How to build/run code
To import the image, run “docker load -i <path to input-validation-image.tar>” on powershell. Create a container by running the command “docker run -p 7151:7151 input-validation-image”. The API should now be running and can be connected to at “http://localhost:7151/swagger/index.html”. NOTE: only works with http, not https.

How to run tests
Postman workspace:
https://www.postman.com/universal-crescent-489435/workspace/my-workspace/api/1cb1078c-86f1-4af7-9be1-d4bd656b5409
Navigate to the unit tests in the phonebook API, select “Run”, then run all unit tests.

There are a total of 30 unit tests. 9 good input tests, 9 bad input tests for phone numbers, 6 bad input tests for names, all based on the example inputs from the assignment description, and 6 extra tests I created for extra input. Each input test returns whether they got the expected return value of 200 for good input or 400 for bad input.

How code works
I used the template project for the code, since despite not knowing anything about c# or .net, I figured it would be faster to add a little more to that project rather than starting from square one. All of the changes to the code are in DictionaryPhoneBookService.cs and PhoneBookController.cs. In the dictionary file, I added the regex for both the name and phone number in the add function, along with throwing an error if the input did not match with them. The entries are stored in a text file, with each entry separated by a newline and the name and number are separated by commas. When the program starts, it checks if the file exists and will read in the existing entries if it does. Whenever a call is made that adds/removes from the dictionary, the same change is made to the text file. The log is also called for any request and errors made, and is another text file where lines are added for each update. The only change made to the controller file is adding a catch to the add function to return a 400 error when the regex doesn’t match.

Regex
Phone
^(?!.*\(.{3}\)\s.*)(?![\d]{6,})(?:(?:\+?\d{1,3}\s*(?:\(\d{2,3}\)|\d+)[.\s-]?)|(?:\(\d{2,3}\))|[.\s-]?)?(?:\d{1,4}[. -]?){5}$
(?!.*\(.{3}\)\s.*) - removes strings where whitespace follows parenthesis with 3 characters in it
(?![\d]{6,}) - removes strings with more than 6 digits straight in a row
(?:(?:\+?\d{1,3}\s*(?:\(\d{2,3}\)|\d+)[.\s-]?)|(?:\(\d{2,3}\))|[.\s-]?)? - optional area code, either with a plus or in parenthesis
(?:\d{1,4}[. -]?){5} - rest of phone number, 5 groups of 1-4 digits separated by space, period, or hyphen

Name
^(?:[a-zA-Z]+(?:['’][a-zA-Z]+)?(?:[a-zA-Z]+|[. ,-]{1,2})){1,3}$

[a-zA-Z]+(?:['’][a-zA-Z]+)? - first part of name, only allowing letters and a couple apostrophe letters
(?:[a-zA-Z]+|[. ,-]{1,2}) - checks for comma or other separation of letters
{1,3} - allows up to three parts of a name

Assumptions
One assumption was that using a text file for the logs would be sufficient. There are existing logger objects that provide the functionality, but for the purposes of this project and what was requested for the log the text file seems to be sufficient, and achieves everything that was outlined in the assignment description.

Pros/cons
A pro of the approach I took is that by using the template, I was able to focus on the implementation of the important requirements rather than spending time creating an API. Additionally, with the use of text files for both the entries and the logs, I was able to focus on the functionality of implementing them in the code without worrying about extra logistics that come with using specific special objects or frameworks for them. A con of this however, is that by going for a simpler approach with text files, I miss out on lots of extra functionality that come with the extra implementation since everything is stored entirely locally and everything has to be specifically hard coded into the files, missing out on the abstraction that could otherwise be utilized.