Q4: Problem Solver

Problem

The issue we decided to tackle was a problem that we encountered multiple times in working on projects for different courses. When developing web applications, we often create the frontend and backend separately for greater efficiency, and in these cases either platform needs a stub for development purposes on the other platform.

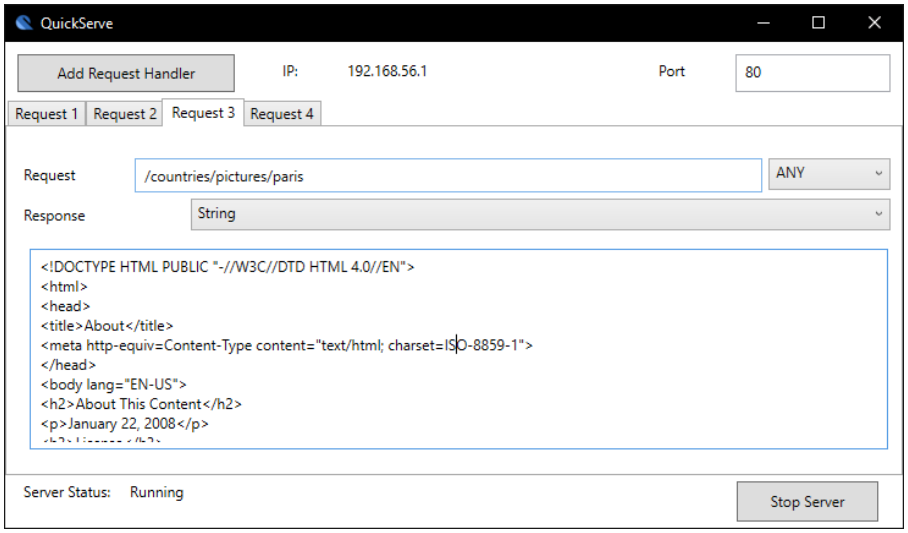
- While working on the **backend** or server-side application, we can simulate the frontend using an application called Postman, which is a very handy tool used to simply send requests and get responses from the server application we are developing and testing. This allows easy testing and debugging of the backend of the web application.
- However, when developing a **frontend** application, we do not have any simple ways stubbing the backend. So far we have been using simple PHP scripts returning given values for frontend testing before integrating with the backend, however that is comes with some baggage as well. Setting up apache servers for simple request returns is a bit extreme, so we came up with a solution.

Solution

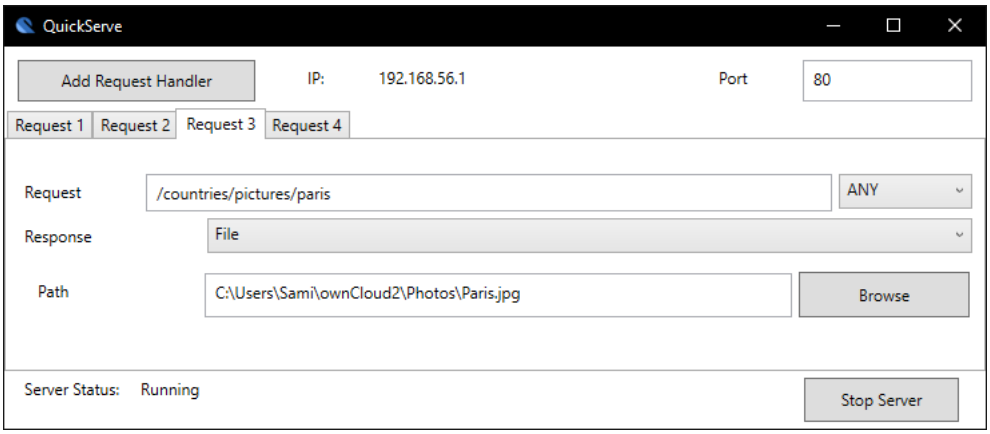
The solution we came up with is a standalone application called QuickServe that allows you to create a temporary microserver for reasons of testing and stubbing. The interface would allow you to change the return values real time, and no extra setup would be required. This permits us to create server stubs very fast and with minimal hassle.

The QuickServe application has three main request handling features:

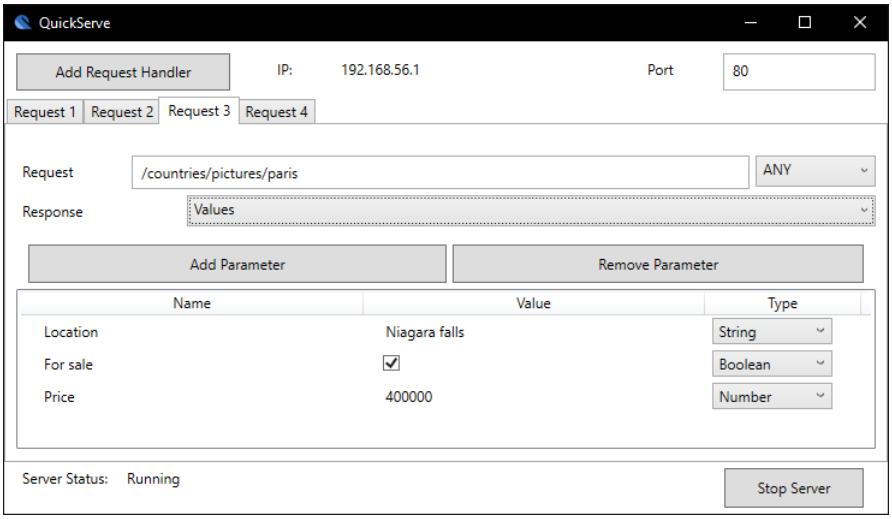
- String return – Allows the user to specify and return a string to the given request. By setting the string value to an html string you can serve a web page as well.



- File Return – Allows the user to specify a file to server for the request. QuickServe will automatically detect the file type and server it accordingly. HTML files will be displayed as web pages and image files will be returned as browser viewable.

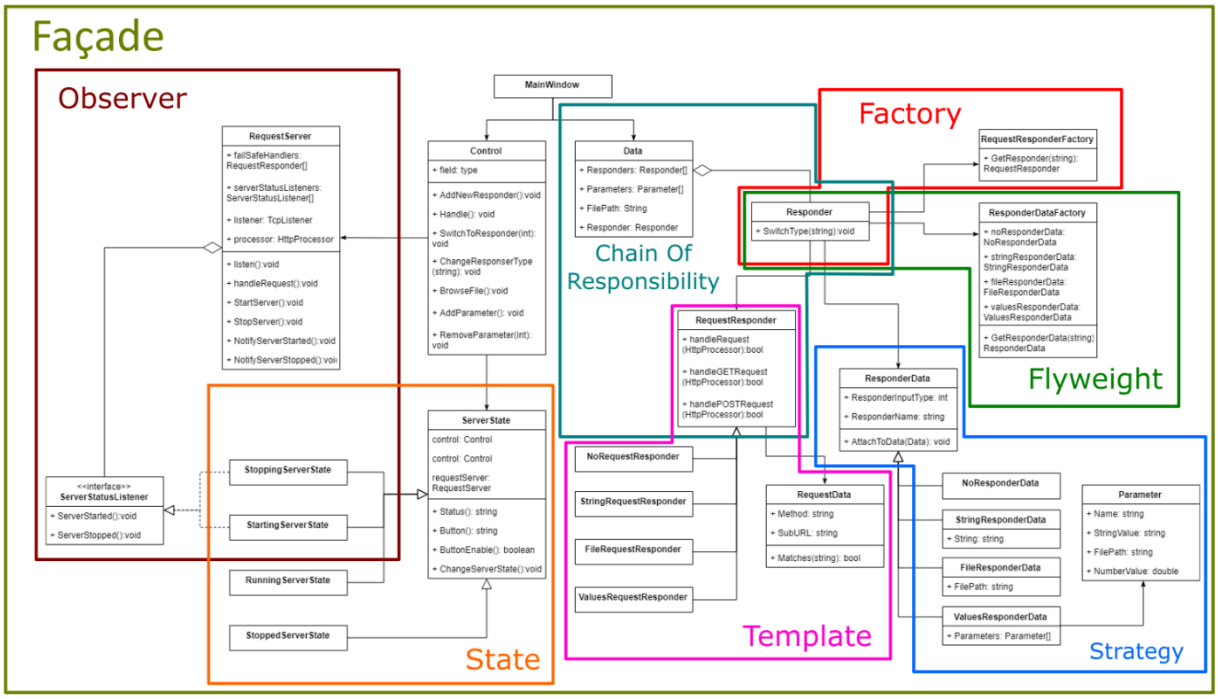


- Values Return – Allows user to specify a set of parameters that QuickServe will return as a JSON string to the request.



Design Patterns

A multitude of design patterns were used in the development of this tool.



Factory

Anytime a RequestResponder is switched from File to String, or from Values to File on the interface a new RequestResponder of the specified type needs to be created and attached to the responder. The RequestResponderFactory receives the type directly from the interface provides the corresponding RequestReponder object. Due to this pattern changes made when adding a new type of RequestResponder are localized to the RequestResponderFactory.

Chain of Responsibility

When a request is received by the RequestServer, it iterates through the list of responders and calls handleRequest on each one. If the conditions of the responder are met by the request, the responder handles the request and returns a true value, and the iteration is ceased. This allows separation of the handling logic from the server and dynamic manipulation of handlers during runtime.

Façade

The GUI Interface contains two objects of the Control and Data class, and each object has a set of objects belonging to other classes that facilitate operations within the application. This hierarchical structure allows the applications to be easily modifiable and easily understood.

Observer

Since the server is launched on a separate thread from the interface, whenever a time consuming operation is executed within this separate thread, the main thread should be notified so as to update the interface. For this we use the observer pattern, so that after the server thread has been sent the signal to stop or start, the interface can show the status of the server when it has finally started or been stopped.

Strategy

The Data class uses the strategy method when binding the ResponderData object to the interface. Whenever a user cycles through the request tabs, the ResponderData of the current Responder is selected and attached to the Data object, so that changes made via the interface affect the data within the selected request. This simplifies the process of defining new kinds of ResponderData since other ResponderData objects do not have to be modified in the process.

State

There are four states of the RequestServer: Starting, Running, Stopping and Stopped. The RequestServer object needs to respond differently to different bindings and calls depending on the state it is in, therefore the RequestServer has a ServerState object that it passes such responsibilities to. This has isolated the functions executed at each state and saved a lot of if statements and disorganization.

FlyWeight

The ResponderDataFactory is a flyweight object despite its name, since it stores the different classes it has created for a responder. A Responder has its own ResponderDataFactory, and whenever the response type is switched for that request, the ResponderDataFactory returns the stored one if there is any, and a new one if there isn't. The Responder retains the stored data for that type of ResponderData and required types of ResponderData are created on demand.

Template

Iterating through the available requests, the handleRequest method is called which is implemented by the RequestResponder class. The classes inheriting from this class implement the handleGETRequest and handlePOSTRequest methods, so the base class is the one that assesses the type of request, extracts the variables and calls one of these methods of the child class depending on the type of request received.