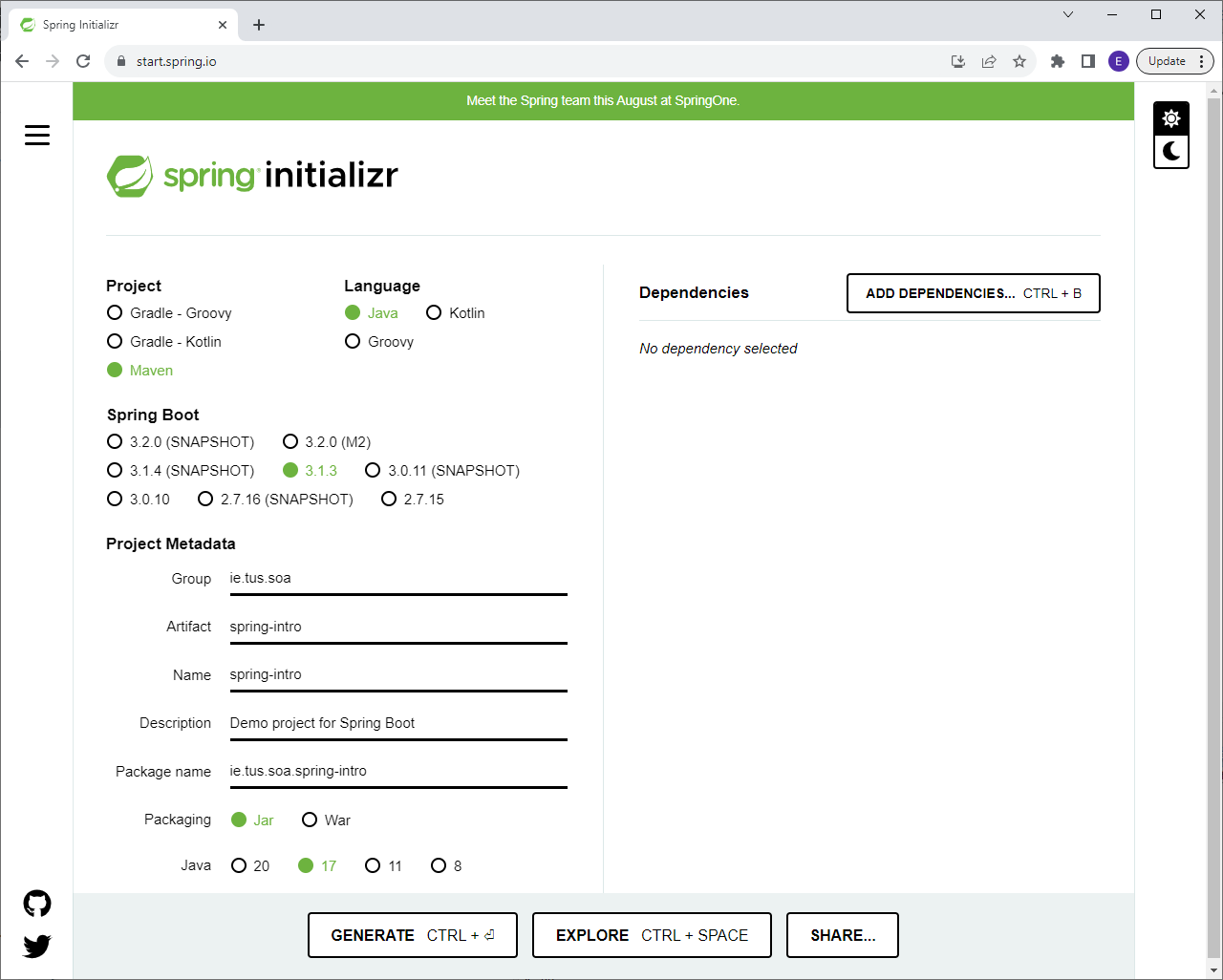
**Lab – Intro to Spring**

**Step 1: Go to start.spring.io and download a starter project.**



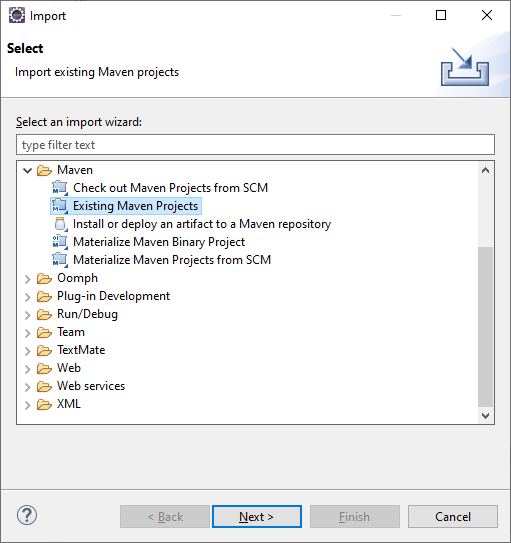
**Notes:**

* Select Maven
* Use latest version 3.x.x that isn’t a snapshot
* Adjust Group and Artifact as shown
* Finally, click ‘Generate’ to download the project.

