## Weather App Documentation:

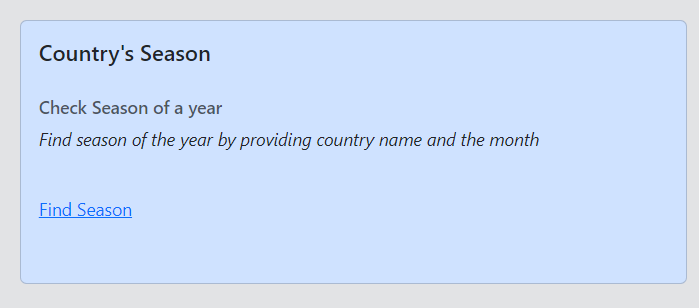
**Introduction:**

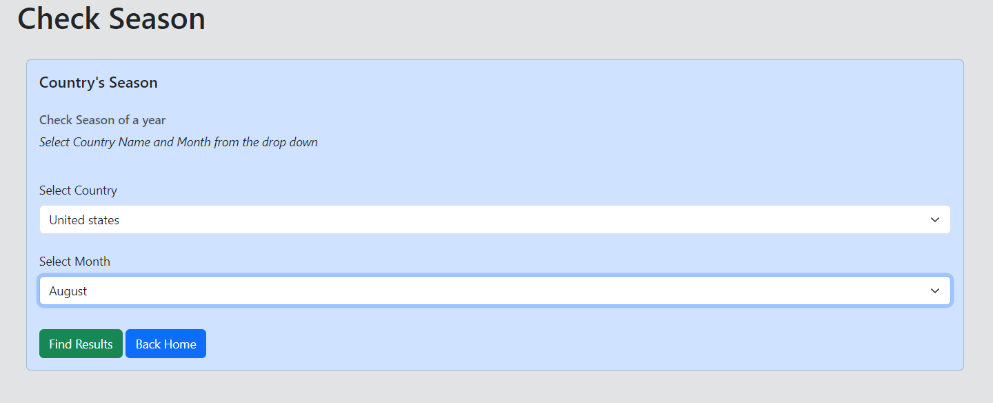
Weather Application is a web based application, build on top of Java EE. The application is accessible from web browser as long as your Web or Application server such as Apache Tomcat or Glass Fish is running in the background.

This app has two major goals.

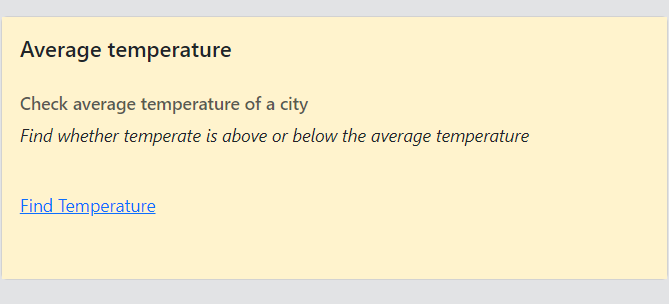
1. Find Season of a Country.
2. Find City’s temperature difference from its average.

