Spring Boot Auto-configuration for Tanzu GemFire

Suppose we need to create a customer service application that stores customer data and allows the user to search for customers by name. This lab is a modified version of the guide provided by John Blum at apache-geode-docs

We will walk you through building a simple Customer Service Spring Boot application using Apache Geode to manage Customer interactions. You should already be familiar with Spring Boot and Apache Geode.

By the end of this lesson, you should have a better understanding of what Spring Boot for Apache Geode's (SBDG) *auto-configuration* support actually does. Additional resources can be found here: link:https://docs.spring.io/spring-boot-data-geode-build/current/reference/html5/guides/boot-configuration.html link:https://docs.spring.io/spring-boot-data-geode-build/current/reference/html5/index.html#geode-samples

You can either work from an editor, vscode (code) or Spring Tool Suite. The directory for the first lab project can be found by:

```
cd spring-geode-workshop/configuration
```

Your choice of command line editors are

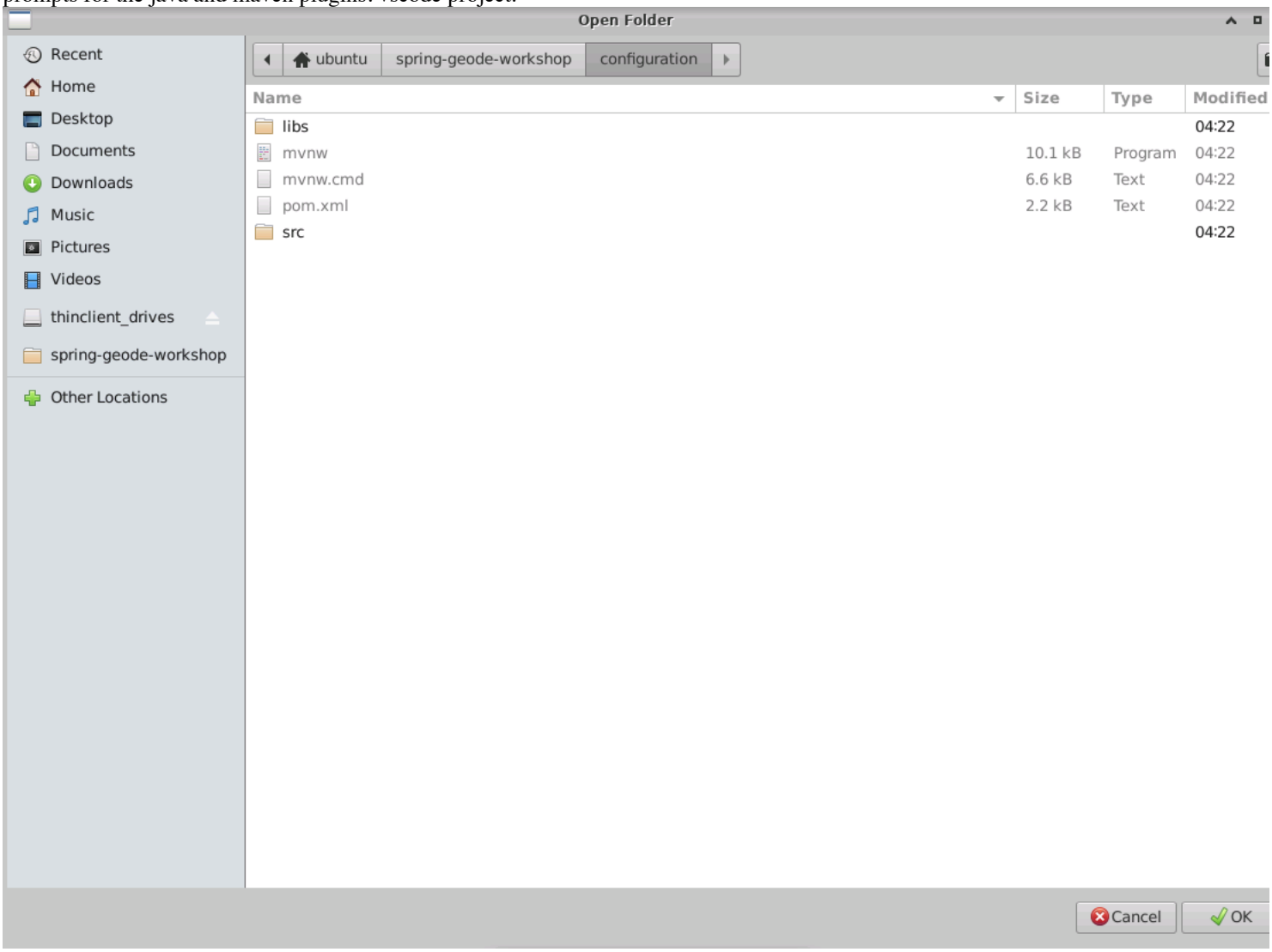
- emacs
- vi

and your choice of IDEs available from the command line that are found on your $PATH are:

- SpringToolSuite4
- code

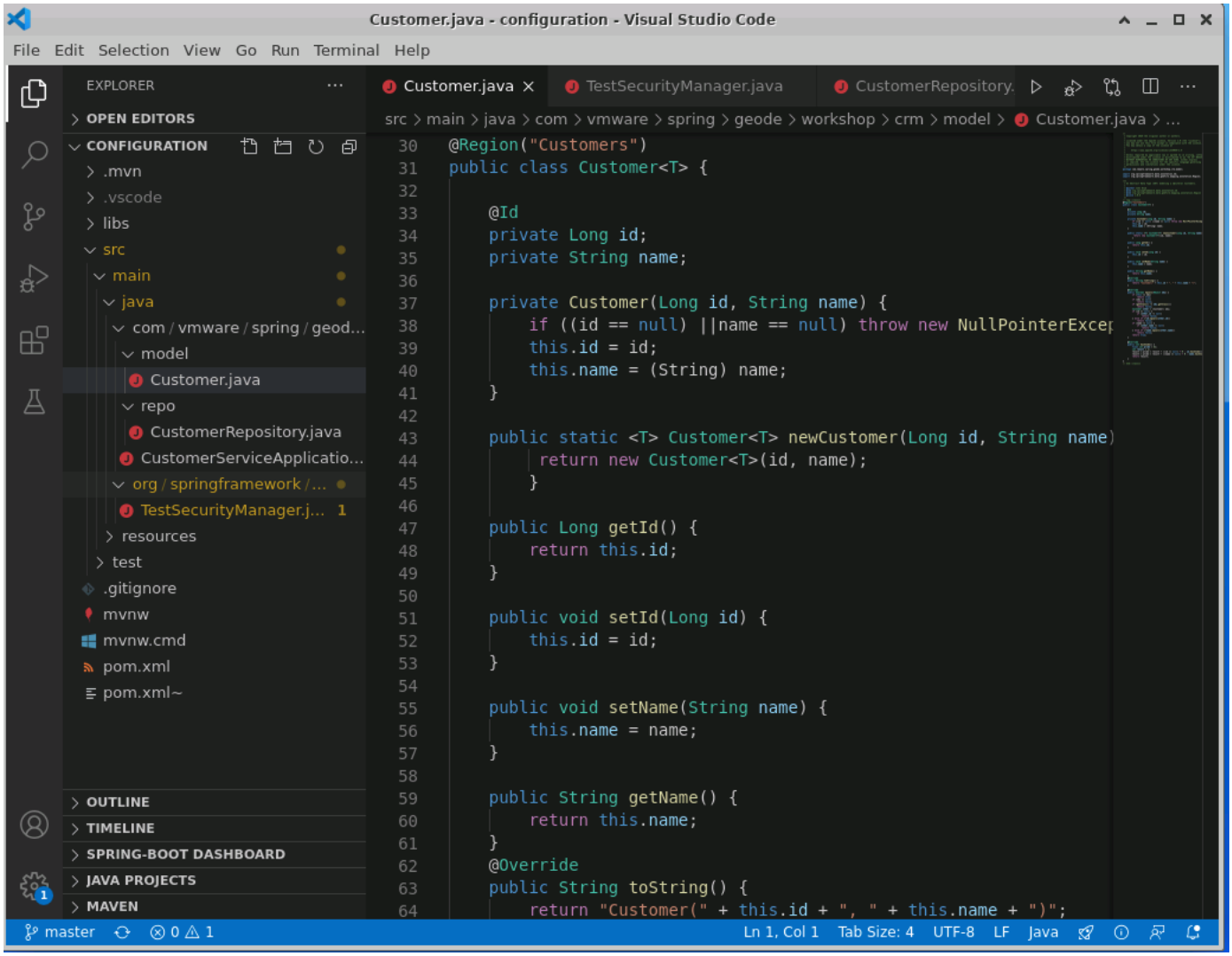
The packages and classes have been previously created. You will want to review the code and pay special attention to the annotations for he respective stereotypes used in the solution for the lab. In Visual Studio Code simply Open the Folder listed above and accept the

prompts for the java and maven plugins. vscode project:



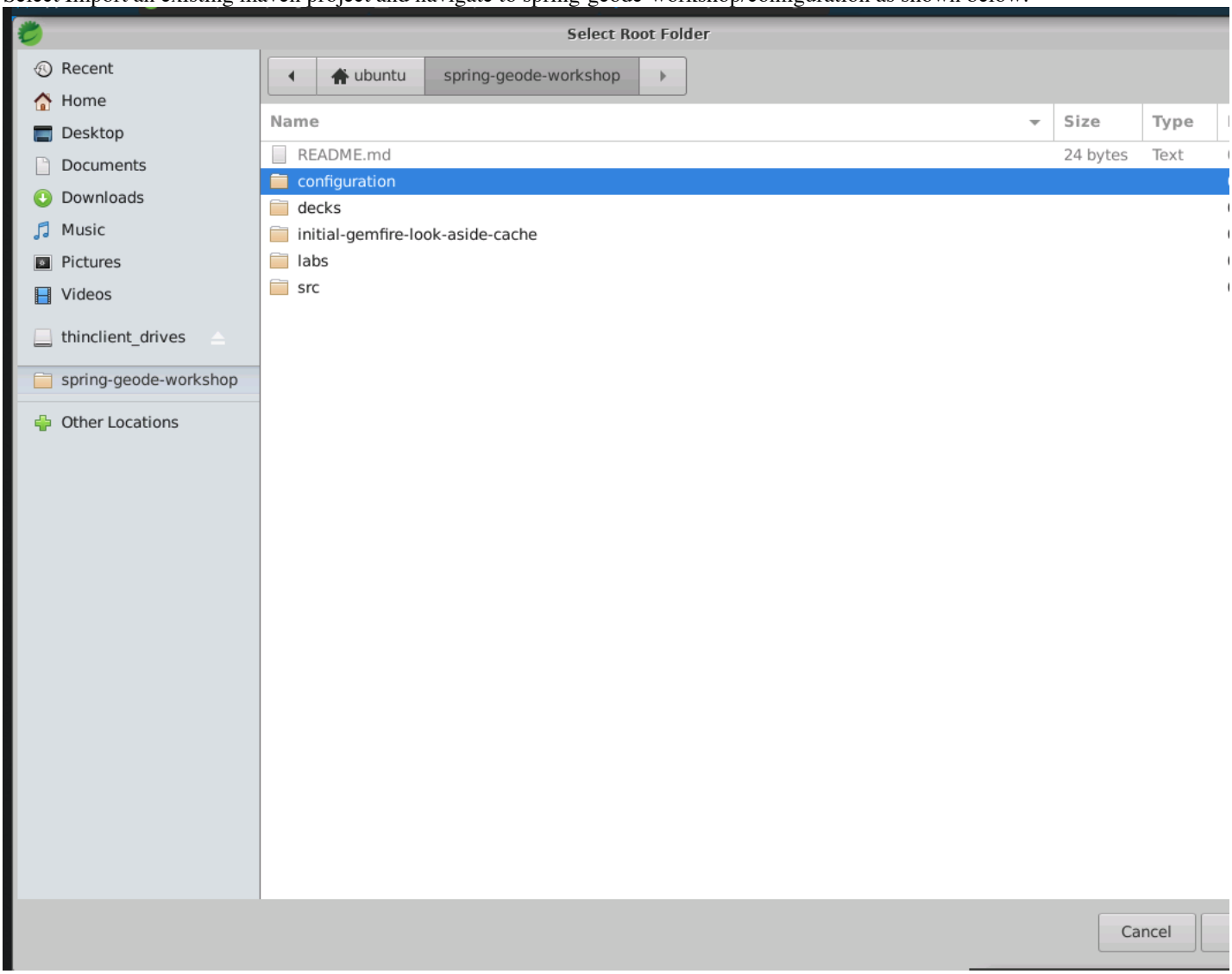
Wih the configuration project open you will be able to navigate through the various packages to review the src code under the standard src folder (src/main/java) as shown below:

Review Src Code:

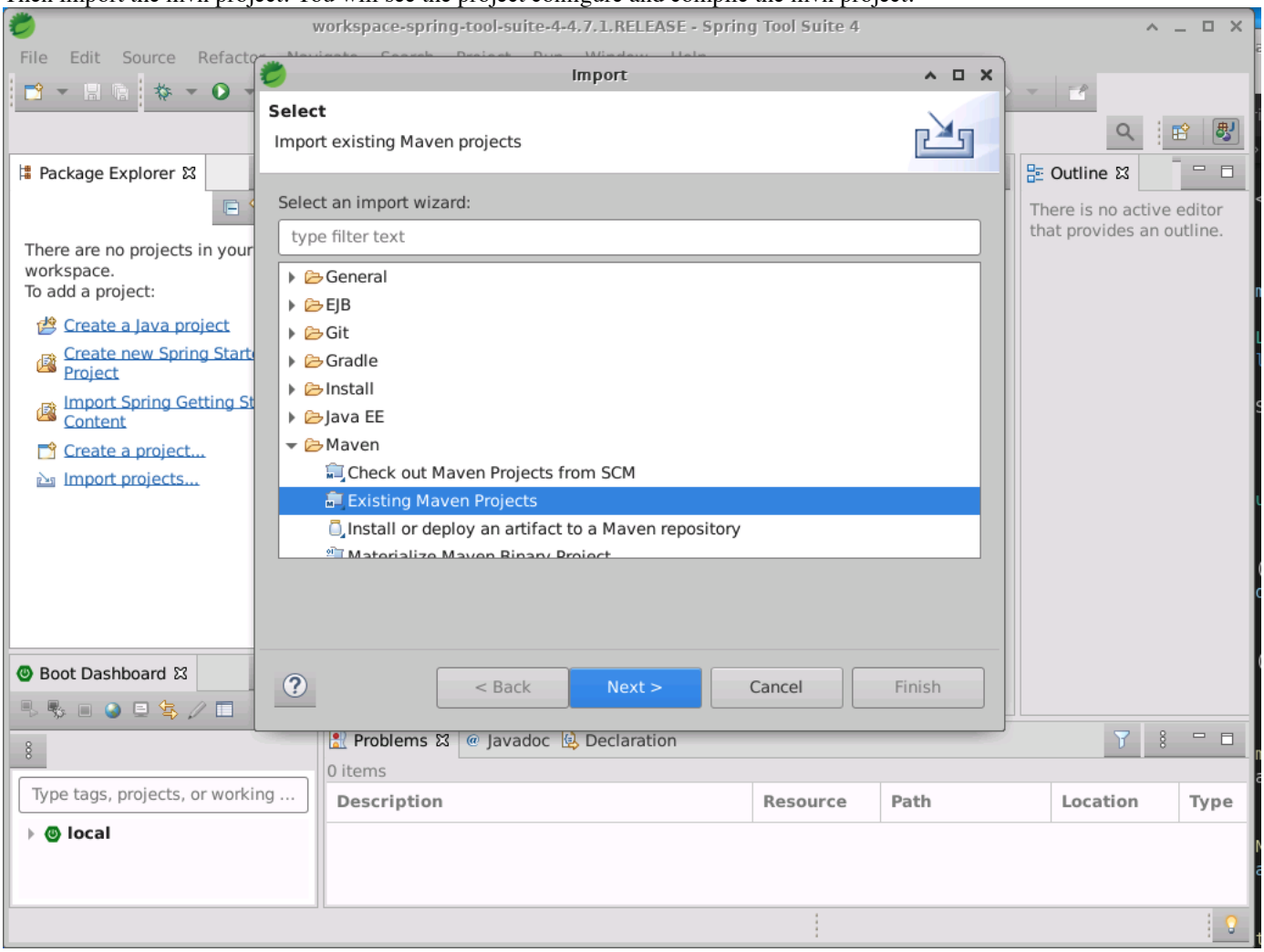


With Spring Tool Suite you follow a similar approach.

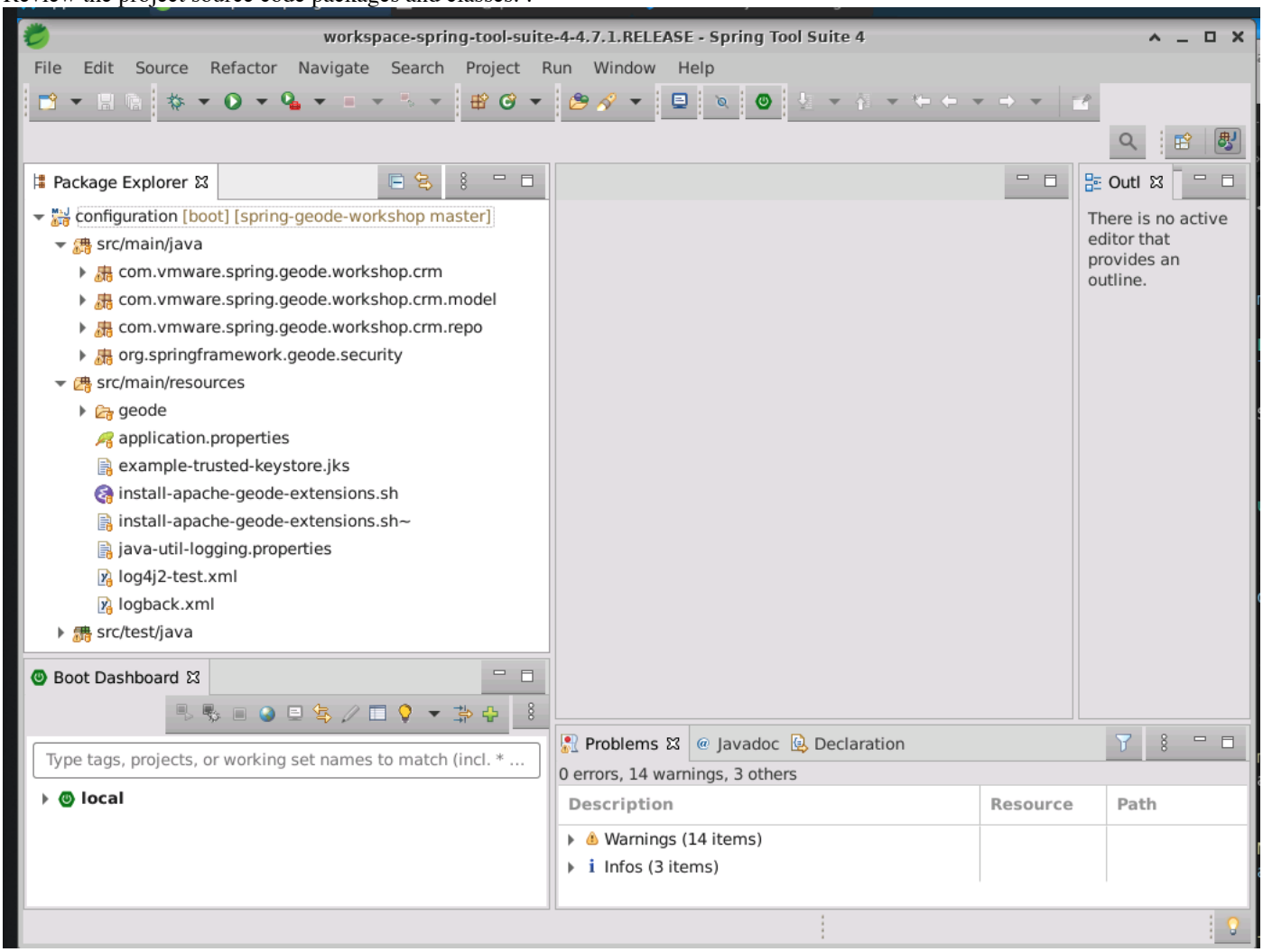
1. Select Import an existing maven project and navigate to spring-geode-workshop/configuration as shown below:



2. Then import the mvn project. You will see the project configure and compile the mvn project.



3. Review the project source code packages and classes: :



Application Domain Classes

We will review the src code from the view of building the Spring Boot, Customer Service application from the ground up.

`Customer` class

First, we need to define an application domain object to encapsulate customer information. We begin by modeling the data our application needs to manage, namely a `Customer`. For this example, the `Customer` class is implemented as follows:

Customer class.

```
@Region("Customers")
public class Customer<T> {

    @Id
    private Long id;
    private String name;

    private Customer(Long id, String name) {
        if ((id == null) ||name == null) throw new NullPointerException("name required");
        this.id = id;
        this.name = (String) name;
    }

    public static <T> Customer<T> newCustomer(Long id, String name) {
         return new Customer<T>(id, name);
        }
}
```

The `Customer` class is annotated with Spring Data Geode's (SDG) `@Region` annotation. `@Region` is a mapping annotation declaring the Apache Geode cache `Region` in which `Customer` data will be persisted. A `Region` is a distributed version of `java.util.Map`.

Tthe `@Id` annotation is used to designate the `Customer.id` field as the identifier for `Customer` objects. The identifier is the Key used in the Entry stored in the "Customers"`Region`. A `Region` is a distributed version of `java.util.Map`.

`CustomerRepository` interface

Next, we create a *Data Access Object* (DAO) to persist `Customers` to Apache Geode. We create the DAO using Spring Data's *Repository* abstraction:

CustomerRepository interface.

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {

    Customer findByNameLike(String name);

}
```

`CustomerRepository` is a Spring Data `CrudRepository`. `CrudRepository` provides basic CRUD (CREATE, READ, UPDATE, and DELETE) data access operations along with the ability to define simple queries on `Customers`.

Spring Data Geode will create a proxy implementation for your application-specific *Repository* interfaces, implementing any query methods you may have explicitly defined on the interface in addition to the data access operations provided in the `CrudRepository` interface extension.

In addition to the base `CrudRepository` operations, `CustomerRepository` has additionally defined a `findByNameLike(:String):Customer` query method.

`CustomerServiceApplication` (Spring Boot main class)

Now that we have created the basic domain classes of our Customer Service application, we need a main application class to drive the interactions with Customers:

CustomerServiceApplication class.

CustomerServiceApplication class.

```
@SpringBootApplication
@EnableClusterAware
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
public class CustomerServiceApplication {

    public static void main(String[] args) {

        new SpringApplicationBuilder(CustomerServiceApplication.class)
            .web(WebApplicationType.NONE)
            .build()
            .run(args);
    }

    @Bean
    ApplicationRunner runner(CustomerRepository customerRepository) {

        Fairy fairy = Fairy.create();

        // Generate a random name from data generator
        Person person = fairy.person();


    return args -> {
            // Locating the max id to be enable incrementing of our ID.
            Long id = 0L;
            id = customerRepository.count();

            assertThat(customerRepository.count()).isEqualTo(id);

            Customer randomCustomer = Customer.newCustomer(id + 1L, person.fullName());
```

```
            System.err.printf("Saving Customer [%s]%n", randomCustomer);

            randomCustomer = customerRepository.save(randomCustomer);

            assertThat(randomCustomer).isNotNull();
            assertThat(randomCustomer.getId()).isEqualTo(id + 1);
            assertThat(randomCustomer.getName()).isEqualTo(person.fullName());
            assertThat(customerRepository.count()).isEqualTo(randomCustomer.getId());

            String query = "Querying for Customer [SELECT * FROM /Customers WHERE name LIKE " + randomCustome
            System.err.println(query);

            Customer queriedrandomCustomer = customerRepository.findByNameLike(person.fullName());

            assertThat(queriedrandomCustomer).isEqualTo(randomCustomer);

