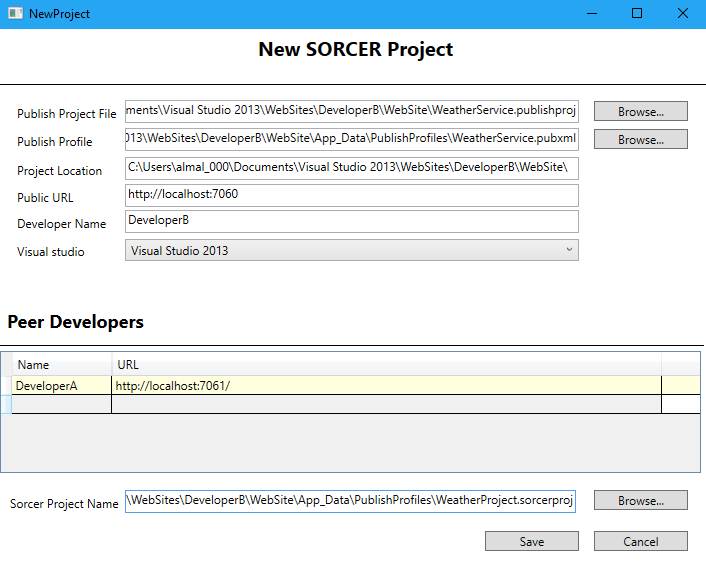
1. **Using SORC**

SORCER provides an implementation of SORC that can be used to support the distributed development of web service based applications. This section presents a case study to show how SORCER can be used to create a service based web application and, in doing so, how SORC is used to support such distributed development.

## 8.1 Example Scenario

The goal of this work is to implement the SORC system and be able to conduct concurrent development. This section shows the creation of a weather application using SORCER to illustrate the development steps. The weather application is implemented by two developers: DeveloperA builds the web application to show the weather forecast sourced from the weather web service provided by DeveloperB. In the beginning, both developers publish their projects to Microsoft IIS. Once they publish their projects they get two files, .pubxml and website.publishproj. These are the main files required to create project. The following screen shows the first steps in the development of the weather application using SORCER to create the project.

****

Each field of the main window for creating a new SORCER project is described below:

**Publish Project File:** an XML based file that contains a list of properties and

functionalities (a .publishproj file)

**Publish Profile:**  name of the publish profile (a .pubxml file)

**Project Location:** the location of the project in the developer’s workspace

**Public URL:** the IP address of the developer creating the project

**Developer Name:** the name of the developer creating the project

**Visual Studio Version:** whatversion of Visual Studio .Net the project uses

(it supports 2012 and 2013)

**Peer Developer:** List of developer URLs (IP addresses) involved in the project team

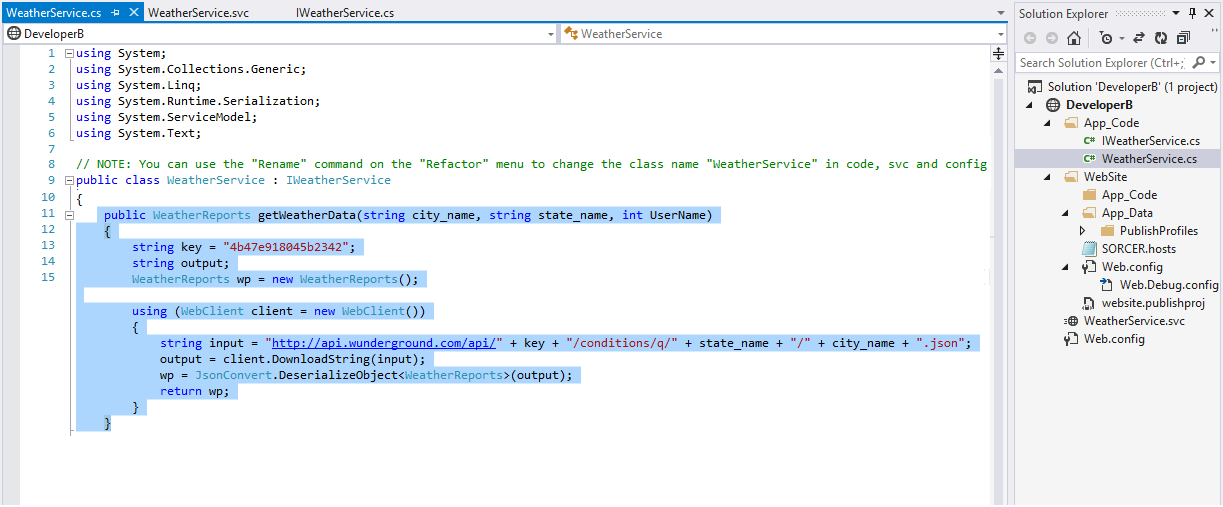
**Sorcer Project Name:** The name of the project being developed.

Project settings are stored in the developer’s workspace in a file with extension .sorcerproj.

