Section 3 – Development

Version 1

Version 1 represents the initial work done on this project before I added the project to GitHub. In reality this was more than one version but since it was pre GitHub I have no changelog or history about it. The project was fairly well developed by this point, consisting of 13 pages 3 helper scripts. This version was also using the old front end system, before I moved to bootstrap and the new design. This meant that adding new pages took a lot of time as each element needed its own custom CSS.

This also meant that the website wasn't responsive. This means that all the styling on the website was designed for one screen size and resolution. Whist it was still useable on some similar or bigger screen resolutions, the overall user experience was pretty poor and the website didn't look very good. After this release I learnt about bootstrap, a responsive front-end web framework that seemed to provide everything I needed.

Version 1 had quite a bit functionality, although most of the codebase has been re-written in more recent commits as my knowledge and understanding of PHP increased. You could create accounts, log in, ask questions, see questions in a table and view individual questions. Although heavily edited and refactored, all of these files exist in the most recent version to date.

Below is an analysis of the notable code used in version 1:

account.php:

First the script specifies the doctype of the page as html, and initiates a new PHP session by calling the session_start() function. It then includes the connect.php script, which is used in the early versions of the program to connect to the database. Next it creates a new MySQLi connection object called $connection by using the connect() function from connect.php.

The script then checks if a user is logged in or not, by checking the value of the SESSION variable 'username'. If this variable is not set, then no user is logged in and the script displays a link to the 'Sign up / Login' page on the header. If it is then a user is logged and it displays a link to the 'My account' page. It then displays the rest of the header links.

The same check is then done again later on in the script, displaying a 'Log out' link if a user is logged in or a 'Sign up / Login' link if not.

This is the code for a blank template page under the old system, and is used for all pages that were undeveloped.

admin.php:

First the script performs a similar check to the username SESSION variable check in the previous script, except on this page the script isn't just checking that the variable has been set, but comparing the value of the variable to a hardcoded constant. This is to only allow one user with one username to access this page. If this check fails, the page redirects back to the homepage instantly. It does this by calling the built in header() function, using a String consisting of 'Location:', and the URL of the homepage at this time.

However, if this check succeeds, the script displays an HTML form where the user can enter an IP address to block. This is a POST form and it redirects to the processipblock.php script when the submitted.

ask.php:

This page starts off using the blank page template used by account.php, with additional features like a basic HTML form with inputs for the question title and question body. As with admin.php, this is a POST form, and redirects to processquestion.php.

downloadquestions.php:

Possibly the most complex PHP script in the entire project, and one that's relatively unchanged since its addition in version 1, except for some minor bug fixes, the purpose of this script is to download and filters the questions from the database in accordance with the filters specified by the dropdown on questions.php. These are then outputted in a format readable by a JavaScript script running on questions.php in order to populate the questions table.

This script starts off the same way as all the others in this version, starting a PHP session and including the connect.php script in order to create a connection object. Then, as database connectivity is vital for this script to function, it then checks to see if the connection as established successfully. If this check fails, then the script will exit with an error.

Providing a connection was successfully established to the database, the script gets the type of filter from the URL. The script then prepares a MySQL query for each filter. The 'top' and 'new' filters are the easiest, as all the filtering and sorting are all done in the MySQL queries as you can sort by columns and order in ascending and descending order.

However, the 'hot' filter is much more complicated than the others, and cannot be filtered by a simple MySQL query. It then creates two empty arrays, $scoreArray and $array. Then, it iterates through the $result array. A $points variable is created and set equal to the number of votes question currently being iterated over has (given by $row["votes"]). A variable called $order is also created. The value of $order is given by ($\log_e$(max(abs($points), 1, 10). Then, if the $points variable is bigger than 0, A variable called $sign is created and set to equal 1. If the $points variable is smaller than 0, A variable called $sign is created and set to equal -1. If neither of the two above conditions are met, a variable called $sign is created and set to equal 0.

Then, a variable called $seconds is created and set to the Unix time when the question was asked - the constant 1516221943 (Wednesday, 17 January 2018 20:45:43, the date the system was first implemented). Another variable, $score, is also created, and assigned the value of ($order + $sign * $seconds / 45000) rounded to 7 decimal places. The rounding is done using the build in method round(). It then pushes $score to the $scoreArray array, and pushes a new array composed of $score, $row["title"], $row["id"] and $row["votes"] to the $array array.

Then, after all the returned questions have been processed, it sorts the array by a user defined order as defined in the sortOrder() function. The function to sort an array by a custom sorting function is provided by the build in method usort().

Then, the contents of $array are outputted in a format that can be read by the scripts running on question.php.

index.php:

This is the script for the homepage of the website in version 1. It's also based off the blank page template, but with a lot of modifications including logging details about the user into the database.

The logging at the top of the script works by collecting the user's IP Address, the current time and the current date, and inserting them into a 'visits' table in the database. The script does not attempt to log anything if it cannot establish a connection to the database in order to prevent errors.

There is also a footer at the bottom of the page which renders status icons for both the website and database, for development purposes.

logout.php:

This script logs the user out of the website by wiping the 'username' SESSION variable and redirecting the user back to the homepage. This is the simplest script in this version of the project and remains relatively unchanged throughout subsequent versions.

processipblock.php:

The process*.php scripts in this project appear in every single version and are used to process form data and interface with the database.

This script takes the form data from admin.php and uses an INSERT INTO MySQL query to insert the user's IP address and the date into the database, providing a connection to be established to the database.

processlogin.php:

As with processipblock.php, this is a backend script that runs after a specific form has been submitted. This script is runs after a user clicks 'log in' and is used to validate their credentials either log them in or produce an error message depending on the username/password combination they enter in the form. These fields are sent as POST data and are received by the script by assigning $username and $password to the value of $_POST['username'] and $_POST['password'] respectively.

As always, first the script checks that it can establish a connection to the database, exiting with an error message if not. Then it gets the username and password that the user entered as described above and creates a new MySQL query to select the 'id' and 'password' attribute from the 'users' table from the record where the 'username' attribute matched the username that the user entered on the previous page. It then runs the query and stores the result.

If there is a result from this query, then the username entered must be a valid username stored in the database because the mysqli_query function returns null if no data is returned. Therefore the script next compares the result returned from the database against null. If the result is not null then the script continues, but if it is then the user doesn't exist and the script exists with an error message telling the user that their username does not exist.

Providing the username exists, the script then creates an associative array of the data returned from the database. It then uses this array to get the encrypted password hash returned from the database, and uses the build in function 'password_verify()' using the hash returned from the database and the password the user entered as the parameters. If this returns true then the

password matches the hash and the login details the user entered are correct. The user can then be logged in and the page redirects back to the homepage.

However if this function returns false then the user entered the wrong password and the script displays an error message and a button to try again.

processquestion.php:

This is the backend script that uploads a question to the database. As well as simply adding the question to the database, this script does some minor processing in the form of checking if all the fields contains data (using the same process as described in the above script) and checking if the user has been blocked.

Providing the checks pass without issue, the script uploads the question fields into the database.

processsignup.php:

This is the script that is ran from the signup form on signup.php. First it collects all the POST data from the form, then it performs the same series of checks as the scripts above, plus some more specialised checks for its specific purpose. Theses checks involve validating the email address and checking that the username has not already been taken.

Validating the email address is only done primitively at this stage. I use the built in strpos function which is used to return the position of a substring in a String. However, it can be used to return true/false if no comparison operator is used. Therefore I used this function, on the username variable to check if it contained an "@" symbol. This method is improved in later versions because this would validate "@" as a valid email, which it isn't.

Checking if the username is already taken however, is a more complex task. To do this, I create a MySQL query to select only the username attribute (to save processing time and bandwidth) from the database from whichever record has a username value of the username sent from the signup form. Then I use the built in function mysqli_num_rows to get the number of rows returned from the database. If this is less than 1 (0) then the username is unique and the user can be created, However if not then the username has already been used and the script returns and error.

Providing the username is unique, the script create the user by adding a new record to the database with all the data entered in the form. At this stage I haven't implemented password hashing, so this is less secure. Adding the record to the database is again done with a simple MySQL query.

qa.php:

qa.php is the homepage for the questions section of the site – the most important of the three. The page is mostly built from static HTML but when the page loads, it calls on a JavaScript function called Download() with "hot" as the parameter. This function is used to asynchronously download, filter, process and display questions from the database. The parameter specifies which filter to use. I'll discuss how this function works in the section for filterQuestion.js. On qa.php there is a table with no entries. This is where the downloaded questions appear. The purpose of calling the function as soon as the page has loaded is so that questions can be on the screen before the user changes any filters.

There is a dropdown menu above the table. This is used to filter the questions and whenever a user changes the filter the JavaScript script asynchronously downloads the questions for that filter. This

doesn't require a page reload which gives my website an advantage over websites like StackOverflow and reddit which do require a page reload. I believe that this greatly enhances user experience.

question.php:

question.php is the page that displays a question loaded from the database. There are a lot of functions in this script so I moved most of them to an external class called questionFuncs and included them in the top using the PHP 'include' statement. This is used provide access to all the functions in the class as if they were declared in the same page, but with the extra benefits of being able to use the functions from any class (by including the class) and keeping the page (made from predominantly HTML) clean.

When the page has loaded, the page calls a JavaScript function called SetHeight(). I'll talk about how this function works in the section for this page. The purpose of this function is to apply a quick fix for the height of the question vote arrows which were always slightly and inexplicably out of position.

Most of this page is composed of HTML markup for the structure of the page, but the main functionality consists of a PHP function used to download the question from the database based on the URL parameter, and some more PHP used to display the correct colour voting arrows depending on the way the user has (or hasn't) voted on the specific question.

The function for downloading and outputting the question works by getting the ID of the question from the URL, then querying the database to select all the attributes for the question with that ID. It then outputs the attributes along with some HTML markup and styling.

The function for displaying the different types of voting arrows works by calling on several of the functions stored in questionFuncs.php. First it checks if the user is logged in by checking if the 'username' SESSION variable is empty or not.

Assuming the user is logged in, it next uses the UsrVoted function from questionFuncs.php to check if the user has voted on the question or not. This function takes three parameters: the user's id (from $_SESSION["id"]), the question id (from $_GET["id"]) and a reference to the MySQL connection. This returns true if the user has voted on the question or false if they haven't.

If the user has voted on the question, the script next checks which way the user has voted, using the Upvoted function from questionFuncs.php. This function takes the same three parameters as UsrVoted and returns true if the use has voted the question up, and false if the user has voted the question down. It uses this return value to draw the correct colour arrows.

If the user has not voted on the question, or the user isn't logged in, the script draws the standard grey arrows used to represent no votes cast.

questionFuncs.php:

questionFuncs.php is a script used mainly by question.php in order to make the mainly markup based script easier to read and understand. This script isn't designed to be accessed by the web browser, and consists entirely of several function definitions.

The first function defined in questionFuncs.php is called UsrVoted, and takes three parameters: the user's id (from $_SESSION["id"]), the question id (from $_GET["id"]) and a reference to the MySQL connection. The function executes a MySQL query to check if the user has voted on a question or not. This is a SELECT query which selects the `id` attribute (the smallest data to collect about a record) from the `votes` table where the user id attribute (`uID`) is equal to the $id parameter and the question if attribute (`qID`) is equal to the $qID parameter.

It then uses the mysqli_num_rows function on the data returned from the query to check how many rows have been returned from the database. Due to the nature of the database and the query, the number of rows will only every be 1 or 0, 1 representing the user having voted on the question, and 0 representing the user not having voted on the question.

The second and final function in this file in this version is called Upvoted, and is used to determine which way a user that has voted on a question (represented by their user id and question id) has voted. This function takes the same three parameters as the previous function: the user's id (from $_SESSION["id"]), the question id (from $_GET["id"]) and a reference to the MySQL connection.

filterQuestions.js:

The first JavaScript script in this release, filterQuestions.js is used to download, process, filter and output questions from the database. It consists of two functions: Download() and Output(), and is in the style of a typical (but very simple) async JavaScript script.

The first function, Download, takes one parameter: the type of question to download from the database (an integer inclusively from 0-2). This function is called when questions.php is loaded, and is used to prepare and send an asynchronous GET request used to download the questions from the database.

First, it makes a new XMLHttpRequest object, and sets the URL target. This URL is the downloadquestions.php script (for interfacing with the database and downloading the data) with a URL parameter called type, set to the value of the type parameter passed into the Download function. The function next checks to see if there is any errors creating the XMLHttpRequest object, outputting an error message if there are.

Providing there are no errors, the function sets the onreadystatechange attribute of the XMLHttpRequest to the Output function, specifies the XMLHttpRequest as a GET request and sends it, initiating the asynchronous download.

The second function, Output, takes no parameters, and is known as a callback function. This means that the function is passed into another function as an argument, and called from that function. In this case, Ouptut is an asynchronous callback from the XMLHttpRequest object. This means that it's only executed once an action has finished executing. The purpose of this function is to collect the downloaded questions when they have finished being downloaded, and then output them into the HTML table on questions.php.

First, the script checks to see if the XMLHttpRequest has finished, and that there were no errors, outputting error messages if not. Assuming both theses conditions are passed, the function stores the output of the request, then creates an array of the response by splitting the response by "
"

(a line break in HTML). Each question in the response takes up three elements in the array. The function then loops over the HTML table and deletes all the non-header rows.

Then, the function iterates over the response array using a for loop, incrementing by 3 each time. For each iteration, the function creates HTML elements for each of the parts of the table (table row, table data, links, paragraphs and text nodes) and assembles them all together. It then creates and sets the attributes for certain elements including custom CSS, adding classes and id's, and setting link locations. After all this is finished, it adds a new table row to the table, representing one downloaded question. It does the same in the following iterations for the rest of the questions fetched from the database.

Finally, the function adds some padding in the form of 3 leading spaces to the number of votes for each question.

questionHeightFix.js:

This is a simple JavaScript script used to set the height of the votes div to the same height as the question-container div. It does this by getting the raw CSS height of the question-container div using the window.getComputedStyle function, and then adding a new height attribute to the votes div and setting it equal to the value returned from the window.getComputerStyle().height.

signup.php:

This page is entirely comprised from HTML and CSS and used as the front-end for the signup/login system. It consists of 2 HTML forms, one for login and one for signup.

***.css:**
The last 6 files in version 1 of NullPointerException are all stylesheets. These are used in the earliest versions of my project to add all the styling to the website, however just like the files only consisting of HTML, they are long and not suited to a detailed explanation.

Notes about version 1:

- Writing custom stylesheets for each page was time consuming and didn't produce good results
- Too much code was written in one version making it hard to document
- No comments in this version means documenting and maintaining it was harder than it should have been
- This version was pre version control which made it harder to revert any mistakes

Version 1 was the biggest version added to the project and contained all the code the entire project is based on. I made lots of mistakes in this early version of the project mainly in terms of the tools

and processes I used. However, it provides a good starting point for the rest of the project to follow from.

Version 1 testing:

I tested version 1 using two different methodologies: white box and destructive. White box testing is when the programmer tests the program using different inputs, with reference to the code. I did this to test specific inputs in specific forms. For example using invalid email addresses in the sign up form and unexpected characters when asking a question. I chose this methodology because the project wasn't complete enough to ask other people to test it, and by using the code I could ensure my tests were relevant. I also used destructive testing to test the interface by changing the resolution and using different browsers. Destructive testing is when you try and break the program by misuse. You try and act like the worst user in order to test your program. End users always end up using your program in ways you've never expected, and this type of test aims to protect against this. This is an effective test because it takes very little time and it simulates real use.

The results of my testing can be seen below:

Browser tests:

Browser	**Works/Doesn't work**	**Notes**
Mozilla Firefox	Works	None
Google Chrome	Works	None
Internet Explorer 11	Works	None
Microsoft Edge	Unknown	Won't connect to website at all so impossible to test
Mozilla Firefox (Android)	Doesn't work	Logo is too wide, site isn't mobile friendly
Google Chrome (Android)	Doesn't work	Logo is too wide, site isn't mobile friendly

Basic functionality:

Page	**Test**	**Test data**	**Works/Doesn't work**	**Notes**
signup.php	Can create an account	All valid data	Works	Script creates an account using the data provided

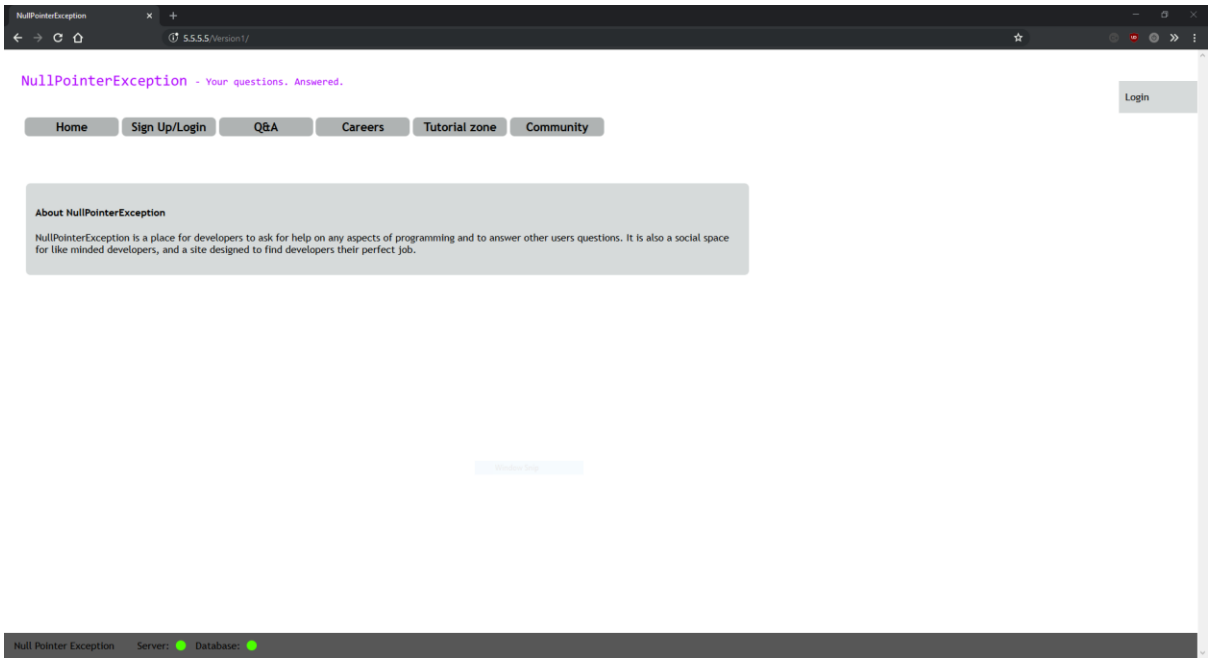
signup.php	Can create an account	No data	Works	Script prompts user to fill in missing fields
signup.php	Can create an account	☺☺☺☺☺	Doesn't work	Script doesn't create account and gives error message
login.php	Can log in to existing account	Username and password for an account existing in the database	Works	Script logs user into account
login.php	Can't log into account with wrong password	Username for an account that exists in the database, wrong password for said account	Works	Script prompts user that username/password combination is invalid
login.php	Can't log into account that doesn't exist	Username and password that doesn't exist in database	Works	Script prompts user that username/password combination is invalid

Non-working tests will be addressed in future versions of the project

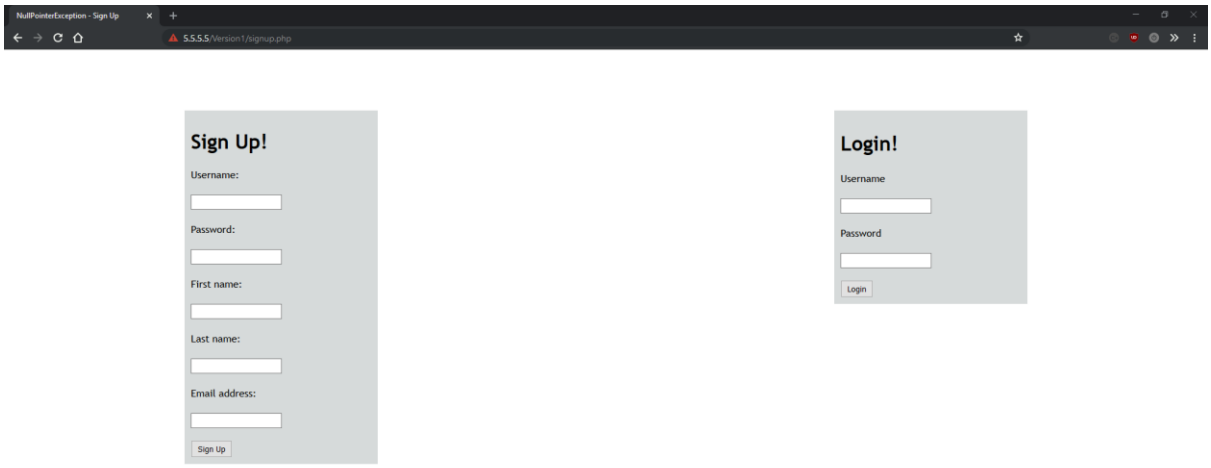
Version 1 screenshots:

NullPointerException - Your questions. Answered.

index.php

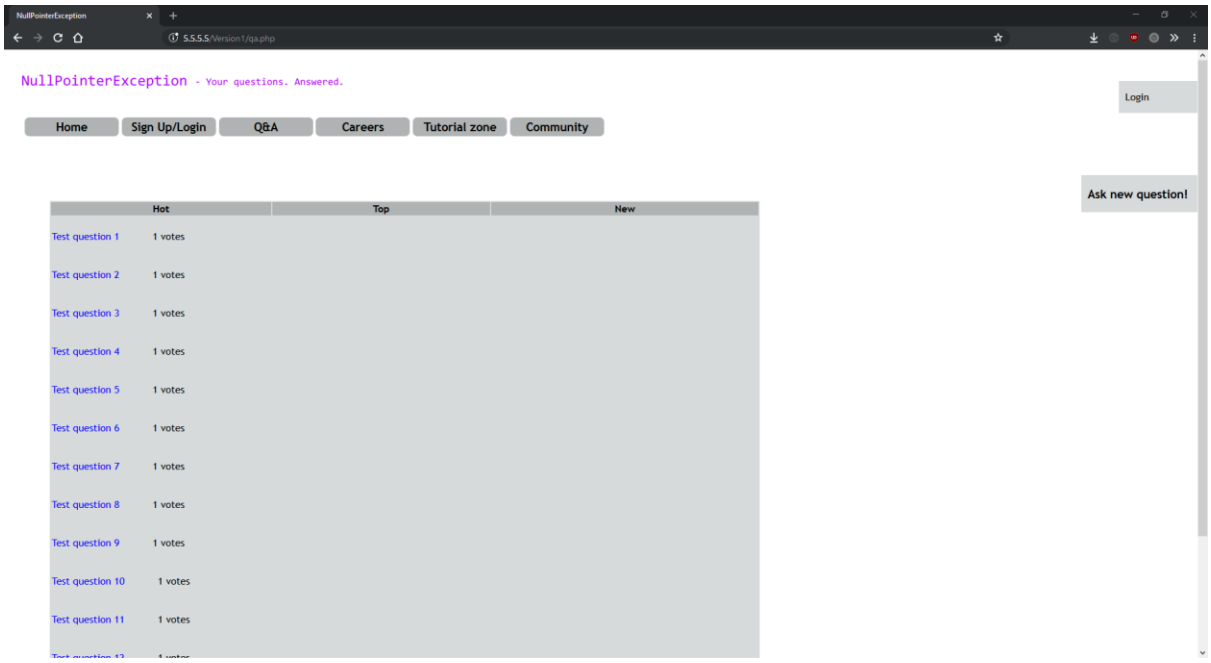


signup.php

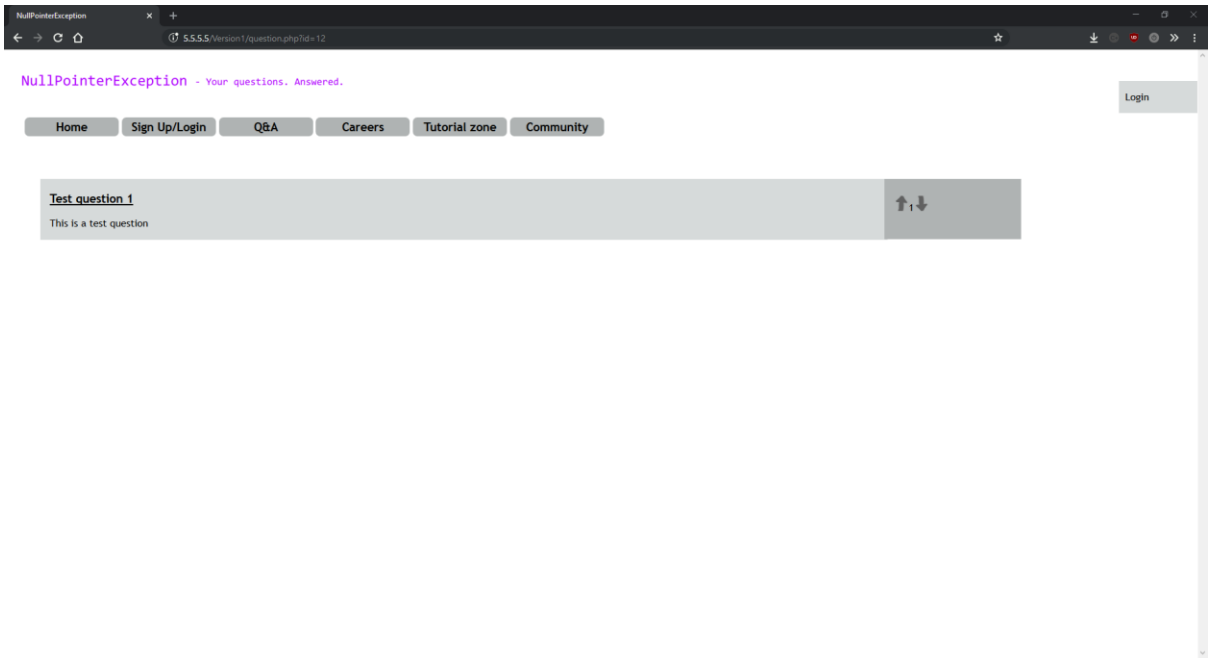


NullPointerException - Your questions. Answered.

qa.php

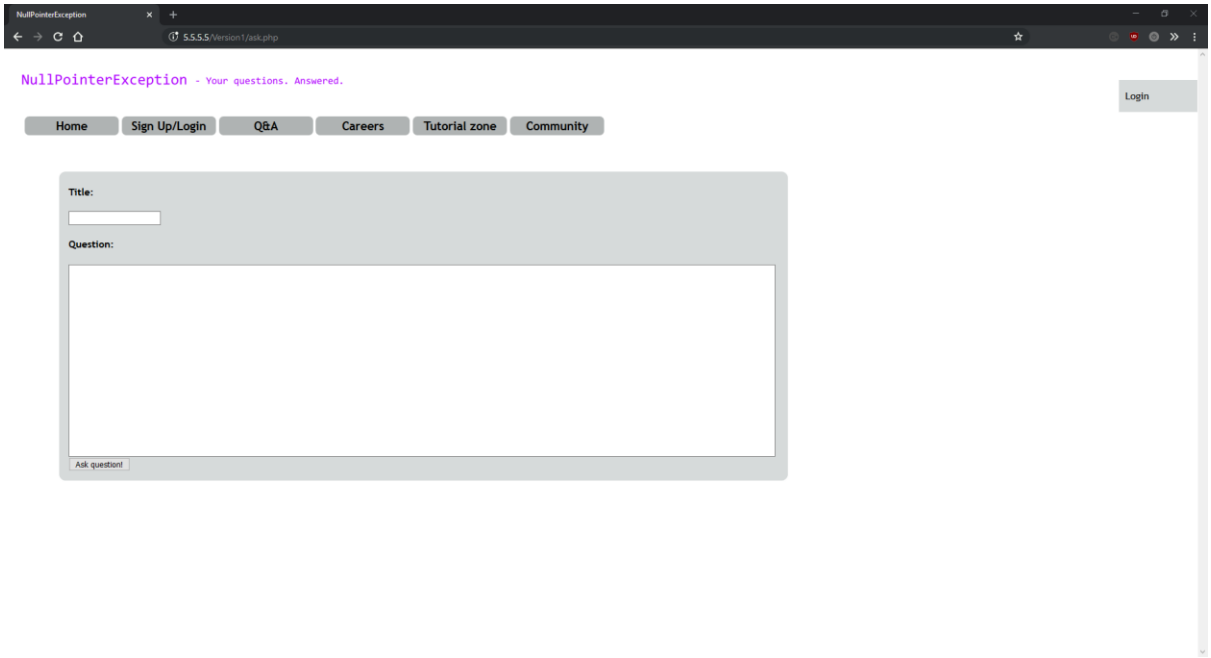


question.php



NullPointerException - Your questions. Answered.

ask.php:



Version 2

Version 2 of NullPointerException can also be referred to as rewrite 1. Over the course of my project I rewrote and redesigned the site between 3 and 4 times. This came from bad planning and I regret it because it cost me a lot of time, however the effect on the project was definitely worth it.

Version 2 was when I redesigned the entire site, using a CSS framework called bootstrap to do most of the heavy lifting. The front-end of the site hasn't changed at all since this redesign and I feel that it makes the site much more user friendly, which was one of my personal success criteria.

This version is made up from merging the front-end-design branch into the master branch. front-end-redesign was a git branch I made for converting the entire site to run on bootstrap. This process involved removing the stylesheet links from a page, replacing the custom classes and id's with bootstrap classes, and linking the bootstrap CSS and JS.

I also added two new pages to the site: tutorial.php and python3.php. tutorial.php is the landing page for the tutorial section of the website, and python3.php is the landing page for the Python 3 tutorials. Both of these pages are entirely written in HTML + bootstrap.

All the changes in this version are markup changes and adding/removing CSS classes, neither of which are suited to a write-up in the same style as the version one PHP scripts so I've linked the git diffs which show all the changes in this version.

https://github.com/OliRadlett/NullPointerException/commit/b0e8f8d133e447bb5acccc9c7c6994e7d2bb70c0

Version 2 testing:

Version 2 is tested using the same testing methodologies as version 1.

The results are shown below.

