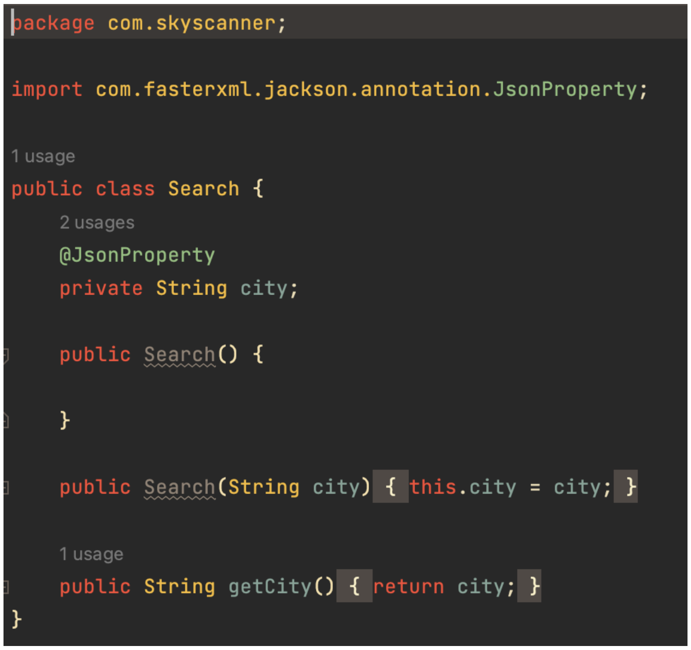
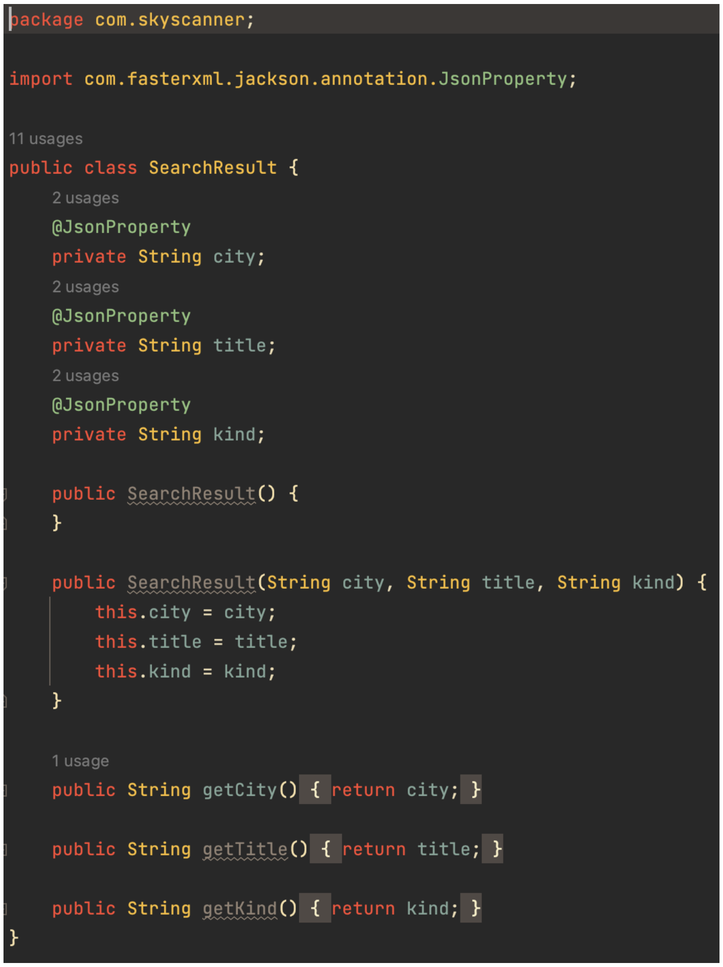
Task Instructions

