# ConnectedCitiesAppV2

**Refactoring ConnectedCitiesApp - Spring Boot REST API to find connected cities**

**Whats new: Factory Design Pattern, DisjointUnionSets**

## Code Challenge:

You’re asked to write a program in Java which determines if two cities are connected.

Two cities are considered connected if there’s a series of roads that can be traveled from one city to another.

List of roads is available in a file.

File contains a list of city pairs (one pair per line, comma separated), which indicates that there’s a road between those cities.

It will be deployed as a spring-boot app and expose one endpoint:

<http://localhost:8080/connected?origin=city1&destination=city2>

Your program should respond with ‘yes’ if city1 is connected to city2, ’no’ if city1 is not connected to city2.

Any unexpected input should result in a ’no’ response.

*Example:*

city.txt content is:

Boston, New York

Philadelphia, Newark

Newark, Boston

Trenton, Albany

Assumption: There is only one possible path

<http://localhost:8080/connected?origin=Boston&destination=Newark>

Should return yes

<http://localhost:8080/connected?origin=Boston&destination>=Philadelphia

Should return yes

<http://localhost:8080/connected?origin=Philadelphia&destination=Albany>

Should return no

## How to execute and test this app?

Commands to deploy the app:

1. Go the folder where this application is saved and change to the ConnectedCitiesApp directory as the current directory

Example:

cd /…………………/ConnectedCitiesApp

1. You can build the app using the following command:

mvn clean package

1. Run the app using the following command:

./mvnw spring-boot:run

Testing:

|  |  |
| --- | --- |
| url | Result |
| <http://localhost:8080/connected?origin=Boston&destination=Newark> | yes |  |
| <http://localhost:8080/connected?origin=Boston&destination>=Philadelphia | Yes |  |
| http://localhost:8080/connected?origin=Philadelphia&destination=Albany | no |  |

## Testing using swagger:

When the app is running (using the previous commands)

Go to the following url to visit the api docs:

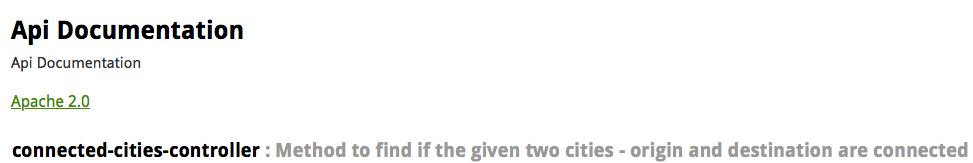
[*http://localhost:8080/v2/api-docs*](http://localhost:8080/v2/api-docs)

To test the app using the swagger ui:

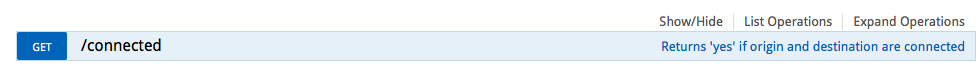
1. Go to the following url:

[*http://localhost:8080/swagger-ui.html*](http://localhost:8080/swagger-ui.html)

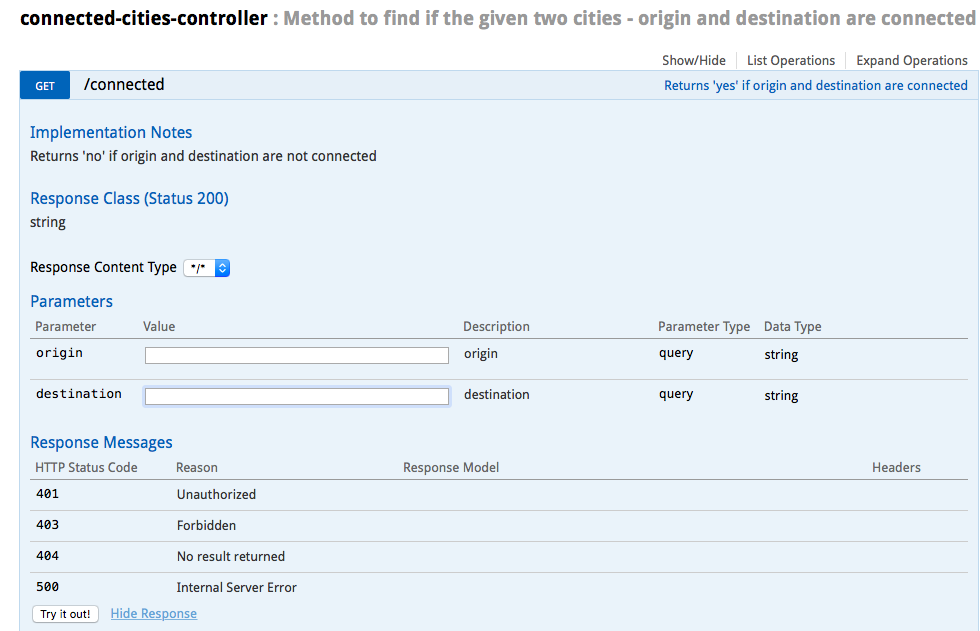
1. You will get the below screen:



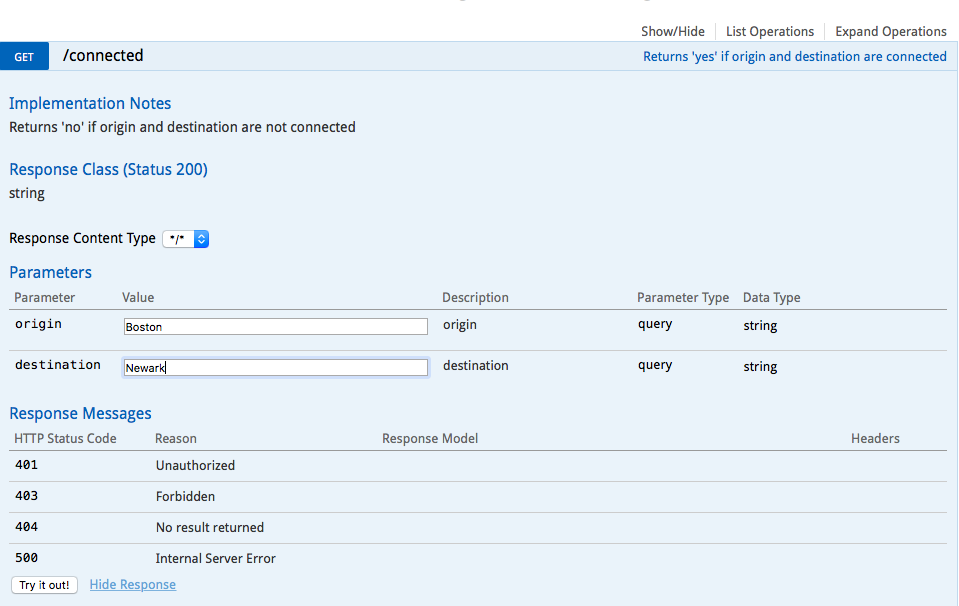
1. Click on the connected-cities-controller and get the below screen:



1. Click on the /connected (Alternatively, you can also click on Expand Operations) and get the below screen:



1. Enter different values for the parameters ‘origin’ and ‘destination’ and click on the ‘Try it out!’ button to observe the results.





## Technical Specifications:

Technology Stack:

Maven, Spring boot, Java8, Junit5, Swagger2, Factory Design Pattern

Graph Representation:

The graph is represented using adjacency list. An adjacency list is an array of separate lists. Each element of the array is a list, which contains all the vertices that are adjacent to vertex i. Size of array will be number of vertices in graph.

Example:

Input:

Boston, New York

Philadelphia, Newark

Newark, Boston

Trenton, Albany

The following example gives the representation of the cities (as given in the input) in the graph

Cities and their ids:

Boston:0

Newark:1

Albany:2

Trenton:3

New York:4

Philadelphia:5

Graph representation: Adjacency List showing the adjacent(connected) cities for each city

Adjacency list of vertex 0

head -> 4 -> 1

Adjacency list of vertex 1

head -> 5 -> 0

Adjacency list of vertex 2

head -> 3

Adjacency list of vertex 3

head -> 2

Adjacency list of vertex 4

head -> 0

Adjacency list of vertex 5

head -> 1

Graph Search Algorithm:

The following graph search algorithms are implemented:

1. Depth First Search (DFS)
2. Breadth First Search (BFS)
3. DisjointSets

The choice of the algorithm depends on the value of the connectedcities.graphsearch.algorithm property in the application.properties file.

