Contact Managment API:

Technology:

The API was developed using .NET Core 2.1 and Microsoft SQL Server. The approach was using Code First for the Database.

Code Approach:

To properly organize the code(having in mind a bigger solution than the one required), I divided the solution in 3 projects, first the API solution with all the controllers and startup, then the Domain solution with the DBContext and all it's models and last, the Provider solution which handles all the database transactions in a transparent manner.

Database Structure:

Since one of the bullet points was to search by State and/or City, those fields have their own table, to avoid registers of the same place typed differently.

State:

```
public class State
{
    [Key]
    public int StateId { get; set; }
    [Required]
    public string Name { get; set; }
    public ICollection<City> Cities { get; set; }
}
```

City:

```
public class City
{
    [Key]
    public int CityId { get; set; }
    [Required]
    public string Name { get; set; }
    [Required]
    public int StateId { get; set; }
    public State State { get; set; }
}
```

To group information in a tidier manner and add flexibility, all the information regarding a contact's address is saved in a table called Address with the following fields:

```
public class Address
{
    [Key]
    public int AddressId { get; set; }
    [Required]
    public string StreetInformation { get; set; }
    [Required]
    public int StateId { get; set; }
    public State State { get; set; }
    public int? CityId { get; set; }
    public City City { get; set; }
}
```

Last, the main table is called Contact and contains all the information of the person and a reference to an Address:

```
public class Contact
{
    [Key]
    public int ContactId { get; set; }
    [Required]
    public string Name { get; set; }
    [Required]
    public string Company { get; set; }
    public string ProfileImageFileName { get; set; }
    [Required]
    public string Email { get; set; }
    public DateTime? BirthDate { get; set; }
    public string WorkPhoneNumber { get; set; }
    public string PersonalPhoneNumber { get; set; }
    [Required]
    public int AddressId { get; set; }
    public Address Address { get; set; }
}
```

The field ProfileImageFileName is the name of a file that was previously uploaded in the server. There is a method in api/files to upload a file that return a fileName with a timestamp.

Testing and Documentation:

The API is self-documented with the help of Swagger. The root of the API directs to the documentation, which also allows to test the methods from there.

For testing I've provided two tools:

First, as required, there is a unit test project testing some of the methods. The xUnitTestProject in the solution, to run it, in Visual Studio go to **Test->Windows->Tests Explorer** from that window you can run all the test, or each on individually.

Second, I've created a PostMan collection which can be ran from the PostMan Runner and is useful to manually test the API.

Postman Environment:
https://app.getpostman.com/join-team?invite_code=e6ab9cc878efcacad996b2b869acca37

Postman Collection:
https://www.getpostman.com/collections/173096bc2685f075addd

Note that the base URL of the api is a variable from the Environment which should be set accordingly, the rest of the variables are filled upon executing the requests.

Running and testing the API:

The code for the API is located at the following GitHub Repository:

https://github.com/MartinPereiraN/SolsticeContactManagment

As stated before, the API was created with a Code First approach, it has also been configured to create the Database when running(if it does not exist). The connectionString for the database is at the appsettings.json, to properly run the app it should be changed to a valid string for the environment used.

Since it was not in the scope of the challenge and it is inconvenient for the tester to have to create states and cities from the API, or worse, the database, the API populates the database upon creating it with all the states and cities of the USA. So once the API starts you can begin testing.

With this minor configuration the API should be ready to run from Visual Studio, or be deployed in a server.