Browser tests:

Browser	**Works/Doesn't work**	**Notes**
Mozilla Firefox	Works	None
Google Chrome	Works	None
Internet Explorer 11	Works	None
Microsoft Edge	Unknown	Won't connect to website at all so impossible to test
Mozilla Firefox (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work, Logo is too wide
Google Chrome (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work, Logo is too wide

Basic functionality:

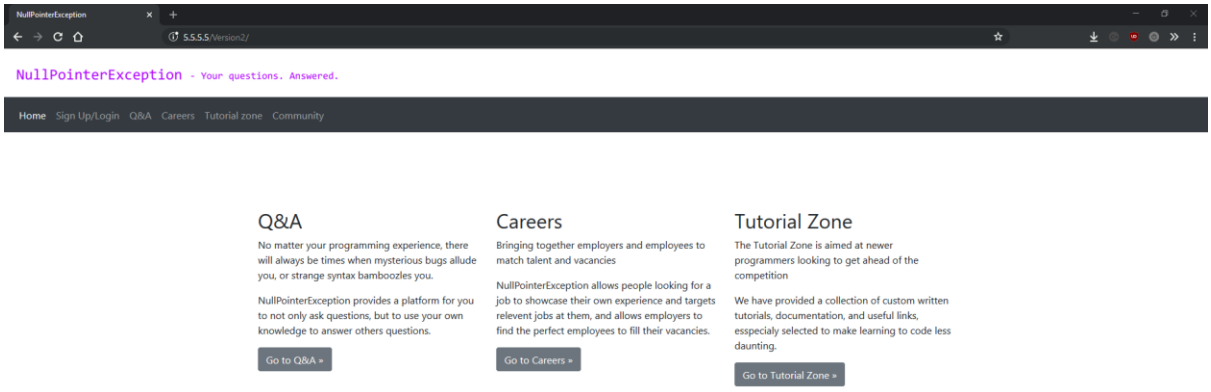
Page	**Test**	**Test data**	**Works/Doesn't work**	**Notes**
signup.php	Can create an account	All valid data	Works	Script creates an account using the data provided
signup.php	Can create an account	No data	Works	Script prompts user to fill in missing fields
signup.php	Can create an account	☺☺☺☺☺	Works	Script prompts the user to try again

login.php	Can log in to existing account	Username and password for an account existing in the database	Works	Script logs user into account
login.php	Can't log into account with wrong password	Username for an account that exists in the database, wrong password for said account	Works	Script prompts user that username/password combination is invalid
login.php	Can't log into account that doesn't exist	Username and password that doesn't exist in database	Works	Script prompts user that username/password combination is invalid

Non-working tests will be addressed in future versions of the project

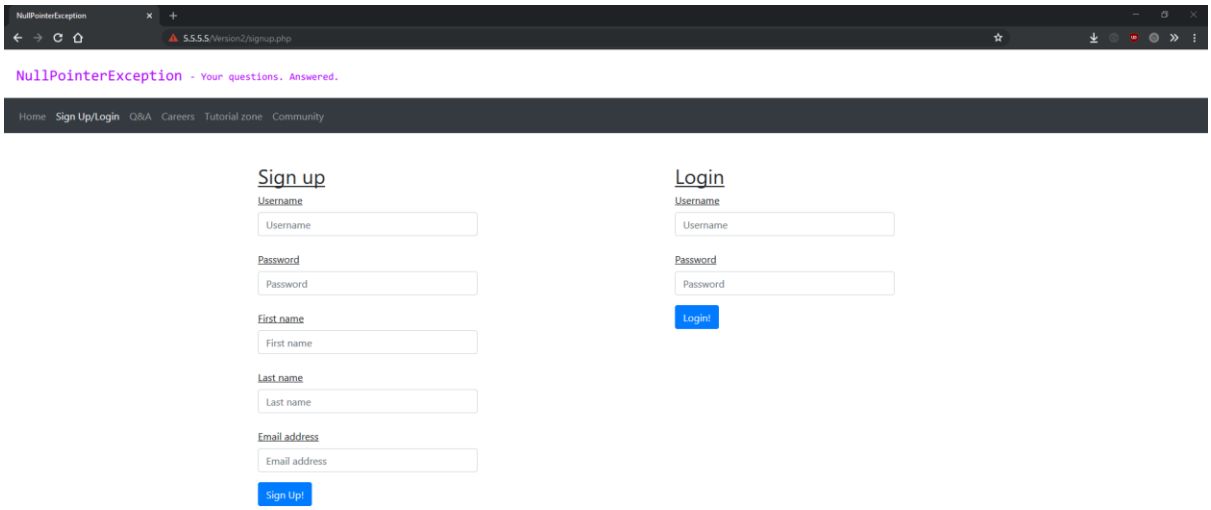
Version 2 screenshots:

index.php

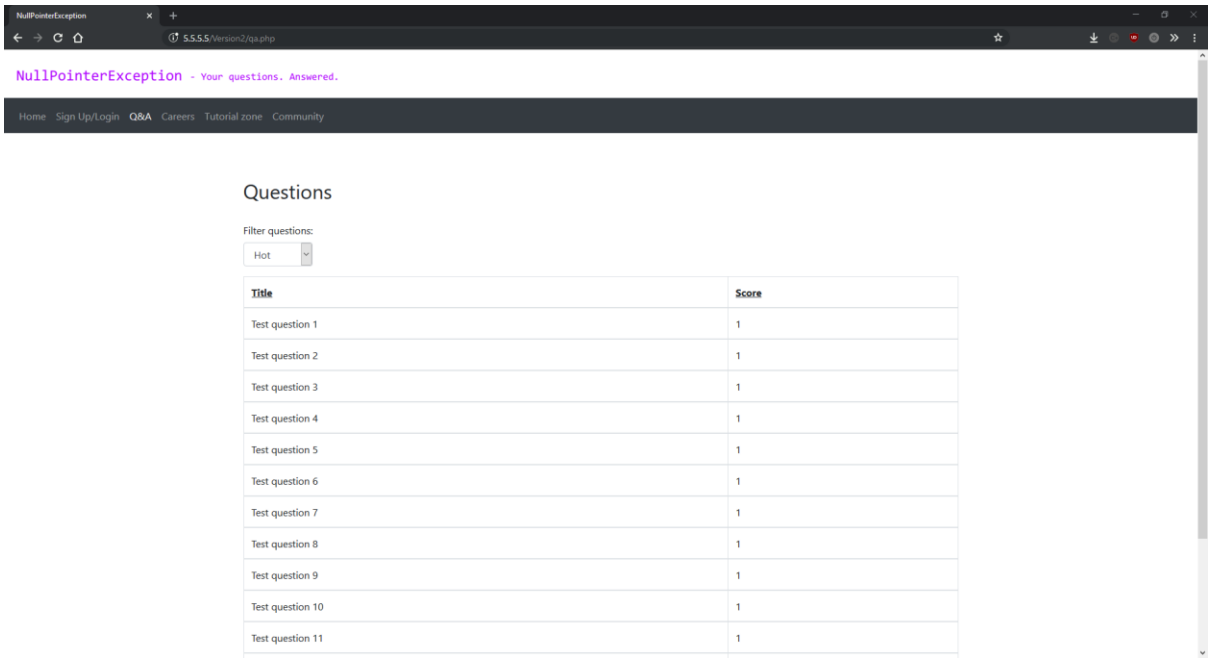


NullPointerException - Your questions. Answered.

signup.php

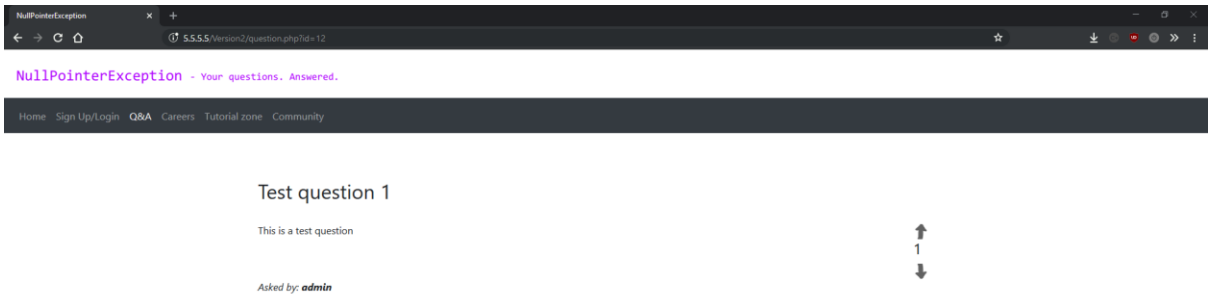


qa.php:

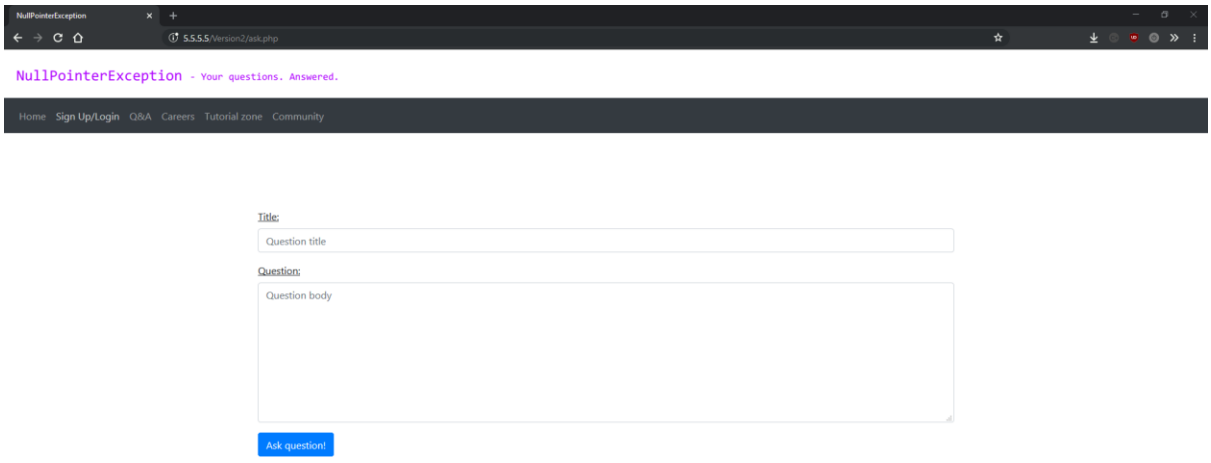


NullPointerException - Your questions. Answered.

question.php

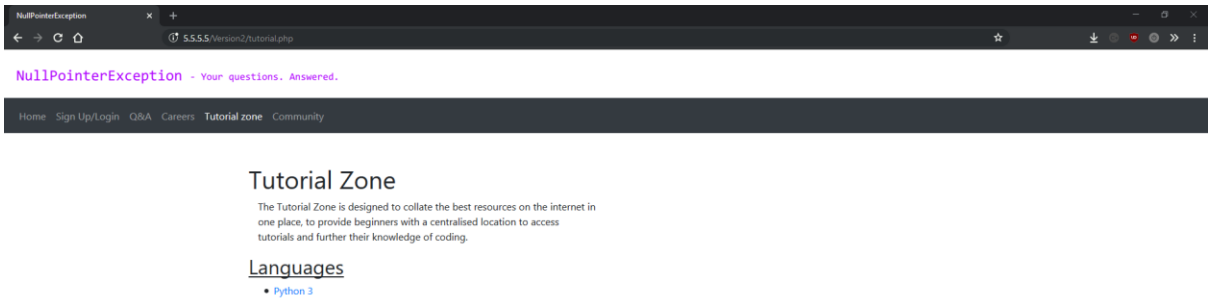


ask.php

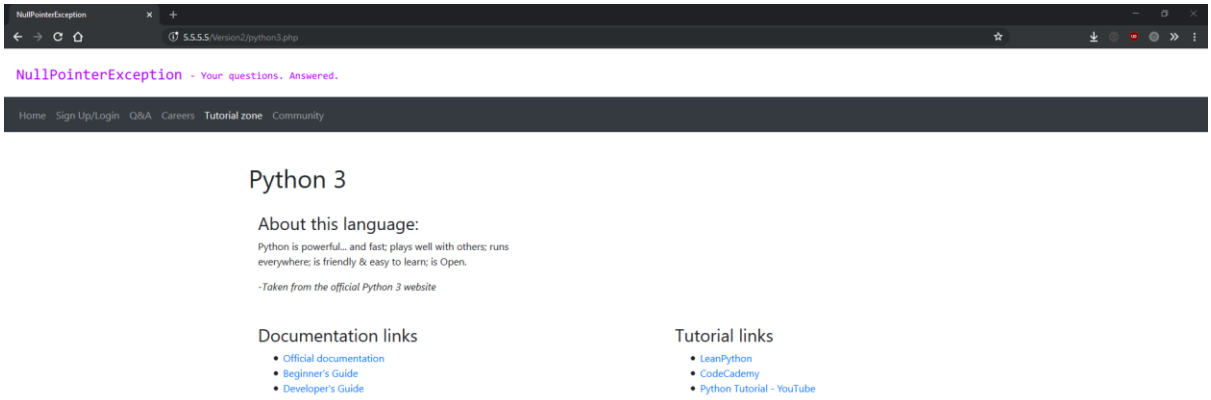


NullPointerException - Your questions. Answered.

tutorial.php



python3.php



Version 3

Version 3 was mainly bug fixes and adding some smaller features. When I first started my project I decided to split the development into 4 main blocks: the core site, the q&a system, the careers system and the tutorial zone. This allowed me to organise my time to finish one section before starting another – an approach which I mainly stuck to throughout the project and found very beneficial. Most of the work done in version 3 was to the core site, including adding favicons, a small JavaScript script to change to a more mobile friendly logo if viewed on a phone, bugfixing and other minor improvements.

Notable changes are shown below:

logo-fix.js:

A new script in this version, logo-fix.js is used to change the logo above the menu bar to a more narrow varient when the site viewed on a mobile device. This is because the desktop version of the logo is wider than the width of some phones in portrait mode. The script works by creating a function called checkLogo() which compares the width of the browser (from window.innerWidth) with a constant value of 576. This value was chosen because it was the size of the window that the styling broke using the old image. If the window width is smaller than 576 then the script changes the src attribute of the image element to the relative path of the mobile logo. If not the script does the same but uses the path for the desktop logo.

The script sets this function to run after two browser events: window.onload and window.onresize. window.onload is fired when the page has finished loading, and window.onresize is fired every time the browser window changes size.

***.php:**

All php scripts had the following added to the top of them:

session_start();

include connect.php;

The first of these two lines are used to connect the script to the PHP session in order to access SESSION_ cookies and data. This must be the first line in the PHP script in order to work properly.

The second line is used to include the connect.php script. This is where the functions to connect to the database are located, and is stored in a separate file and included to avoid writing the same duplicate code in every script. This line allows every script to connect and interact with the database.

***-OLD.php:**

Any file with the -OLD prefix was removed from the repository. -OLD files were the origin files from version one before the first rewrite. I kept them until this point because I wasn't fluent enough in git to trust myself to delete files and be able to recover them if necessary but by version 3 I felt comfortable to remove them.

Version 3 testing:

Version 3 is tested using the same testing methodologies as version 1 and 2.

The results are shown below.

Browser tests:

Browser	**Works/Doesn't work**	**Notes**
Mozilla Firefox	Works	None
Google Chrome	Works	None
Internet Explorer 11	Works	None
Microsoft Edge	Unknown	Won't connect to website at all so impossible to test
Mozilla Firefox (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work.
Google Chrome (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work

Basic functionality:

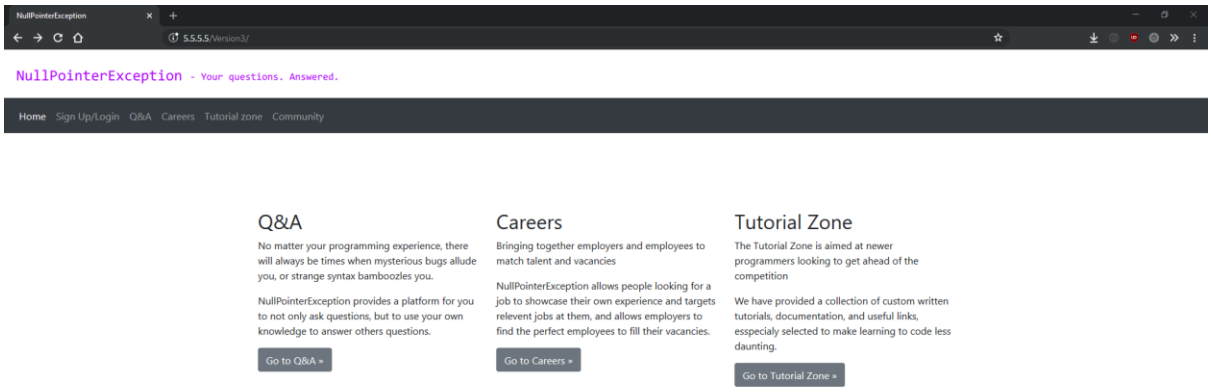
Page	**Test**	**Test data**	**Works/Doesn't work**	**Notes**
signup.php	Can create an account	All valid data	Works	Script creates an account using the data provided
signup.php	Can create an account	No data	Works	Script prompts user to fill in missing fields
signup.php	Can create an account	☺☺☺☺☺	Works	Script prompts the user to try again
login.php	Can log in to existing account	Username and password for an account existing in the database	Works	Script logs user into account
login.php	Can't log into account with wrong password	Username for an account that exists in the database, wrong	Works	Script prompts user that username/password combination is invalid

login.php	Can't log into account that doesn't exist	password for said account Username and password that doesn't exist in database	Works	Script prompts user that username/password combination is invalid

Non-working tests will be addressed in future versions of the project

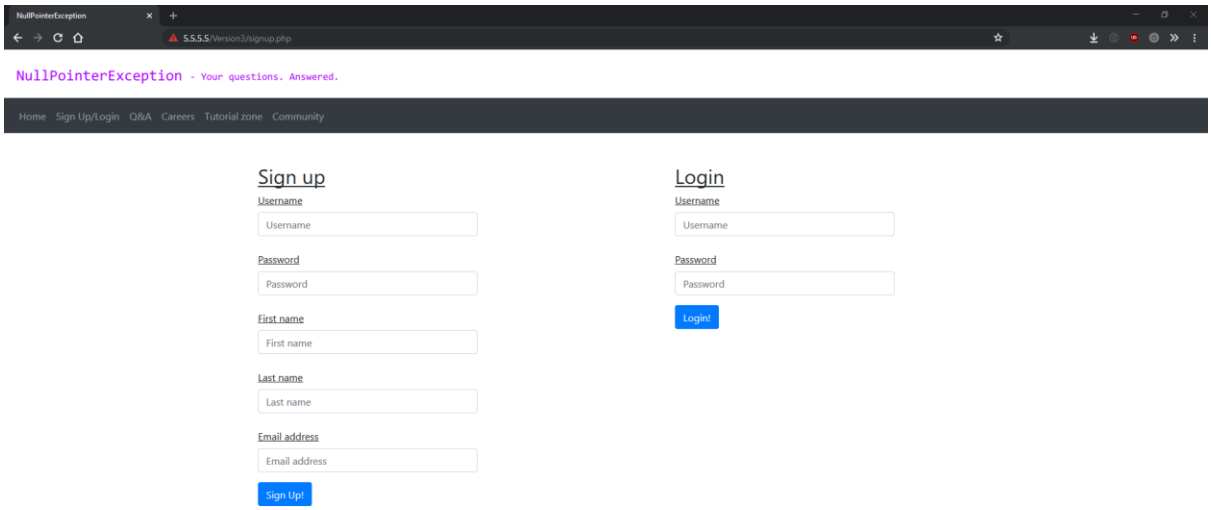
Version 3 screenshots:

index.php

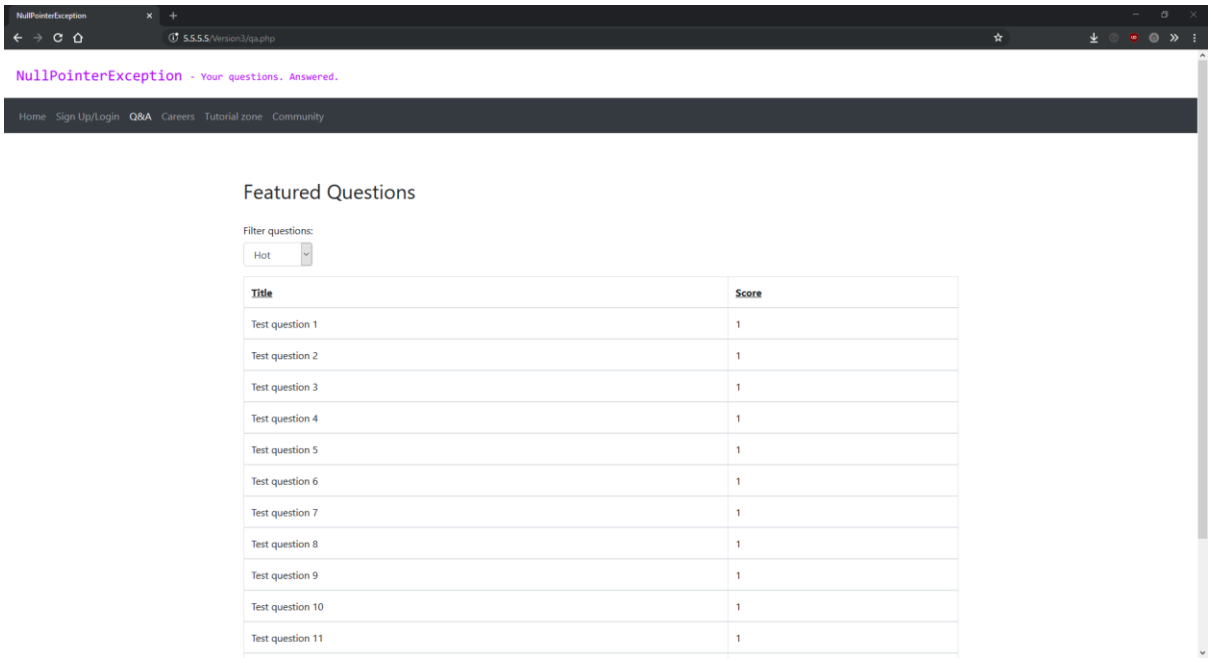


NullPointerException - Your questions. Answered.

signup.php

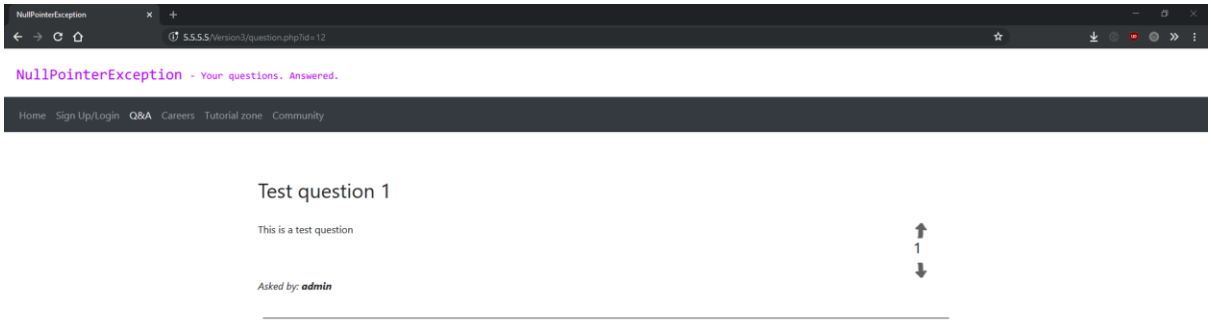


qa.php

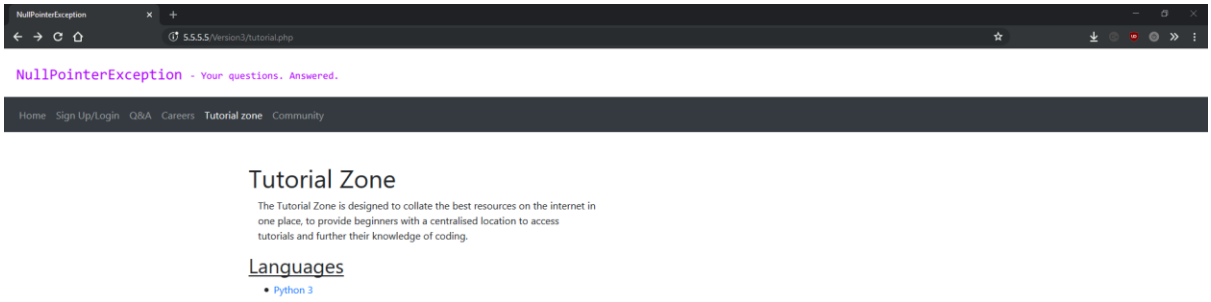


NullPointerException - Your questions. Answered.

question.php

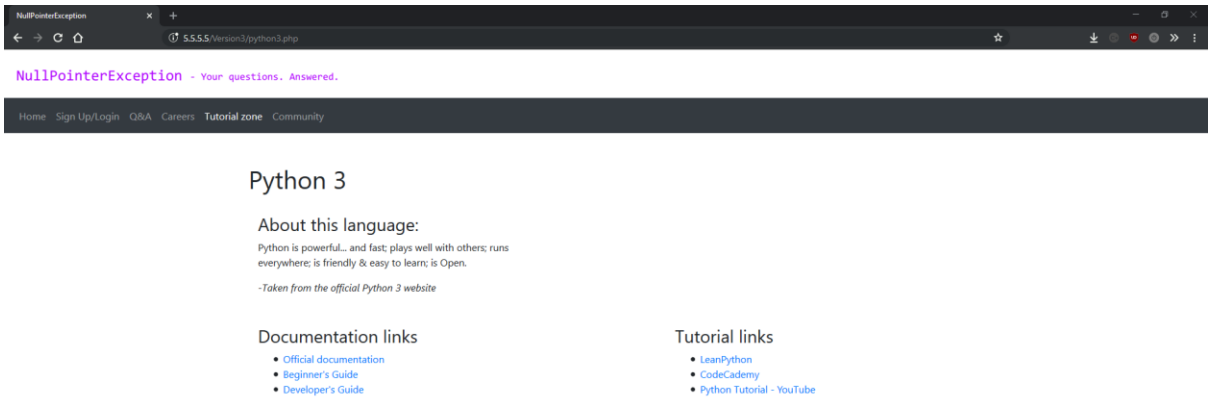


tutorial.php

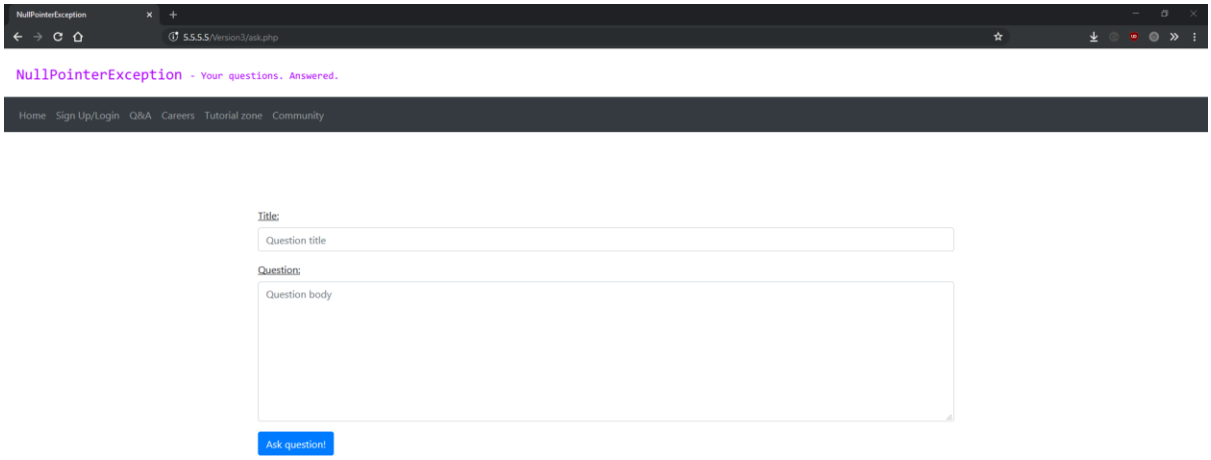


NullPointerException - Your questions. Answered.

python3



ask.php



NullPointerException - Your questions. Answered.

career.php

community.php

Version 4

Version 4 was re-write number 2. Whilst the first rewrite was focused on the front end, version 4 focused on the back-end. I mainly focused on breaking down all the large JavaScript and PHP scripts down into smaller functions, using a weird mix of procedural programming with lots of functions

which I now regret. However, this is one of the many things I learnt while doing this project and did help make the project more modular and easier to develop/maintain (in the short term).

Notable changes are shown below:

header.html:

A new file in this version, header.html contains the markup and bootstrap classes for the header shown at the top of every page on the website. This includes the logo and the menu bar with the links to all the pages, as well as linking a JavaScript script used to add the "active" bootstrap class to the menu bar link for whichever page is currently being shown. The theory behind this file is to remove duplicate code wherever possible, and all this code (with minor differences) was written in on every page which added to load times (before the browser caches it) and made the PHP scripts harder to read and maintain. It was always my intention to move this code to an external file but up until version 4 I didn't realise that PHP's include expression could work with non-PHP files, and what the behaviour of including markup would be. Luckily it simply writes out the contents of the HTML document to the browser at the point of the script where the include expression is located.

addActive.js:

addActive.js is a small script used to add the "active" bootstrap class to the header link for the page that the user is on. This used to be hardcoded to the header for each individual page, but since moving the header to one universal file, this approach no longer works. The script gets the current URL and uses it to determine which page the user is on. It then adds the active bootstrap class to the header link element for the correct page.

An interesting side effect of this script is that it stops any window.onload events from firing as they are taken up by this script. I deal with this by providing a space at the bottom of the main function in addActive.js for calling page specific onload functions.

All other changes in this version are minor bug fixes and structure changes to make the solution more modular and easier to read.

Version 4 was quite a time consuming version to write even though it didn't further the state of the project much. However, the backend changes implemented in this version helped speed up the development of future versions. Another unexpected benefit of making a more modular solution is that it's easier to document and write up. This is because all the changes are now in their own functions rather than spread out all over multiple files.

Version 4 testing:

Version 4 is tested using the same testing methodologies as version 1, 2 and 3.

The results are shown below.

NullPointerException - Your questions. Answered.