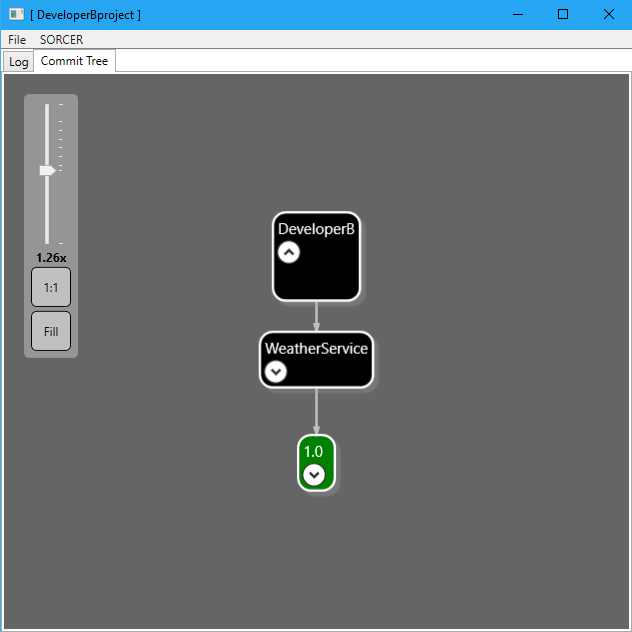
Once the project team has been created, concurrent development implements the weather application scenario described above. First, we need each developer undertakes the following steps.

* **DeveloperB:**

1. Create the WeatherService that has a method getWeatherData. This method returns the weather conditions of a certain city.

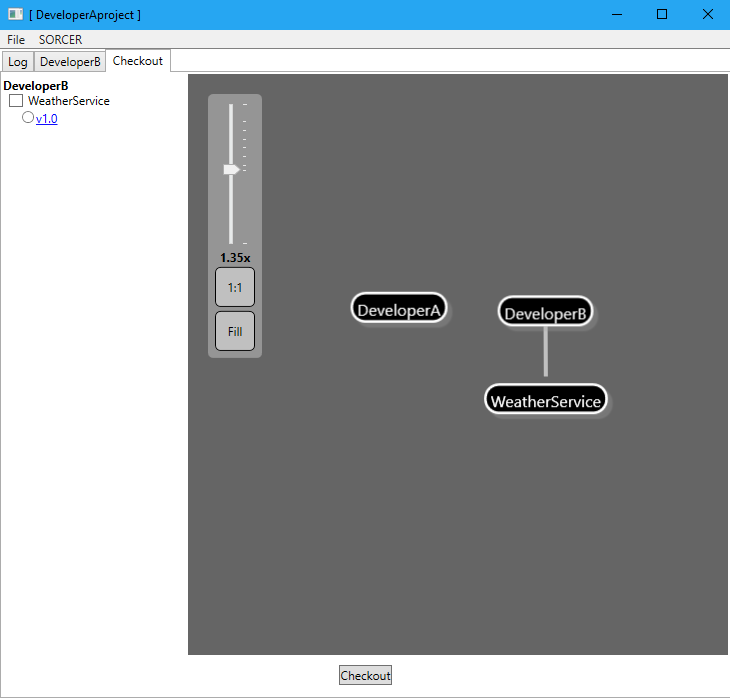


1. Commit the service (the first version is named WeatherService\_1\_0) so that it is available to DeveloperA.

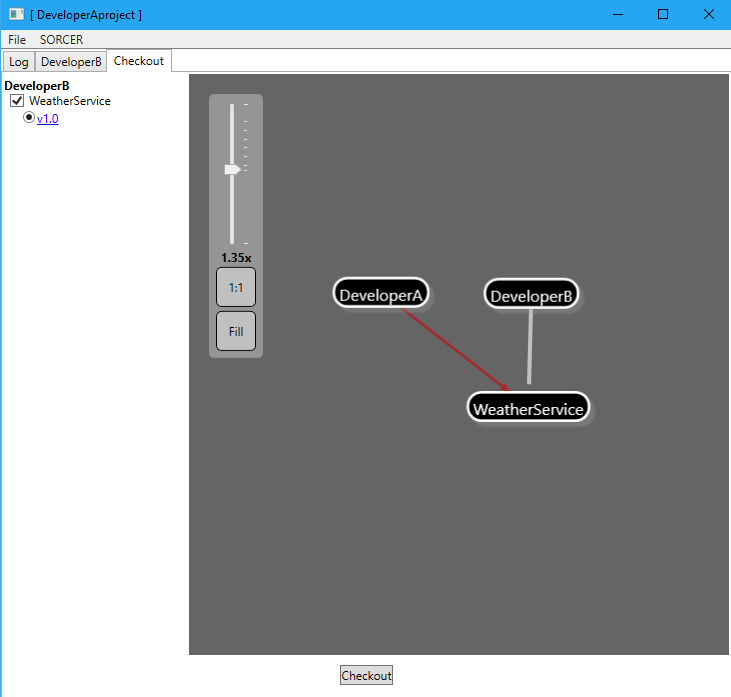


* **DeveloperA:**

CheckOut to discover and use the weather service from DeveloperB.



