**HTTPRouter using a Trie**

For this exercise we are going to implement an HTTPRouter like you would find in a typical web server using the Trie data structure we learned previously.

There are many different implementations of HTTP Routers such as regular expressions or simple string matching, but the Trie is an excellent and very efficient data structure for this purpose.

The purpose of an HTTP Router is to take a URL path like "/", "/about", or "/blog/2019-01-15/my-awesome-blog-post" and figure out what content to return. In a dynamic web server, the content will often come from a block of code called a handler.

**[[A picture containing shirt

Description automatically generated](https://classroom.udacity.com/nanodegrees/nd256/parts/da17de0f-f834-46f8-bb48-ee2705d95dc4/modules/bd252a0b-e9e7-473b-bcc1-bc7e3153568b/lessons/8ec390d0-e99d-44c0-88f9-f8f9faf467fc/concepts/8a8492a5-f76d-4dcc-a07c-3e2a2ca9b3c6)](https://classroom.udacity.com/nanodegrees/nd256/parts/da17de0f-f834-46f8-bb48-ee2705d95dc4/modules/bd252a0b-e9e7-473b-bcc1-bc7e3153568b/lessons/8ec390d0-e99d-44c0-88f9-f8f9faf467fc/concepts/8a8492a5-f76d-4dcc-a07c-3e2a2ca9b3c6)**

First we need to implement a slightly different Trie than the one we used for autocomplete. Instead of simple words the Trie will contain a part of the http path at each node, building from the root node /

In addition to a path though, we need to know which function will handle the http request. In a real router we would probably pass an instance of a class like Python's [**SimpleHTTPRequestHandler**](https://docs.python.org/3/library/http.server.html#http.server.SimpleHTTPRequestHandler) which would be responsible for handling requests to that path. For the sake of simplicity we will just use a string that we can print out to ensure we got the right *handler*

We could split the path into letters similar to how we did the autocomplete Trie, but this would result in a Trie with a very large number of nodes and lengthy traversals if we have a lot of pages on our site. A more sensible way to split things would be on the parts of the path that are separated by slashes ("/"). A Trie with a single path entry of: "/about/me" would look like:

(root, None) -> ("about", None) -> ("me", "About Me handler")

We can also simplify our RouteTrie a bit by excluding the suffixes method and the endOfWord property on RouteTrieNodes. We really just need to insert and find nodes, and if a RouteTrieNode is not a leaf node, it won't have a handler which is fine.

*# A RouteTrie will store our routes and their associated handlers*

**class** **RouteTrie**:

**def** **\_\_init\_\_**(self, ...):

*# Initialize the trie with an root node and a handler, this is the root path or home page node*

**def** **insert**(self, ...):

*# Similar to our previous example you will want to recursively add nodes*

*# Make sure you assign the handler to only the leaf (deepest) node of this path*

**def** **find**(self, ...):

*# Starting at the root, navigate the Trie to find a match for this path*

*# Return the handler for a match, or None for no match*

*# A RouteTrieNode will be similar to our autocomplete TrieNode... with one additional element, a handler.*

**class** **RouteTrieNode**:

**def** **\_\_init\_\_**(self, ...):

*# Initialize the node with children as before, plus a handler*

**def** **insert**(self, ...):

*# Insert the node as before*

Next we need to implement the actual Router. The router will initialize itself with a RouteTrie for holding routes and associated handlers. It should also support adding a handler by path and looking up a handler by path. All of these operations will be delegated to the RouteTrie.

Hint: the RouteTrie stores handlers under path parts, so remember to split your path around the '/' character

Bonus Points: Add a not found handler to your Router which is returned whenever a path is not found in the Trie.

More Bonus Points: Handle trailing slashes! A request for '/about' or '/about/' are probably looking for the same page. Requests for '' or '/' are probably looking for the root handler. Handle these edge cases in your Router.

*# The Router class will wrap the Trie and handle*

**class** **Router**:

**def** **\_\_init\_\_**(self, ...):

*# Create a new RouteTrie for holding our routes*

*# You could also add a handler for 404 page not found responses as well!*

**def** **add\_handler**(self, ...):

*# Add a handler for a path*

*# You will need to split the path and pass the pass parts*

*# as a list to the RouteTrie*

**def** **lookup**(self, ...):

*# lookup path (by parts) and return the associated handler*

*# you can return None if it's not found or*

*# return the "not found" handler if you added one*

*# bonus points if a path works with and without a trailing slash*

*# e.g. /about and /about/ both return the /about handler*

**def** **split\_path**(self, ...):

*# you need to split the path into parts for*

*# both the add\_handler and loopup functions,*

*# so it should be placed in a function here*

**Test Cases**

*# Here are some test cases and expected outputs you can use to test your implementation*

*# create the router and add a route*

router = Router("root handler", "not found handler") *# remove the 'not found handler' if you did not implement this*

router.add\_handler("/home/about", "about handler") *# add a route*

*# some lookups with the expected output*

print(router.lookup("/")) *# should print 'root handler'*

print(router.lookup("/home")) *# should print 'not found handler' or None if you did not implement one*

print(router.lookup("/home/about")) *# should print 'about handler'*

print(router.lookup("/home/about/")) *# should print 'about handler' or None if you did not handle trailing slashes*

print(router.lookup("/home/about/me")) *# should print 'not found handler' or None if you did not implement one*

NEXT