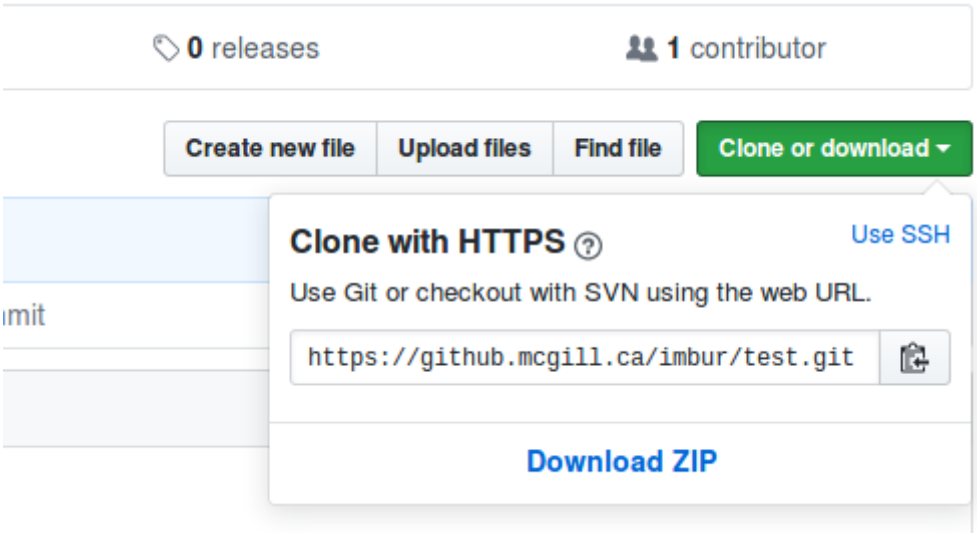
Install the Git version control system (VCS) from https://git-scm.com/downloads.

1.4.2. Creating a remote git repository on GitHub

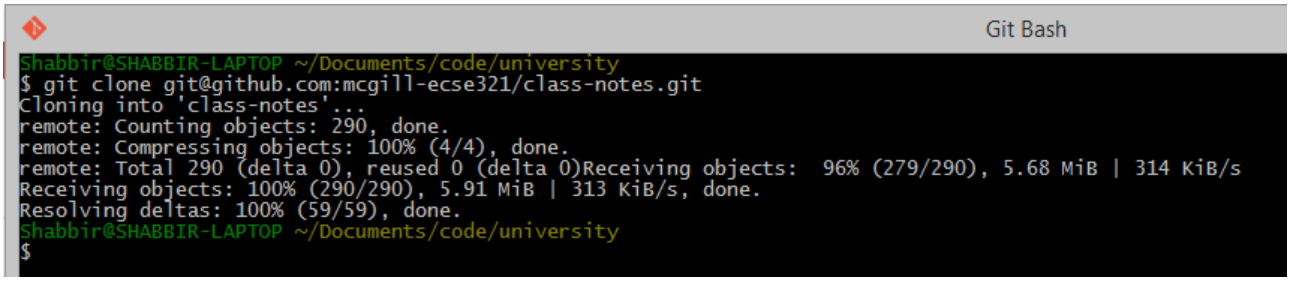
1. Go to https://github.com/new
2. Set *test* as the name of the repository
3. Check the checkbox *Initialize this repository with a README*
4. Click on create repository

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).



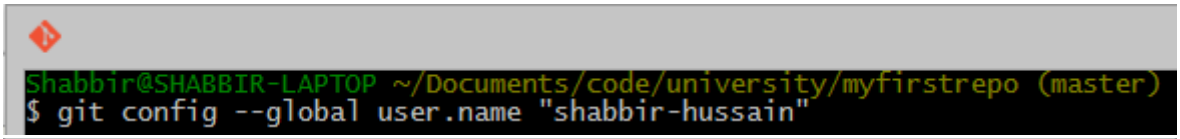
Then, issue `git clone https://url/of/the/repository.git`
You should get an output similar to this:

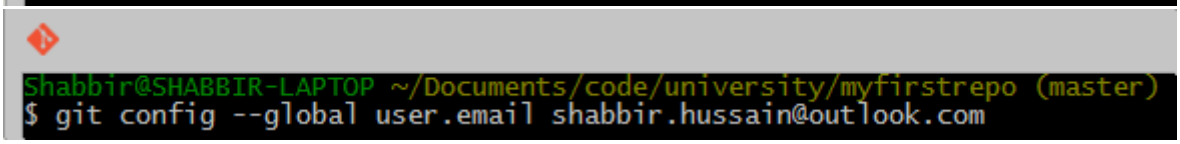


4. Verify the contents of the *working copy* of the repository by `ls -la ./test`. The *.git* folder holds version information and history for the repository, while the *README.md* is an auto-generated text file by GitHub.

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.

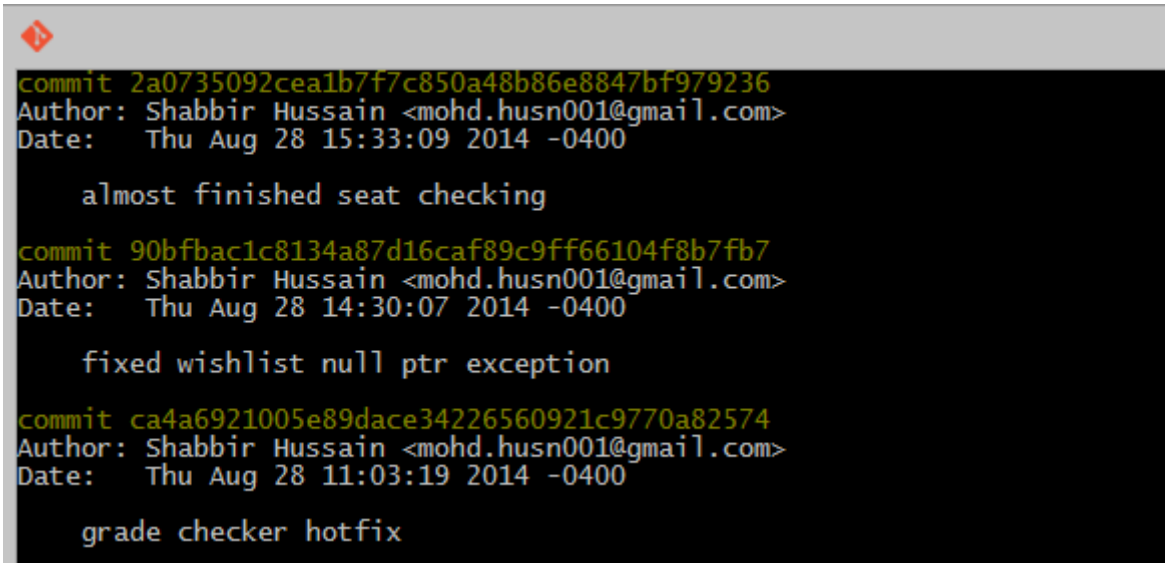




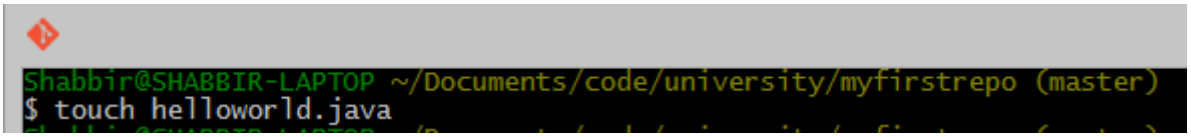
Glossary — Part 1:

   - **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:



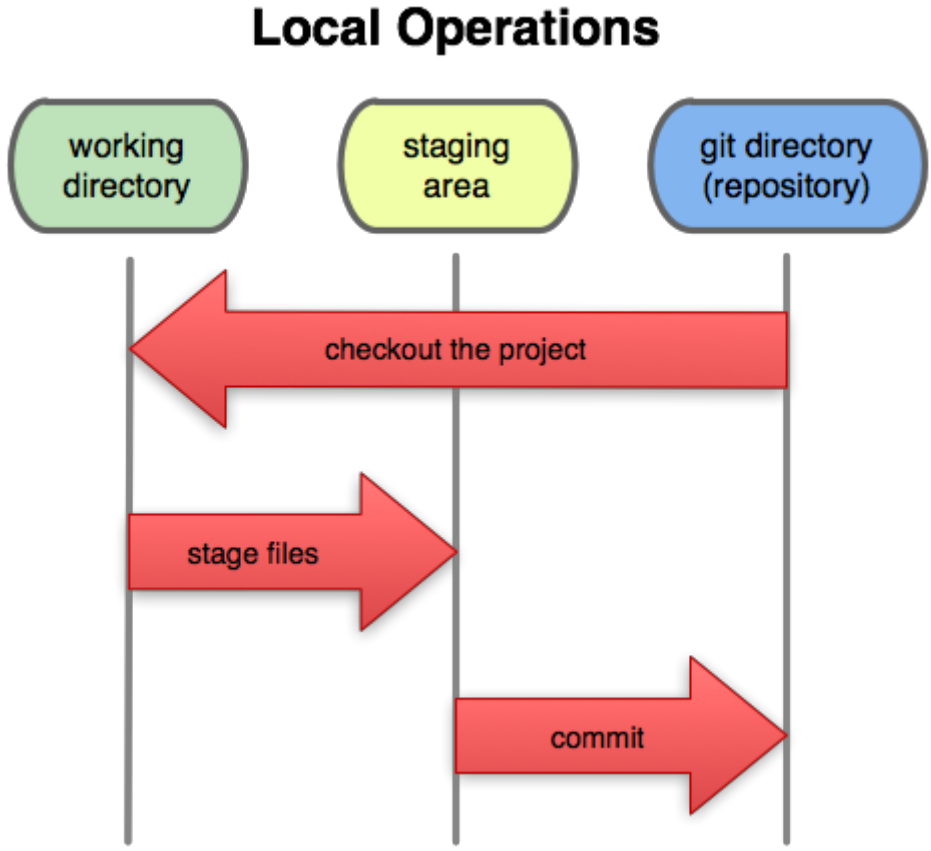
3. Adding and commiting a file: use the `git add` and `git commit` commands.



```
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:



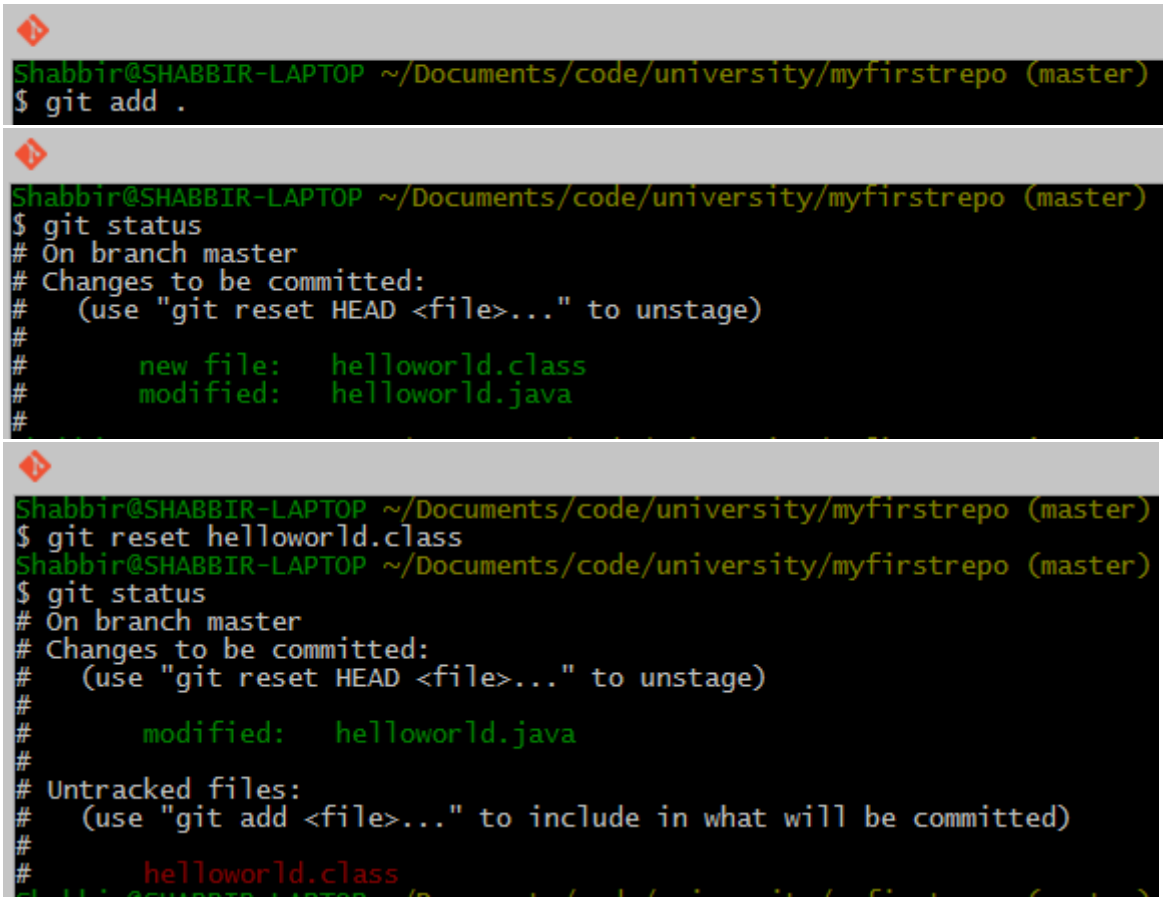
Glossary — Part 2:

- **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`.

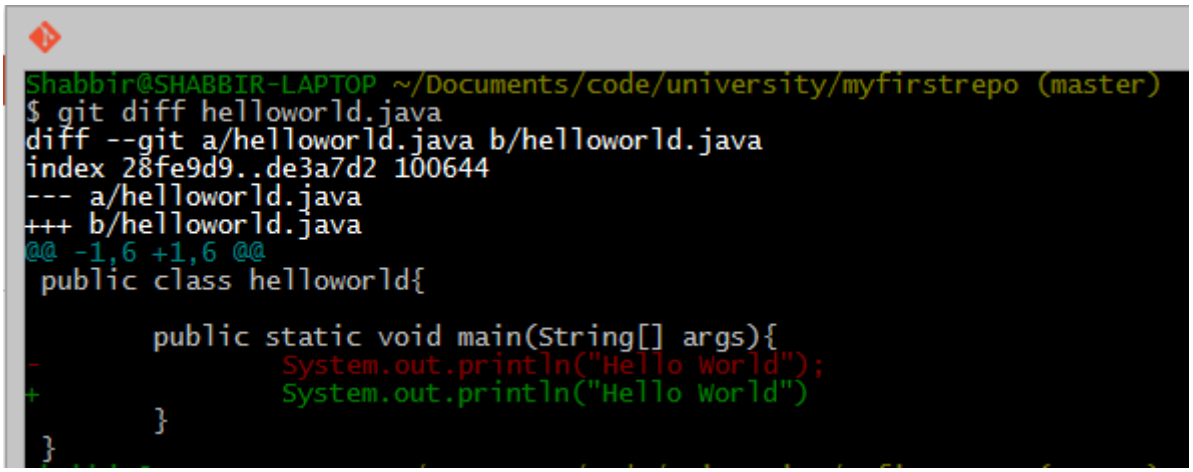
```
Shabbir@SHABBIR-LAPTOP ~/Documents/code/university/myfirstrepo (master)
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   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.

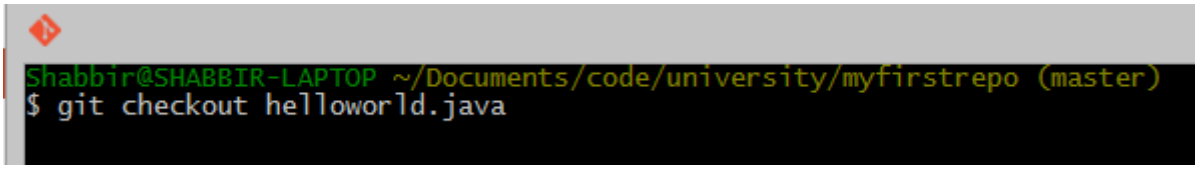


CAUTION	Only staged files will be included in the next commit.

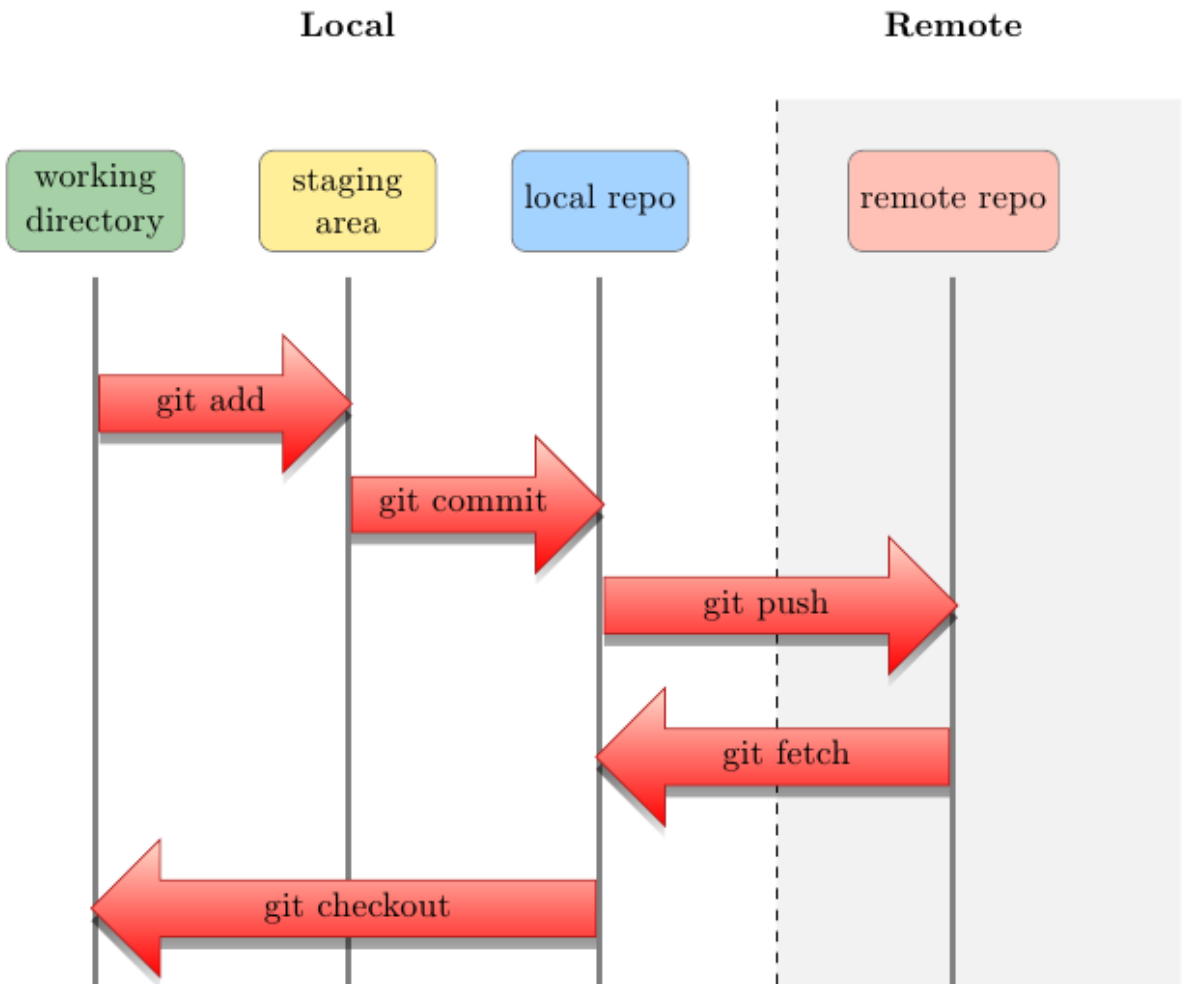
6. To display detailed changes in unstaged files use `git diff`, while use `git diff --staged` to show changes within files staged for commit.



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

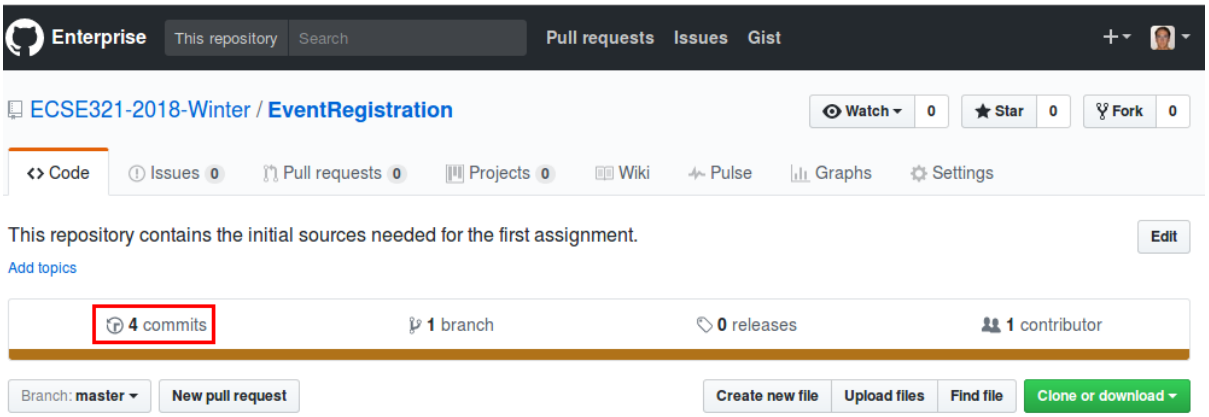


8. The commands `git pull` (or the `git fetch` + `git rebase` combination) and `git push` are used to synchronize local and remote repositories.

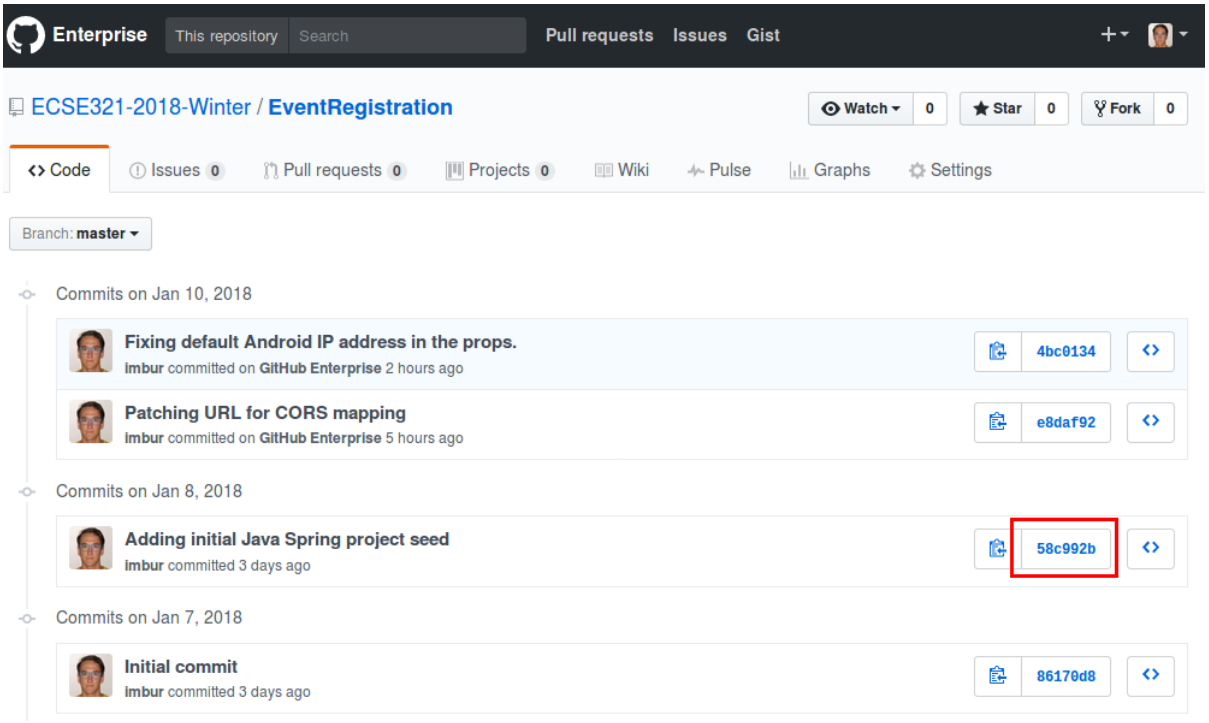


1.4.5. Browsing commit history on GitHub

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



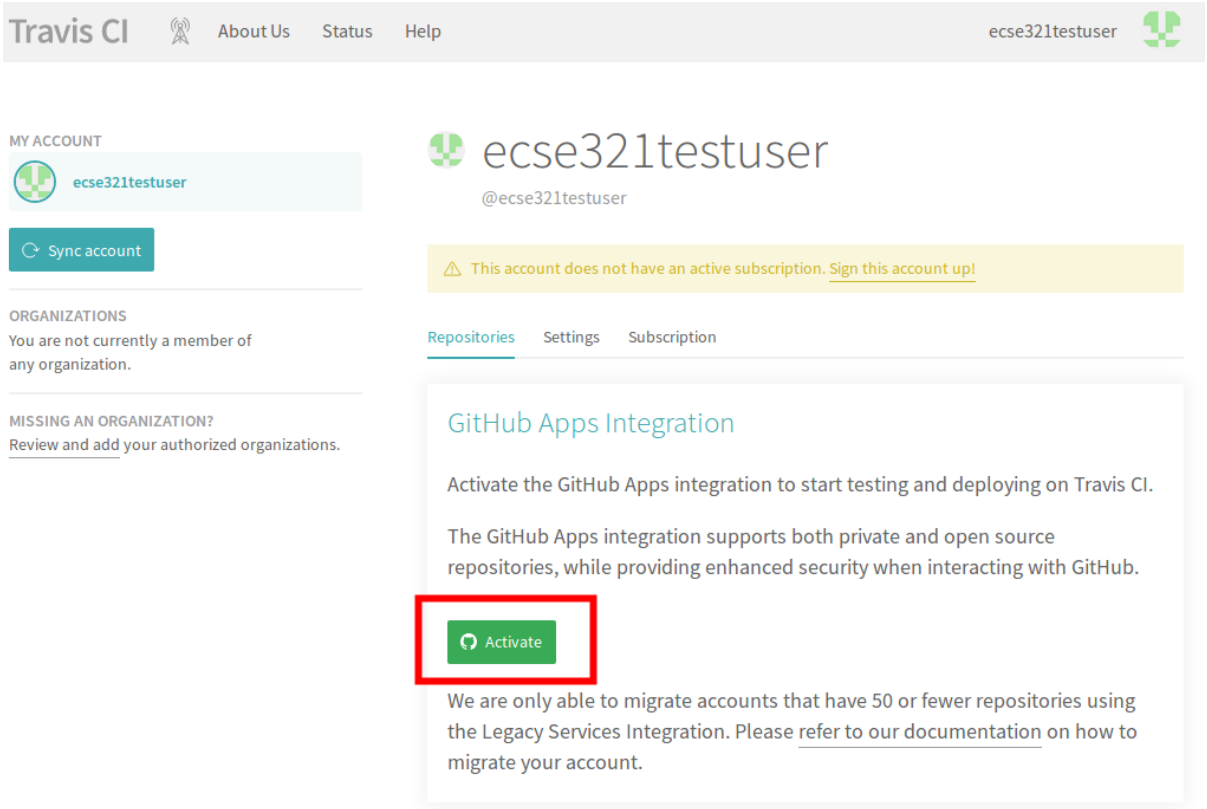
To get a link for a specific commit, click on the button with the first few characters of the hash of the commit.