1. First, you'll need to set up your local development environment.
    a. While you can technically use any Java IDE to complete this task, you'll need to get ahold of IntelliJ to follow along with the instructions. Go ahead and install it now if you haven't already.
    b. Next, use Git to **fork** and **clone** the starter repo. This repository contains some Dropwizard scaffolding, as well as the car rental and hotel lists. If you are unfamiliar with Git, read through the first **two chapters** of the git book.
        i. If you're interested in using Dropwizard for future projects, a similar scaffold can be generated (quite easily) using the Dropwizard Maven Archetype.
    c. Use IntelliJ to **open** the project, then give the IDE a moment to finish getting acquainted with the new repo.
    d. A pop-up in the bottom right-hand corner will notify you that IntelliJ has found some **Maven build scripts**. Click **Load Maven Project** to run through some automated initialisation steps.
    e. You may need to install a Java Software Development Kit (**JDK**), used to compile and run Java programmes. If you receive an error, follow the steps provided by your IDE or consult the documentation. This project uses **OpenJDK 19**.
    f. Once your IDE is finished loading the project, it's time to see if everything works! Click the **green arrow** in the upper right-hand corner to **run** the project. You should receive a collection of difficult-to-understand log messages (don't worry about these) and a banner in the logs that says **Welcome to Hoen Scanner!**
    g. Congrats, you're finally done setting up the project!
2. We have access to two files in the resources folder of the application: **rental_cars.json** and **hotels.json**. These contain the search results we'd like to surface to users – take a look at both files to get a feel for how this data is formatted.
3. We need some way to represent the data moving through our application, namely, the searches users submit and the results we return. Naturally, since this is a Java application, we'll represent both with classes. Users are going to submit searches to the microservice using **JSON-encoded POST** requests, so we need a way to deserialise the data in these requests to Java objects. In a similar fashion, we'd like to return **JSON**-encoded responses with search results, so we'll need to be able to **serialise** result objects. To accomplish this, we'll be using the **Jackson** library to annotate instance variables with **@JsonProperty**, which will take care of serialisation/deserialisation for us automagically.
    a. Create a **Search** class with a single serialisable field: **city**. Use the following snippet as a template:



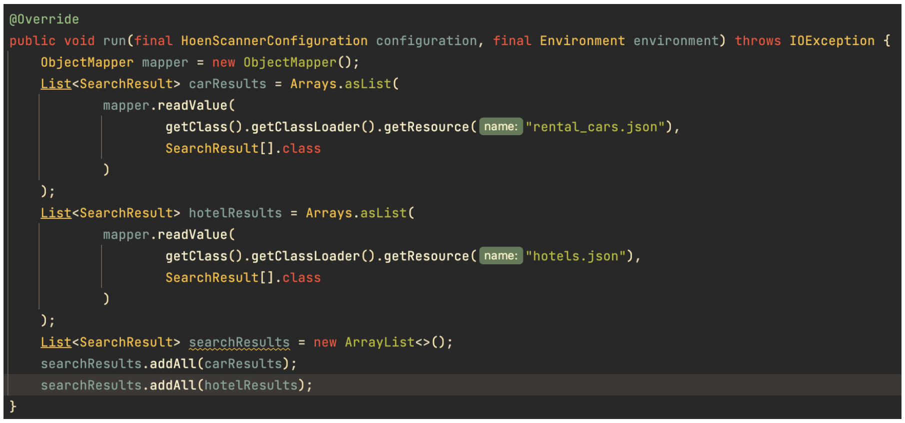
```
package com.skyscanner;

import com.fasterxml.jackson.annotation.JsonProperty;

1 usage
public class Search {
    2 usages
    @JsonProperty
    private String city;

    public Search() {

    }

    public Search(String city) { this.city = city; }

    1 usage
    public String getCity() { return city; }
}
```

b. Create a **SearchResult** class with three serialisable fields: **city**, **kind** and **title** (to match the format of the search result files). Use the following snippet as a template:



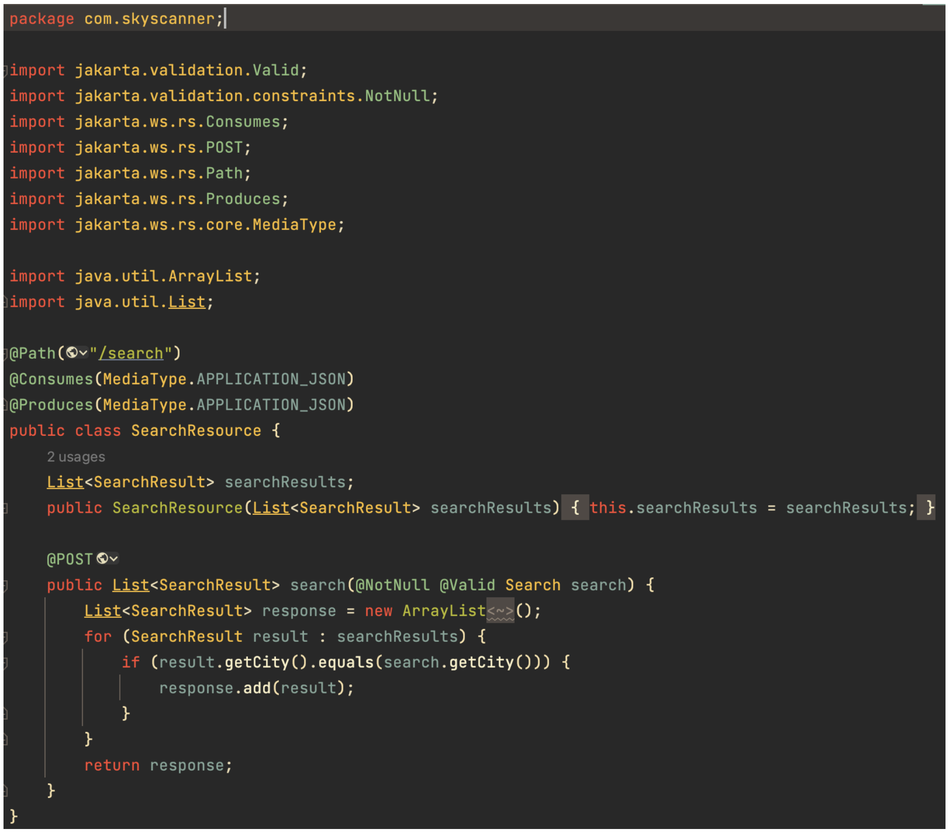
```java
package com.skyscanner;

import com.fasterxml.jackson.annotation.JsonProperty;

11 usages
public class SearchResult {
    2 usages
    @JsonProperty
    private String city;
    2 usages
    @JsonProperty
    private String title;
    2 usages
    @JsonProperty
    private String kind;

    public SearchResult() {
    }

    public SearchResult(String city, String title, String kind) {
        this.city = city;
        this.title = title;
        this.kind = kind;
    }

    1 usage
    public String getCity() { return city; }

    public String getTitle() { return title; }

    public String getKind() { return kind; }
}
```

4. The application needs to load both JSON files into a single list of search results that can be returned to users. We'll handle this in the **run** method of the **HoenScannerApplication** class, prior to registering any **resources** (we'll cover resources in a moment). Modify the run method to match the following code snippet:



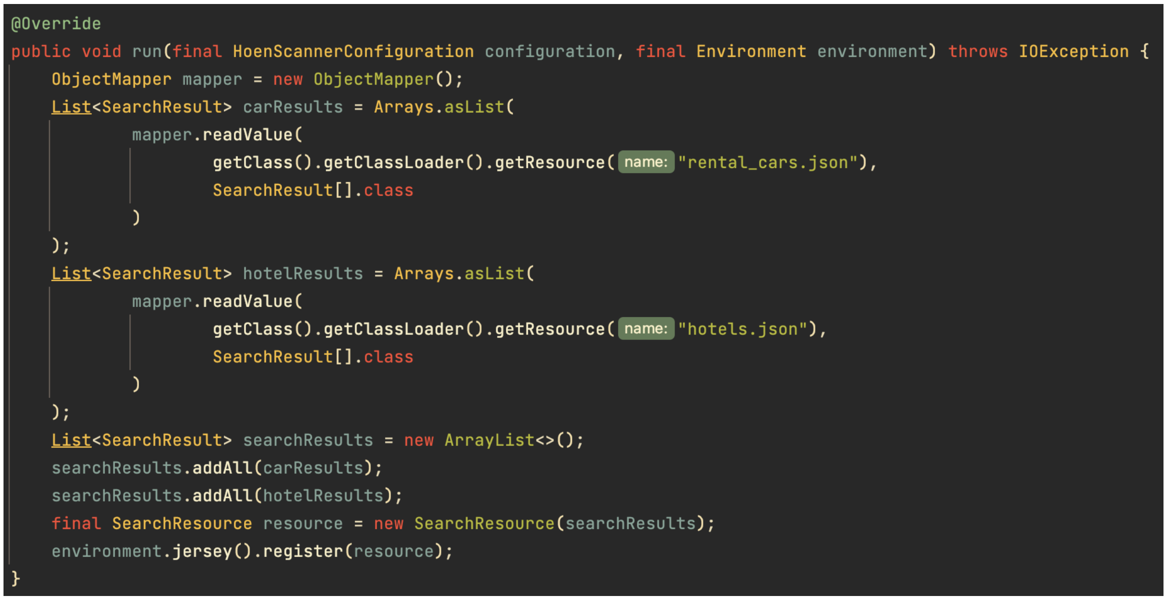
```java
@Override
public void run(final HoenScannerConfiguration configuration, final Environment environment) throws IOException {
    ObjectMapper mapper = new ObjectMapper();
    List<SearchResult> carResults = Arrays.asList(
            mapper.readValue(
                    getClass().getClassLoader().getResource( name: "rental_cars.json"),
                    SearchResult[].class
            )
    );
    List<SearchResult> hotelResults = Arrays.asList(
            mapper.readValue(
                    getClass().getClassLoader().getResource( name: "hotels.json"),
                    SearchResult[].class
            )
    );
    List<SearchResult> searchResults = new ArrayList<>();
    searchResults.addAll(carResults);
    searchResults.addAll(hotelResults);
}
```