Browser tests:

Browser	**Works/Doesn't work**	**Notes**
Mozilla Firefox	Works	None
Google Chrome	Works	None
Internet Explorer 11	Works	None
Microsoft Edge	Unknown	Won't connect to website at all so impossible to test
Mozilla Firefox (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work.
Google Chrome (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work

Basic functionality:

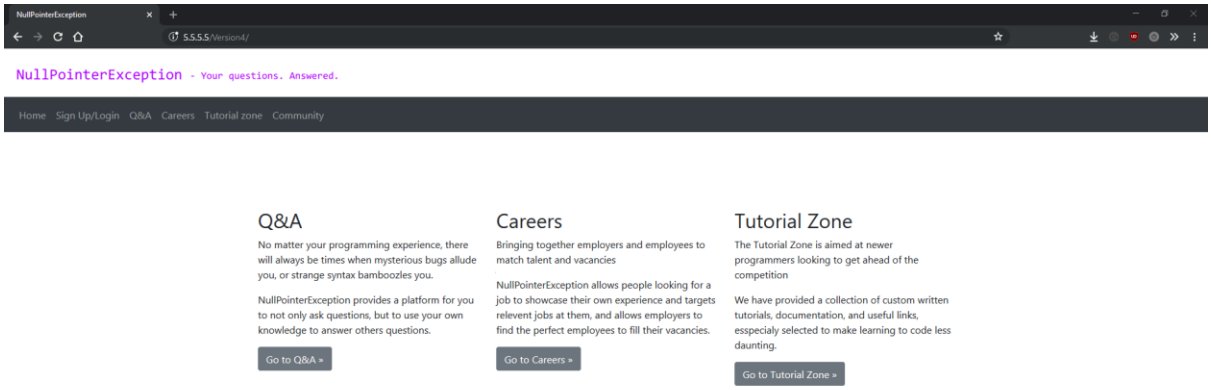
Page	**Test**	**Test data**	**Works/Doesn't work**	**Notes**
signup.php	Can create an account	All valid data	Works	Script creates an account using the data provided
signup.php	Can create an account	No data	Works	Script prompts user to fill in missing fields
signup.php	Can create an account	☺☺☺☺☺	Works	Script prompts the user to try again
login.php	Can log in to existing account	Username and password for an account existing in the database	Works	Script logs user into account
login.php	Can't log into account with wrong password	Username for an account that	Works	Script prompts user that

		exists in the database, wrong password for said account		username/password combination is invalid
login.php	Can't log into account that doesn't exist	Username and password that doesn't exist in database	Works	Script prompts user that username/password combination is invalid

Non-working tests will be addressed in future versions of the project

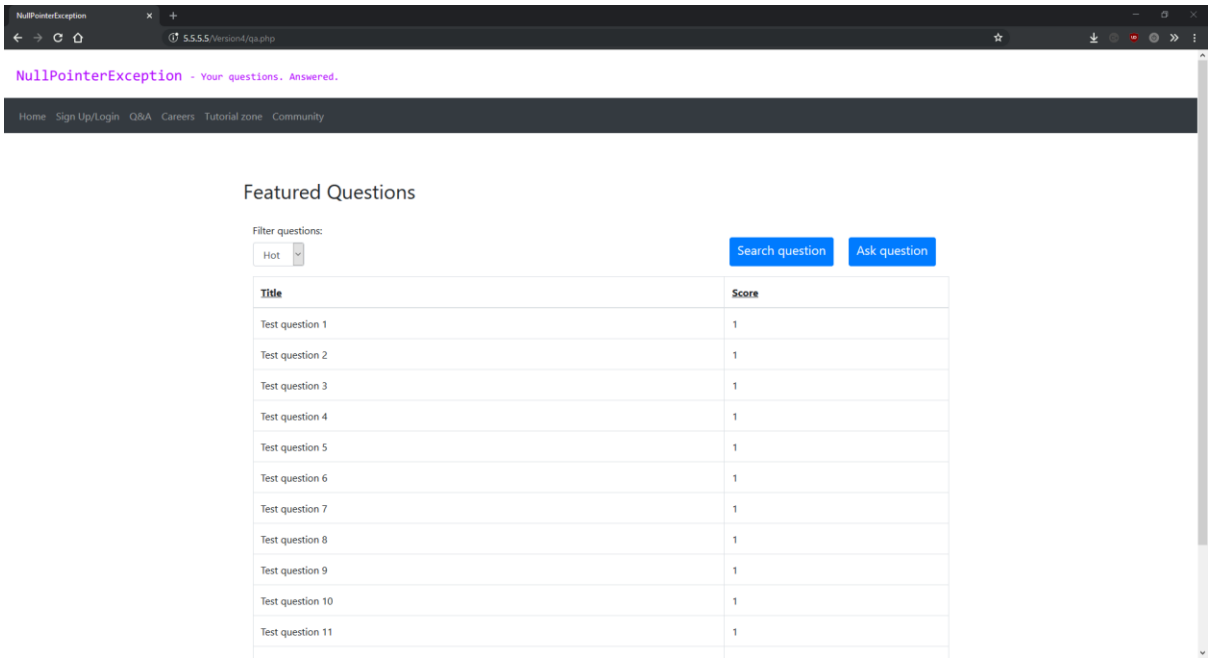
Version 4 screenshots:

index.php

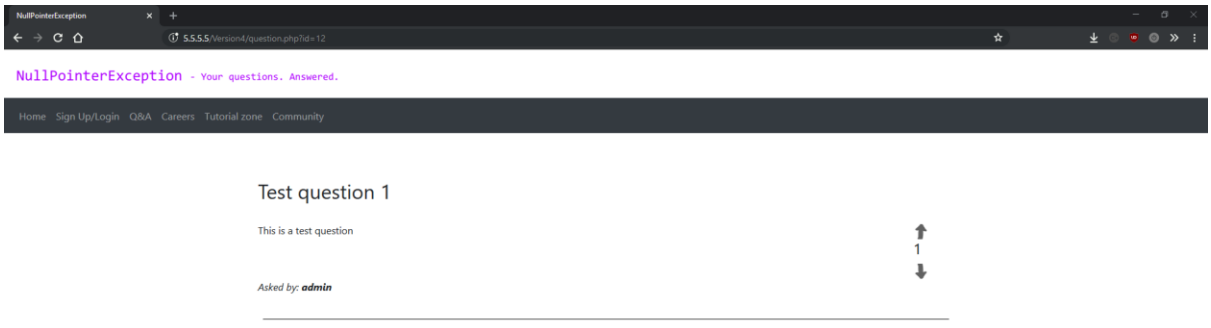


NullPointerException - Your questions. Answered.

qa.php

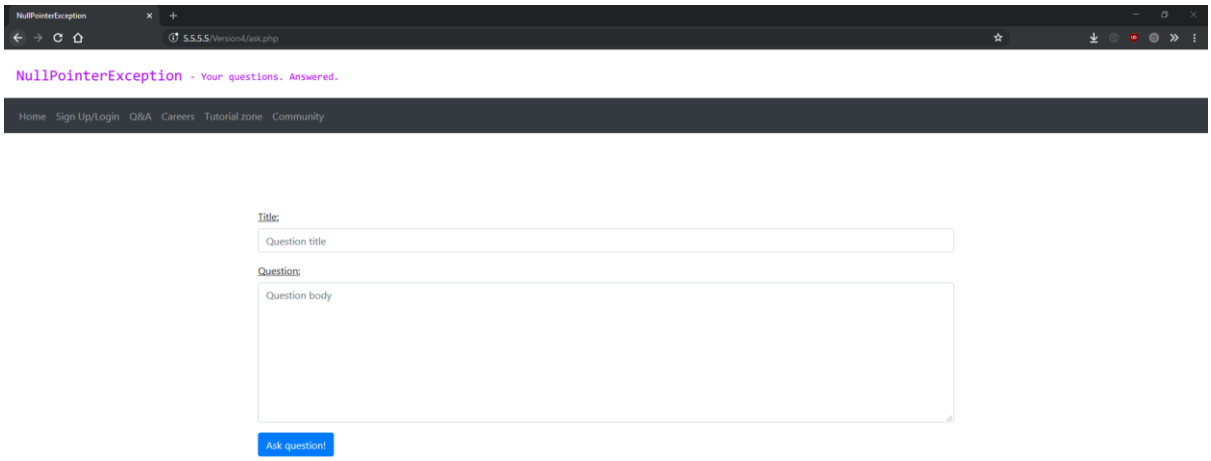


question.php

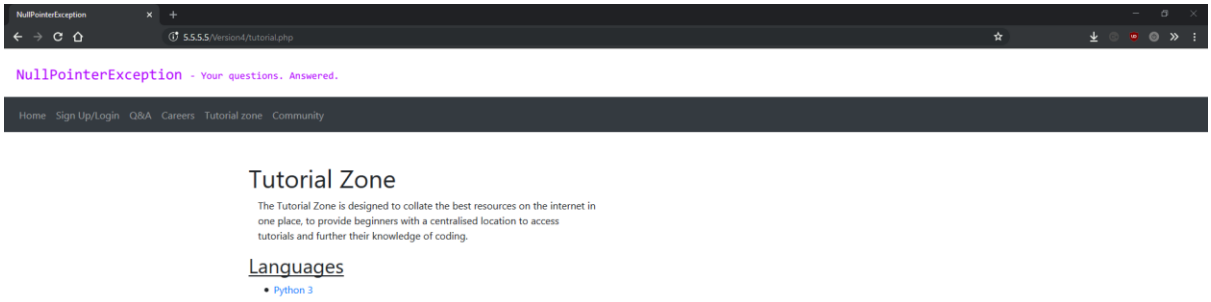


NullPointerException - Your questions. Answered.

ask.php

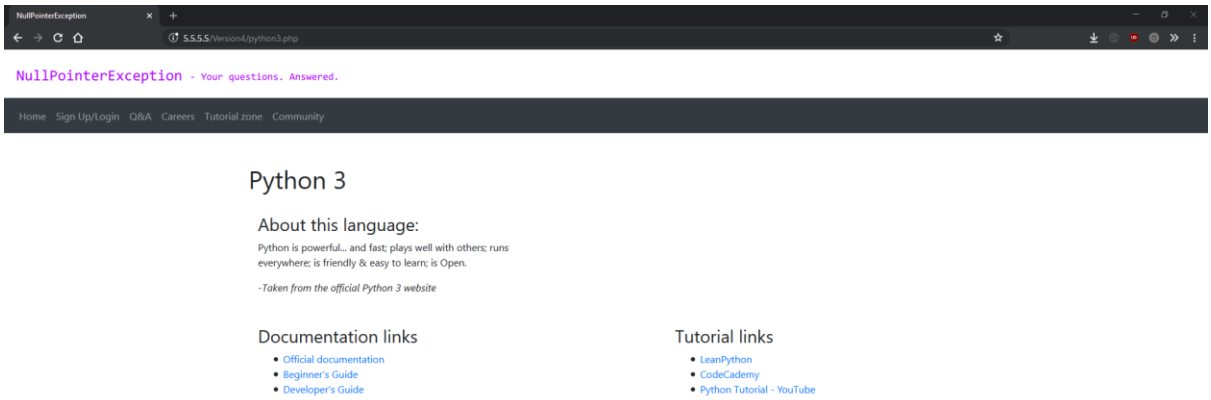


tutorial.php

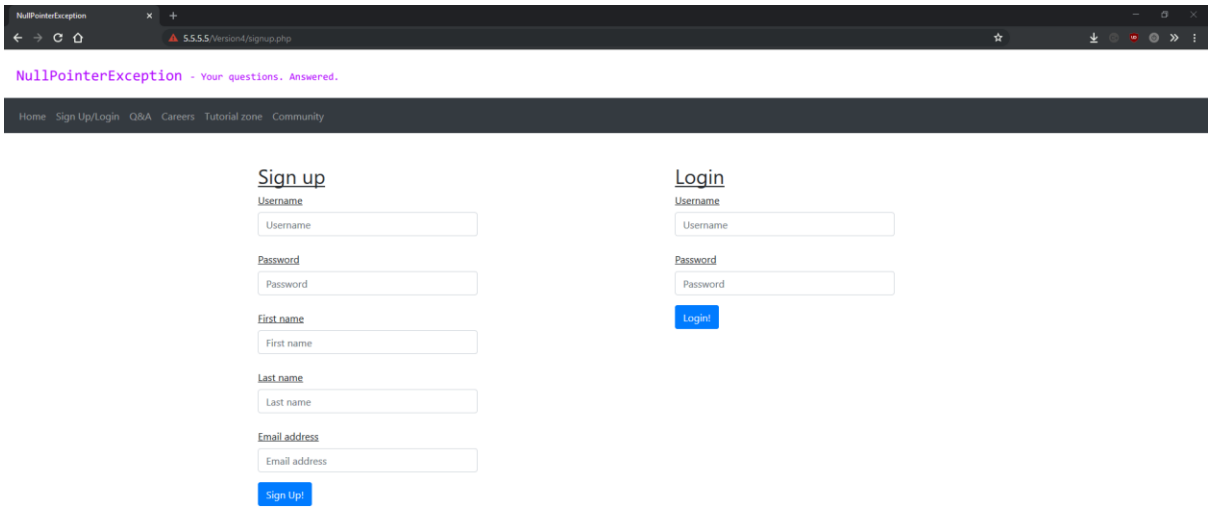


NullPointerException - Your questions. Answered.

python3.php



signup.php



NullPointerException - Your questions. Answered.

community.php

career.php

Version 5

Version 5 added a new feature for searching questions, the initial commenting system, a “404” page, along with several enhancements and bug fixes. This is the most new features added in a single version since the initial upload of the project to GitHub.

One minor change that had a decent impact on the scalability of the project was adding the String "LIMIT 10" to the MySQL queries on qa.php. This does exactly what you'd expect – limits the number of results that can be returned from the database to 10. This saves on database processing power, and saves on network requests from the computer to the server and back again. I made this change because the page started to look silly when the database had over 10 questions in (during testing), and limiting the query was a much better solution that only displaying 10 for the reasons mentions above.

A potential fix to an irritating bug I discovered after version 4 – where session variables seemingly randomly disappeared mid-session – was the function session_write_close(). I make a call to this function after any session functions and just before any calls to the header() function. This function is used to save the state of the session, which normally happens anyway, but when session function were used in conjunction with setting the headers to redirect the user, it started causing errors. Adding session_write_close() seemed to fix this behaviour.

Full details of script specific changes are below:

downloadquestions.php:

Added "LIMIT 10" to all three of the MySQL queries on the downloadquestions.php. This is used to reduce load on the database server and to limit bandwidth used between the database server and the users computer. This was a very quick fix that provided very good improvements.

downloadsearchq.php:

A new file added in this version, downloadsearchq.php is the search system's equivalent to downloadquestions.php. Its purpose is to search the database for questions based off a provided query and output the results in a format that the JavaScript script that initiates the search and receives the results can understand.

The script works by first getting the search query from a URL parameter, then preparing a MySQL query to search the database, using the "LIKE" keyword. "LIKE" in MySQL works by only returning records that match a certain pattern. The pattern I'm using for NullPointerException is "%{$q}%". Whist this seems confusing at first, its actually quite simple. The % signs are used to represent any number (including zero) characters – meaning that the main patter can occur anywhere within the record. The {$q} section of the pattern represents the PHP variable $q, which is used to store the search query the user entered.

The rest of the script simply executes the query, and outputs it in the form expected by the JavaScript script initiating the search.

filterSearchQuestions.js:

Another new script in this version, this script is an almost identical copy of filterQuestions.js and is used for initiating searches and outputting the results. Most of the code in this file has already been explained in version 1.

One issue with this script is that it involves duplicate code – something I always to try to avoid while programming. However, at this stage of development I was behind my personal schedule for this project – partly due to the complete re-writes. This means that I was prepared to use the quick fix of duplicating a JavaScript file and changing some of the functions at this stage, instead of combining the two similar scripts into one universal multi-use master script. However I hope to fix this in a future release.

questionFuncs.php:

I added three more functions in questionFuncs.php in this version: GetComments, Comment and getUserID. As their names suggests, these functions are all part of the new comment system which was added in this version. These are helper functions added in this separate file to keep the main questions.php file free from clutter.

The first, GetComments, takes two parameters: $qID (an integer representing the question id in the database), and $connection (a reference to the MySQLi connection object). This function is used to return all the comments that have been made about a specific question. This is a very simple function because it relies on a second new function in this class for most of the heavy lifting. GetComments works by preparing a simple MySQL function for select all the comments from the database that have the question id attribute equal to the one provided in an argument to the function. Then it iterates over all the results returned and calls the Comment function for every result.

The Comment function is used to output a comment returned from the database. It takes two parameters: $row (An associative array comprised of one result row returned from the database) and $connection (a reference to the MySQLi connection object). It then uses the echo expression to output HTML content onto the page containing a mixture of elements, bootstrap classes and PHP variables. This, combined with some custom CSS, outputs a comment onto the question page.

The final function is not part of the new comment system, but is more of a quality of life improvement. Its called getUserID and it takes two parameters: $connection (a reference to the MySQLi connection object), and $username (A String containing a username to return the id of). The function simple queries the database to return the user id attribute for the user with the username matching the username parameter passed into the function. It then returns that value.

processcomment.php:

The final new (non-markup) file in this version, processcomment.php – the back-end system for posting a new comment for a question. The script starts off by connecting to the database, getting the attributes for the database from the URL, POST data, SESSION data, and the build in time() function (used for returning the current UNIX timestamp). It then checks the user hasn't been banned from the database, and if not it uploads it into the database (assuming that the comment body is not empty).

Version 5 was a large release with new features which work well and some small big fixes and minor improvements. It showed some more of the benefits of the modular design I've been employing by having a separate script for functions relating to the questions. It was quite an easy version to develop as the new features simply needed programming, before they could be slotted in – again thanks to the modular approach I've been using during development so far.

The rest of the work in this version was on bug fixing and minor improvements.

Version 5 testing:

Version 5 is tested using the same testing methodologies as version 1, 2, 3 and 4.

The results are shown below.

Browser tests:

Browser	**Works/Doesn't work**	**Notes**
Mozilla Firefox	Works	None
Google Chrome	Works	None
Internet Explorer 11	Works	None
Microsoft Edge	Unknown	Won't connect to website at all so impossible to test
Mozilla Firefox (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work.
Google Chrome (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work

Basic functionality:

Page	**Test**	**Test data**	**Works/Doesn't work**	**Notes**
signup.php	Can create an account	All valid data	Works	Script creates an account using the data provided

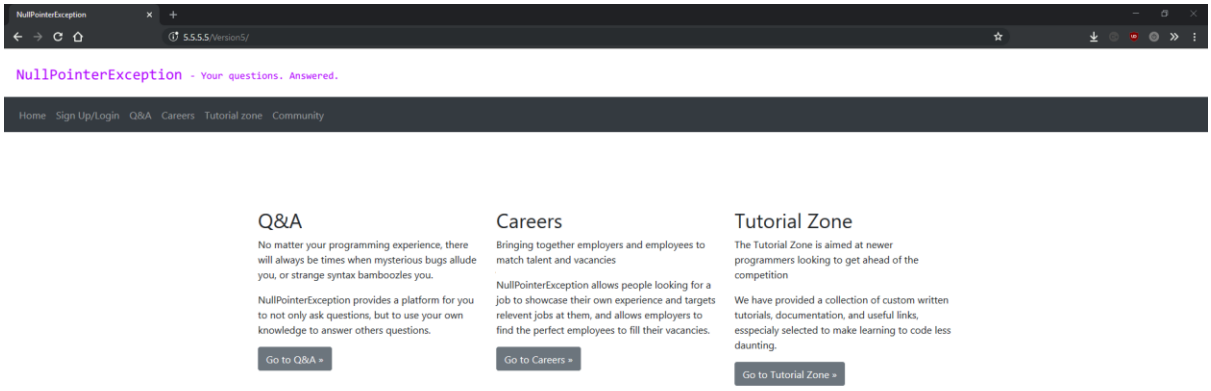
signup.php	Can create an account	No data	Works	Script prompts user to fill in missing fields
signup.php	Can create an account	☺☺☺☺☺	Works	Script prompts the user to try again
login.php	Can log in to existing account	Username and password for an account existing in the database	Works	Script logs user into account
login.php	Can't log into account with wrong password	Username for an account that exists in the database, wrong password for said account	Works	Script prompts user that username/password combination is invalid
login.php	Can't log into account that doesn't exist	Username and password that doesn't exist in database	Works	Script prompts user that username/password combination is invalid
ask.php	Can ask a question with valid inputs	Valid question title and question body	Works	Script adds a new question to the database
ask.php	Can't ask a question with nothing filled in	No data	Works	Script prompts the user to fill in every field
ask.php	Can ask a question containing unexpected characters	☺☺☺☺☺	Works	Script adds a new question to the database
searchq.php	Can search for a question with valid inputs	Valid search String	Works	Script shows all questions that match the search query
searchq.php	Can't search for a question without supplying any query	No data	Works	Script prompts the user to search for questions

searchq.php	Can search for a question containing unexpected characters	☺☺☺☺☺	Works	Script shows all questions that match the search query
comment.php	Can comment on a question using valid input	Valid input comprised on standard characters	Works	Script adds a new comment to the database
comment.php	Can't comment on a question without supplying a comment	No data	Works	Script prompts the user to input a comment
comment.php	Can comment on a question using unexpected input	☺☺☺☺☺	Works	Script adds a new comment to the database

Non-working tests will be addressed in future versions of the project

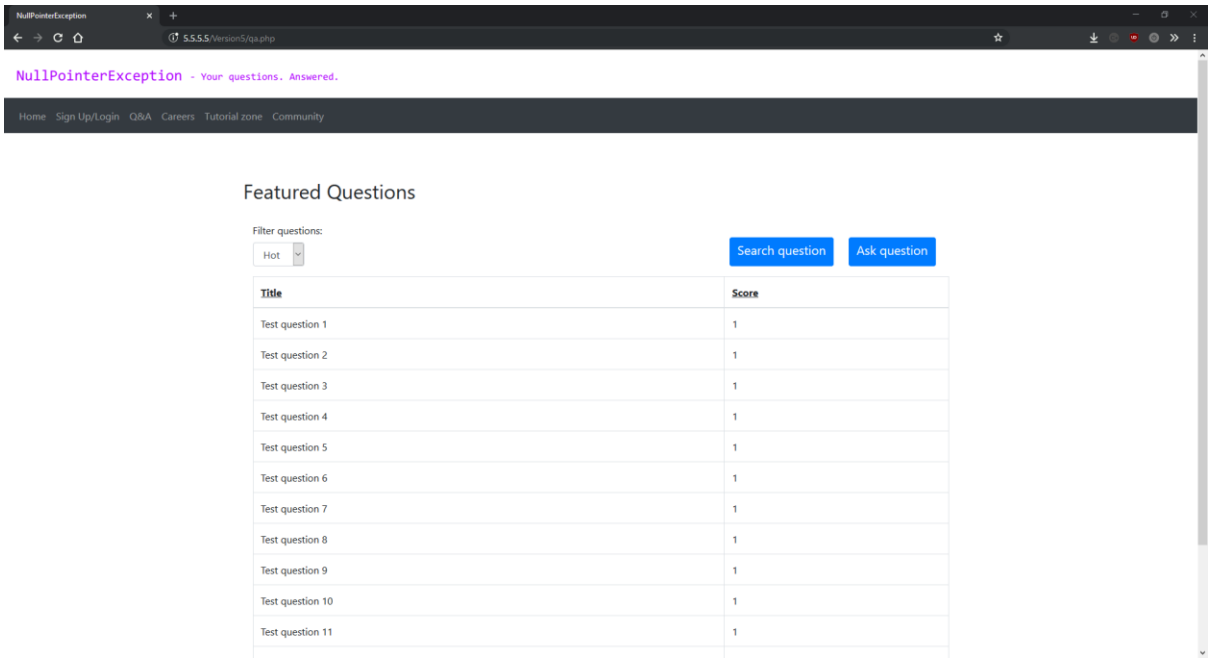
Version 5 screenshots:

index.php

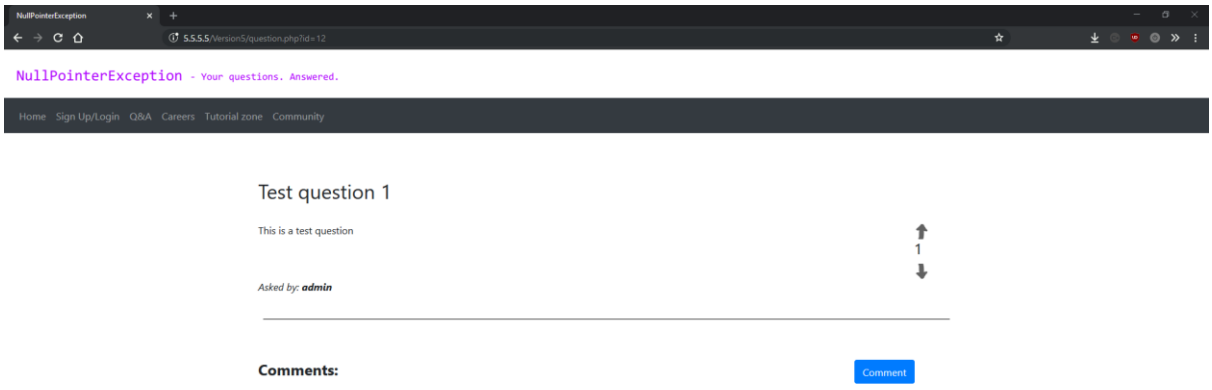


NullPointerException - Your questions. Answered.

qa.php

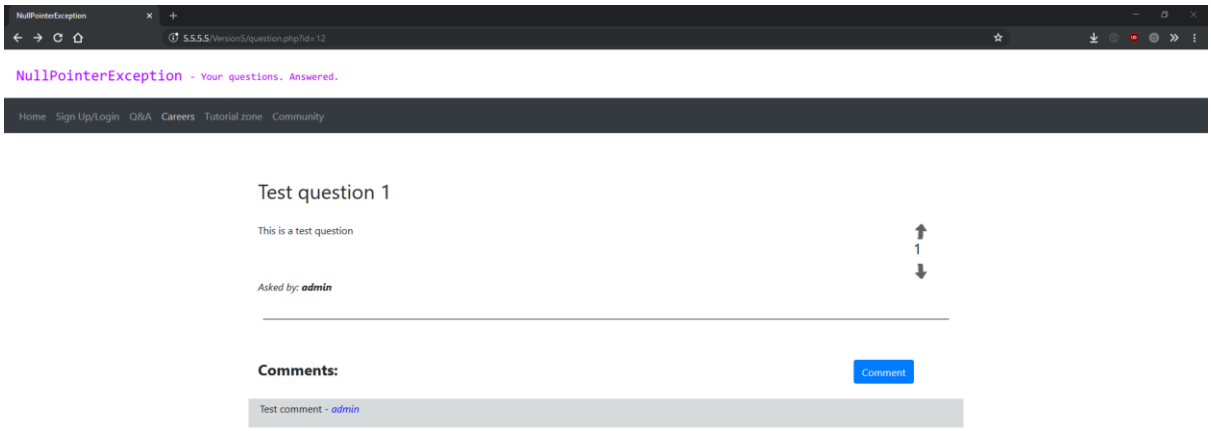


question.php

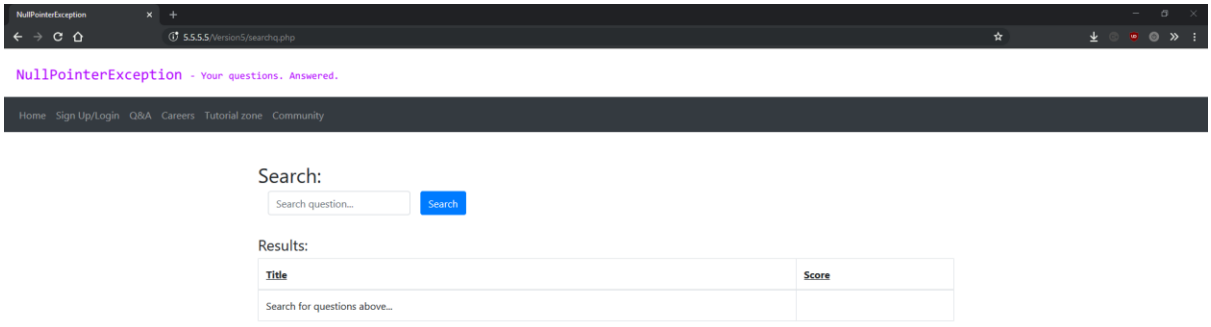


NullPointerException - Your questions. Answered.

question.php (with comment)

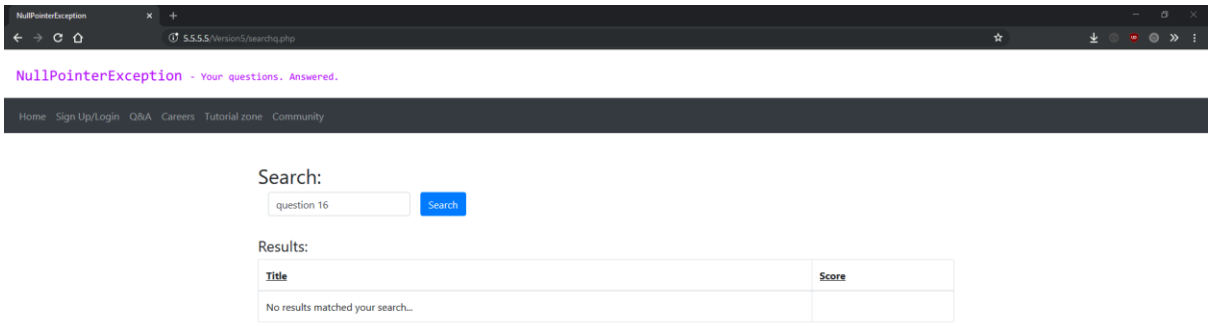


search.php

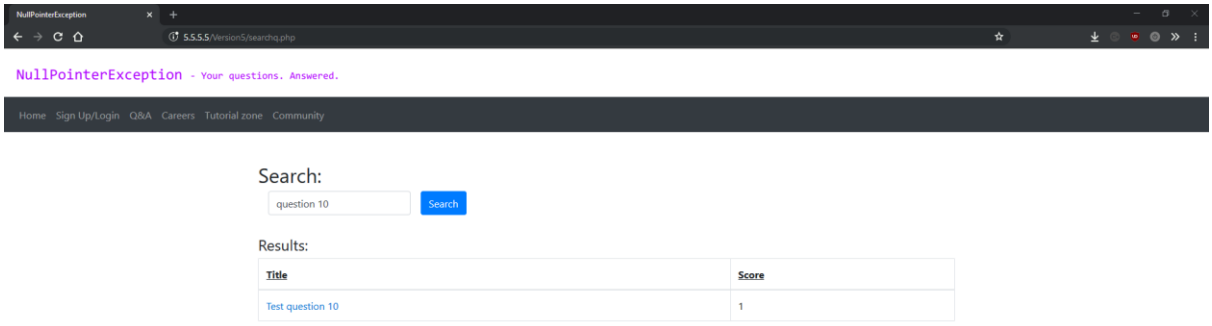


NullPointerException - Your questions. Answered.

search.php (no results)

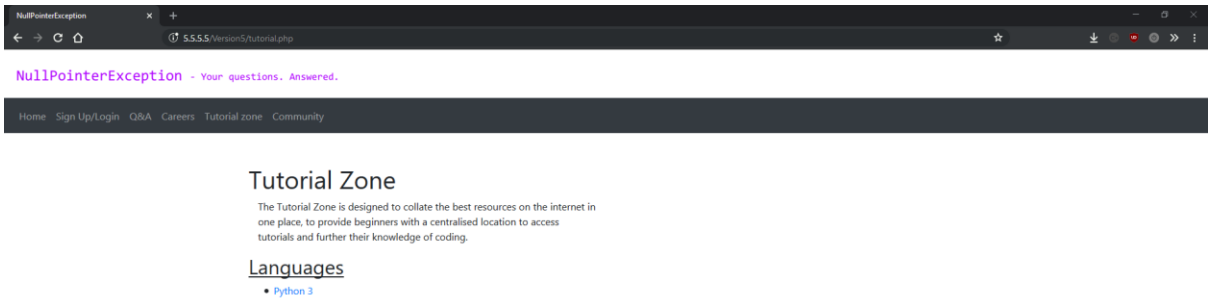


search.php (with results)