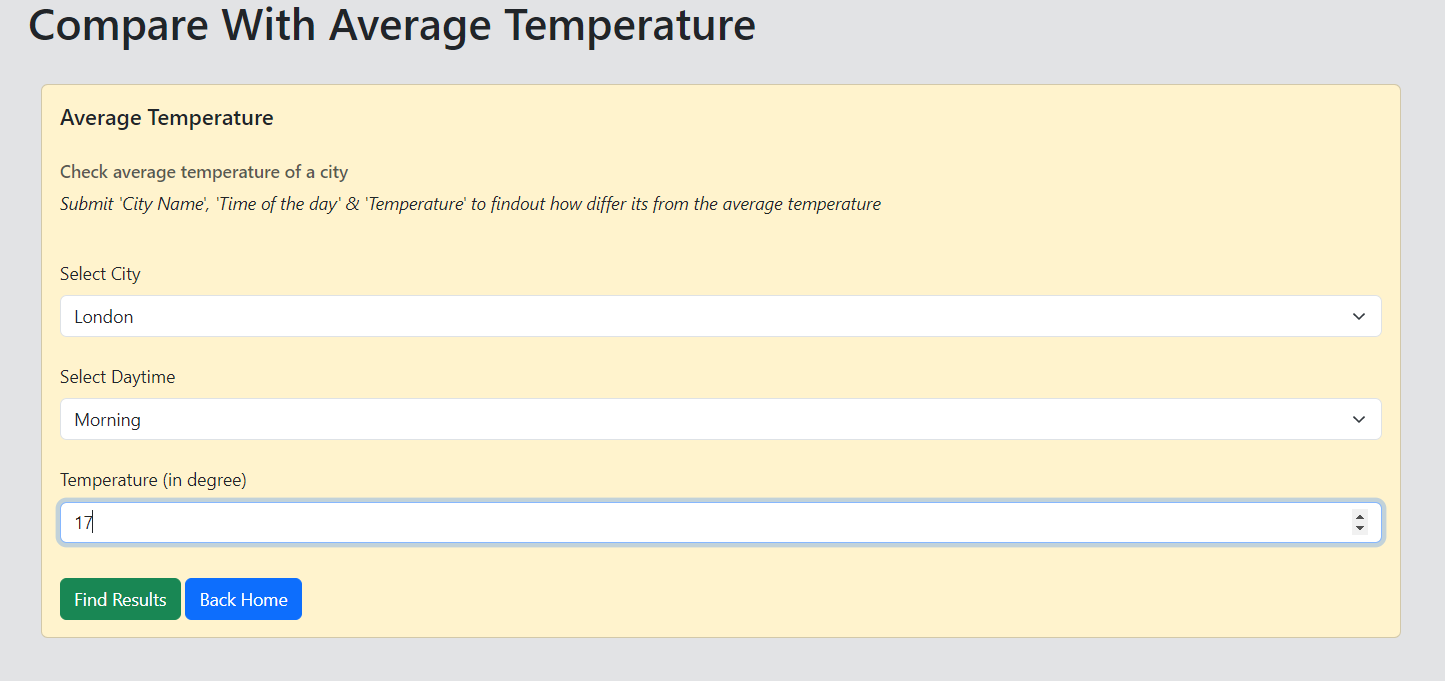
* Season of Country

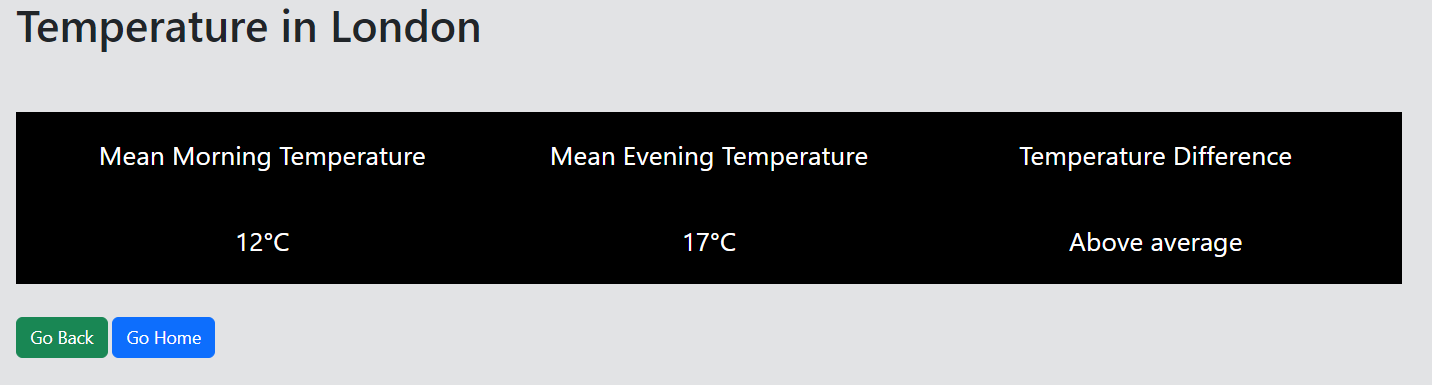




* Find temperature difference







Following are the technologies we have use in this project:

* Java 17
* Apache Tomcat 9.0
* JSP
* Servlets

**Module descriptions:**

The Weather App can be split down further into number of small modules, but for the sake of simplicity and better understanding I've focused on following main modules as shown below.