            System.err.printf("Customer was [%s]%n", queriedrandomCustomer);
        };
    }
}
```

The `CustomerServiceApplication` class is annotated with `@SpringBootApplication`. Therefore, the main class is a proper Spring Boot application equipped with all the features of Spring Boot (e.g. *auto-configuration*).

We then use a @ClusterAware annotation. "The @EnableClusterAware annotation is arguably the most powerful and valuable Annotation in the set of Annotations!" - See here.

We use @EnableEntityDefinedRegions(basePackageClasses = Customer.class) to locate our Domain class that has our region annotation so that the Region will be created in whatever topology is deployed.

We use Spring Boot's `SpringApplicationBuilder` in the `main` method to configure and bootstrap the Customer Service application.

Then, we declare a Spring Boot `ApplicationRunner` bean, which is invoked by Spring Boot after the Spring container (i.e. `ApplicationContext`) has been properly initialized and started. Our `ApplicationRunner` defines the Customer interactions performed by our Customer Service application.

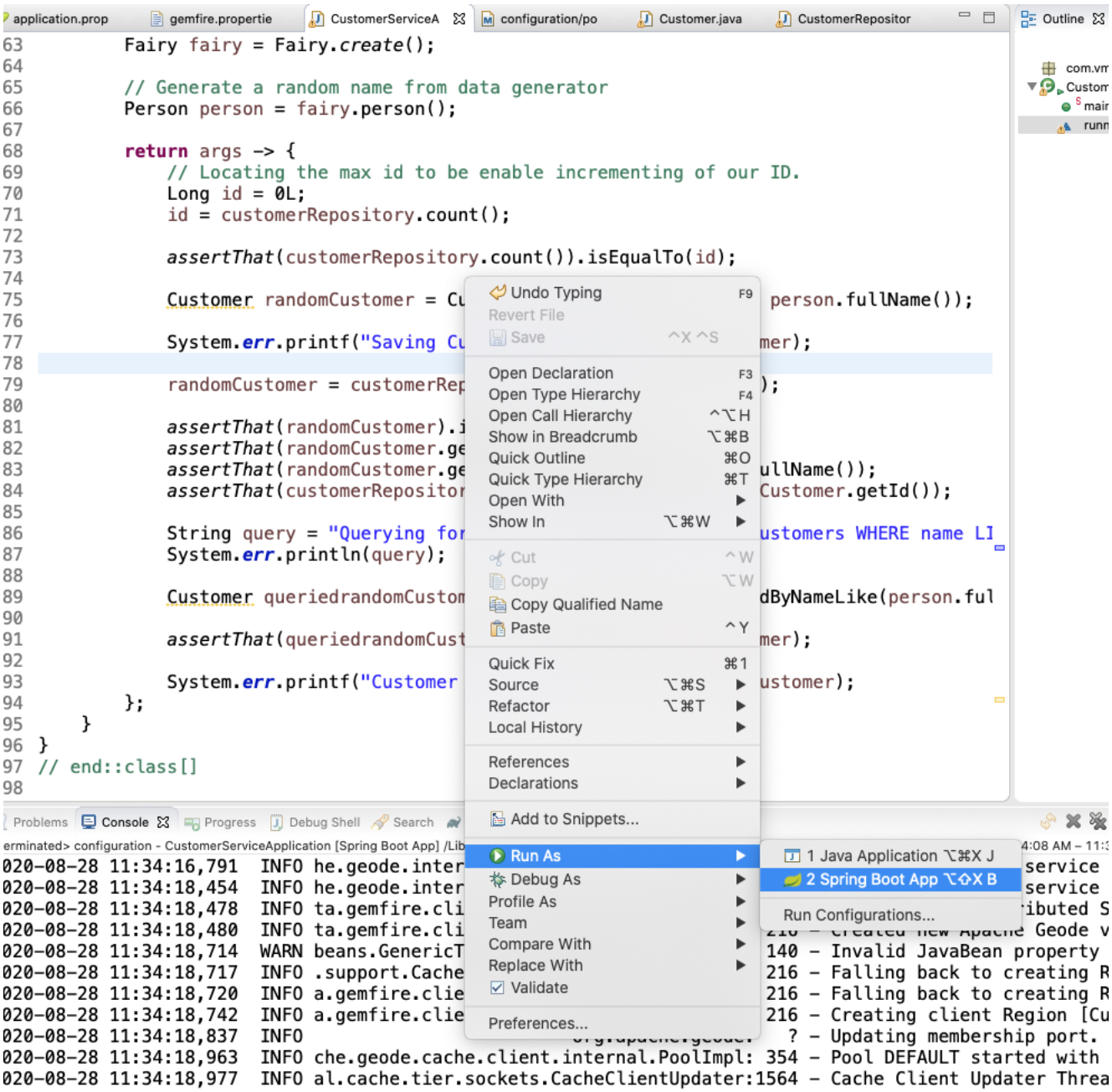
Specifically, the runner creates a new random `Customer` object generated through a mock data generator called jfairy.`.

`%` is the wildcard for OQL text searches.

Running the Example

Running within the IDE

You can run the `CustomerServiceApplication` class from your IDE (e.g. Spring Tool Suite or IntelliJ IDEA). From the IDE (STS) you can right click on the main class and runa spring boot app. STS run command:



Running from the command line