## Design Considerations:

1. Advantages of Spring boot:

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes

If there are beans/components in other packages that are not sub-packages of the main package, you should manually add them as @ComponentScan

In a non-Spring Boot Project, we would typically define the component scan explicitly in an XML application context or a Java Application Context

SpringBoot can autodetect the “application.properties” file in the “src/main/resources” directory and any loaded properties can be injected easily. There is no need to tell the Spring where the property file is located or even explicitly register a ‘PropertySource’.

1. Advantages of using @GetMapping over @RequestMapping at the method level:

At the handler methods level, It is best to use the more specific @GetMapping than the lengthier @RequestMapping(method=RequestMethod.GET) because with @RequestMapping, there is a possibility that the method attribute will be left off, as it is not mandatory.

But it is always best practice to be specific about the HTTP method that will be handled and @GetMapping is specific for the GET requests.

1. Advantages of using Adjacency List over Adjacency Matrix:

The graph is represented as an Adjacency List. Adjacency list uses less memory when to compared to Adjacency matrix, there is a lot of wastage of memory especially if the matrix is sparse.

It is faster to iterate over all the edges in the Adjacency matrix as the adjacent vertices can be accessed directly.

And it is faster to add/delete a vertex.

Generally, Adjacency matrix representation is well suited when the graph is expected to be dense and Adjacency lists representation is well suited when the graph is expected to be sparse.

1. In the City class, the instance variable Id is an Integer type and not an int. Because it is possible to check for null by using an Integer type.
2. The city pairs from the input file is stored in a set to avoid duplicate entries.
3. Enum Comparison. Enum members are compared using == operator rather than equals() methos. This is because == operator never throws a NullPointerException and is subject to type compatibility check at compile time and is faster than equals() method.
4. Factory design pattern is used here to promote loose coupling, high cohesion and extensibility by extracting the responsibility of creating a new instance and handling it only in the factory class, it avoids repeatability of certain code. Since we are dealing with conditional instantiation of the graph search algorithms, Factory design pattern is a good choice.

Factory design pattern can be implemented in just two concise steps:

RouteFinderFactory routeFactory = **new** RouteFinderFactory(vertices, cityPairs);

IConnectedCitiesFinder finder = routeFactory.getFinderAlgorithm(GraphSearchAlgorithm.valueOf(algorithm));

**public** IConnectedCitiesFinder getFinderAlgorithm(GraphSearchAlgorithm algorithm) {

**if** (algorithm == **null**) {

**return null**;  // check for null before switch statement

} **else** {

**switch** (algorithm) {

**case** DFS:

    {

    initGraph();

**return new** DFS(graph);

    }

**case** BFS:

{

initGraph();

**return new** BFS(graph);

}

**case** DisjointSets:

{

initDisjointSets();

**return** disjointsets;

}

**default**:

**return null**;

}

}

}

1. Use the factory to get an object of concrete algorithm class by passing the information of algorithm to use. (See the code above)
2. Get an object of the concrete class and call its isConnected(srcCityId, destCityId) method to find if two cities are connected (using the passed in algorithm)

finder.isConnected(srcCityId, destCityId);

1. The package structure of the project is designed to reduce coupling across packages, avoid circular dependencies and support high cohesion.
2. Switch statement is used in the place of if-else statement in the RouteFinderFactory.getFinderAlgorithm() method for the following reasons:

* Switch is better for multiway branching: In the case of switch statement, the compiler is aware that the case statements are all same type and simply must be compared for equality with the switch expression, while in case of if expressions, the compiler has no such knowledge. This is because when compiler compiles a switch statement, it will inspect each of the case constants and create a “jump table” that it will use for selecting the path of execution depending on the value of the expression. Therefore, if we need to select among a large group of values, a switch statement will run much faster than the equivalent logic coded using a sequence of if-elses.
* Speed: If a switch contains more than five items, it’s implemented using a lookup table or a hash list. This means that all items get the same access time, compared to a list of if:s where the last item takes much more time to reach as it has to evaluate every previous condition first.

Here we are having less than 5 case values where there is not much difference in speed between the switch statement and the if-else statement but we have preferred the use of switch statement for the purpose of extensibility

* Better Readability.

1. We use if statement to check for null before entering the switch statement because switch(i) would throw a NullPointerException if I is null.