1. **Data**

**Description**

* + *Name*: Weather Data
  + *Objective*: As the name suggest this module is responsible for holding and providing the data, required by the entire web application. Since at this stage our web application aims to serve only two purposes i.e. to find a country's season and to find deviation with an average temperature of a city. Therefore this module currently responsible to provide us SeasonData and CitiesData.
  + *Behave*: For the sake of simplicity, I've added a dummy data inside this module. Therefore it’s not interacting with any external system (i.e. file or database).
  + *Input*: As mentioned, module has hard-coded data values. Hence it does not take data as an input.
  + *Output*: Other module can access it, by calling its public function which return data value.

1. **Service**

**Description**

* *Name*: Weather App Service
* *Objective*: This module is like an engine of the entire application. This is where the actual implementation exists. It mainly serve two purposes. Finding a country's season and Find temperature difference of a city with its average temperature.
* *Behave*: Module interact with Data module (described above) to perform its action against the input data such as finding a city or a country.
* *Input*: This module takes input as an argument.
* *Output*: After processing, both sub-modules (find\_season, find\_temp) returns the output as string value.

1. **Controller**

**Description**

* *Name*: Weather App Servlet
* *Objective*: Module enhance app's basic functionality. It is in-charge of managing operations and responding to user requests that originate from HTML pages or View Modules. Internally, it transmits data from the View Module to the Service Module and then correctly updates the View.
* *Behave*:

Below is a typical behavior it follows:

User =>

Web Browser (View Module) =>

Server (Controller Module) =>

Processing (Service) =>

Return View (View Module)

* *Input*: Takes input as part of request parameter
* *Output*: Redirect to the View Module.

1. **View**

**Description**

* *Name*: Weather App View
* *Objective*: It is responsible to serve web pages. It allow users to interact with a web interface. Users can select UI elements using cursor to run the application. Moreover, it uses visuals (UI) for an interactive user experience throughout.
* *Behave*: User of this application comes in contact with the module via browser. It further communicate/pass user input to the Controller for processing.
* *Input*: User interact with HTML or JSP elements i.e. (text, drop-down and buttons) through keyboard and mouse.
* *Output*: Return output to the screen as HTML or JSP page.

**Modularity**

Since it’s a web based application. It needs to be deployed at application server or servlet container such as Apache Tomcat in order to function correctly. IntelliJ IDEA comes with a built-in Tomcat’s support, upon execution, it auto deploy the WAR (Web Archive) file into the tomcat’s deploy folder.

One can opt manual approach, here are following steps:

* Create a WAR file
  + Go inside the project directory of your project (outside the WEB-INF), then write the following command:

jar -cvf projectname.war \*

* If Tomcat is running, stop/kill it.
* Go to the tomcat installation folder (this must be C:\Documents and Settings\tomcat9x for you), let's call it <tomcat>.
* In <tomcat>, delete the temp and work folders. They only contain temporary files.
* If it's a jar file maybe is for configuration, so drop it in <tomcat>/lib folder. If it's a war file, drop it in <tomcat>/deploy or in <tomcat>/webapps folder.
* Start your tomcat.

**How good Modularity:**

The Weather App is well organized. Code is written using Java programming language. Application is divide into various packages, where each package serves a specific purpose. Following are the packages in this app:

* Data: Hold/Server data to the app.
* DTO: Data Structure.
* Service: Business logic for find\_temp and find\_season.
* Servlet/Controller: communicate between Service and Web Module.
* Web/HTML pages: HTML page and views are defined here.
* Test: It contains all the test cases.

Here is the Application's directory in tree structure.