5. Dropwizard defines endpoints in the form of **Resources**. A Dropwizard resource specifies a **@Path**, which determines the URL where the resource can be reached; a **@Consumes**, which determines the type of requests it will handle; and a **@Produces**, which determines the type of responses it will handle. Request methods are determined by the annotation on a given method, such as **@Post**. We need to create a method that accepts **search** objects, filters the **searchResults** list we created in the last step and returns the trimmed results. Create a new **SearchResource** class using the following snippet as a guide:



```java
package com.skyscanner;

import jakarta.validation.Valid;
import jakarta.validation.constraints.NotNull;
import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

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

@Path("/search")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class SearchResource {
    2 usages
    List<SearchResult> searchResults;
    public SearchResource(List<SearchResult> searchResults) { this.searchResults = searchResults; }

    @POST
    public List<SearchResult> search(@NotNull @Valid Search search) {
        List<SearchResult> response = new ArrayList<~>();
        for (SearchResult result : searchResults) {
            if (result.getCity().equals(search.getCity())) {
                response.add(result);
            }
        }
        return response;
    }
}
```

6. Modify the **run** method to register our new resource. This will expose the **/search** endpoint on the microservice. Modify the run method to match the following code snippet:



```java
@Override
public void run(final HoenScannerConfiguration configuration, final Environment environment) throws IOException {
    ObjectMapper mapper = new ObjectMapper();
    List<SearchResult> carResults = Arrays.asList(
            mapper.readValue(
                    getClass().getClassLoader().getResource( name: "rental_cars.json"),
                    SearchResult[].class
            )
    );
    List<SearchResult> hotelResults = Arrays.asList(
            mapper.readValue(
                    getClass().getClassLoader().getResource( name: "hotels.json"),
                    SearchResult[].class
            )
    );
    List<SearchResult> searchResults = new ArrayList<>();
    searchResults.addAll(carResults);
    searchResults.addAll(hotelResults);
    final SearchResource resource = new SearchResource(searchResults);
    environment.jersey().register(resource);
}
```

7. Click the **run** button again to reload your microservice.
8. At this point, the application should be fully functional. All we need to do now is test it!
    a. Download Postman, a polished GUI for sending HTTP requests and testing Rest APIs.
    b. Open it up after installation and follow the prompts to create an account or click **skip and go to the app**.
    c. Under the **Get Started** sidebar on the right, click **Create a Request**.
    d. In the request method section, change **GET** to **POST**.
    e. In the **Enter Request URL** input box, add **localhost:8080/search**.
    f. Select the **Body** tab of the request.
    g. Select the **Raw** radio button.
    h. Select **JSON** from the formatting dropdown.
    i. Enter the following request body to search for results in the city of Petalborough: **{"city": "petalborough"}**
    j. Click **Send**.
    k. The above steps will submit a **POST** request to our microservice with a JSON body representing a search for all rental cars and hotels in the city of Petalborough. You should see a list of these returned in the response section. Congrats, your application is up and running! Try the cities of **Rustburg** and **Shaleport** (lowercase), as well as some invalid searches to make sure everything works as expected.
9. All that's left now is to submit your work, commit and push your changes and pass along a URL to your repo below.