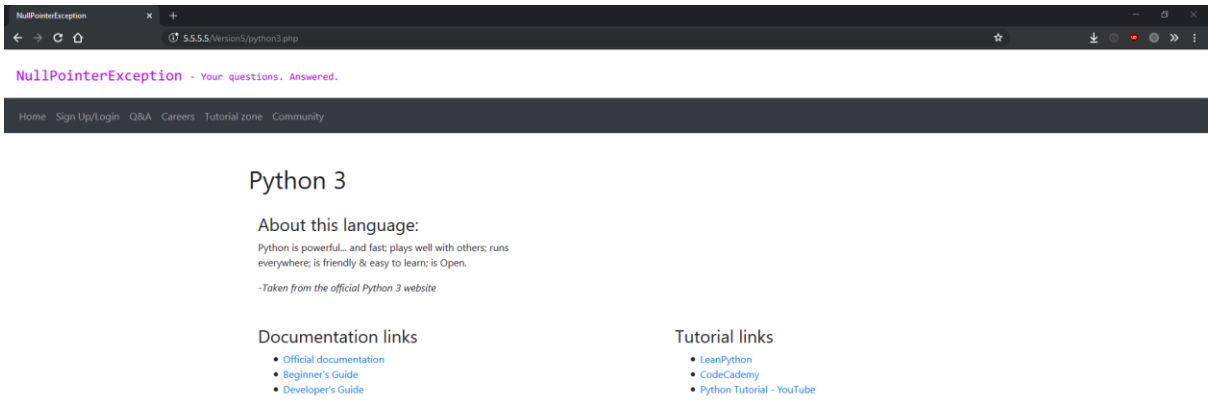


NullPointerException - Your questions. Answered.

tutorial.php

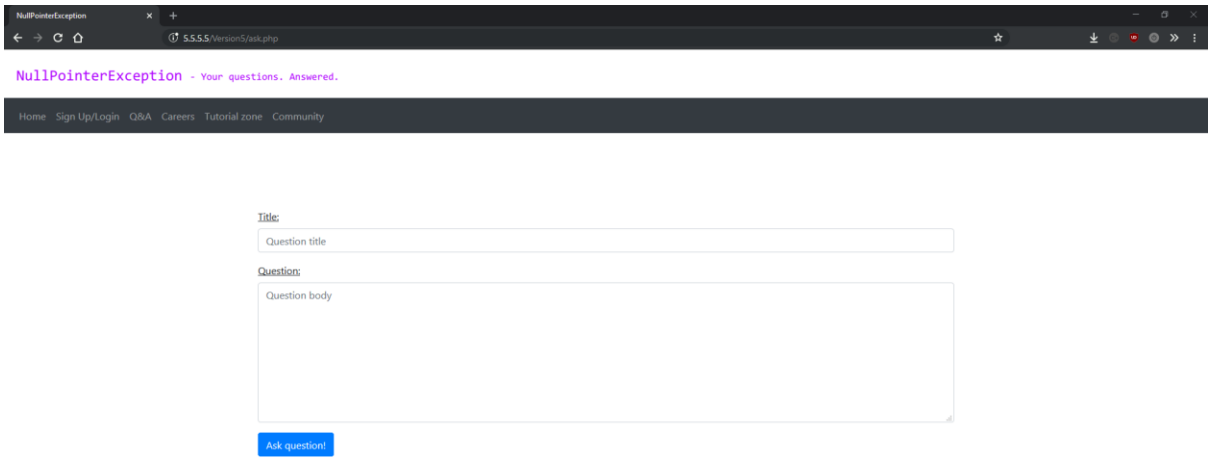


python3.php

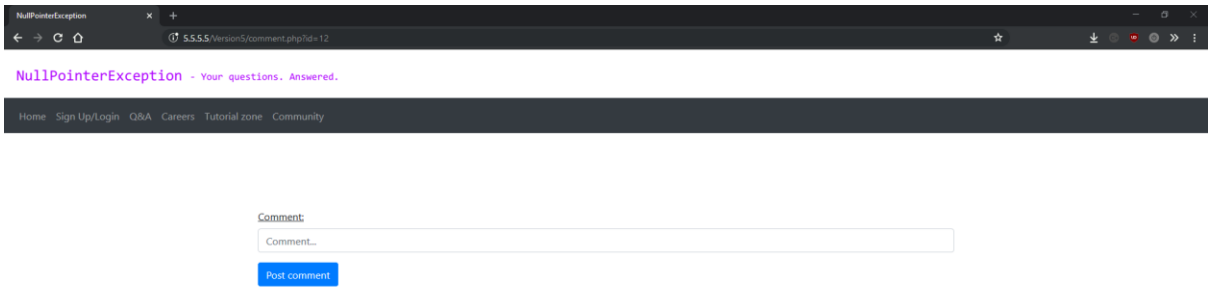


NullPointerException - Your questions. Answered.

ask.php



comment.php



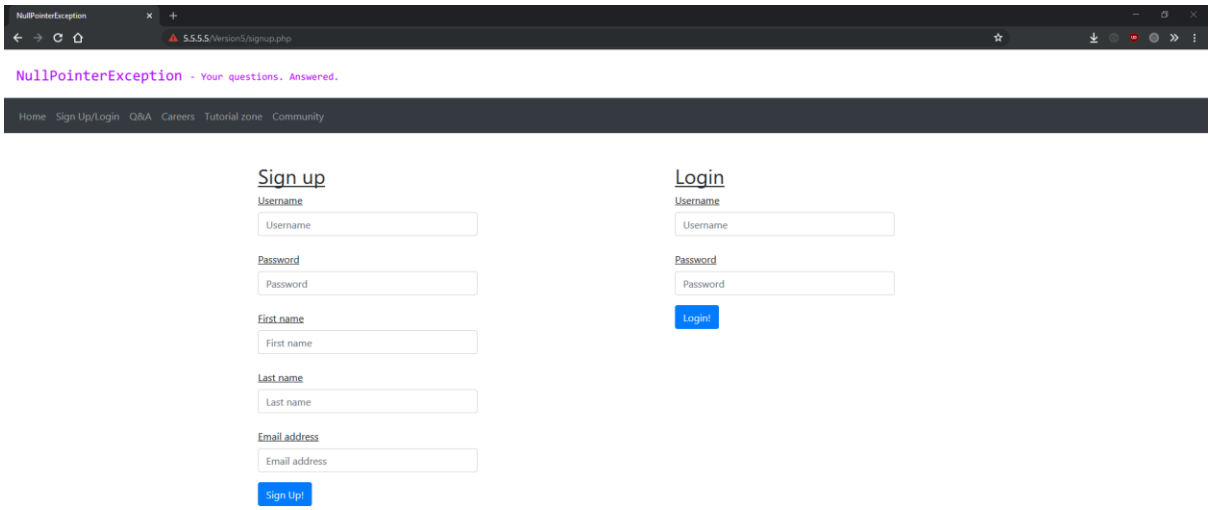
NullPointerException - Your questions. Answered.

community.php

career.php

NullPointerException - Your questions. Answered.

signup.php



Version 6

Version 6 is the last commit before the final re-write of the project. It adds lots of bug fixes, an edit comment system, question voting and code markdown using a CSS library called prism. Code markdown is where the use can embed code into their questions and when they are displayed on the website they will appear in a special block, and have syntax highlighting unique for whatever language the code is written in. This was a very hard part of the website to write because I needed to parse the question from the database to detect when code appears and stops appearing.

The code markdown system works by detecting the start and finish points of every code block in a question before outputting the question on to the page, then using some new functions added to questionFuncs.php to output the HTML tags and prism classes needed to render the code markdown. An interesting issue that arose during this commit was that the actual code you could put in the code markdown could cause security and cosmetic issues on the page. This was fixed by using the built in PHP function “htmlspecialchars()” when outputting the code for the markdown. This function is used to convert special characters to HTML entities which no longer cause any problems.

Changes to individual files can be seen below:

question.php:

Part of the code markdown system, this version changes the way that the question is outputted. Instead of simply using the “echo” keyword on the String contents of the question, the script now splits the question String by line breaks into an array. It the iterates over the array and detects the start and finish of any code blocks and calls the StartCodeBlock() and EndCodeBlock() functions

respectively. These are new functioned added to questionFuncs.php in this version. As intended, most of the heavy lifting is done by the functions in questionFuncs.php for simplicity.

questionFuncs.php:

I added 5 new functions to questionFuncs.php. The first three are for the code markdown system and the second two are for the edit comment system. At this stage of development, you can clearly see the benefit of my modular design – keeping as much of the work done as "modules" in this script and not taking up extra space across lots of other files.

The three functions for the code markdown system are: SplitLines(), StartCodeBlock() and EndCodeBlock(). SplitLines() is the most complicated of the three, and takes one parameter: $questionArray – A PHP array containing each line of the question split up by each newline (\n). First it creates 2 blank arrays – one for text and one for lines of code markdown. It then iterates over the question array and puts code block lines in the code block lines array, and normal question lines in the normal array. Finally the function returns an array comprised of the two arrays created at the start of the function.

The other two new code markdown functions added in this release are called StartCodeBlock and EndCodeBlock. These are very simple functions used to output the HTML tags and prism CSS classes needed to render the code markdown. Prism is a CSS / JavaScript library used for highlighting source code. You can customise it for whatever styles and languages you want, and it consists of one JavaScript script and one CSS document.

The final two new functions added to this script in this release are called isUsersComment() and GetComments() and both do exactly what their respective titles suggest. The first, isUsersComment() is used to check if a comment was made by a specific user or not. This is used to display the "edit comment" button next to a comment if the user that is logged in made it. It takes three parameters: $connection (a reference to the MySQLi connection object), $username (A string containing a username) and $id (An integer representing the comment to check). It then runs and executes a MySQL query to select the author attribute about a comment where the comment's id attribute is the same as the id argument in the function and the author attribute of the comment is the same as the author argument in the function. The function then returns either true/false depending on whether the author wrote the comment in question.

The last new function in this version, GetComments(), takes two parameters: $connection (a reference to the MySQLi connection object) and $id (An integer representing the comment id of the comment to retrieve from the database). The function works by selecting the comment with the id matching the id argument in the function. It then returns the comment fetched from the database.

modifyvote.php:

This is another new script added in this version, modifyvote.php is designed to run PHP functions from some JavaScript front-end in order to vote on questions. The type of vote is given as a URL parameter when the JavaScript front-end makes a request to the script. Out of all the scripts in the project, the question voting system is the least reliable and the thing I'm least proud of. However, it does what it's supposed to do and the feature enhances the site.

It works by using a switch case block to run the correct function from the URL parameter, then running separate MySQL statements depending on the function given in the URL. These include SELECT, INSERT INTO, DELETE and MODIFY MySQL statements, each in their own function in the script. One of the main reasons I'm not proud of this system is that the URL provides a direct interface with the database, with no passwords or any other security. This is a ridiculously unsafe thing to do in a production app and I'm only using this approach in my project because I was very short on time and it only directly exposes voting functions (however even this could bigger issues such as DoS attacks etc.). If I had more time to finish my project (which was so much larger than I expected), this would be one of the first things I'd look at.

downloadquestionvotes.php:

Another new script added in this version, also part of the question voting system, downloadquestionvotes.php is used to query the votes table and determine how a user has voted on a specific question. The question is given by a URL parameter called id, specified in the request from the JavaScript front-end. The script works by selecting the vote status (if any) from the table of question votes in the database where the uid (user id) attribute in the table is equal to the user id in the URL parameter and the qid (question id) attribute in the table is equal to the question id in the URL parameter.

getUserVotes.js:

This is the script that's used to prepare and send the queries to check which way the user voted on a question (if at all). This is used to change the colour of the voting arrows depending on which way the user voted on a question (grey if they didn't vote). When the user clicks an arrow, a different function is run dependent on the current colour of the arrow. These functions are in the form of GET requests to modifyvote.php, specifying different parameters based on the way the user last voted.

The main functions in this script are Request() and Download(). Request() takes three functions: url (A String representing the URL you want to make a request true), method (A String representing the request method, either GET/POST) and callback (A function to run upon completion of the request). Request() is a universal function used to make a network request and run a callback after the request succeeded. It's used throughout this script.

The other function, Download(), is ran immediately after the page has loaded, and takes no parameters. It's used to get the user's current voting state for the question if someone is logged in, and to disable voting if the user isn't logged in.

The remaining files are simply used to draw the correct coloured arrows and run the Request() function to vote on a question.

Version 6 testing:

Version 6 is tested using the same testing methodologies as version 1, 2, 3, 4 and 5.

The results are shown below.

NullPointerException - Your questions. Answered.

Browser tests:

Browser	**Works/Doesn't work**	**Notes**
Mozilla Firefox	Works	None
Google Chrome	Works	None
Internet Explorer 11	Works	None
Microsoft Edge	Unknown	Won't connect to website at all so impossible to test
Mozilla Firefox (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work.
Google Chrome (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work

Basic functionality:

Page	**Test**	**Test data**	**Works/Doesn't work**	**Notes**
signup.php	Can create an account	All valid data	Works	Script creates an account using the data provided
signup.php	Can create an account	No data	Works	Script prompts user to fill in missing fields
signup.php	Can create an account	☺☺☺☺☺	Works	Script prompts the user to try again
login.php	Can log in to existing account	Username and password for an account existing in the database	Works	Script logs user into account
login.php	Can't log into account with wrong password	Username for an account that	Works	Script prompts user that

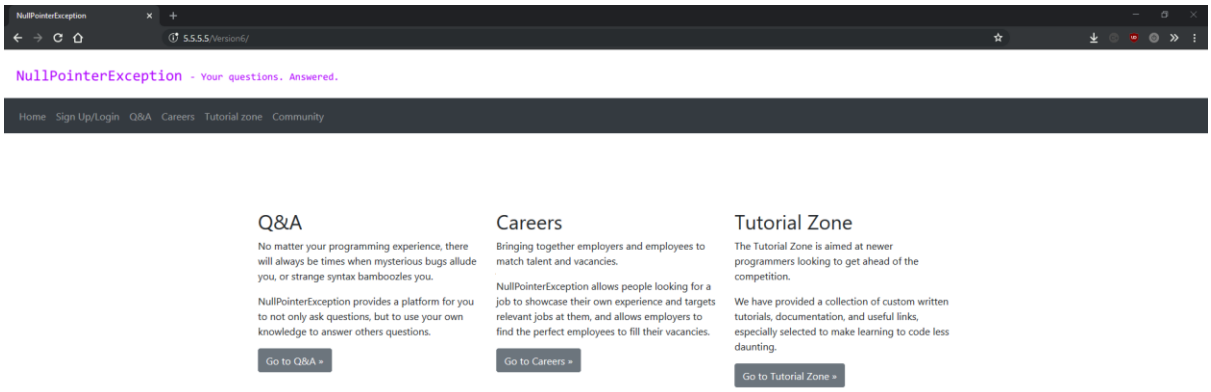
		exists in the database, wrong password for said account		username/password combination is invalid
login.php	Can't log into account that doesn't exist	Username and password that doesn't exist in database	Works	Script prompts user that username/password combination is invalid
ask.php	Can ask a question with valid inputs	Valid question title and question body	Works	Script adds a new question to the database
ask.php	Can't ask a question with nothing filled in	No data	Works	Script prompts the user to fill in every field
ask.php	Can ask a question containing unexpected characters	☺☺☺☺☺	Works	Script adds a new question to the database
searchq.php	Can search for a question with valid inputs	Valid search String	Works	Script shows all questions that match the search query
searchq.php	Can't search for a question without supplying any query	No data	Works	Script prompts the user to search for questions
searchq.php	Can search for a question containing unexpected characters	☺☺☺☺☺	Works	Script shows all questions that match the search query
comment.php	Can comment on a question using valid input	Valid input comprised on standard characters	Works	Script adds a new comment to the database
comment.php	Can't comment on a question without supplying a comment	No data	Works	Script prompts the user to input a comment

comment.php	Can comment on a question using unexpected input	☺☺☺☺☺	Works	Script adds a new comment to the database

Non-working tests will be addressed in future versions of the project

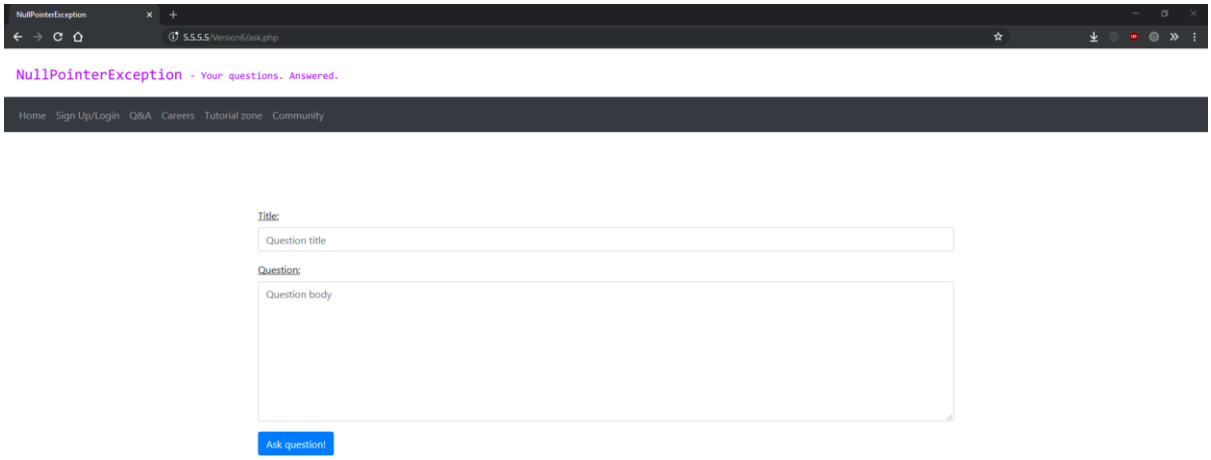
Version 6 screenshots:

index.php

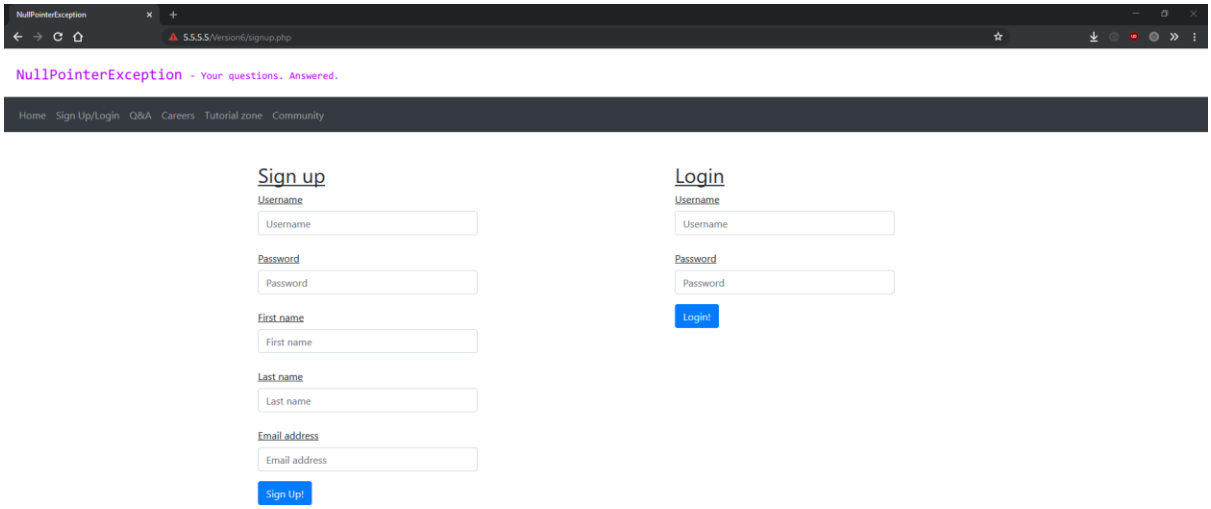


NullPointerException - Your questions. Answered.

ask.php



signup.php



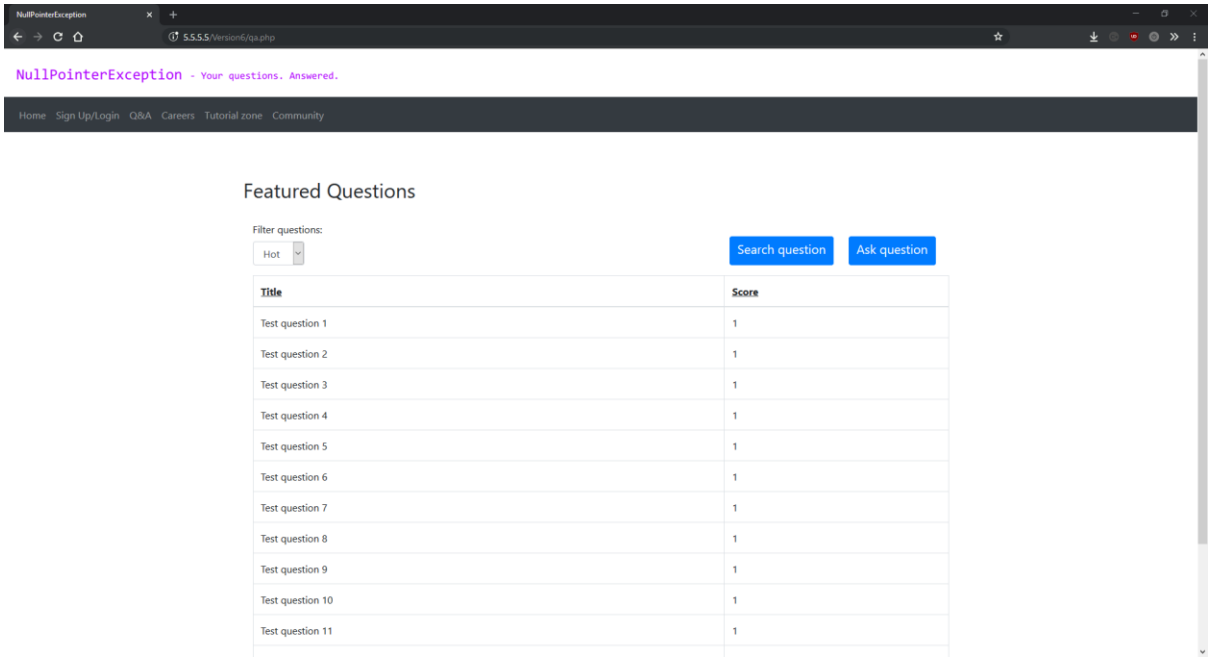
NullPointerException - Your questions. Answered.

career.php

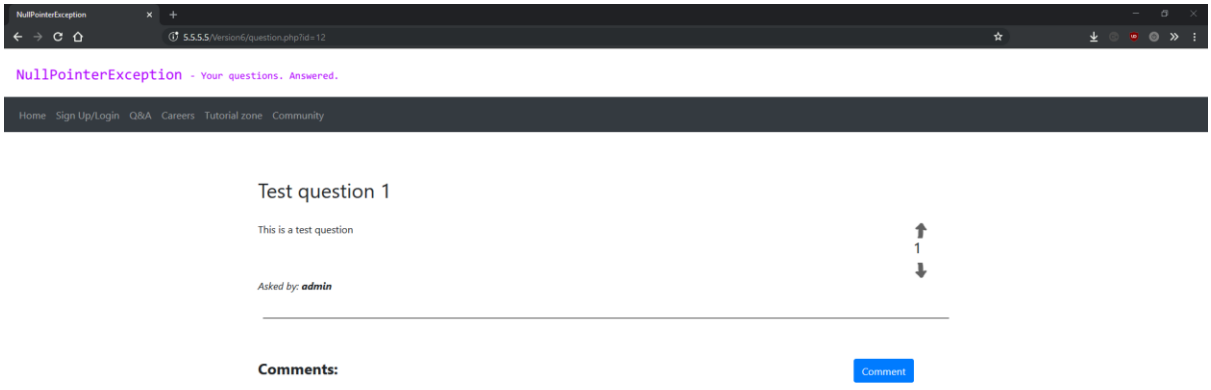
community.php

NullPointerException - Your questions. Answered.

qa.php

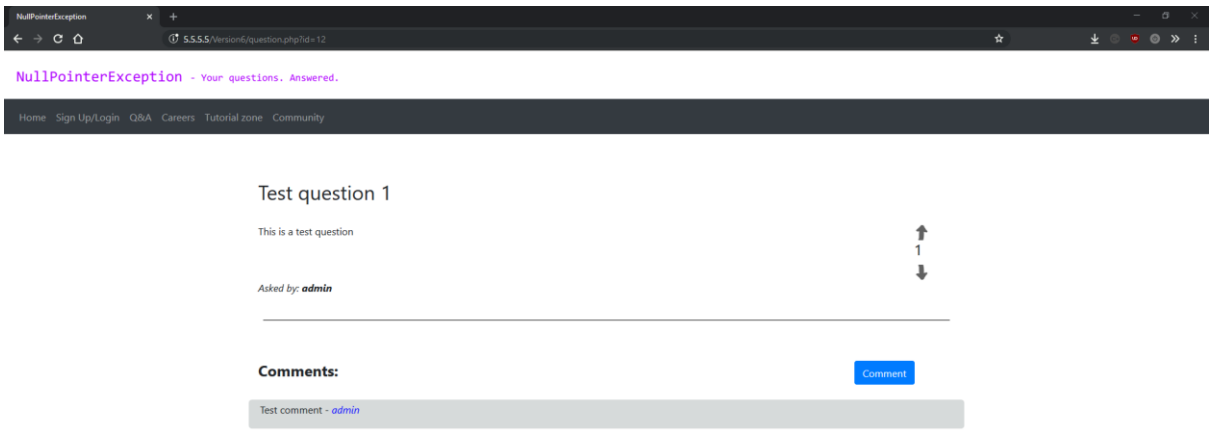


question.php

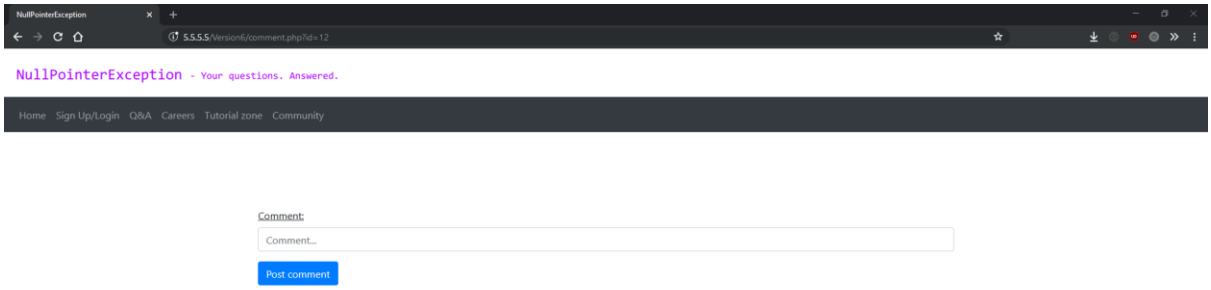


NullPointerException - Your questions. Answered.

question.php (with comment)

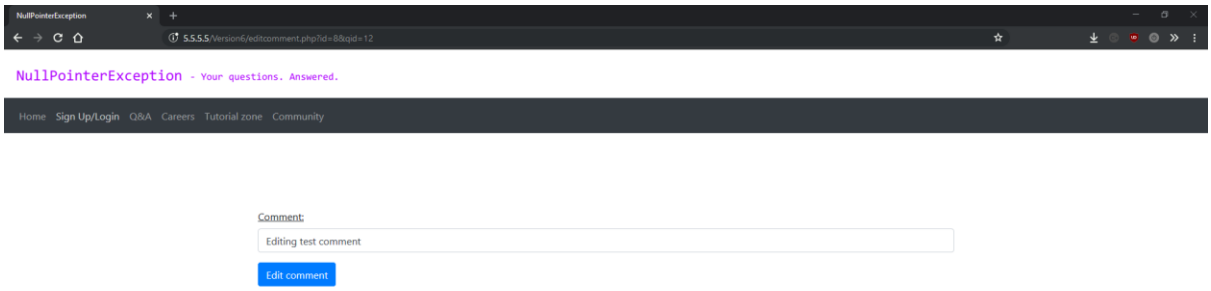


comment.php

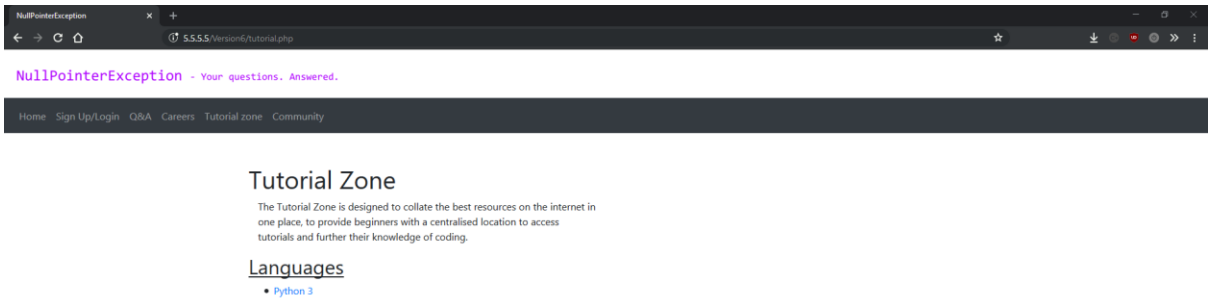


NullPointerException - Your questions. Answered.