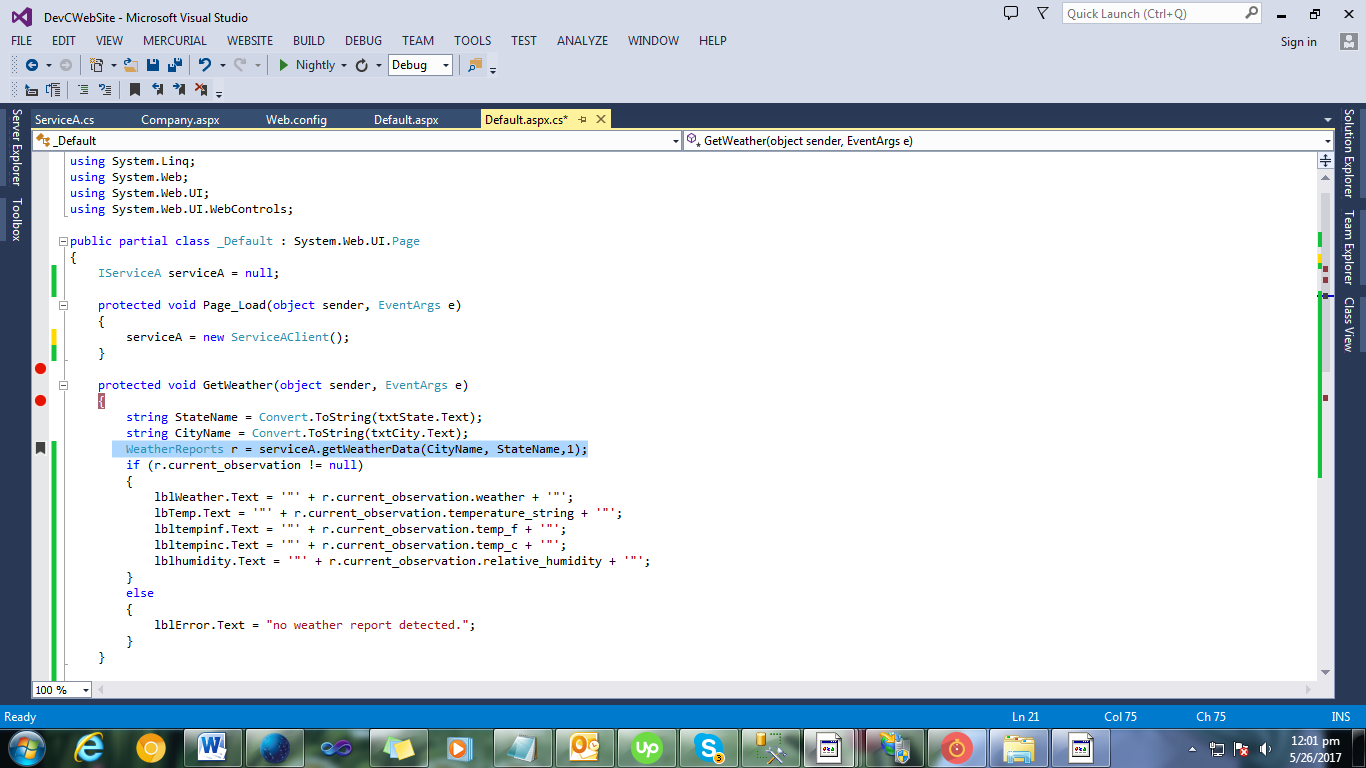
Once the service is discovered, DeveloperA selects the service and the version number to generate the local proxy.



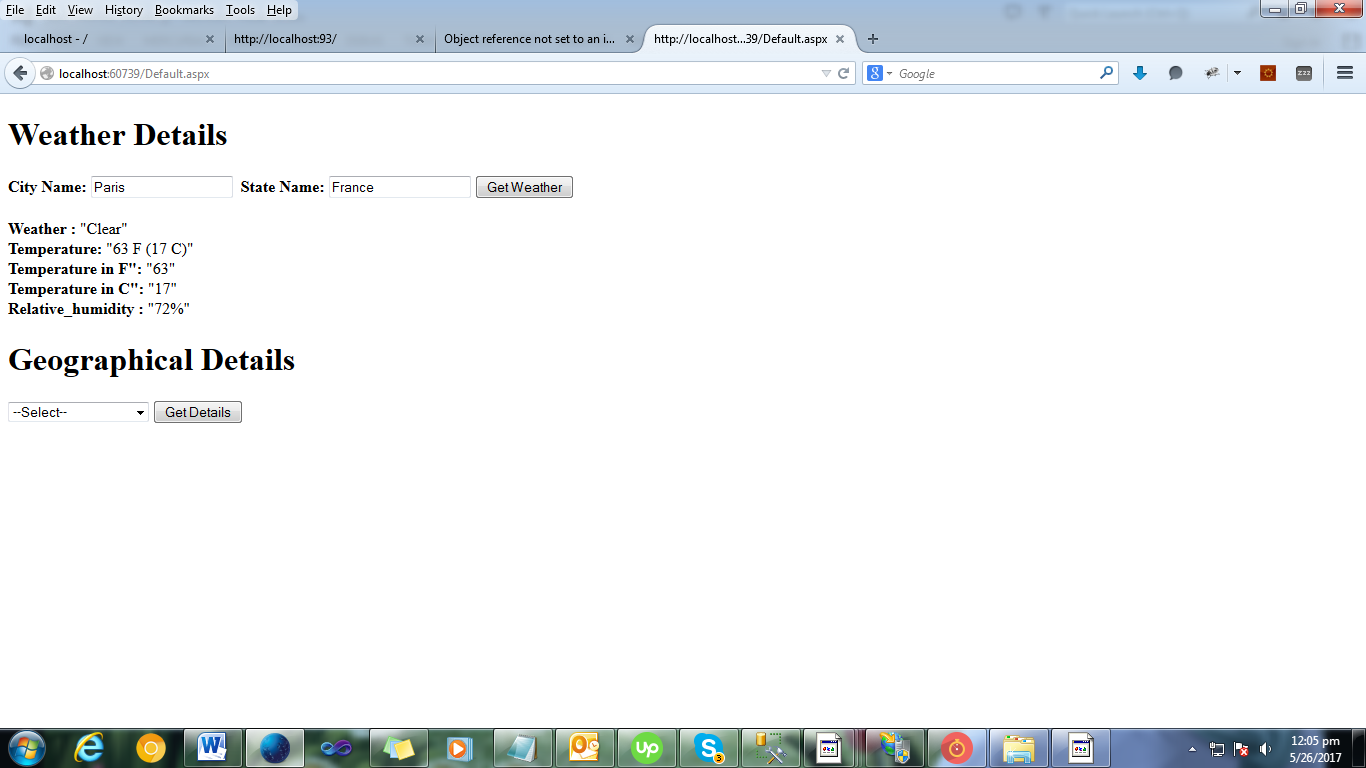
After generating the proxy, DeveloperA gets the weather service in the project solution in his/her workspace