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

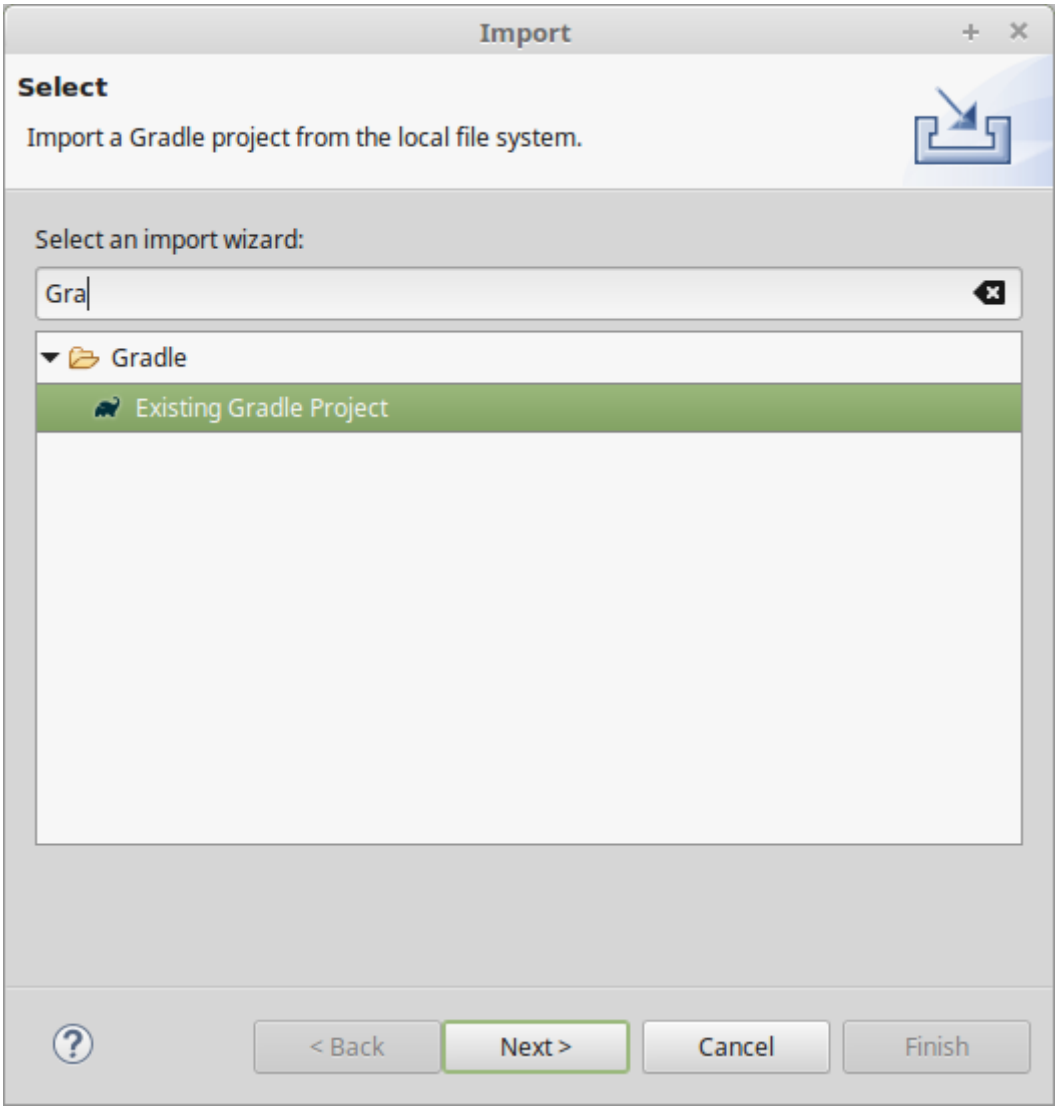
Travis should run your build using Gradle.

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 project with the below gradle structure is available for you to download and import in your IDE.

1. Go to this link and download the *computation.zip*. Once it is downloaded, extract the project from zip.
2. Open your IDE and import the extracted project(from step 1) as *Existing Gradle Project*.



3. After improting the project, check that your imorted project has the required structure as shown below:

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

4. Open the *build.gradle* file and check for the relevant parts as discussed in the steps (5 to 9) below.
5. `java` and the `application` plugins are added in 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'
```

6. JUnit libraries are added in the `dependencies` section.

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

7. A task `compile(type: JavaCompile)` has been added 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'}
  }
}
```

8. The main class has been specified as shown below. Now, you can proceed and run the application.

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

In the command line issue `gradle run`

9. The `jar` Gradle task (defined by the `java` plugin) has been added to produce an executable jar file into `distributable/`.

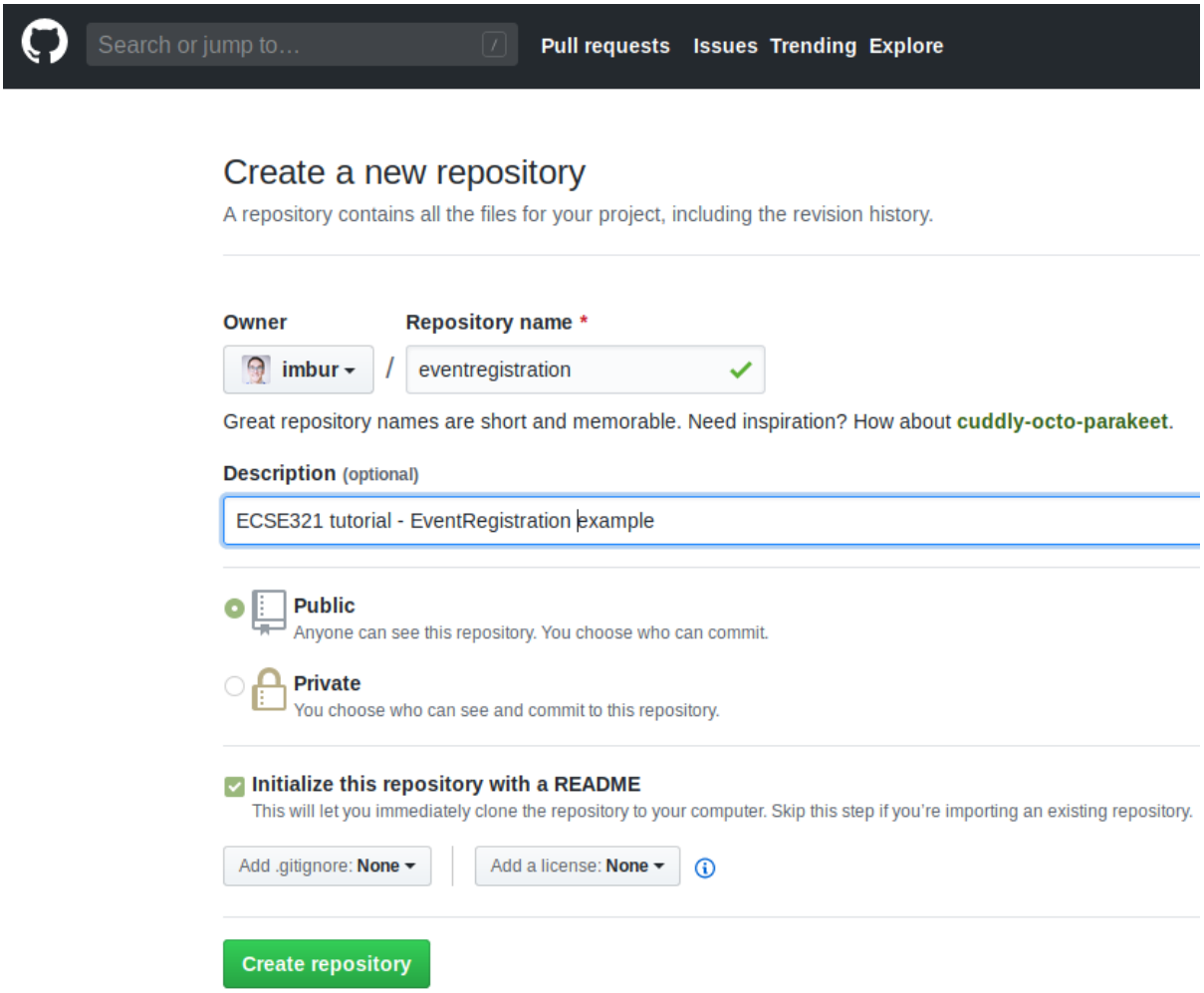
```
jar {
  destinationDir=file('distributable')
  manifest {
    // It is smart to reuse the name of the main class variable instead of
hardcoding it
    attributes "Main-Class": "$mainClassName"
  }
}
```

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

2. Backend

2.1. 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
```

2.2. Heroku

2.2.1. 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`.

2.2.2. 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 the below command to create an application named "eventregistration-backend-<UNIQUE_ID>".

```
heroku create eventregistration-backend-<UNIQUE_ID> -n
```

> **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.

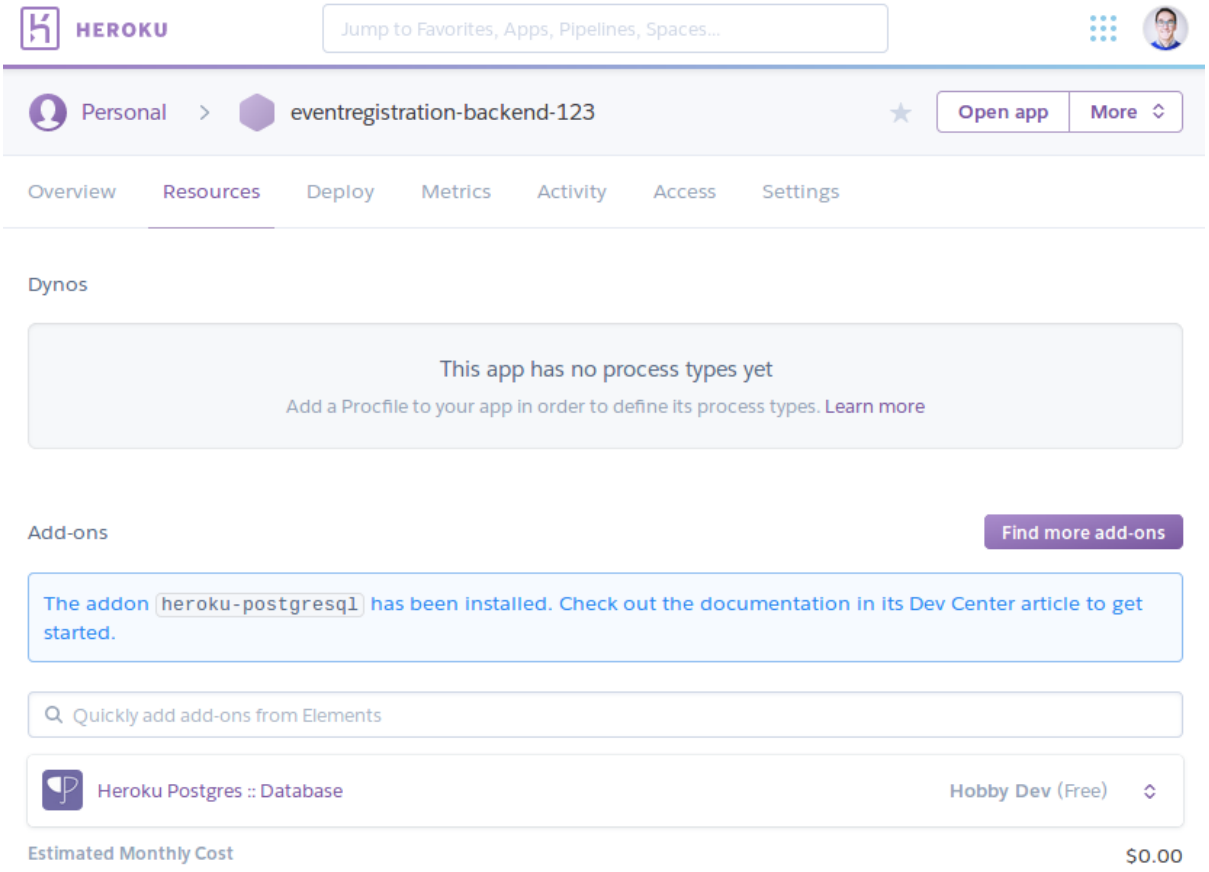
1. Add the multi procfile and Gradle buildpacks to the app.

```
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
```

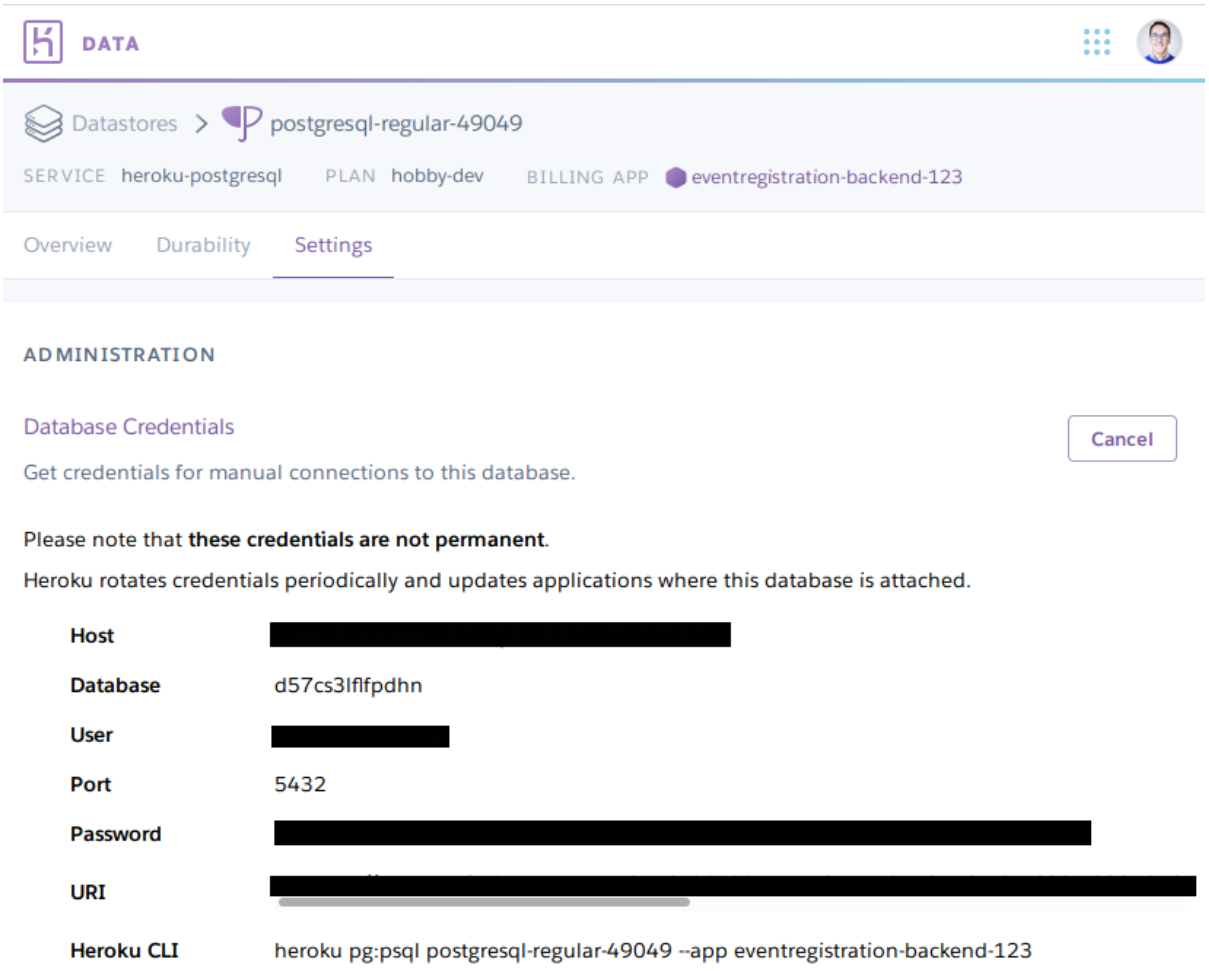
> **CAUTION** Order is important.

2.2.3. 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.)

2.2.4. 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"
```

2.2.5. 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>
```

2.2.6. 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
```

2.3. Domain modeling and code generation

2.3.1. Installing UML Lab

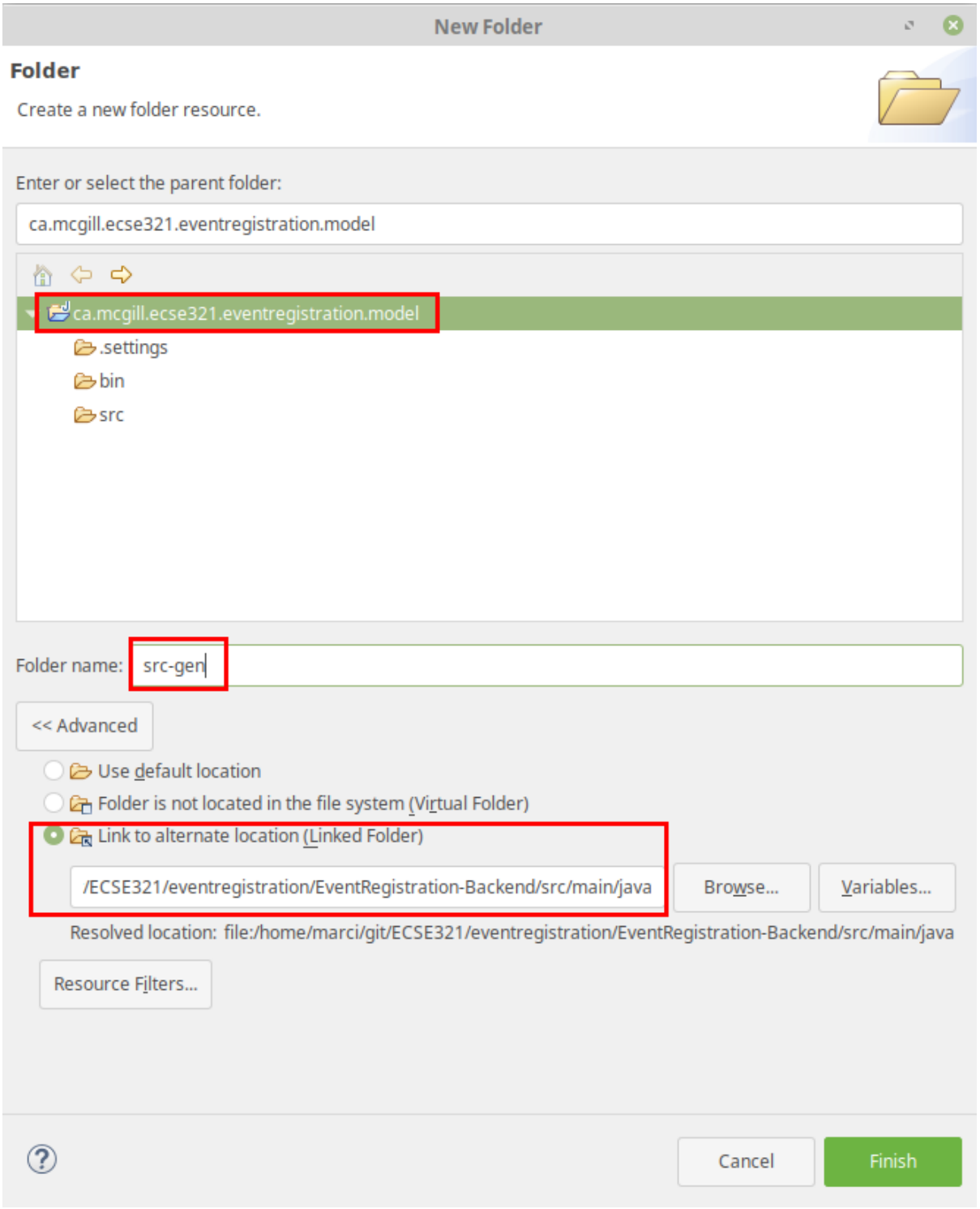
Go to the download page of UML Lab and install it on your machine. To activate it, use the licence key shared in the MyCourses announcement.

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.

2.3.2. UML Lab project setup

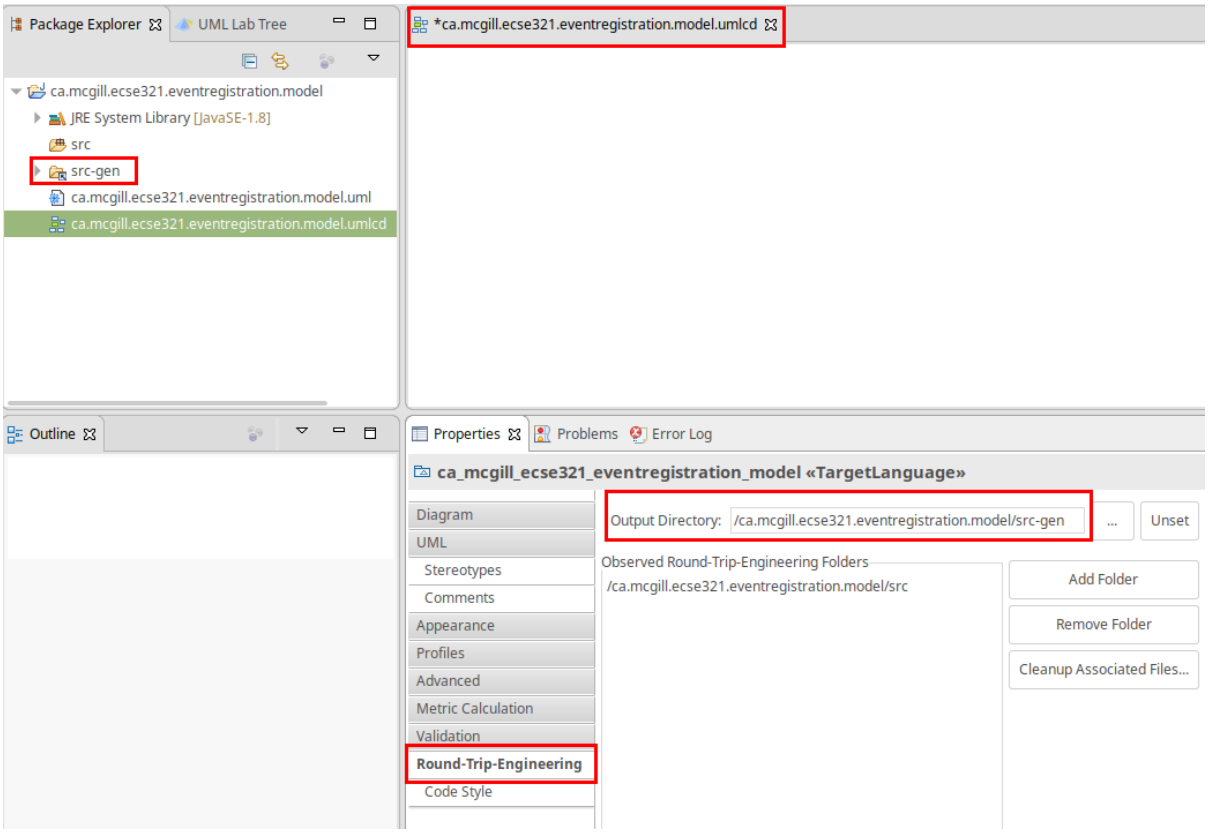
NOTE	Once you start UML Lab, there are some useful tutorials that help you learn about the features of the modeling tool. Furthermore, there is an introduction on how to use and configure UML Lab among the resources of Rice University.

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.

