Lab: Restful services

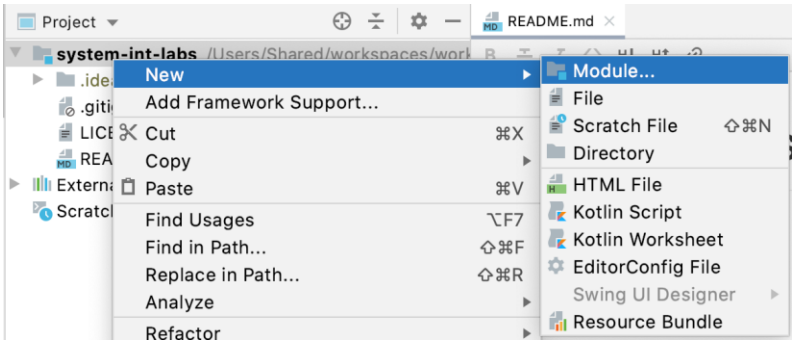
The aim of this lab is to allow you to create a RESTful service that will supply book information.

Step 1: Create a new Project and Module

You may wish to continue to use the Project you created for the previous course; however if you wish create a new IntelliJ project.

Whether you use a new project or not; create a new Module as we will configure this module to allow us to run our RESTful service with an embedded Tomcat web application server.

To create a new Module use the right mouse menu New>Module… option:



Call the module something like bookshop-rest or bookshop-service.

Make sure you select the Maven option and the correct version of Java (at least 1.8).

Step 2: Update the POM file

The default POM file will now need to be updated to include the necessary libraries.

In this case we need something that will run Tomcat and use it to handle the HTTP requests that need to be routed to our RESTful service.

There are several ways in which we can do this but the simplest is to use Spring Boot. We therefore need to add the dependencies to allow us to run Spring boot, with Tomcat and implement a REST interface.

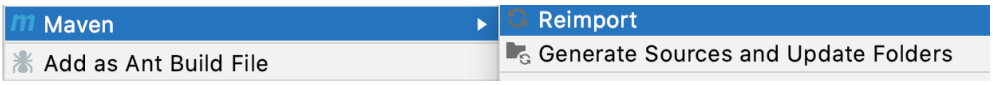
Therefore update your POM as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.0</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <groupId>com.jjh</groupId>
  <artifactId>bookshop</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <java.version>8</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-jersey</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>


</project>
```

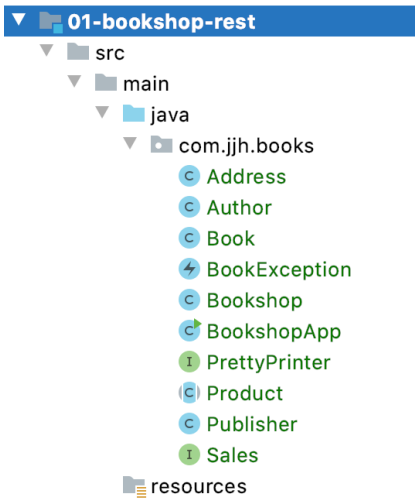
Note that you should adjust the java.version property as required by your project. I am using 13 but you may only have 1.8 installed on your machines etc.

Next you may need to tell IntelliJ to reimport the POM file and update the libraries used by the module; to do this select the pom.xml file in the Project view and from the right mouse menu select Maven>Reimport:



Step 3: Copy the Book classes

Copy the book classes that we created in the last course over into this new module, for example:



Now make sure every class has a zero parameter constructor – this is required by the JAXB Java API for Binding infrastructure that will convert your objects into JSON and back again.

Step 4: Create a new package

We will now create a different package for the service related code to go in. For example, create a package with service in the name, e.g. com.jjh.service.

Step 5: Add the Service Controller

Add a class that will define the RESTful service controller for the bookshop. I have called this class BookshopController which follows the convention of adding Controller to the end of the class name.

An initial structure for this class is given below. Note that the API of the Bookshop class you created last week might be slightly different from mine depending on exactly how you implemented it. You should adjust your code to match. One additional method which we did not implement last week was the ability to select a class by ISBN – you will need to implement this yourself in your Bookshop class.

```
package com.jjh.service;

import com.jjh.books.Book;
import com.jjh.books.BookException;
import com.jjh.books.Bookshop;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;
```

```
import java.util.List;

@RestController
@RequestMapping("bookshop")
public class BookshopController {

   private Bookshop bookshop = new Bookshop();

   @GetMapping("{title}")
   public Book getBook(@PathVariable String title) {
      System.out.println("BookshopController.getBook(" + title + ")");
      return this.bookshop.getBookByTitle(title);
   }

   @GetMapping("list")
   public List<Book> getAllBooks() {
      System.out.println("BookshopController.getAllBooks()");
      return bookshop.getBooks();
   }
}
```

Step 6: Add the main application class

Finally we need to add the main application class that will start up Tomcat and run our service for us. Call this class BookshopService.

This is essentially boiler plate code – you should only need to change any print messages.

Place it in the same package as the controller class.

```
package com.jjh.service;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class BookshopService {
   public static void main(String[] args) {
      System.out.println("Starting Bookshop Service");
      SpringApplication.run(BookshopService.class, args);
      System.out.println("Setup finished");
   }
}
```

Step 7: Run the Application

Select the class containing your main method and from the right mouse menu select to run this application.

You should see output in your console indicating that Spring Boot is starting up and that your application is starting.

If you look at the Spring Boot output; you should also see that Tomcat has been started as an embedded Java Web (HTTP) Application Server.

Step 8: Access the RESTful service

Using a Web Browser such as Chrome enter the URL that will access your application.

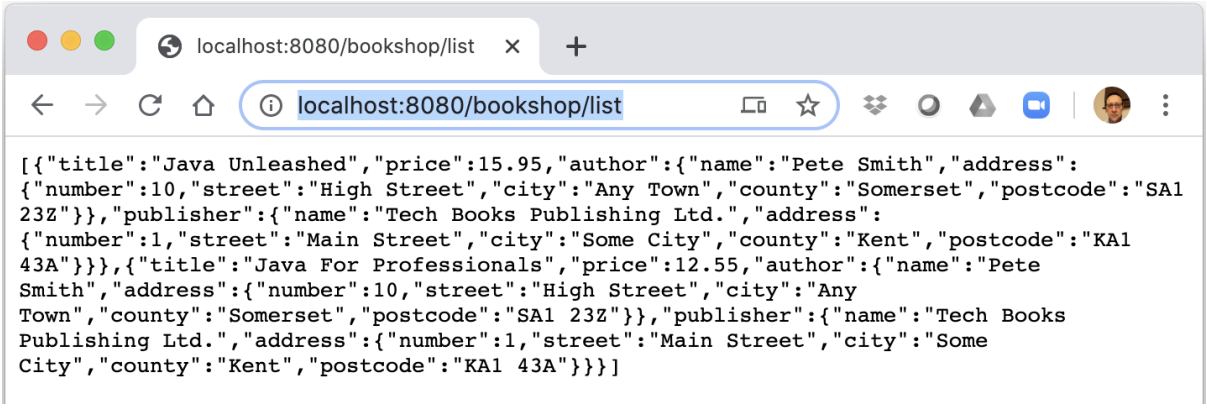
You can determine this based on the information provided in the Controller class.

In my case this was “bookshop” at the class level and “list” to list all books and “{title}” for an individual book.

As the application is running locally (on the localhost) via port 8080 (the default for tomcat) I need to enter:

http://localhost:8080/bookshop/list

In the browser. Doing this I can see the result is a set of JSON objects describing the objects held in the bookshop:

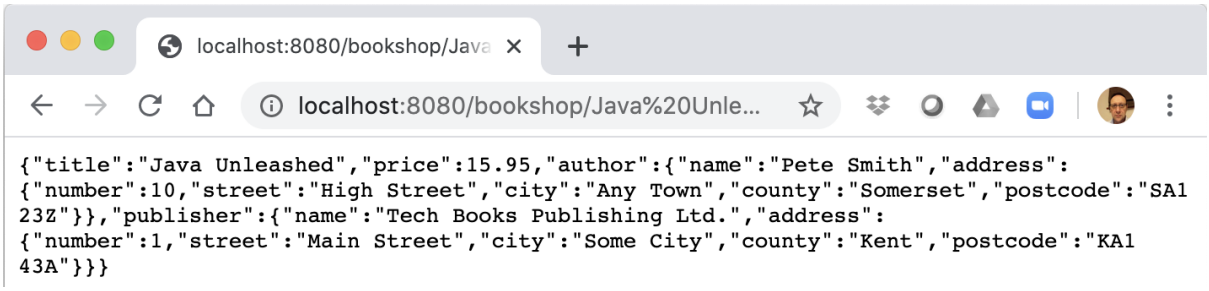


If I select one Book to Return I need to provide the title of the book:

Note title have spaces in so we will need to replace the space with %20 code to represent the space:

http://localhost:8080/bookshop/Java%20Unleashed

The result of this is:



Extension Points

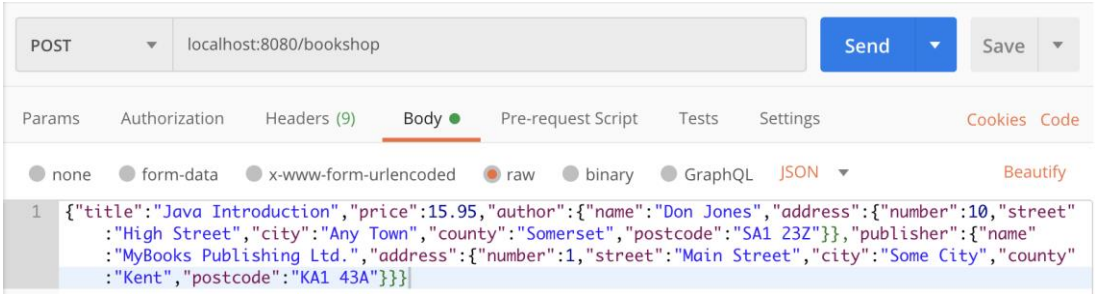
1. Add a method to allow a Book to be added to the bookshop
2. Use Postman to test this functionality
3. Add a method to update a Book in the Bookshop
4. Again use Postman to test this functionality
5. Add a method to remove a Book from the bookshop
6. Use Postman to test this functionality

Notes

For adding a book; you will need to make sure your JSON is correct; for example in my case the JSON to add a new book might be:

{"title":"Java Introduction","price":15.95,"author":{"name":"Don Jones","address":{"number":10,"street":"High Street","city":"Any Town","county":"Somerset","postcode":"SA1 23Z"}},"publisher":{"name":"MyBooks Publishing Ltd.","address":{"number":1,"street":"Main Street","city":"Some City","county":"Kent","postcode":"KA1 43A"}}}

This is used in the body of the Postman tool:



Lab: Microservice Registration and Discovery

The aim of this lab is to use Spring and the Eureka service registry to register two services and allow once service to use another.

Step 1: Set up 3 new modules

To do this we will need to create three separate modules.

One module will be for the registry; one module for the shop service and one for the existing books service (so that changes we make to this service to allow it to run within a service registry environment are separate from the previous version of the service).

The three modules can be called:

- microservice-shop
- microservice-books
- microservice-register

Each one should have a POM file with the following configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.0</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <groupId>com.jjh</groupId>
  <artifactId>02-registry-service</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <!-- Fixes a compatibility issue with Spring Cloud and Spring Boot 2.4.0 -->
    <spring-cloud.version>2020.0.0-M6</spring-cloud.version>

    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
    <maven.compiler.target>11</maven.compiler.target>
    <maven.compiler.source>11</maven.compiler.source>
</properties>

<dependencies>

    <dependency>
        <!-- Spring Cloud starter -->
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
        <exclusions>
            <exclusion>
                <groupId>org.junit.vintage</groupId>
                <artifactId>junit-vintage-engine</artifactId>
            </exclusion>
        </exclusions>
    </dependency>

    <dependency>
        <!-- Eureka service registration -->
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
    </dependency>

</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