DeveloperA called the method of service in its function as shown:



And below is the output

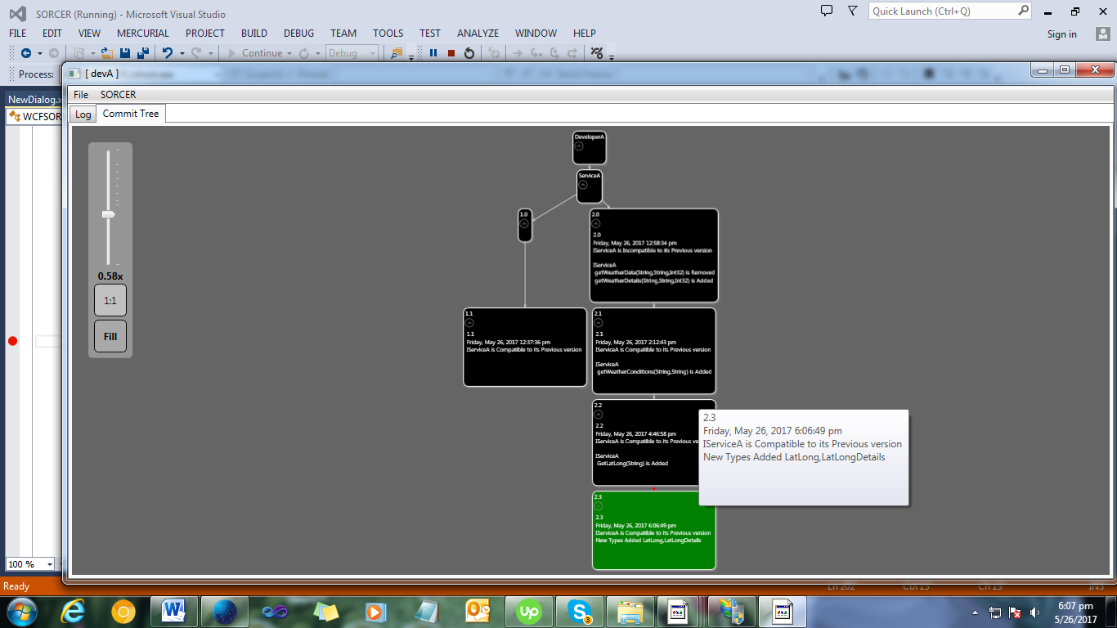


With this application base established, the next two sections look at what happens when a compatible and incompatible change is made.

1. **Change Scenario 1 (compatible change)**
   1. Service provider side (DeveloperB)

