**Tasks**

**Singleton Pattern**

For this project, use RestTemplate to make http requests to two external services to obtain the disease count. To implement the singleton pattern, you are required to create only a single instance of the resttemplate class.

So, in the project, head over to the RestServiceGenerator.java file in the config package. Inside the file, you should be able to see a class named RestServiceGenerator. Inside this class, you should be able to see a method named GetInstance.

Inside this method, you need to write the functionality that ensures that only one instance of the RestTemplate class is created. We have already provided you with the code that ensures that a restTemplate object is created. However, you need to ensure that the object is created only once by checking if the restTemplate object already exists or not. If the restTemplate object exists, you need to return the existing object, else you create a new one only once.

**Adapter Pattern**

When you have a 3-pin plug but a 2-pin socket, what do you do? You use an adapter!  
An adapter allows objects with incompatible interfaces to interact. In the design pattern world, the adapter design pattern has the same philosophy.

In the project, the user expects Indian disease count. The two data sources do not return the data in this format. So, an adapter (interface) is created and two separate classes implementing this interface are created that specify how the data is collected from the source.

The interfaces package contains a file named IndianDiseaseStat.java, which, in turn, contains an interface named IndianDiseaseStat that has a method declaration for GetActiveCount. Now, head over to the Strategies package. Implement this interface and create implementations for the GetActiveCount method.

In the DiseaseShStrategy.java file, write the logic to fetch the disease count from the Disease.io source. Moving ahead, create an implementation for the method that was declared in the IndianDiseaseStat interface. The method getDiseaseShResponseResponses returns an object that contains the country name, the total cases and today’s cases. The getCases() method might come in handy.

Well that was quite easy, right? The implementation of the GetActiveCount method for the JohnHopkinsStrategy class is a bit more tricky. So, head over to the JohnHopkinsStrategy class and think along the similar lines.

Remember, you are required to change the data that looks something like the following image to a single integer that is basically the sum of all the confirmed cases where the country value is ‘India’.

A screenshot of a computer code

Description automatically generated

**Factory Pattern**

In the factory pattern, you define an interface or an abstract class to create an object but let the subclasses decide which class to instantiate. In the project, you shall implement the capability of choosing from one of two available sources during runtime to obtain the disease count.

So, head over to the IndianDiseaseStatFactory.java in the Service package. Inside the IndianDiseaseStatFactory class, you are supposed to implement the GetInstance method. The method provides a pretty basic functionality of choosing a strategy that provides you with the disease count. There are two strategies, namely diseaseshstrategy, which uses disease.io as its source, and JohnHopkinsStrategy, which uses John Hopkins as its source.

So, create a method named GetInstance that takes SourceType type as method parameter and returns an IndianDiseaseStat (a type of strategy) object. Inside the method, depending upon the Sourcetype provided, return a disease strategy. The switch analogy in the adapter pattern might come in handy in some other ways.

**Facade Pattern**

In the project, the request to obtain the disease count is made from the controller without the controller worrying about any additional details.

Now, head over to the DiseaseCountFacade.java file in the Service package. Inside the DiseaseCountFacade class, create and implement two methods, namely getDiseaseShCount and getJohnHopkinCount, with ‘Object’ as their return type. Recall that you implemented the GetActiveCount and GetInstance while implementing the solution for the adapter pattern and the factory pattern. These methods might come in handy here.

Now, head over to the DiseaseController.java file in the controller package. Here, you see that the methods you just created (getDiseaseShCount and getJohnHopkinCount) are being called. Also, notice that the controller only calls these methods without having to worry about any underlying complexities.

**Chain of Responsibility Pattern**

Chain of responsibility pattern is a behavioural design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain. In the project, depending on the input, the type of authentication is decided. Depending upon whether user credentials or jwt tokens are provided, a specific type of authentication is used.

Now, head over to the Middleware package. You should see the following three files:

* AuthenticationProcessor.java
* BasicAuthProcessor.java
* JwtAuthProcessor.java

The AuthenticationProcessor.java file contains an abstract class named AuthenticationProcessor, which, in turn, contains an abstract method named isAuthorized that takes a parameter of type AuthenticationProvider and returns a boolean value.

The BasicAuthProcessor and JwtAuthProcessor classes contain the implementations for the isAuthorised method mentioned above. The isAuthorised method in the BasicAuthProcessor class checks if it can authenticate the user using basic user credentials. If it can, it calls the Authenticate method that contains the actual authentication logic, else it passes the handle to the next processor. Similarly, the isAuthorised method in the JwtAuthProcessor class checks if it can authenticate the user using jwt tokens. If it can, it calls the Authenticate method, else it passes the handle to the next processor.

The implementations for the Authenticate method can be found in BasicAuthProvider and the JwtAuthProvider classes in the Authentication package.

Now, head over to the Authenticator.java file in the Authentication package. Here, create a public static GetAuthProcessor method that returns an AuthenticationProcessor method. Inside this method, you are required to chain the above-mentioned authentication processors.

In the next segment, you will learn about the steps required to run the project.

**Project Execution Guidelines**

To run the project, follow the following steps:

1. Right-click on the project.
2. Click on ‘Run As’.
3. Select Maven build.
4. Mention the goal as ‘Spring-boot:run’.

Now, open your favourite browser.