```
mvn clean package`
```

Then run the `CustomerServiceApplication` class from the command-line using the command

```
$ mvn spring-boot:run
```

We have just run our application taking advantage of SBDG to automatically detect that we are in local mode with no connection to a gemfire cluster. SBDG will create our region for us in memory and store our data in a local cache or some refer to this configuration as a near cache.

Auto-configuration for Apache Geode, Take One

While it is not apparent, there is a lot of hidden, intrinsic power provided by Spring Boot Data Geode (SBDG) in this example.

Cache instance

First, in order to put anything into Apache Geode you need a cache instance. A cache instance is also required to create `Regions` which ultimately store the application's data (state). Again, a `Region` is a Key/Value data structure, similar to a `java.util.Map`, mapping a

Key to a Value or an Object.

SBDG is opinionated and assumes most Apache Geode applications will be client applications in Apache Geode's *Topologies and Communication* Therefore, SBDG auto-configures a `ClientCache` instance by default. You can disable these features by adding:

```
@SpringBootApplication(exclude = ClientCacheAutoConfiguration.class)
```

but in most case there is an easier way to move from a local ClientCache to a Client/Server Topology.

Without *auto-configuration* settin gup and runnning this quickly would not be possible. SBDG assumes standard practices for the gemfire developer to be able to begin being productive immediately when using Spring Boot

Switching to Client/Server

Initially we want to get up and running as quickly as possible. By not starting a locator/region and using *@EnableClusterAware* we allow SBDG to detect that we are using a client `LOCAL` Region and we are not required to start a cluster of Gemfire servers for the client to be able to store data. This allows rapid local development for developers.

While client `LOCAL` Regions can be useful for some purposes (e.g. local processing, querying and aggregating of data), it is more common for a client to persist data in a cluster of servers, and for that data to be shared by multiple clients (instances) in the application architecture, especially as the application is scaled out to handle demand.

We continue with our example by switching from a local context to a client/server topology.

If you are rapidly prototyping and developing your application and simply want to lift off the ground quickly, then it is useful to start locally and gradually migrate towards a client/server architecture.

To switch to client/server, all you need to do is startup a locator(s) and server(s) and SBDG will automatically detect your topology.

There are several ways in which to start a cluster. For this example, we are going to use the tools provided with Tanzu Gemfire, i.e. *Gfsh* (GemFire/Geode Shell). For this workshop, we have installed GFSH for you.

To start the command shell locally, open a command/terminal prompt and type $ gfsh

Running Gfsh.

```
    _________________________     __
   / _____/ _______/ ______/ /____/ /
  / /  __/ /___   /_____   / _____  /
 / /__/ / ____/   _____/ / /     / /
/______/_/       /______/_/     /_/    1.12.1

Monitor and Manage Apache Geode
gfsh>
```

We have a script you can run from gfsh to start a Locator and Server in the simplest cluster:

```
cd spring-geode-workshop/configuration
gfsh
gfsh>run --file=configuration/src/main/resources/geode/bin/start-simple-cluster.gfsh
```

You are set to go.

With our gemfire-cluster create with an two Apache Geode Locators and (Cache) Servers running we can verify by listing and describing the members:

List and Describe Members.

```
gfsh>list members
   Name     | Id
----------- | ----------------------------------------------------
LocatorOne | 10.99.199.24(LocatorOne:68425:locator)<ec><v0>:1024
ServerOne  | 10.99.199.24(ServerOne:68434)<v1>:1025


gfsh>describe member --name=ServerOne
Name        : ServerOne
Id          : 10.99.199.24(ServerOne:68434)<v1>:1025
```

```
Host        : 10.99.199.24
Regions     :
PID         : 68434
Groups      :
Used Heap   : 27M
Max Heap    : 3641M
Working Dir : /Users/jblum/pivdev/lab/ServerOne
Log file    : /Users/jblum/pivdev/lab/ServerOne/ServerOne.log
Locators    : 10.99.199.24[10334]