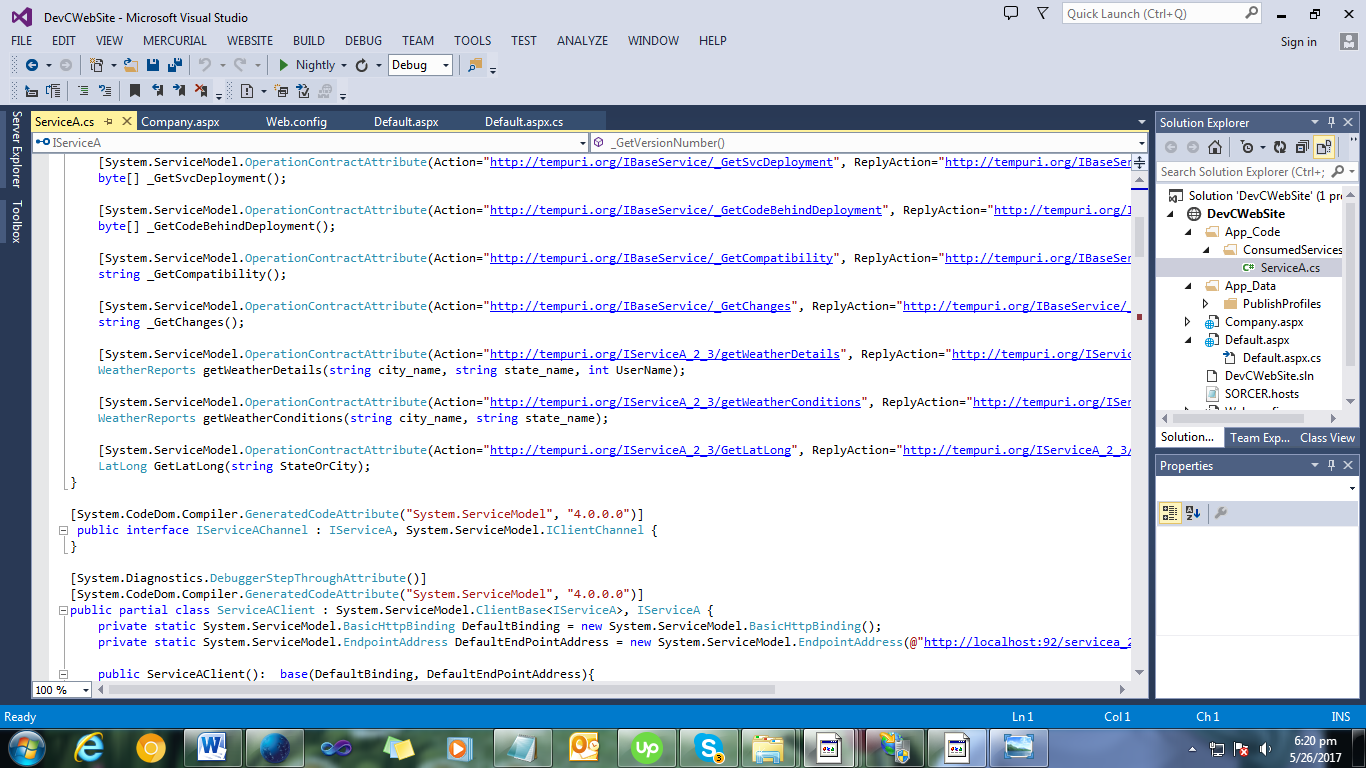
DeveloperB makes a compatible change (see Table-3), by adding a new method “GetLatLong” to the Weather service as snapshot below:

And then commits the new version. As the change is compatible, the new version will be WeatherService\_1\_1. It is clear from the screenshot that version 1.1 (operation is added).