editcomment.php

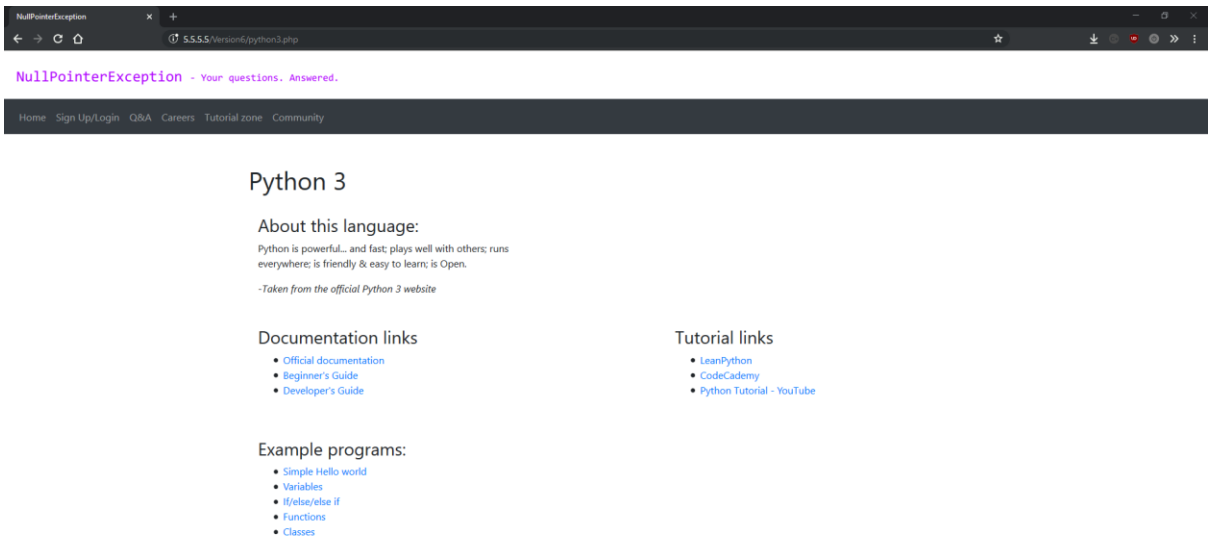


tutorial.php

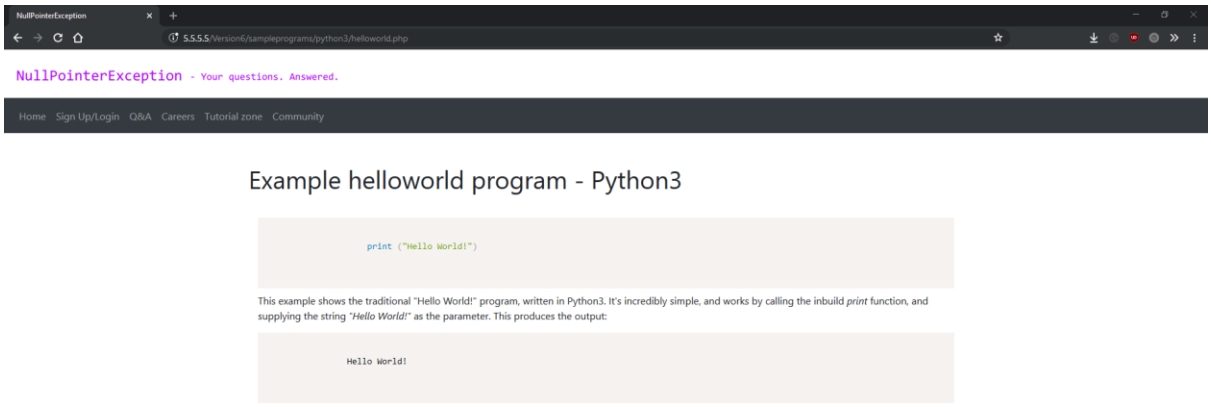


NullPointerException - Your questions. Answered.

python3.php

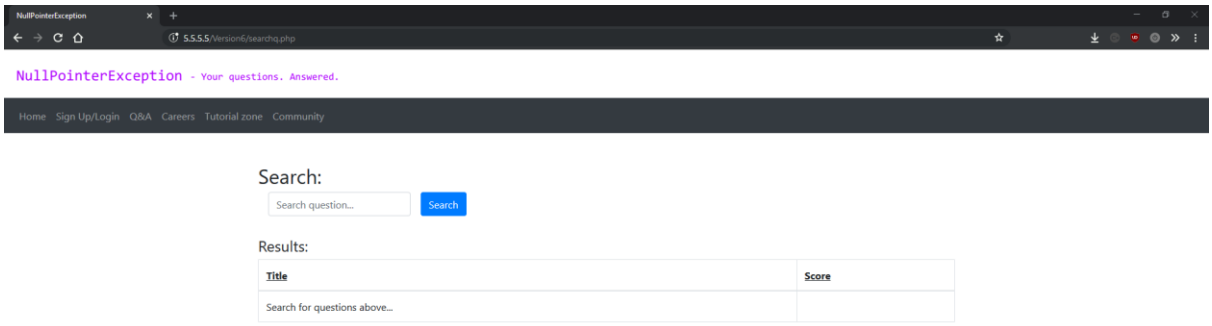


helloworld.php

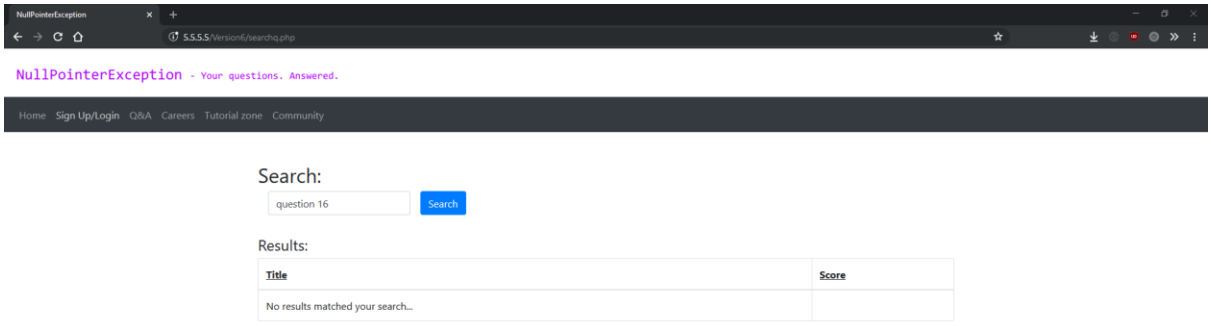


NullPointerException - Your questions. Answered.

search.php



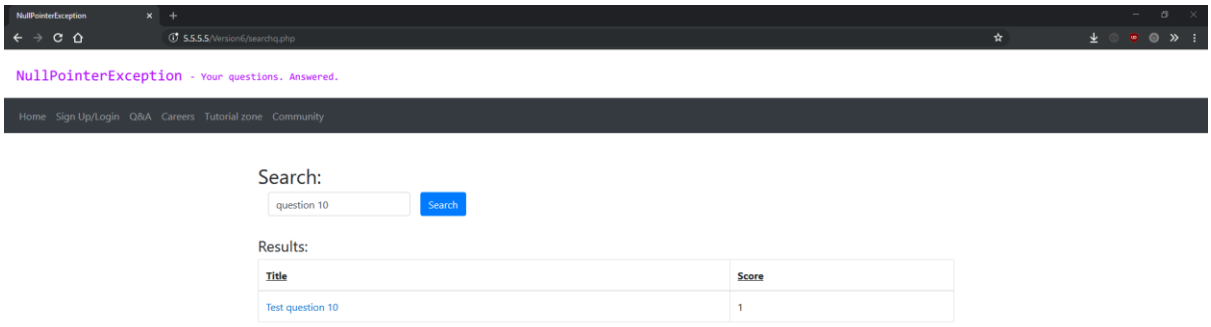
search.php (no results)



5.5.5.5/Version6/signup.php

NullPointerException - Your questions. Answered.

search.php (with results)



variables.php

Version 7

Version 7 of the project is the final rewrite I did, and also the best change to the overall structure of the project in my opinion. Everything after this version is new features and the overall code quality

from this version onwards is much better. In version 7 I swapped from a procedural style of coding to a more Object Oriented approach. This made it much easier to add new feature to the site as everything is now fully modular, and much easier to document and write-up about each script as I added comments to every function in every file.

Procedural programming is programming paradigm where the program is split up into statements and functions. The statements are functions that perform some kind of action, but there is no concept of objects. Object Oriented Programming is a programming paradigm where classes become objects which have their own properties and functions. Object Oriented Programming can be seen as an enhancement to Procedural Programming.

Version 7 was also the version I started to use Vagrant. Vagrant is a virtualisation program (similar to Docker) that uses a simple Ruby script to create and configure headless virtual machines so that each of your development environments, and your production server, are all identical. Using Vagrant made is so much easier to develop and test my project, as it provides a simple command line interface for creating, turning on/off and destroying the virtual machines based on the contents of the Ruby config script (called the VagrantFile). The way Vagrant works is you create your VagrantFile and commit it to your git repository, and then every time you pull/push to/from your repo vagrant will provision the virtual machine with the most up to date version of your VagrantFile. This means that if you make changes that require a new dependency on your laptop, then pull the changes from the git repository on your desktop, the virtual machine you run your server in is identical to the one on the laptop.

Using Vagrant was almost as big a change in terms of productivity as using git/GitHub. Because I did a lot of the project on holiday, and because I was constantly moving between my mum's house and my dad's house, having exact copies of my dev environment was incredibly useful. I'll cover how my custom VagrantFile works below.

Version 7 was also the version I finished the question and answer system. This is a significant section of the site, and was pretty much finished before this version, but the fixes and improvements I added in this version allow me to consider it finished. When I first started developing this project, I split it up into four main sections: The core site, the question and answer section, the careers section and the tutorial zone. This version marks the completion of the core site, the question and answer system and the tutorial zone. This was a massive moment in the development of the project.

The final new feature I added in this version was the complete tutorial zone. This could have been moved to it's own release due to it being a major section of the site, however since it's mainly markup and plain text, I added it to this release.

A list of the major changes in this version can be seen below:

VagrantFile:

A new file added in this version, VagrantFile is a Ruby script used to configure the headless virtual machine created by Vagrant. With Vagrant you start off by selecting the operating system you want to use for your virtual machine. I chose to use headless Ubuntu because I'm very familiar with Linux, and I use a similar distro at home (Linux Mint) which also uses the same package manager (apt). I then selected the IP and port that the LAN server should run on (5.5.5.5:80), and selected the synced

folder. One of Vagrant's strengths is you can sync the contents of a directory on your system into a directory on the virtual machine. I use this feature to sync the contents of my git repository (represented by ./) into the public_html folder of the apache installation on the virtual machine.

The next section of the VagrantFile is used to setup a prompt class to prompt the user for input and store the output as a variable. This section was very short but came from StackOverflow because I'd never used Ruby before. I use this later on in the VagrantFile to prompt the user to enter their GitHub account details in order to clone the private git repository used to store the database connection Strings.

The final section of the VagrantFile is the setup instructions. It is in this section where I install and configure packages and setup the virtual machine into a PHP development server. First I update the package manages cache to make sure all the software I'm installing is up to date, then I install the latest version of the apache2 server, git and PHP 7.1. Then I have to install two modules for PHP: mysqli and xml. These both do exactly what their names suggest. After installing these modules we install the en_GB.UTF-8 language pack for ubuntu and configure the server to use it. This is for when we want to display currencies in the careers section in a future release.

After installing all the software, we need to configure it. I use the Linux sed command to perform changes to the php.ini config file to enable full diagnostics and error reporting. I then run phpenmod mysqli to enable the MySQLi extension the script installed earlier, and restart the apache server.

Finally, we delete some auto-generated files and folders, then clone the private git repo that stores the database connection Strings so that the website can connect to the database.

The server is then running on 5.5.5.5 on the local network, serving the files stored in the git repo and reflecting any changes made to them in automatically. Vagrant uses this file when you first switch on the virtual machine, and will apply any changes if the VagrantFile changes.

***.js:**

All JavaScript scripts used to use var for creating variables but in this release I changed them to use let to reflect recent best practices in JavaScript.

Also all JavaScript files now have complete commenting, and were re-written to use a more OOP style, as well as generally more readable and efficient.

***.php:**

Every PHP script that connects to the database was changed to use the new Database class.

Also all PHP files now have complete commenting, and were re-written to use a more OOP style, as well as generally more readable and efficient.

Database.php:

This is the file that started the re-write. I deleted the old connect.php and created a new class called Database which uses a fully Object Orientated Programming approach. It's fully commented and has seven functions in this version, including a constructor. A constructor is a function that's run automatically when a new Database object is created. In the constructor, the class tries to connect to the database by first loading the connection Strings from an ini file, and then setting the $connection member variable to a new MySQLi connection object. This effectively gives an instance of the class it's own MySQLi connection object, stored as a property of the object. If the script fails to connect to the database then it throws an exception which will stop the rest of the scripts on the page from running, giving a custom error message.

The first function of the Database class is called query, and it completely revolutionised the way my project interfaced with the database. The purpose of this function is to run a MySQL query against the database, and one of the main benefits is that it already has access to the MySQLi connection object through the member variable initialised in the constructor. The other benefit of this function is that it uses one function for both MySQL queries that return a result (such as SELECT) and MySQL queries that don't return a result (such as INSERT INTO). This works because PHP allows functions that are sometimes void (Doesn't return any values) and sometimes non-void (Returns values).

The next three functions are all known as getter functions. In programming, getter functions are functions that simple return the value of a variable/object. The three getter functions in this release are getQuery(), getConnection() and getResult(). None of these take any parameters. getQuery() returns the current query stored as a member variable of the instance of the object. getConnection() returns the current MySQLi connection object stored as a member variable of an instance of the class. getResult() returns the last result returned from a MySQL query stored as a member variable of an instance of the class.

The last two functions in this class are both used to run operations on data stored from the last query ran by an instance of the class. The first, fetchAssoc() runs the mysqli_fetch_assoc() function on the data stored from the last query run by an instance of the class. The second is very similar, and runs the mysqli_num_rows() on the same data and returns the result.

core.php:

Another major file added in this version, core.php contains several core classes that are used throughout the program. The methodology of putting core classes and functions into a separate file is the same as the questionFuncs script except that the benefits are enhanced now that the entire project uses an Object Orientated Programming approach. There are two classes in core.php at the moment: User and Util. User is a class represents a User and all the functionality towards each user, and Util is a collection of static functions that are used in multiple other scripts throughout the project.

User consists of thirteen functions including a constructor. As described earlier in this document, the constructor is a special function that's called the first time an instance of an object is created. The constructor of the User function is used to create an instance of a User. However, this creates an interesting issue as this class is used in both the sign-up and the log-in processes. This means that when the scripts create a User object, they know different details about the user. The fix for this is using a special PHP feature where you can make function parameters optional. This means that

scripts can supply the password if they know it but not if they don't. This means I could use a one function fits all approach when designing this class. The constructor also create the password hash if a password is provided as an argument.

The next six functions are all getter functions. Since I've already described what getter functions are and how they work above, I will not go into detail about these functions here. These functions are: getPassword(), getPasswordHash(), getUsername(), getFirstName(), getLastName(), getEmail().

After these six functions, there is a function called generateHash() which takes no parameters. The hashing is done by the PHP function password_hash() using the PASSWORD_DEFAULT algorithm.

Then there is a slightly more complicated function called verifyHash(). It takes one parameter: $password (A String containing the password you want to compare the hash against). It uses the PHP function password_verify to check if the password provided in the argument can be matched against the password stored as a member variable of an instance of the User class. It runs true f the password matches the hash and false if it doesn't.

The next function is called getID() and takes one argument: $database (An instance of the Database class). It's used to get the ID of an instance of the User class from the database, and works by running a query against the database using Database→query(), a function from the database class used to run MySQL queries. It then returns the user's id from the database.

After this the next function in core.php is called allAttributesFilled(), and is used to check how many much information we know about a user object. This is needed because I designed the user object to work universally across the project, which means that sometimes we need to create a user object without knowing every single detail about the user (e.g. during the login stage where we need a validate a user object against the database, we only know a username and password). This function is used to check if we know everything about a user or not. It works by calling on a function from the Util class (also in core.php and explained below) called _isset(). This is an extension of the PHP build in isset() function used to check if a single variable is null/not-null. _isset() is used to accept as many arguments as you want and returns true if all of them pass the isset() function, or false if one or more of them doesn't pass. allAttributesFilled() takes no parameters and either returns true/false.

The final function in the User class in this version of the project is called create() and takes one parameter: $database (A reference to an instance of the Database class). This function is used to add a user object to the database, and is mainly used in by the signup section of the project to create a new user. First, this function calls the generateHash() function (also from the User class) and sets the passwordHash member variable of the instance of the User object to the hash that's returned. It then adds the user into the database by running a MySQL INSERT INTO query using Database→query(). Finally, it runs another MySQL query to select the auto generated id assigned to each new user added to the database, which is needed along with the user's username to log the user in. Finally, it closes the session and redirects the new user to the homepage.

This is the end of the first class in core.php, but there is also another class called Util which contains a collection of very useful statically accessed functions used throughout the site. Statically accessed functions are are functions that can be accessed directly from a script, without first creating an object. Util stands for Utility, as all the functions are functions are stand-alone helper functions that perform a mixture of simple and more complex tasks across the website.

The first version of the Util class contains five functions: redirect(), isIPBlocked(), _isset(), emailValid() and Error(). The first of these: redirect() is probably the most used function across the site. It takes one parameter: $url (a String containing the URL you want to redirect the client to). The purpose of this function is to safely redirect a client to another page. The safely is very important, because we don't want to lose important SESSION data or any other type of data, and we also want to avoid showing the user any errors if we don't have to. To avoid this I made two seperate ways of redirecting the user: calling the standard PHP header() function, and using the echo statement to run a JavaScript script to redirect the client. The reason I use two different methods of redirect is because the header() functions only works if the script calling it hasn't already sent any headers. Headers are a way of the client and the server communicating additional information not included in the standard HTTP request/response. PHP provides a way to check if the headers have already been sent, using the headers_sent() function. If the headers haven't been sent then the function runs the header() function to send headers that will redirect the user.

If they have been sent however, the function uses a series of echo statements to output and run inline JavaScript to redirect the user. This works if headers have been sent, and the combination of the two methods provides a robust and reliable redirection function that's utilised throughout the project.

The next function in the Util class is called isIPBlocked() and takes two parameters: $address (a String containing the IP address we are checking) and $database (a reference to an instance of the Database class). It works by first preparing a MySQL query to select any records in the blocked_ipaddr table where the address attribute matches the $address parameter provided when the function is called. It then uses the Database→numRows() function to return the number of rows returned from the query. If any values have been returned (max. 1) then the function returns true. If no results have been returned from the database then the function returns false.

Then we have the _isset() functions which takes an unspecified number of parameters. In PHP this is represented by ...$varName. The function then uses a foreach loop to iterate through all the parameters provided, calling the built in function isset() on each one. If any of theses isset() calls returns false then the functions automatically exits and returns false. If this doesn't happen then they must have all passed the isset() check and the function returns true by default.

The second to last function in the Util class is called emailValid() and takes one parameter: $email (a String containing the email address we want to validate). The function is used to check if a String could be used a valid email address as part of the validation process for new users. It works by calling the built-in PHP function filter_var(), using the filter as the FILTER_VALIDATE_EMAIL constant. The function returns true if the provided String matches the filter_var pattern, and false if it doesn't match the filter_var pattern.

The final function in this class is called Error() and takes between two-three parameters. The first two are compulsory: $message (a String containing the error message to show on the page) and $backButton (a Boolean which states whether to display a back button along with the error). The third parameter is called $backURL and only needs supplying if the second parameter ($backButton) is supplied as true. $backURL is a String containing the relative URL to redirect the user to. The function works by displaying the message, and then if the $backButton parameter is true, displaying a HTML button which redirects the user to the URL supplised in $backURL.

error.php:

Another new file added in this version, error.php is used to show the user a meaningful error in the event that something breaks or they try to access something that they shouln't. This enhances user experience as showing them nothing, or a PHP stack-trace, is not good user experience.

It works by getting the specific error from the URL parameter "error=", and switching over the error to display the error message for each case. It also provides a button to take the user back to the homepage.

The majority of this version is behind the scenes work fully commenting and documenting every file and script, and re-writing core components, as well as the style change from Procedural to OOP. Whilst this was a frustrating version to write as it doesn't add many new features to the project, it was incredibly important for the rest of the development. Adding comments and planning code before I wrote it was something I should have done from the start, but from here onwards I started thing about best practices and design instead of just mindlessly adding new features with little thought as to how they would fit into the main project. This version was a major turning point in the development of the project. It also marked the completion of the core site, the question and answer system and the tutorial zone. This was a massive milestone in the development of this project, leaving just the careers zone left to do.

Version 7 testing:

Version 7 is tested using the same testing methodologies as version 1, 2, 3, 4, 5 and 6.

The results are shown below.

Browser tests:

Browser	**Works/Doesn't work**	**Notes**
Mozilla Firefox	Works	None
Google Chrome	Works	None
Internet Explorer 11	Works	None
Microsoft Edge	Unknown	Won't connect to website at all so impossible to test
Mozilla Firefox (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work.

Google Chrome (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work

Basic functionality:

Page	**Test**	**Test data**	**Works/Doesn't work**	**Notes**
signup.php	Can create an account	All valid data	Works	Script creates an account using the data provided
signup.php	Can create an account	No data	Works	Script prompts user to fill in missing fields
signup.php	Can create an account	☺☺☺☺☺	Works	Script prompts the user to try again
login.php	Can log in to existing account	Username and password for an account existing in the database	Works	Script logs user into account
login.php	Can't log into account with wrong password	Username for an account that exists in the database, wrong password for said account	Works	Script prompts user that username/password combination is invalid
login.php	Can't log into account that doesn't exist	Username and password that doesn't exist in database	Works	Script prompts user that username/password combination is invalid
ask.php	Can ask a question with valid inputs	Valid question title and question body	Works	Script adds a new question to the database
ask.php	Can't ask a question with nothing filled in	No data	Works	Script prompts the user to fill in every field

NullPointerException - Your questions. Answered.

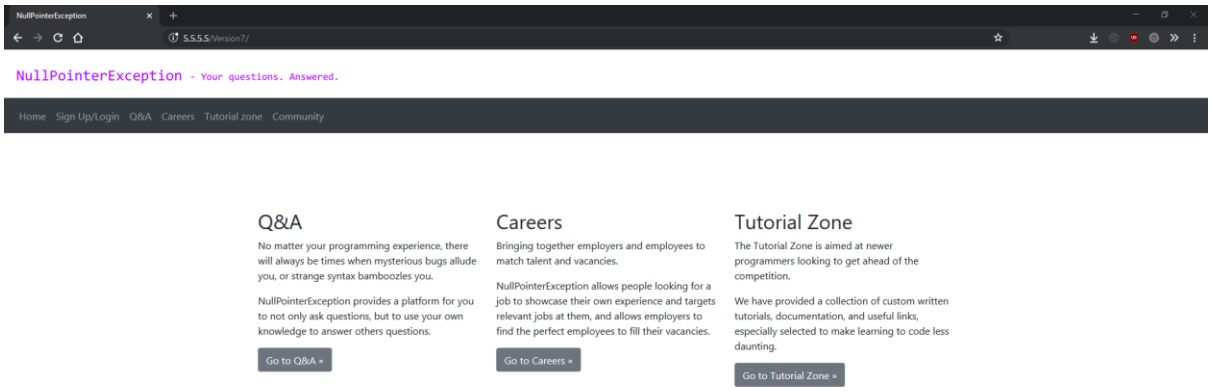
ask.php	Can ask a question containing unexpected characters	☺☺☺☺☺	Works	Script adds a new question to the database
searchq.php	Can search for a question with valid inputs	Valid search String	Works	Script shows all questions that match the search query
searchq.php	Can't search for a question without supplying any query	No data	Works	Script prompts the user to search for questions
searchq.php	Can search for a question containing unexpected characters	☺☺☺☺☺	Works	Script shows all questions that match the search query
comment.php	Can comment on a question using valid input	Valid input comprised on standard characters	Works	Script adds a new comment to the database
comment.php	Can't comment on a question without supplying a comment	No data	Works	Script prompts the user to input a comment
comment.php	Can comment on a question using unexpected input	☺☺☺☺☺	Works	Script adds a new comment to the database

Non-working tests will be addressed in future versions of the project

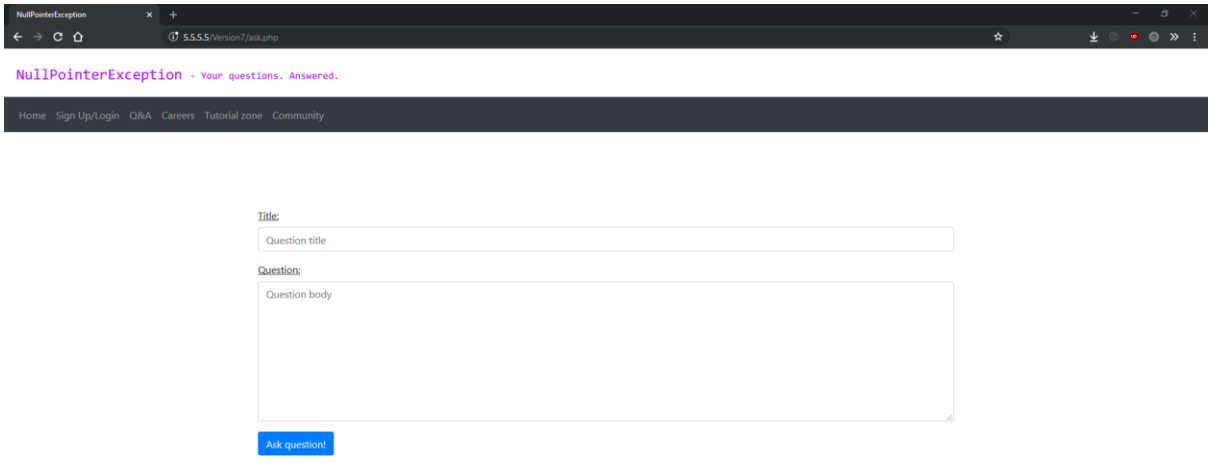
Version 7 screenshots:

NullPointerException - Your questions. Answered.

index.php



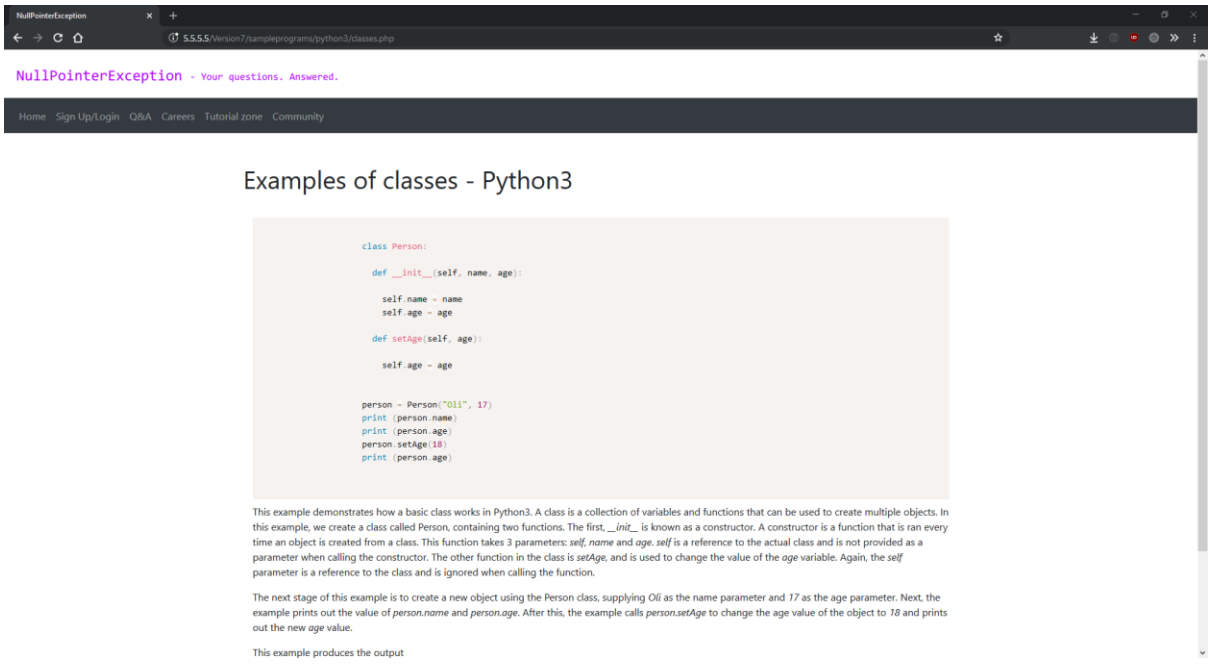
ask.php



NullPointerException - Your questions. Answered.

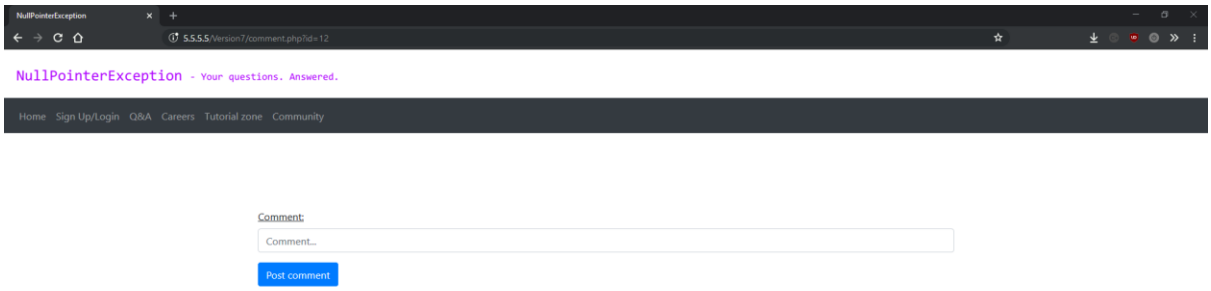
career.php

classes.php



NullPointerException - Your questions. Answered.

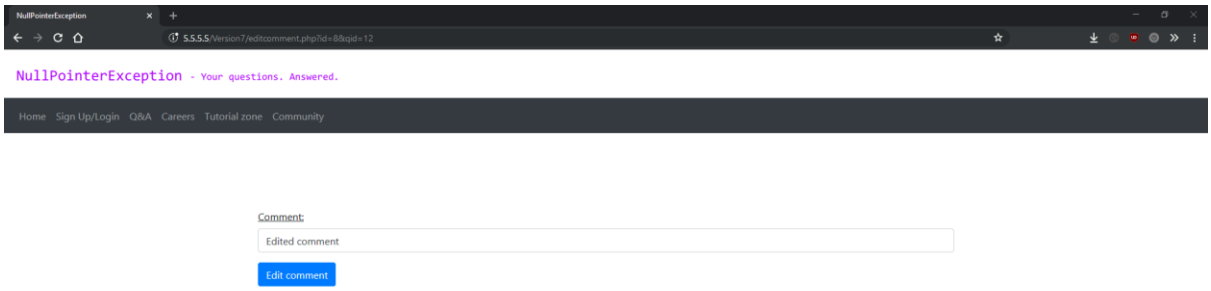
comment.php



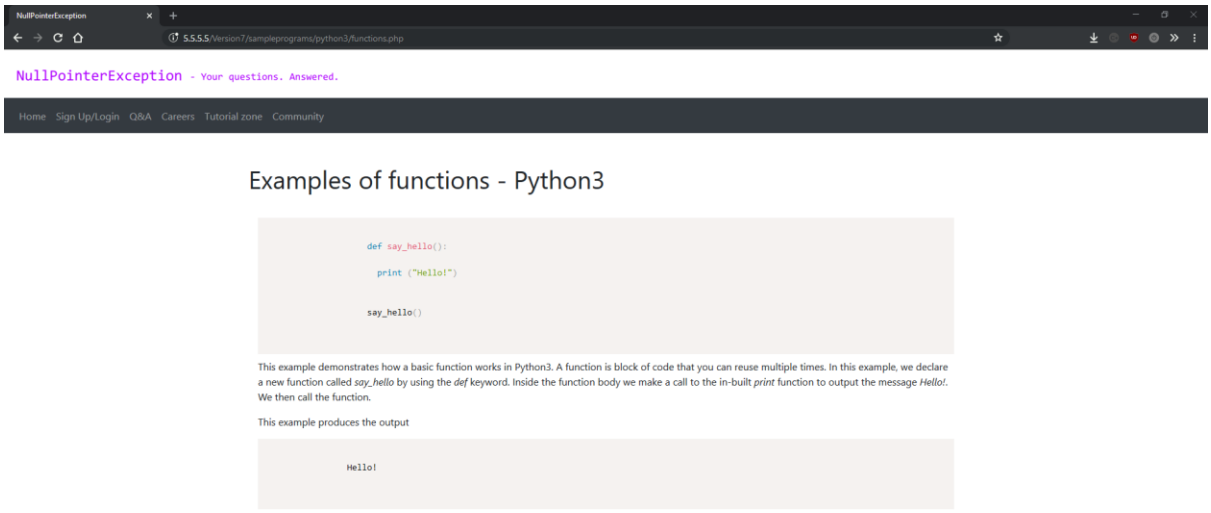
community.php

NullPointerException - Your questions. Answered.

editcomment.php

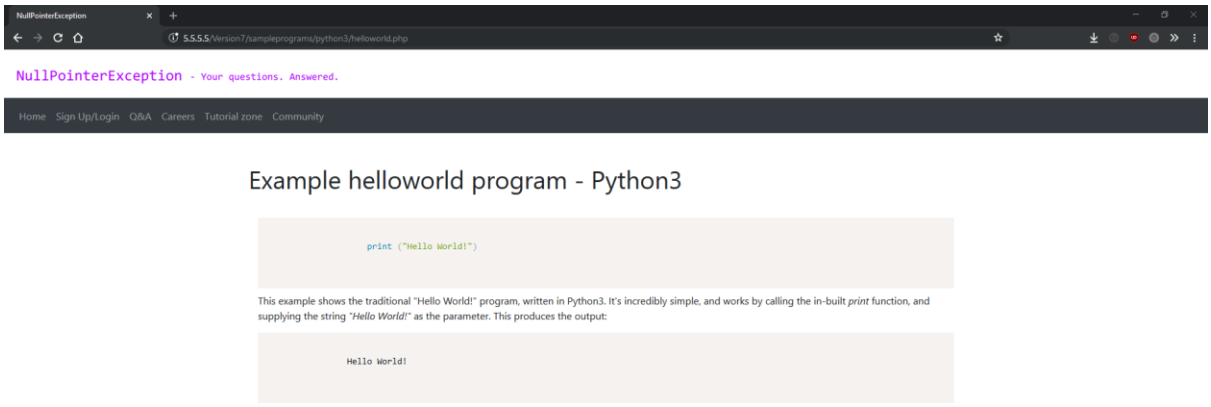


functions.php

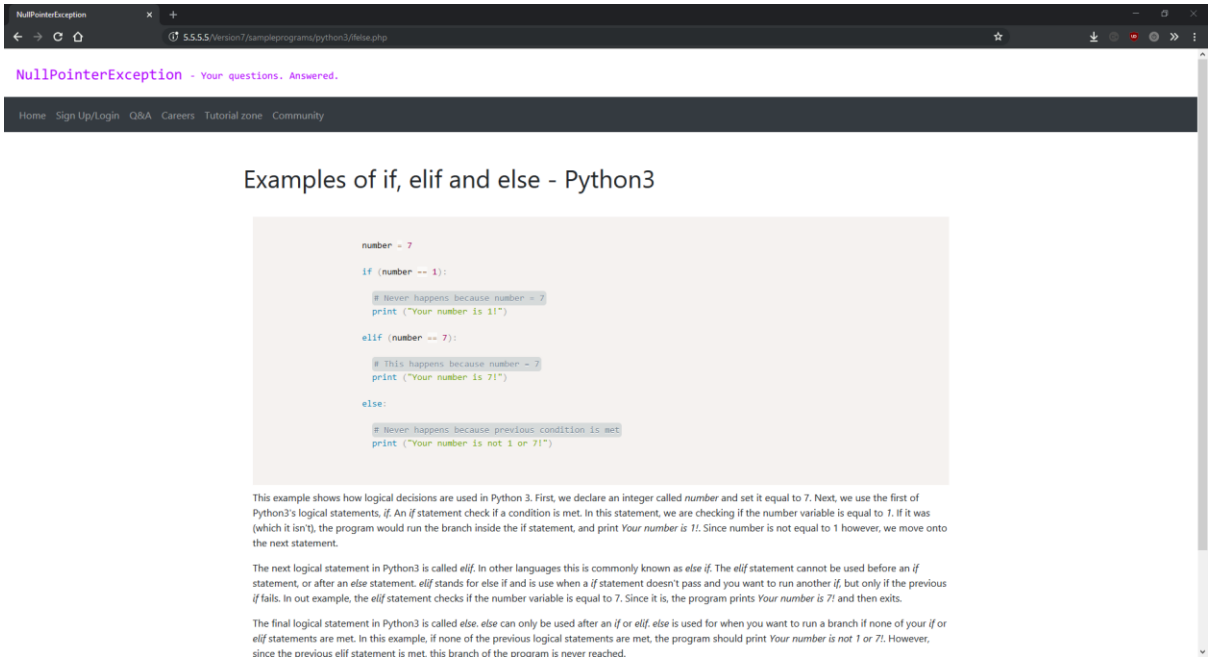


NullPointerException - Your questions. Answered.

helloworld.php

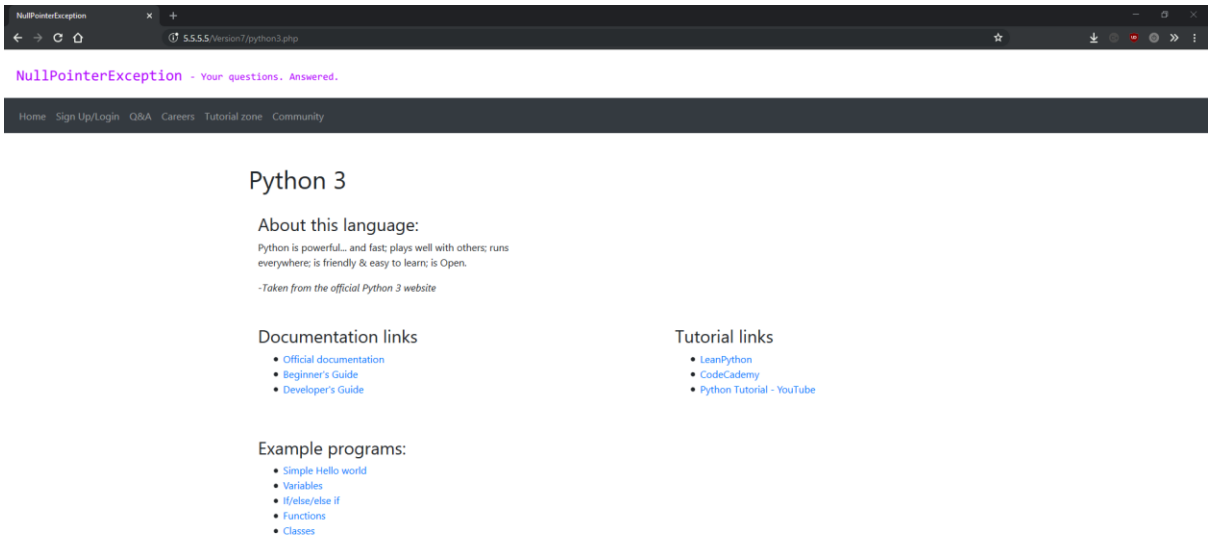


ifelse.php

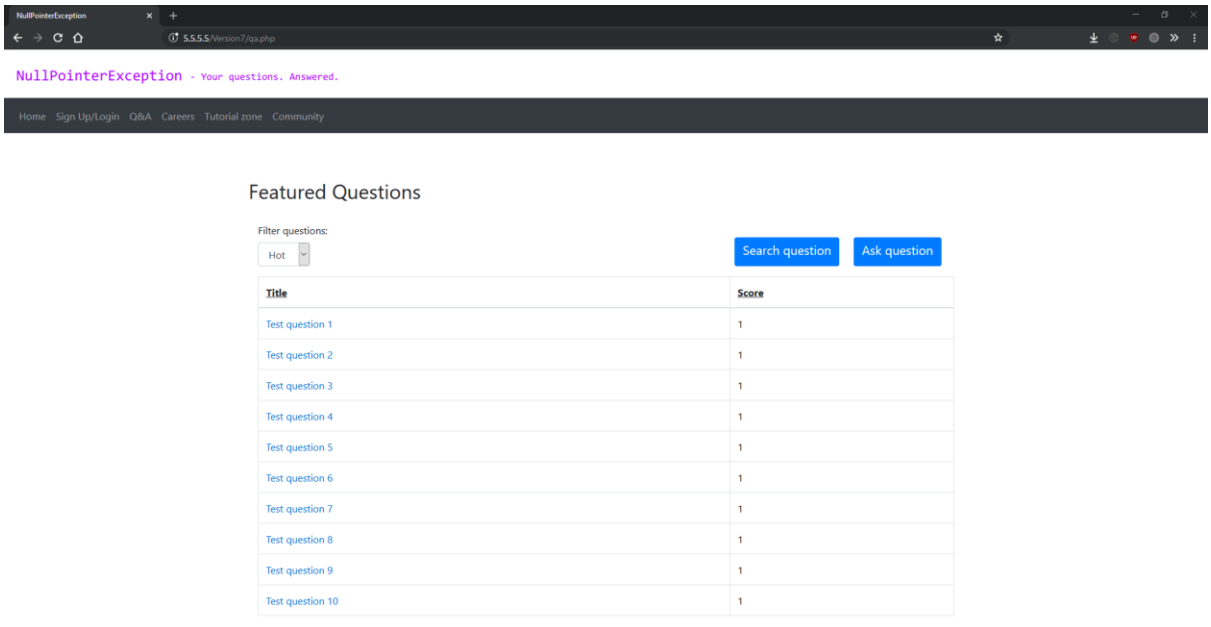


NullPointerException - Your questions. Answered.

python3.php

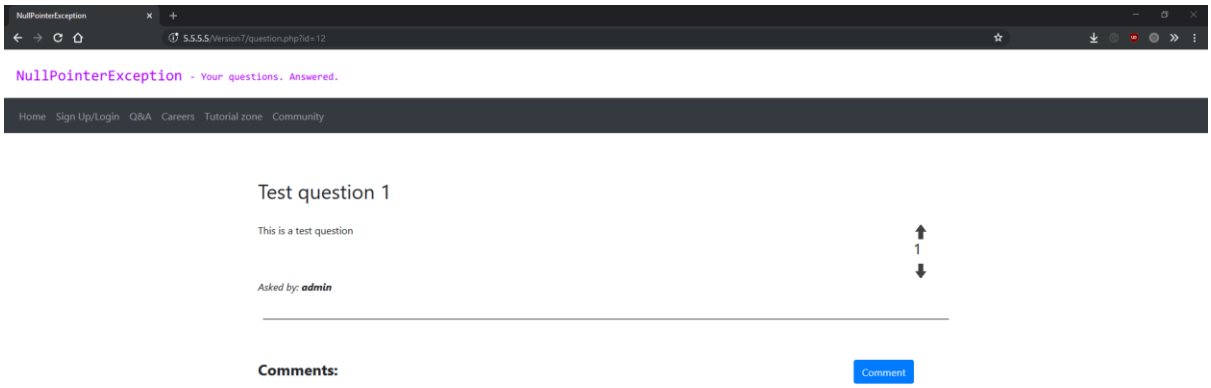


qa.php

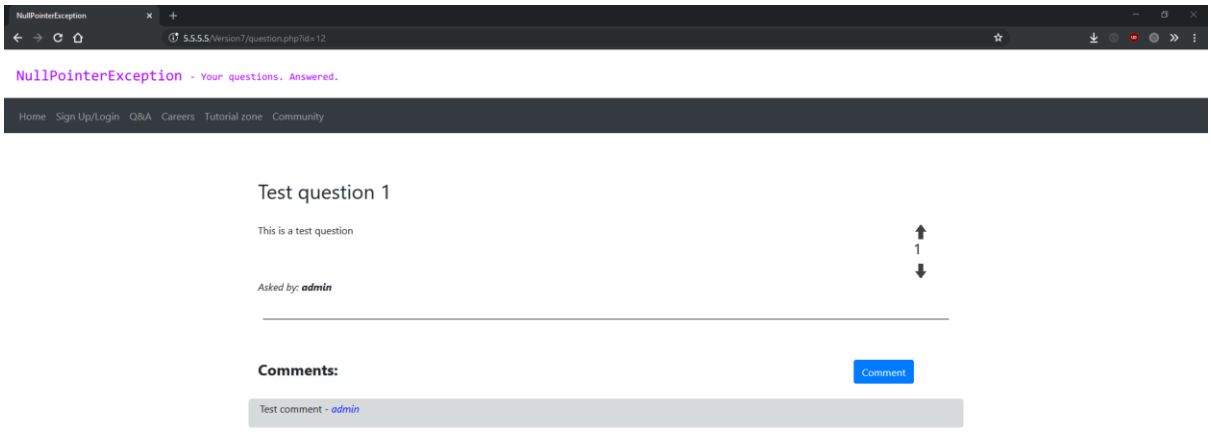


NullPointerException - Your questions. Answered.

question.php

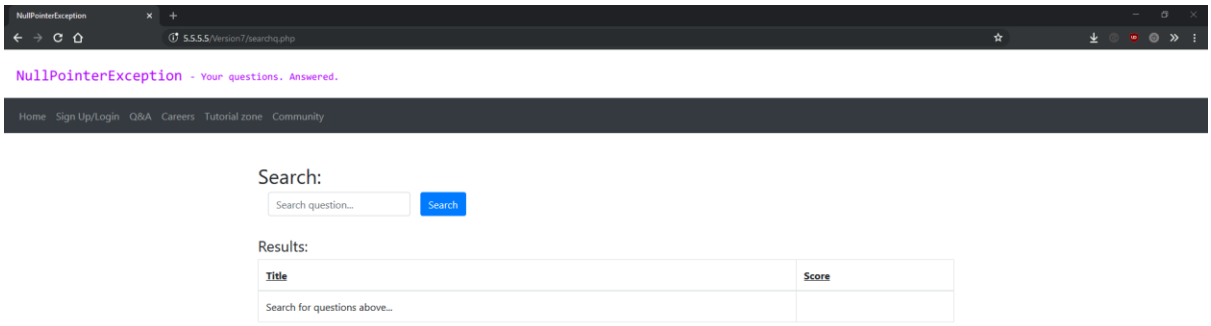


question.php (with comment)

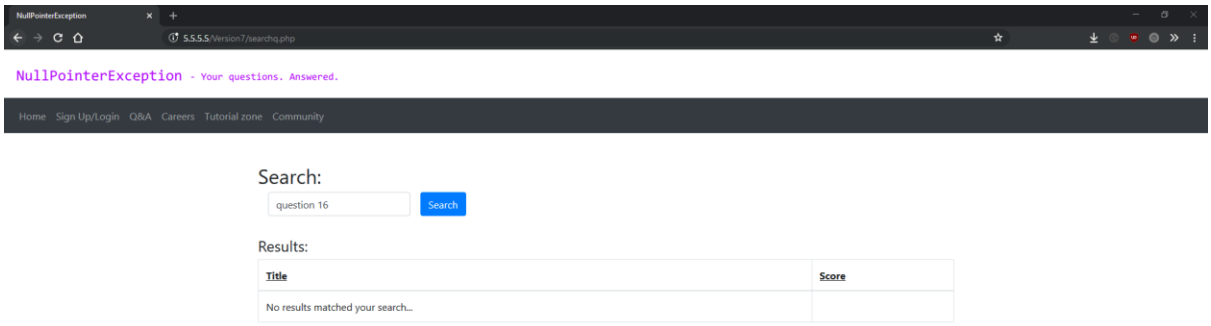


NullPointerException - Your questions. Answered.

search.php

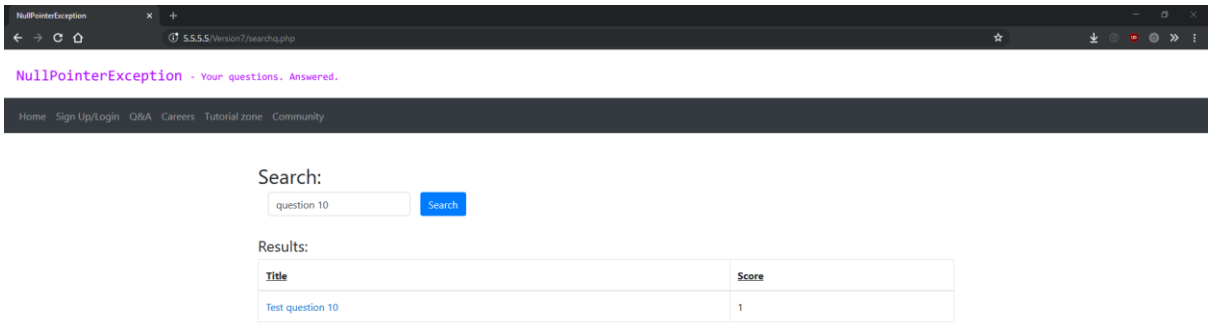


search.php (no results)

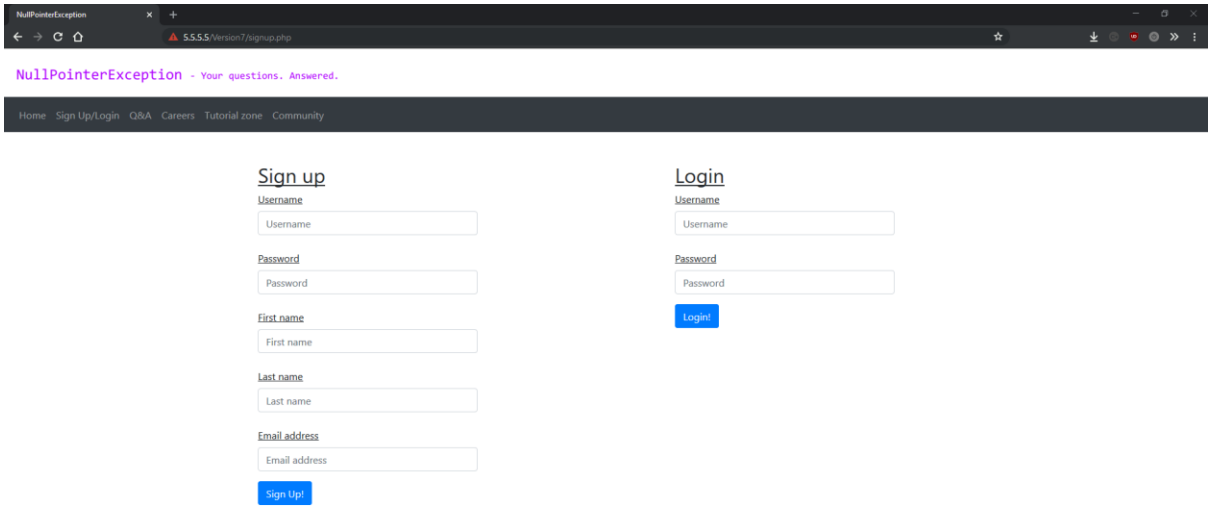


NullPointerException - Your questions. Answered.

search.php (with results)

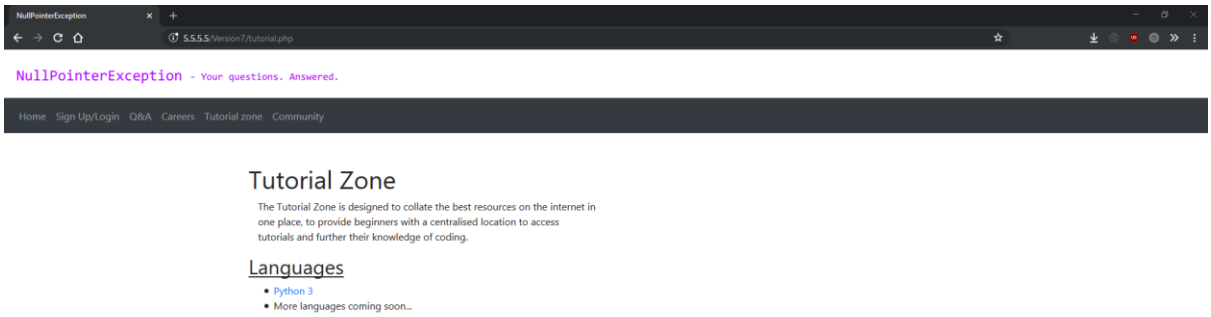


signup.php



NullPointerException - Your questions. Answered.

tutorial.php



variables.php

Version 8

The final feature update in the project, version 8 adds the careers section of the site. This is part four of the four part plan I split the project into when I was first thinking about how I wanted to approach the development of such a large project. By this stage of the project I was quite familiar with git, so I was pre-planning versions by creating branches for each large feature/release. This meant I could

work on different features at the same time without messing breaking other sections of the site, and when the feature was complete I could merge it into the main project.

However even after nearly two years of working on this project I was still making silly mistakes. I'd been using JetBrains PHP Storm IDE for about a year at this point and was finding it incredibly useful compared to just using a text editor like Sublime or Visual Studio Code. I use most of their project suite for other personal projects, so I'd been using their Toolbox app which manages projects and editors. The silly mistake was when I spent over two weeks trying to fix a bug where no matter what changes I made to the project, the virtual server was showing the same result. I was completely clueless as to what was happening, and I spent time trying to install more debugging tools onto the server, clearing the cache in the browser, and adding lots of debug statements and commands inside the project to try and understand what was happening. The issue turned out to be that I'd accidentally cloned the git repository onto two hard drives in the same computer, and JetBrains toolbox was only showing the wrong one, so I was editing one version of the project then running the server for the other version. This was infuriating and cost me lots of time, however I learned lots about PHP debugging as a result.

The careers system was quite a fun section of the project to write as I'd finished with re-writing the project, and some of the features that I was adding were nothing like anything I'd made before while working on NullPointerException. The tags system in particular was very interesting to design and write. This version marks the end of the main development of this project just leaving me write-up and some little bugfixes left to do.

This update added sixteen new scripts and changes to eleven others making it one of the two biggest versions throughout the project. Every script is fully commented and everything was designed before it was written. This version represents some of my best work during this project.

All major changes are shown and explain below:

VagrantFile:

I added -y as an argument to most commands in VagrantFile. This stands for "yes" and is used to default to yes when the commands ask a yes/no question. This is because these commands are normally entered by a human who then makes a decision about whether to carry on with the installation or whether it's going to install too much software or take up too much disk space. However vagrant runs these commands automatically and we definitely want to install everything and fully run every command so the -y argument simulates a user agreeing to everything the commands say.

I also added commands to install and configure the en_GB-UTF8 locale with PHP, due to an unexpected error with currency formatting on job.php. I go into detail about the error, the cause of the error, and how installing this locale on the server fixed it in the description for job.php.

career.php:

This is the landing page for the careers section of the site, however unlike qa.php, this page is mainly made up of HTML/CSS, simply asking the user if the are looking for a job or looking to hire someone

for a job. This makes it far simpler than qa.php which actually displays a list of questions that can be filtered. I chose to separate looking for a job and looking to hire in order to remove clutter and make the two systems more user friendly.

database.php:

I changed the query error message to show the actual MySQL error instead of just the query that caused it. This was added when I was trying to discover the issue cause by accidentally making two versions of the project, but was something I probably should have added into the Database class from the beginning.

downloadJobs.php:

This is the first large new file written as part of the careers system, downloadJobs.php is used to find all the jobs listed in the database which include tags that match the tags filter set on findJobs.php. It works by first creating an array of all the tags specified on findJobs.php, then querying the database to find all the jobs that match those tags and adding their ID's to an array called $matchingQuestionIDs. Since most of the jobs in the database will have multiple tags, I use a built in PHP function called array_unique() which creates an array from a set of values or another array, but without any duplicate values. This means if a user specifies tags of "JavaScript" and "jQuery", and there is a job which also contains tags of "JavaScript" and "jQuery", it will only appear in the results table once.

After finding all the jobs that match the tags filter, the script simply iterates over them all and outputs them in a format that the JavaScript script that initiated the request can read to output them into the results table in real time.

downloadNewJobs.php:

A very simple new script added with this version, downloadNewJobs.php is used exactly how it's name suggests, to download then newest jobs added from the database to show before the user changes the tags on findJobs.php. It works by using a MySQL SELECT statement, and using "ORDER BY id DESC" to order the results by the auto-generated auto-incrementing id attribute in descending order. It this outputs the jobs in a form expected by the JavaScript script that initiated the request.

downloadTags.php:

Another very simple script added for this version, downloadTags.php is used to select all the tags in the tags table of the database, for use in the autocomplete tags input. It works by simply querying the tags table using a "SELECT *" query and outputting all the results.

downloadsearchjobs.php:

Another new script added in this version, downloadsearchjobs.php is the careers system variant of downloadsearchq.php from the questions and answers section. It's purpose is to search the database based on a search query given to it in a request, and output the results in a format readable by the same JavaScript script that made the initial search request. As usual, it starts of by including the Database class and retrieving the search query from the URL. It then uses the Database to query the database using the same search pattern as downloadsearchq.php. As this has already been explained above, I will not go into further detail here. It then outputs the results in the expected format.

findjobs.php:

This script is almost entirely made from HTML/CSS, and is used to show a table of jobs returned from the database. These jobs are initially the most recent jobs added to the database, but when the user adds tags to their filter, the jobs get filtered in real time with no page reloads. The page includes a table for the jobs to be displayed in, as well as a prompt to add tags to their filter, including autocomplete similar to a search engine. As with all the HTML pages in this project, all the heavy lifting is done by external PHP/JavaScript.

findpeople.php:

findpeople.php is the page in the careers section where employers post new jobs. As with findjobs.php, it's almost entirely made from HTML/CSS. It features a form for the employer to add their job, as well as a tag input with autocomplete similar to the one on findjobs.php. This page was unexpectedly complicated to write due to having the tag autocomplete in the same HTML form as the rest of the page. The issue was that when you clicked the button to add a tag it would submit the form instead. This was fixed after lots of hours work by setting the type attribute to "button" on the "add tag" button. For some reason in HTML if you make an element <button type="button"></button> it no longer makes the an HTML form submit when clicked, enabling you to run a JavaScript instead. This bug occurred at the same time as the multiple versions bug where I was editing the wrong version of the project, which made this issue take considerably longer to fix than normal. The rest of the file is just HTML/CSS.

job.php:

job.php is the page used to display any job from the database, just like question.php from the question and answer system. This script also presented some unexpected issues that ended up needing some major changes to the server and VagrantFile. The page works by creating all the formatting using HTML and bootstrap classes, then using PHP's echo statement to output the correct part of the job in the correct place. The job is retrieved from the database based on it's id, which is given by a URL parameter called id.

The issue whilst developing this script was due to how PHP deals with currency formatting. The salary for the job is stored as an integer in the database, and was outputted using a PHP function called money_format() which allowed me to specify parameters such as number of decimal places and currency etc. However for some reason this wouldn't draw the "£" symbol whenever I ran it.

After about 2 days of research and debugging, I found out that it had to do with how PHP handles locales and currency formatting. The fix was installing the languagepack-en package on the server and using it to install the en_GB.UTF-8 locale, and set it as the locale of the server. Then, I had to call on some PHP functions to use this as the default locale of the page when the page first loaded. These functions were *setLocale(LC_ALL, "en_GB.UTF-8")* and *header('Content-Type: text/html; charset=ISO-8859-1')*. Then I needed to call the *utf8_decode()* function before outputting the salary of the job. I'm not sure why this combination of those functions worked, but any other combination of these and other functions would not render the "£" sign. This was a frustrating bug to fix but it ended up working perfectly and I learned a lot about working with PHP and Unicode.

proccessjob.php:

Another new file added in this version, processjob.php is used to process and upload a new job submitted by an employer to the database. It works very similar to processquestion.php. First, it includes the core.php and database.php so that it can utilise the Util and Database classes. Then it removes the array formatting done by JavaScript, by removing the "[" symbol and "]", representing the start and finish of an array in JavaScript. Then it converts the salary into an integer and performs a null check on all the attributes before uploading the new job to the database and redirecting the user to the jobs homepage.

filterJobs.js:

This is a JavaScript script used to filters the jobs in the job table on findjobs.php in real time in accordance to the tag filters the user has specified. This was a fun script to write and design as it involved some interesting approaches to the problem. It's a very modular script with ten explicit functions and two lambda functions. It starts off with a lambda (anonymous) function which is called when the script first loads, and is used to run two of the functions from this script: getTags() and downloadNewestJobs(). The purpose of these scripts will be described in more detail below.

The first function defined in this script is called addTagToFilter() and does eactly what it's name suggests, adding a tag to the current filter. It takes no parameters, and is called when the user clicks the "add tag" button. This function works by first getting the String value of whatever the user has written in the tag input box, then checking if they have actually written anything. Assuming they have, it adds the contents of the tag box to the current filter using the Array.push() function. Then it creates the HTML elements required to display the tag in a box to show what filters the user currently has, along with adding an onclick reference to the function to delete the tag. Finally, it calls the downloadJobs() function to redownload and filter the jobs using the new tag combinations, and re-hides the "add tag" button.

The next function in this script is called showAddTagButton and also takes no parameters. It's used to display a button inline with the tag input box. It works by first checking to see if the tag input box has anything written in it, then it either hides it if nothing has been written, or it shows it if it does. This function is ran every time the user presses any key on their keyboard whilst the tag input is in focus.

After this, the next function is called getTags() and also takes no parameters. This function is used to initiate a network request to get the tags from the database and store them in an array. It works by creating an XMLHttpRequest object, specifiying the request type as a GET request, then using it to download the contents of downloadTags.php. After the request has been competed, the function stores the returned value in an array called tagsarray and adds the results to the autocomplete system.

Then we have removeTag. This is the function than runs when the user clicks on one of the tags in their filter in order to remove it. It takes one parameter: tag (a String containing the tag to remove from the filter). It works by first deleting the HTML element, then removing the tag from the filter by removing it from the currentTags array. Then it checks to see if there are any other tags left in the filter. If there are then it simply calls downloadJobs() to re-download and re-filter the jobs in the table. If not then it calls downloadNewestJobs() to add the newest jobs to the table because the user hasn't specified any filters.

The next function in the script is one of the most important ones. It's called downloadJobs, and it's used to download jobs that match the filter specified by the user. It takes no parameters and works by first creating an XMLHttpRequest object to make the actual request and setting it to be a GET request. Then it creates a comma delimited String of all the tags in the current filter. Then it sends the query off to downloadJobs.php for processing and database interaction, setting the outputJobs() function as it's oncomplete callback.

The outputJobs() function takes one parameter: httpRequest (an XMLHttpRequest object representing the GET request that was used to download the jobs based on the tag filter). This function is the callback for downloadJobs() which means that when the GET request is complete it will call outputJobs(). The function works by splitting the response up into individual questions , then calling the wipeJobs function to remove any jobs that are already in the jobs table. Then it calls the createJobElements() function to create HTML elements for the jobs downloaded from the database.

The next function is called createJobElements() and is used to create HTML elements for every new job that needs adding to the table. It takes two parameters: jobsArray (an array contain every job fetched from the database) and jobsTable (a reference to the HTML table element that the jobs need adding to). This function works by iterating over every job in the array, and creating the required elements using the document.createElement() function. The elements that need creating include: <tr>, <td>, <a> and <p>. This function also performs tasks such as setting link locations and using some regex to replace commas with a comma followed by a space for better user experience.

The final function in this script is called downloadNewestJobs() and is very similar to downloadJobs(). The difference is that this function is used when no filters have been supplied, to show the newest jobs added to the database. It works on the same principles as downloadJobs(), creating an XMLHttpRequest object set as a GET request, then using it to query the downloadNewJobs.php page. It also sets outputJobs() as it's callback.

filterSearchJobs.js:

This is another new script, used to make a search request, filter the results and output them onto the screen. It contains three functions: download(), output() and createJobElements(). These are similar functions to the ones in filterSearchQuestions.js. This is because filterSearchJobs.js is based

on filterSearchQuestions.js. The first function: download() takes no parameters and is ran as soon as the page loads or whenever someone clicks the search button. It works by getting the search query from the search bar, then it creates a GET request to the downloadsearchjobs.php passing the value of the search bar as the search query. It sets output() as it's request callback. If nothing has been typed into the search bar the it simply shows a prompt to search for questions.

The second function in this script is called output() and is used to output the results from the query. It's used as the callback from the GET request, which means that its called when the GET request has finished. First it checks that the request was successful and executed correctly. Assuming it has, the function creates an array of the request response then calls the wipeTabe() function from core.js. This is used to remove all the elements from the table so that new results can be shown. Finally it checks to see if any results got returned from the database, calling the createJobsElements() function if there were, and displaying an error message if there weren't.

The final function in this script is called createJobsElements(). This is a function used to create all the HTML elements needed to show the results from the search query. It takes two parameters: jobsArray (an array containing the search results from the database) and table (a reference to the HTML table element used to display the search results in). It does this by using the document.createElement() function to create the elements. The elements that need creating are: <tr>, <td>, <a> and <p>. It also uses some regex (regular expression) to add a space after every comma in the response from the database to improve user experience. It then adds all these elements to the table.

salary.js:

This is a simple script designed to fix a minor issue with the form to submit a new job. The issue was with the salary input box which used to allow users to enter non-numeric values such as letters or symbols. This script sets the characters the user can enter, and runs every time the user presses a key whilst inside the salary input box. The second to last line from this script is taken from a StackOverflow post to provide an elegant, functional and fast solution.

tags.js:

tags.js is the script used for managing the tags for job posting form. It's made up of three functions: addTagToFilter(), showAddTagButton() and getTags(), none of which require any parameters. The first, addTagToFilter(), is used for adding a tag entered in the tags input box to the filter. It also hides the "add tag" button. It does this by first checking if the user has written anything in the "add tag" input box. Assuming that they have, it then creates the HTML elements for displaying a new tag using the document.createElement() function, and then creating a new URL String used to handle then form after the user clicks submit. This URL is comprised of the page (processjob.php), then a URL parameter called tags. This parameter is a String starting with a "[" character and ending with a "]" character. In between the two square brackets is a comma separated list of all the tags. For example, a URL could be "processjob.php?tags=[php,mysql]". The processjob.php script reads the tags from the URL and uses them to search the database.

The next function is called showAddTagButton() and is the simplest function in the script. It's used to show the "add tag" button if the user writes anything in the "add tag" input box and hide it if they delete what was already written.

The final function inside this script is called getTags() and is used to provide a list of tags in the database to the autocomplete "add tags" input box. It takes no parameters and works by creating a GET request to download the tags from the downloadTags.php page. It then creates a lambda (anonymous) callback function to split the response up into an array based on newline HTML entities and send them to the autocomplete system.

This was a massive version to create. It represents about a month of planning and programming and completes the development of my project. This version felt like all the work on modularity and scalability was paying off, and all the research into software and tools had come together to make this one of the easiest versions to program, despite it's scale.

Version 8 testing:

Version 8 is tested using the same testing methodologies as version as all the previous versions, but is also tested in the evaluation document by other people using a black-box approach. I will go into more detail about these further tests in the evaluation document.

The results of the original tests are shown below.

Browser tests:

Browser	**Works/Doesn't work**	**Notes**
Mozilla Firefox	Works	None
Google Chrome	Works	None
Internet Explorer 11	Works	None
Microsoft Edge	Unknown	Won't connect to website at all so impossible to test
Mozilla Firefox (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work.
Google Chrome (Android)	Mostly works	Site hasn't been specifically designed for mobile but most pages work

NullPointerException - Your questions. Answered.

Basic functionality:

Page	**Test**	**Test data**	**Works/Doesn't work**	**Notes**
signup.php	Can create an account	All valid data	Works	Script creates an account using the data provided
signup.php	Can create an account	No data	Works	Script prompts user to fill in missing fields
signup.php	Can create an account	☺☺☺☺☺	Works	Script prompts the user to try again
login.php	Can log in to existing account	Username and password for an account existing in the database	Works	Script logs user into account
login.php	Can't log into account with wrong password	Username for an account that exists in the database, wrong password for said account	Works	Script prompts user that username/password combination is invalid
login.php	Can't log into account that doesn't exist	Username and password that doesn't exist in database	Works	Script prompts user that username/password combination is invalid
ask.php	Can ask a question with valid inputs	Valid question title and question body	Works	Script adds a new question to the database
ask.php	Can't ask a question with nothing filled in	No data	Works	Script prompts the user to fill in every field
ask.php	Can ask a question containing unexpected characters	☺☺☺☺☺	Works	Script adds a new question to the database

searchq.php	Can search for a question with valid inputs	Valid search String	Works	Script shows all questions that match the search query
searchq.php	Can't search for a question without supplying any query	No data	Works	Script prompts the user to search for questions
searchq.php	Can search for a question containing unexpected characters	☺☺☺☺☺	Works	Script shows all questions that match the search query
comment.php	Can comment on a question using valid input	Valid input comprised on standard characters	Works	Script adds a new comment to the database
comment.php	Can't comment on a question without supplying a comment	No data	Works	Script prompts the user to input a comment
comment.php	Can comment on a question using unexpected input	☺☺☺☺☺	Works	Script adds a new comment to the database
findpeople.php	Can post a job listing using valid data	Valid job title, description, location, company, salary and tags	Works	Script adds a new job to the database
findpeople.php	Can't post a job listing with no data	No data	Works	Script prompts the user to fill in all the fields
findpeople.php	Can post a job listing with unexpected characters	☺☺☺☺☺	Works	Script adds a new job to the database
findpeople.php	Can add a valid tag	A valid tag	Works	Script adds a new tag to the filter
findpeople.php	Can't add a tag with no data	No data	Works	Button to add tag does not appear if

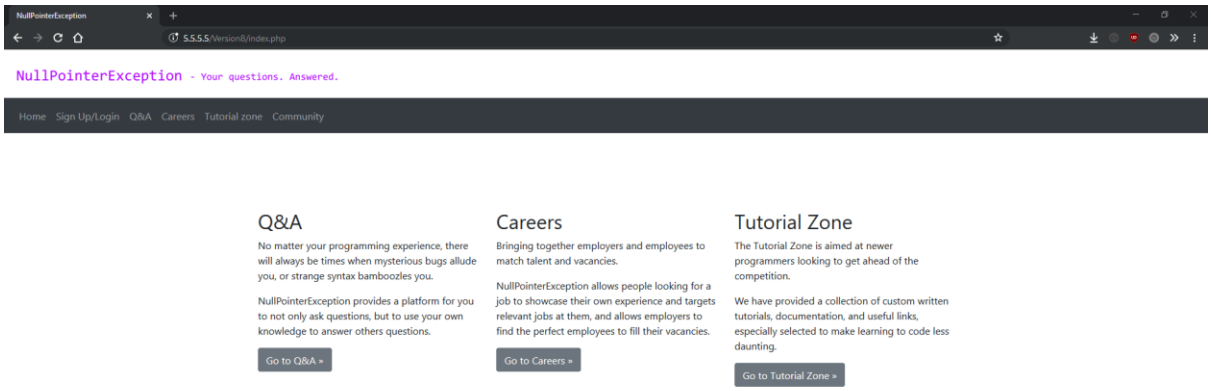
				the input box is empty
findpeople.php	Can add a tag with unexpected characters	☺☺☺☺☺	Works	Script adds a new tag to the filtered
findPeople.php	Can search for a job with valid input	Valid search String	Works	Script shows all jobs that match the search query
findPeople.php	Can't search for a jobs without supplying any query	No data	Works	Script prompts the user to search for jobs
findPeople.php	Can search for a job containing unexpected characters	☺☺☺☺☺	Works	Script shows all jobs that match the search query
findjobs.php	Can add a valid tag	A valid tag	Works	Script adds a new tag to the filter
findjobs.php	Can't add a tag with no data	No data	Works	Button to add tag does not appear if the input box is empty
findjobs.php	Can add a tag with unexpected characters	☺☺☺☺☺	Works	Script adds a new tag to the filtered

Non-working tests will be addressed in future versions of the project

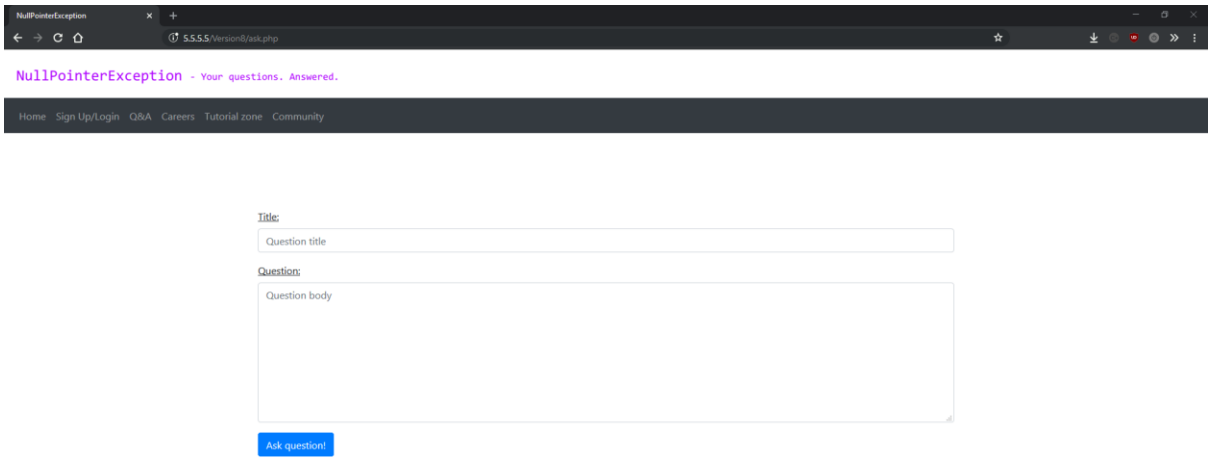
Version 8 screenshots:

NullPointerException - Your questions. Answered.

index.php

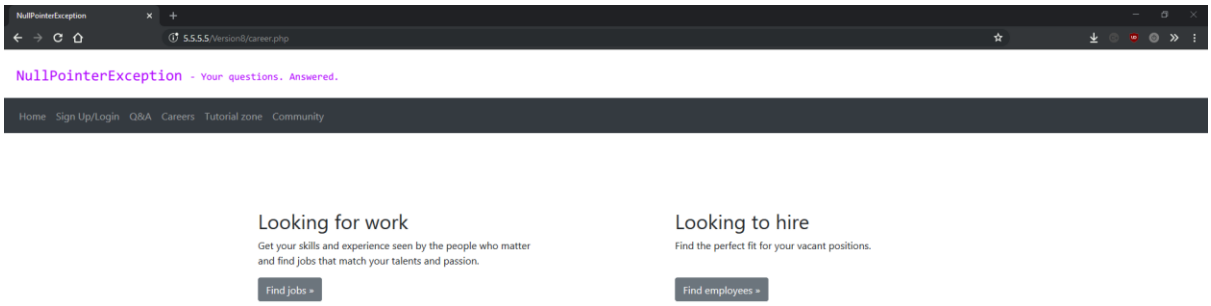


ask.php

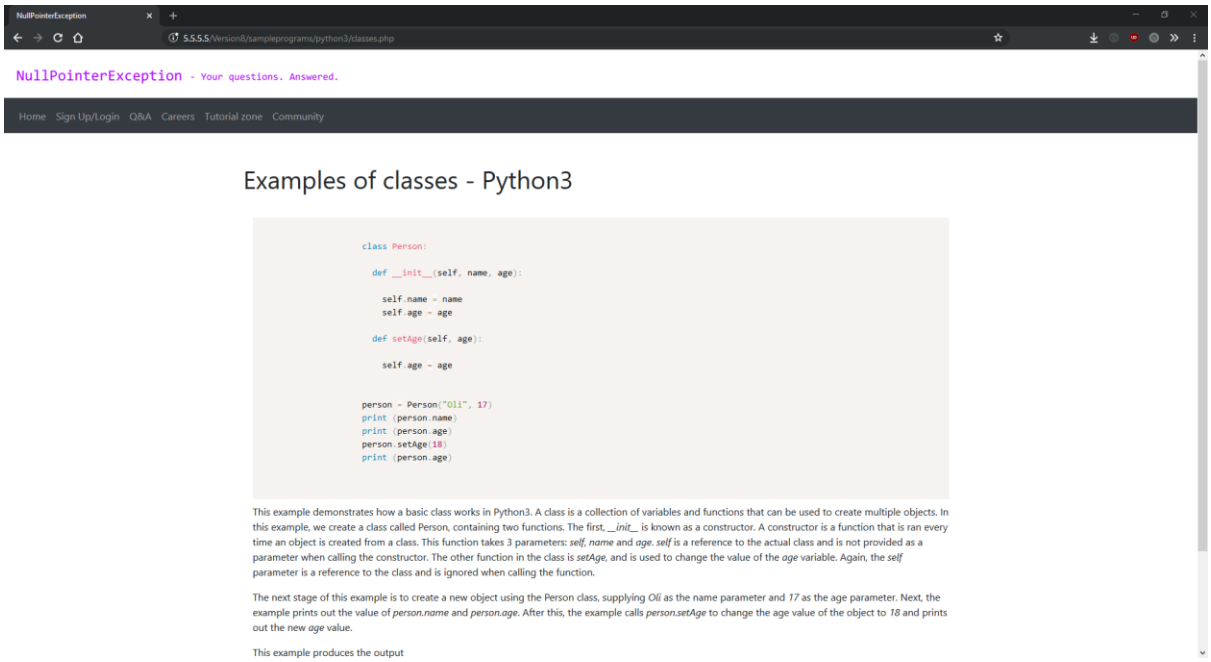


NullPointerException - Your questions. Answered.

career.php

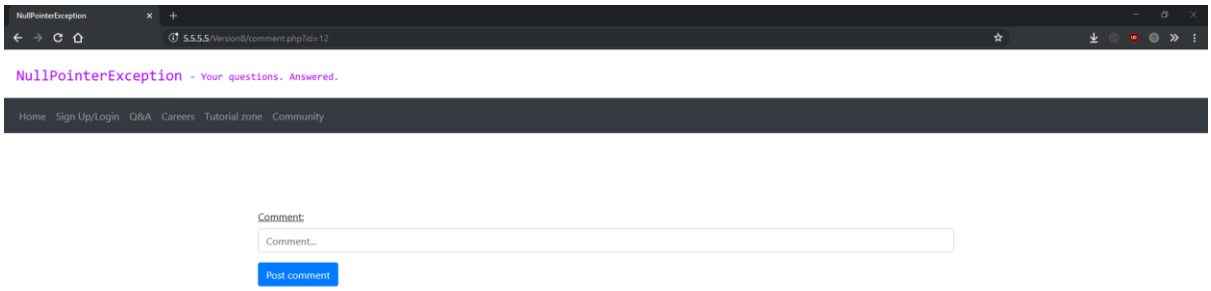


classes.php

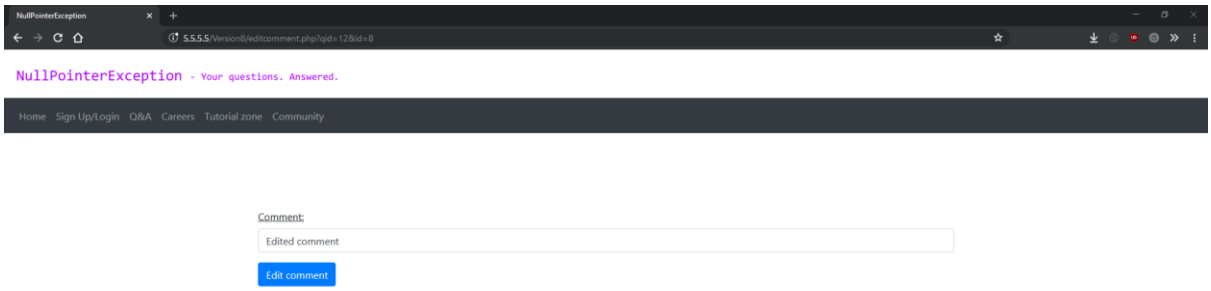


NullPointerException - Your questions. Answered.

comment.php

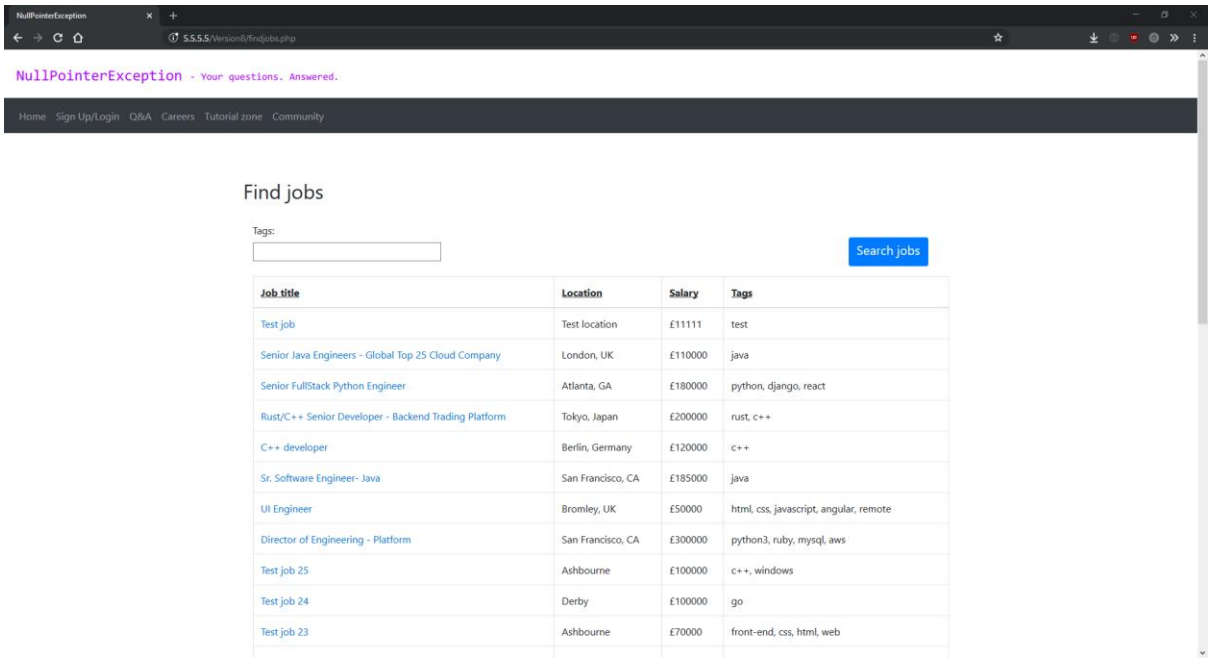


editcomment.php

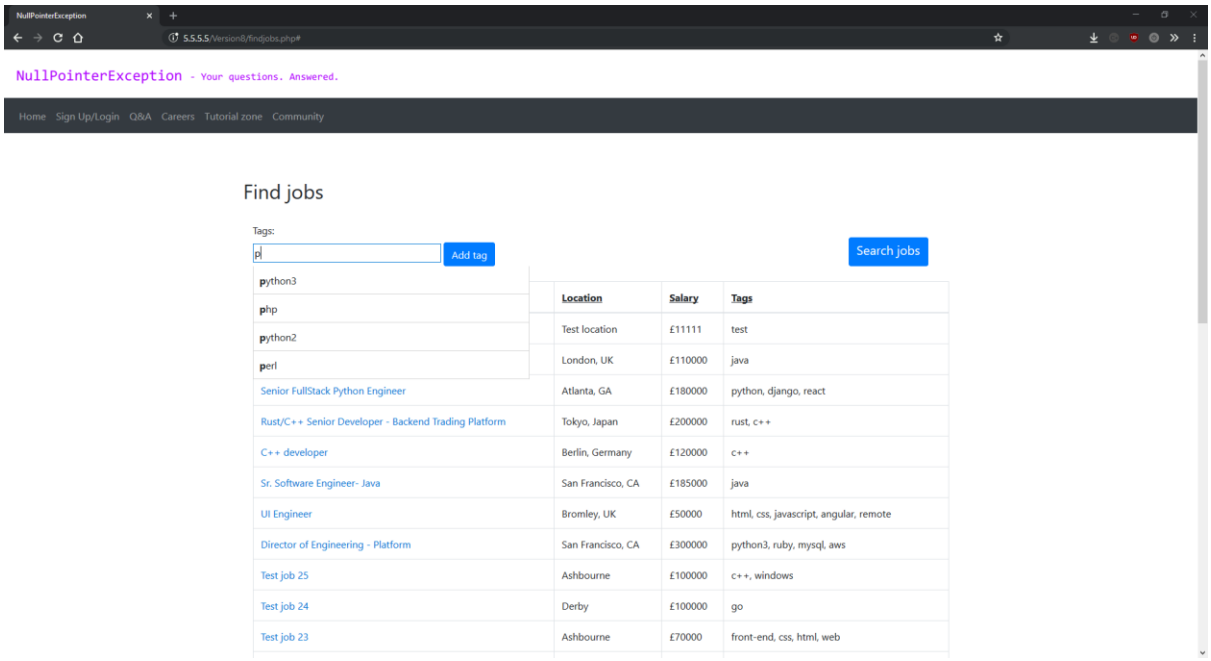


NullPointerException - Your questions. Answered.

findjobs.php

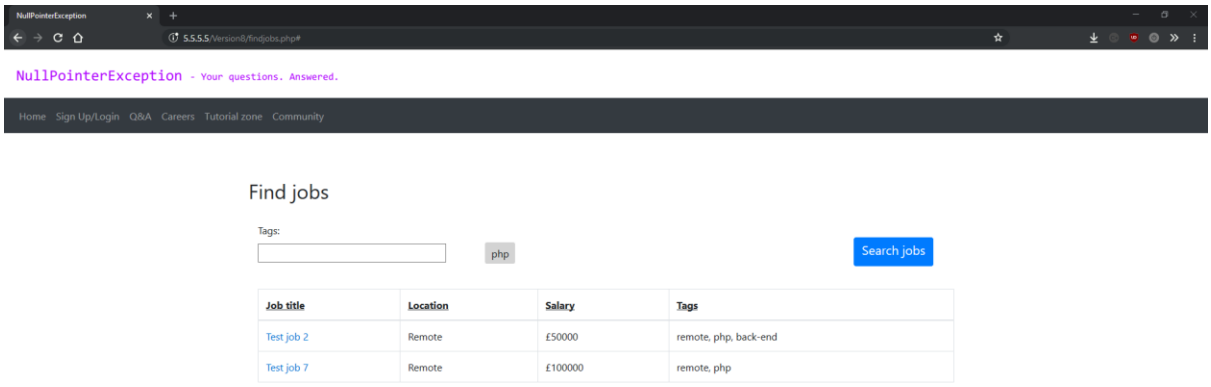


findjobs.php (with autocomplete)

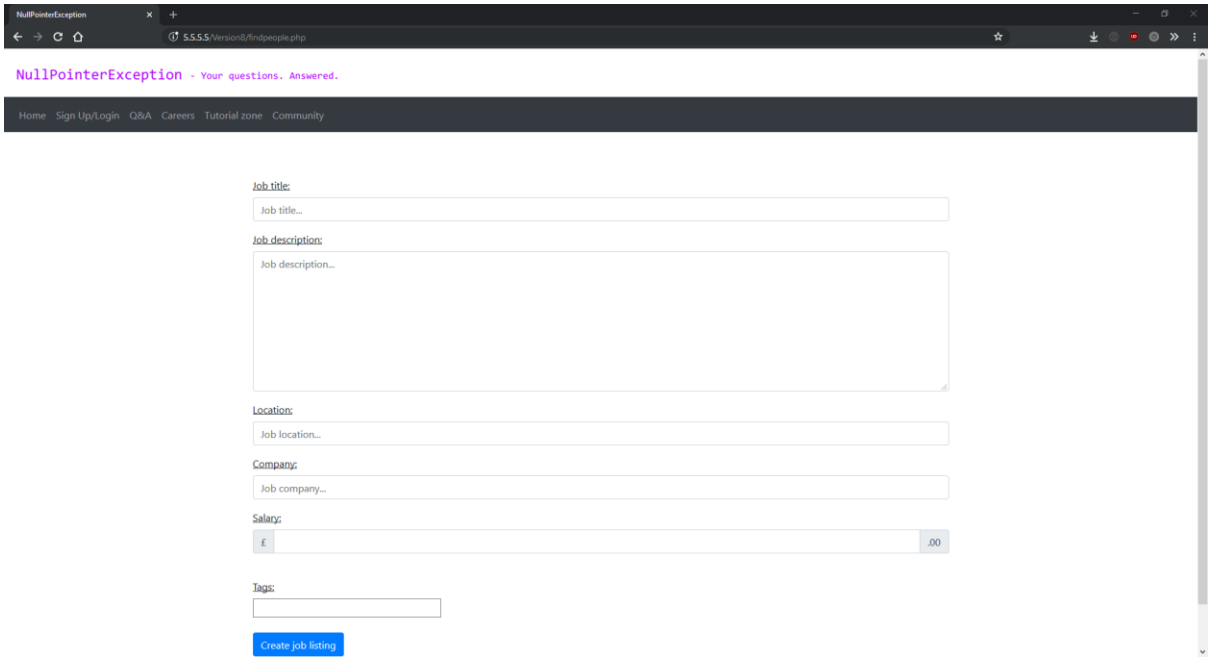


NullPointerException - Your questions. Answered.

findjobs.php (with tags)

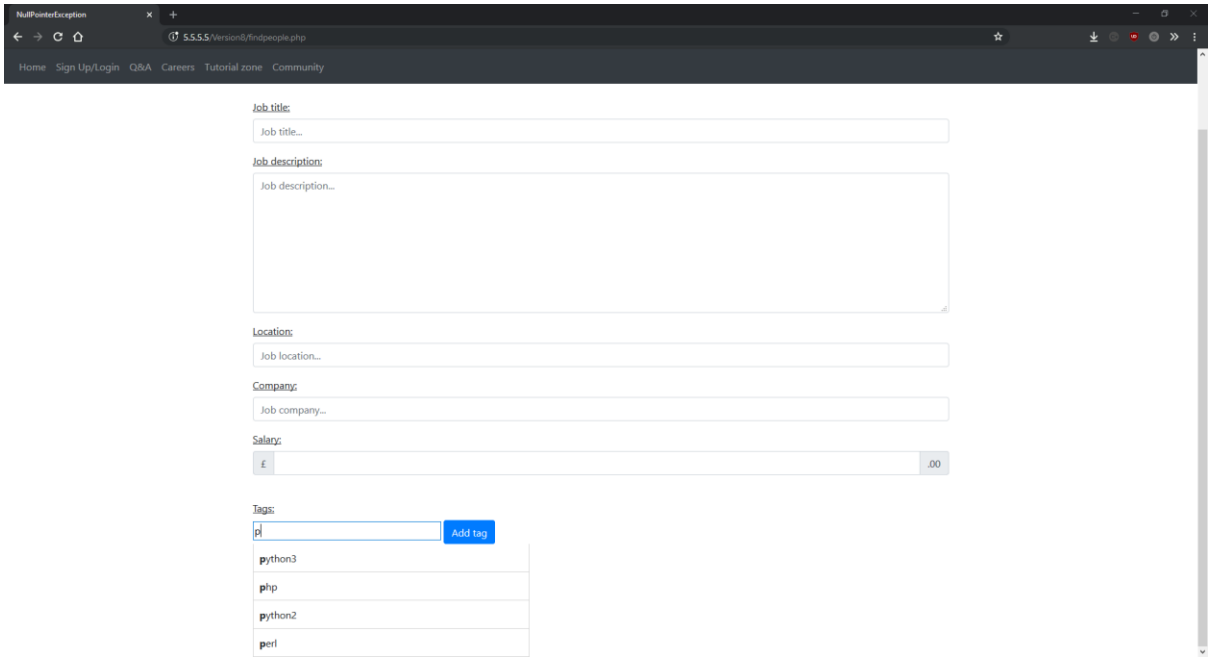


findpeople.php

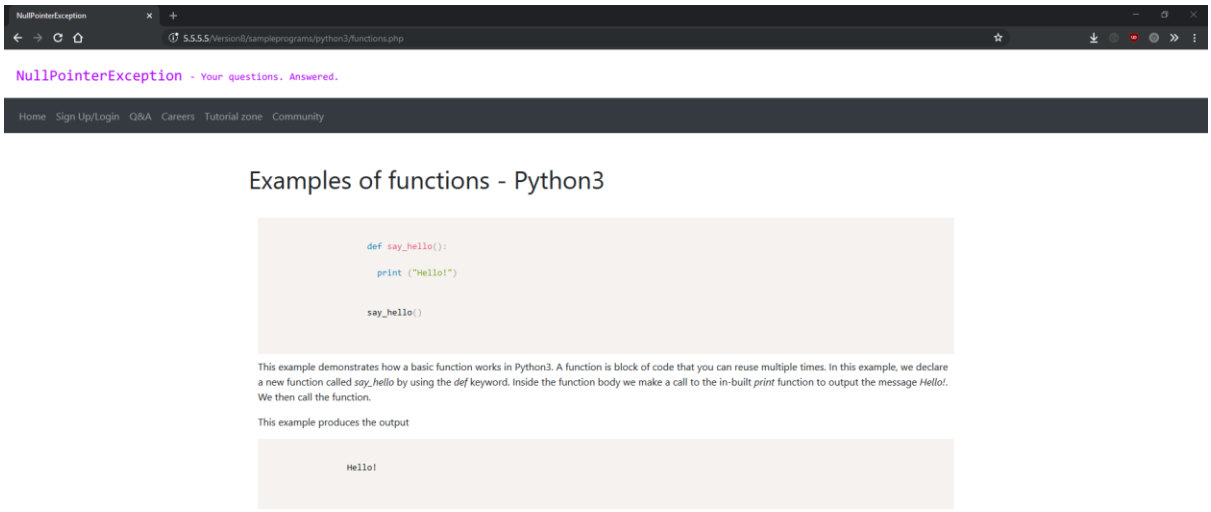


NullPointerException - Your questions. Answered.

findpeople.php (with autocomplete)

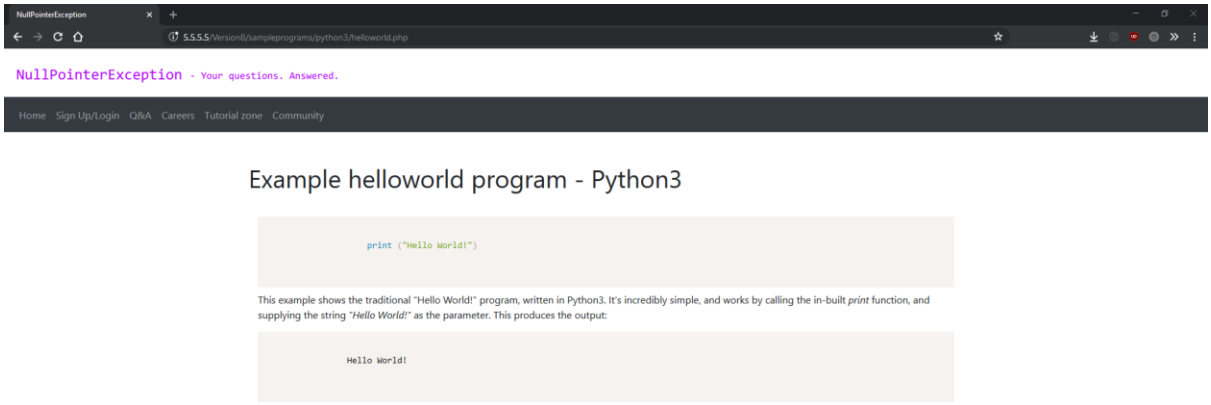


functions.php

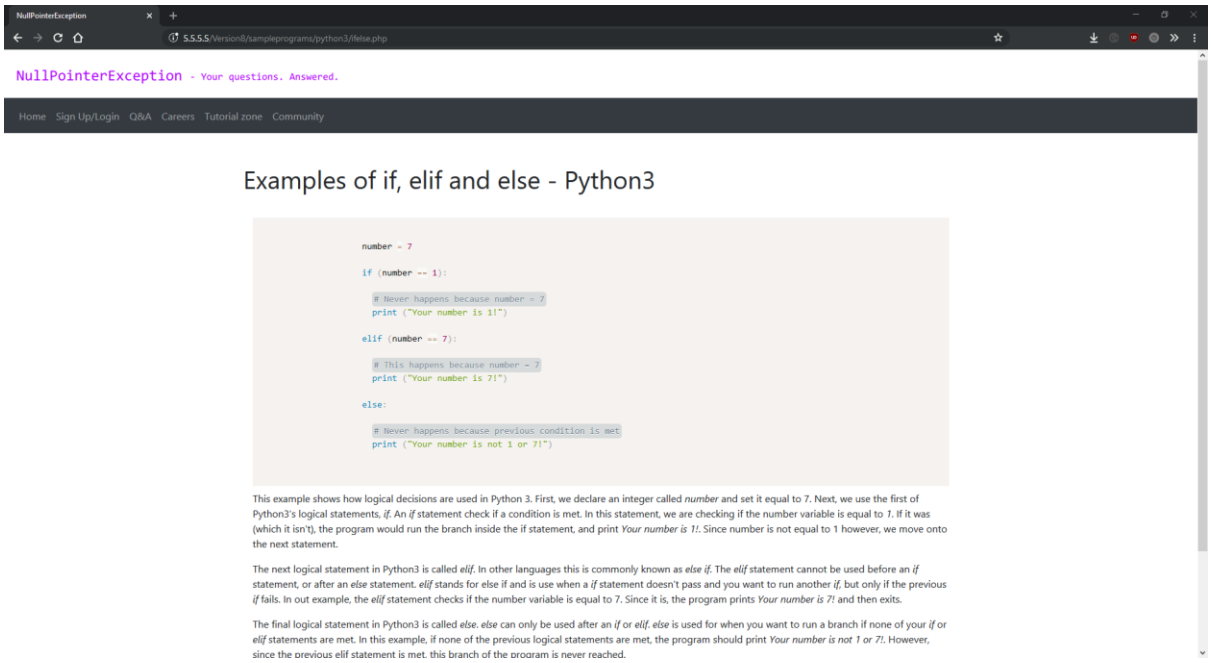


NullPointerException - Your questions. Answered.

hellowworld.php

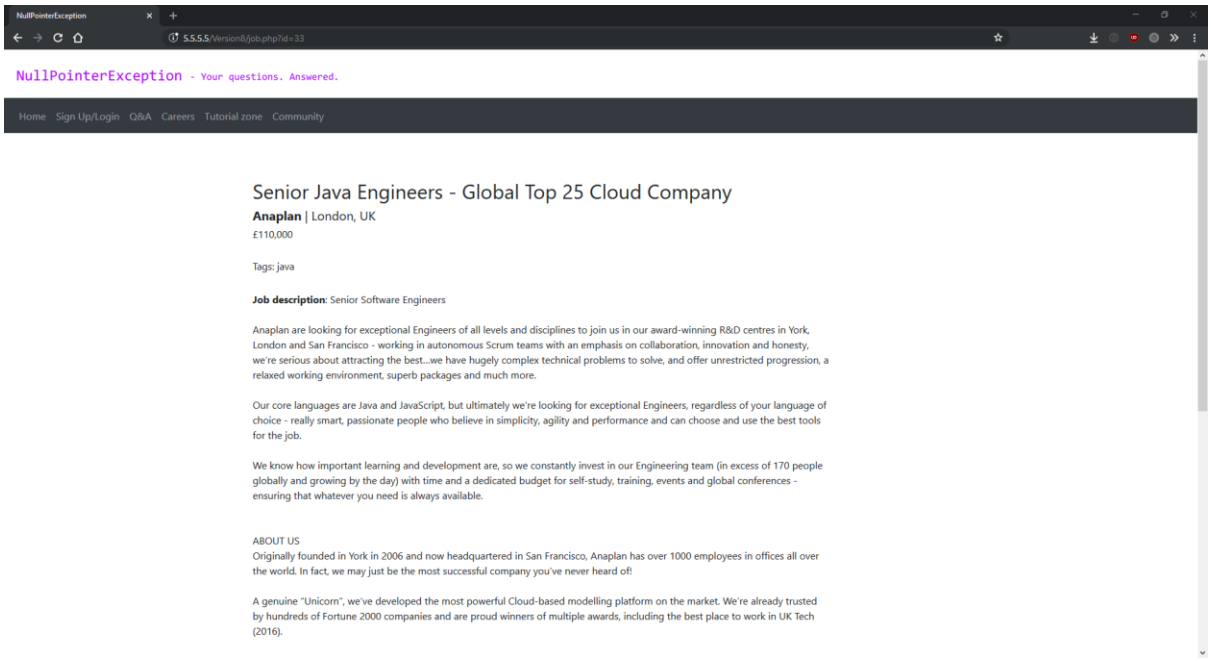


ifelse.php

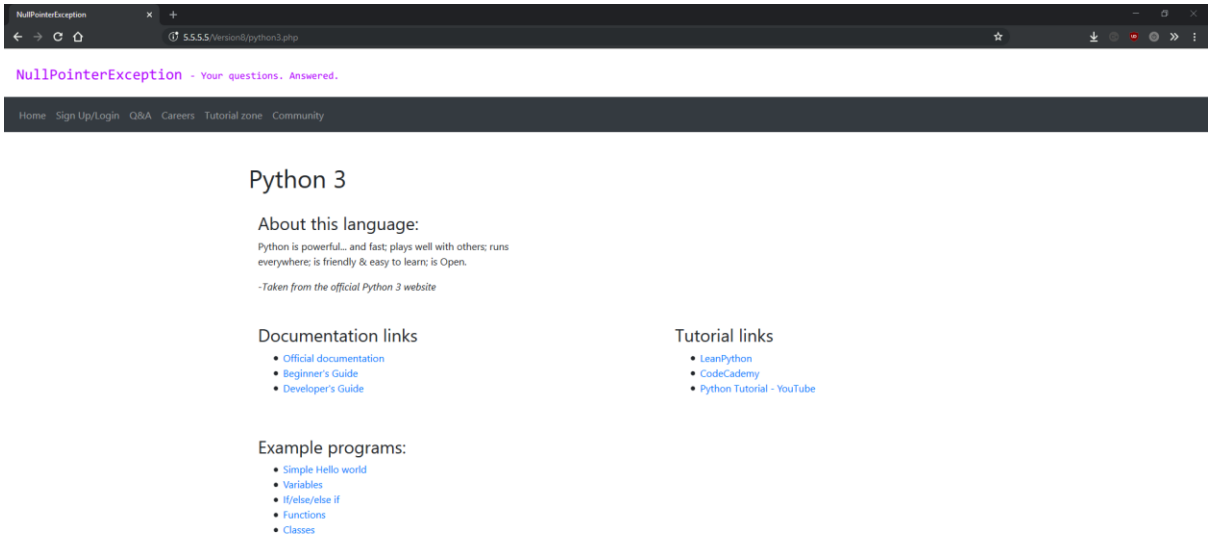


NullPointerException - Your questions. Answered.

job.php

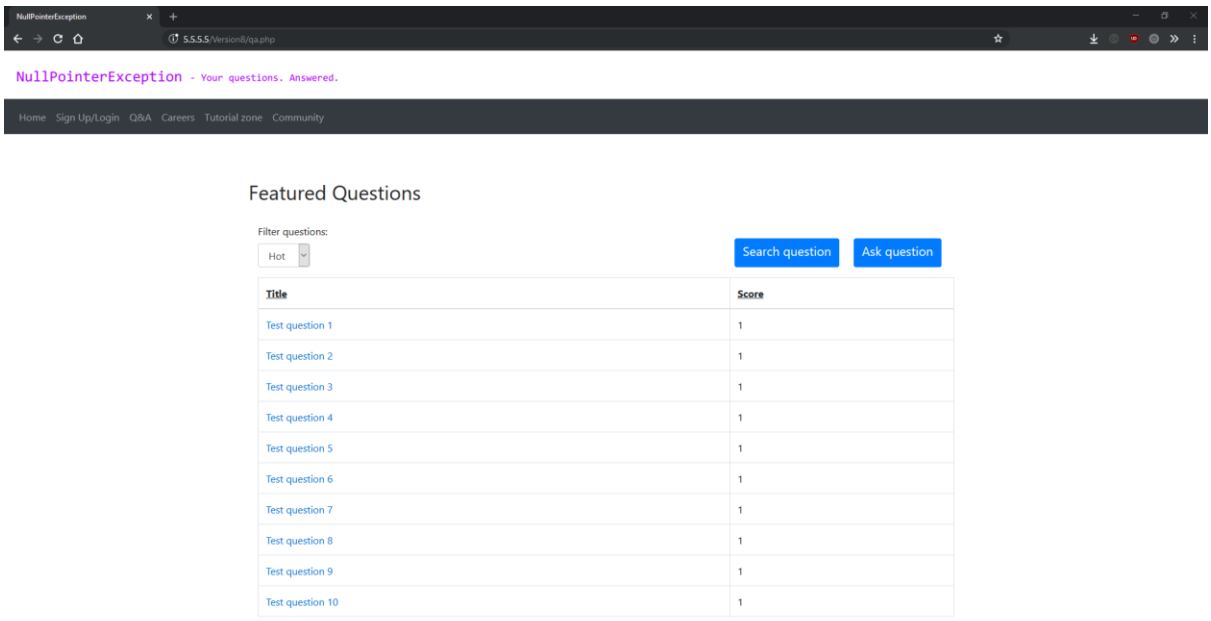


python3.php

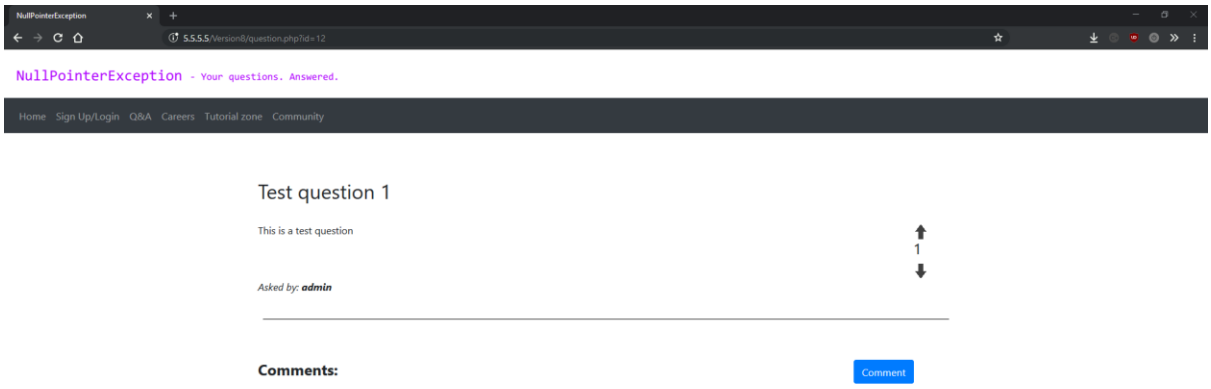


NullPointerException - Your questions. Answered.

qa.php

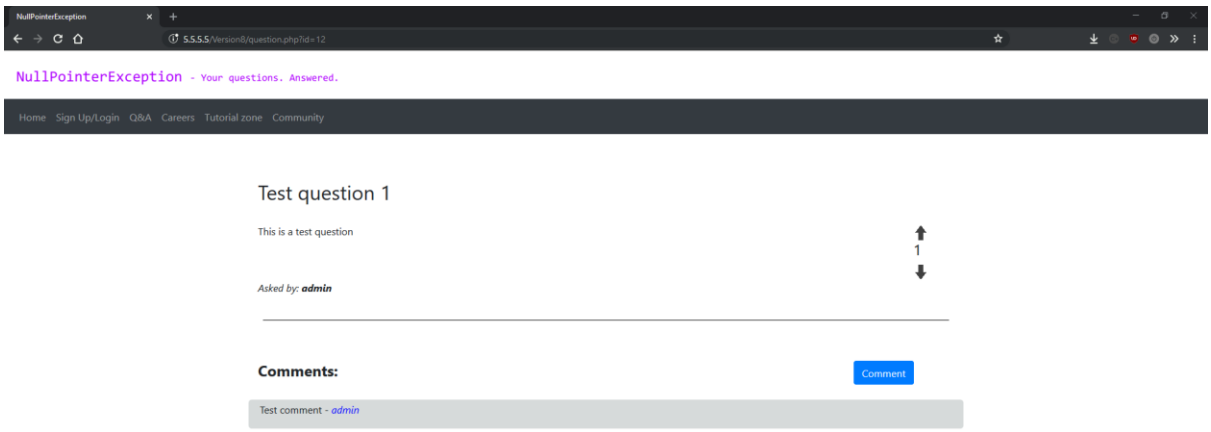


question.php

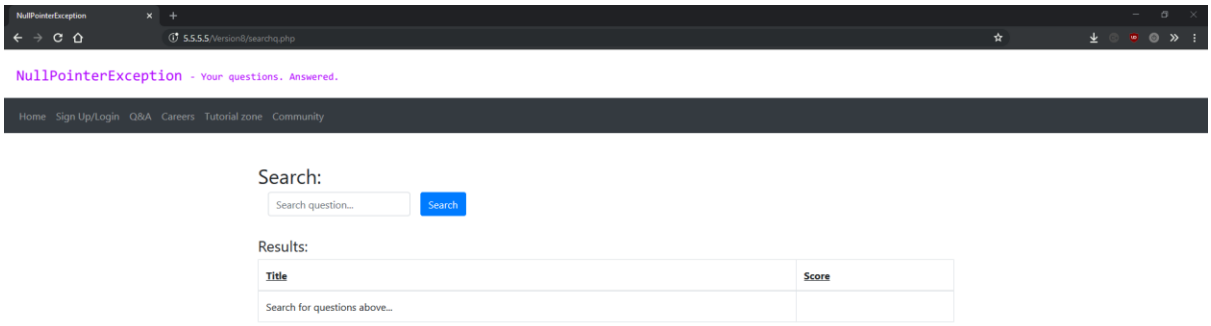


NullPointerException - Your questions. Answered.

question.php (with comment)

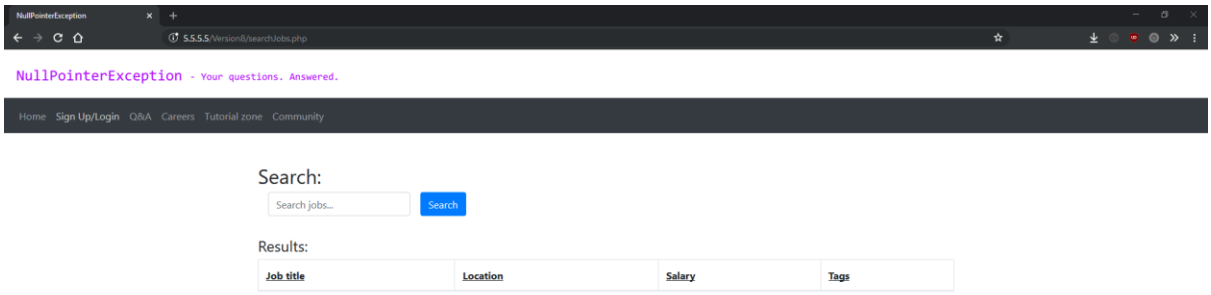


search.php

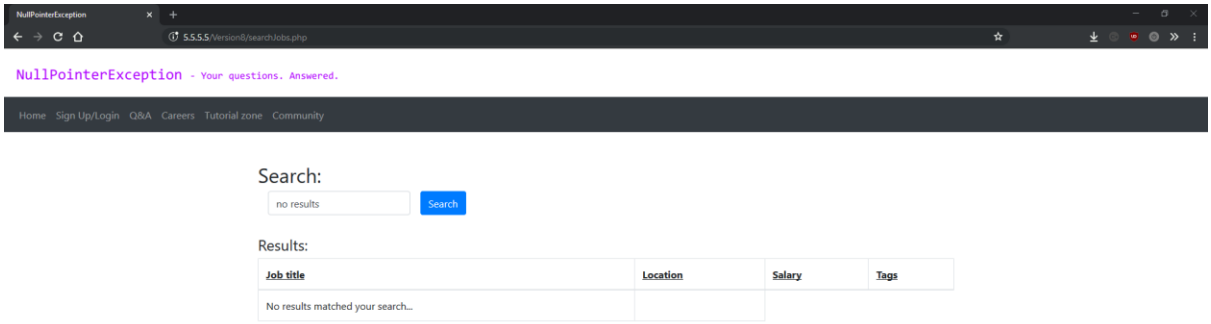


NullPointerException - Your questions. Answered.

searchjobs.php

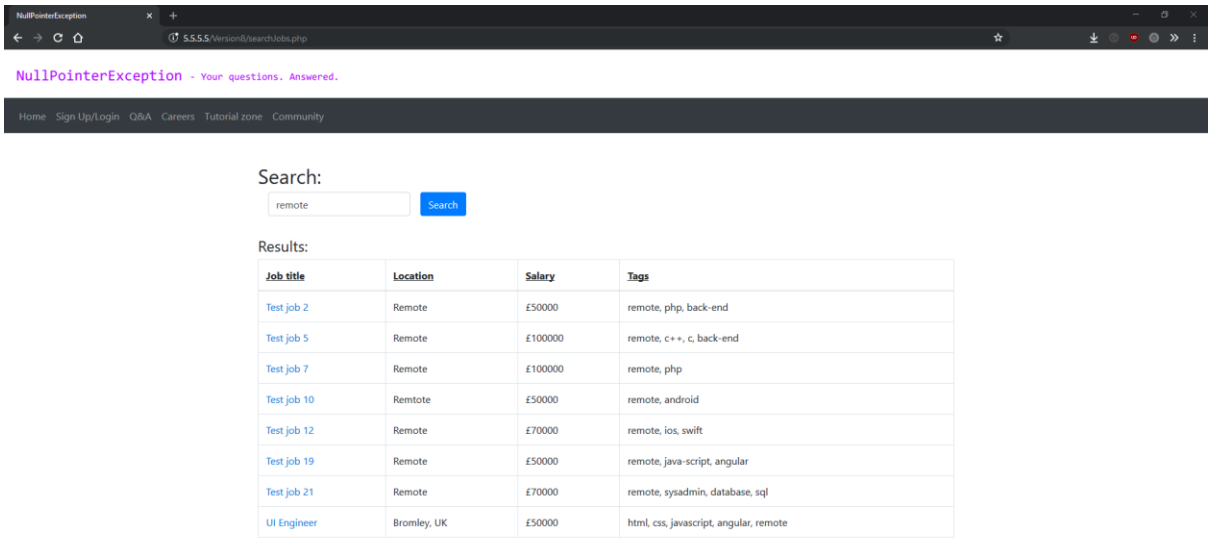


searchjobs.php (no result)

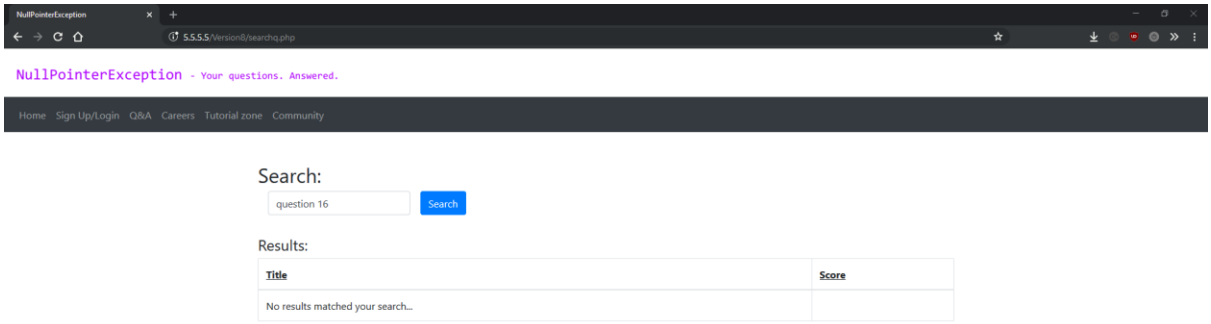


NullPointerException - Your questions. Answered.

searchjobs.php (with results)

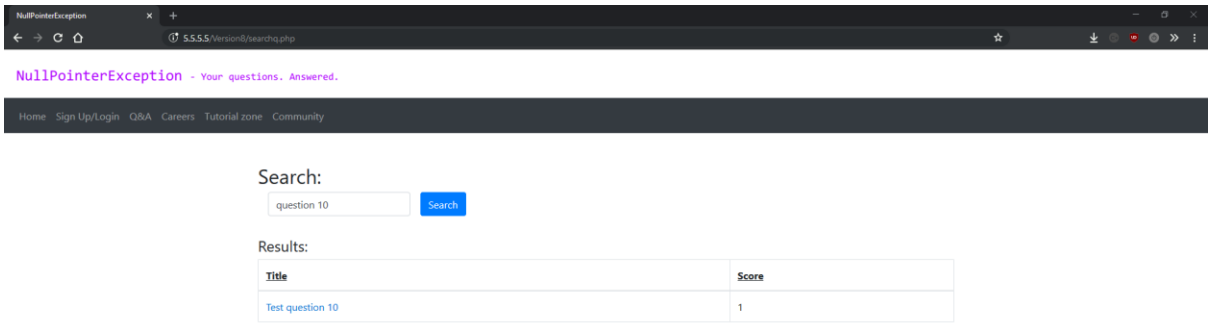


search.php (no results)

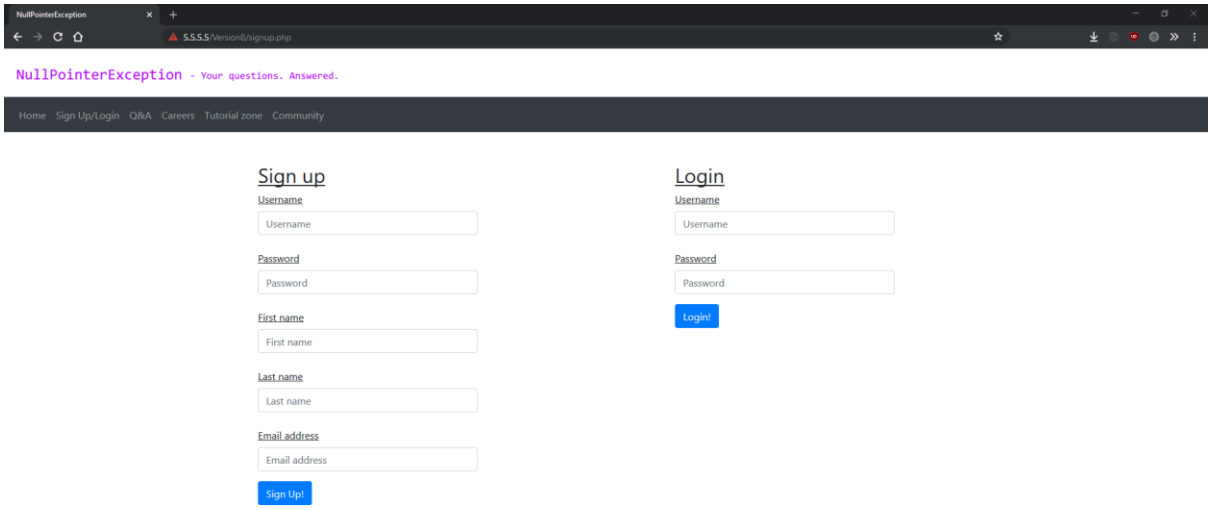


NullPointerException - Your questions. Answered.

search.php (with results)

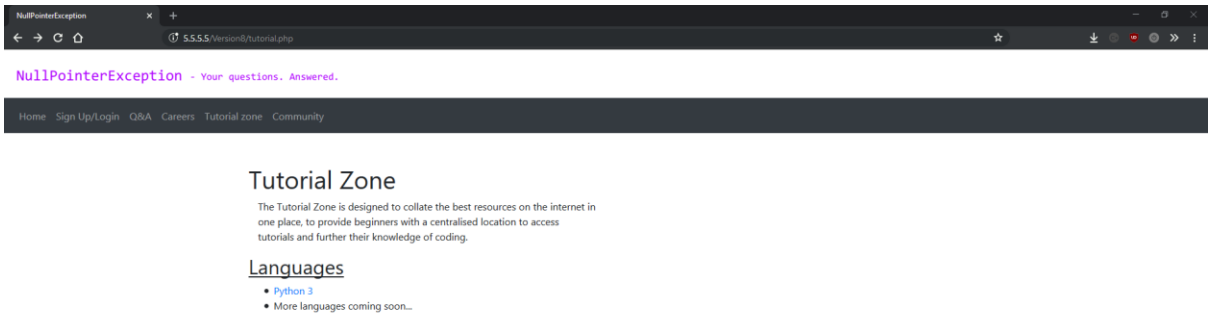


signup.php

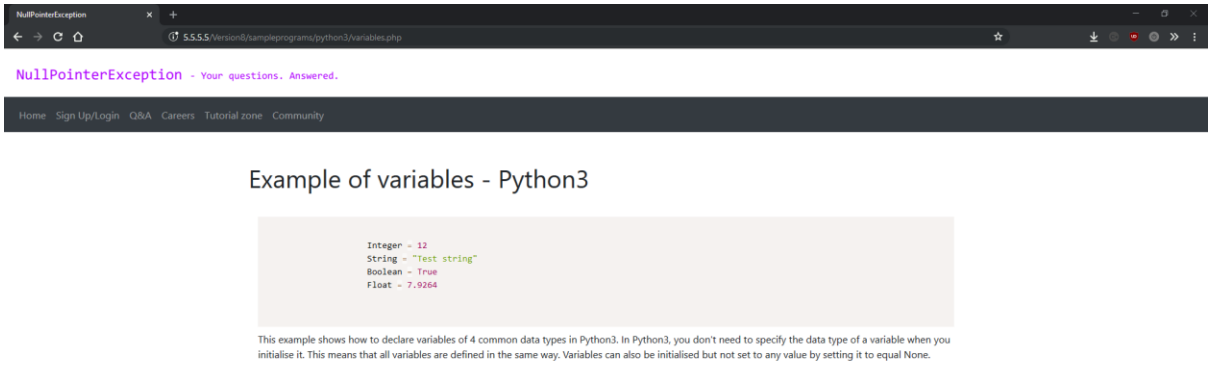


NullPointerException - Your questions. Answered.

tutorial.php



variables.php



Deployment

The development version of my project ran via vagrant in a headless Ubuntu virtual machine. Headless just means the virtual machine is running in the background with no GUI or command line, it can only be accessed by a SSH connection. This makes it faster as it uses less resources. The added benefit of this is the host computer runs faster because it's giving less resources to the virtual machine.

The production version of the website (version 8) runs on another headless virtual machine, but this time hosted by Digital Ocean. It runs Ubuntu Server and is used to host the website and the database. It's port-forwarded in order to allow incoming external connections (so people can visit it without being on the same local network). The web server I'm using is apache because it's powerful and easy to setup/install and the database is MySQL server because I prefer SQL over NoSQL solutions such as mongodb. I chose to run the project on a Linux server over a Windows machine for several reasons. One of the main reasons is that I use Linux on a day to day basis and prefer it over Windows. Linux is also the industry standard for servers, open source and free, which saves me spending upwards of £100 on a Windows licence.

When it came to hosting my site I had three choices: only run it on LAN, run a server of my local network and port-forward it, or use a virtual hosting service such as Digital Ocean or Microsoft Azure etc. I didn't want to run a server on my local network and port-forward it because it would leave my network open to attacks if not configured properly, and would slow down our already slow internet. I immediate disregarded the idea of only running it locally as no users except me would be able access it which would be completely pointless. This left me with using a virtual hosting service. I was initially going to use Microsoft Azure when I was first planning this project however since then I've started using Linux as my main operating system so I'd realised all the benefits in running a Linux server. It is also significantly cheaper, as a virtual machine with similar specification (bare minimum) on Microsoft Azure would cost £9 per month whereas the service I ended up going with cost £4 per month. I ended up choosing Digital Ocean because of the amount of control they give you over the virtual machine and low price.

The virtual machine starts with a clean install of Ubuntu server and a port-forwarded SSH connection for managing it. I then installed and set up apache, PHP 7 and MySQL, as well as some PHP extensions (xml and MySQL). I used MySQL workbench to connect to the server and setup the database. Finally I cloned the git repository into the apache directory.

I came across some unexpected issues when it came to deploying my project. One of these was that the apache directory structure on the production server was slightly different to the apache directory structure on the Vagrant development machine. This meant that the ini file used to store the database connection Strings is in a different location in the production server and the development machine, which meant I had to change the Database class to use the new path. The same is true for every single link on the website.