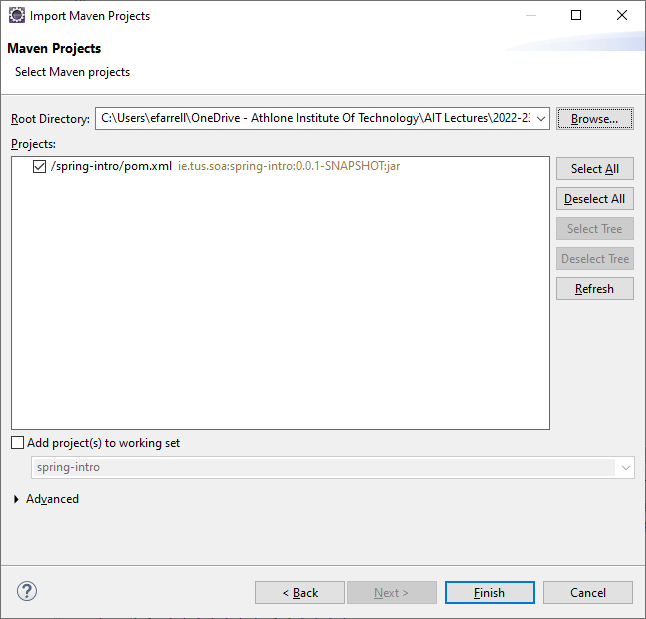
**Step 2: Import into Eclipse**

Extract the starter project zip file.

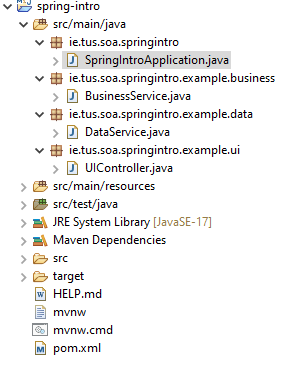
In Eclipse, choose File – Import, and choose ‘Existing Maven Projects’:



Browse to where you unzipped the starter project to and click ‘Finish’ to import the project:



**Step 3:**



Create the files as shown above (take care to get the package names correct).

This outlines a typical ‘layered’ application that you’ll often find in enterprise applications. Typical layered applications split an application into layers to better organise the application. A typical layering might include *UserInterface – Business – Data – Database*. Each layer depends on the layer below it i.e. contains an object of the layer below it. This can get complex and difficult to manage in enterprise applications – Spring makes it easier by managing a lot of this process of creating objects and passing them to other objects etc.

The following files will outline a typical (very simple) layered application:

**SpringIntroApplication.java** – Autogenerated by Spring so no need to create this!

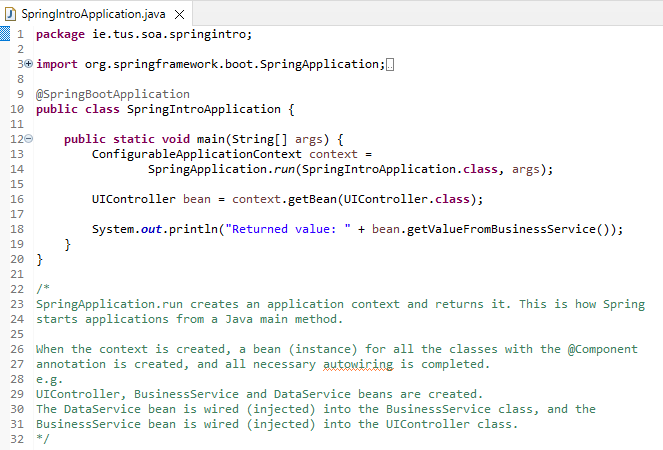
**UIController.java** – Mimics a User Interface that will be presented to the user of an application. A class in the User layer of an application.

**BusinessService.java** – A class in the Business layer of an application.

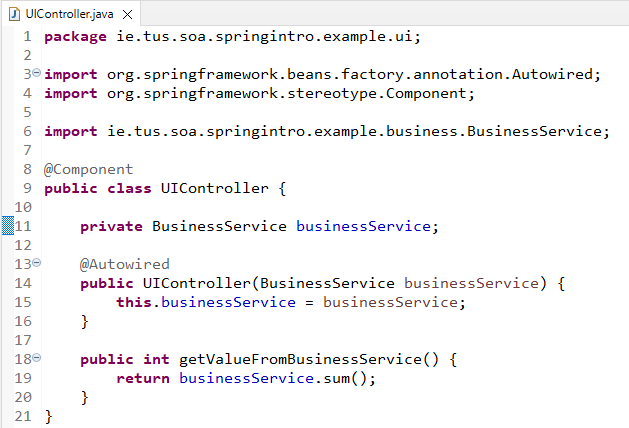
**DataService.java** – A class in the Data layer of an application.