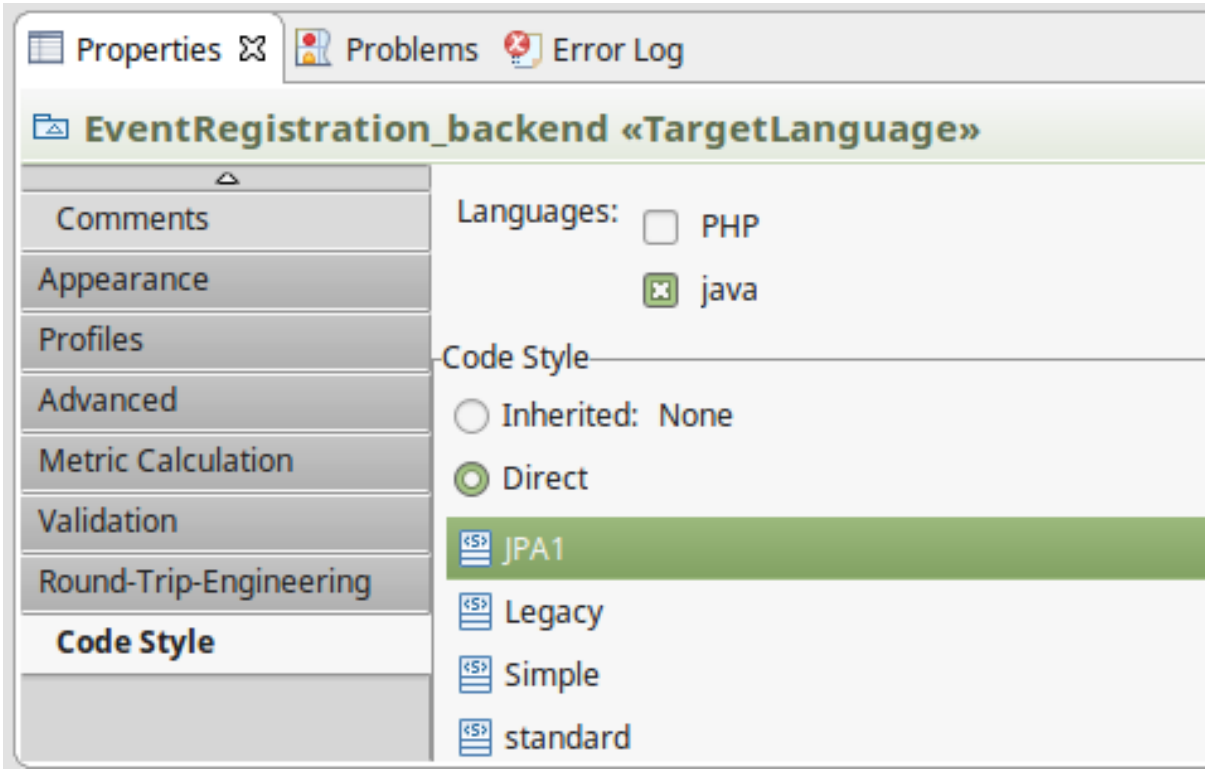


CAUTION	Links to folders will not be versioned, so each team member needs to set this link individually after cloning the project.

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.

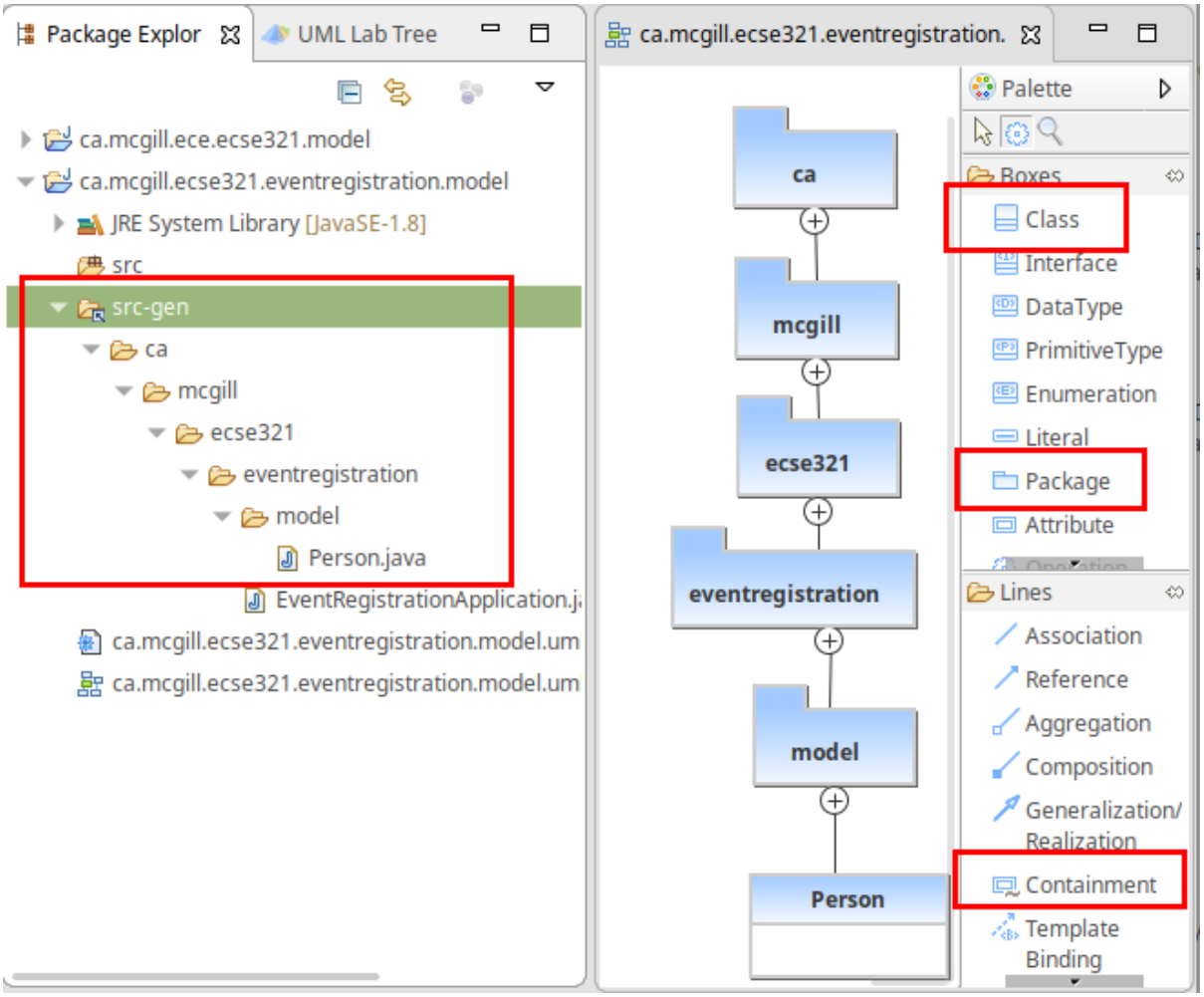


5. In the same *Properties view*, apply the *Direct* > *JPA1* code style.



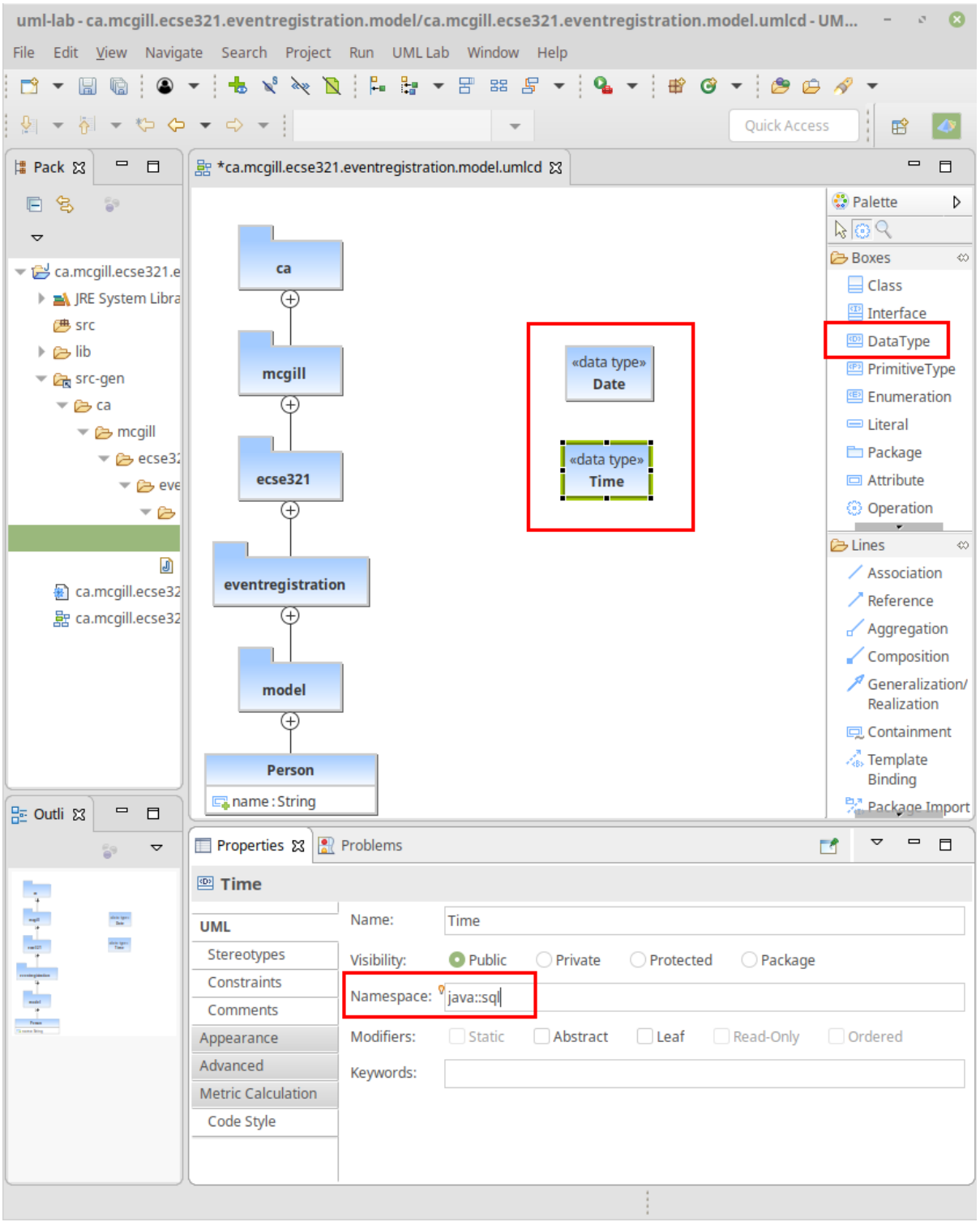
2.3.3. Domain modeling exercise: the Event Registration System

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).

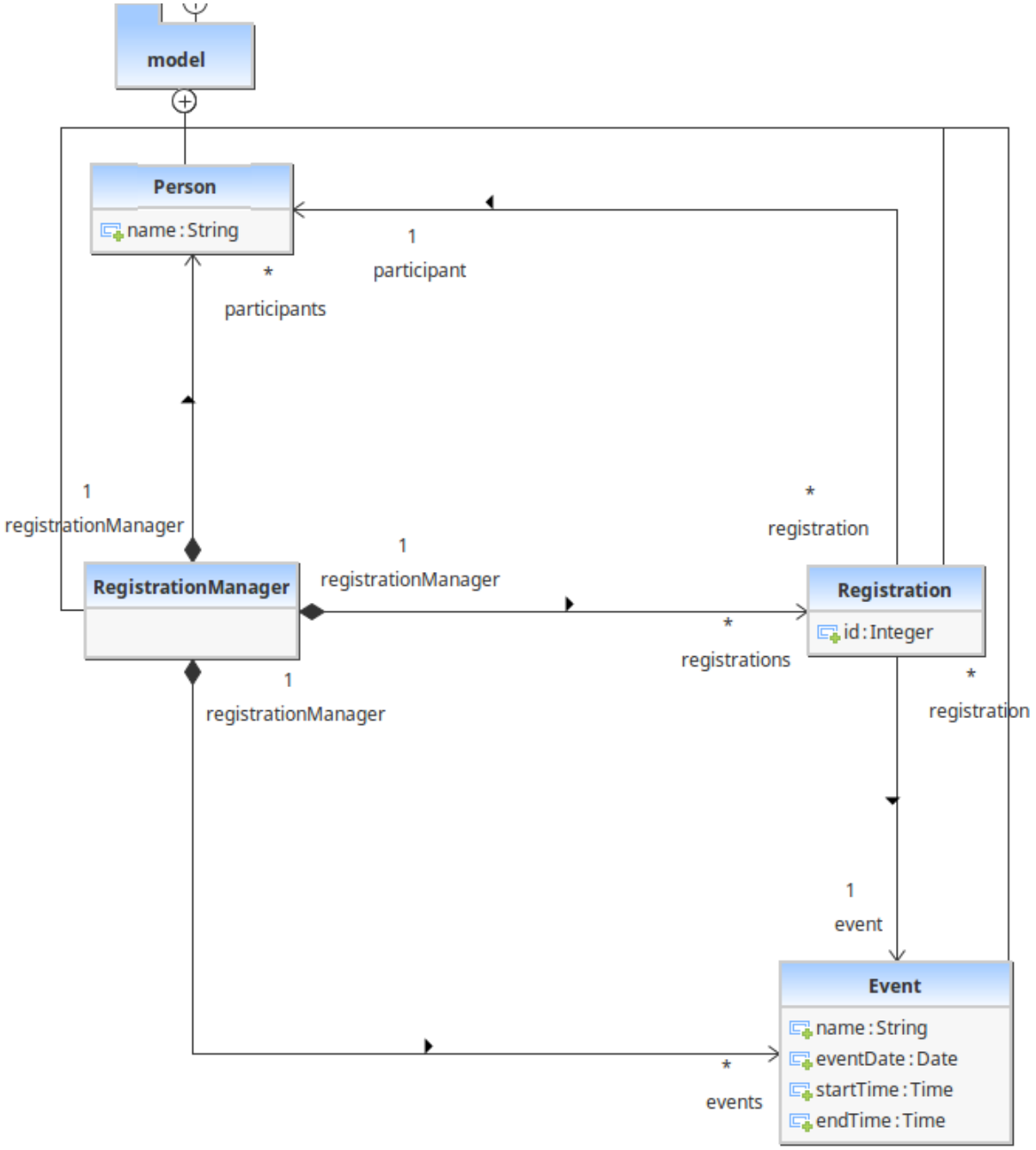


NOTE	If you disabled the automatic code generation on file save action, then you need to do *right click the diagram → generate code* manually.

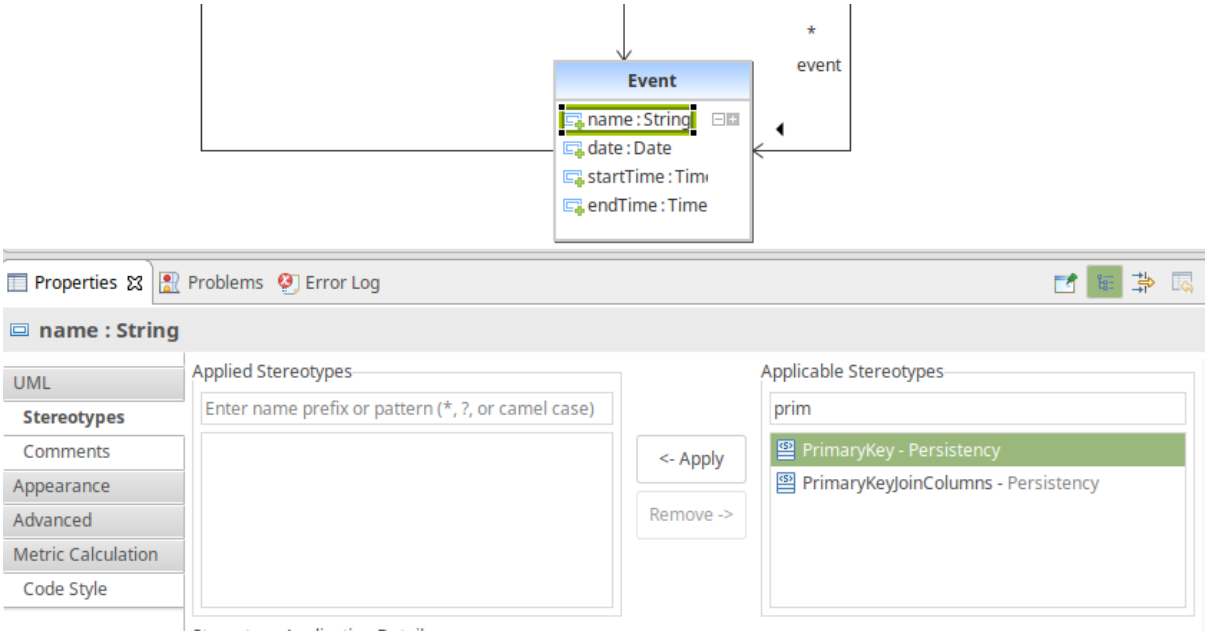
2. Study the generated `Person` class in the `ca/mcgill/ecse321/eventregistration/model` package (folder)!
3. In the upcoming steps, we will use the `java.sql.Time` and `java.sql.Date` data types from the Java Runtime Library, so we need to add them to the model as datatypes.



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.



5. Select attributes to be primary keys (`Person`: id is `name`, `Event`: id is `name`, `Registration`: id is `id`)

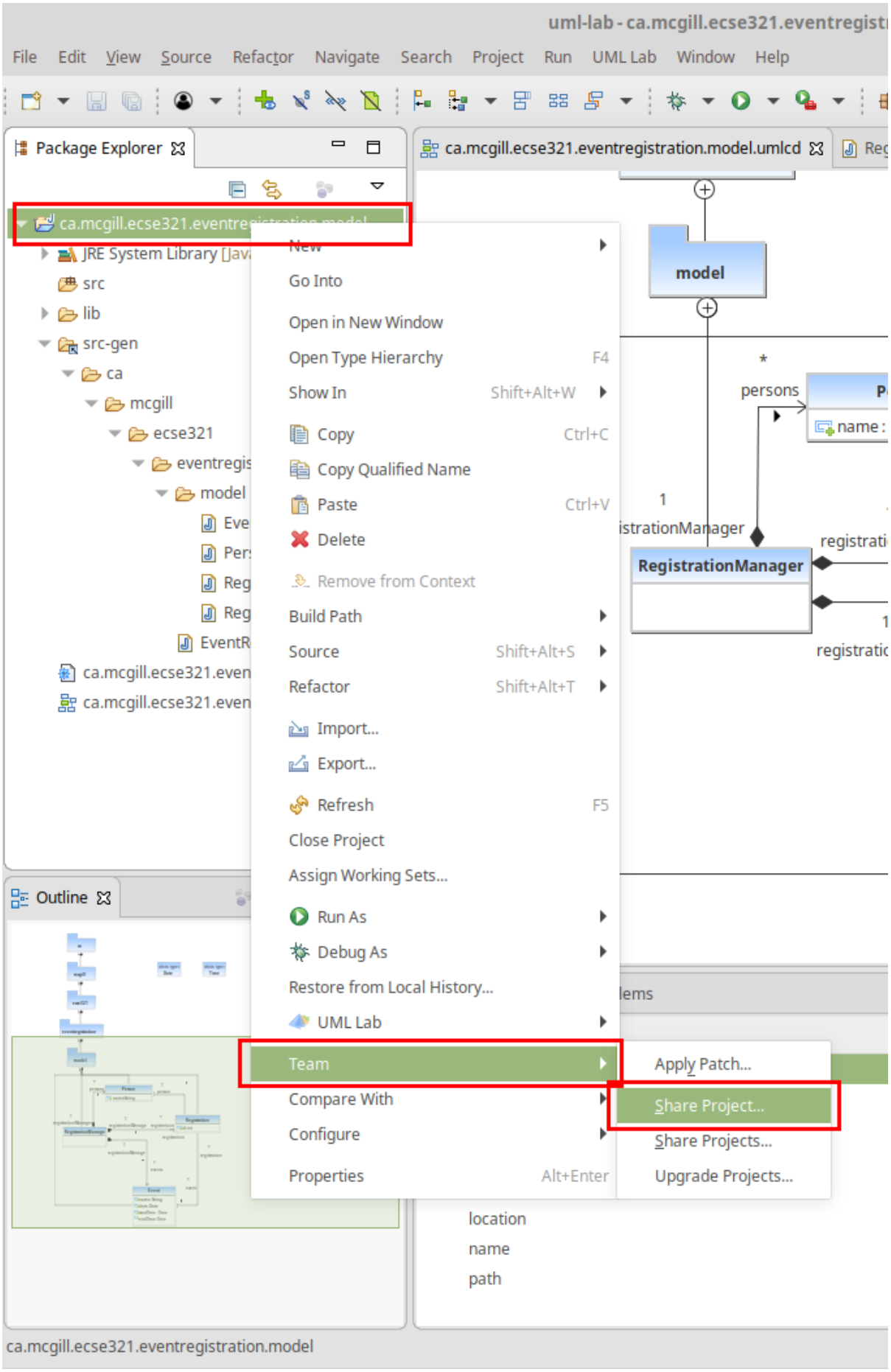


NOTE	Verify the generated code:remove any `@OneToOne` annotations from getters associated with `Date` and `Time` from the `Event` class.

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.

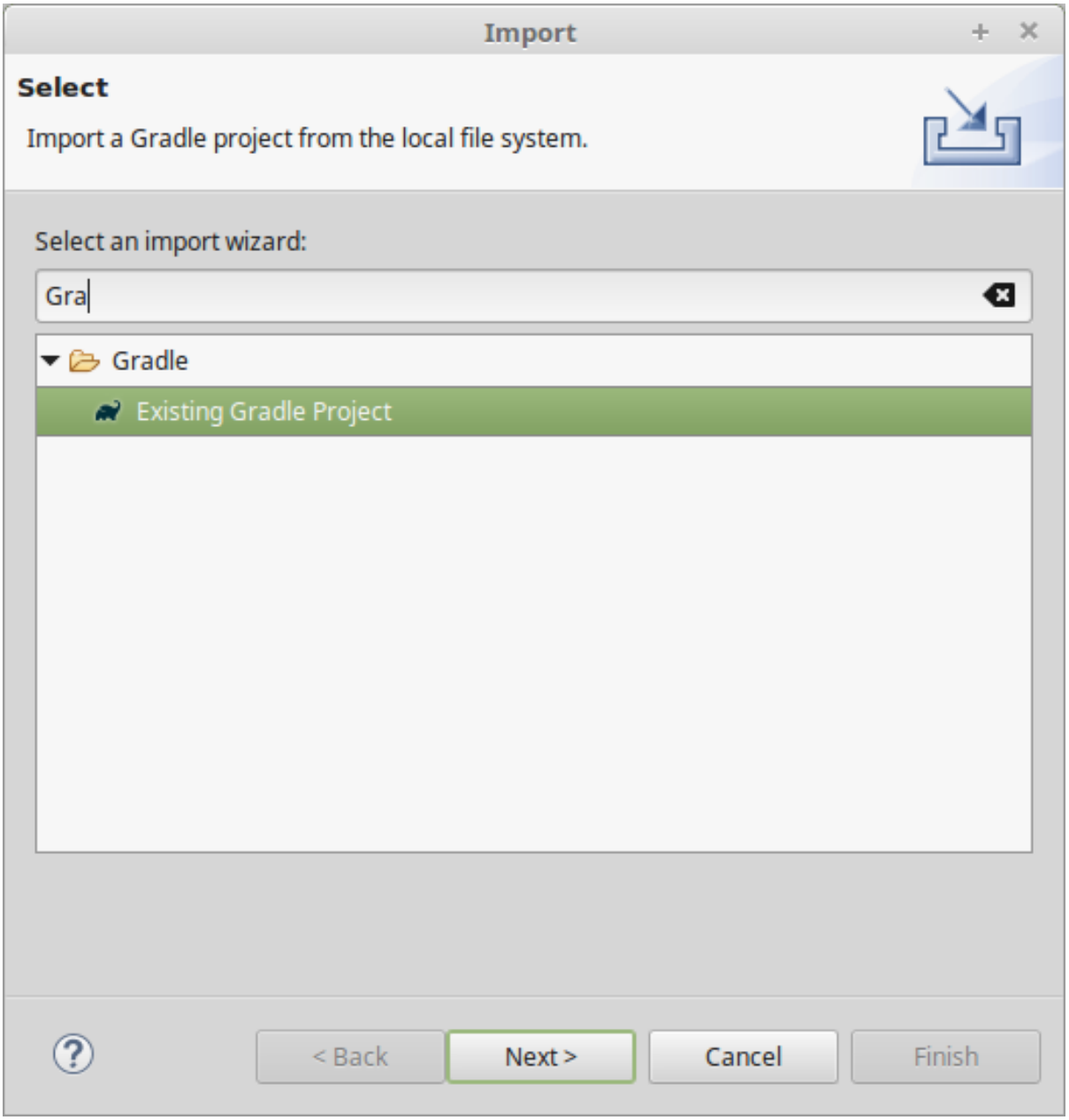

uml-lab - ca.mcgill.ecse321.eventregist
File Edit View Source Refactor Navigate Search Project Run UML Lab Window Help
Package Explorer
ca.mcgill.ecse321.eventregistration.model.umlcd
JRE System Library [Java
src
lib
src-gen
ca
mcgill
ecse321
eventregis
model
Eve
Pers
Reg
Reg
EventR
ca.mcgill.ecse321.even
ca.mcgill.ecse321.even
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In Shift+Alt+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete
Remove from Context
Build Path
Source Shift+Alt+S
Refactor Shift+Alt+T
Import...
Export...
Refresh F5
Close Project
Assign Working Sets...
Run As
Debug As
Restore from Local History...
UML Lab
Team
Compare With
Configure
Properties Alt+Enter
Apply Patch...
Share Project...
Share Projects...
Upgrade Projects...
model
persons
name :
1
istrationManager
RegistrationManager
registrati
registratic
Outline
location
name
path
ca.mcgill.ecse321.eventregistration.model

2.4. Setting up a Spring-based Backend

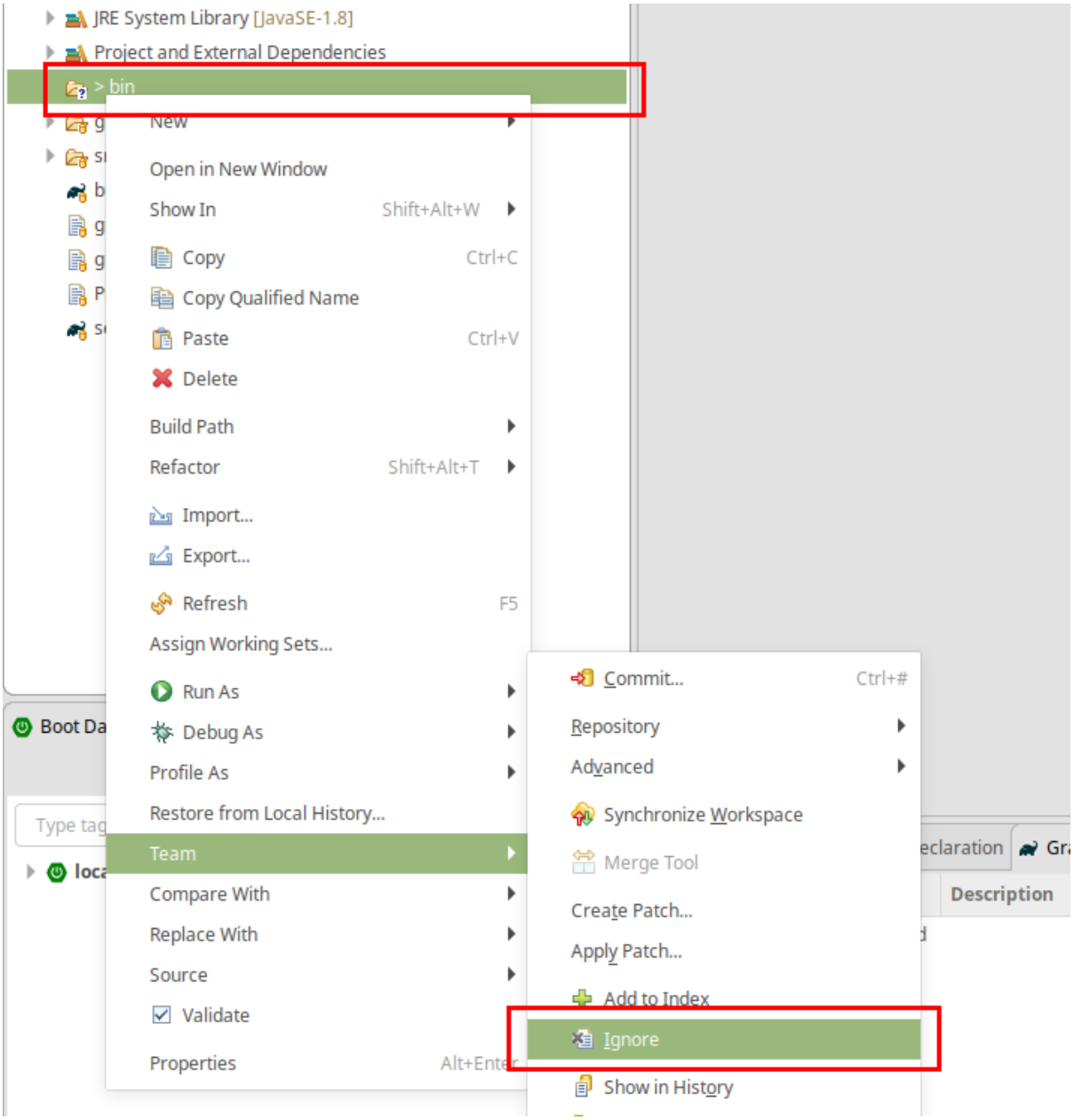
You can download the Spring Tools Suite IDE from here.

2.4.1. 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.



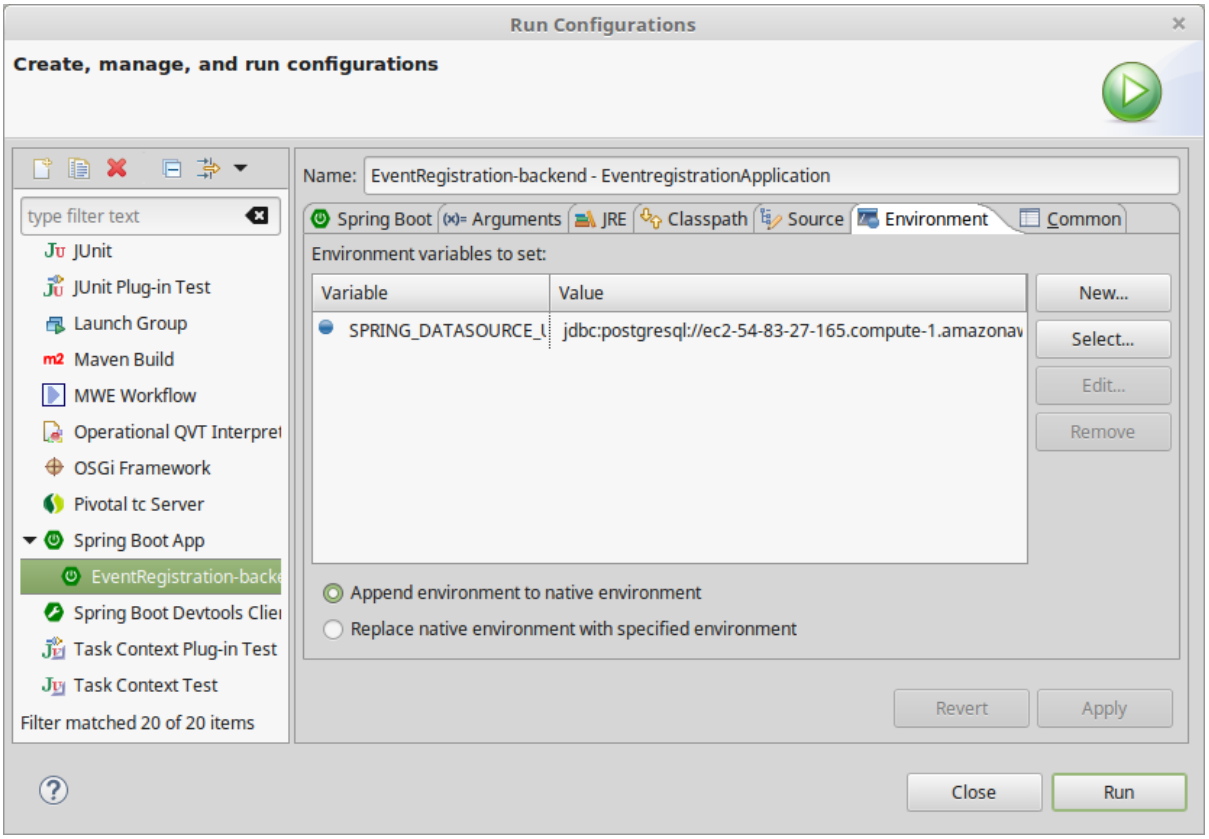
2. Ignore the bin folder.



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.



6. Add the `spring.jpa.hibernate.ddl-auto=update` 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 `update`). 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.

2.4.2. 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.

2.4.3. 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
$>
```

2.5. CRUD Repositories and Services

Previously, in the `ca.mcgill.ecse321.eventregistration.dao.EventRegistrationRepository` class we used an instance of `javax.persistence.EntityManager` from Hibernate to directly to implement the required operations related to saving/retrieving data to/from a database (Create, Read, Update, and Delete operations, shortly, CRUD). This section will introduce the Spring framework's inbuilt support for such CRUD operations via the `org.springframework.data.repository.CrudRepository` interface and will show how to use such repositories to implement your use cases in so-called *service* classes.

If you would like to, you can obtain a version of the project that already has the changes from the previous tutorials here.

2.5.1. Creating a CRUD Repository

1. Create a new interface `PersonRepository` in the `ca.mcgill.ecse321.eventregistration.dao` package and extend the `CrudRepository<Person, String>` interface
2. Create a new method `Person findByName(String name)`

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

import org.springframework.data.repository.CrudRepository;

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

public interface PersonRepository extends CrudRepository<Person, String>{

    Person findPersonByName(String name);

}
```

3. Since Spring supports automated JPA Query creation from method names (see possible language constructs here) **we don't need to implement the interface manually**, Spring JPA will create the corresponding queries runtime! This way we don't need to write SQL queries either.
4. Create interfaces for the `Event` and `Registration` classes as well
EventRepository.java:

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

import org.springframework.data.repository.CrudRepository;

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

public interface EventRepository extends CrudRepository<Event, String> {

    Event findEventByName(String name);

}
```

RegistrationRepository.java:

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

import java.util.List;

import org.springframework.data.repository.CrudRepository;

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

public interface RegistrationRepository extends CrudRepository<Registration,
Integer> {

    List<Registration> findByPerson(Person personName);

    boolean existsByPersonAndEvent(Person person, Event eventName);

    Registration findByPersonAndEvent(Person person, Event eventName);

}
```

2.5.2. Implementing Services

We implement use-cases in *service classes* by using the CRUD repository objects for each data type of the domain model.

1. In *src/main/java*, create a new package `ca.mcgill.ecse321.eventregistration.service`.
2. In this package, create the `EventRegistrationService` class as shown below

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

import java.sql.Date;
import java.sql.Time;
```

```
import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import ca.mcgill.ecse321.eventregistration.dao.EventRepository;
import ca.mcgill.ecse321.eventregistration.dao.PersonRepository;
import ca.mcgill.ecse321.eventregistration.dao.RegistrationRepository;
import ca.mcgill.ecse321.eventregistration.model.Event;
import ca.mcgill.ecse321.eventregistration.model.Person;
import ca.mcgill.ecse321.eventregistration.model.Registration;

@Service
public class EventRegistrationService {

    @Autowired
    EventRepository eventRepository;
    @Autowired
    PersonRepository personRepository;
    @Autowired
    RegistrationRepository registrationRepository;

    @Transactional
    public Person createPerson(String name) {
        Person person = new Person();
        person.setName(name);
        personRepository.save(person);
        return person;
    }

    @Transactional
    public Person getPerson(String name) {
        Person person = personRepository.findPersonByName(name);
        return person;
    }

    @Transactional
    public List<Person> getAllPersons() {
        return toList(personRepository.findAll());
    }

    @Transactional
    public Event createEvent(String name, Date date, Time startTime, Time endTime)
{
        Event event = new Event();
        event.setName(name);
        event.setDate(date);
        event.setStartTime(startTime);
        event.setEndTime(endTime);
```

```
        eventRepository.save(event);
        return event;
    }

    @Transactional
    public Event getEvent(String name) {
        Event event = eventRepository.findEventByName(name);
        return event;
    }

    @Transactional
    public List<Event> getAllEvents() {
        return toList(eventRepository.findAll());
    }

    @Transactional
    public Registration register(Person person, Event event) {
        Registration registration = new Registration();
        registration.setId(person.getName().hashCode() *
event.getName().hashCode());
        registration.setPerson(person);
        registration.setEvent(event);

        registrationRepository.save(registration);

        return registration;
    }

    @Transactional
    public List<Registration> getAllRegistrations(){
        return toList(registrationRepository.findAll());
    }

    @Transactional
    public List<Event> getEventsAttendedByPerson(Person person) {
        List<Event> eventsAttendedByPerson = new ArrayList<>();
        for (Registration r : registrationRepository.findByPerson(person)) {
            eventsAttendedByPerson.add(r.getEvent());
        }
        return eventsAttendedByPerson;
    }

    private <T> List<T> toList(Iterable<T> iterable){
        List<T> resultList = new ArrayList<T>();
        for (T t : iterable) {
            resultList.add(t);
        }
        return resultList;
    }

}
```

2.6. Unit Testing Persistence in the Backend Service

1. In a fresh Spring Boot project, there is already a single test class `EventRegistrationApplicationTests` in the *src/test/java* folder that looks like the following:

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

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class EventRegistrationApplicationTests {

    @Test
    public void contextLoads() {

    }

}
```

2. Run this test that checks if the application can successfully load by *right clicking on the class → Run as... → JUnit test*

IMPORTANT	You need to set the `SPRING_DATASOURCE_URL` for the test run configuration as well if you use an environment variable to set datasource URL. See step 5 in section 3.3.1.

3. Add a new test class `ca.mcgill.ecse321.eventregistration.service.TestEventRegistrationService` and implement tests for the service

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

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;

import java.sql.Date;
import java.sql.Time;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.util.Calendar;
import java.util.List;

import org.junit.After;
import org.junit.Test;
```

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import ca.mcgill.ecse321.eventregistration.dao.EventRepository;
import ca.mcgill.ecse321.eventregistration.dao.PersonRepository;
import ca.mcgill.ecse321.eventregistration.dao.RegistrationRepository;
import ca.mcgill.ecse321.eventregistration.model.Event;
import ca.mcgill.ecse321.eventregistration.model.Person;

@RunWith(SpringRunner.class)
@SpringBootTest
public class TestEventRegistrationService {

    @Autowired
    private EventRegistrationService service;

    @Autowired
    private PersonRepository personRepository;
    @Autowired
    private EventRepository eventRepository;
    @Autowired
    private RegistrationRepository registrationRepository;

    @After
    public void clearDatabase() {
        // Fisrt, we clear registrations to avoid exceptions due to inconsistencies
        registrationRepository.deleteAll();
        // Then we can clear the other tables
        personRepository.deleteAll();
        eventRepository.deleteAll();
    }

    @Test
    public void testCreatePerson() {
        assertEquals(0, service.getAllPersons().size());

        String name = "Oscar";

        try {
            service.createPerson(name);
        } catch (IllegalArgumentException e) {
            // Check that no error occurred
            fail();
        }

        List<Person> allPersons = service.getAllPersons();

        assertEquals(1, allPersons.size());
        assertEquals(name, allPersons.get(0).getName());
```

```
    }

    @Test
    public void testCreatePersonNull() {
        assertEquals(0, service.getAllPersons().size());

        String name = null;
        String error = null;

        try {
            service.createPerson(name);
        } catch (IllegalArgumentException e) {
            error = e.getMessage();
        }

        // check error
        assertEquals("Person name cannot be empty!", error);

        // check no change in memory
        assertEquals(0, service.getAllPersons().size());

    }

    // ... other tests

    @Test
    public void testRegisterPersonAndEventDoNotExist() {
        assertEquals(0, service.getAllRegistrations().size());

        String nameP = "Oscar";
        Person person = new Person();
        person.setName(nameP);
        assertEquals(0, service.getAllPersons().size());

        String nameE = "Soccer Game";
        Calendar c = Calendar.getInstance();
        c.set(2016, Calendar.OCTOBER, 16, 9, 00, 0);
        Date eventDate = new Date(c.getTimeInMillis());
        Time startTime = new Time(c.getTimeInMillis());
        c.set(2016, Calendar.OCTOBER, 16, 10, 30, 0);
        Time endTime = new Time(c.getTimeInMillis());
        Event event = new Event();
        event.setName(nameE);
        event.setDate(eventDate);
        event.setStartTime(startTime);
        event.setEndTime(endTime);
        assertEquals(0, service.getAllEvents().size());

        String error = null;
        try {
            service.register(person, event);
```

```
        } catch (IllegalArgumentException e) {
            error = e.getMessage();
        }

        // check error
        assertEquals("Person does not exist! Event does not exist!", error);

        // check model in memory
        assertEquals(0, service.getAllRegistrations().size());
        assertEquals(0, service.getAllPersons().size());
        assertEquals(0, service.getAllEvents().size());

    }


    // ... other tests
}
```

4. See the complete test suite here.
5. Run the tests and interpret the test error messages! You should see only a few (at least one) tests passing.
6. Update the implementation (i.e., replace the current service method codes with the ones provided below) of the following methods with input validation in the `EventRegistrationService` service class to make the tests pass (Test-Driven Development)

```
@Transactional
public Person createPerson(String name) {
    if (name == null || name.trim().length() == 0) {
        throw new IllegalArgumentException("Person name cannot be empty!");
    }
    Person person = new Person();
    person.setName(name);
    personRepository.save(person);
    return person;
}
```

```
@Transactional
public Person getPerson(String name) {
    if (name == null || name.trim().length() == 0) {
        throw new IllegalArgumentException("Person name cannot be empty!");
    }
    Person person = personRepository.findPersonByName(name);
    return person;
}
```

```
@Transactional
public Event getEvent(String name) {
    if (name == null || name.trim().length() == 0) {
        throw new IllegalArgumentException("Event name cannot be empty!");
    }
    Event event = eventRepository.findEventByName(name);
    return event;
}
```

```
@Transactional
public Event createEvent(String name, Date date, Time startTime, Time endTime) {
    // Input validation
    String error = "";
    if (name == null || name.trim().length() == 0) {
        error = error + "Event name cannot be empty! ";
    }
    if (date == null) {
        error = error + "Event date cannot be empty! ";
    }
    if (startTime == null) {
        error = error + "Event start time cannot be empty! ";
    }
    if (endTime == null) {
        error = error + "Event end time cannot be empty! ";
    }
    if (endTime != null && startTime != null && endTime.before(startTime)) {
        error = error + "Event end time cannot be before event start time!";
    }
    error = error.trim();
    if (error.length() > 0) {
        throw new IllegalArgumentException(error);
    }

    Event event = new Event();
    event.setName(name);
    event.setDate(date);
    event.setStartTime(startTime);
    event.setEndTime(endTime);
    eventRepository.save(event);
    return event;
}
```

```
@Transactional
public Registration register(Person person, Event event) {
    String error = "";
    if (person == null) {
        error = error + "Person needs to be selected for registration! ";
    } else if (!personRepository.existsById(person.getName())) {
        error = error + "Person does not exist! ";
    }
    if (event == null) {
        error = error + "Event needs to be selected for registration!";
    } else if (!eventRepository.existsById(event.getName())) {
        error = error + "Event does not exist!";
    }
    if (registrationRepository.existsByPersonAndEvent(person, event)) {
        error = error + "Person is already registered to this event!";
    }
    error = error.trim();

    if (error.length() > 0) {
        throw new IllegalArgumentException(error);
    }

    Registration registration = new Registration();
    registration.setId(person.getName().hashCode() * event.getName().hashCode());
    registration.setPerson(person);
    registration.setEvent(event);

    registrationRepository.save(registration);

    return registration;
}
```

```
@Transactional
public List<Event> getEventsAttendedByPerson(Person person) {
    if (person == null ) {
        throw new IllegalArgumentException("Person cannot be null!");
    }
    List<Event> eventsAttendedByPerson = new ArrayList<>();
    for (Registration r : registrationRepository.findByPerson(person)) {
        eventsAttendedByPerson.add(r.getEvent());
    }
    return eventsAttendedByPerson;
}
```

7. Run the tests again, and all should be passing this time.

2.7. Creating RESTful Web Services in Spring

Previously, we used CRUD repository objects for each data type of the domain model and implemented use-cases in service classes. In this section, we will Implement REST API for eventregistration (people, events and registrations) and expose Sping Data.

If you would like to, you can obtain a version of the project that already has the changes from the previous tutorials here.

2.7.1. Preliminaries

1. Set database URL to start application and accessing the database remotey. You can do the same by either adding an environment variable called `SPRING_DATASOURCE_URL` or by specifying `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password` in *src/main/resources/application.properties*.
2. Add the dependency 'spring-boot-starter-data-rest' in *build.gradle* file of your backend. It is required to expose Spring Data repositories over REST using Spring Data REST.

```
implementation 'org.springframework.boot:spring-boot-starter-data-rest'
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
implementation 'org.springframework.boot:spring-boot-starter-web'

runtimeOnly 'org.postgresql:postgresql'
testImplementation 'org.springframework.boot:spring-boot-starter-test'
```

2.7.2. Building a RESTful Web Service

We will discuss two ways to build RESTful web service for our Spring Boot project.

Building a RESTful Web Service Using Controller and DTOs

1. We will first create a new package in EventRegistration-Backend and then create EventRegistrationRestController class inside it. We have added the annotation `@RestController` above the controller class so that HTTP requests are handled by EventRegistrationRestController. In addition, we enabled the Cross-Origin Resource Sharing at the controller level using '@CrossOrigin' notation.

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

@CrossOrigin(origins = "*")
@RestController
public class EventRegistrationRestController {
}
```

2. We further create another package ca.mcgill.ecse321.eventregistration.dto and create the below files inside that package. First we create EventDto.java.

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

import java.sql.Date;
import java.sql.Time;

public class EventDto {

    private String name;
    private Date eventDate;
    private Time startTime;
    private Time endTime;

    public EventDto() {
    }

    public EventDto(String name) {
        this(name, Date.valueOf("1971-01-01"), Time.valueOf("00:00:00"),
Time.valueOf("23:59:59"));
    }

    public EventDto(String name, Date eventDate, Time startTime, Time endTime) {
        this.name = name;
        this.eventDate = eventDate;
        this.startTime = startTime;
        this.endTime = endTime;
    }

    public String getName() {
        return name;
    }

    public Date getEventDate() {
        return eventDate;
    }

    public Time getStartTime() {
        return startTime;
    }

    public Time getEndTime() {
        return endTime;
    }

}
```

3. Next, we create `PersonDto` Java class.

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

import java.util.Collections;
import java.util.List;

public class PersonDto {

    private String name;
    private List<EventDto> events;

    public PersonDto() {
    }

    @SuppressWarnings("unchecked")
    public PersonDto(String name) {
        this(name, Collections.EMPTY_LIST);
    }

    public PersonDto(String name, List<EventDto> arrayList) {
        this.name = name;
        this.events = arrayList;
    }

    public String getName() {
        return name;
    }

    public List<EventDto> getEvents() {
        return events;
    }

    public void setEvents(List<EventDto> events) {
        this.events = events;
    }

}
```

4. Finally, we create `RegistrationDto` Java class.

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

public class RegistrationDto {

    private PersonDto person;
    private EventDto event;

    public RegistrationDto() {
    }

    public RegistrationDto(PersonDto person, EventDto event) {
        this.person = person;
        this.event = event;
    }

    public PersonDto getperson() {
        return person;
    }

    public void setperson(PersonDto person) {
        this.person = person;
    }

    public EventDto getEvent() {
        return event;
    }

    public void setEvent(EventDto event) {
        this.event = event;
    }
}
```

5. Now, we will add the methods in the controller class. Also, we will add annotaions to map web requests.

```
    @PostMapping(value = { "/persons/{name}", "/persons/{name}/" })
    public PersonDto createPerson(@PathVariable("name") String name) throws
IllegalArgumentException {
        // @formatter:on
        Person person = service.createPerson(name);
        return convertToDto(person);
    }
```

@RequestMapping annotation is used to map web requests to Spring Controller methods. Since, @RequestMapping maps all HTTP operations by default. We can use @GetMapping, @PostMapping and so forth to narrow this mapping.

Moreover, in the above snippet, we use "value" and @PathVariable to bind the value of the query

string parameter name into the name parameter of the createPerson() method.

1. You can add other methods similarly with appropriate mappings.

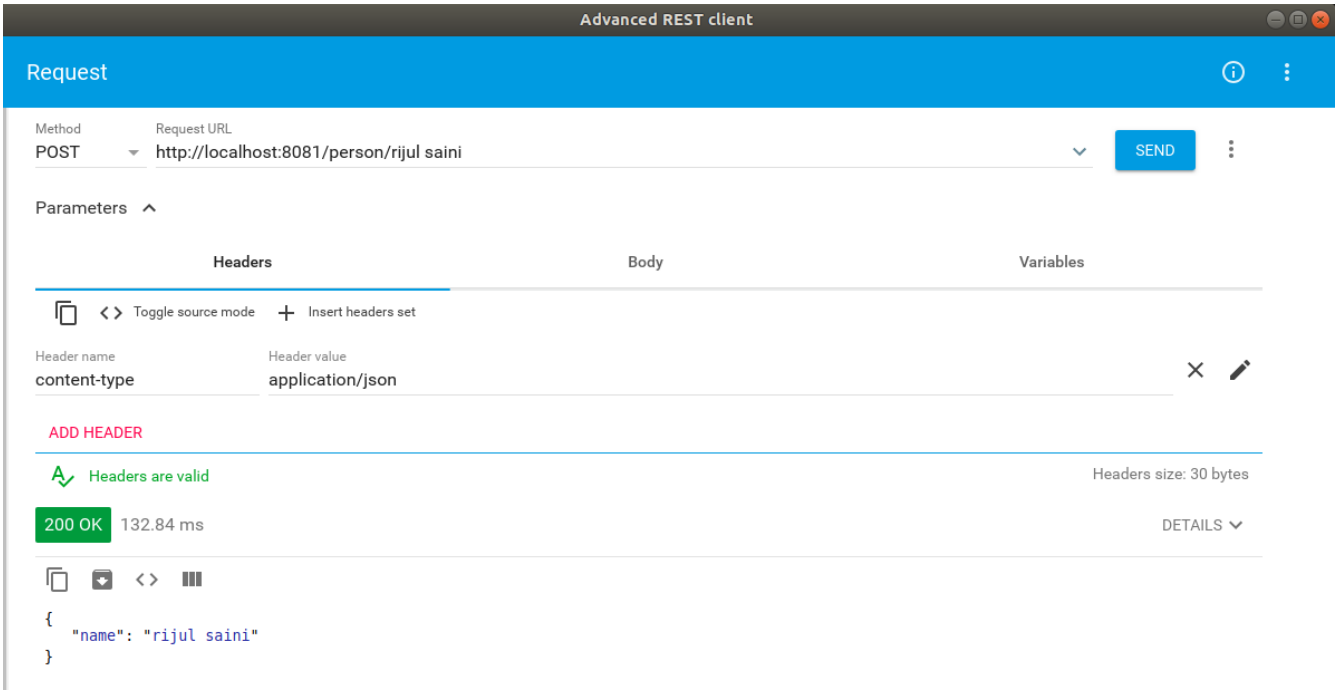
```
@PostMapping(value = { "/events/{name}", "/events/{name}/" })
public EventDto createEvent(@PathVariable("name") String name, @RequestParam Date
date,
@RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.TIME, pattern = "HH:mm")
LocalTime startTime,
@RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.TIME, pattern = "HH:mm")
LocalTime endTime)
throws IllegalArgumentException {
    Event event = service.createEvent(name, date, Time.valueOf(startTime),
Time.valueOf(endTime));
    return convertToDto(event);
}

@GetMapping(value = { "/events", "/events/" })
public List<EventDto> getAllEvents() {
    List<EventDto> eventDtos = new ArrayList<>();
    for (Event event : service.getAllEvents()) {
        eventDtos.add(convertToDto(event));
    }
    return eventDtos;
}

@PostMapping(value = { "/register", "/register/" })
public RegistrationDto registerPersonForEvent(@RequestParam(name = "person")
PersonDto pDto,
    @RequestParam(name = "event") EventDto eDto) throws IllegalArgumentException {
    Person p = service.getPerson(pDto.getName());
    Event e = service.getEvent(eDto.getName());

    Registration r = service.register(p, e);
    return convertToDto(r, p, e);
}

@GetMapping(value = { "/registrations/person/{name}",
"/registrations/person/{name}/" })
public List<EventDto> getEventsOfPerson(@PathVariable("name") PersonDto pDto) {
    Person p = convertToDomainObject(pDto);
    return createEventDtosForPerson(p);
}

@GetMapping(value = { "/events/{name}", "/events/{name}/" })
public EventDto getEventByName(@PathVariable("name") String name) throws
IllegalArgumentException {
    return convertToDto(service.getEvent(name));
}

private EventDto convertToDto(Event e) {
```

```
    if (e == null) {
        throw new IllegalArgumentException("There is no such Event!");
    }
    EventDto eventDto = new
EventDto(e.getName(),e.getDate(),e.getStartTime(),e.getEndTime());
    return eventDto;
}

private PersonDto convertToDto(Person p) {
    if (p == null) {
        throw new IllegalArgumentException("There is no such Person!");
    }
    PersonDto personDto = new PersonDto(p.getName());
    personDto.setEvents(createEventDtosForPerson(p));
    return personDto;
}

private RegistrationDto convertToDto(Registration r, Person p, Event e) {
    EventDto eDto = convertToDto(e);
    PersonDto pDto = convertToDto(p);
    return new RegistrationDto(pDto, eDto);
}

private Person convertToDomainObject(PersonDto pDto) {
    List<Person> allPersons = service.getAllPersons();
    for (Person person : allPersons) {
        if (person.getName().equals(pDto.getName())) {
            return person;
        }
    }
    return null;
}

private List<EventDto> createEventDtosForPerson(Person p) {
    List<Event> eventsForPerson = service.getEventsAttendedByPerson(p);
    List<EventDto> events = new ArrayList<>();
    for (Event event : eventsForPerson) {
        events.add(convertToDto(event));
    }
    return events;
}
```

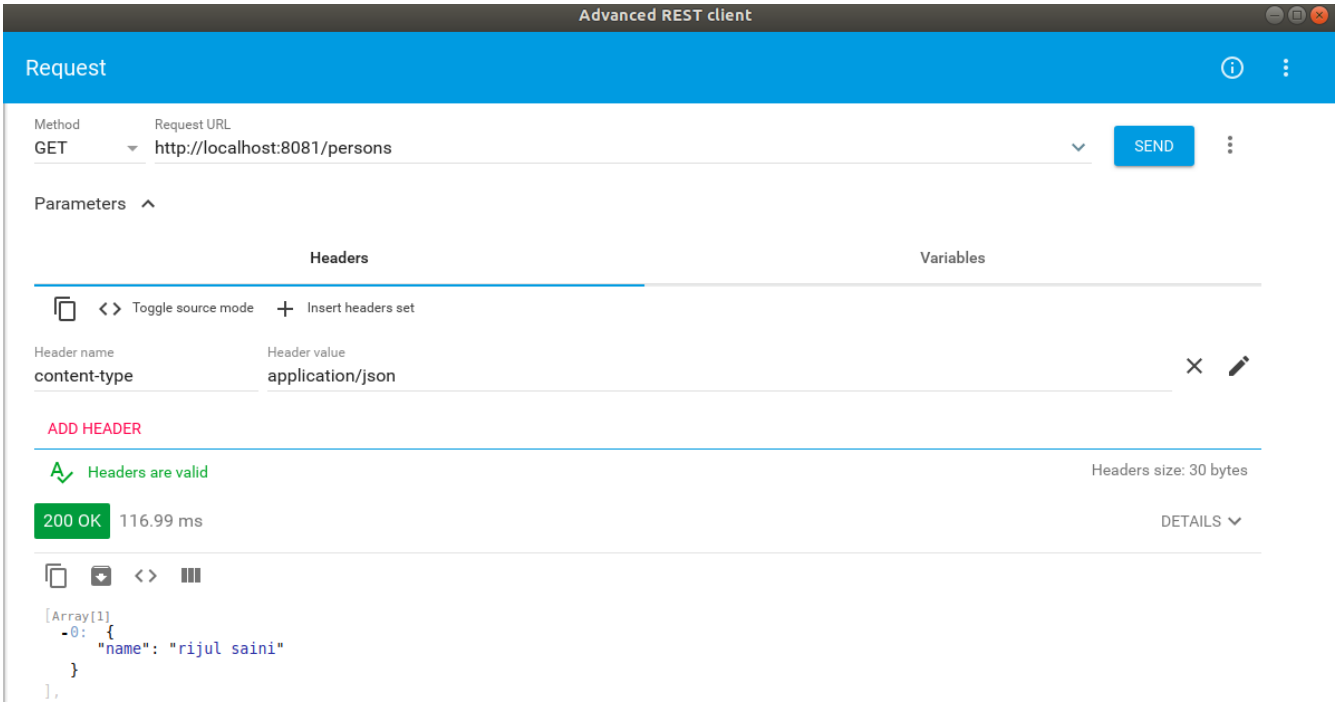
2.7.3. Test the Service

We can test the application using, e.g., the RESTClient browser plugin, Advanced Rest Client, Postman or curl.

Once you launch the client, you can specify the path and select the method as shown in the below figures.



Once we use POST, the record is persisted and then we can use the GET method to retrive the same.



Similary, we can test the other methods.

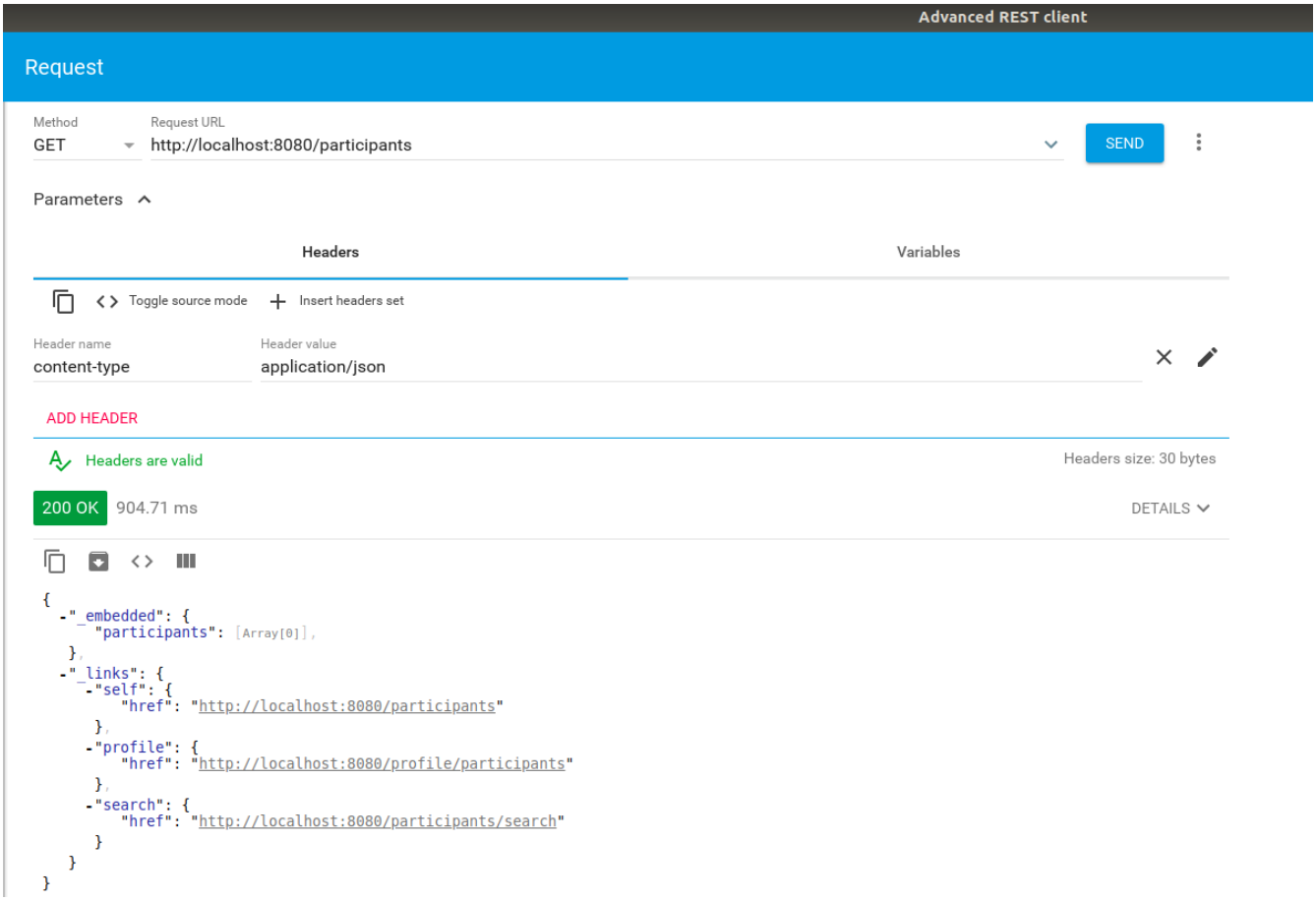
2.7.4. Spring Data - an Alternative to Exposing the Database

The advantage of using Spring Data Rest is that it can remove a lot of boilerplate that's natural to REST services. Spring would automatically create endpoints like /events, /people as we saw above and these endpoints can be further customized.

1. We have already added the dependency 'spring-boot-starter-data-rest' in preliminary section to to expose Spring Data repositories over REST using Spring Data REST.
2. Next, we can go to repository interfaces and add @RepositoryRestResource annotaion.

```
@RepositoryRestResource(collectionResourceRel = "participants", path =
"participants")
public interface PersonRepository extends CrudRepository<Person, String>{

    Person findPersonByName(String name);

}
```

3. Finally, we can access this REST API in the browser or REST Client and will receive the JSON as shown below.



2.8. Testing Backend Services

We implement a first unit test for testing the application service logic.

2.8.1. Preparations

1. Open the project and ensure that you add *JUnit 5* and *Mockito 2+* to the project dependencies in `build.gradle`:

```
dependencies {
  // Add these lines to the dependency configuration, don't replace the existing
dependencies
    testImplementation "junit:junit:4.12"
    testRuntime('org.junit.jupiter:junit-jupiter-engine:5.3.1')
    testImplementation('org.mockito:mockito-core:2.+')
    testImplementation('org.mockito:mockito-junit-jupiter:2.18.3')
}
```

> **NOTE** Finding configuration settings for your Gradle/Maven project is very simple by searaching for them on MVNRepository: https://mvnrepository.com/

2. If you also would like to run your project from Eclipse, add an additional dependency:

```
testImplementation group: 'org.junit.platform', name: 'junit-platform-launcher',
version: "1.3.1"
```

3. Create a test class (in case you don't already have one) `EventregistrationServiceTests` in the corresponding package under `src/test/java`:

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

@RunWith(MockitoJUnitRunner.class)
public class EventregistrationServiceTests {

}
```

4. Build your project to ensure its dependencies are correctly loaded.

2.8.2. Writing tests

1. First, update/optimize the `createPerson` service method:

```
@Transactional
public Person createPerson(String name) {
  if (name == null || name == "") {
    throw new IllegalArgumentException("Person name cannot be empty!");
  }
  Person person = personRepository.findPersonByName(name);
  if ( person == null ) {
    person = new Person();
    person.setName(name);
    personRepositroy.save(person);
  }
  return person;
}
```

2. Add the following imports to the test class:

```
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.springframework.boot.test.context.SpringBootTest;
import ca.mcgill.ecse321.eventregistration.dao.EventRepository;
import ca.mcgill.ecse321.eventregistration.dao.PersonRepository;
import ca.mcgill.ecse321.eventregistration.dao.RegistrationRepository;
import ca.mcgill.ecse321.eventregistration.service.EventRegistrationService;
import org.mockito.invocation.InvocationOnMock;
import org.springframework.test.context.junit4.SpringRunner;
import
ca.mcgill.ecse321.eventregistration.controller.EventRegistrationRestController;
import ca.mcgill.ecse321.eventregistration.model.Person;
```

3. Add the following static imports for methods:

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.mockito.ArgumentMatchers.anyString;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
```

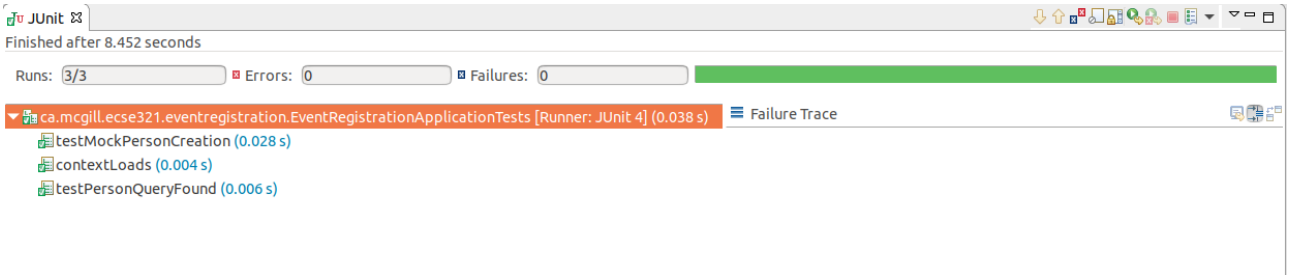
4. Create the DAO mock for person

```
@Mock
private PersonRepository personDao;

@InjectMocks
private EventRegistrationService service;

private static final String PERSON_KEY = "TestPerson";
private static final String NONEXISTING_KEY = "NotAPerson";

@Before
public void setMockOutput() {
    when(personDao.findPersonByName(anyString())).thenAnswer( (InvocationOnMock
invocation) -> {
        if(invocation.getArgument(0).equals(PERSON_KEY)) {
            Person person = new Person();
            person.setName(PERSON_KEY);
            return person;
        } else {
            return null;
        }
    });
}
```

5. Add test cases

```
@Test
public void testCreatePerson() {
    assertEquals(0, service.getAllPersons().size());

    String name = "Oscar";

    try {
        person = service.createPerson(name);
    } catch (IllegalArgumentException e) {
        // Check that no error occurred
        fail();
    }

    assertEquals(name, person.getName());
}

@Test
public void testCreatePersonNull() {
    String name = null;
    String error = null;

    try {
        person = service.createPerson(name);
    } catch (IllegalArgumentException e) {
        error = e.getMessage();
    }

    // check error
    assertEquals("Person name cannot be empty!", error);
}
```

6. Run the tests with `gradle test` from the command line in the root of the project, or in Eclipse, right click on the test class name then select *Run As... > JUnit test*.



2.9. Code Coverage using EclEmma

This tutorial covers the basics of EclEmma and retrieves code coverage metrics using it.

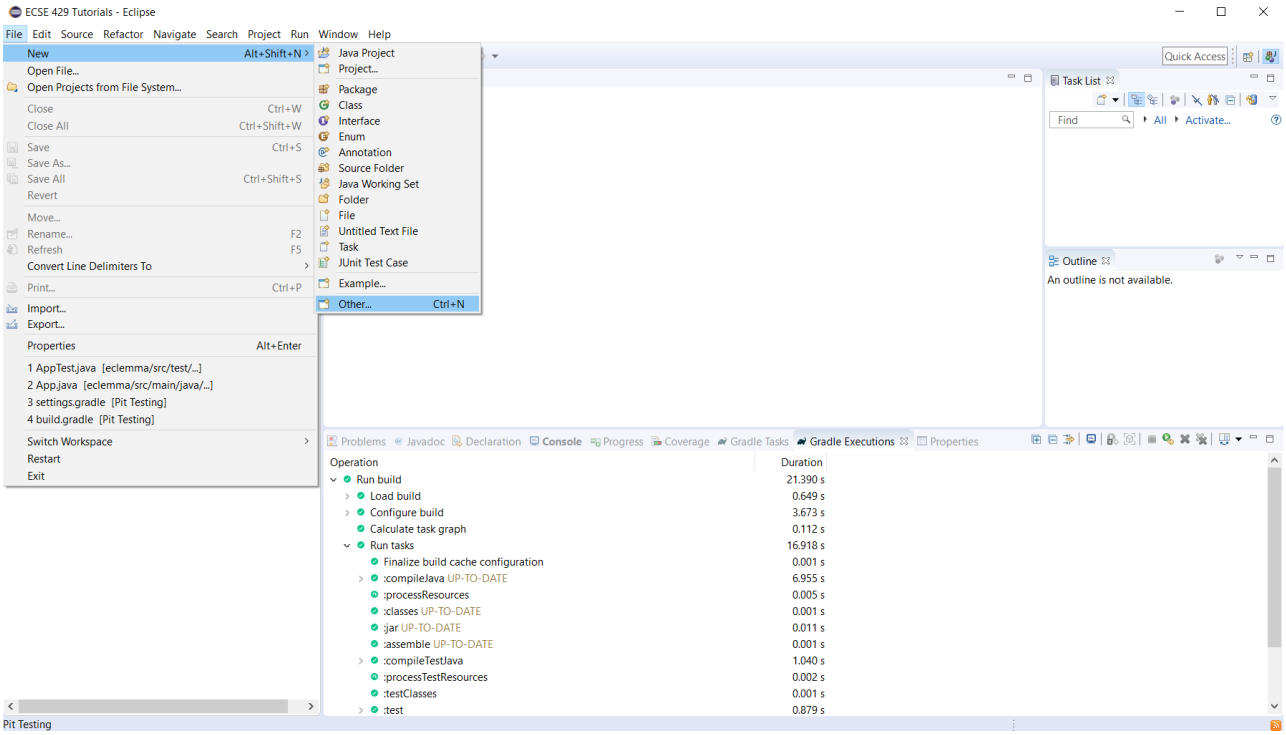
2.9.1. Preliminary

1. Install EclEmma as a plugin in your Eclipse IDE from here.

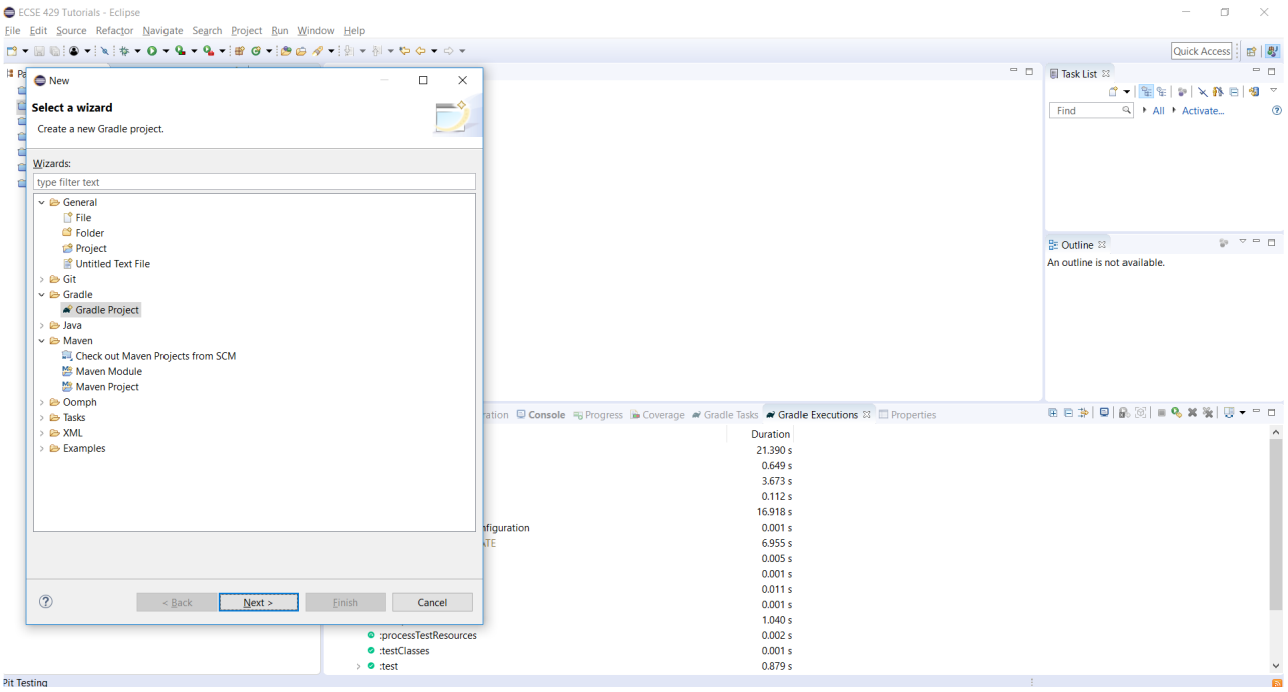
2.9.2. Creating a Gradle Project

NOTE	We will create a Gradle project from scratch and be testing a simple method *returnAverage(int[], int, int, int)* .

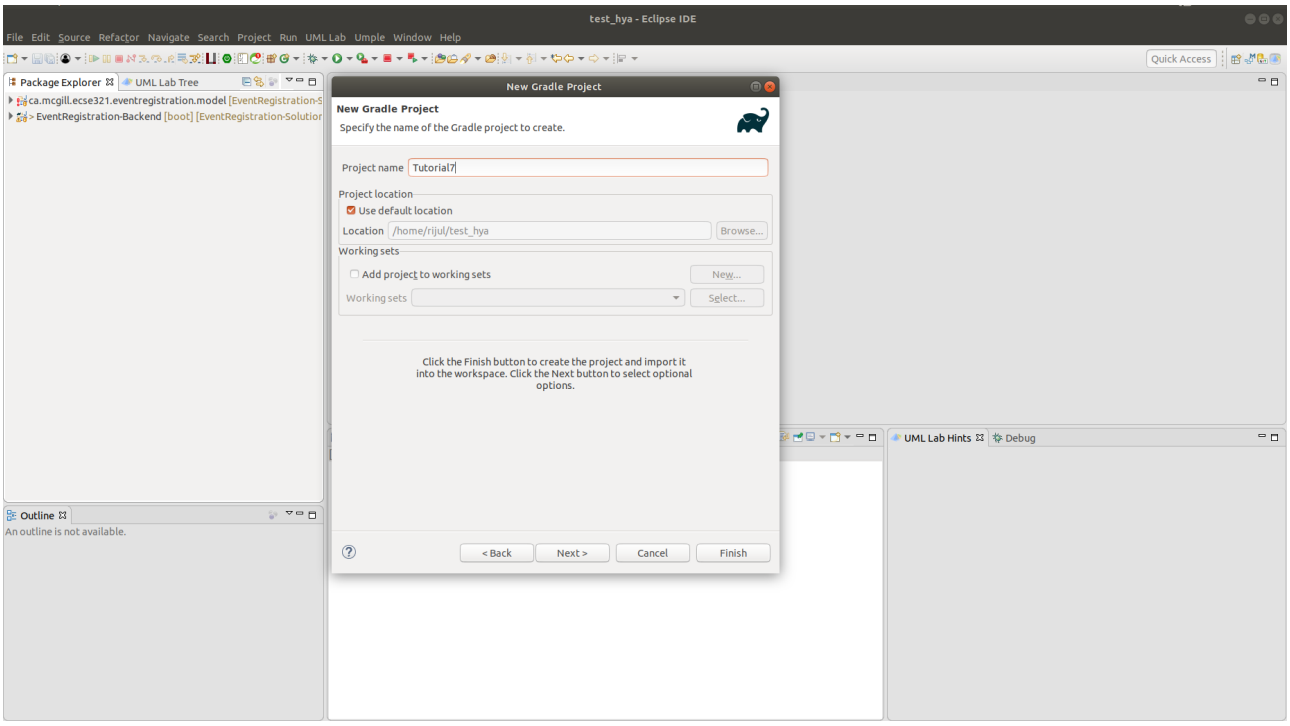
1. Create a new Gradle project in Eclipse by clicking on *File* > *New* > *Other*



2. Under *Gradle*, choose Gradle Project

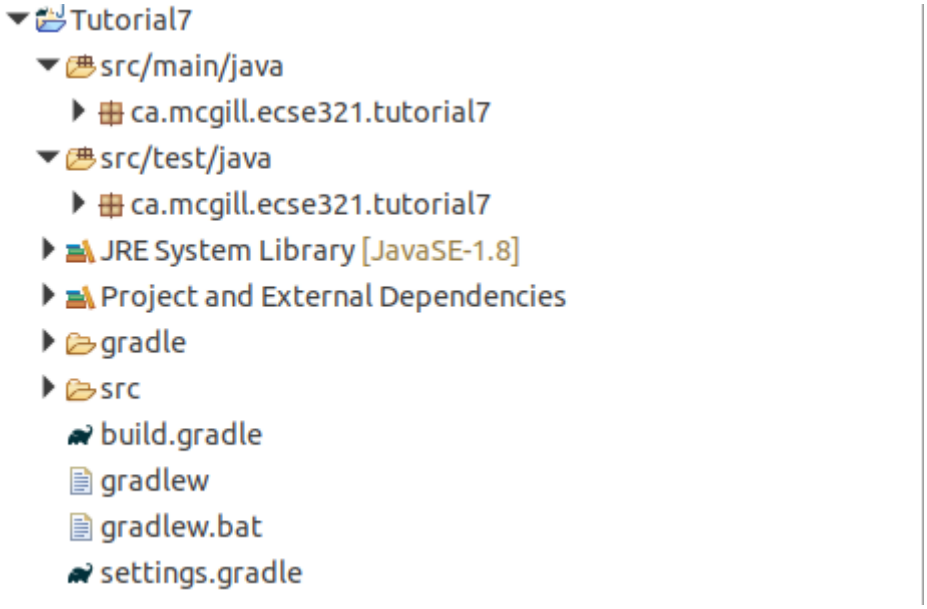


3. Click on *Next*, then name your project *tutorial7*, click on *Finish*



> **NOTE** The project may take some time to be created.

4. Create a new package instead of the default ones for both the source and test folders (e.g `ca.mcgill.ecse321.tutorial7`) and move the default generated classes (`Library` and `LibraryTest`) to this package.



5. Change the code in the `Library` class

```
package ca.mcgill.ecse321.tutorial7;

public class Library {

    public static double returnAverage(int value[], int arraySize, int MIN, int
MAX) {
        int index, ti, tv, sum;
        double average;
        index = 0;
        ti = 0;
        tv = 0;
        sum = 0;
        while (ti < arraySize && value[index] != -999) {
            ti++;
            if (value[index] >= MIN && value[index] <= MAX) {
                tv++;
                sum += value[index];
            }
            index++;
        }
        if (tv > 0)
            average = (double) sum / tv;
        else
            average = (double) -999;
        return average;
    }
}
```

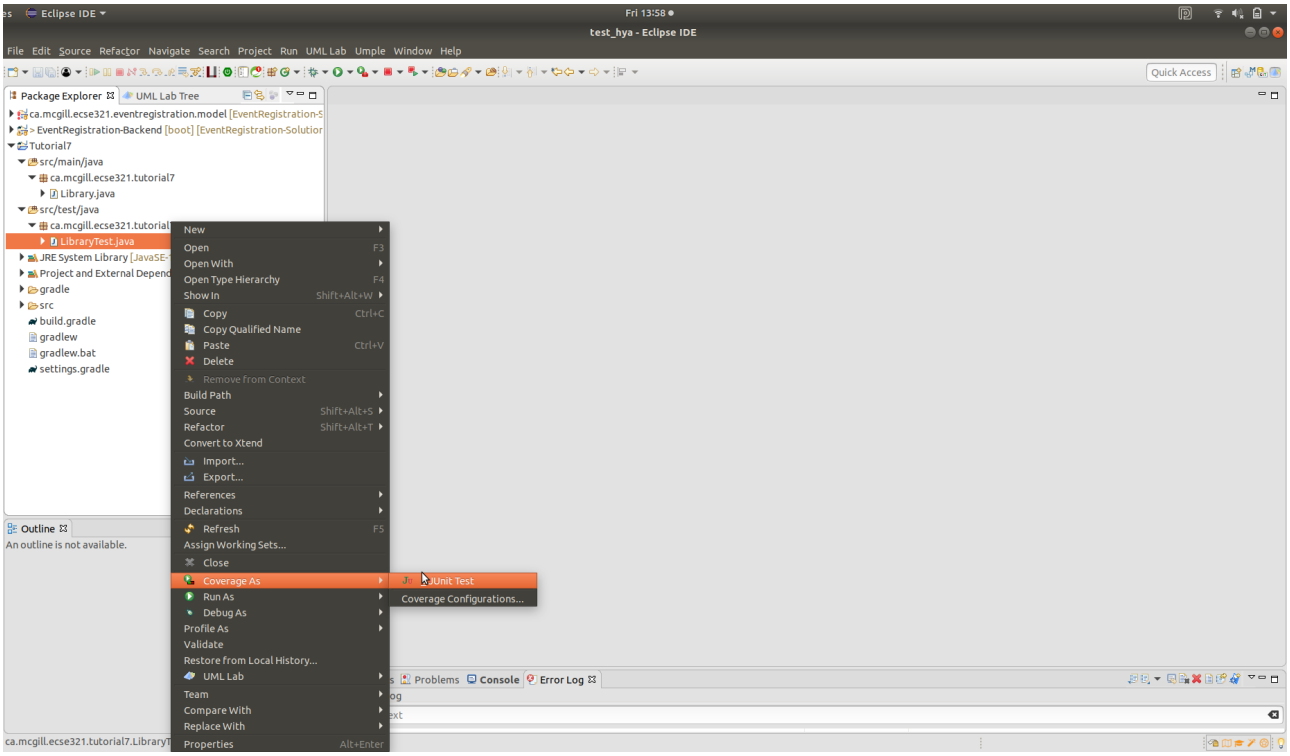
6. Change the code in the `LibraryTest` class

```
package ca.mcgill.ecse321.tutorial7;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class LibraryTest {

    @Test
    public void allBranchCoverageMinimumTestCaseForReturnAverageTest1() {
        int[] value = {5, 25, 15, -999};
        int AS = 4;
        int min = 10;
        int max = 20;
        double average = Library.returnAverage(value, AS, min, max);
        assertEquals(15, average, 0.1);
    }

    @Test
    public void allBranchCoverageMinimumTestCaseForReturnAverageTest2() {
        int[] value = {};
        int AS = 0;
        int min = 10;
        int max = 20;
        double average = Library.returnAverage(value, AS, min, max);
        assertEquals(-999.0, average, 0.1);
    }
}
```

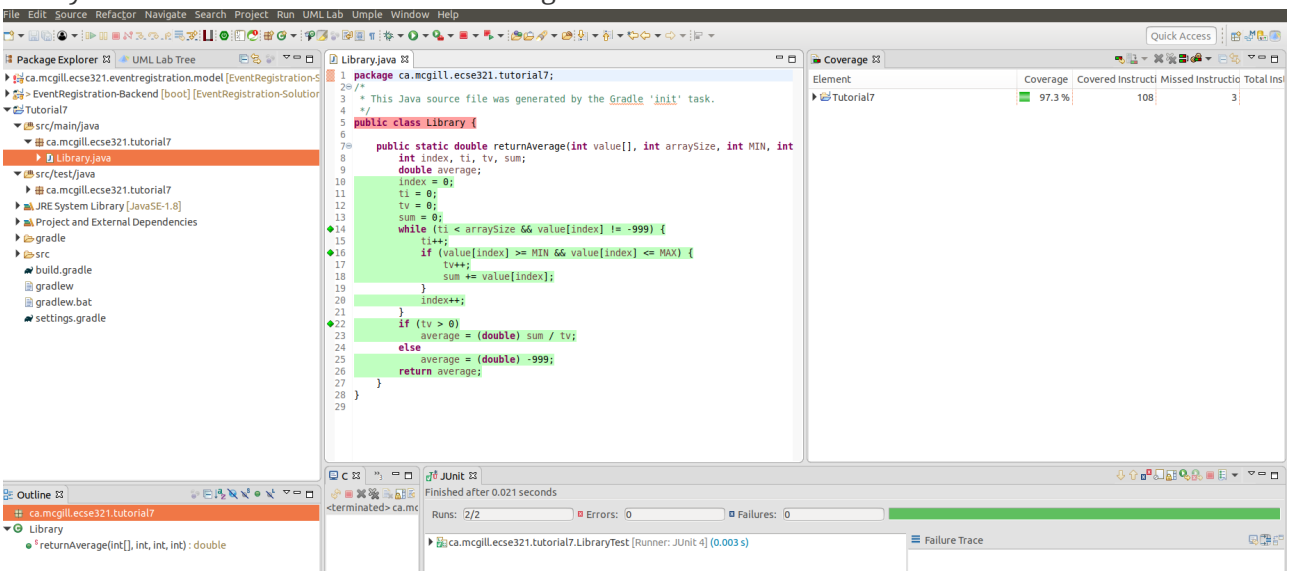
2.9.3. Retrieving Test Coverage Metrics

NOTE	We can straightforwardly manage code coverage using JaCoCo inside Eclipse with no configuration if we are using EclEmma Eclipse plugin.

1. Run the Test in coverage mode using Eclemma. Click on *LibraryTest*, *Coverage As*, *1 JUnit Test*



2. Verify that we have 100% branch coverage.




```
package ca.mcgill.ecse321.tutorial7;
/*
 * This Java source file was generated by the Gradle 'init' task.
 */
public class Library {

    public static double returnAverage(int value[], int arraySize, int MIN, int
        int index, ti, tv, sum;
        double average;
        index = 0;
        ti = 0;
        tv = 0;
        sum = 0;
        while (ti < arraySize && value[index] != -999) {
            ti++;
            if (value[index] >= MIN && value[index] <= MAX) {
                tv++;
                sum += value[index];
            }
            index++;
        }
        if (tv > 0)
            average = (double) sum / tv;
        else
            average = (double) -999;
        return average;
    }
}
```

3. Web Frontend

3.1. Installation Instructions: Vue.js

Vue.js is a popular web frontend for building user interfacs in Javascript, which is considered to be easier to learn compared to React and Angular.

3.1.1. Install Vue.js

1. Open a shell (or run **cmd.exe** in Windows)
2. Check that you successfully installed **node.js** and **npm** e.g. by checking their versions:

```
$ node -v
v8.10.0
$ npm -v
3.5.2
```

3. Install the command line interface (CLI) for Vue: `sudo npm install --global vue-cli`

3.1.2. Generate initial Vue.js project content

1. Navigate to your local Git repository of the Event Registration System

```
$ cd ~/git/eventregistration
```

1. Generate initial content as follows
   - Hit `Enter` after each line if not indicated otherwise
   - Detailed instructions at https://github.com/vuejs-templates/webpack and https://bootstrap-vue.js.org/docs

```
$ vue init bootstrap-vue/webpack eventregistration-frontend
? Project name (EventRegistration-Frontend) EventRegistration-Frontend
? Project description (A Vue.js project) A Vue.js frontend for Event
Registration App
? Author (varrodan <daniel.varro@gmail.com>)
? Vue build (Use arrow keys):
> Runtime + Compiler
  Runtime-only
? Install vue-router (Y/n): Y
? Use ESLint to lint your code (Y/n): n
? Setup unit tests with Karma + Mocha (Y/n) n
? Setup e2e tests with Nightwatch (Y/n) Y

vue-cli · Generated "EventRegistration-Frontend".
```

2. Now execute the following commands (one after the other)

```
$ cd EventRegistration-Frontend
$ npm install
$ npm run dev
```

3. As a result A sample web page should appear at http://localhost:8080/
4. You can stop this development server by pressing `Ctrl+C` in the shell

3.1.3. Install additional dependencies

1. Install JQuery and Axios (we will use these dependencies for issuing REST API calls):

```
npm install --save axios
npm install --save jquery
```

3.1.4. Setting up your development server

1. We change the default port to **8087** (instead of the default 8080) and the default IP address by using a configuration file. The rationale behind this step is that other Tomcat servers may already listen at the default localhost:8080 port which may clash with our development server.
2. Open **./config/index.js** and add `port: 8087` to `module.exports` (both `build` and `dev` part)
   - The development server is set up at localhost, i.e. http://127.0.0.1:8087
   - The production server is set up in accordance with the virtual machines
   - We also store the host IP address and port of the backend server in similar environment variables (`backendHost` and `backendPort`).

```
module.exports = {
  build: {
    env: require('./prod.env'),
    host: 'eventregistration-frontend-123.herokuapp.com',
    port: 443,
    backendHost: 'eventregistration-backend-123.herokuapp.com',
    backendPort: 443,
    //...
  },
  dev: {
    env: require('./dev.env'),
    host: '127.0.0.1',
    port: 8087,
    backendHost: '127.0.0.1',
    backendPort: 8080,
    //...
  }
}
```

3. Open **./build/dev-server.js**, and change the `uri` assignment as follows:
    - The original line of code can be commented or deleted.

```
//var uri = 'http://localhost:' + port
var host = config.dev.host
var uri = 'http://' + host + ':' + port
```

4. Start again your development server by `npm run dev`. The same web application should now appear at http://127.0.0.1:8087/
5. Stop the development server by `Ctrl+C`.

3.1.5. Commit your work to Github

1. If everything works then commit your work to your Github repository.
2. Notice that many libraries and files are omitted, which is intentional. Check the `.gitignore` file for details.

3.2. Create a Static Vue.js Component

Vue.js promotes the use of components which encapsulate GUI elements and their behavior in order to build up rich user interfaces in a modular way. A component consists of

- **template**: A template of (a part of) an HTML document enriched with data bindings, conditional expressions, loops, etc.
- **script**: The behavior of the user interface programmed in JavaScript.
- **style**: The customized graphical appearance of HTML document elements.

We will first create a new Vue.js component and then connect it to a backend Java Spring service via a Rest API call.

3.2.1. Create a component file

NOTE	We use `.` below to refer to the `EventRegistration-Frontend` directory.

1. Create a new file **EventRegistration.vue** in **./src/components** with the following initial content:

```
<template>
</template>
<script>
</script>
<style>
</style>
```

2. Create some static HTML content of the template part starting with a `<div>` element corresponding to your component. We

```
<template>
  <div id="eventregistration">
    <h2>People</h2>
    <table>
      <tr>
          <td>John</td>
          <td>Event to attend</td>
      </tr>
      <tr>
          <td>
              <input type="text" placeholder="Person Name">
          </td>
          <td>
              <button>Create</button>
          </td>
      </tr>
    </table>
    <p>
      <span style="color:red">Error: Message text comes here</span>
    </p>
  </div>
</template>
```

3. Customize the `<style>` part with your designated CSS content. A detailed CSS reference documentation is available at https://www.w3schools.com/CSSref/. The final result of that part should like as follows.

```
<style>
  #eventregistration {
    font-family: 'Avenir', Helvetica, Arial, sans-serif;
    color: #2c3e50;
    background: #f2ece8;
  }
</style>
```

3.2.2. Create a new routing command

1. We need to route certain HTTP calls to a specific URL to be handled by **EventRegistration.vue**.
2. Open **./src/router/index.js** and add a new route by extending the existing `routes` property.

```
export default new Router({
  routes: [
    {
      path: '/',
      name: 'Hello',
      component: Hello
    },
    {
      path: '/app',
      name: 'EventRegistration',
      component: EventRegistration
    }
  ]
})
```

- You should not change the number of spaces used as indentation otherwise you get error messages, if you have LInt enabled in your project.
- Import the new component `EventRegistration.vue` at the beginning of **./src/router/index.js** after all existing imports!

```
// add import after all existing imports
import EventRegistration from '@/components/EventRegistration'
```

3. Start the development server and navigate your browser to http://127.0.0.1:8087/#/app. Your new Vue.js component should be rendered (with the static HTML content).

3.3. Vue.js Components with Dynamic Content

3.3.1. Add data and event handlers

Next we add event handling and dynamic content to our EventRegistration.vue component.

1. Create another file **registration.js** in the same folder which will contain the Javascript code for the EventRegistration.vue component.
2. Create constructor methods:

```
function PersonDto (name) {
  this.name = name
  this.events = []
}

function EventDto (name, date, start, end) {
  this.name = name
  this.eventDate = date
  this.startTime = start
  this.endTime = end
}
```

3. Add data variables to the export declaration of the component.

```
export default {
  name: 'eventregistration',
  data () {
    return {
      people: [],
      newPerson: '',
      errorPerson: '',
      response: []
    }
  },
  //...
}
```

4. Add an initialization function below the data part.

```
created: function () {
  // Test data
  const p1 = new PersonDto('John')
  const p2 = new PersonDto('Jill')
  // Sample initial content
  this.people = [p1, p2]
},
```

5. Add event handling method **createPerson()**:

```
methods: {
  createPerson: function (personName) {
    // Create a new person and add it to the list of people
    var p = new PersonDto(personName)
    this.people.push(p)
    // Reset the name field for new people
    this.newPerson = ''
  }
}
```

3.3.2. Create dynamic data bindings

1. Open `EventRegistration.vue` and link the Javascript file as script:

```
<script src="./registration.js">
</script>
```

2. Change the static template content for the person list to dynamic bindings:
   - We iterate along all people in data property `people` and dynamically print their name by `{{ person.name }}` (see list rendering)
   - We print the (currently empty) list of events to which a person is registered to.

```
<table>
  <tr v-for="person in people" >
      <td>{{ person.name }}</td>
      <td>
        <ul>
          <li v-for="event in person.events">
            {{event.name}}
          </li>
        </ul>
      </td>
  </tr>
<!-- ... -->
</table>
```

3. Link input field content with data variable `newPerson` and button clicks for **Create Person** for event handler method `createPerson()`.

```
<table>
  <!-- ... -->
  <tr>
    <td>
        <input type="text" v-model="newPerson" placeholder="Person Name">
    </td>
    <td>
        <button @click="createPerson(newPerson)">Create Person</button>
    </td>
  </tr>
</table>
```

4. Bind the error message to the corresponding variable `errorPerson` by extending the `<span>` tag with conditional rendering.
   - The error message will only appear if the data property `errorPerson` is not empty.
   - You may wish to further refine error handling in case of empty string content for `newPerson` by adding `&& !newPerson` to the condition.

   ```
   <span v-if="errorPerson" style="color:red">Error: {{errorPerson}} </span>
   ```

5. Run your frontend application and observe that two people are listed.

3.4. Calling Backend Services

Next we change our frontend to issue calls to the backend via the Rest API provided by the Java Spring framework. Please refer to the section 3.6.2 where we enabled the Cross-Origin Resource Sharing at the controller level using '@CrossOrigin' notation.

3.4.1. Calling backend services in from Vue.js components

We need to modify our frontend to make calls to backend services.

1. Open **registration.js** and add the following content to the beginning:
   - Note that instead of hard-wired IP addresses and ports, details are given in a configuration file.

```
import axios from 'axios'
var config = require('../../config')

var frontendUrl = 'http://' + config.dev.host + ':' + config.dev.port
var backendUrl = 'http://' + config.dev.backendHost + ':' +
config.dev.backendPort

var AXIOS = axios.create({
  baseURL: backendUrl,
  headers: { 'Access-Control-Allow-Origin': frontendUrl }
})
```

2. Now navigate to the **created** function, and replace existing content with the following lines:

```
created: function () {
  // Initializing people from backend
    AXIOS.get(`/persons`)
    .then(response => {
      // JSON responses are automatically parsed.
      this.people = response.data
    })
    .catch(e => {
      this.errorPerson = e;
    });
}
```

3. Navigate to the **createPerson()** method and change its content as follows:

```
createPerson: function (personName) {
  AXIOS.post(`/persons/`+personName, {}, {})
  .then(response => {
    // JSON responses are automatically parsed.
    this.people.push(response.data)
    this.newPerson = ''
    this.errorPerson = ''
  })
  .catch(e => {
    var errorMsg = e.message
    console.log(errorMsg)
    this.errorPerson = errorMsg
  });
}
```

4. Run the frontend application and check that
    - New people can be added
    - They immediately appear in the people list.

3.5. Build and Travis-CI

The project should build using `npm run build`. This will create a *build/* directory in the frontend folder, where you can run the frontend server using `node build/dev-server.js`.

Travis-CI supports building nodejs projects. However, we do not want to run the default `npm test` command. Instead, the build should do `npm install` and `npm run build` only.

3.6. Additional steps in the tutorial

3.6.1. Steps to complete

The description of the next steps is intentionally high-level and sketchy to force you to face and solve several emerging problems.

You need to provide the following functionality by extending the Vue.js component:

1. **List all events (name, date, startTime, endTime)**
   - Introduce an array **events** in the frontend data store
   - Call the appropriate backend service to fill the contents
   - Provide a dynamic list in the component and bind it to **events**
2. **Create a new event (name, date, startTime, endTime)**
   - Introduce an object **newEvent** in the frontend data store with four properties (e.g. name, date, startTime, endTime).
     - Set the initial values of these properties to something
   - Provide a button to initiate creating a new event
   - Provide HTML input fields to set event details
   - Create a call to the appropriate backend service, i.e. **createEvent()**
   - Introduce an object **errorEvent** for error message related to event creation
   - Provide corresponding HTML field for displaying the error message (e.g. `<span>`), and set its appearance condition to the content of the error message
   - **Hint**: you can use the following input types for setting date and time

     ```
     <input type="date" v-model="newEvent.eventDate" placeholder="YYYY-MM-DD">
     <input type="time" v-model="newEvent.startTime" placeholder="HH:mm">
     ```

3. **Register a person to an event** (when a new event should occur in the list of events printed next to a person)
   - Provide a selection of people
     - You need a corresponding data variable (e.g. **selectedPerson**)
     - You can use the HTML `<select v-model="selectedPerson">` tag where each option (`<option>` tag with `v-for` Vue.js parameter) is filled dynamically from the list of people.
     - **Hint:** You can add a first disabled option as follows:

       ```
       <option disabled value="">Please select one</option>
       ```

   - Provide a selection of events in a similar way.
   - Provide a button to initiate registration

    - Enable the button only if both a person and an event are selected
4. In all use cases,
    - Report application specific errors if the backend service fails
    - Prevent to enter invalid data to backend

3.6.2. Further documentation

- Vue.js guide: https://vuejs.org/v2/guide/
- Vue.js API: https://vuejs.org/v2/api/
- Build commands: http://vuejs-templates.github.io/webpack/commands.html
- Vue.js and Webpack integration: http://vuejs-templates.github.io/webpack/env.html
- Html-Webpack: https://github.com/jantimon/html-webpack-plugin
- Vue Router: https://github.com/vuejs/vue-router
- Vue Router tutorial: https://scotch.io/tutorials/getting-started-with-vue-router

4. Mobile Frontend

4.1. Setting up your repository

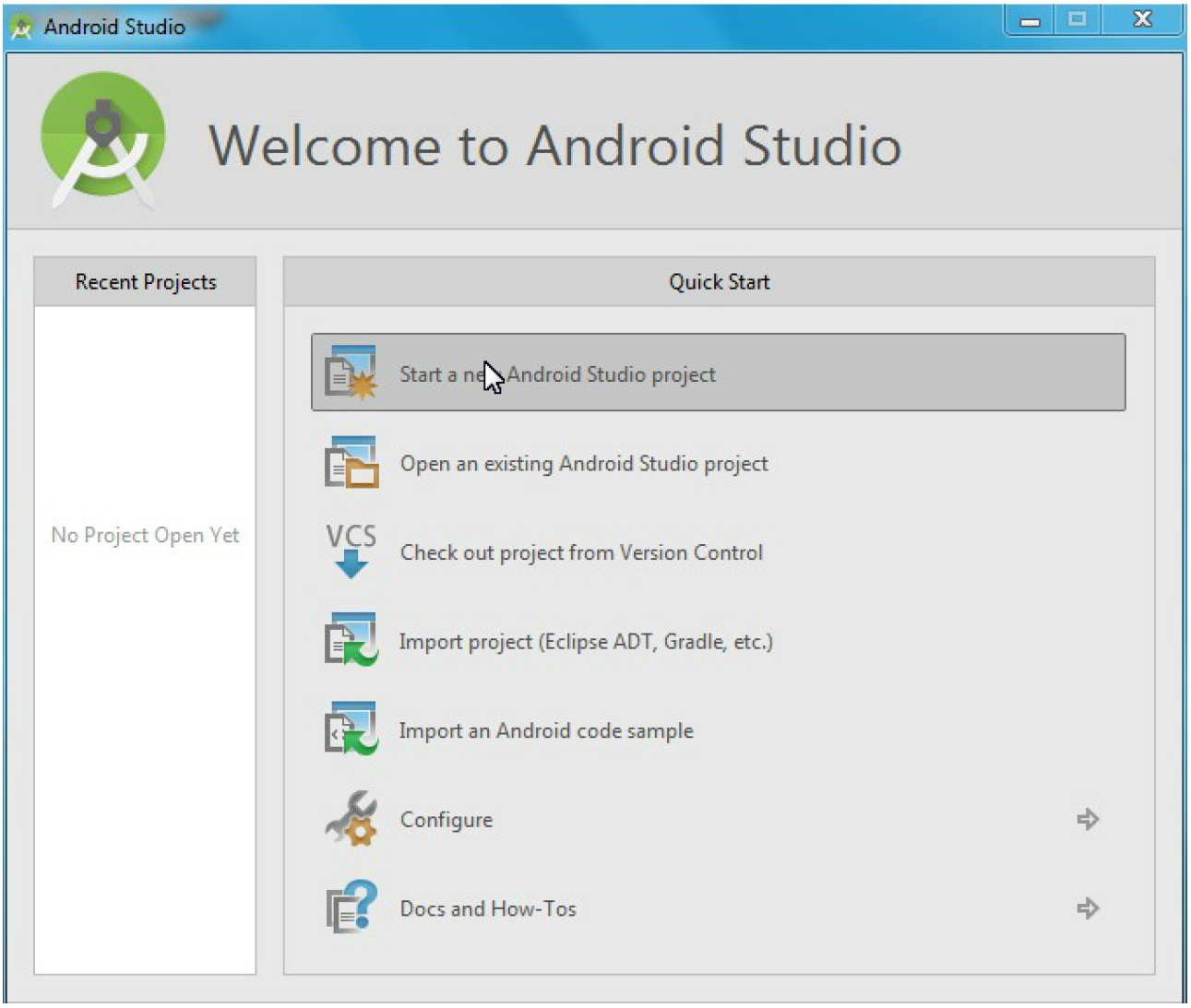
1. Navigate to the directory where the EventRegistration application resides, e.g., */home/user/git/eventregistration*
2. Initialize a new, orphan branch with the following commands

```
git checkout --orphan android
git reset
rm ./* .gitignore
```

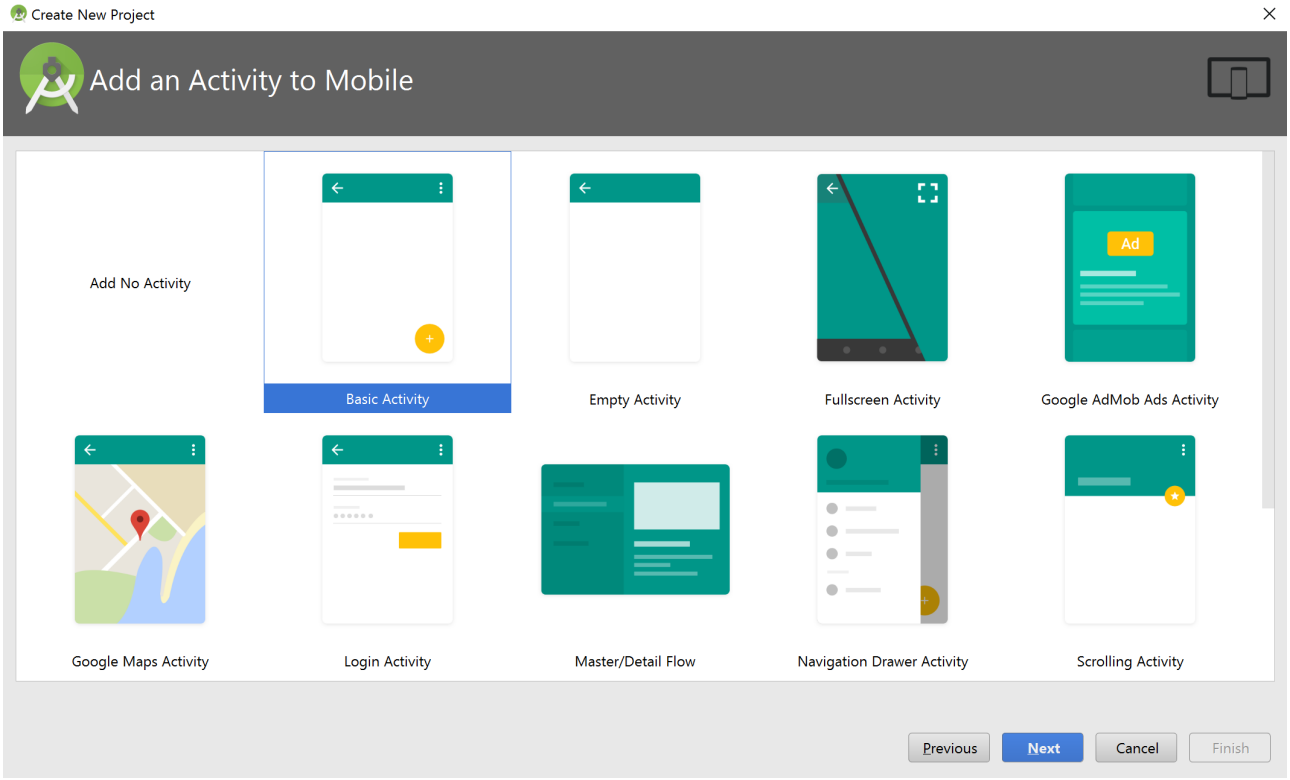
NOTE One-liner command that does the same for those who like living dangerously:
`git checkout --orphan android && git reset && rm ./* .gitignore`

4.2. Create an Android project

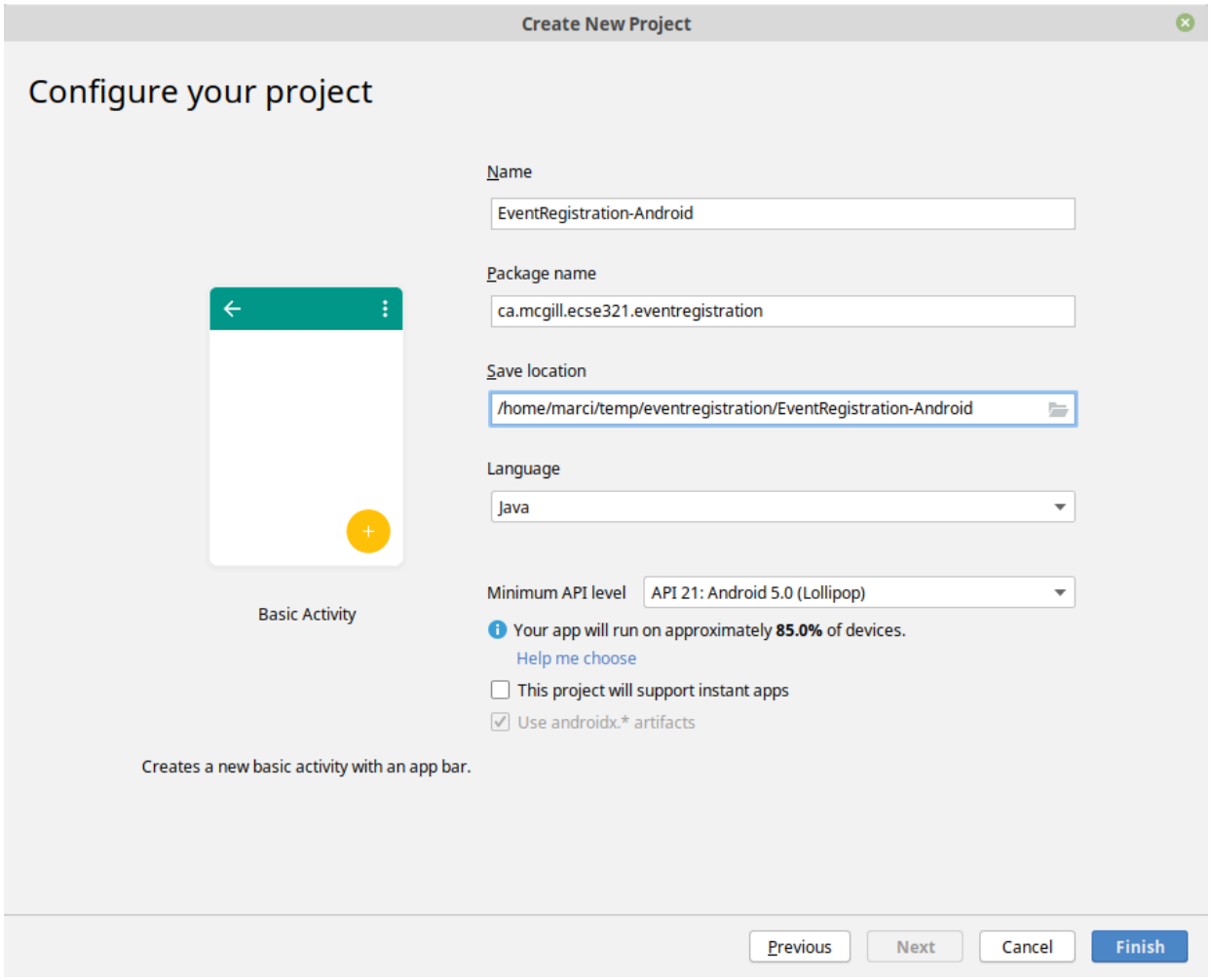
1. Start a new Android project



2. Select a **Basic Activity** and click **Next**



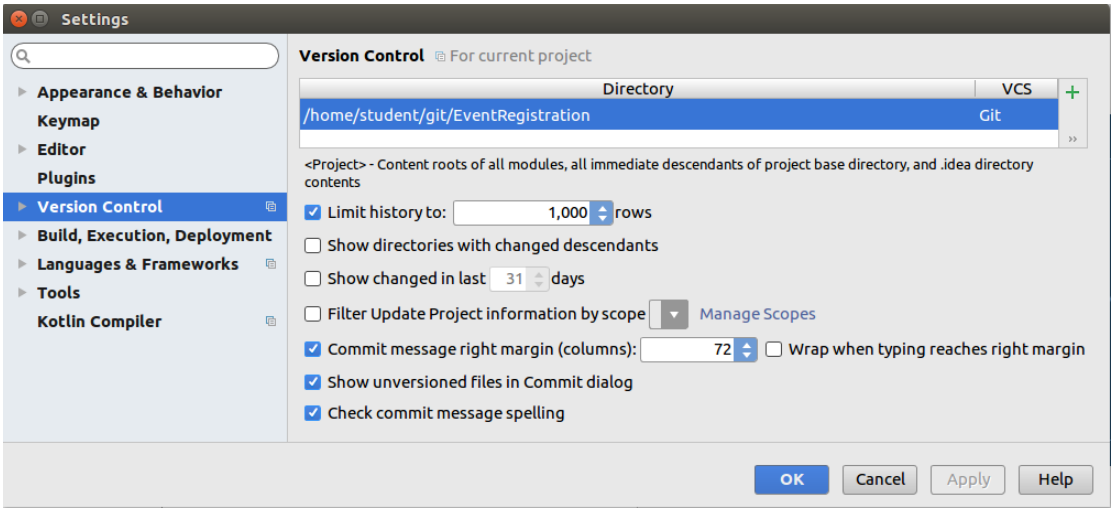
3. Specify project details and click on **Finish**
    - Application name: **EventRegistration-Android**
    - Package name: `ca.mcgill.ecse321.eventregistration`
    - Project location: create the project within the prepared working copy of the git repository say, */home/user/git/eventregistration/EventRegistration-Android`*)
    - Select *Java* as language
    - Click on **Finish**



4. Wait until the project is built by Gradle, this takes a minute or two
5. Optional step.

TIP

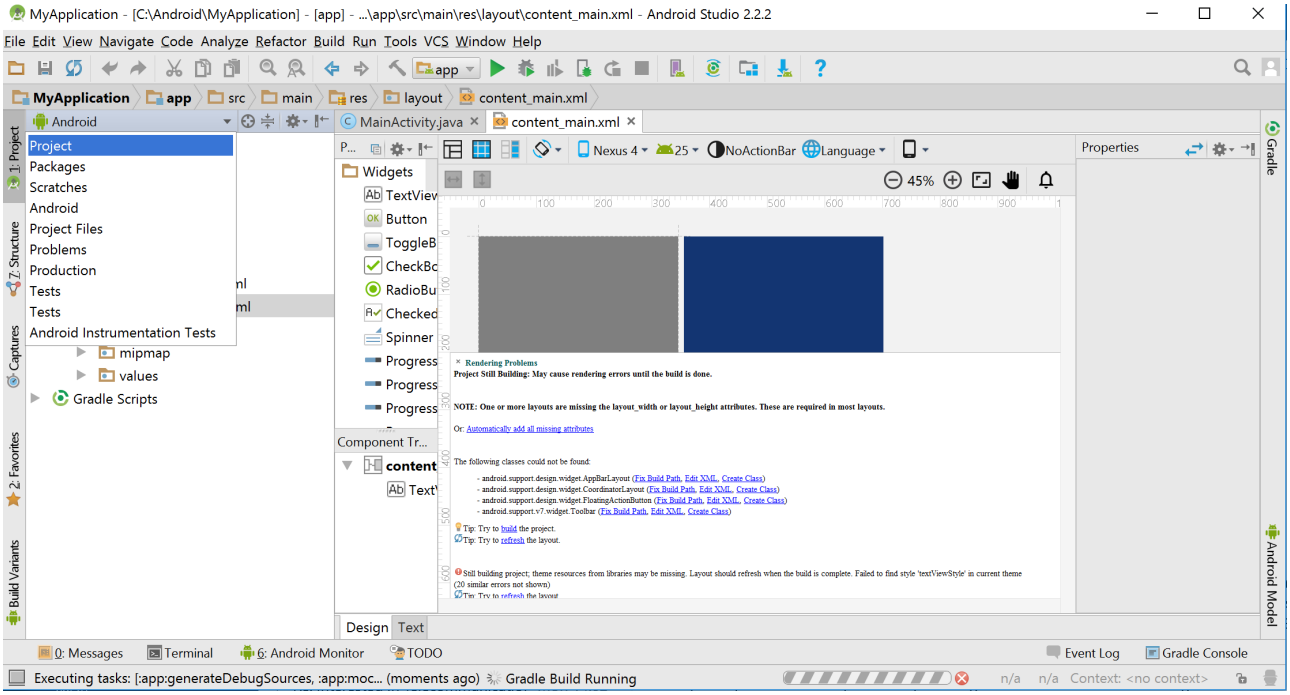
Optionally, to setup version control integration, go to *File/Settings.../Version Control* and add the repository as Git root to the project settings



Then, you can issue Git commands from the *VCS* menu in Android Studio while developing the application. Regardless whether you complete this step or not, you can still use git from the command line, since the project is created in the working directory of your git repository.

6. Select the **Project** view in the left pane (instead of the default **Android** view) and observe three

files:



- **MainActivity.java**: application code is written here (located in **app/src/main/java**)
- **content_main.xml**: layout specifications of the UI are provided in XML (located in *app/src/main/res/layout*)
- **strings.xml**: naming of resources (located in *app/src/main/res/values*)

7. Include a dependency for network communication by adding the following line to the `build.gradle` file located in the *app* folder to the end within the `dependencies{ ⋯ }` part (see figure, but the content is different).

```
    implementation 'com.loopj.android:android-async-http:1.4.9'
```

8. Open the **AndroidManifest.xml** file (located in `app/src/main` within the Android project), and add the following XML tag for setting permissions appropriately (before the existing `<application>` tag)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="ca.mcgill.ecse321.eventregistration">

    <uses-permission android:name="android.permission.INTERNET"/>
    <!-- Existing content with <application> tag -->
</manifest>
```

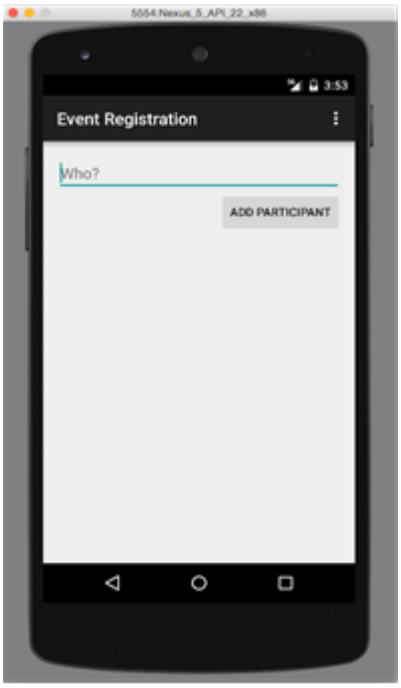
9. As the gradle build file has changed, click on the **Sync** link.
10. Re-build the project by **Build** | **Make Project** if still needed.

4.3. Developing for Android: Part 1

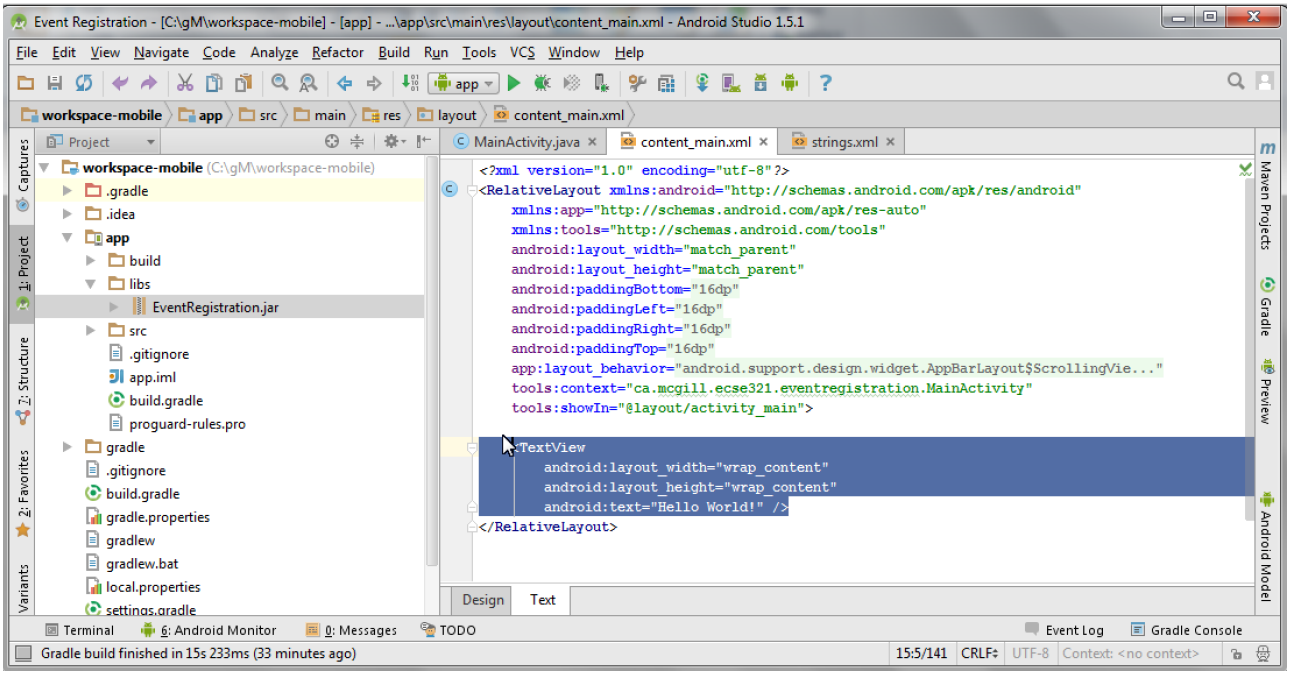
4.3.1. Developing the View Layout

In the next steps, we will develop a simple GUI as the view for the mobile EventRegistration app with (1) one text field for specifying the name of a person, and (2) one **Add Person** button

The GUI will look like as depicted below.



1. Open the **content_main.xml** file, which contains a default **Hello World** text.
2. Replace the highlighted default content with the following XML tags.



```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent">

    <TextView
            android:id="@+id/error"
            android:layout_height="wrap_content"
            android:layout_width="wrap_content"
            android:visibility="gone"
            android:text=""
            android:textColor="@color/colorAccent"/>

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/newperson_name"
        android:hint="@string/newperson_hint"/>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="end"
        android:text="@string/newperson_button"
        android:onClick="addPerson"/>
</LinearLayout>
```

- **LinearLayout** declares a vertical layout to hold the GUI elements;
- **EditText** adds a textfield to enter the name of the person;
- **Button** provides a button to add a person.

Some erroneous tags are marked in red, which will be corrected in the following steps.

Specifying a text field and a button

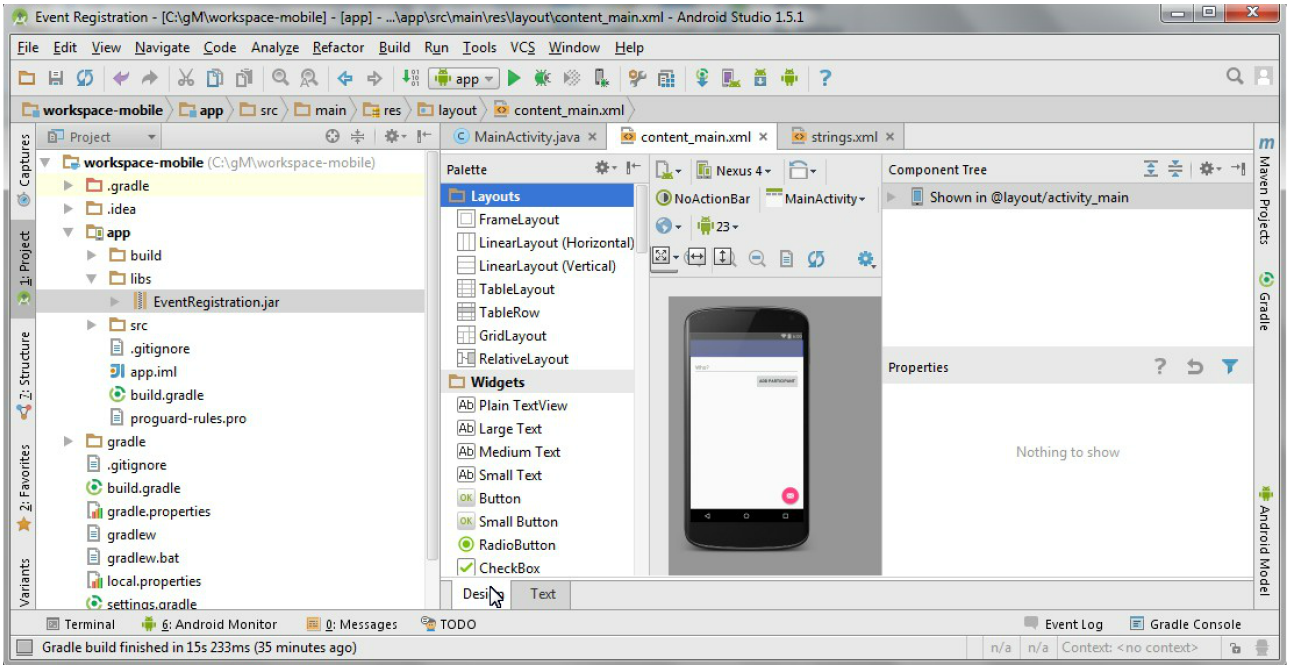
1. We place new literals in the **strings.xml**

```
<string name="newperson_hint">Who?</string>
<string name="newperson_button">Add Person</string>
```

2. Save **strings.xml**

Observing the view

1. Save the file **content_main.xml**.
2. Click on the **Design** tab to check the graphical preview of the app.



4.3.2. Connecting to backend via RESTful service calls

As a next step, we define a view depicted below and add Java code to provide behavior for the view, e.g. what should happen when the different buttons are clicked. The key interactions of our application are the following:

a. What to do when the application is launched? (`onCreate()`)

b. What to do when a button is clicked? (`addPerson()`)

Create a utility class for communicating with HTTP messages

1. Make sure you have the `implementation 'com.loopj.android:android-async-http:1.4.9'` dependency (among others) in the `build.gradle` file for the `app` module (see the section on project setup for more details)
2. Create the `HttpUtils` class in the `ca.mcgill.ecse321.eventregistration` package and add missing imports as required with `Alt+Enter`

> **TIP** You may need to wait a few minutes after dependencies have been resolved to allow the IDE to index classes

```
public class HttpUtils {
    public static final String DEFAULT_BASE_URL = "https://eventregistration-
backend-123.herokuapp.com/";

    private static String baseUrl;
    private static AsyncHttpClient client = new AsyncHttpClient();

    static {
        baseUrl = DEFAULT_BASE_URL;
    }

    public static String getBaseUrl() {
        return baseUrl;
    }

    public static void setBaseUrl(String baseUrl) {
        HttpUtils.baseUrl = baseUrl;
    }

    public static void get(String url, RequestParams params,
AsyncHttpResponseHandler responseHandler) {
        client.get(getAbsoluteUrl(url), params, responseHandler);
    }

    public static void post(String url, RequestParams params,
AsyncHttpResponseHandler responseHandler) {
        client.post(getAbsoluteUrl(url), params, responseHandler);
    }

    public static void getByUrl(String url, RequestParams params,
AsyncHttpResponseHandler responseHandler) {
        client.get(url, params, responseHandler);
    }

    public static void postByUrl(String url, RequestParams params,
AsyncHttpResponseHandler responseHandler) {
        client.post(url, params, responseHandler);
    }

    private static String getAbsoluteUrl(String relativeUrl) {
        return baseUrl + relativeUrl;
    }
}
```

Further helper methods

1. Open the **MainActivity.java** file.
2. Add a new attribute to the beginning of the class for error handling.