* Enter the following URL to obtain the disease count using the DiseaseSh strategy.
  + http://localhost:8080/v1/disease/disease-sh-io/count

A screen shot of a computer

Description automatically generated

* Enter the following URL to obtain the disease count using the JohnHopkin
  + http://localhost:8080/v1/disease/john-hopkins/count

A screen shot of a computer

Description automatically generated

* Enter the following URL to obtain the infected ratio.
  + http://localhost:8080/v1/disease/infected-ratio?sourceType=DiseaseSh

A screen shot of a computer

Description automatically generated

In the next segment, you will understand the package structure of the project.

**Code Guide**

High level Code documentation

* **com.upgrad.patterns** -> DiseaseApplication.java
  + Contains the main class that starts the application
* **com.upgrad.patterns.Authentication** -> AuthenticationProvider.java
  + Contains an abstract class *AuthenticationProvider* and an abstract method *Authenticate*
* **com.upgrad.patterns.Authentication** -> Authenticator.java
  + Contains a class named *Authenticator*
  + Contains a method named *GetAuthProcessor*. The purpose of this method is to chain the authentication processors. First, the JWT processor is tried, and if it cannot be used, basic authentication (username and password) is tried.
  + Contains a method named *GetAuthProvider*
* **com.upgrad.patterns.Authentication**-> BasicAuthProvider.java
  + An implementation of the *AuthenticationProvider* class; implements the *Authenticate* method
  + Contains the implementation of basic authentication; authenticates the username and password provided
* **com.upgrad.patterns.Authentication** -> JwtAuthProvider.java
  + An implementation of the *AuthenticationProvider* class; implements the *Authenticate*method.
  + Contains implementation of the Jwt authentication; authenticates the jwt token
* **com.upgrad.patterns.config** -> RestServiceGenerator.java
  + Contains the implementation for the singleton pattern
  + *RestTemplate* object is instantiated only once and used throughout the application; *GetInstance* method contains the implementation for the same
* **com.upgrad.patterns.Constants**-> sourceType.java
  + Contains an enum named SourceType with 2 enumerators namely *DiseaseSh* and *JohnHopkins*
* **com.upgrad.patterns.Controllers**->DiseaseController.java
  + Contains the controller class *DiseaseController*
  + *GetDiseaseShCount* method returns the disease count using *Disease.io* as the source
  + *GetJohnHopkinsCount* method returns the disease count using John Hopkins as the source
  + *GetInfectedRatio* method returns the infected ratio
* **com.upgrad.patterns.Entity** -> DiseaseShResponse.java
  + Contains the class named *DiseaseShResponse*
  + JSON property names are mapped to the java field’s name
* **com.upgrad.patterns.Entity** -> JohnHopkinResponse.java
  + Contains the class named *JohnHopkinResponse*
  + JSON property names are mapped to the java field’s name
* **com.upgrad.patterns.Entity** -> stat.java
  + Contains the class named *Stat*
  + JSON property names are mapped to the java field’s name
* **com.upgrad.patterns.filters** -> AuthFilter.java
  + <visit again>
* **com.upgrad.patterns.interfaces** -> IndianDiseaseStat.java
  + Contains the *IndianDiseaseStat* interface that contains the declaration of a method named *GetActiveCount*
* **com.upgrad.patterns.Middleware** -> AuthenticationProcessor.java
  + Contains the abstract class named *AuthenticationProcessor* along with an abstract method named*isAuthorised*
* **com.upgrad.patterns.Middleware**-> BasicProcessor.java
  + Contains the implementation of the abstract method named *isAuthorised*; if username and password are provided, use them to authenticate, else use the next processor.
* **com.upgrad.patterns.Middleware** -> JwtAuthProcessor.java
  + Contains the implementation of the abstract method named*isAuthorised*; if JWT tokens are provided, use them to authenticate, else use the next processor.
* **com.upgrad.patterns.Service** -> DiseaseCountFacade.java
  + Contains the DiseaseCountFacade class that has implementations of two methods, namely *getJohnHopkinsCount*and *getDieseaseShCount*
  + Contains the implementation for the facade pattern. The *getJohnHopkinsCount* and the *getDieseaseShCount*functions of the *DiseaseCountFacade* class encapsulate the complexities of fetching from each source. These can be called from the controller without the controller worrying about any additional details.
* **com.upgrad.patterns.Service**-> IndianDiseaseStatFactory.java
  + Contains the *IndianDiseaseFactory* class that contains the implementation for *GetInstance* method
  + Contains the implementation for the factory pattern. The *GetInstance* method allows to fetch the strategy to obtain active case count from the source type mentioned in the request parameter. You can use the functionality in runtime based on the strategy mentioned by the user.
* **com.upgrad.patterns.Strategies**-> DiseaseShStrategy.java & JohnHopkinsStrategy.java
  + You create separate classes for each source and implement the *IndianDiseaseNumbers* interface. In the *GetActiveCount()*, write the implementations that uniquely describe how to fetch data from each source. Hence, you wrap around different sources to fetch the required data from different sources.