The classes are outlined in the following slides:

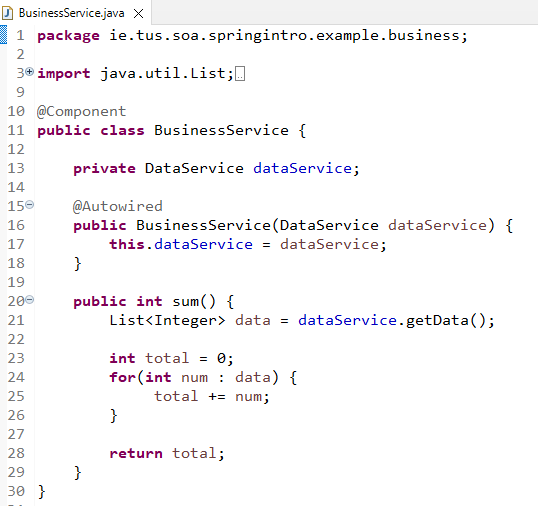
**SpringIntroApplication.java**



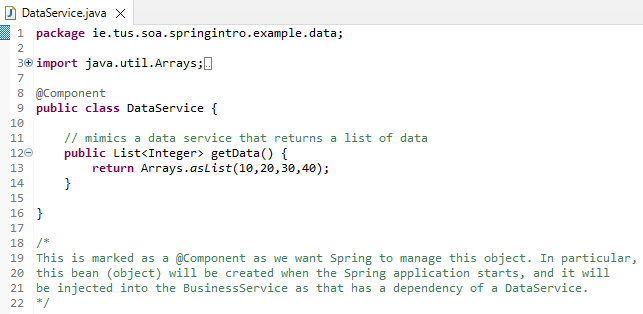
**UIController.java**



**BusinessService.java**



**DataService.java**



**Terminology:**

The goal of this lab is to learn the technology and structure involved in using Spring Applications. The following are a review of the concepts we saw in the previous slides:

**@Component** is a class that’s managed by the Spring Framework.

**Dependency** – something you need to be able to create an instance of (a specific class) e.g. if there’s another class @Autowired, then that’s a dependency. DataService is a dependency for the BusinessService. BusinessService needs DataService to carry out its operations. You can say that the BusinessService is ***dependent*** on the DataService.

**Component Scan**: How does Spring Framework find @Component classes?

Answer: searches packages defined in @ComponentScan

This is defined in all @SpringBootApplication classes and the **default** behaviour is to scan in the current package and all subpackages

Can also define your own:

@ComponentScan(“com.tus.ie”) // one package

@ComponentScan({“com.tus.ie”, “com.ait.ie”}) // two packages

**Dependency Injection**: Spring is known as a dependency injection framework – it identifies beans, their dependencies and wires them together (provides **IOC** – Inversion of Control)

**IOC** – Spring framework is in control of creating the objects, not us!

**Spring Beans**: Objects managed by Spring Framework – creation and lifecycle managed by Spring Framework.

**IoC container**: Component inside the Spring framework that manages the lifecycle of beans and dependencies

Types: ApplicationContext(complex), BeanFactory(simpler features – rarely used)

**Autowiring**: Process of wiring in dependencies for a Spring Bean

**Point of Spring Framework**

Many enterprise applications are very complex with many layers and classes in those layers (User Interface – Business – Data – Database)

With Spring Framework, instead of focusing on objects, their dependencies and wiring, you can focus on the business logic of the application

* Spring manages the lifecycle of components:
  + Mark components with annotations: @Component etc.
  + Mark dependencies using @Autowired

**Dependency Injection Types**

**Constructor-based**: Dependencies are set by creating the Bean using its Constructor. You can put the @Autowired on the constructor as seen in the examples. All the examples we’ll use are of this type – the type recommended by spring.

**Setter-based**: Dependencies are set by calling setter methods on your beans. In this case the @Autowired is put on the setter method of the field you’re setting.

**Field**: No setter or constructor. Dependency is injected using reflection. In this case the @Autowired is put on the field being set.

Which one should you use?

* Spring recommends Constructor-based injection as dependencies are automatically set when an object is created.

**Final Note:**

You’ll have noticed that when we run the application nothing happens (apart from the Spring application starting). Usually this is the case with Spring Boot and Web Applications. You’ll remember from second year that a web application is invoked via a client e.g. Web Browser, Web Client (HTML/JavaScript) or a program like Postman. We’ll see this in future labs when we add RESTful methods to an application.

The point of this lab was to see how we can create different classes and wire them together with Spring Boot.