```
  <build>
    <resources>
      <resource>
        <!-- Make sure resources is on the classpath -->
        <directory>src/main/resources</directory>
      </resource>
    </resources>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

  <repositories>
    <repository>
      <id>spring-milestones</id>
      <name>Spring Milestones</name>
      <url>https://repo.spring.io/milestone</url>
    </repository>
  </repositories>

</project>
```

Step 2: Configure the Service Registry

The service registry module will contain two files; the ServiceRegistrationServer.java file and the registration-server.yml file.

The ServiceRegistrationServer class is defined as shown below:

```
package com.jjh.register;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.actuate.trace.http.HttpTraceRepository;
import org.springframework.boot.actuate.trace.http.InMemoryHttpTraceRepository;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistrationServer {

  public static void main(String[] args) {
    // Should be run first
    System.out.println("Starting Registration Server");
    // Tell Boot to look for registration-server.yml
    System.setProperty("spring.config.name", "registration-server");
```

```
        SpringApplication.run(ServiceRegistrationServer.class, args);
        System.out.println("Registration Server Started");
    }

    @ConditionalOnMissingBean
    @Bean
    public HttpTraceRepository httpTraceRepository() {
        return new InMemoryHttpTraceRepository();
    }
}
```

The registration-server.yml file is defined as shown beloiw:

```
# Configure this Discovery Server
eureka:
 instance:
   hostname: localhost
 client:  # Not a client, don't register with yourself
   registerWithEureka: false
   fetchRegistry: false
 server:
  renewalPercentThreshold=0.85


server:
 port: 1111   # HTTP (Tomcat) port
```

In Intellij these are located as shown below:

Step 3: Create the shop service

The shop service will be a service that uses the book service you previously created.

In the microservice-shop module we will create three files; the ShopService.java, ShopController.java and the shop-service.yml files.

The contents of these files is given below:

ShopService.java

```
package com.jjh.shop;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;
```

```
@SpringBootApplication
@EnableDiscoveryClient
public class ShopService {

  public static void main(String[] args) {
    System.out.println("Starting the Shop Service");

    // Will configure using shop-service.yml
    System.setProperty("spring.config.name", "shop-service");

    SpringApplication.run(ShopService.class, args);
    System.out.println("Shop Service started");
  }

  @LoadBalanced    // Make sure to create the load-balanced template
   @Bean
   RestTemplate restTemplate() {
      return new RestTemplate();
   }

}
```

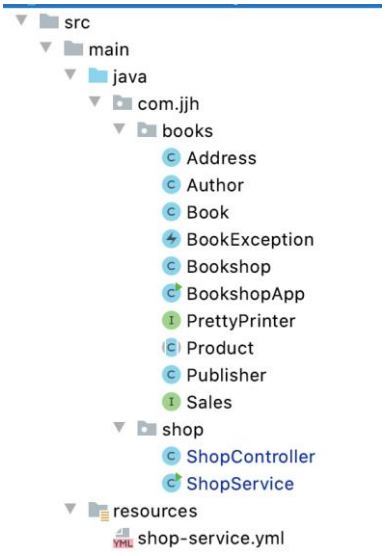
The ShopController.java is defined as:

```
package com.jjh.shop;

import com.jjh.books.Book;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@RestController
public class ShopController {

  private static final String USER_SERVICE_URL = "http://book-service";

  @Autowired
  @LoadBalanced
  private RestTemplate restTemplate;


  @GetMapping
  public Map<String, List<Book>> getBooks() {
    System.out.println("ShopService.getBooks()");
    Map<String, List<Book>> map = new HashMap<String, List<Book>>();
    List<Book> results = (List<Book>) restTemplate.getForObject(USER_SERVICE_URL + "/bookshop/list",
List.class);
    map.put("Technical", results);
    return map;
  }

}
```

Finally, the shop-service.yml file is defined as:

```
# Spring properties
spring:
 application:
   name: shop-service

# HTTP Server
server:
 port: 3333   # HTTP (Tomcat) port

# Discovery Server Access
#  1. DEV ONLY: Reduce the lease renewal interval to speed up registration
#  2. Define URL of registration server (defaultZone)
eureka:
 client:
  serviceUrl:
   defaultZone: http://localhost:1111/eureka/
 instance:
  leaseRenewalIntervalInSeconds: 5   # DO NOT DO THIS IN PRODUCTION
```

In addition as this service will now need to work with the books returned by the books service; you will need to copy the books domain classes into this module as well. The end result is illustrated below:



Step 4: Update the Books service

We need to make two changes to the books service we created in the last lab; these are:

1. Add the book-service.yml file to the resources directory
2. Update the main class so that it enables registrations server discovery and knows what the configuration fiel is called

The book-service.yml file is given below:

```
# Spring properties
spring:
 application:
   name: book-service

# HTTP Server
server:
```

```
  port: 2222   # HTTP (Tomcat) port

# Discovery Server Access
#  1. DEV ONLY: Reduce the lease renewal interval to speed up registration
#  2. Define URL of registration server (defaultZone)
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/
  instance:
    leaseRenewalIntervalInSeconds: 5   # DO NOT DO THIS IN PRODUCTION
```

The revised BookshopService class with the main method is given blow:

```
package com.jjh.service;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class BookshopService {
    public static void main(String[] args) {
        System.out.println("Starting Bookshop Service");

        // Tell server to look for book-server.properties or
        // book-server.yml
        System.setProperty("spring.config.name", "book-service");

        SpringApplication.run(BookshopService.class, args);
        System.out.println("Setup finished");
    }
}
```

Step 5: Run the Services

You **must** first run the Service Registry (otherwise the other services will fail on start up as they won't find the service registry to register with).

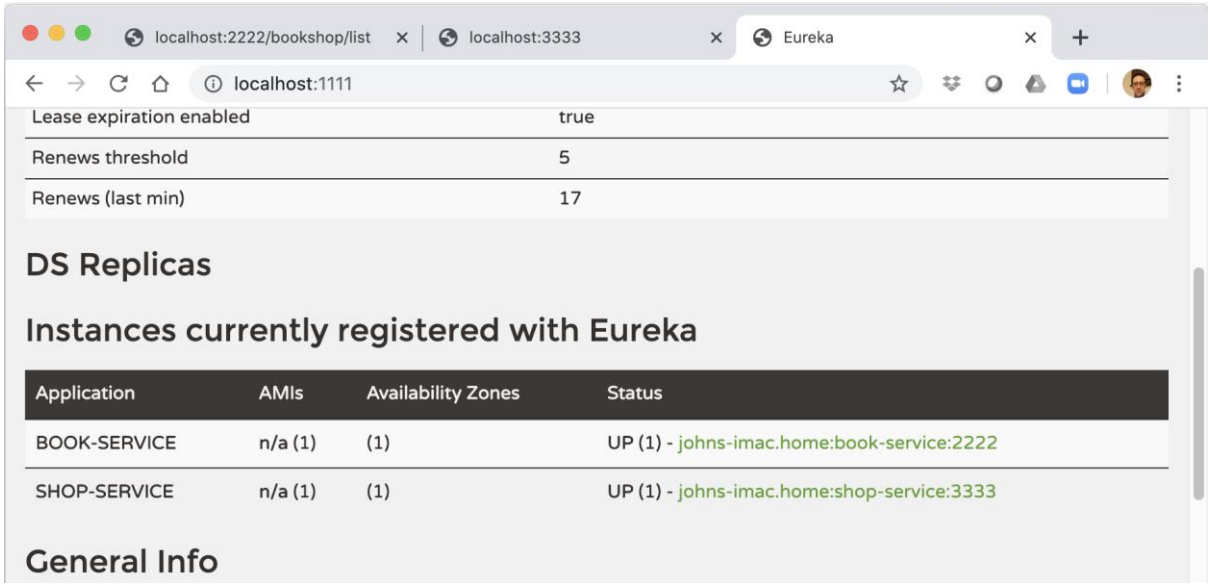
Next run both the microservices.

Note that each time you run a separate module in IntelliJ it starts a new process with its output console.

Step 6: Check the Eureka Console

In Chrome enter the URL http://localhost:1111/

This will display the Eureka console

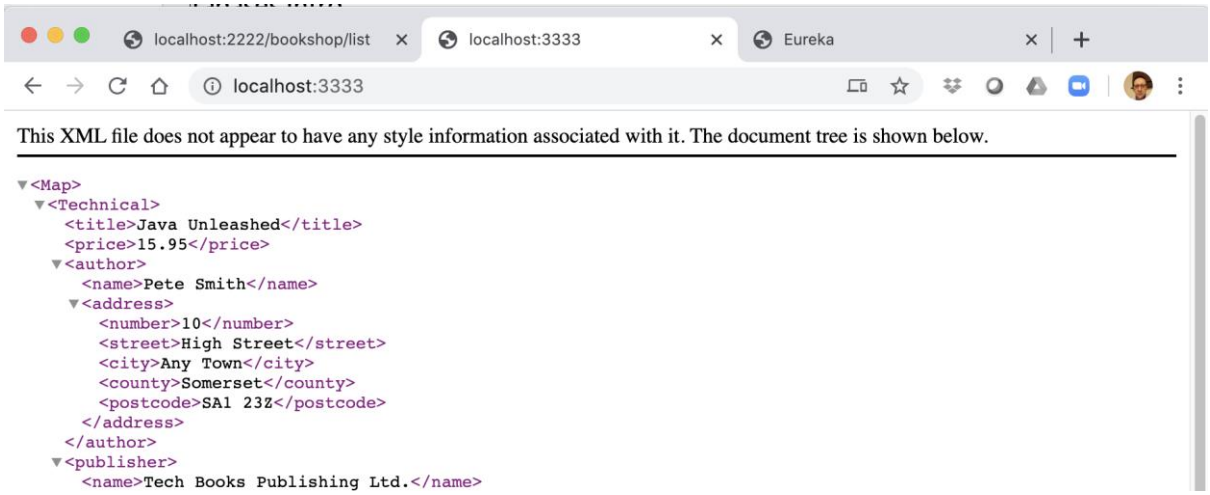


Check the information presented.

Step 7: Access the Shop Service

In Chrome enter the URL http://localhost:3333/

This will run the Shop service – you should see the results returned to you in XML format (the default used by Eureka):

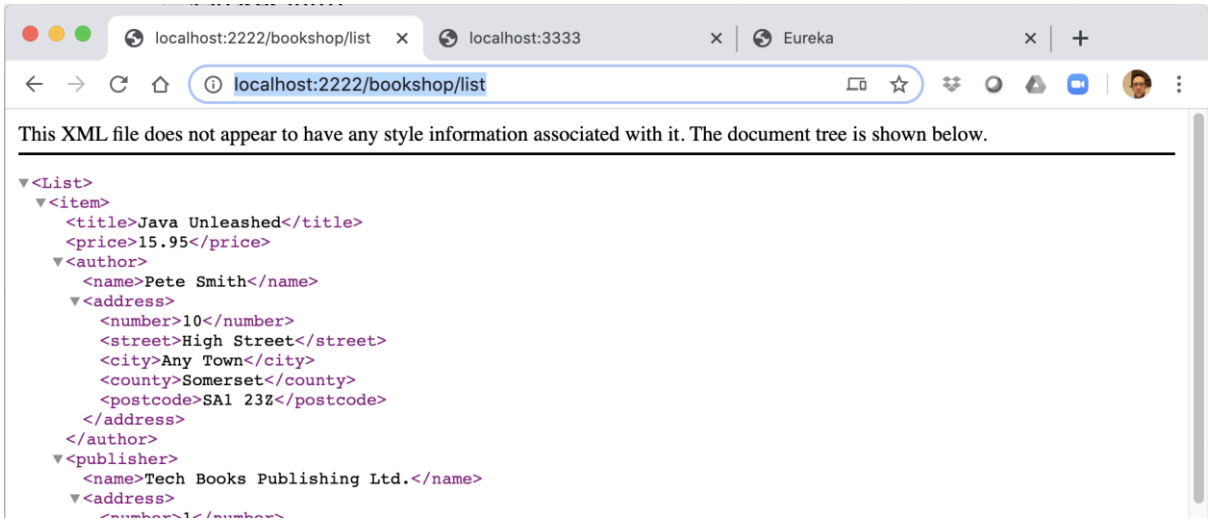


Step 8: Access the books service directly

You can still access the books service directly if you wish.

Use the URL http://localhost:2222/bookshop/list

Now you should see


localhost:2222/bookshop/list
localhost:3333
Eureka
localhost:2222/bookshop/list
This XML file does not appear to have any style information associated with it. The document tree is shown below.
<List>
<item>
<title>Java Unleashed</title>
<price>15.95</price>
<author>
<name>Pete Smith</name>
<address>
<number>10</number>
<street>High Street</street>
<city>Any Town</city>
<county>Somerset</county>
<postcode>SA1 23Z</postcode>
</address>
</author>
<publisher>
<name>Tech Books Publishing Ltd.</name>
<address>

Lab: GraphQL Service

The aim of this lab is to use the GraphQL technology to allow a user to select what information they will return from a simplified book service.

Step 1: Create a new Module

We will create a new module within IntelliJ for us to work in. This should again be a Maven managed module. The content of the POM file should be:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.jjh</groupId>
  <artifactId>graphql-services</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>graphql-services</name>
  <url>http://maven.apache.org</url>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.0</version>
    <relativePath />
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.target>8</maven.compiler.target>
    <maven.compiler.source>8</maven.compiler.source>
    <!-- Properties used for the Graph QL libraries -->
    <graphiql.version>7.0.1</graphiql.version>
    <graphiql.java.version>6.0.2</graphiql.java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>com.graphql-java-kickstart</groupId>
```

```
        <artifactId>graphql-spring-boot-starter</artifactId>
        <version>${graphiql.version}</version>
      </dependency>
      <dependency>
        <groupId>com.graphql-java-kickstart</groupId>
        <artifactId>graphiql-spring-boot-starter</artifactId>
        <version>${graphiql.version}</version>
        <scope>runtime</scope>
      </dependency>
      <dependency>
        <groupId>com.graphql-java-kickstart</groupId>
        <artifactId>graphql-java-tools</artifactId>
        <version>${graphiql.java.version}</version>
      </dependency>
      <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
      </dependency>
    </dependencies>
    <build>
      <finalName>spring-graphql-service</finalName>
      <plugins>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
      </plugins>
    </build>

</project>
```