──src

├───main

│ ├───java

│ │ └───com

│ │ └───learning

│ │ └───weatherapp

│ │ ├───data

│ │ ├───dto

│ │ ├───service

│ │ └───servlet

│ ├───resources

│ └───webapp

│ ├───images

│ └───WEB-INF

└───test

├───java

│ └───com

│ └───learning

│ └───weatherapp

└───resources

**Review Checklist**

* Read all the requirements and understand them well.
* Prepare a design (enough to meet the application's basic goals).
* Think about what modules you may need in this application.
* Document them all.
* Brainstorm to identify suitable branches names for your version control.
* Start writing code.
* Initialize a Git directory and add the basic app to branch named 'basic-version'
* Enhance your code to make it more user friendly and add the updated code to a new branch named 'complete-version'
* Write test cases to see if everything works or behave as intended.
  + At this stage you can perform rigorous testing on your application to see how well it perform.
  + Test with valid and invalid inputs.
* If you find any ambiguity fix those issues and repeat the tests again.
* If everything works fine, Deploy the app.

**Black-box test cases**

A black box test can simulate user activity and see if the system delivers on its promises. In this application we have covered following:

Equivalence partitioning

Boundary value analysis

* **EQUIVALENCE PARTITIONING**

In this type of testing we divide the input domain into classes of data (valid and invalid range). Here we will take two invalid and one valid equivalence classes.

Assumption: We are providing a valid country name

|  |  |  |
| --- | --- | --- |
|  | **Equivalence Partitioning (Season)** |  |
| Invalid | Valid | Invalid |
| month < 1 | 1-12 | month > 12 |

|  |  |  |
| --- | --- | --- |
| **Equivalence Partitioning (Temperature)** | | |
| Invalid | Valid | Invalid |
| temperature < -20 | -20 – 60 | temperature > 60 |

* **BOUNDARY VALUE.**

|  |  |  |
| --- | --- | --- |
| **Boundary Value Analysis (Season)** | | |
| Invalid | Valid | Invalid |
| month = 0 | 1 – 12 | month = 13 |

|  |  |  |
| --- | --- | --- |
| **Boundary Value Analysis (Temperature)** | | |
| Invalid | Valid | Invalid |
| temperature = -21 | -20 – 60 | Temperature = 61 |

* **White-box test cases**

One can test the quality or internal design of code. Let's say we perform white box test to ensure system return the correct records that we have for demonstration purpose we will check the season data size in our system.

**Test implementation and execution**

Black Box Test Implementation

*Equivalence Partitioning:* Below program demonstrate the equivalence partitioning

**INVALID**: month value below 1  
Assertions.*assertEquals*("month\_invalid",seasonService.find\_season("Pakistan",-5));

**VALID**: month value between 1-12 Assertions.assertNotEquals("month\_invalid",seasonService.find\_season("Pakistan",7));

*Boundary value analysis:* Below program demonstrate the equivalence partitioning

**INVALID**: month value above 12 Assertions.assertEquals("month\_invalid",seasonService.find\_season("Pakistan",14));

**INVALID**: month shouldn’t exceed 12 or can't be below 1 Assertions.assertEquals("month\_invalid",seasonService.find\_season("Pakistan",13));

Assertions.assertEquals("month\_invalid",seasonService.find\_season("Pakistan",0));

**VALID**: month should be between 1 and 12 Assertions.assertNotEquals("month\_invalid",seasonService.find\_season("Pakistan", 1));

Assertions.assertNotEquals("month\_invalid",seasonService.find\_season("Pakistan", 12));

**White Box Test case**

final int SEASONS\_DATA\_SIZE = 4;

Assertions.assertEquals(SEASONS\_DATA\_SIZE, seasonService.getSeasonsInfo().size());

**Version control**

Application has following three branches

* basic-version (hold all the fundamental service)
* complete-version (improved version. Better user experience through JSP/HTML pages)
* main (production-ready-version, contains test case, documentation etc.)

**Ethics**

Dependence on the system could have negative effects if the system is fed with inaccurate data. As a result, it is advised to conduct a little additional investigation before accepting the results it produced.