Cache Server Information
Server Bind              : null
Server Port              : 40404
Running                  : true
Client Connections       : 0
```

What happens if we try to run the application now? You will find that because we use a default locator port to connect to that our @EenableClusterAware annotation runs through logic that detects we are now in client / server mode and will automatically connect us to the cluster, create the region and persist our data into that regions. If you have hundreds of application domain objects each requiring a Region for persistence SBDG will enable the creation of those regions in your behalf. It is not an unusual or unreasonable requirement in any practical enterprise scale application.

SBG through its integration with Spring Data Gemfire (SDG) is careful not to stomp on existing Regions since those Regions may have data already. Declaring the @EnableClusterConfiguration` annotation, which @EnableClusterAware automatically turns on for usm, is a useful development-time feature, but it is recommended that you explicitly define and declare your Regions in production environments, either using *Gfsh* or Spring confg. One of the additional benefits of @EnableClusterAware annotation as you see above is that it will push configuration metadata from the client to the server (or cluster). [Note: Cluster Configuration is outside the scope of our lab but ensures the meta data is consistent across all nodes]

@EnableClusterAware comes from the Spring Data Gemfire (SDG) project. SDG's `@EnableClusterAware` annotation makes it unnecessary to explicitly have to configure parameters like `clientRegionShortcut` on the SDG `@EnableEntityDefinedRegions` annotation (or similar annotation, e.g. SDG's `@EnableCachingDefinedRegions`). Finally, because the SBDG's`@EnableClusterAware` annotation is meta-annotated with SDG's `@EnableClusterConfiguration annotation` is automatically configured with the `useHttp` attribute to `true`.

Now, we can run our application again, and this time, you will see you region created on the server you just launched and can query for data. it just works!

Client/Server Run

```
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.9.RELEASE)

Saving Customer [Customer(name=Jon Doe)]
Querying for Customer [SELECT * FROM /Customers WHERE name LIKE '%Doe']
Customer was [Customer(name=Jon Doe)]

Process finished with exit code 0
```

In the cluster (server-side), we will also see that the "Customers" Region was created successfully:

List & Describe Regions.

```
gfsh>list regions
List of regions
---------------
Customers


gfsh>describe region --name=/Customers
..........................................................
Name            : Customers
Data Policy     : partition
Hosting Members : ServerOne
```

```
Non-Default Attributes Shared By Hosting Members

 Type  |    Name     | Value
------ | ----------- | ---------
Region | size        | 1
       | data-policy | PARTITION
```

We see that the "Customers" Region has a size of 1, containing "Jon Doe".

We can verify this by querying the "Customers" Region:

Query for all Customers.

```
gfsh>query --query="SELECT customer.name FROM /Customers customer"
Result : true
Limit  : 100
Rows   : 1

Result
-------
Jon Doe
```

That was easy!

Auto-configuration for Apache Geode, Take Two

What may not be apparent in this example up to this point is how the data got from the client to the server. Certainly, our client did send `Jon Doe` to the server, but our `Customer` class is not `java.io.Serializable`. So, how was an instance of `Customer` streamed and sent from the client to the server then (it is using a Socket)?

Any object sent over a network, between two Java processes, or streamed to/from disk, must be serializable.

As further evidence, we can adjust our query slightly:

Invalid Query.

```
gfsh>query --query="SELECT * FROM /Customers"
Message : Could not create an instance of a class example.app.crm.model.Customer
Result  : false
```

If you tried to perform a `get`, you would hit a similar error:

Region.get(key).

```
gfsh>get --region=/Customers --key=1 --key-class=java.lang.Long
Message : Could not create an instance of a class example.app.crm.model.Customer
Result  : false
```

So, how was the data sent then? How were we able to access the data stored in the server(s) on the cluster with the OQL query `SELECT customer.name FROM /Customers customer` as seen above?

Apache Geode and Tanzu GemFire provide 2 proprietary serialization formats in addition to *Java Serialization*: {apache-geode-docs}/developing/data_serialization/gemfire_data_serialization.html[Data Serialization] and {apache-geode-docs}/developing/data_serialization/gemfire_pdx_serialization.html[PDX], or *Portable Data Exchange*.

While *Data Serialization* is more efficient, PDX is more flexible (i.e. "portable"). PDX enables data to be queried in serialized form and is the format used to support both Java and Native Clients (C++, C#) simultaneously. Therefore, PDX is auto-configured in Spring Boot Data Geode (SBDG) by default.

This is convenient since you may not want to implement `java.io.Serializable` for all your application domain model types that you store in Apache Geode. In other cases, you may not even have control over the types referred to by your application domain model types to make them `Serializable`, such as when using a 3rd party library.

So, SBDG auto-configures PDX and uses Spring Data Geode's `MappingPdxSerializer` as the `PdxSerializer` to de/serialize all application domain model types.

If we were to disable PDX *auto-configuration*, you would see the effects of trying to serialize a non-serializable type, `Customer`. This would be apparent when you tried to query /Customers. We will leave this as an exercise for the student.

If you were not using SBDG, then you would need to enable PDX serialization explicitly.

The Spring Boot Data Geode/Gemfire PDX *auto-configuration* provided by SBDG is equivalent to:

Equivalent PDX Configuration with Spring Data Gemfire.

```
@SpringBootApplication
@ClientCacheApplication
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
@EnableClusterConfiguration(useHttp = true)
@EnablePdx
public class CustomerServiceApplication {
  // ...
}
```

In addition to the `@ClientCacheApplication` annotation, you would need to annotate the `CustomerServiceApplication` class with SDG's `@EnablePdx` annotation, which is responsible for configuring PDX serialization and registering SDG's `MappingPdxSerializer`.

Sample Applications.