Step 2: Create the bookservice.graphqls file

This file should be created under the resources directory. For example

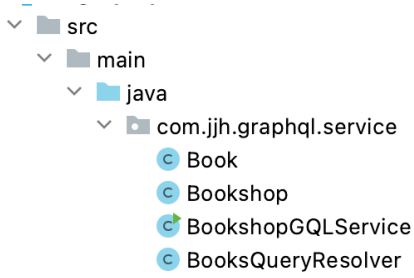
IntelliJ has an (optional) plug-in that can be used to view the contents of the graphqls file – if you have the rights you may be able to install this plug-in but it is not required.

The contents of the bookservice.graphqls file are given below:

```
type Book {
  isbn: ID!
  title: String!
  category: String
  author: String
}

type Query {
  books(count: Int): [Book]!
  allbooks: [Book] !
}
```

Step 3: Create the Book Domain class

We will put all of the Java classes for this application in a single package – call this something like com.jjh.graphql.service, for example:



In this lab we will create a new version of the Book class as given below. This class is intended to be a simplified version of our Book type to allow for a simpler GraphQL implementation.

The Books.java class is given below:

```
package com.jjh.graphql.service;

public class Book {
  private String isbn;
  private String title;
  private String category;
  private String author;

  public Book() {}

  public Book(String isbn, String title, String category, String author) {
    this.isbn = isbn;
    this.title = title;
    this.category = category;
    this.author = author;
  }
```

```
  public String getIsbn() {
    return isbn;
  }

  public String getTitle() {
    return title;
  }

  public void setTitle(String title) {
    this.title = title;
  }

  public String getCategory() {
    return category;
  }

  public void setCategory(String category) {
    this.category = category;
  }

  public String getAuthor() {
    return author;
  }

  public void setAuthor(String author) {
    this.author = author;
  }
}
```

Next we will create a simplified version of the Bookshop.java class:

```
package com.jjh.graphql.service;

import org.springframework.stereotype.Component;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

@Component
public class Bookshop {
  private List<Book> books = new ArrayList<>();

  public Bookshop() {
    books.add(new Book("121", "Java", "Technical", "John Smith"));
    books.add(new Book("345", "Death in the Spring", "Detective", "Denise Jones"));
    books.add(new Book("987", "Henry VI", "Historical", "Phoebe Davies"));
  }
```

```
  public List<Book> getBooks(int count) {
    System.out.println("BookDao.getBooks");
    return books.stream().limit(count).collect(Collectors.toList());
  }

  public List<Book> getAllBooks() {
    return books;
  }

}
```

And we need a class with our main method in it:

```
package com.jjh.graphql.service;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

@SpringBootApplication
public class BookshopGQLService extends SpringBootServletInitializer {

  public static void main(String[] args)  {
    System.out.println("Starting App setup");
    SpringApplication.run(BookshopGQLService.class, args);
    System.out.println("Setup finished");
    System.out.println("\tAccess: http://localhost:8080/graphiql");
  }

}
```

Note that this class is again boiler plate code.

Step 4: Adding GraphQL support

We now need to create a query resolver. We will call this file the BooksQueryResolver and give it the following definition:

```
package com.jjh.graphql.service;

import graphql.kickstart.tools.GraphQLQueryResolver;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class BooksQueryResolver implements GraphQLQueryResolver {
```

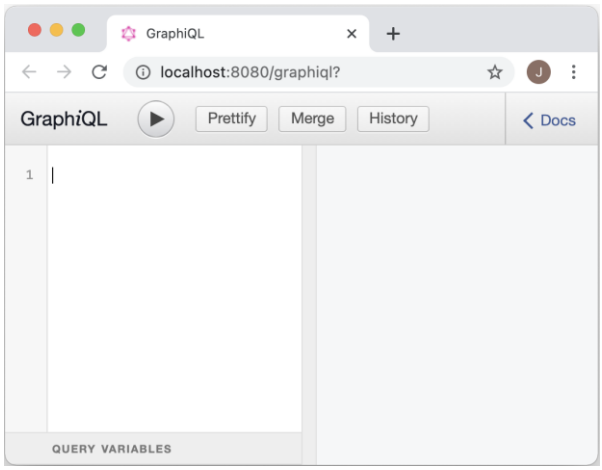
```
  private Bookshop bookshop;

  @Autowired
  public BooksQueryResolver(Bookshop bookshop) {
    this.bookshop = bookshop;
  }

  public List<Book> getBooks(int count) {
    System.out.println("Query.getBooks(" + count + ")");
    return bookshop.getBooks(count);
  }

  public List<Book> getAll_books() {
    System.out.println("Query.getAllBooks()");
    return bookshop.getAllBooks();
  }
}
```

Step 5: Run the application

We can now run the main `BookshopGQLService` in the normal manner.

Step 6: test the Service

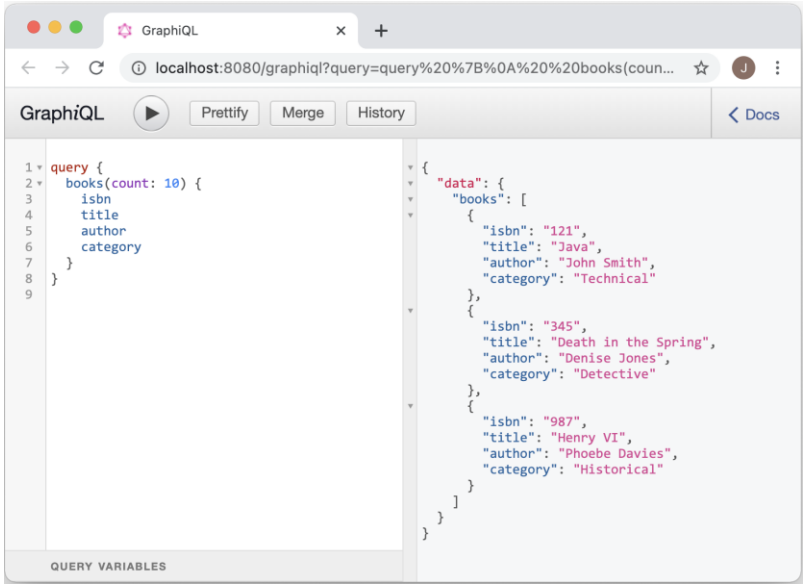
Go to a Browser and open the url http://localhost:8080/graphiql. You should see something like:



In the query window on the left enter:

```
query {
  books(count: 10) {
    isbn
    title
    author
    category
  }
}
```

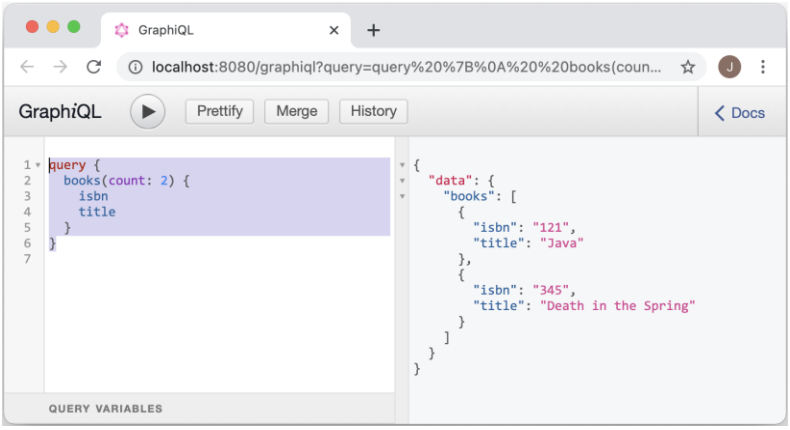
Click on the round arrow button and you should now see the results of running the query:



Now change the information you are requesting by removing the category and author attributes. Also change the number of books to be returned to 2. For example:

```
query {
  books(count: 2) {
    isbn
    title
  }
}
```

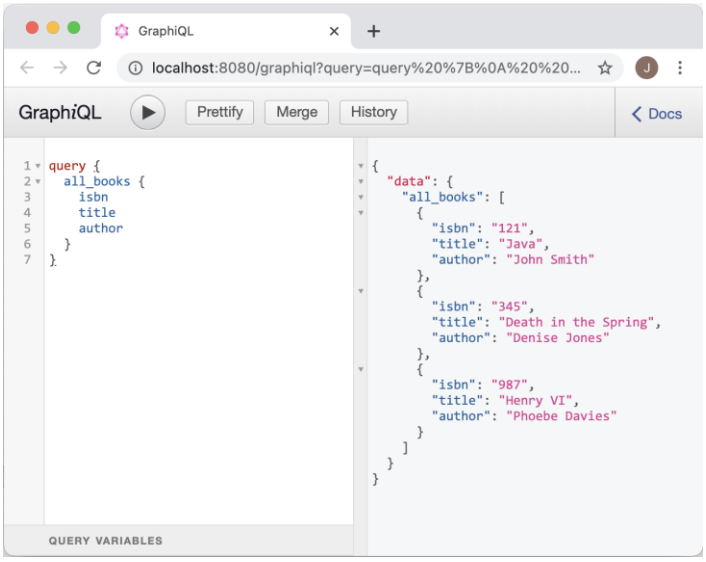
and re run the request. You should now see:



Finally invoke the all_books service. To do this replace the query with:

```
query {
  allboo {
    isbn
    title
    author
  }
}
```

Now re run the query and you should see something similar to:



Lab 4: MySQL Lab

The aim of this lab is to allow you to create and populate tables within a database using MySQL and MySQL Workbench.

Step 1: Make sure that MySQL is running

If MySQL is set up as a service on your machine, then it will already be running.

If not you will need to start the MySQL database server. You can do this by opening a Command Window or Terminal and typing

```
C:\> mysqld
```

If mysqld is not in the Path then you will need to provide the full path to it such as:

```
C:\> "C:\Program Files\MySQL\MySQL Server 8.0\bin\mysqld"
```

The exact path will depend on where MySQL has been installed.

To shutdown the server you can use

```
C:\> mysqladmin -u root shutdown
```

Note if the root account has a password set then you will need to invoke **mysqladmin** with the -p option and supply the password when prompted.

Step 2: Open MySQLWorkbench

Open the mySQLWorkBench application.

When you open the workbench you will need to create a MySQL Connection to connect up to the database you are going to be using.

You can do this by adding a new connection using the '+' symbol next to the MySQL Connections heading.

This will display the 'Setup New Connection' dialog. On this dialog you can specify the Name of the connection, the Username to connect with and password if required.

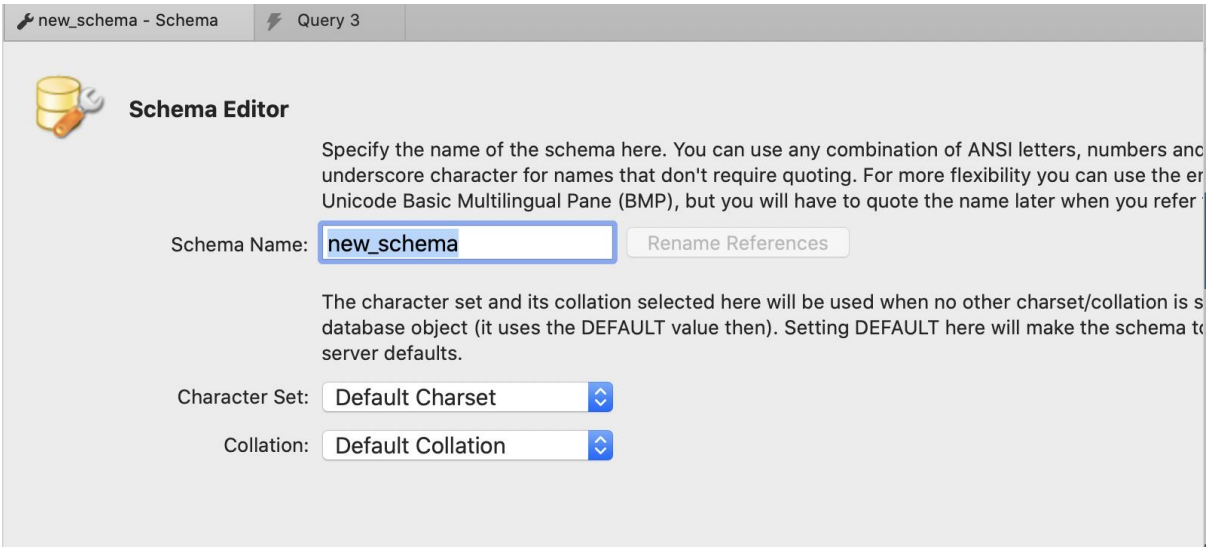
Do this to create a new connection to the MySQL database server.

Now use the 'Test Connection' button at the bottom of the dialog to verify that the connection information supplied is correct.

Step 3: Create the Bookshop schema

Next we will create a new schema for the bookshop database. To do this select the new schema button:

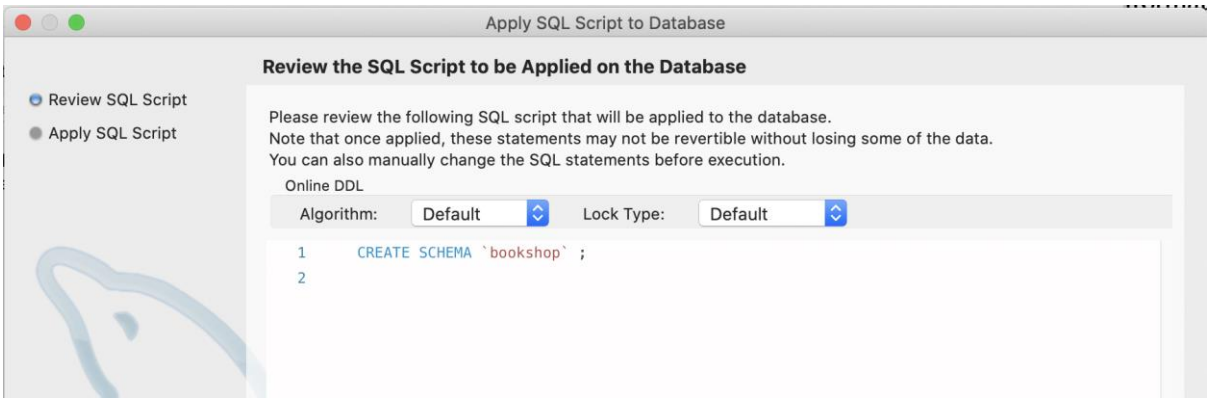
This will display the Schema Editor in the right-hand display panel.



Enter the name of the schema – for example bookshop.

Then click on the ‘apply’ button.

You can then review the DDL statements that will be used by MySQLWorkbench to create the new schema.



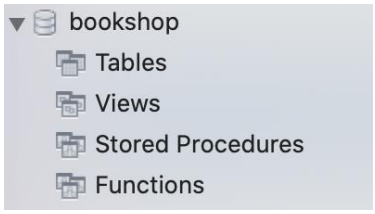
If you are happy with this click on ‘Apply’.

You should now see a new database schema listed on the left-hand side of the workbench.

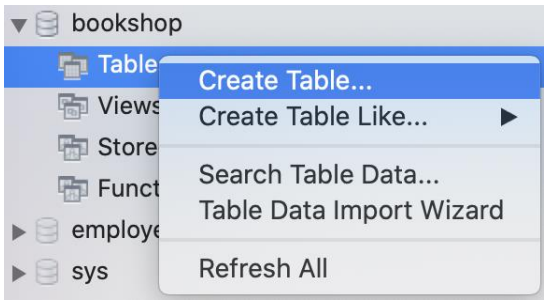
Step 4: Create the books table

Expand the bookshop node in the left-hand schemas tree.

You should see something like:

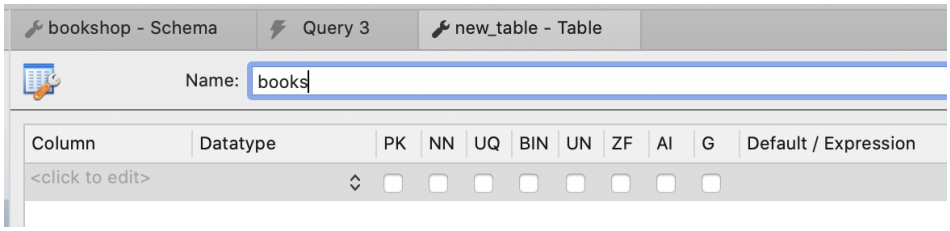


Select the Tables node and then from the right mouse button menu select 'Create Table …':



You will now see the new_table display in the right-hand panel.

Give the table the name books, for example:



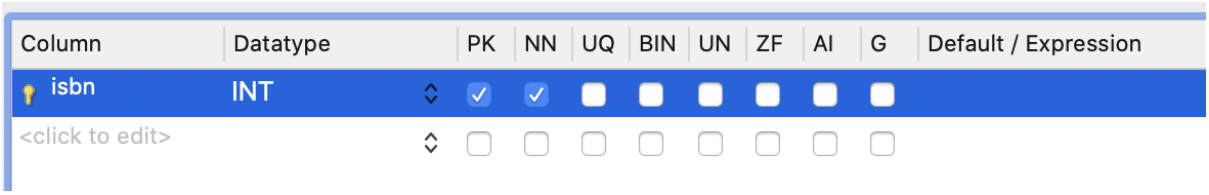
We will now define the structure of the table.

The table will have four columns:

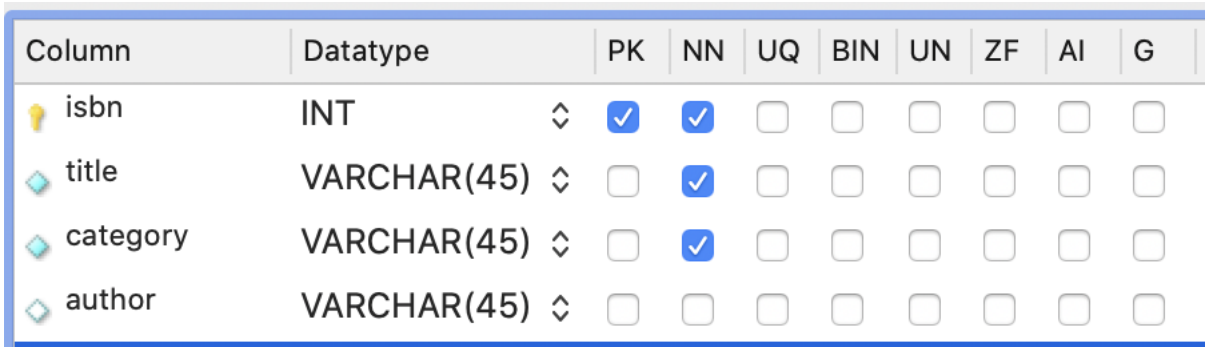
- isbn – unique value used as the primary key of the table
- title- a Vachar and cannot be null – every book must have a title
- category - a Varchar – cannot be null – every book must have a category
- author – a Varchar – can be null as books can be anonymous.