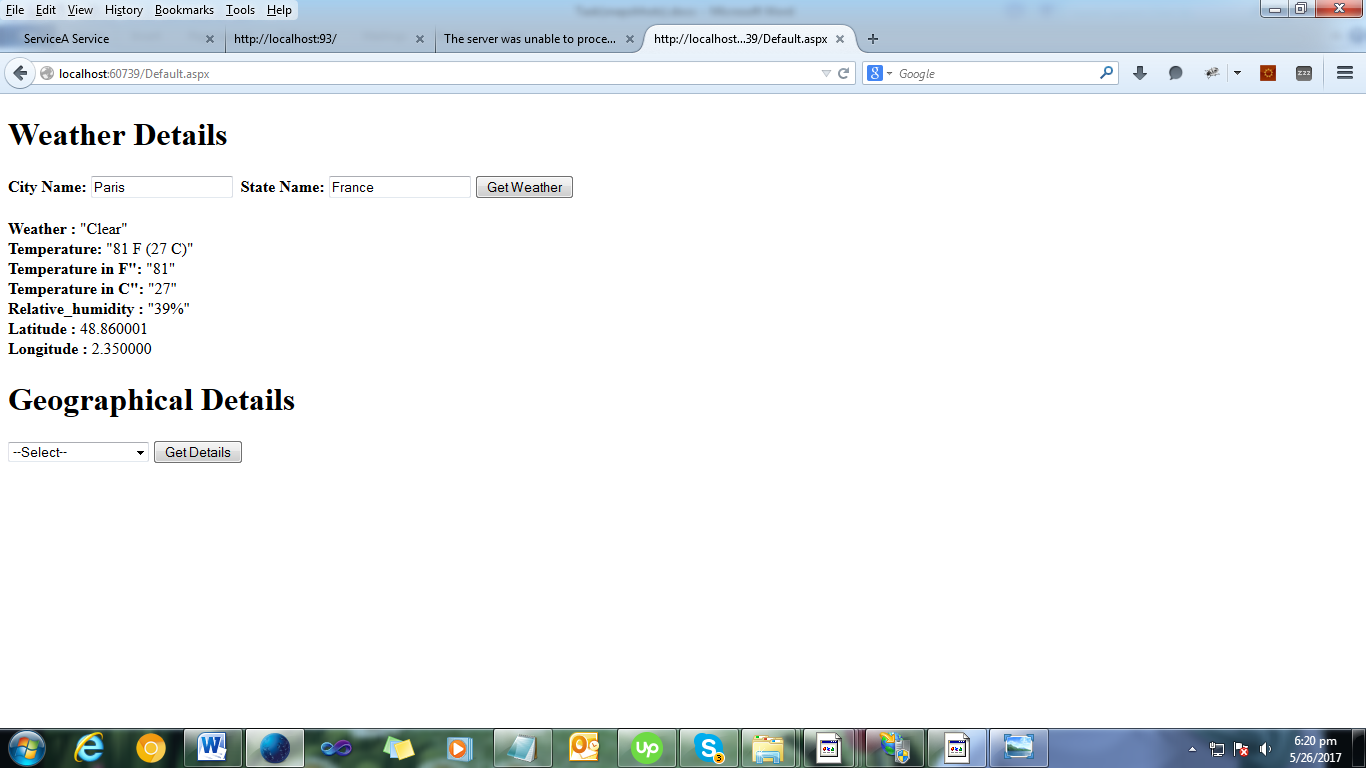


* 1. Service consumer side (DeveloperA)

After the new version is committed by DeveloperB, DeveloperA runs the Update command and automatically generates the proxy class

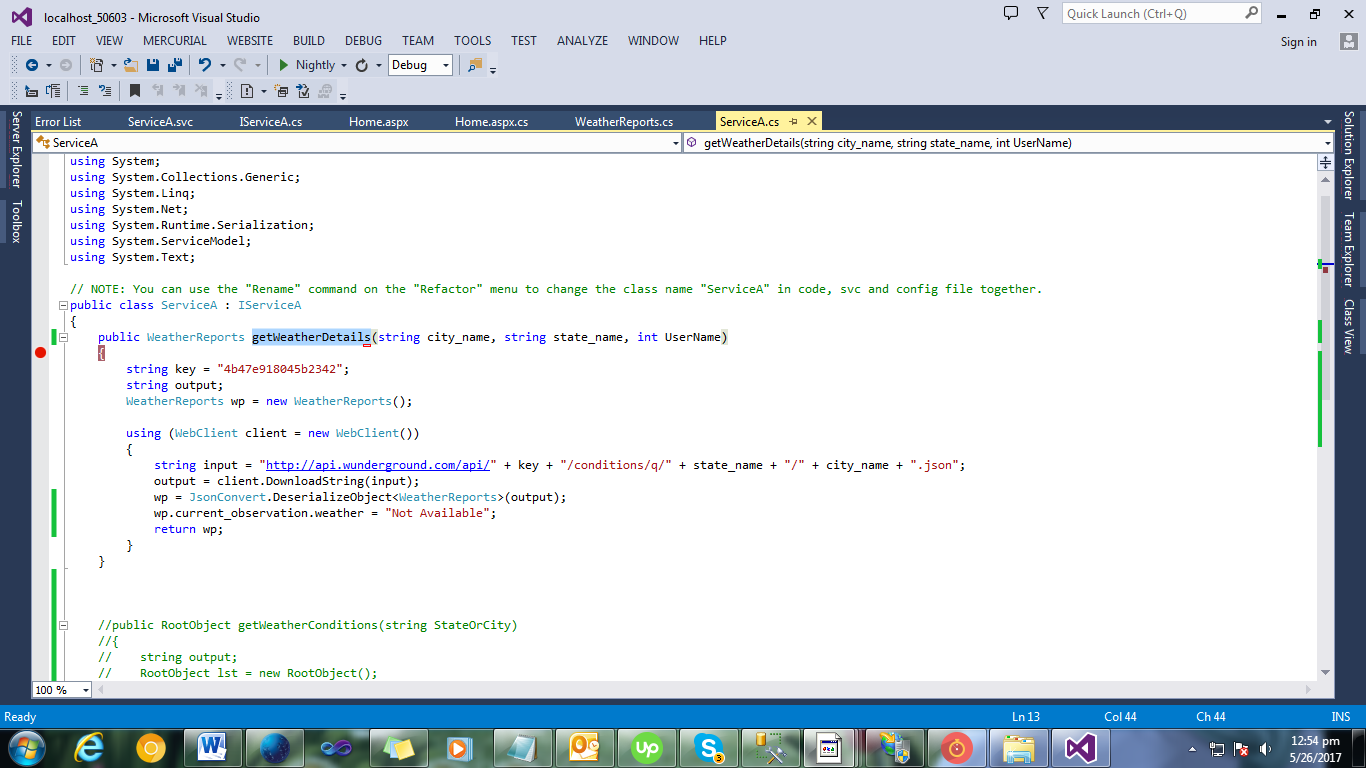


And the below is the output:

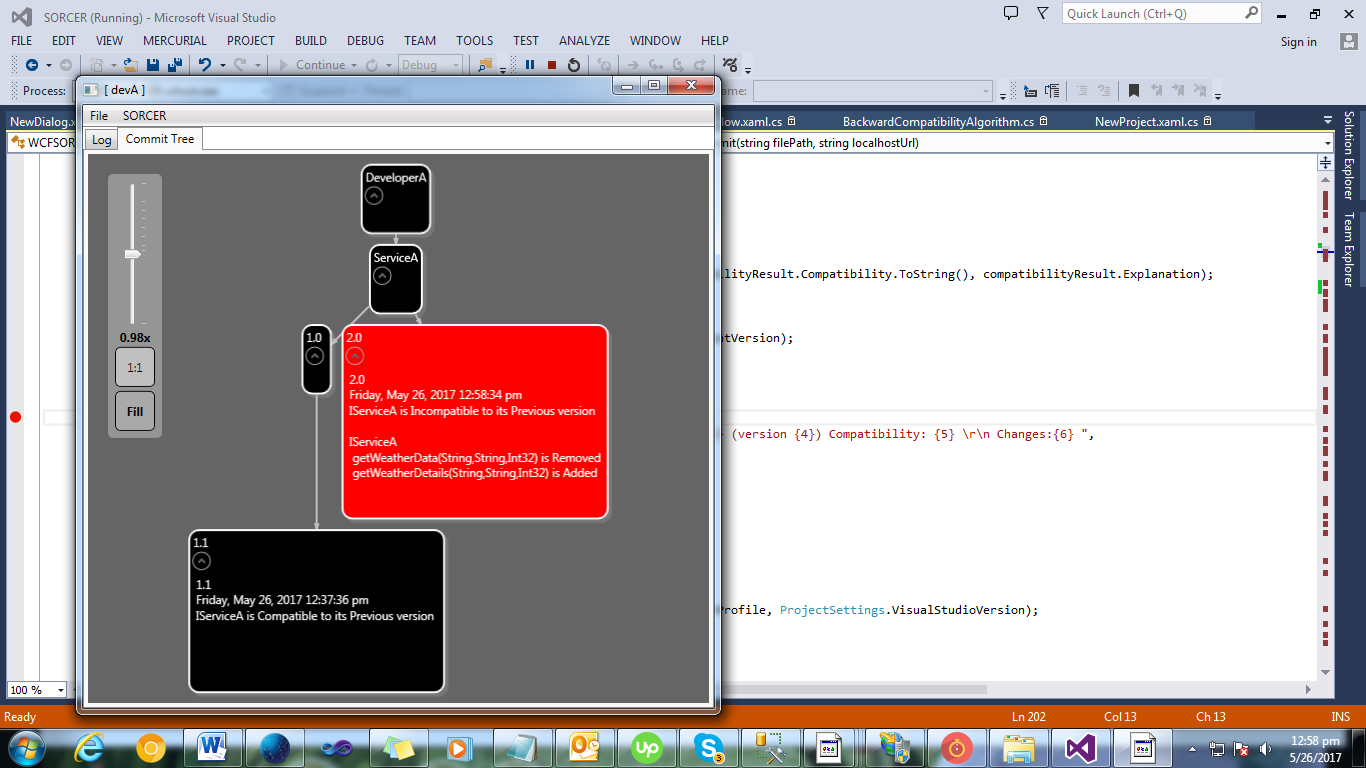


1. **Change Scenario 2 (incompatible change)**
   1. Service provider side (DeveloperB)

DeveloperB makes an incompatible change (see Table-3) to the Weather service, by renaming the operation



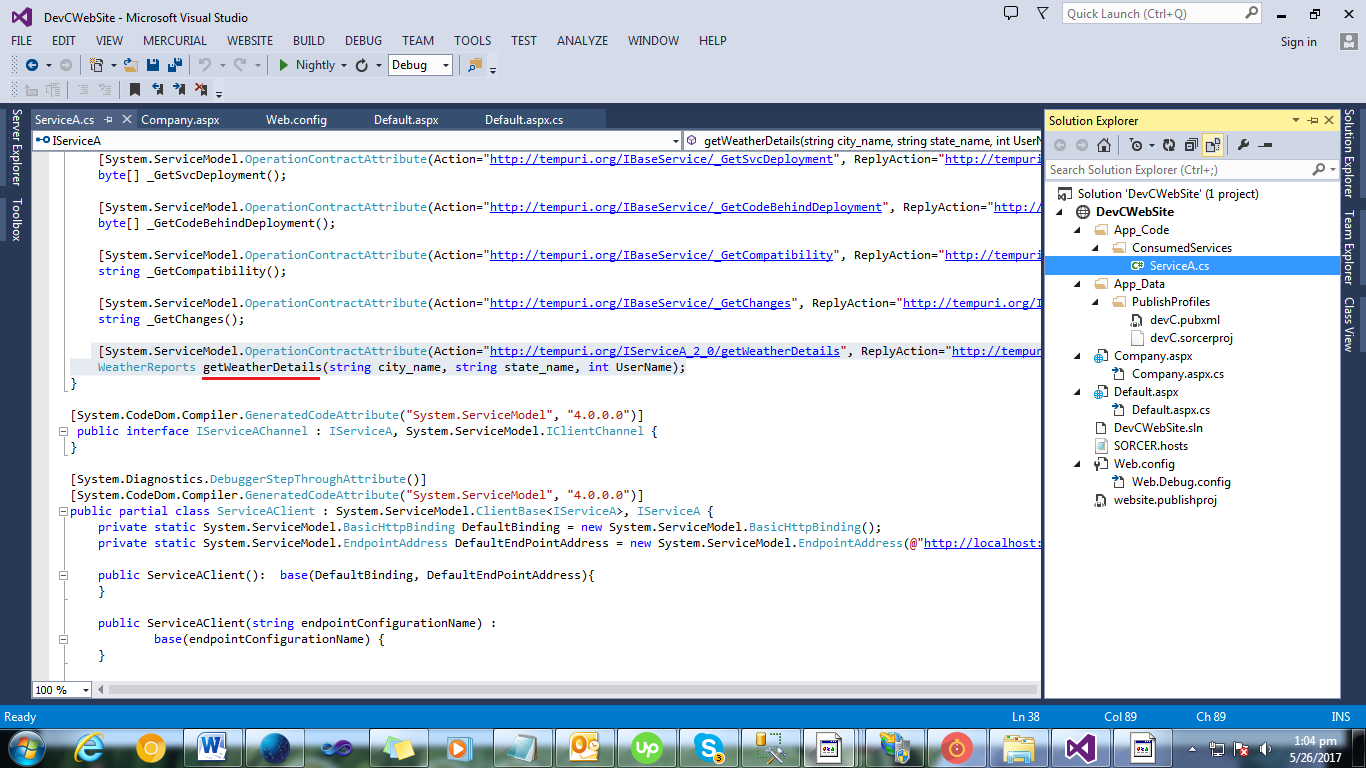
And then commits the new version. As the change is incompatible, the new version will be weather\_2\_0.