```
// ...
public class MainActivity extends AppCompatActivity {
  private String error = null;

  // ...
}
```

3. Implement the `refreshErrorMessage()` method to display the error message on the screen, if there is any.

> **NOTE** Again, add imports with `Alt+Enter` (import is needed for `TextView`)

```
private void refreshErrorMessage() {
  // set the error message
  TextView tvError = (TextView) findViewById(R.id.error);
  tvError.setText(error);

  if (error == null || error.length() == 0) {
    tvError.setVisibility(View.GONE);
  } else {
    tvError.setVisibility(View.VISIBLE);
  }
}
```

4. Add code to initialize the application in the `onCreate()` method (after the auto-generated code).

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  // ...
  // INSERT TO END OF THE METHOD AFTER AUTO-GENERATED CODE
  // initialize error message text view
  refreshErrorMessage();
}
```

Creating a handler for Add Person button

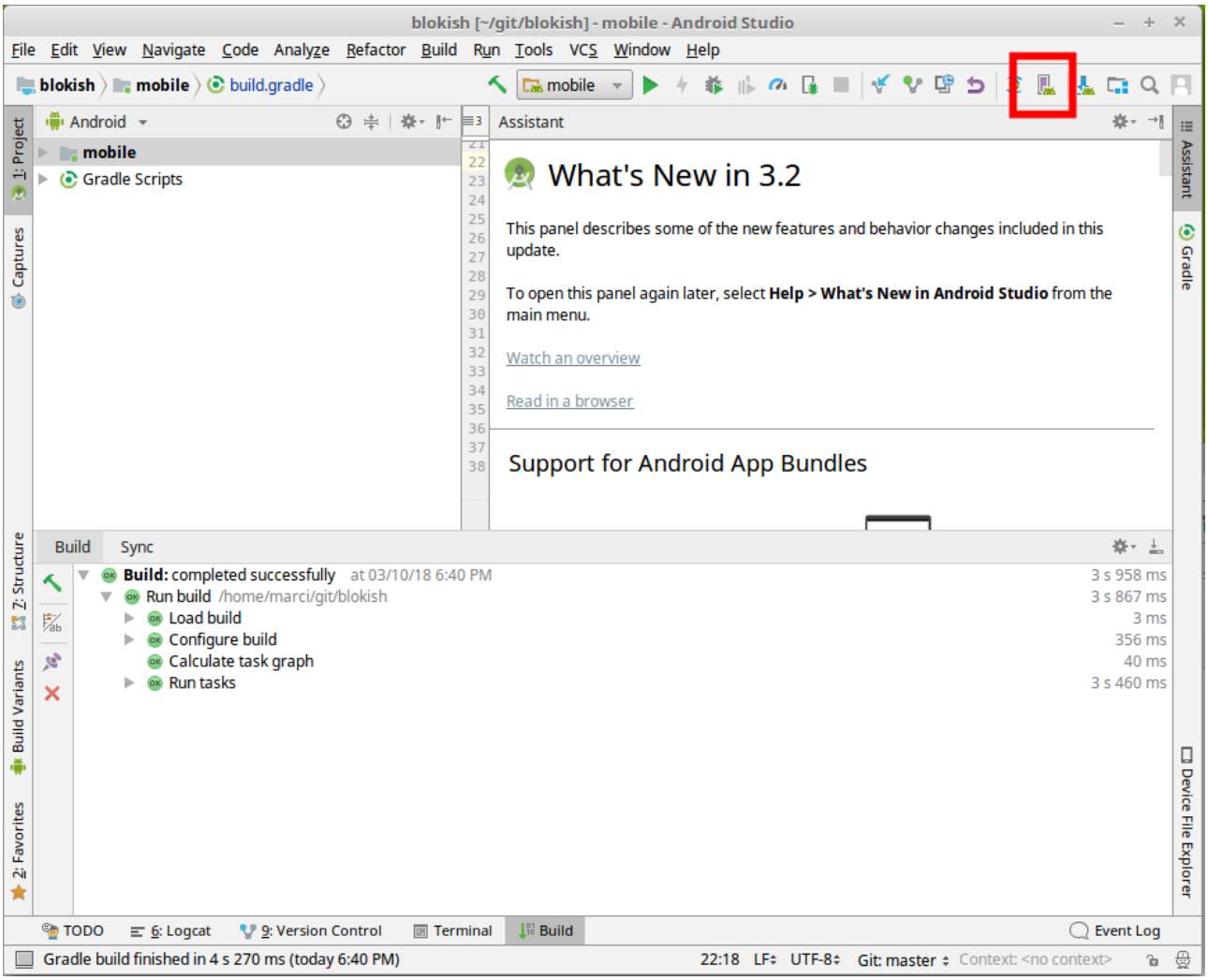
1. Implement the `addPerson()` method as follows

```
public void addPerson(View v) {
  error = "";
  final TextView tv = (TextView) findViewById(R.id.newperson_name);
  HttpUtils.post("persons/" + tv.getText().toString(), new RequestParams(), new
JsonHttpResponseHandler() {
      @Override
      public void onSuccess(int statusCode, Header[] headers, JSONObject response)
{
          refreshErrorMessage();
          tv.setText("");
      }
      @Override
      public void onFailure(int statusCode, Header[] headers, Throwable throwable,
JSONObject errorResponse) {
          try {
              error += errorResponse.get("message").toString();
          } catch (JSONException e) {
              error += e.getMessage();
          }
          refreshErrorMessage();
      }
  });
}
```

2. Import the missing classes again with `Alt+Enter`. There are multiple `Header` classes available, you need to import the `cz.msebera.android.httpclient.Header` class.

4.4. Running and Testing the Application on a Virtual Device

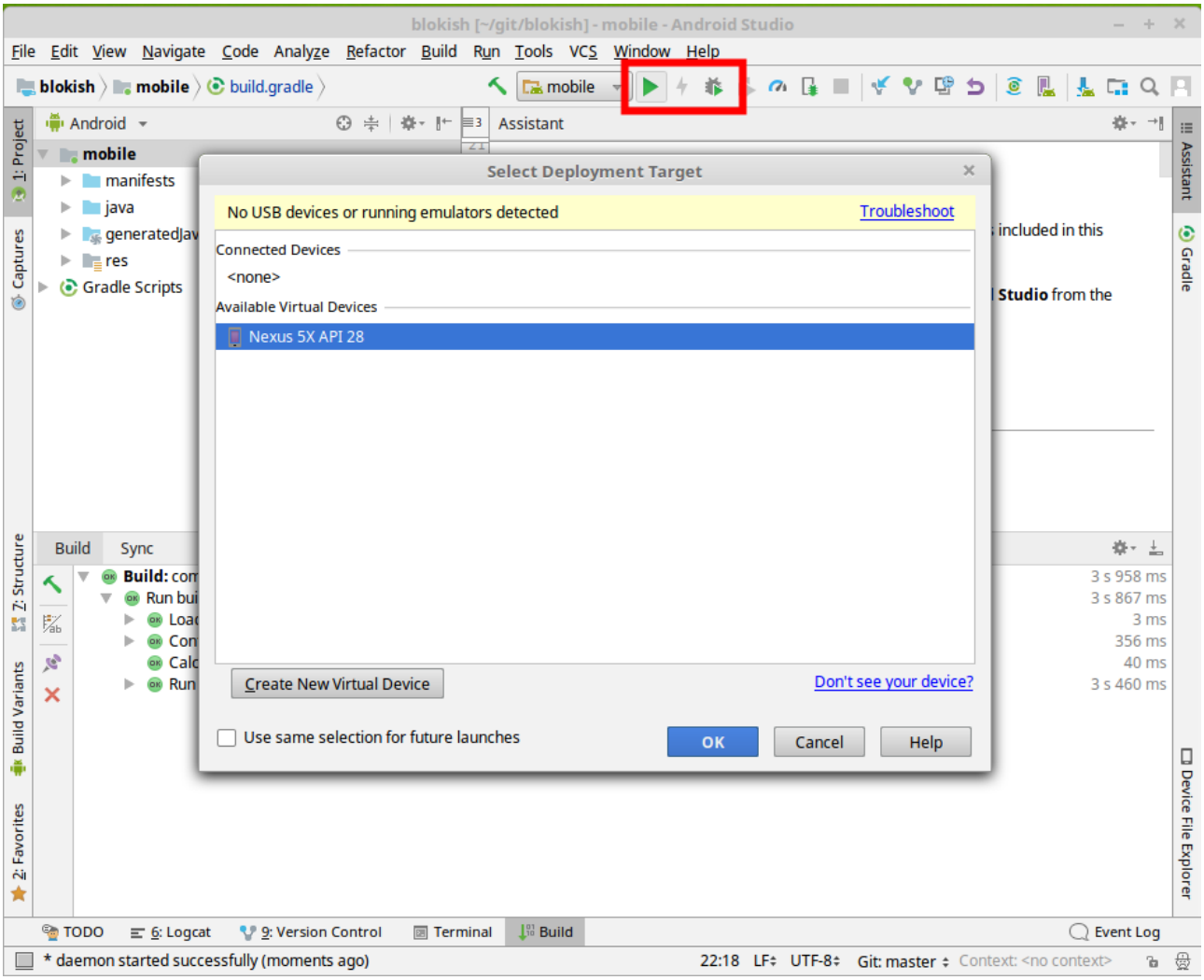
1. Start the Spring backend application on Heroku, while ensure that the `DEFAULT_BASE_URL` in `HttpUtils` is configured accordingly.
2. Click on the *AVD manager* button in Android Studio near the top right corner of the window



3. Add a new device with default settings.

> **NOTE** You might be asked to download emulator files. If this happens, click OK.

4. Start/Debug the application using the buttons highlighted on the figure below



5. After you select the device (in this case Nexus 5X) as deployment target, the application will automatically be deployed and started within an Android VM, and it is ready to use. Be patient, deployment and startup may take a few seconds.
6. Leave the text field empty, then try adding a person. You should get an error message on screen saying a person cannot be added with no name.
7. Supply a name for a new person, then try adding it. Upon successful completion, the text field and the error message should clear.

4.5. Developing for Android (Part 2)

NOTE	You can use the https://eventregistration-backend-123.herokuapp.com backend URL for the event registration example in the `HttpUtils` class.

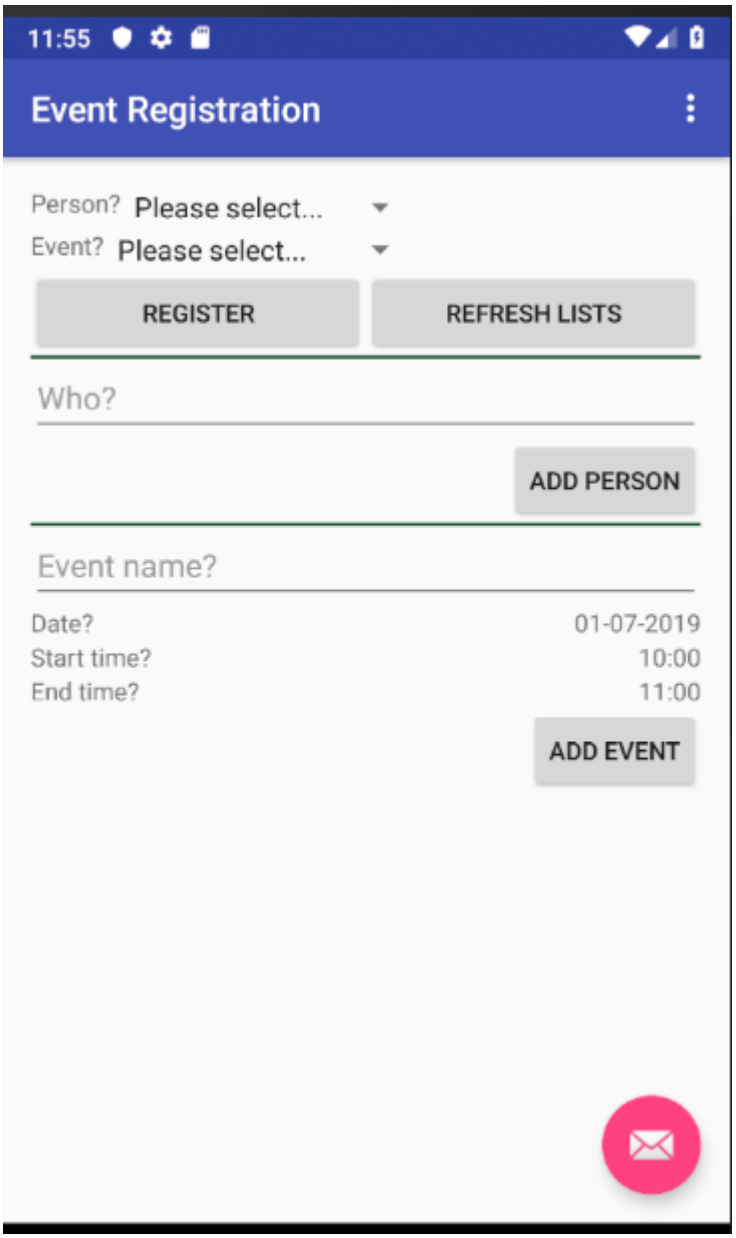
As a next step, we extend the view and its behavior. Key interactions of our application added in this phase are the following:

a. What to do when the application is launched? (`onCreate()`)

b. What to do when application data is updated? (`refreshLists()`)

c. What to do when a button is clicked? (`addEvent()`, and `register()`)

The expected layout of the application:



4.5.1. Create helper classes

1. Create the classes included in the next two steps within the `ca.mcgill.ecse321.eventregistration`

package

2. Create a new class called `DatePickerFragment`

> **NOTE** Import missing classes with `Alt+Enter`. If the import option offers multiple classes, choose the ones that are not deprecated.

```
public class DatePickerFragment extends DialogFragment
        implements DatePickerDialog.OnDateSetListener {

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Use the current date as the default date in the picker
        final Calendar c = Calendar.getInstance();
        int year = c.get(Calendar.YEAR);
        int month = c.get(Calendar.MONTH);
        int day = c.get(Calendar.DAY_OF_MONTH);

        // Parse the existing time from the arguments
        Bundle args = getArguments();
        if (args != null) {
            year = args.getInt("year");
            month = args.getInt("month");
            day = args.getInt("day");
        }

        // Create a new instance of DatePickerDialog and return it
        return new DatePickerDialog(getActivity(), this, year, month, day);
    }

    public void onDateSet(DatePicker view, int year, int month, int day) {
        MainActivity myActivity = (MainActivity)getActivity();
        myActivity.setDate(getArguments().getInt("id"), day, month, year);
    }
}
```

3. Create a new class called `TimePickerFragment`

> **NOTE** The method `setTime` is missing from the `MainActivity` class and will be added later (i.e., the error message which complains that this method is missing is normal)

```
public class TimePickerFragment extends DialogFragment
        implements TimePickerDialog.OnTimeSetListener {

    String label;

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        int hour = 0;
        int minute = 0;

        // Parse the existing time from the arguments
        Bundle args = getArguments();
        if (args != null) {
            hour = args.getInt("hour");
            minute = args.getInt("minute");
        }

        // Create a new instance of TimePickerDialog and return it
        return new TimePickerDialog(getActivity(), this, hour, minute,
                DateFormat.is24HourFormat(getActivity()));
    }

    public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
        MainActivity myActivity = (MainActivity)getActivity();
        myActivity.setTime(getArguments().getInt("id"), hourOfDay, minute);
    }
}
```

4. Add the following helper methods within the `MainActivity` class to support date and time pickers

```
private Bundle getTimeFromLabel(String text) {
    Bundle rtn = new Bundle();
    String comps[] = text.toString().split(":");
    int hour = 12;
    int minute = 0;

    if (comps.length == 2) {
        hour = Integer.parseInt(comps[0]);
        minute = Integer.parseInt(comps[1]);
    }

    rtn.putInt("hour", hour);
    rtn.putInt("minute", minute);

    return rtn;
}

private Bundle getDateFromLabel(String text) {
```

```
    Bundle rtn = new Bundle();
    String comps[] = text.toString().split("-");
    int day = 1;
    int month = 1;
    int year = 1;

    if (comps.length == 3) {
        day = Integer.parseInt(comps[0]);
        month = Integer.parseInt(comps[1]);
        year = Integer.parseInt(comps[2]);
    }

    rtn.putInt("day", day);
    rtn.putInt("month", month-1);
    rtn.putInt("year", year);

    return rtn;
}

public void showTimePickerDialog(View v) {
    TextView tf = (TextView) v;
    Bundle args = getTimeFromLabel(tf.getText().toString());
    args.putInt("id", v.getId());

    TimePickerFragment newFragment = new TimePickerFragment();
    newFragment.setArguments(args);
    newFragment.show(getSupportFragmentManager(), "timePicker");
}

public void showDatePickerDialog(View v) {
    TextView tf = (TextView) v;
    Bundle args = getDateFromLabel(tf.getText().toString());
    args.putInt("id", v.getId());

    DatePickerFragment newFragment = new DatePickerFragment();
    newFragment.setArguments(args);
    newFragment.show(getSupportFragmentManager(), "datePicker");
}

public void setTime(int id, int h, int m) {
    TextView tv = (TextView) findViewById(id);
    tv.setText(String.format("%02d:%02d", h, m));
}

public void setDate(int id, int d, int m, int y) {
    TextView tv = (TextView) findViewById(id);
    tv.setText(String.format("%02d-%02d-%04d", d, m + 1, y));
}
```

4.5.2. Update view definition

1. The corresponding (but partly incomplete) view definition in the `content_main.xml` file is the following:

```
<LinearLayout
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:orientation="vertical">
    <TextView
        android:id="@+id/error"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:visibility="gone"
        android:text=""
        android:textColor="@color/colorAccent"/>

    <LinearLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:orientation="vertical">
        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:orientation="vertical">
            <LinearLayout
                android:orientation="horizontal"
                android:layout_height="wrap_content"
                android:layout_width="match_parent">
                <TextView
                    android:layout_height="wrap_content"
                    android:layout_width="wrap_content"
                    android:text="@string/personspinner_label"/>
                <Spinner
                    android:layout_height="wrap_content"
                    android:layout_width="wrap_content"
                    android:layout_gravity="end"
                    android:id="@+id/personspinner"/>
            </LinearLayout>
            <LinearLayout
                android:orientation="horizontal"
                android:layout_height="wrap_content"
                android:layout_width="match_parent">
                <TextView
                    android:layout_height="wrap_content"
                    android:layout_width="wrap_content"
                    android:text="@string/eventspinner_label"/>
                <Spinner
                    android:id="@+id/eventspinner"
                    android:layout_width="match_parent"
```

```
                android:layout_height="wrap_content"
                android:layout_gravity="end"
                android:layout_margin="0dp"/>
        </LinearLayout>
    </LinearLayout>
    <!-- TODO add a Register and Refresh Lists buttons here -->
</LinearLayout>

<View
    android:layout_height="2dp"
    android:layout_width="fill_parent"
    android:background="#16552e"/>

<LinearLayout
    android:orientation="vertical"
    android:layout_height="wrap_content"
    android:layout_width="match_parent">
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/newperson_name"
        android:hint="@string/newperson_hint"/>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="end"
        android:text="@string/newperson_button"
        android:onClick="addPerson"/>
</LinearLayout>

<View
    android:layout_height="2dp"
    android:layout_width="fill_parent"
    android:background="#16552e"/>

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <EditText android:id="@+id/newevent_name"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent"
        android:hint="@string/newevent_hint"/>
    <LinearLayout
        android:orientation="horizontal"
        android:layout_height="wrap_content"
        android:layout_width="match_parent">
        <TextView
            android:layout_height="wrap_content"
            android:layout_width="0dp"
            android:layout_weight="1"
```

```
                android:text="@string/newevent_date_label"/>
            <TextView
                android:layout_height="wrap_content"
                android:layout_width="wrap_content"
                android:text="@string/newevent_date_first"
                android:layout_gravity="end"
                android:id="@+id/newevent_date"
                android:onClick="showDatePickerDialog"/>
        </LinearLayout>
        <LinearLayout
            android:orientation="horizontal"
            android:layout_height="wrap_content"
            android:layout_width="match_parent">
            <TextView
                android:layout_height="wrap_content"
                android:layout_width="0dp"
                android:layout_weight="1"
                android:text="@string/starttime_label"/>
            <TextView
                android:layout_height="wrap_content"
                android:layout_width="wrap_content"
                android:text="@string/starttime_first"
                android:layout_gravity="end"
                android:id="@+id/starttime"
                android:onClick="showTimePickerDialog"/>
        </LinearLayout>
        <!-- TODO add a label and a time picker for event end time -->
        <!-- TODO add Add Event button here -->
    </LinearLayout>
</LinearLayout>
```

2. The missing string definitions go in the `res/values/strings.xml` resource

```
<resources>
    <string name="app_name">EventRegistration-Android</string>
    <string name="action_settings">Settings</string>
    <string name="newperson_hint">Who?</string>
    <string name="newperson_button">Add Person</string>
    <string name="newevent_date_label">Date?</string>
    <string name="personspinner_label">Person?</string>
    <string name="eventspinner_label">Event?</string>
    <string name="starttime_label">Start time?</string>
    <string name="newevent_date_first">01-07-2019</string>
    <string name="starttime_first">10:00</string>
    <string name="newevent_hint">Event Name?</string>
</resources>
```

   - **TODO**: add a *Register* button to allow registering a selected person to a selected event (call the `register()` method when clicked - this is to be implemented in the upcoming steps)

- **TODO**: add a *Refresh Lists* button that refreshes the contents of the event and person spinners (call the `refreshLists()` method when clicked)
- **TODO**: add a label with text *End?* below the *Start?* label
- **TODO**: add a time picker to select the end time of a new event
- **TODO**: add an *Add Event* button to allow creating new events from the user interface (call the `addEvent()` method when clicked - this is to be implemented in the upcoming steps)

4.5.3. Initialization on application launch

1. Open the **MainActivity.java** file.
2. Add a few new attributes to the beginning of the class as helpers for persistence and error handling.

```
public class MainActivity extends AppCompatActivity {
  private String error = null;
  // APPEND NEW CONTENT STARTING FROM HERE
  private List<String> personNames = new ArrayList<>();
  private ArrayAdapter<String> personAdapter;
  private List<String> eventNames = new ArrayList<>();
  private ArrayAdapter<String> eventAdapter;

  //...
}
```

   - Import missing classes (e.g. use `Alt+Enter`)
3. Add code to initialize the application with data from the server in the `onCreate()` method (after the auto-generated code).

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  // ...
  // INSERT TO END OF THE METHOD
  // Add adapters to spinner lists and refresh spinner content
  Spinner personSpinner = (Spinner) findViewById(R.id.personspinner);
  Spinner eventSpinner = (Spinner) findViewById(R.id.eventspinner);

  personAdapter = new ArrayAdapter<String>(this,
android.R.layout.simple_spinner_item, personNames);

personAdapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item
);
  personSpinner.setAdapter(personAdapter);

  eventAdapter = new ArrayAdapter<String>(this,
android.R.layout.simple_spinner_item, eventNames);

eventAdapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
;
  eventSpinner.setAdapter(eventAdapter);

  // Get initial content for spinners
  refreshLists(this.getCurrentFocus());
}
```

At this point the `refreshLists()` method is missing, this is to be implemented in the upcoming steps.

4.5.4. Reactions to updated data

1. Create the missing new method `refreshLists()` which seeks for the event and person spinners and sets their content according to the data retrieved from the server

```
public void refreshLists(View view) {
    refreshList(personAdapter ,personNames, "people");
    refreshList(eventAdapter, eventNames, "events");
}

private void refreshList(final ArrayAdapter<String> adapter, final List<String>
names, final String restFunctionName) {
    HttpUtils.get(restFunctionName, new RequestParams(), new
JsonHttpResponseHandler() {

        @Override
        public void onSuccess(int statusCode, Header[] headers, JSONArray response)
{
            names.clear();
            names.add("Please select...");
            for( int i = 0; i < response.length(); i++){
                try {
                    names.add(response.getJSONObject(i).getString("name"));
                } catch (Exception e) {
                    error += e.getMessage();
                }
                refreshErrorMessage();
            }
            adapter.notifyDataSetChanged();
        }

        @Override
        public void onFailure(int statusCode, Header[] headers, Throwable
throwable, JSONObject errorResponse) {
            try {
                error += errorResponse.get("message").toString();
            } catch (JSONException e) {
                error += e.getMessage();
            }
            refreshErrorMessage();
        }
    });
}
```

2. Implement the `addEvent()` method

```
public void addEvent(View v) {
    // start time
    TextView tv = (TextView) findViewById(R.id.starttime);
    String text = tv.getText().toString();
    String comps[] = text.split(":");

    int startHours = Integer.parseInt(comps[0]);
    int startMinutes = Integer.parseInt(comps[1]);
```

```
    // TODO get end time

    // date
    tv = (TextView) findViewById(R.id.newevent_date);
    text = tv.getText().toString();
    comps = text.split("-");

    int year = Integer.parseInt(comps[2]);
    int month = Integer.parseInt(comps[1]);
    int day = Integer.parseInt(comps[0]);

    // name
    tv = (TextView) findViewById(R.id.newevent_name);
    String name = tv.getText().toString();

    // Reminder: calling the service looks like this:
    // https://eventregistration-backend-
123.herokuapp.com/events/testEvent?date=2013-10-23&startTime=00:00&endTime=23:59

    RequestParams rp = new RequestParams();

    NumberFormat formatter = new DecimalFormat("00");
    rp.add("date", year + "-" + formatter.format(month) + "-" +
formatter.format(day));
    rp.add("startTime", formatter.format(startHours) + ":" +
formatter.format(startMinutes));
    // TODO add end time as parameter

    HttpUtils.post("events/" + name, rp, new JsonHttpResponseHandler() {
        @Override
        public void onSuccess(int statusCode, Header[] headers, JSONObject
response) {
            refreshErrorMessage();
            ((TextView) findViewById(R.id.newevent_name)).setText("");
        }

        @Override
        public void onFailure(int statusCode, Header[] headers, Throwable
throwable, JSONObject errorResponse) {
            try {
                error += errorResponse.get("message").toString();
            } catch (JSONException e) {
                error += e.getMessage();
            }
            refreshErrorMessage();
        }
    });
}
```

- **TODO**: get the end time of the new event
- **TODO**: supply the end time to the REST request as an additional parameter

3. Implement the `register()` method

```
public void register(View v) {

    Spinner partSpinner = (Spinner) findViewById(R.id.personspinner);
    Spinner eventSpinner = (Spinner) findViewById(R.id.eventspinner);

    error = "";

    // TODO issue an HTTP POST here
    // Reminder: calling the service looks like this:
    // https://eventregistration-backend-
123.herokuapp.com/register?person=testPreson&event=testEvent


    // Set back the spinners to the initial state after posting the request
    partSpinner.setSelection(0);
    eventSpinner.setSelection(0);

    refreshErrorMessage();
}
```

- **TODO**: implement the HTTP POST part of the `register()` method on your own