You can define these columns by typing into the table displayed in the new table panel. The Column allows the name to be defined and the check boxes allow specific attributes of that column to be specified such as the type, whether it is the Primary key (PK) or not, whether it can be Null or not etc.

For example, for the isbn column:

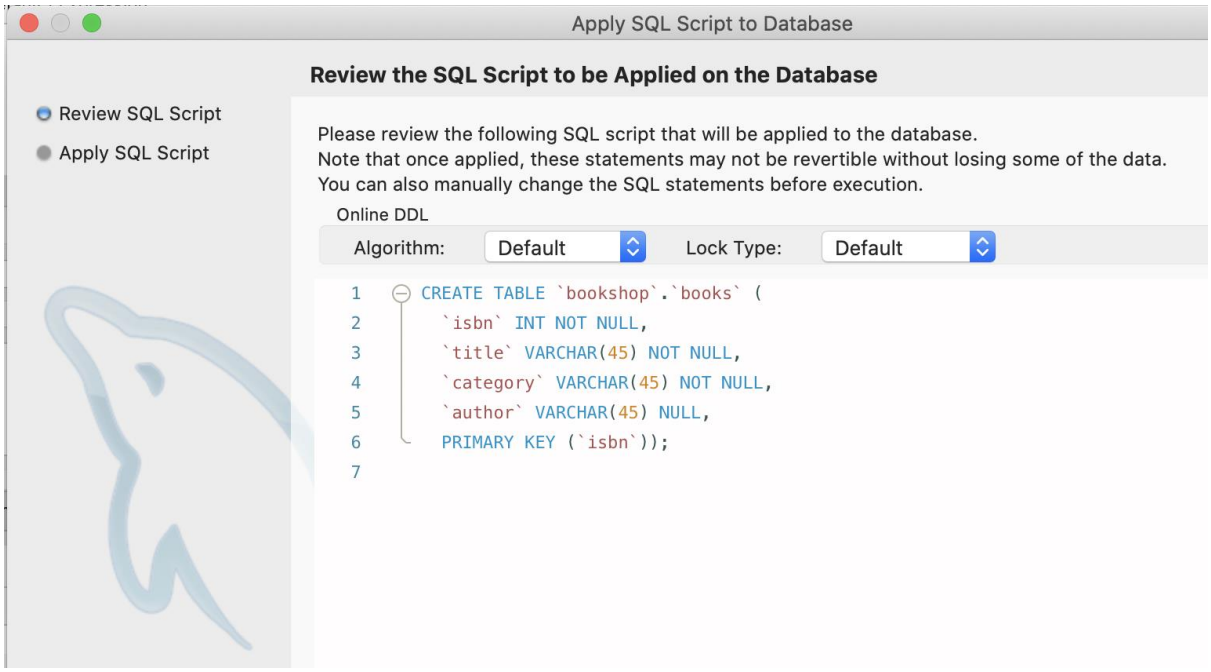


The end result is shown below:



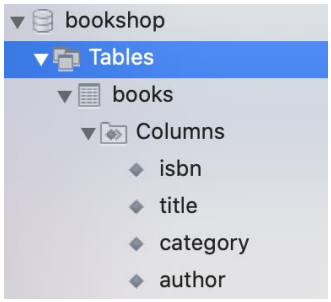
When you have entered this information select the ‘Apply’ button.

You will now see a confirmatory dialog showing you the DDL used to define the table:



Click ‘Apply’

You can now expand the Tables node in the left-hand panel and explore the books table:



Step 5: Populate the Books Table

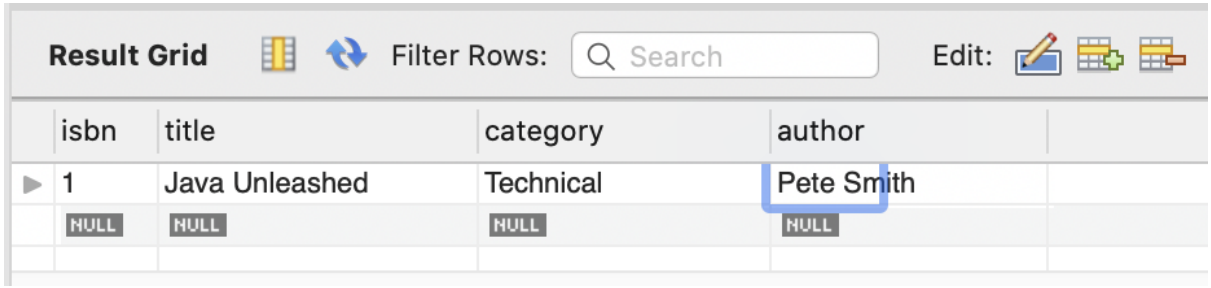
Select the books table in the left-hand panel and then from the right mouse menu select the 'Select Rows – Limit 1000' menu option:

You will now see in the right-hand panel a Query using SQL that can be used to select all entries in the book table. Notice that unless you select the bookshop database as your default schema 'bookshop' prefixes the name of the table.

Within the right-hand panel you should see the Result Grid.

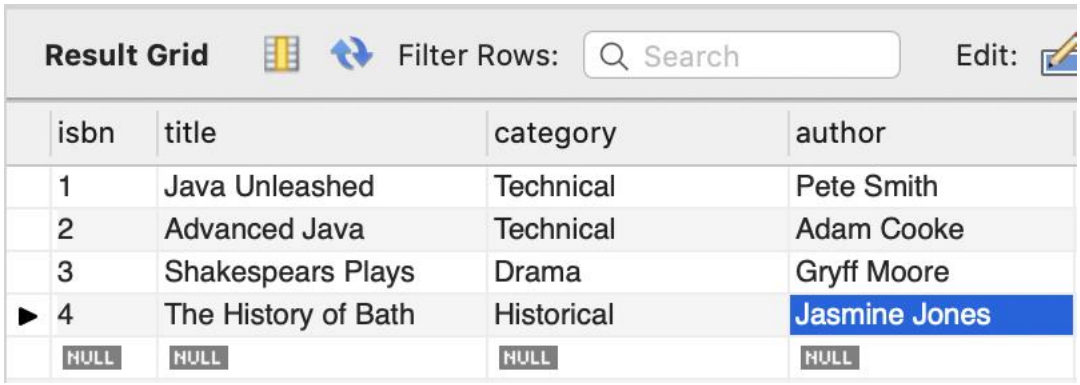
You can use this grid to enter data to be held in the database.

For example:



Enter the details for several books. You can invent whatever titles you want but make at least two of the books Technical books and use the values, 1, 2, 3 etc. for the ISBN numbers.

For example:



Now click on the 'Apply' button on the bottom right-hand side of the table.

You should now see a confirmatory dialog showing the DML statements that would be used to add these rows to the database:

Click on the 'Apply' button.

Step 6: Run some SQL queries

Now use the Select statement at the top of the screen as the basis to write a variety of SQL statements.

For example,

1. add a WHERE clause so that you return the book with the ISBN 2.
2. Modify the WHERE clause to return all books with the category TECHNICAL
3. Add an Order By clause to change the order in which the books are returned

Lab: SQL

The aim of this lab is to write some SQL statements to work with the data in the bookshop database.

You should:

1. Write an SQL query statement to retrieve all the books in the books shop
2. Write an SQL statement to retrieve a book based on its ISBN
3. Write a insert statement to insert a new book into the database (you can provide the details of the book)
4. Now write a query statement to retrieve the book you just added
5. Now write an update statement to change the title of the book
6. Now write a query statement to retrieve the book you just updated
7. Now write a statement to delete the book you just added

Lab: JDBC

The aim of this lab is to write some Java to allow books to be:

1. Retrieved from the database
2. Added to the database
3. Deleted from the database

To do this we will write a BooksDAO (Data Access object) that will act as an interface between the database (and JDBC) and the rest of the application.

Step 1: Create a new Module

To do this create a new Module – as usual this module needs to be a Maven module.

The contents of the POM file should be:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.jjh</groupId>
  <artifactId>bookshop</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <name>students</name>
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.20</version>
    </dependency>
  </dependencies>

</project>
```

Step 2: Set up the domain objects

We will use the definition of the Book class that we used for the GraphQL lab.

Create a package such as com.jjh.bookshop.books and copy the Book class in here (note if the package is different then adjust it as necessary). Note you should change the ISBN to be an int as this is the type we used in the database.

Step 3: Create the DAO object

Create a Data Access Object for the Book class.

This object should handle connecting to the database.

It should also handle:

1. Retrieving a list of all books
2. Retrieving a book based on ISBN
3. Adding a book to the database
4. Deleting a book from the database
5. (if you have time include updating a book in the database as well)

The API for the BookDAO might contain the following methods:

- `public Book getBookByISBN(int isbn) throws SQLException`
- `public List<Book> getAllBooks() throws SQLException`
- `public int saveBook(Book book) throws SQLException`
- `public int deleteBook(Book book) throws SQLException`

Step 4: Update your Bookshop class

This class should now use the BookDAO to retrieve and update books in the bookshop.

Step 5: The BookshopApp

This version of the bookshop application should now look like:

```
package com.jjh.bookshop.books;

import java.sql.SQLException;

public class BookshopApp {
   public static void main(String [] args) throws SQLException {
      Bookshop bookshop = new Bookshop();
      System.out.println(bookshop.getBooks());

      Book b1 = bookshop.getBookByISBN(2);
      System.out.println(b1);

      Book b2 = new Book(7, "Java For Professionals", "Technical", "John Anders");
      bookshop.saveBook(b2);

      System.out.println(bookshop.getBooks());
```

```
        bookshop.deleteBook(b2);
        System.out.println(bookshop.getBooks());

    }
}
```

Step 6: Run the application

An example of the output generated by the sample solution for this application is given below:

```
Bookshop.getBooks
[Book{isbn='1', title='Java Unleashed', category='Technical', author='Pete Smith'},
Book{isbn='2', title='Advanced Java', category='Technical', author='Adam Cooke'},
Book{isbn='3', title='Shakespears Plays', category='Drama', author='Gryff Moore'},
Book{isbn='4', title='The History of Bath', category='Historical', author='Jasmine Jones'}]
Book{isbn='2', title='Advanced Java', category='Technical', author='Adam Cooke'}
Bookshop.saveBook(Book{isbn='7', title='Java For Professionals', category='Technical',
author='John Anders'})
INSERT INTO books (isbn, title, category, author) VALUES('7', 'Java For Professionals',
'Technical', 'John Anders')
Bookshop.getBooks
[Book{isbn='1', title='Java Unleashed', category='Technical', author='Pete Smith'},
Book{isbn='2', title='Advanced Java', category='Technical', author='Adam Cooke'},
Book{isbn='3', title='Shakespears Plays', category='Drama', author='Gryff Moore'},
Book{isbn='4', title='The History of Bath', category='Historical', author='Jasmine Jones'},
Book{isbn='7', title='Java For Professionals', category='Technical', author='John Anders'}]
Bookshop.deleteBooks
DELETE FROM books WHERE ISBN = '7'
Bookshop.getBooks
[Book{isbn='1', title='Java Unleashed', category='Technical', author='Pete Smith'},
Book{isbn='2', title='Advanced Java', category='Technical', author='Adam Cooke'},
Book{isbn='3', title='Shakespears Plays', category='Drama', author='Gryff Moore'},
Book{isbn='4', title='The History of Bath', category='Historical', author='Jasmine Jones'}]
```

Lab: JPA

The aim of this lab is to refactor the BookDAO class so that it now uses JPA (and Hibernate under the hood) rather than directly using JDBC.

You will need to update your POM so that it includes appropriate JPA dependencies:

```
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.20</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.1-api</artifactId>
    <version>1.0.2.Final</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.4.15.Final</version>
  </dependency>
</dependencies>
```

Next you will need to make your Book class into an Entity, for example:

```
@Entity
@Table(name="books")
public class Book {

  @Id
  private int isbn;
  private String title;
  private String category;
  private String author;
```

You should replace the JDBC statements in your DAO with appropriate (equivalent) JPA operations using an EntityManager.

Note that JPA does not throw managed exceptions nor do operations such as delete return an int. you should therefore adjust the API for the DAO accordingly, for example it might now look like:

```
public Book getBookByISBN(int isbn
public List<Book> getAllBooks()
public void saveBook(Book book)
public void deleteBook(Book book)
```

You should end up with less code in the DAO class than before.

Don't forget to set up a suitable persistence.xml file. Mine is:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
       http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <persistence-unit name="BookshopJPA" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <properties>
      <property name="hibernate.connection.url" value="jdbc:mysql://localhost/bookshop" />
      <property name="hibernate.connection.driver_class" value="com.mysql.cj.jdbc.Driver" />
      <property name="hibernate.connection.username" value="user" />
      <property name="hibernate.connection.password" value="user123" />
      <property name="hibernate.archive.autodetection" value="class" />
      <property name="hibernate.connection.pool_size" value="10" />
      <!-- <property name="hibernate.connection.autocommit" value="true"/> -->
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
      <!-- validate/create/update/create-drop -->
      <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>

  </persistence-unit>

</persistence>
```

Now rerun your application.

Lab: Data Repository

The aim of this lab is to use the Spring Data Repository mechanism to simplify the use of a database further.

Step 1: Update your POM

You will need to update your POM so that it now has the following dependencies and parent POM.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.7.RELEASE</version>
    <relativePath /> <!-- lookup parent from repository -->
  </parent>

  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>demo</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>1.8</java.version>
  </properties>
  <dependencies>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <scope>runtime</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
```

```
    </plugins>
  </build>

</project>
```

Step 2: Define a Repository class

You need to create a class that extends the CrudRepository abstract class, for example:

```
package com.jjh.bookshop.books;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface BookRepository extends CrudRepository<Book, Integer> {}
```

Step 3: Define an application.yaml file

Define an application YAML file that can eb used to handle the database connection:

```
spring:
 datasource:
  url: jdbc:mysql://localhost:3306/bookshop
  password: user123
  username: user
  driver-class-name: "com.mysql.cj.jdbc.Driver"
 jpa:
  database-platform: org.hibernate.dialect.MySQL5InnoDBDialect
  hibernate:
   ddl-auto: update
```

Step 4: Update the Bookshop

Update the Bookshop class so that it now uses the BookRepository rather than the JPA based DAO you previously used.

Step 5: Update the BookshopApp

The BookshopApp now needs to be a Spring based application.

The easiest way to so this is to annotate your class with @SpringBootApplication and then access the Bookshop object from the Spring contact (gives access to all objects managed by Spring). For example:

```
package com.jjh.bookshop.books;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;
```

```
@SpringBootApplication
public class BookshopApp {
   public static void main(String [] args)  {
      System.out.println("Starting Bookshop setup");
      ConfigurableApplicationContext context = SpringApplication.run(BookshopApp.class, args);
      System.out.println("Setup finished");

      Bookshop bookshop = context.getBean(Bookshop.class);
      System.out.println(bookshop.getBooks());

      Book b1 = bookshop.getBookByISBN(2).get();
      System.out.println(b1);

      Book b2 = new Book(7, "Java For Professionals", "Technical", "John Anders");
      bookshop.saveBook(b2);

      System.out.println(bookshop.getBooks());

      bookshop.deleteBook(b2);
      System.out.println(bookshop.getBooks());
   }
}
```

Step 6: Now run the application

Run the application as normal.

Lab: Jenkins

The aim of this lab is to allow you to set up your own Jenkins CI server and to configure a job.

Jenkins has already been downloaded for you to your machines.

You should find that a file – the Jenkins.war file is available.

This is the 'Generic Java package' or war file.

Step 1: Starting Jenkins

Copy the Jenkins.war file to an appropriate location, for example to a course directory under the user home.

Open a Command Window / terminal and change directory to the location that you have stored the Jenkins war file in.

Run the following command:

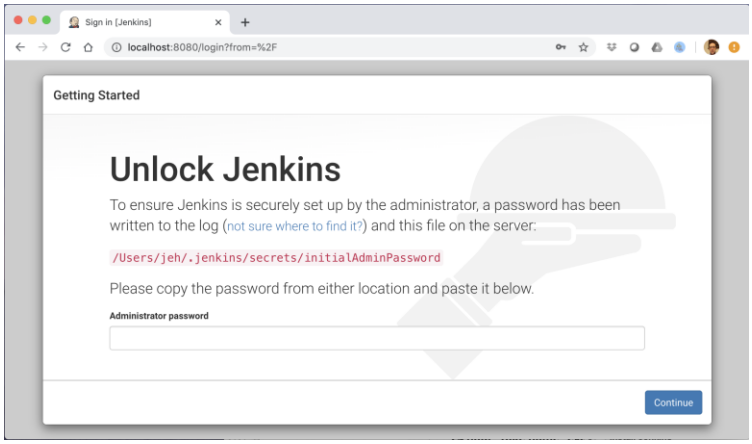
```
java -jar jenkins.war --httpPort=8080 --enable-future-java
```

This will start the Java Virtual Machine (JVM) – it will run the jenkins.war file passing in defaults for the port and for the use of newer Java features.

Step 2: Unlock Jenkins

Open a web browser to http://localhost:8080 to complete installation.

You should now see



The administrator password can be found in the command line output. Look for:

****

```
************************************************************
************************************************************
```

Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:

2e0fbb69000841a5b7fee69d013aaa65

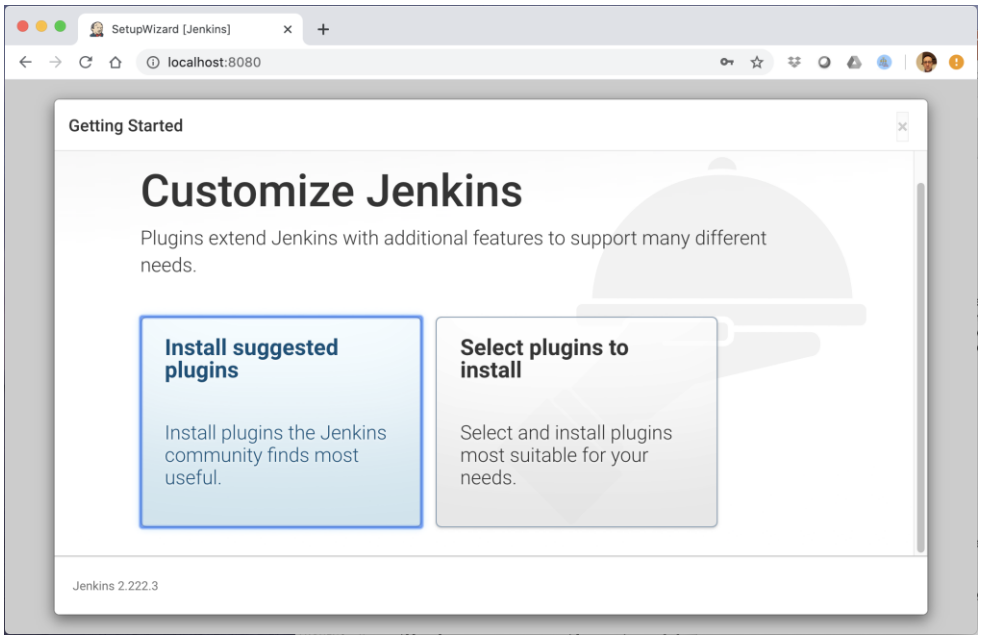
This may also be found at: /Users/jeh/.jenkins/secrets/initialAdminPassword