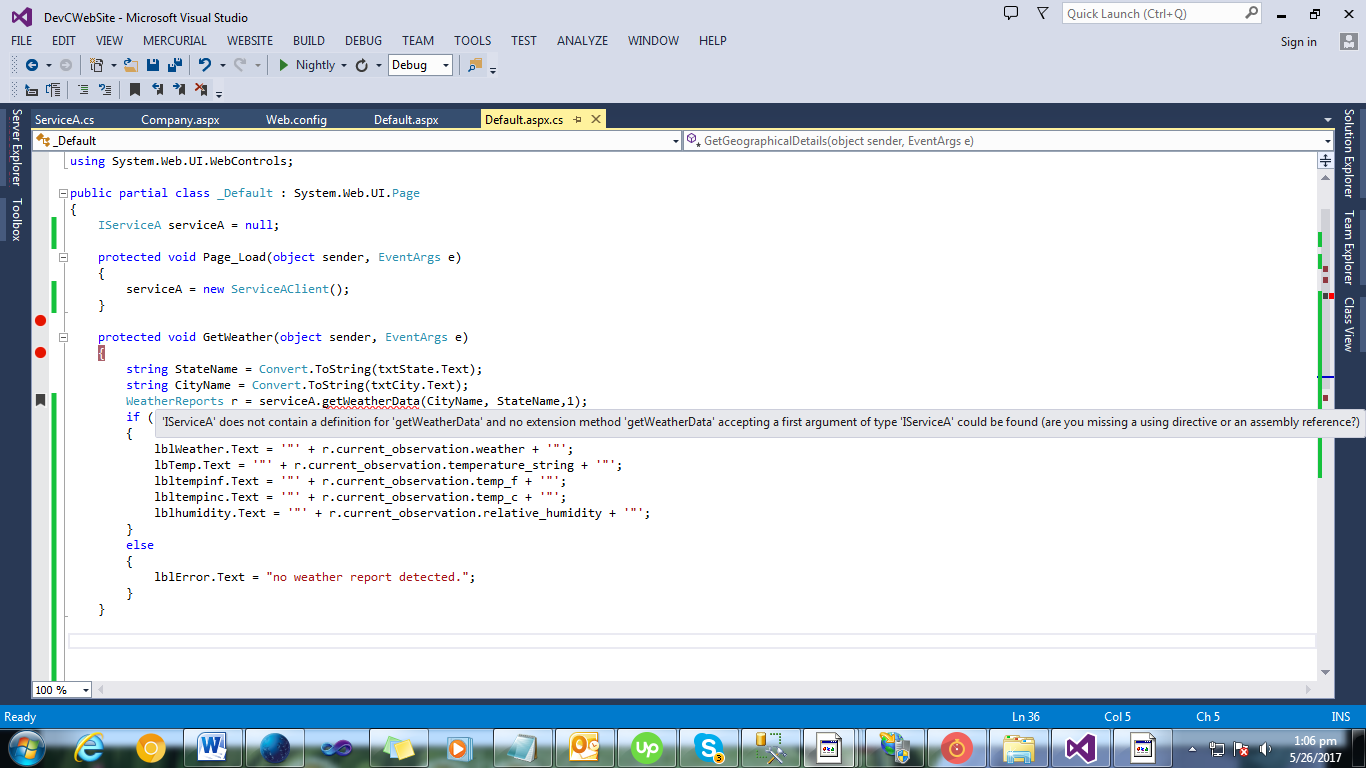
* 1. Service consumer side (DeveloperA)

After the new version is committed by DeveloperB, DeveloperA runs the Update command. This will result in a message asking for confirmation of whether the upgrade to the incompatible version should proceed as below:

If DeveloperA proceed to upgrade to the incompatible version then the he/she will generate the proxy and you can see the updated operation:



The code where DeveloperA was using the method getWeatherData is now showing error.



Its now showing the added change.