```
************************************************************
************************************************************
************************************************************
```

Where the password should be different for you as well as the location of this information.

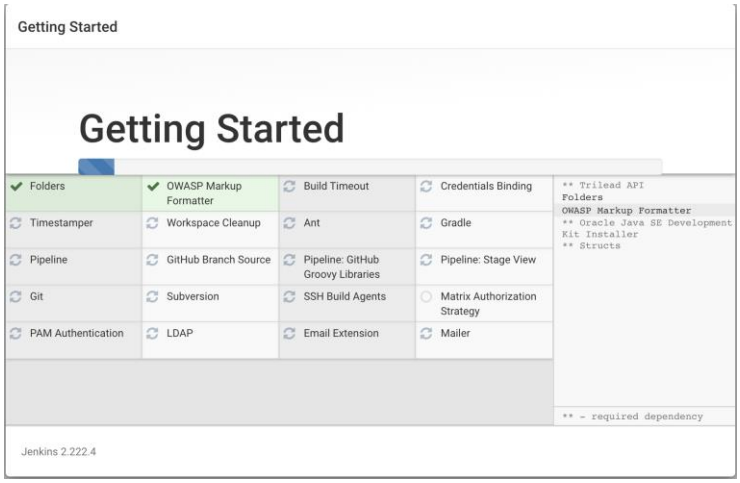
Step 3: Setup Jenkins

Once you are logged in you should see:

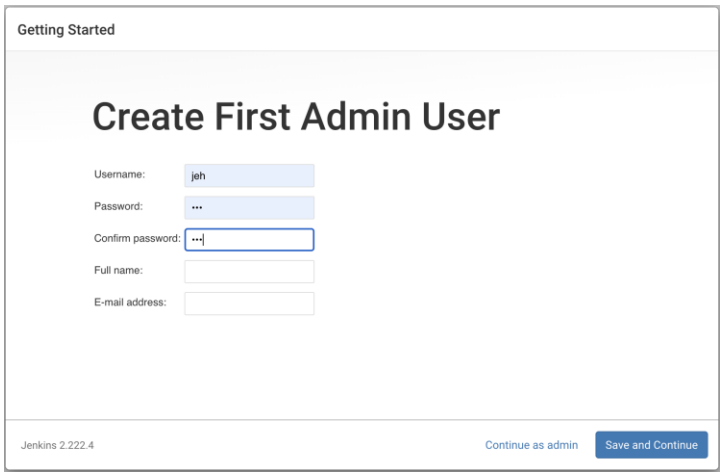


Click on the ‘Install suggested plugins’ option.

You will now see something like:

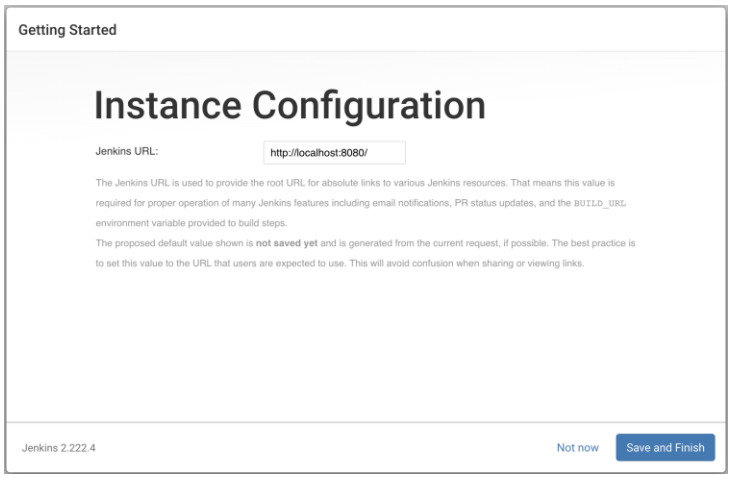


You will now be asked to create a first admin user:

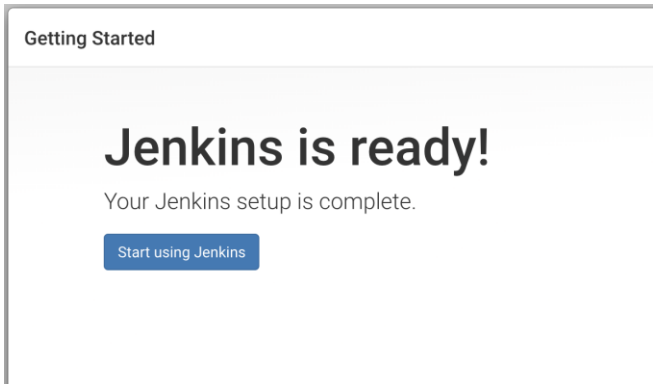


Enter a user name and password and then save and continue.

Finally confirm the URL – by clicking on Save and Finish:



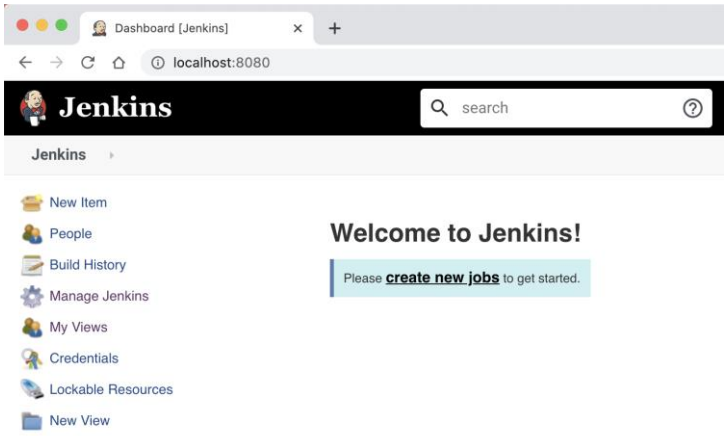
You should now see:



Click on ‘Start using Jenkins’.

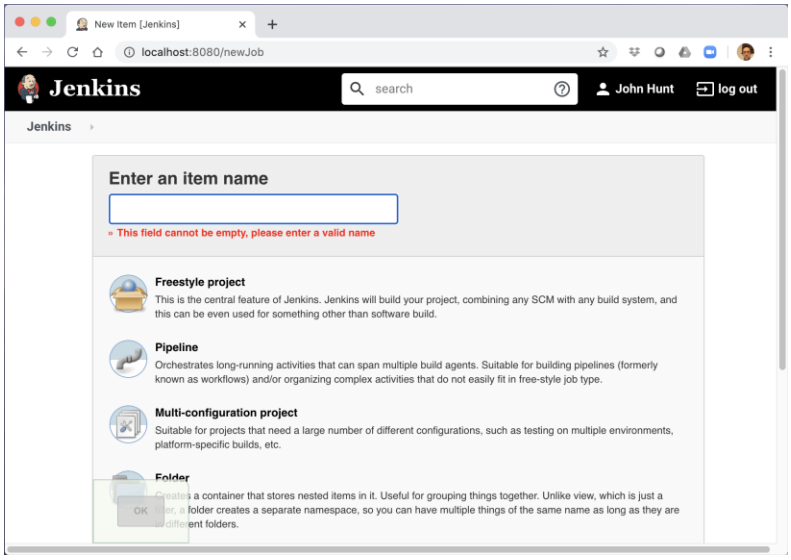
Step 4: Set up a Job

You should now be presented with the default Jenkins desktop as shown below:

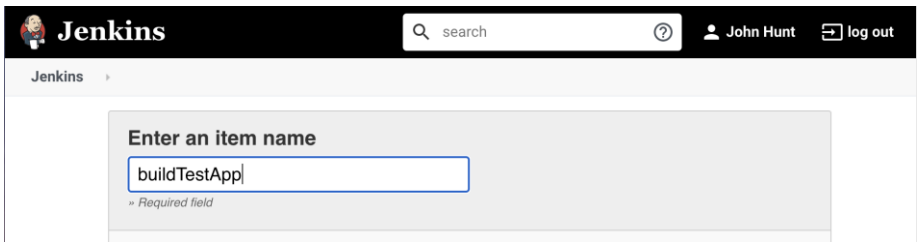


Click on the ‘create new jobs’ link.

You will now be shown the job creation page:

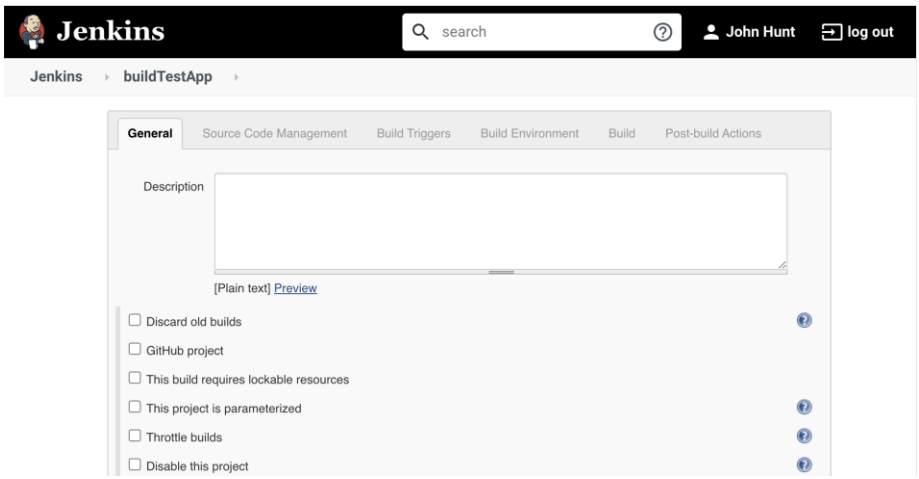


Provide a name for this job – for example, ‘buildTestApp’



Select ‘Freestyle project’ and then click on OK.

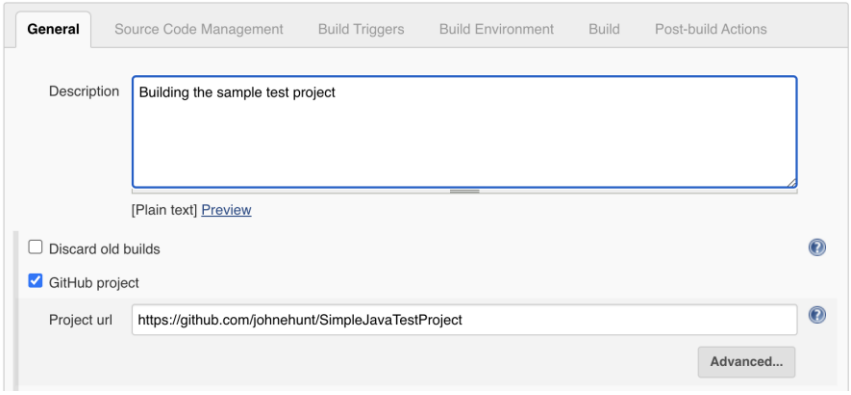
You will now see the job configuration page.



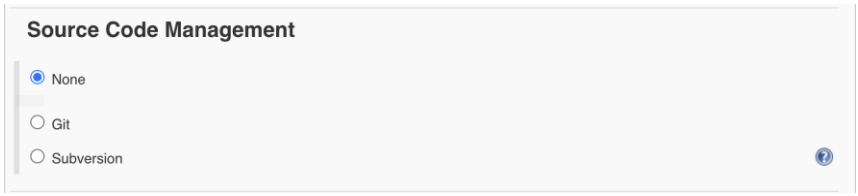
In this dialog:

- Provide a description for the job
- Select GitHub project
- Provide the URL for your Git hub repo

For example:

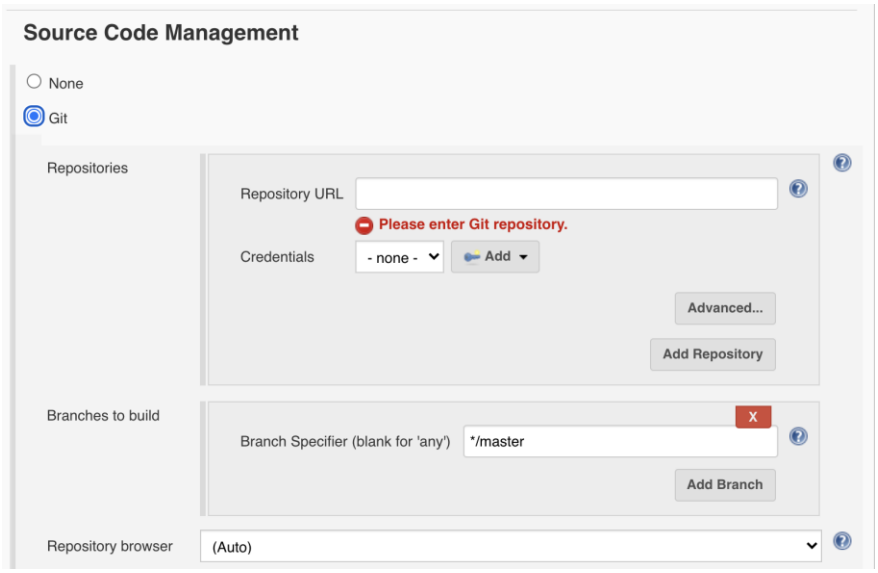


Next scroll down to the 'Source Code Management' section:



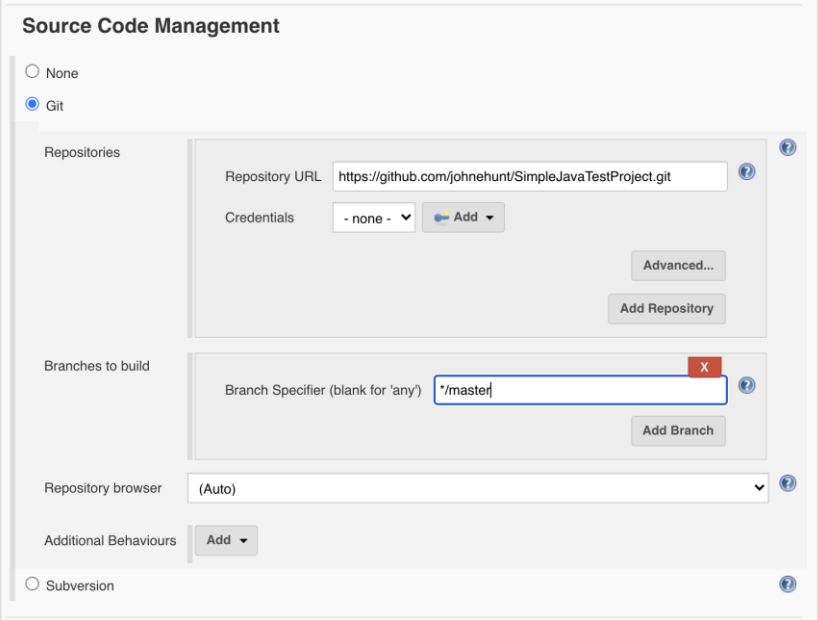
Select the Git option

This will result in the repository details panel being displayed:



You will need to provide the Git repo URL and the branch to use.

You should select the branch that is the one you have been using for your project work.

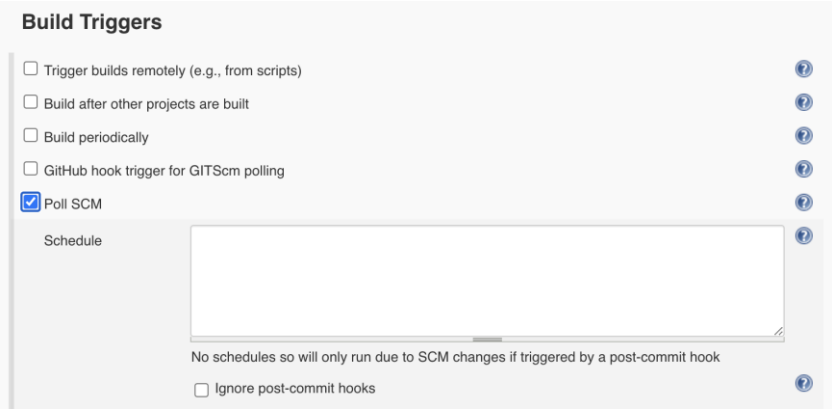


Now scroll down to the Build Triggers section

This is used to specify when the build will be triggered.

We will select to poll the SCM (Scourde Code Management system) at regular intervals.

Therefore click on the 'Pool SCM' option.



This displays the schedule panel. This is used to specify the schedule used for polling.

This field follows the syntax of cron (with minor differences). Specifically, each line consists of 5 fields separated by TAB or whitespace:
`MINUTE HOUR DOM MONTH DOW`

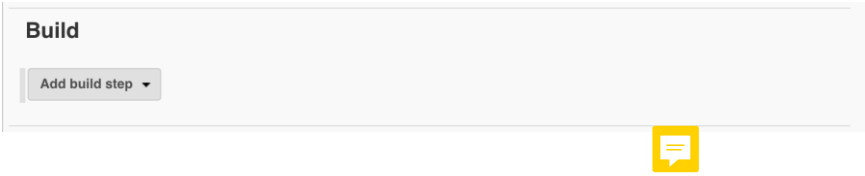
MINUTE	Minutes within the hour (0–59)
HOUR	The hour of the day (0–23)
DOM	The day of the month (1–31)
MONTH	The month (1–12)
DOW	The day of the week (0–7) where 0 and 7 are Sunday.

Examples:

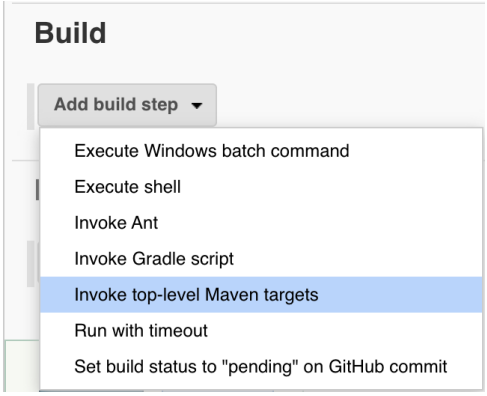
```
# every fifteen minutes (perhaps at :07, :22, :37, :52)
H/15 * * * *
# every ten minutes in the first half of every hour (three times, perhaps
at :04, :14, :24)
H(0-29)/10 * * * *
# once every two hours at 45 minutes past the hour starting at 9:45 AM and
finishing at 3:45 PM every weekday.
45 9-16/2 * * 1-5
# once in every two hours slot between 9 AM and 5 PM every weekday (perhaps
at 10:38 AM, 12:38 PM, 2:38 PM, 4:38 PM)
H H(9-16)/2 * * 1-5
# once a day on the 1st and 15th of every month except December
H H 1,15 1-11 *
```

Select something appropriate for this project, for example every 5 minutes using the Hash option to balance the load:

Next scroll down to the Build option and select the ‘Add build step’:



From the drop-down menu displayed select the ‘Invoke top-level Maven targets’ option:



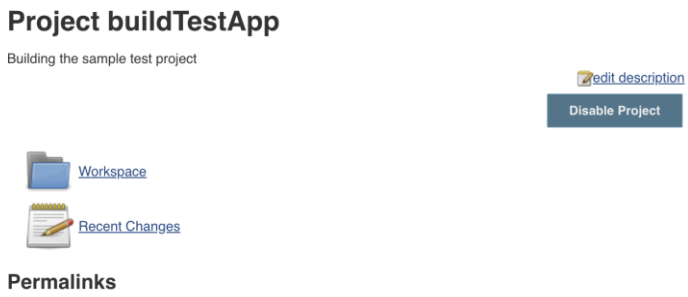
You will then be presented with the Maven targets panel:

In the goals enter a set of Maven targets, such as clean install:

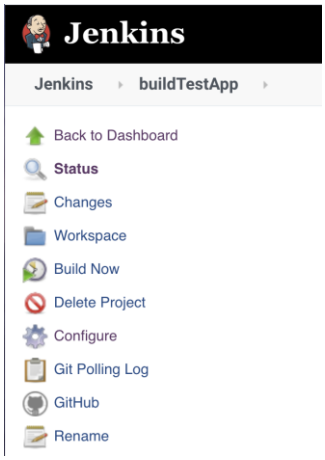
Now select the ‘Save’ button.

Step 5: Run the Jenkins Job

You will now be taken back to the main Jenkins desktop and should see your project listed there:



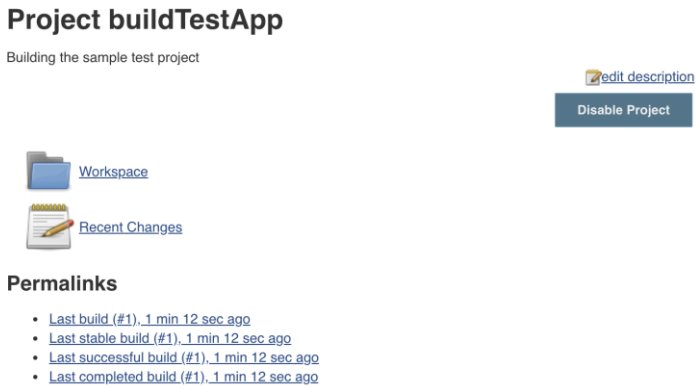
Next click on the ‘Build now’ option on the left-hand side of the Jenkins desktop:



Alternatively wait 5 minutes for your job to start.

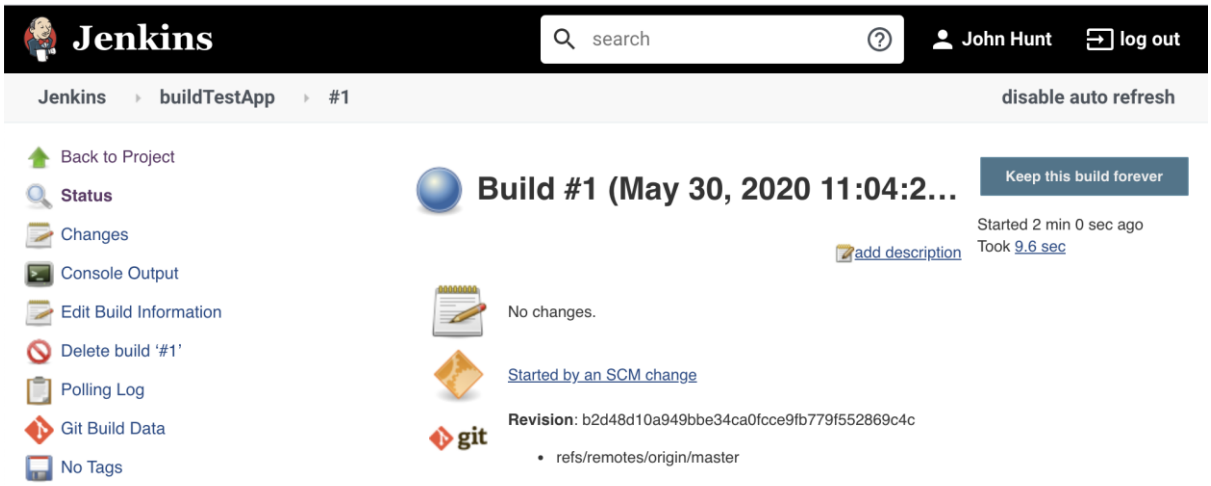
Once the job starts you should see a progress bar indicating that the job is building.

Once the build has completed you should see that the Jenkins desktop has updated with the results of the latest build:



Click on the last build link.

You will now see the Status page listing the git version, how it was started etc:



Click on the ‘Console output’ on the left-hand side.

You should now see the results of the operations you issued:

```
tearDownAfterClass
[WARNING] Tests run: 6, Failures: 0, Errors: 0, Skipped: 1, Time elapsed: 0.076
s - in com.jjh.sample.CalculatorTest
[INFO]
[INFO] Results:
[INFO]
[WARNING] Tests run: 6, Failures: 0, Errors: 0, Skipped: 1
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ sample ---
[INFO] Building jar: /Users/jeh/.jenkins/workspace/buildTestApp/target/sample-
0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ sample ---
[INFO] Installing /Users/jeh/.jenkins/workspace/buildTestApp/target/sample-
0.0.1-SNAPSHOT.jar to /Users/jeh/.m2/repository/com/jjh/sample/0.0.1-
SNAPSHOT/sample-0.0.1-SNAPSHOT.jar
[INFO] Installing /Users/jeh/.jenkins/workspace/buildTestApp/pom.xml to
/Users/jeh/.m2/repository/com/jjh/sample/0.0.1-SNAPSHOT/sample-0.0.1-
SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4.301 s
[INFO] Finished at: 2020-05-30T11:04:38+01:00
[INFO] Final Memory: 14M/47M
[INFO] ------------------------------------------------------------------------
Finished: SUCCESS
```

If you now return to the main dashboard – do this by selecting Jenkins in the top left-hand edge below the logo – you should see the history of recent builds including your build: