TAIORA TRIAL PWA

Taiora Trial PWA

Introduction

Mental health is a growing crisis. Rates of mental illness in children and adolescents increased three-fold from 2006/2007 to 2015/2016, with studies of NZ youth showing rates of serious depressive symptoms range from 12.1-13.9% and suicide attempts range from 2.7-6.5%, according to NZ Health Study *He Ara Oranga*. With current treatments not seeming to the job well enough, research is urgently needed to expand and explore new and better treatment options.

Technology offers a unique opportunity to reach many people for research in an affordable way. For this reason, psychologists at the University of Canterbury specializing in mental health and nutrition decided to undertake a study investigating the effect of micronutrients on emotionally dysregulated teenagers over the internet. This would expand the number of participants that they would be able to reach.

The psychologists originally wanted to develop a mobile app for this study; many developers will preach about how the convenience of mobile apps boost engagement. This was especially relevant as the target of the study was teenagers, who are in general most comfortable with using their phones. However, quotes from companies to develop such an app was in the range of $35,000! The psychologists were forced to settle with a website, which would provide less engagement but was a lot cheaper.

However, as one of these psychologists knew that I enjoyed coding, they approached me asking if I could possibly build them the app they wanted, which would be in companion to the website and would still be a useful asset to boost engagement. Understanding the importance of the research, I was only too happy to take on this challenge. Little did I know just how challenging it would be.

First Meeting with Stakeholders

My primary stakeholders are the psychologists at the University of Canterbury who run this study:

- Professor Julia Rucklidge. Professor Rucklidge is the ‘principal investigator’ of the Taiora Trial.
- Meredith Blampied. Meredith is Professor Rucklidge’s PhD student

To start on this project, I needed an idea of what functionality the app would require and how this would relate to the website. I initiated a meeting with Professor Rucklidge, where she explained details of the trial, the functionality of the website, and requirements of the app. This is what I learnt:

The Taiora Trial assesses emotional regulation in teenagers. Each teen, along with their parent, takes baseline, weekly, and end-of-trial surveys, with the surveys different for each group. The administrators could then retrieve this data from the website, and the data from each survey could be downloaded in a comma separated value (CSV) file to the clinicians after completion.

I suggested developing the app primarily for the end-user – that is, the teens and parents. Professor Rucklidge agreed, as the clinicians would still be able to access all of the data on the website and having an app for this would be more of an inconvenience when it came to data processing.

We also agreed on some other key requirements:

- The app must be bug-free
- Be easy to use

- Have the same look and feel as the website
- Have the same functionality as the website:
    - Log in/log out
    - Take surveys specific to account type (i.e. parent or teen)
    - View trial information
- Correctly insert survey data into the same database as the website
- Transfer and handle data very securely
- Send users notifications to complete quizzes
- Must work on both Android and Apple

We agreed that these last three requirements were very important. The data collected was very personal health data, thus it could not afford to be leaked. Additionally, as the whole point of the app was to boost engagement, push notifications would be a key aspect of the app as these have been shown to do just that, and users should be able to use it on all devices.

With a clear goal in mind, Professor Rucklidge gave me the details of the website developers so that I could contact them to find out further technical information. These two developers would soon become two more stakeholders in my project:

- Justin LeGrice. Justin built a website for a previous study, which the other developer used as a basis for the new Taiora Trial site. Justin works at the company that hosts the website and has access to the database of the site.
- Merlin. Merlin works for a web development company called Square Dot and was hired to build and design the website for the Taiora Trial.

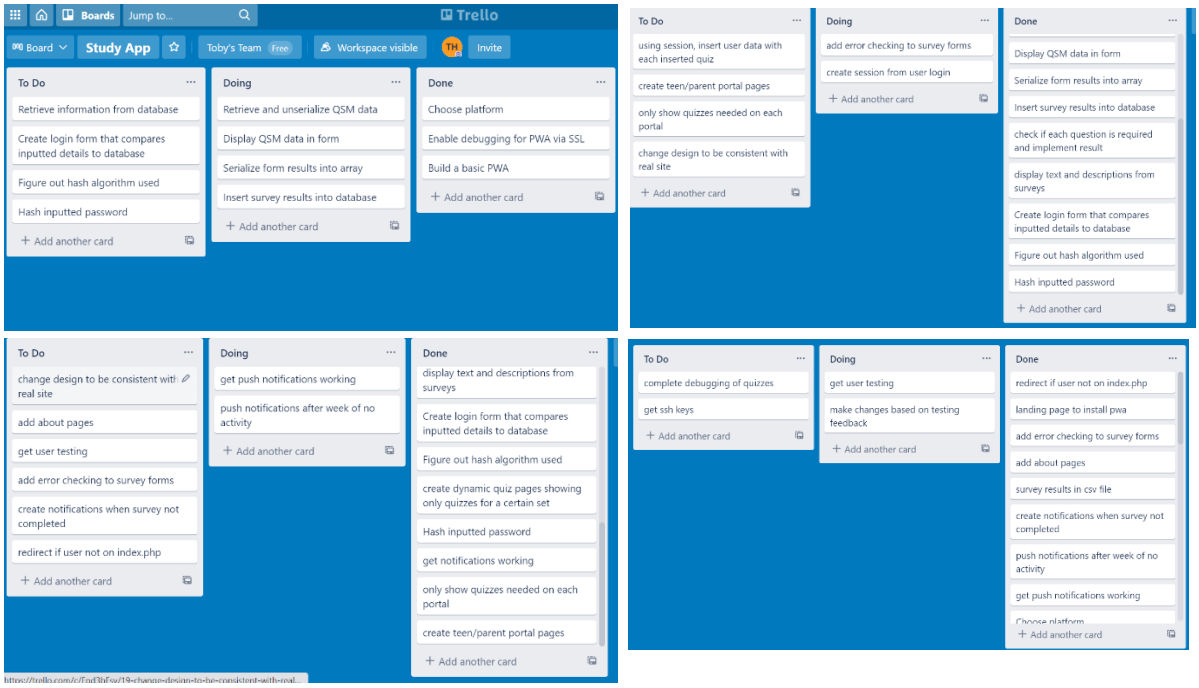
Other Stakeholders

It is worth noting at this point some other important stakeholders in my project:

- The teens and their parents who will be using the app. This app is primarily designed for these two groups, so keeping them at the forefront of my mind was paramount to doing the best job I could. Interaction with them would increase later when testing began.
- Mr Adams, my digital technologies teacher at school. Mr Adams would provide plenty of support and guidance throughout this project.

Project Management

Before starting anything else, I knew that for a large project with multiple different components (e.g. the quizzes, design, notifications or installability) it would be essential to have a robust planning scheme. For this, I used the program Trello. This project-management software would allow me to both break the project into smaller chunks and know what needed to be done. Additionally, this meant that if I got stuck or couldn't continue working in one area (e.g. If I was waiting for stakeholder feedback), I could easily pivot to another task that needed doing, reducing downtime and making me more efficient in development.



Getting Underway

To start the actual project, I contacted Justin with some questions about the website. According to Meredith, he is not very fast replying to emails so I texted him instead. Unfortunately, I do not have a screenshot of these texts as I have bought a new phone since then. However, I asked him for some information that I would need in order to interact with the website's database: where it was stored, what the table structure was,

Without a quick reply, I decided to start on work that I didn't need his help with, such as starting the app development. I had had no previous experience in app design, so I had a lot to find out about the different options. I learnt that I would either have to learn two different languages for each operating system (Swift for Apple and Java for Android) which would mean repeating the entire coding process twice, or use a cross-platform language such as Xamarin. I deliberated on these two options, but overall decided that as a solo developer, it would be much less time consuming to have to repeat the process with two different languages. I decided that I would begin teaching myself Xamarin.

A few days had passed, and Justin had still not replied, even after several follow-up messages sent to him and Meredith. I decided to contact Merlin instead to see if he could answer my questions. Thankfully, he replied promptly:

Good morning Toby,

Nice to meet you.
From your initial questions, it sounds like it might be more suitable for you to get in touch with Justin LeGrice who is taking care of the hosting for the project. He ran the original project.

My position is to recreate the site design, but otherwise to maintain everything the same as it was.

As a starting point, we are using Profile Builder for the user accounts (https://wordpress.org/plugins/profile-builder/) and QSM for the surveys / questionnaires (https://quizandsurveymaster.com/) - but that won't tell you much about databases.

As mentioned though, to know more about the database structures etc. it would be more suitable to speak with Justin since he has access to that side of the project.

I'll be speaking with Meredith (from the University) again soon so I can ask her about you reaching him. Or do you already happen to know him through Julia / Meredith?

Thanks,
Merlin

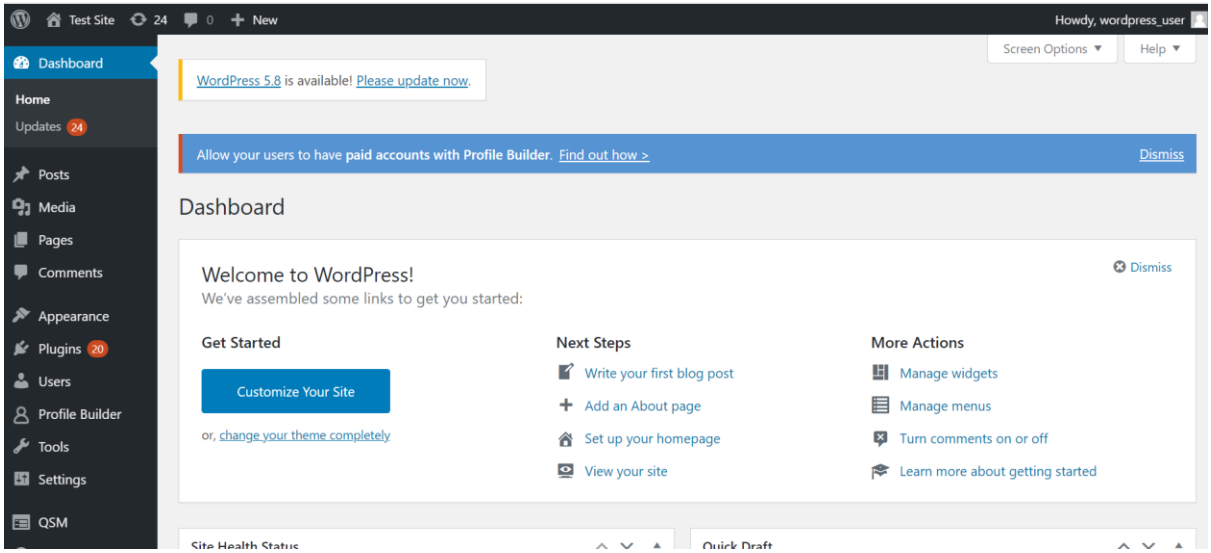
Although Merlin couldn't answer my questions (instead pointing me back towards Justin), I now learnt that the website was being developed with WordPress, which was an important piece of technical information. With Justin remaining unresponsive, I put learning app design to the side and decided to create my own test WordPress site to try and figure everything out myself.

WordPress Test Site

To do this, I wanted full access of the WordPress site and database. Therefore, I decided to test locally on an XAMPP server, which I had had experience with in the past. I followed a tutorial to do this, which included registering for, installing, and relocating WordPress to the local server on my device, setting up a new database in PHPMyAdmin, and altering several lines in the configuration file for WordPress – such as the ones below, which allow me to connect to the database. This also meant that I could view the database on PhPMyAdmin, a database administering software that I also had previous experience in.

```
// ** MySQL settings - You can get this info from your web host ** //
/** The name of the database for WordPress */
define( 'DB_NAME', 'wordpress' );

/** MySQL database username */
define( 'DB_USER', 'wordpress_user' );
```

The tutorial did not cover how to install plugins locally. However, with a bit of nosing around the WordPress file structure I found the folder wp-content/plugins and was able to download the plugins to this folder. After this setup, my WordPress site worked. Here is what the admin dashboard looked like locally:



I was only concerned with figuring out how these plugins worked. After some playing around I discovered this about the two plugins:

- Profile Builder was simply a front-end application to create forms for users to register, log out, etc, from their account. The data from this was inserted into default WordPress tables wp-users and wp-usermeta. I found that the wp-users table was structured like this:
- Quiz and Survey Master provided complete functionality to create and deploy surveys, as well as view results. They stored data in custom tables in the database.

#	Name
1	ID
2	user_login
3	user_pass
4	user_nicename
5	user_email
6	user_url
7	user_registered
8	user_activation_key
9	user_status
10	display_name

It seemed that understanding profile builder would not be necessary. However, interaction with the QSM databases would be vital to this project, as I would be directly retrieving and inserting quiz information from their tables. Thus, I set up my own quizzes up on the QSM panel (which can be seen in the bottom left corner in the above picture) to see how they stored their quizzes.

I could then view these on my WordPress database. However, I had no idea what I was looking at. Their data was stored in a very confusing manner which I will go over later. However, looking at it now, I had no idea how I was going to interpret this – let alone in a completely new programming interface.

Progressive Web App?

It seemed that untangling the survey information from the database would be a real challenge. Additionally, research told me that to do this from an app I would need to use the WordPress REST API, which would add another layer of complexity to database queries. Doing this in an unfamiliar language (i.e. using Xamarin) seemed like a daunting task. Furthermore, my experience teaching myself Xamarin over the past couple of days was frustrating. – I found that even basic tasks proved troubling to deploy on the two differing platforms. Looking on online forums for solutions for some of these, I found many people complaining of the difficulties of Xamarin. Furthermore, I talked with a

However, with all the online research of mobile app options, I learnt about what is called a progressive web app (PWA). This increasingly-popular application form is hosted on the web, but looks and feels like a native app and can be installed to a mobile phone's home screen. It could be developed in

html/css/php/js, which I had had experience in before. The more I thought about this the more it seemed like a good idea, which I summarized at the time in the below table of pros and cons:

	Progressive Web App	Native App
Feels like an app?	Yes	Yes
Installable?	Yes	Yes
Push notifications?	Android – Yes Apple – No	Yes
Easy connection to database?	Yes	No
Familiarity with language?	Yes	No
Time to deployment	Short	Long

One aspect that stands out in the above table is that Apple did not currently support notifications from PWAs. This was a major problem – 45% of people in NZ have an Apple Phone, meaning that we would not reach half of our participants with notifications. However, at the same time I was beginning to see that the learning curve for app development for what would be required in this app would be enormous. I arranged to meet with Julia to discuss this problem, where I explained the differences between the two platforms. She agreed that this was a let-down, but overall concluded that reaching even half of participants was better than none. Apple users would still have the app, which we felt would still increase engagement, especially in youth. She agreed that if she wanted an app developed at all for under $30,000, a PWA would be her best bet.

I confirmed that creating a PWA was remarkably simple by following an online tutorial. My steps (which I implemented myself for the real project) will be explained later. However, with the confirmation that this was possible, I was ready to start coding.

WordPress PWA?

However, soon into development a thought occurred to me: If a PWA is just an installable website, then could we simply make the actual website a PWA and be done with it? It was no surprise to learn that WordPress had multiple plugins to do just this. However, after some investigation I found some problems with this idea:

- While there were plugins to send push notifications, they offered little control over which users they were sent to (e.g. could not send them only to teen accounts). This meant it would also not be possible to send specific notifications to users who had not filled in their survey for that week. This meant that notifications would be rendered (at best) to a generic “Your surveys are ready for this week”.
- I learnt that the basic structure that Justin had used for the website for the previous study would be used again for this site. However, upon accessing this I found that it was not user-friendly: it took 5 clicks to access a survey from the home page. In an app that was supposed to maximise user friendliness to encourage users to fill in their quizzes, this was not ideal. Additionally, the website layout that would be natural for a WordPress site would not have the app-like feel desired in a PWA. This cemented in my mind that the WordPress website would not be suitable as a PWA.

Despite this, I still mentioned the alternate solution to Julia. However, after hearing my arguments against this, she agreed that this may not be the best and gave me the green light for continuing with my own app.

Quiz List & Decoding QSM

I will start by saying: the quiz and survey master database is an absolute mess. I decided to first make a dynamic list of quizzes which would link to each individual one. I first browsed through their custom tables, finding three of interest: wp_mlw_quizzes, wp_mlw_questions, and wp_mlw_results. wp_mlw_quizzes had 56 rows, which began structured like this:

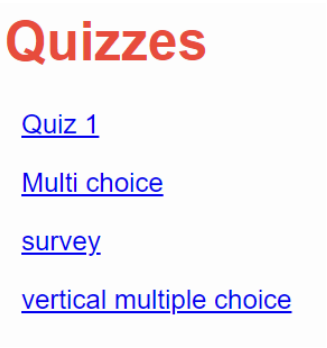
#	Name
1	quiz_id
2	quiz_name
3	message_before
4	message_after
5	message_comment
6	message_end_template
7	user_email_template
8	admin_email_template
9	submit_button_text
10	name_field_text
11	business_field_text
12	email_field_text
13	phone_field_text
14	comment_field_text
15	email_from_text

Thus, making a list of quizzes was not difficult. I first connected my PWA to the database:

```
$dbconnect = mysqli_connect("localhost", "root", "", "wordpress");
```

I could then perform a simple query on a page I titled quizzes.php:

```
// get quizzes
$sql = "SELECT quiz_name,quiz_id FROM wp_mlw_quizzes";
$qry = mysqli_query($dbconnect, $sql);

// display quizzes
while ($aa = mysqli_fetch_assoc($qry)){
  $quiz_name = $aa['quiz_name'];
  $quiz_id = $aa['quiz_id'];

  echo "<p><a href='index.php?page=quiz&quiz_id=$quiz_id'>$quiz_name</a></p>";
}
```



This created a dynamic list of quizzes, each linked to a page called ‘quiz.php’. In the $_GET array, I stored the Quiz ID so that I could dynamically display the quizzes on that page. I had created 4 quizzes using QSM on the WordPress dashboard, thus successfully had a list of quizzes.

Displaying Quiz

Displaying the individual quizzes is where it got interesting. I could see that most of the 54 columns of wp_mlw_quizzes correlated to a functionality of the plugin that we did not need (e.g. “show_score” or “leaderboard_template”). Thus, I was feeling confused about where QSM stored the actual quiz data. For example, I knew they must have a list of question ids for each quiz that could be updated as the quizzes are edited. However, under the column “quiz_settings”, this is what was stored for my quiz “Quiz 1”:

a:5:{s:12:"quiz_options";s:1411:"a:43:{s:9:"quiz_name";s:6:"Quiz
1";s:9:"form_type";s:1:"1";s:6:"system";s:1:"0";s:12:"progress_bar";s:1:"0";s:14:"score_roundoff";i:0;s:14:"require
_log_in";i:0;s:10:"pagination";i:0;s:11:"timer_limit";i:0;s:29:"enable_result_after_timer_end";i:0;s:27:"skip_valid
ation_time_expire";i:1;s:16:"total_user_tries";i:0;s:19:"limit_total_entries";i:0;s:19:"question_from_total";i:0;s:
20:"scheduled_time_start";s:0:"";s:18:"scheduled_time_end";s:0:"";s:16:"randomness_order";i:0;s:15:"randon_category
";s:0:"";s:21:"contact_info_location";i:0;s:21:"loggedin_user_contact";i:0;s:15:"comment_section";i:1;s:18:"questio
n_numbering";i:0;s:15:"store_responses";i:1;s:23:"disable_answer_onselect";i:0;s:17:"ajax_show_correct";i:0;s:24:"c
ontact_disable_autofill";i:0;s:21:"form_disable_autofill";i:0;s:22:"show_category_on_front";i:0;s:22:"enable_quick_
result_mc";i:0;s:17:"end_quiz_if_wrong";i:0;s:32:"enable_quick_correct_answer_info";i:0;s:25:"enable_retake_quiz_bu
tton";i:0;s:22:"enable_pagination_quiz";i:0;s:22:"enable_deselect_option";i:0;s:29:"disable_description_on_result";
i:0;s:34:"disable_scroll_next_previous_click";i:0;s:14:"quiz_animation";s:0:"";s:20:"result_page_fb_image";s:91:"ht
tp://localhost/wordpress-test/wp-content/plugins/quiz-master-next/assets/icon-
200x200.png";s:14:"legacy_options";s:0:"";s:12:"social_media";i:0;s:9:"user_name";i:2;s:9:"user_comp";i:2;s:10:"use
r_email";i:2;s:10:"user_phone";i:2;}";s:17:"quiz_leaderboards";s:28:"a:1:
{s:8:"template";s:0:"";}";s:9:"quiz_text";s:1823:"a:30:{s:14:"message_before";s:27:"Welcome to your %QUIZ_NAME
%";s:15:"message_comment";s:37:"Please fill in the comment box

The actual data was about 3 times this length, but in the interest of saving space I have not displayed it all.

This was what I was looking for – there was no other place the data could be. However, I had no idea what this mess I was looking at was. Luckily Mr Adams was there to tell me that this was a serialized array. He pointed me to a PhP function that would convert this (which was a string) to an array. But upon using this on the serialized array, I realized that it was multidimensional, and the function only unserialized one layer at a time. Thus, with each new array I came across I had to unserialize it.

By looking through the entire serialized array, I could see what could be some question IDs:

```
s:9:"questions";a:2:{i:0;s:1:"1";i:1;s:1:"2";}
```

However, accessing this in the mess of serialized array upon serialized array was much more difficult than I can describe. Eventually, through trial and error I found its path:

quiz_settings >qpages>*some unnamed array*>questions

Thus, I could retrieve the list of questions ids by unserializing one layer at a time:

```php
$quiz_id=$_GET['quiz_id'];

// get quiz data
$quiz_sql = "SELECT * FROM wp_mlw_quizzes WHERE quiz_id=$quiz_id";
$quiz_questions = mysqli_query($dbconnect, $quiz_sql);
$quiz_aa = mysqli_fetch_assoc($quiz_questions);
// sort through quiz data and get question ids
$quiz_settings = unserialize($quiz_aa['quiz_settings']);
$qpages = unserialize($quiz_settings['qpages']);
$qpages2 = unserialize($qpages[0]);
$questions = unserialize($qpages2['questions']);
```

I could then easily retrieve the questions from the wp_mlw_questions table.

However, what followed next was the same process… but with questions instead. Here I needed a lot more information such as the question type (of which it was either multiple choice, a number, or a short answer). I did this step by step including, but this file path I created for reference at the time demonstrates it:

>wp_mlw_questions (table)
answer_array (for multiple choice questions this serialized array holds answer choices)
>question_settings
question_title
required (boolean of whether the question is required or not)
question_type_new (the question type)

Thus, I was ready to display the quizzes. I would loop through the array 'questions', and retrieve the data for each question using MYSQL. Each question would be of an input type in an HTML form, which I would then send to a page using the POST array to be inserted.

```php
$question_settings = unserialize($question_aa['question_settings']);
echo $question_settings['question_title'];
echo "<br>";

$answer_array = unserialize($question_aa['answer_array']);
$question_type = $question_aa['question_type_new'];
```

To display each question, I had to first check which question type it was. Then, I could simply use the appropriate HTML form input type. I found which ID correlated to which question type by inspection. For multiple choice, I could loop through the answer_array and display each option as a radio input.

```php
// multichoice question type
if ($question_type == 0){
// display mutlichoice options
  foreach($answer_array as $answer){
    // option is each multiple choice option
    $option = $answer[0];
    echo("<input type='radio' name='$question_number' value='$option' $required>$option");
  }
}
// short answer question type
else if ($question_type == 3){
  echo "<input type='text' name='$question_number' $required><br>";
}
// number choice
else if ($question_type == 7){
  echo "<input type='number' name='$question_number' $required>";
}
```

How are you?
- Very good
- Good
- Neutral
- Bad
- Very bad

Please outline why this is:

Submit

Above, you will notice that the variable $required is echoed with each question type. At first, I considered having another if-else block to check whether a question was required and display the question accordingly. To make an HTML input type required, one need only include the attribute ‘required’. Thus, I had the idea that for each question, set a variable that was either ‘required’ or an empty string, and echo this as a parameter. This is what I have implemented above, and it worked great.

```php
// 0 = required, 1 = not required
$required = ($question_settings['required'] == 0 ? 'required' : '');
```

Inserting Records

I was able to successfully fill out questionnaires. Next, however, I needed to insert them into the database. As previously mentioned, Julia required this to be seamless with the existing system. QSM stored results in a wp_mlw_results table. Therefore I would need to match exactly what data (and their types) that were being inserted into here. If I managed to do this, then I hoped that this would display correctly on the WordPress admin panel. Below is the structure of wp_mlw_results.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	result_id	mediumint(9)			No	None		AUTO_INCREMENT
2	quiz_id	int(11)			No	None		
3	quiz_name	text	utf8_general_ci		No	None		
4	quiz_system	int(11)			No	None		
5	point_score	float			No	None		
6	correct_score	int(11)			No	None		
7	correct	int(11)			No	None		
8	total	int(11)			No	None		
9	name	text	utf8_general_ci		No	None		
10	business	text	utf8_general_ci		No	None		
11	email	text	utf8_general_ci		No	None		
12	phone	text	utf8_general_ci		No	None		
13	user	int(11)			No	None		
14	user_ip	text	utf8_general_ci		No	None		
15	time_taken	text	utf8_general_ci		No	None		
16	time_taken_real	datetime			No	None		
17	quiz_results	mediumtext	utf8_general_ci		No	None		
18	deleted	int(11)			No	None		
19	unique_id	varchar(255)	utf8_general_ci		No	None		
20	form_type	int(11)			No	None		

My plan was to create a variable for each of these rows, figure out what it was, and thus insert them all into the database. I could see that rows 4-8 were irrelevant thus set each to 0 as they would not be needed. I would deal with rows 9-13 once I set up user accounts. Through inspection of results I had filled out myself on the real WordPress, I could see that row 18 (deleted) had a default value of 0. Row 20 (form_type) was 1, which meant that it was a survey and not a quiz.

However, the other rows were not so simple. Rows 15 and 16 stored the time that the quiz was taken, but for some reason in different formats. A consultation of W3Schools told me how to create these.

```
$time_taken = date("h:m:s A m/d/Y");
$time_taken_real = date("Y-m-d h:m:s A");
```

time_taken	time_taken_real
09:04:44 AM 03/05/2021	2021-03-05 09:04:44

A StackOverflow post told me how to get the client's IP address for row 14. Row 1 (result_id) was auto increment. For row 2 and 3 (quiz_id and quiz_name), I decided to pass in this information in the GET array, as otherwise I would have to go through the steps of unserializing quiz information all over again. I decided to worry about row 19 (unique_id) later, and passed in the same string from a previous result. This dealt with a lot of meta information. However, I had not yet inserted any answers from the quiz. I had only one row left to deal with: row 17, quiz_results, which was (surprise) a serialized array.

Again, making sense of this gave me a headache. This time I would have to reconstruct its entire structure precisely. For much of the needed information, I could recreate (or ignore) it in a manner similar to what I did above. However, index 1 of quiz_results was an array of arrays for each question answer. It needed, among other things, the question ID, type, and the actual answer. This was a problem – my HTML form simply posted the answer for each question. I needed a way to also include this other information with each result. With some testing, I also realized that answers that had not been filled in were not included in the post array – a problem as WordPress *did* have these answers, which I would thus need as well.

My solution was not the most elegant, however it solved both these problems at once. For each question, I included a hidden input type. This would have an ID of the format 'type[question_number]' (where question number incremented by 1 for each question and did not correlate to the question id) and a value of… a serialized array of this needed information. I had become the thing that I swore to destroy.

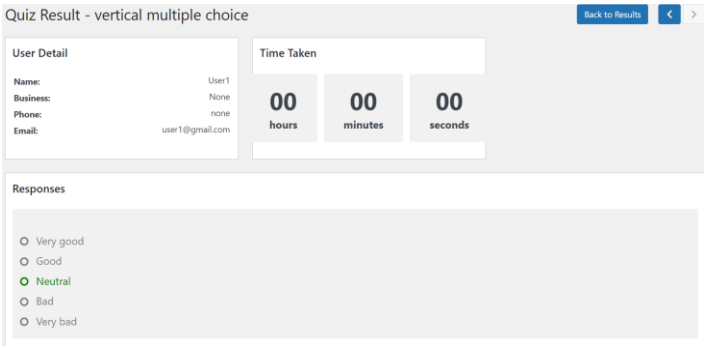
```
$info = serialize(array($question_id, $question_type));
echo "<input type='hidden' name='type$question_number' value='$info'/>";
```

Thus, I could then check if this hidden input existed, allowing me to get all of the questions no matter if they had been filled in or not. Here was my initial code to create the array for each answer:

```php
while (array_key_exists("type$question_number", $_POST)) {

  // info from hidden input
  $question_info = unserialize($_POST["type$question_number"]);
  $question_id = $question_info[0];
  $question_type = $question_info[1];
  $question_name = $question_info[2];

  $question_answer = $_POST["$question_number"];

  $question_answer_array = array(
    0 => '',
    // this values seems to be the actual answer
    1 => "$question_answer",
    2 => '',
    3 => '',
    'correct' => 'correct',
    'id' => "$question_id",
    'points' => 0,
    'category' => '',
    'question_type' => "$question_type",
    'question_title' => "$question_name",
    'user_compare_text' => ''
  );

  array_push($all_answer_arrays, $question_answer_array);

  $question_number += 1;
}
```

I appended each of these to one array, which was then added to a quiz_results array along with all of the other needed information. This was then serialized, and along with all the other information created earlier, inserted into the database using MYSQL.

With some anticipation, I tested out my code. Previous attempts had given me errors over incorrect input types. However, I was shocked to see a message saying 'records inserted successfully'. I checked first PhPMyAdmin and then the WordPress dashboard. Here is what I saw on the recent results page:



All of information here matched the temporary values that I had used. It seemed that my code worked.

Reconnecting with Stakeholders

Writing all this code was largely an individual process. I was able to develop this locally so had little contact with my stakeholders. Thus, I decided it was important to give a full update to Julia on how my progress was going.

I also needed to test out my code with the actual database – after all, there would be no better testing than with the real thing. I preferred to contact Merlin as I knew he was a keen replier over Justin and thought that he would have access to some kind of PhPMyAdmin panel for their database. He could then export the data and send it to me. However, an email conversation told me otherwise:

To: Toby Harvie Cc: Julia Rucklidge

taioratrial.WordPress.2021-05-0...
584.72 KB

Hi Toby,

I've been trying to get a copy of the entire site for you using a plugin but am running into a dreaded host error (i.e: the host settings are causing an error in the files being built).

I'm looking into a further option so hopefully that works.

Either way, I've exported the basics and attached them below. This is essentially what is included in the XML file, which I'm not sure will help you further - but at least you have it to test and see if there's any value there for your project:

"This will contain all of your posts, pages, comments, custom fields, terms, navigation menus, and custom posts."

I'll reach out and let you know if I get any further with the entire duplication at all.

Regards,
Merlin

Hi Merlin

This is looking good so far. However, what I'm looking for is the database from the site, which would include all of the surveys. To do this, from my experience, I had to find the wp-config.php file, change some values and then log in to phpMyAdmin. However, it sounds like you don't have access to this file – if you did, you could just send it and all the other files through. It really seems that Justin is the only one who can do this. However, if you have any other ideas then I am happy to hear them.

Thanks,
Toby

Hi Tony,

Let me see if I can find another way to access what you're needing.
It might continue to be a combination of plugins etc. to try and get it.

Local dev is so much more enjoyable with that since everything is in your own space.

I'll get back to you with what I find.

Cheers,
Merlin

Authentication

In this waiting time, I decided to start working on some other important tasks – namely, authentication and logging in. Because accounts were created by the university, I would only need the functionality

to log in or out. I again referred to the WordPress database – the table wp_users had all I needed. Most importantly, I tried figuring out what hash system WordPress used so that I could replicate this. For example, here was a password that I was faced with:

user_login	user_pass
wordpress_user	PB4HZ0Y7Ak5FMrtIkg/sgZCQuYuVDzU/

I consulted multiple forums to try and figure out what hash this was, or perhaps some code that could replicate it. I spent a long time scratching my head as nothing anybody suggested seemed to work. Finally, I found a StackOverflow post that mentioned a file under the hood of WordPress: (https://stackoverflow.com/questions/9854480/using-wordpress-user-password-outside-wordpress-itself). Buried deep in WordPress's files was a script called class-phpass.php, which was in the public domain so I could copy to my project. I then modified a function from the post to verify that a password is correct (rather than hash one in the first place, which the function did). Essentially, this function utilizes a function within the WordPress hasher class that checks if a password matches a hash – functionality can be read through the comments.

```php
function wp_check_password($password, $stored_hash) {
  // this code gets the phpass class from the file class-phpass.php (copied from the public domain)
  // we can use this to compare passwords.
  // this function taken from https://stackoverflow.com/questions/9854480/using-wordpress-user-password-outside-wordpress-itself
  // modified to check password instead of create

  // argument password is user inputted password, and stored_hash is the hash in the database.

      // WordPress's class
      global $wp_hasher;

      // if class does not exist (because function has not been called yet), create wp_hasher
      if ( empty($wp_hasher) ) {
          // embed file
          require_once('class-phpass.php');

          // By default, use the portable hash from phpass
          $wp_hasher = new PasswordHash(8, TRUE);
      }
      // will return true if hashes match
      return $wp_hasher->CheckPassword($password, $stored_hash);
}
```

I could then create a simple login form which would compare the inputted password with the stored hash for the username (if such a user existed in the database). It also added error catching and more security features such as real escape strings when interacting with the database. Additionally, upon logging in I retrieved important user info that would be needed (such as user ID, username, name and email) and stored them in session variables. These were unset by a logout button. Because this was an end user app with functionality only for users who were already signed up, no pages would be able to be accessed without being logged in. Therefore, I changed my site to a one-page website, and created a condition that would redirect users to the login page if these session variables were not set:

```php
if ($_GET['page']!='login' && $_GET['page']!='logout' && !isset($_SESSION['user_ID'])){
    header('Location: index.php?page=login&error=loggedout');
  }
```

The first two conditions prevent infinite redirect loops.

At this point, I made sure to do some testing to ensure that this system worked. I had a couple of classmates use the system. It turned out that my code was quite robust and they couldn't produce any errors.

Connecting with the real database, and a multitude of problems.

By this time, Merlin had managed to download the database for me. He used a plugin that created an SQL file of the WordPress database and sent this to me. I could then easily upload this into PhPMyAdmin and change which database the app connected to. I did have to create a new user by copying rows from the old table because my new login system worked too well and I could not access any quizzes without being logged in, and of course I did not know any of the passwords to the users in this database. However, I was then ready to see if my QSM code worked.

With some trepidation, I logged in and navigated to the list of all surveys. This displayed correctly – I could see a gigantic list of surveys. However, most individual quizzes had some sort of problem.

The most prominent of these was a notice that seemed to appear everywhere, for example:

Notice: unserialize(): Error at offset 1463 of 5172 bytes in **C:\xampp\htdocs\2021-year-13-web-design-classwork-tha7996\study\quiz.php** on line **11**

It was clear that this was a problem with serialized arrays. Research told me that this was caused by some corruption in their format: for each piece of data in a serialized array, an integer stores their length. If this is incorrect, this error will occur. This meant that the data in the actual database would have to be corrupted. For a while I was confused how this would have happened. However, many other forums asking the same question told me that this was a common issue when migrating a WordPress database. Quick fixes did not work, thus I had to assume that when I connected to the real database, this error would not occur.

Another problem was that in some cases, the survey required text in the middle that did not correspond to a question. However, I found that this had been done by putting the desired text as a question name for a multiple-choice question and having no question answers (presumedly because QSM offered no option to do it another way). Thus, my code could still display this but did so in an unappealing format, for example:

Your safety is of the utmost importance to us. If you are concerned about your ability to keep yourself safe, these services can help:

Youthline (Available 24/7) Free call 0800 376 633 | Free text 234 | Webchat (Youthline - Youth Health Services, Youth helpline Program Centre NZ - Youthline NZ)

Lifeline (Available 24/7) 0800 543 354

The Lowdown (Available 24/7) 0800 111 757

Healthline (Available 24/7) 0800 611 116

Samaritans (Available 24/7) 0800 726 666

Suicide Crisis Helpline (Available 24/7) 0800 543 354 (0508 TAUTOKO)

Crisis Resolution Services (Available 24/7) (Canterbury only) 0800 920 092

It was clear that line breaks were being stored as tags in these serialized arrays, but was printing as text. W3 schools gave me the fix. Whereas previously I was simply echoing the question name, what I needed to do was use a special function:

```
echo html_entity_decode($aa['question_name']);
```

Additionally, I first had to check whether a multiple-choice question had answers – if it did, I displayed the question as normal. If not, then I would use this function to display the text. This successfully worked:

Your safety is of the utmost importance to us. If you are concerned about your ability to keep yourself safe, these services can help:

Youthline (Available 24/7) Free call 0800 376 633 | Free text 234 | Webchat (Youthline - Youth Health Services, Youth helpline Program Centre NZ - Youthline NZ)

Lifeline (Available 24/7) 0800 543 354

The Lowdown (Available 24/7) 0800 111 757

Healthline (Available 24/7) 0800 611 116

Samaritans (Available 24/7) 0800 726 666

Suicide Crisis Helpline (Available 24/7) 0800 543 354 (0508 TAUTOKO)

Figuring this out required some rummaging around in the admin panel for the questions. While doing this, I discovered possibly the most frustrating problem: in some cases, question answers or titles did not exist at all – whoever had created the quizzes had clearly not finished. What I discovered was this also resulted in those pesky serialization errors from above, as I was attempting to unserialize data that did not exist. This caused most of the serialization errors that I thought were caused by migrating the site.

It seemed that the surveys had not yet been finished. This was annoying, as I could not thoroughly test each survey until they were. I contacted Julia, who told me that Meredith was creating and updating the surveys, but they had no real timeframe of when they would be finished, apart from before the start of the study which was around August (it was currently May).

However, in this meeting Julia revealed that the actual site would be live in just a couple of days. This would enable her and Meredith to test it out and make suggestions to Merlin, as well as begin to attract participants. For me, this was great: I would be able to compare my site to the real thing to test out the similarities and create a design consistent with the site. However, I had some more back-end to create first that was essential for the design.

Concluding this section, this meant that as far as I could tell the quizzes were error free. However, I would have to wait for the final database to check this completely.

Account types

One important aspect of the site was account types. There were three of these: teen, parent, and clinician. As discussed previously this end user app would only be used by teens and parents, each of which would have different quizzes to fill out. At this stage, my PWA simply displayed a list of all quizzes no matter what user logged in. However, the functionality that I could see on the real site was that the parents and teens each had a 'portal', which had links to each of baseline, weekly, and end of trial surveys (I refer to these as 'categories from here on). These buttons led to a series of surveys: instead of a list, each survey would come one after the other (i.e., completing a survey would take you to the next one). This seemed like an annoying functionality; one would have to complete all the quizzes in one go. However, after bringing this up with Julia while developing this, she explained that it was the simplest way and that participants were told that they should allow time to complete all the surveys in one sitting.

I found that the account types were stored in a table called wp_usermeta, which contained extra information about user accounts. Embarrassingly, I could not initially find this information because PhPMyAdmin displays only 25 rows by default and I could not see anything resembling account types in these first 25 rows (search also seemed to only search these rows), nor in any other table. I thought that the account type functionality must have been added after Merlin sent me the new database, so I contacted him asking if this was the case. After a long correspondence in which he assured me it had already been done which I'm sure wasted a lot of his time, I thought to expand the number of rows past 25, and found it. Here is an example:

umeta_id	user_id	meta_key	meta_value
286	11	wp_capabilities	a:1:{s:4:"teen";b:1;}

Column 2 is a foreign key linking to the user ID from the actual user table. Thus, it was simple to retrieve the account type:

```php
function get_user_type($user_id){
  // get the account type (i.e. teen or parent)

  global $dbconnect;

  // this information stored in usermeta table
  $sql = "SELECT meta_value FROM wp_usermeta WHERE user_id='$user_id' AND meta_key='wp_capabilities'";
  $result = mysqli_query($dbconnect, $sql);
  // stored in serialized array, as users can have mutiple capabilities
  $user_capabilities = unserialize(mysqli_fetch_assoc($result));


  if(array_key_exists("teen", $user_capabilities)){
    return('teen');
  }
  else if(in_array('parent', $user_capabilities)){
    return('parent');
  }
  else{
    // this will occur if a clinician or other account type attempts to log in.
    // they will get to this point in the code as their details are valid, however
    // this end user app is not developed for these accounts, thus return false and disallow them loggin in
    return False;
  }
}
```

This was stored in a session variable. The advantage of these is that they store data beyond single page transitions like POST or GET, and that they have an advantage over cookies as users don't have to accept them (many people refuse cookies these days). They are also destroyed when the browser is closed, meaning that any data stored in them is safely removed when the browser closes. Thus, they were the best function to use.

Next, I created the portal page. This linked to my page showing the list of quizzes that I had used previously. However, I added a parameter in the GET array: the category of the surveys. Thus, using a combination of this and the user type I would be able to retrieve the suitable surveys. The surveys were conveniently named in the format: "[user type] – [category] – [survey name]". I considered carrying out a query that selected these names from my two parameters. However, in the interest of future proofing I did not in case these names changed.

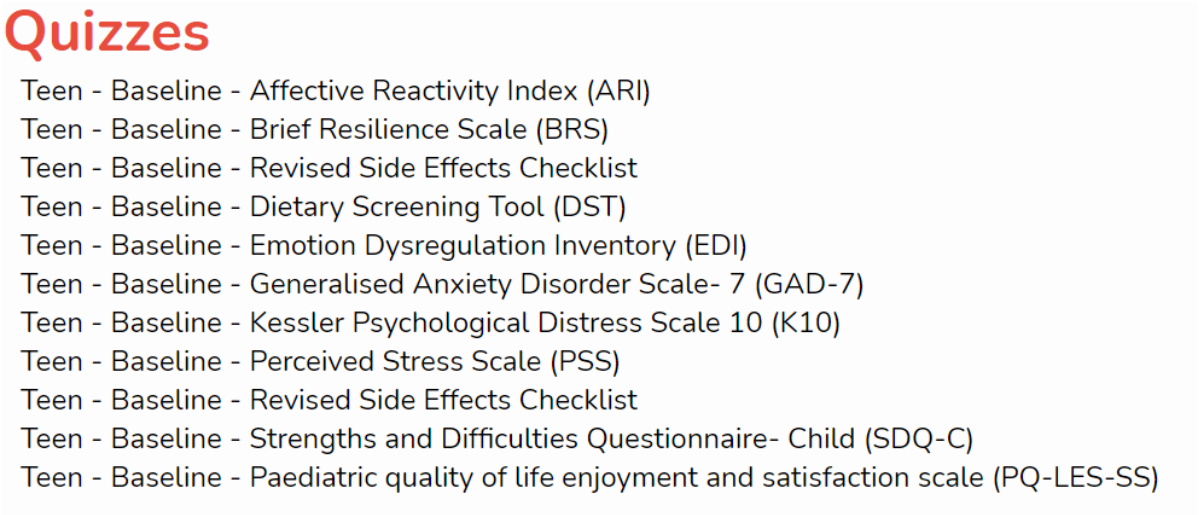
Instead, I manually made lists of all the surveys for each category within each user type. I was aware that this must be in the database somewhere and that this way I may need to update the list later. However, for the life of me I could not find where it was stored and updating the list later would not be strenuous. In order to get on with completing the PWA quickly so that I could get on with testing, I stored these in a series of constants, e.g.:

```php
define("TEEN_BASELINE_QUIZ_IDS", [5,8,13,15,21,23,26,27,29,48]);
```

I then wrote a snazzy piece of code taking my two parameters and turning it into this constant name

```php
$quizzes_to_select = strtoupper($_SESSION['user_type']).'_'.strtoupper($_GET['questions']).'_QUIZ_IDS';
```

The function constant() then used this string to refer to the constants, such as the one above. Thus, I could use it to select all the quizzes matching the quiz IDs in the constant:



Lastly, I needed to make these display one at a time. When manually creating the constants, I placed the quiz IDs in the order that the survey came up on the actual site. Thus, when a survey category was selected, instead of displaying a list of surveys such as above I stored the constant in the session array. The category button directed to the first survey in the category. From here, once a survey was completed and the record inserted, I would redirect to either the next survey, of if it was the last one, a page going home.

```php
// if survey last in category, show success message, else redirect to next quiz
if ($quiz_id==end($_SESSION['quizzes'])) {
  echo "All quizzes inserted successfully!";

  echo "<a href='index.php'>Go home</a>";
}
else{
  // go to next survey
  $next_quiz=$_SESSION['quizzes'][array_search($quiz_id, $_SESSION['quizzes'])+1];
  header("Location: index.php?page=quiz&quiz_id=$next_quiz");
}
```

Increasing Security

Security was a large issue that I had not been too concerned about during the prototyping of the app. However, seeing the types of questions in the real surveys served to reemphasize to me how essential it was to keep all data extremely secure. Thus, with this fresh in my mind and because I could not yet do the planned testing of the real quizzes, I decided to notch up the security of the PWA.

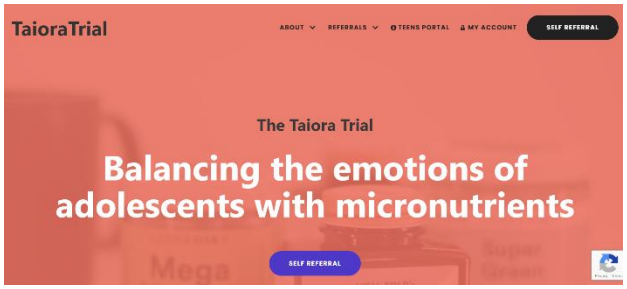
Data from surveys was not retained on the app and could not be retrieved once submitted. Users had to be logged in to even view the surveys, and the passwords were hashed. Thus, the only concern that I personally had was with database interaction (e.g. protecting against SQL injection). Until now, I had used the function mysqli_real_escape_string to protect against this. However, I knew that this was not as secure as using prepared statements. Thus, I changed all of my MySQL queries to use prepared statements. For example, my query to insert the data became:

```php
// prepare sql statement. EOD used to make readable
$sql = <<<EOD
INSERT INTO `wp_mlw_results` (`quiz_id`, `quiz_name`, `quiz_system`, `point_score`,
`correct_score`, `correct`, `total`, `name`, `business`, `email`, `phone`, `user`,
`user_ip`, `time_taken`, `time_taken_real`, `quiz_results`, `deleted`, `unique_id`, `form_type`)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
EOD;

if($stmt = mysqli_prepare($dbconnect, $sql)){

  // Bind variables to the prepared statement as parameters
  mysqli_stmt_bind_param($stmt, "isidiiissssissssisi", $quiz_id, $quiz_name, $quiz_system,
  $point_score, $correct_score, $correct, $total, $name, $business, $email, $phone, $user,
  $user_ip, $time_taken, $time_taken_real, $quiz_results, $deleted, $unique_id, $form_type);
  //  Set the parameters values and execute
  mysqli_stmt_execute($stmt);

  // I have error catching for this, but it is below other code
```

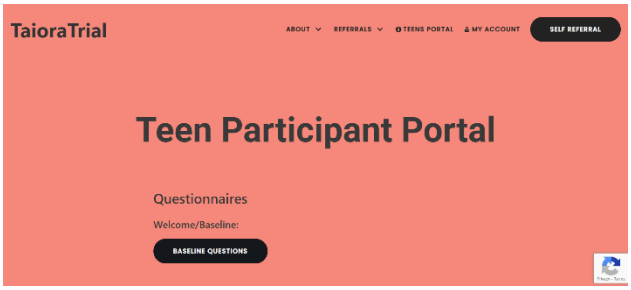
This would protect the database against attacks.

Design

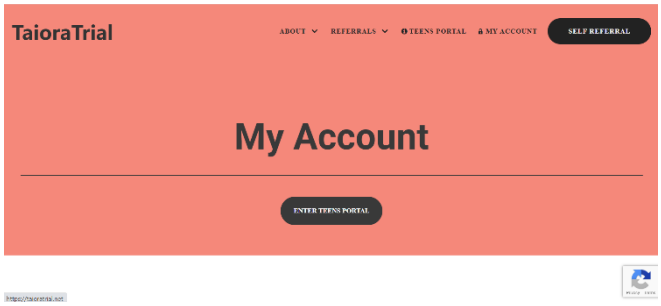
The release of the actual site meant that I could finally work a design for the app. One of the stakeholder requirements was that the app had a consistent look with the actual website. Therefore, I could attempt to replicate the feel for the app. Below are some screenshots from the site of the main functionality my app had as well.



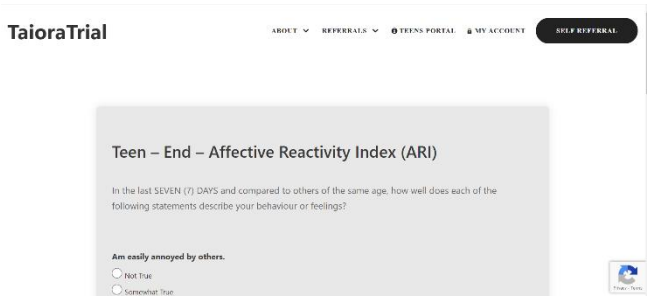
Home Page



Portal Page



Account Page



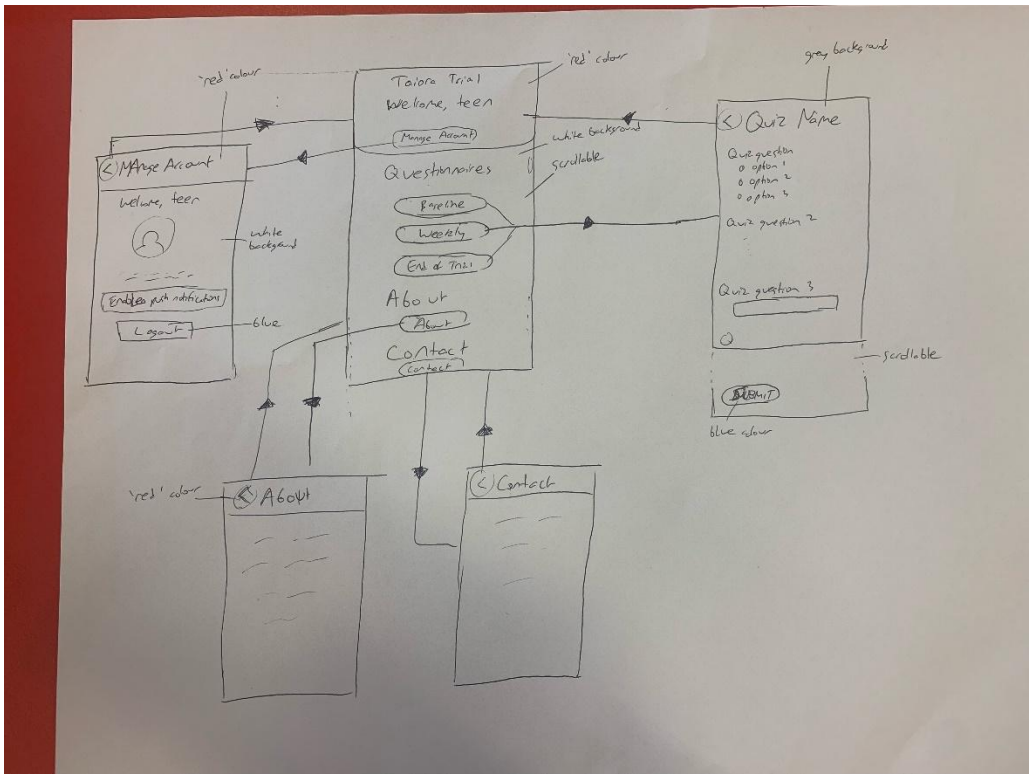
Survey Page

Most pages had that reddish colour scheme, whilst the quizzes had a more serious light grey look.

However, I wanted a simpler, more app-like experience for the PWA than the mess of different pages and links that I found the website had. I thought that I could get rid of the 'portal' and simply contain links to the different surveys on the home page. The home page could also have links to account management, about, and contact pages.

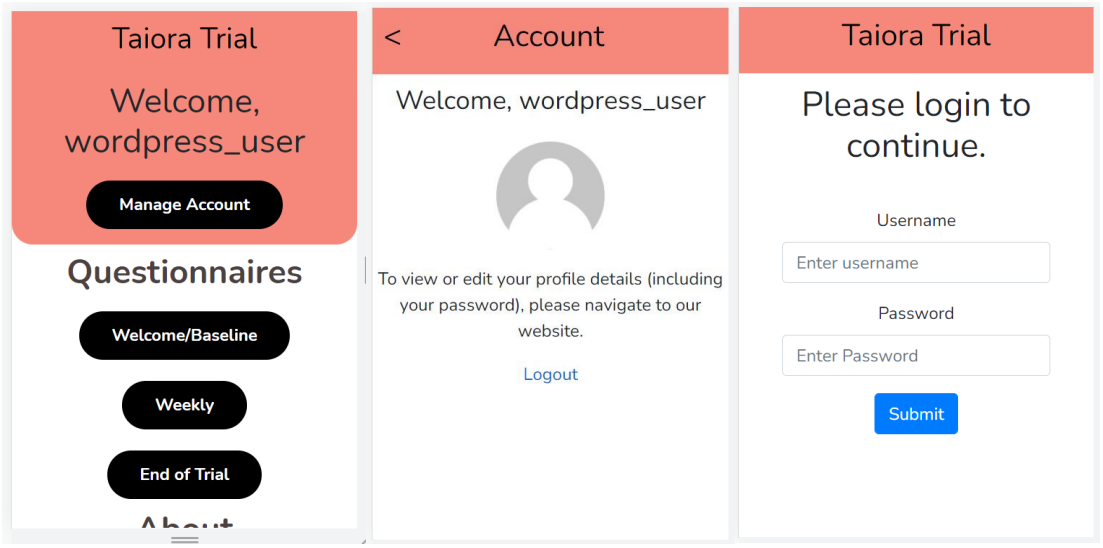
To keep the design consistent, I would use the same reddish colour as a background, and the same rounded-corner black buttons for the links (which would also work well on a touch screen as they are quite big). I saw no reason to change the design of the quizzes at all and would thus attempt to replicate these as close as possible.

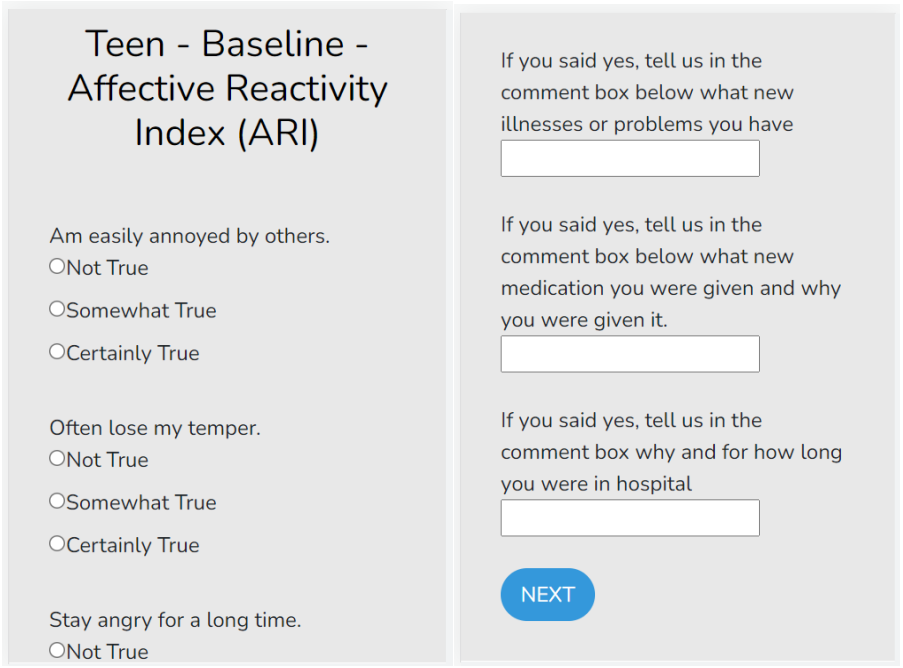
Because this design was so simplistic and much would be replicated from the real site, I did not need to use technology to create an in-depth wireframe. Instead, I did a quick sketch on paper so that I knew where I would be going with the CSS:



I showed this to Julia, in person, who said it looked great.

My proposed design was extremely basic which made the styling less complex. I used a combination of CSS and Bootstrap, both of which I had had experience in before. Because of the simplicity, I often found that using pure CSS was sufficient. However, I did use several Bootstrap classes, such as for the top 'navbar' for several pages (which contained multiple elements such as titles and back buttons: see the account page below) or the submit button in the login form. The benefit of Bootstrap is that it is 'mobile-first', meaning that my PWA will be responsive and look great on a variety of screen sizes – something vital for a PWA being used exclusively on phones and tablets. The most difficulty I had in this design was matching it to the real site. I used an eye dropper tool to get the exact colours and did trial and error until the black rounded buttons looked pretty much the same, and the same for the bright blue "next" buttons at the end of each quiz. However, overall, this process did not take long at all.





I had shown the app to Julia previously, however it had largely been updates to backend development and had been hard for her to appreciate what I had been doing. However, she was suitably impressed by this. When I asked her if the design was too simple, she disagreed: it allows “the teens to get on, do the quizzes, and get out”.

I would have also liked to get feedback on the design and usability by actual participants in the trial. However, at this point the trial was still months away from starting and no participants had been recruited. Instead, I settled with getting feedback from teens and parents that I knew personally. I recruited classmates, my parents and Mr Adams for this, and received the following feedback:

- I should have a button to show the password when logging in, as is conventional
- The spacing in the quiz multiple choice questions was slightly off
- Text answer boxes would look nicer with slightly round borders
- The page when completing a set of quizzes was not well styled

These were all minor problems, and it did not take long to implement changes. However, it is important to note that at this point, I was still developing locally and so testing was done on my computer rather than a mobile device as the PWA was intended for. I realized that it would be important to host the site live on the internet so that I could test on mobile.

Live Testing

Fortunately, Mr Adams had a site for just this purpose, and kindly uploaded the site and database for me which he received through GitHub (funnily enough, after having it on this site for a while, the enormous amount of serialized data in the QSM tables caused him to run out of his 'unlimited' database subscription – although a complaint to the hosting company resolved this).

With this, we found that the design worked just the same as we hoped. Only one problem was that the back buttons such as on the account page had too small a hitbox which should be increased.

Database Update

By this point, Merlin had got back to me with the updated database. While I did still find the odd missing question title, this fixed a lot of the problems that I had encountered with the old one – although the odd serialization did occur. However, I was still reluctant to embark on a robust testing until the final database appeared. This was largely due to this update including a new question type: multiple select (checkboxes). I thus had to go back to my quiz code and add this question type possibility.

Starting notifications

The PWA had most of the in-app functionality that it needed. However, notifications were another important aspect that I had not yet implemented. Thus, I set my sights on this task next. I initially thought that I could not be too hard – it was a hugely important framework that surely would not have been overcomplicated. I was sending web notifications (which could be received by a downloaded PWA), thus JavaScript was the language to use, and I jumped straight into some tutorials. However, not far in to this something became very apparent: I had never used JavaScript before.

```
// register service worker
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {

    navigator.serviceWorker.register('service-worker.js')
    .then(swReg => {
      console.log('Service Worker registered!', swReg);
    })
    .catch(err => {
      console.error('Service Worker Error', err);
    });
  });
}
```

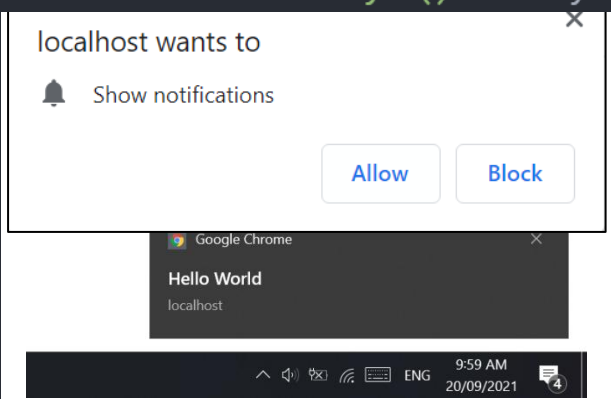
At the level of sending out notifications, documentation (not unfairly) assumed you had a capable understanding of JavaScript. However, I found even the most basic code unreadable (what was "getelementbyID"???). Therefore, I took some time taking some introductory JavaScript courses to at least get the basics down. I would have to learn more advanced concepts later, but I would deal with these as they came up. And so while still a bit shaky, I slowly built up the functionality of the notifications. I found it easier to start this in a completely new project, so that I didn't have all the distracting files from the working PWA.

There is a distinction to be made between regular notifications (which can be displayed outside the page at system level) and push notifications (which can do this even when the page is not open). I would need to utilize push notifications. There are two web APIs: the notification API and push API which make the process easier. I started by experimenting with the notification API.

First, one must check whether notifications are supported, and request permission to send notifications. This simple script below does of this, utilizing the Notification object. When combined with a button, I would be prompted to allow notifications (which only happens once), and then a notification would display on the system.

```
function notifyMe(notificationTitle='Hello World') {
  // check if the browser supports notifications
  if (!("Notification" in window)) {
    alert("This browser does not support desktop notification");
  }

  // check whether notification permissions have already been granted
  else if (Notification.permission === "granted") {
    // if so create a notification
    var notification = new Notification(notificationTitle);
  }

  // else ask user for permission
  else if (Notification.permission !== "denied") {
    Notification.requestPermission().then(function (permission) {
      // If user accepts, create a notification
      if (permission === "granted") {
        var notification = new Notification(notificationTitle);
      }
    });
  }
```

```
<button onclick="notifyMe()">Notify me</button>
```



I could now display notifications based on events in the site. Now, the next task was to get push notifications working. At this level of technicality, I was beginning to get quite lost. I found it helpful to refer to a google web developer training page was extremely helpful (https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications). However, there was one necessary requirement that I had already done:

Service Workers

Service workers were another new tool that I had to learn about and understand. I had built one at the very start of my project when testing out a PWA, though I had not understood it. I now learnt that service workers are scripts that run in the browser separate to the web page and can also run while the page is closed. This opens the door to many other functionalities such as PWAs and push notifications. To 'register' a service worker, the simple script on the right can be used, and the browser must be served over HTTPS. On the service-worker.js page, there were then scripts that would run in the background.

```
// register service worker
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {

    navigator.serviceWorker.register('service-worker.js')
    .then(swReg => {
      console.log('Service Worker registered!', swReg);
    })
    .catch(err => {
      console.error('Service Worker Error', err);
    });
  });
}
```

Push Notifications

Here is an overview of how I understand push notifications and the Push API to work:

When a user grants permission to receive push notifications, they are then 'subscribed' to the browsers push service (which manages push notifications). This creates a subscription object of the sort shown below:

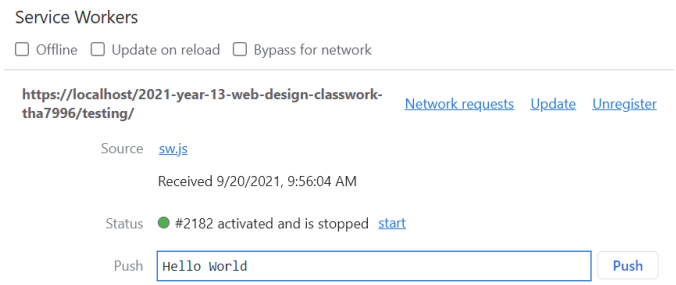
```
{"endpoint":"https://fcm.googleapis.com/fcm/send/dpH5lCsTSSM:APA91bHqjZxM0VImWWqDRN7U0a3AycjUf4O-byuxb_wJsKRaKvV_iKw56s16ekq6FUqoCF7k2
"keys": {
    "p256dh":"BLQELIDm-6b9Bl07YrEuXJ4BL_YBVQ0dvt9NQGGJxIQidJWHPNa9YrouvcQ9d7_MqzvGS9Alz60SZNCG3qfpk=",
    "auth":"4vQK-SvRAN5eo-8ASlrwA=="
    }
}
```

This contains an endpoint, which has a unique identifier that points to the subscribed user's service worker. The server can then attach a notification to this object and send it out, and the service worker will receive it. Additionally, a public key encrypts the object and the message (payload). More on this:

Allegedly, use of VAPID (Voluntary Application Server Identification for Web Push) keys is optional. However, they are required by Chrome so I would need to implement them. Without getting too technical, essentially a pair of public and private keys are used to encrypt all push notifications, adding security and authentication. My web app would have the public key stored and could thus decrypt push messages when it receives them, whereas messages would be encrypted with the private key when I sent them out from the server. Here's my public key:

```
const vapidPublicKey = 'BDD_zKtxaL_P25T2C7AjeenIND2VabW2qBn6tsHyib3-ICZLZ1ovxh5ID1Ilh-EvDw9Fz-sNiJcp37SpCqVKahQ';
```

To generate my public/private key pair, I used a Google tool called Firebase Cloud Messaging. This tool could apparently also be used to test out sending a push notification. However I could never get this to work. Instead, I would test notifications (on the client's page) using a tool I found in the Chrome Developer Tools, which simulates a push event (bottom right in the image).



In code, I largely adapted the functions from the documentation. However, one problem I did come across was that when creating a user subscription object, the public key is required in UintArray9 form. However, when I generated it, it came as a string. Thus, I had to convert the key into the right format before creating the object. I used a function found on the internet for this. This code creates the subscription:

```
swRegistration.pushManager.subscribe({
  // this is a 'promise' meaning browser will only accept push notifications with value
  userVisibleOnly: true,
  // the vapidPublicKey in Uint8Array (because thats what they need).
  applicationServerKey: urlB64ToUint8Array(vapidPublicKey)
```

This was in a function to subscribe the user, which also included additional tasks (such as doing something with this object). Referring to this, I also figured out how to unsubscribe the user, and a handy UI that had an enable/disable subscription button. I would have to adapt all these functions later. Additionally, in the service worker, were several functions to handle events: closing and clicking notifications, as well as receiving them.

With this relatively error-free process, I could now subscribe and unsubscribe to push, and send myself push notifications using that push tool.

Sending Push Notifications

For this, I would need to learn another new skill: Node JS. Using this would allow me to utilize the Web Push library, which helpfully handles many of the technicalities of sending push notifications. After learning how to set up a node environment, I ended up with this simple code that would hopefully send a (simple) push notification to the subscription object that I generated for myself earlier.

```
const webPush = require('web-push');

// VAPID keys
const vapidPublicKey = 'BDD_zKtxaL_P25T2C7AjeenIND2VabW2qBn6tsHyib3-ICZLZ1ovxh5ID1Ilh-EvDw9Fz
const vapidPrivateKey = [redacted]

const payload = 'Do your quizzes!!!';
const options = {
  // time to live. required header. number of seconds message stored if user not available
  TTL: 60,
  vapidDetails: {
    // in case an error occurs. push message not sent here.
    subject: 'mailto: tobyharvie@gmail.oom',
    publicKey: vapidPublicKey,
    privateKey: vapidPrivateKey
  }
};
const pushSubscription = {"endpoint":"https://fcm.googleapis.com/fcm/send/cFtLxrBo4rc:APA91bF

// webPush API function. Sends notfication
webPush.sendNotification(
  pushSubscription,
  payload,
  options
).catch(err => console.error(err));
```

After a few false starts, I was stunned when it worked. With XAMPP running, I could execute the code on the Node command line – in a completely different location to the code that would receive it – and a notification would pop up on my system. Not quite believing it, I had Mr Adams figure out how to work Node again so that he could run the same code on his computer. This caused a notification to pop up on mine. We tested it on several browsers and each one of them worked. Additionally, we found that if a browser was closed, the notification would not appear. However, if the app was installed (still using those old scripts) and the browser (and app) were closed, then it would. This was a major victory.

Storing Subscription Objects

I needed the functionality to store and update subscription objects. Users should be able to subscribe and unsubscribe from push notifications whenever they wanted. However, each time they (re)subscribed, a new subscription object would be created. Thus, I needed this new object inserted into the database when this happened and removed when they unsubscribed (Although I would later decide to simply set the object to NULL when unsubscribed instead of deleting it entirely).

The natural place to do this was in the wp_usermeta table which was perfect for custom functionality such as this. I could have created my own table but decided that this would just be more effort when it came to migrating this to the real site. As a reminder, the table is structured as on the right. By labelling each meta_key as push_subscription, I could then easily find all subscriptions when I needed to send a notification out. The meta_value would be the subscription object itself.

1	**umeta_id**
2	**user_id**
3	**meta_key**
4	**meta_value**

This planning seemed quite simple. However, I opened my code editor and soon realized a problem: I created subscription objects in JavaScript but would need to use PhP to interact with the database. A classmate suggested using AJAX for this, however I eventually found that I preferred a newer method: the Fetch API. This provides a cleaner interface to make many of the same XML requests.

```
// update subscription on server
function updateSubscriptionOnServer(subscription) {

  // FormData object mimics POST data. can thus send subscription object to PhP page
  const subscriptionData = new FormData();
  subscriptionData.append('subscription_object', JSON.stringify(subscription));

  // this page will update the subscription on server
  fetch('update-subscription.php', {
    method: 'POST',
    body: subscriptionData,
  }).then((response) => response.text()).then((text) => {
      console.log(text);
    })

}
```

The fetch function sends a request to the update-subscription.php page with the subscriptionData object in a very similar way to if a user were to submit a form using POST. However, this page is not actually loaded. The code is executed, and whatever text has been printed is sent back, which I log on the console for debugging purposes.

On update-subscription.php, I make sure the user is not accessing the page illegally and retrieve the subscription object using POST.

```
// verify sent here by main.js
if($_SERVER['REQUEST_METHOD'] === 'POST') {
```

```
$sub_object = $_POST['subscription_object'];
```

If the user is unsubscribing, this subscription object will be NULL whereas it will be the actual object if they are subscribing. I retrieve this from the database using the user_ID from the session variable. The rest of the code is simple logic to check whether such a record exists already for this particular user/what it is currently and update accordingly. I again used prepared statements for these queries to maintain security.

It worked great! The toggle of the enable/disable subscription button matched up with the records I was viewing in the database. However, after logging in and out after implementing this code I soon found a frustrating mistake: This fatal error refers to a line that attempts to retrieve data from the wp_usermeta table of the account type (e.g. parent/teen).

Fatal error: Uncaught TypeError: array_key_exists(): Argument #2 ($array) must be of type array, bool given in C:\xampp\htdocs\2021-year-13-web-design-classwork-tha7996\study\verify.php:64
Stack trace: #0 C:\xampp\htdocs\2021-year-13-web-design-classwork-tha7996\study\verify.php(111): get_user_type('1') #1 {main} thrown in **C:\xampp\htdocs\2021-year-13-web-design-classwork-tha7996\study\verify.php** on line **64**

After inspecting this table, I found that I had accidentally updated every single row in the table to the subscription object:

umeta_id	user_id ▲ 1	meta_key	meta_value
1	1	nickname	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
2	1	first_name	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
3	1	last_name	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
4	1	description	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
5	1	rich_editing	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
6	1	syntax_highlighting	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
7	1	comment_shortcuts	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
8	1	admin_color	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
9	1	use_ssl	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
10	1	show_admin_bar_front	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
11	1	locale	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
12	1	wp_capabilities	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
13	1	wp_user_level	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
14	1	dismissed_wp_pointers	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.
15	1	show_welcome_panel	{"endpoint":"https://fcm.googleapis.com/fcm/send/f.

I had forgotten to include the important meta key as a condition in the SQL query. With the changes below (the ? would be replaced in the prepared statement by push_notification) and some time-consuming work to reverse the above blunder, everything worked smoothly.

```
$update_sql = "UPDATE wp_usermeta SET meta_value=? WHERE user_id=?";
```

```
$update_sql = "UPDATE wp_usermeta SET meta_value=? WHERE user_id=? AND meta_key=?";
```

Sending Push Notifications from the Database

I could then adapt my Node code to function in a similar way so that it could send notifications to all users who were currently subscribed. I planned to create a fetch request that echoes a list of all subscription objects, and I send a notification one by one (there is no function to send these in a batch). The MySQL is:

```
$get_subscribed_users_sql = "SELECT meta_value FROM wp_usermeta WHERE meta_key='push_subscription' AND meta_value IS NOT NULL";
```

However, here the important part was receiving rather than sending data. I knew that the fetch request returned this data as a promise, thus had to parse it in …. . Additionally, I knew that the subscription object was JSON (which I had had previous experience with). Therefore, as well as reading this data as a JSON object, I also had to write it as one, which I found out by the errors I received when attempting to pass in the associated array standard to PhP/MySQL interaction. With multiple subscription objects in the server, I had to convert the array of these to JSON format before echoing it for JavaScript to pick up. Below is the fetch request that receives this data.

```
// get subscrbed user's subscriptions using php and mysql
fetch('http://localhost/2021-year-13-web-design-classwork-tha7996/admin-push/get-all-subscribed-users.php')
  .then((response) => response.json())
  // pass responses to function to send notifcations. data is in form of json array
  .then(data => sendEachNotifcation(data))
```

I could see (by logging to console) that this data looked like this:

```
[
  [
    '{"endpoint":"https://fcm.googleapis.com/fcm/send/e8sqXtpjMSU:APA91bEHxHbFNNuTJDCCtWFj77Hfxf3PJD6_RSGemhUSP4nNkKM
bUnRI_b-BB6-EFFsS3lacrH9Qe26fetrA0Gkdn1Onv7pVinNZCK4aIVMKk4ZUOgzAG9UyrazHTFjsYM6xxqjb3lEg","expirationTime":null,"key
s":{"p256dh":"BNMtqoev5BpjrlHEQDX-lDsBAXX-HfHXmlsAtBVNDIhR1jc3y_v1epYawEJB1ZHQLaWLn4XfAXn2cXkYPj-UEJY","auth":"mcVTqT
CgHX88-fQG53YLvg"}}'
  ],
  [
    '{"endpoint":"https://updates.push.services.mozilla.com/wpush/v2/gAAAAABg_LwUfDksHLXutJKzOaNTuwmiBhNiHvhACIc1nY6G
ASDi2v7M5YTCtZB2hSYCf1tXaJzx_LdDgSthqwj7hzE9CcqQrdUgHMLzFERtC0JzlzcQYDuSPJtXpT9YcR4pQeAVK8edzLcNNUJVq2Ksq3oRvJwq5F9Gv
SbpEGMOVYv_CLgS5Y0","keys":{"auth":"QDyZ08eMaAWsnrYaQ1og7A","p256dh":"BPYmDHq2CiaUpK-Cdf67P9AscYMekJnf_QVtoXLD-NN2H9v
Be4HIfrnY0mHSZXuz9wmNjozitIogcE3WkkTN0AA"}}'
  ]
]
```


After attempting to iterate through this and send a notification, the error below made me realize that these were strings rather than JSON objects.

```
(node:16360) UnhandledPromiseRejectionWarning: Error: You must pass in a subscription with at least an endpoint.
    at Object.WebPushLib.generateRequestDetails (C:\xampp\htdocs\2021-year-13-web-design-classwork-tha7996\admin-push\node_modules\w
eb-push\src\web-push-lib.js:87:13)
```

Parsing these to JSON resulted with this successfully working. I also wrote an additional script to send notifications to users who had not filled in any quizzes in the past week: this was done by checking the results table with all subscribed user's IDs and seeing if there was a record from the past week. The purpose of this would be to send a notification to users who had not yet filled in their quizzes that week.

PWA

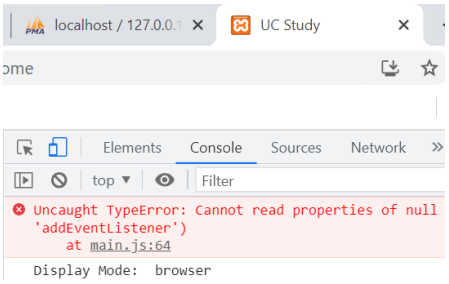
The last major chunk of development was making this web app installable. I had managed to do this at the start of the project, but now had to implement and tailor it for this new site. For a PWA to be installable, it must meet several criteria (which are dependent on the browser but are very similar): it must have a web app manifest, a registered service worker, be served over HTTPS and an icon to represent the app on the device. I had the service worker and the app would be served over HTTPS (fortunately, browsers treat local servers as HTTPS as well). However, I had to create the manifest file, which contains information and options such as the display name on the mobile device or the start URL. For now, I would use the same icon provided in the resources for the tutorial from the beginning.

If all of this is met, the script on the right (located in the service worker) installs the app. The array 'assets' contains a list of the site's assets so that the browser can cache them in the 'staticTaiora' cache so that they can be accessed offline. However, for this script to run the event must be fired from a browser UI. From my own experience I knew that this could be confusing thus talked with Julia, presenting her with some different options of this install process.

```
// install pwa
self.addEventListener("install", installEvent => {
  installEvent.waitUntil(
    caches.open(staticTaiora).then(cache => {
      cache.addAll(assets)
    })
  );
});
```

We both thought that it would be nice to have a custom install process (i.e. the user could install by pressing a button on the site itself). Additionally, considering that the web app would be hosted on the same site, if the user was able to use the app on the web, it would be confusing as there would be two places to fill in the quizzes. Thus, we thought it best that they user could only use the app *if they had downloaded it*. When accessed on the web, it would simply be a 'landing page' prompting them to install the app (or open it if already installed).

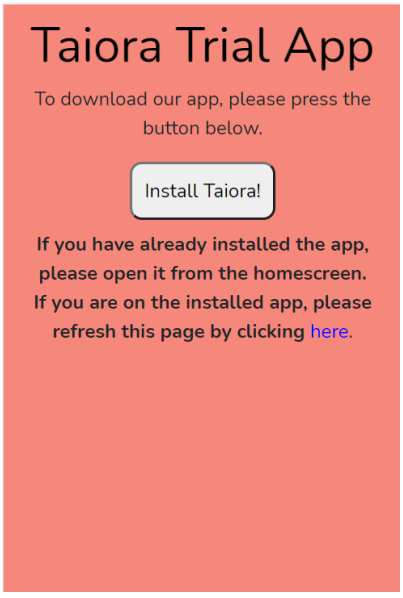
Thus, on each page load I needed to detect whether the user was in the app itself or the website. With an HTML 'onload' and a built-in JS function that checks for this, I could print this to console, as seen on the right (also note the browser's install button in the search bar (and also ignore the error – it got fixed)). But how would I relay this information back to PhP so that it could choose what to display? I made use of my new Fetch knowledge to set this information in a session variable, which could be accessed by whichever page the user was on.



Thus, if a page was displayed in a browser, I would instead display an install button. By stashing the default prompt as in the right, I could then display this when this button is clicked. If the user accepts, the app is installed. I listen for the event 'appinstalled', and reload the page so that the landing page disappears (the app recognizes that it is no longer on a browser).

```
// Initialize prompt for later use
let prompt;

window.addEventListener('beforeinstallprompt', (e) => {
  // prevent default install process
  e.preventDefault();
  // Stash the event so it can be triggered later.
  prompt = e;

  console.log(`'beforeinstallprompt' event was fired.`);
});


var installButton = document.querySelector('.install-button');

installButton.addEventListener('click', async function(){
  // make use of old prompt
  prompt.prompt();
  let result = await that.prompt.userChoice;
})
```

However, as much as I searched for a solution, there seemed to be no way to open the app if the user has already installed it and navigates to this page. Fortunately, this was not a large problem as if the app was installed, the user would not be likely to navigate to this page again. Just in case, I displayed a helpful message for these users.



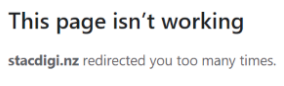
Additionally, I tested this across several browsers as usual. In some (notably Firefox), the page would not reload when the app was installed. I looked for a fix but could not seem to work it out. Given this – and the fact that I had yet to test on mobile devices – I thought it best to also include a refresh button. However, this did emphasize the importance to test the app on mobile ASAP…

Live Testing

By this point, I had pretty much all the functionality that the app needed, and my next priority was some robust testing. Again, this would be done by classmates and other people I knew rather than the actual participants as there were still few enrolments. I had planned to upload the site to a live server and let my digital technologies class loose on it. Mr Adams reuploaded my updated code to his testing site.

Unfortunately, however, this timing coincided with the COVID-19 Level 4 lockdown. But with little else to work on it would be great to get this done. Therefore, I ran several testing sessions over a Teams call with my digital technologies class (of about 16 people). This was far from ideal – it would have been nice to be able to be there to see what people were seeing – however, it did the job. During these sessions, I made sure that a range of devices and operating systems were being used (almost all mobile), and any problems that we ran across I also noted what device it occurred on. Additionally, I set up a large amount of testing accounts for both parents and teens and had everybody login with different accounts to simulate the real thing. However, this was hardly a smooth ride…

Teams Testing #1

The first testing session, nobody could access the app. This was not what I wanted at all. People were either greeted with the classic 

, or could not get past the login page – instead, users would be faced with a blank white screen. My own experience was the redirect error, however I noticed the following mystery: the URL that the site was attempting to send me to was:

```
stacdigi.nz/ucstudy/index.php?page=adminpanel
```

This was strange, because nowhere in my code was there a mention of adminpanel, at all. And yet, I would automatically be sent to this page.

It looked like the problem might not be with my code, but something to do with the hosting. To resolve this, Mr Adams and I set up a one-on-one Teams call a day or so later where we attempted to figure out what was going on…

Teams Debugging with Mr Adams

We started with some head scratching about this 'admin panel' mystery by browsing through the code. Mr Adams was screen sharing the actual admin panel of the hosting site (i.e. the file sdfsd). However, by doing this we noted something interesting. There was no home.php – one of the essential pages of the site. Despite Mr Adams being sure he had copied the page over, we thought that this must be it. He copied the page over again. And refreshed. And strangely enough it was gone again.

This was ridiculous. But naming the file something different had no effect, and so we began wondering if it was something in the code. By commenting out each line one by one, we found the problem:

```
<h2>Welcome, <?php echo $name ?></h2>
```

had to be

```
<?php echo "<h2>Welcome $name</h2>"; ?>
```

. Despite the first being perfectly valid, the hosting site did not accept it.

This allowed home.php to be created, which fixed the redirect error and would definitely have been the cause of the white screen as well. We could then see that the site worked this time and was ready for some real testing.

Teams Testing #2

This session ran a lot smoother and successfully unearthed some problems that would need attention. Before, most people could not access the install button at all. However, now that everybody could access it, there was a problem: it did not work. Well, specifically, it did not work for people with Apple phones. I cover the solution below, however, there were some other issues: first,

Live Site Debugging

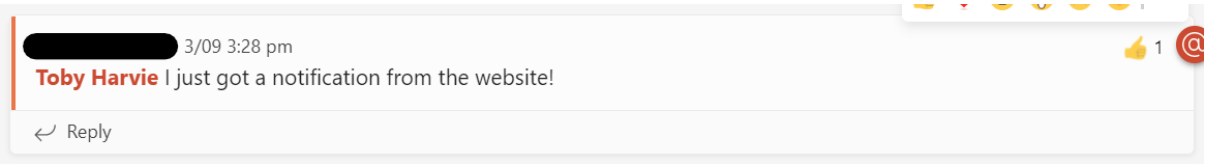
I knew the problem was not with the database connection as the subscription objects were inserting. By debugging the insert query for the survey results, I got this error message:

```
object(mysqli_stmt)#2 (10) { ["affected_rows"]=> int(-1) ["insert_id"]=> int(0) ["num_rows"]=> int(0) ["param_count"]=> int(19) ["field_count"]=> int(0) ["errno"]=> int(1292)
["error"]=> string(124) "Incorrect datetime value: '2021-08-28 09:08:34 PM' for column `stacd342_ucstudy`.`wp_mlw_results`.`time_taken_real` at row 1" ["error_list"]=> array(1) { [0]=>
array(3) { ["errno"]=> int(1292) ["sqlstate"]=> string(5) "22007" ["error"]=> string(124) "Incorrect datetime value: '2021-08-28 09:08:34 PM' for column
`stacd342_ucstudy`.`wp_mlw_results`.`time_taken_real` at row 1" } } ["sqlstate"]=> string(5) "22007" ["id"]=> int(1) } success
```

I had an 'incorrect datetime value' for the column 'time_taken_real'. By inspecting the format for the datetime column it was in the format Y-M-D H:M:S. I was inserting (and this worked for my database)

the same, but with an additional AM/PM. By removing this, the results inserted. I do not know why the results were different

Additionally, I had tried sending a notification to some users who subscribed during this session. This had not been received, and by inspecting this I realized it was because I had commented out some code during debugging, and left this in. I fixed this. A little later, this came through on the class Teams chat:

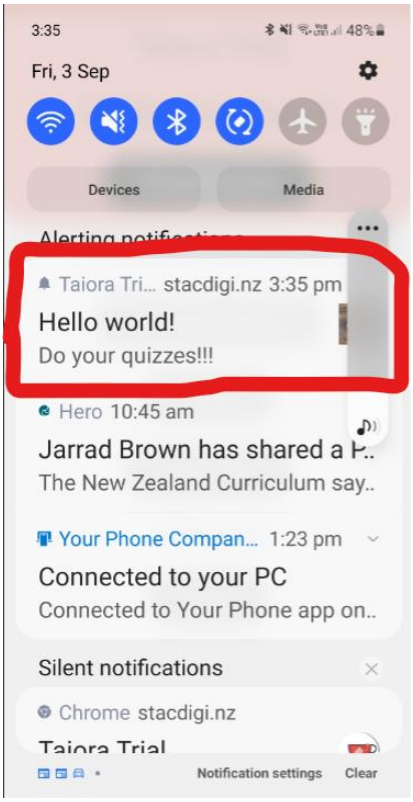


iOS Install Process

It turned out that iOS does not support the custom install process. Instead, users had to use the default install process – and additionally, they had to be in Safari which would have been nice to know earlier. To fix this, I needed to detect whether a user was on iOS. If so, I would instead display instructions for them on how to install the app rather than a button. I used the same function that checked whether they were on browser or app and used an iOS-detection script from the internet. Thus, I could pass it into the Fetch function as well and set the operating system as a session variable. If the mode was a browser, I would display different screens based on which platform the user was on. By uploading this to the live site, I could ensure that it worked. Below the Android (left) and Apple (right) phones are on the same web page, but have different landing pages.

Teams Testing #3

With much of the other sessions spent figuring out various problems, in this one I was especially interested in the notifications. I made sure that for everybody that subscribed, I sent them a notification. Every single one came through. To the right is a screenshot of one.



Other General Feedback

During these testing sessions, I also received some helpful feedback of how to improve the usability of the site, especially specific to the small screen size. These were all fixed without too much trouble:

- On mobile, long question answers would spill onto a line separate to their checkbox, e.g.:
- The toggle notifications on/off still appeared for Apple phones (with the text 'push not supported'). It was pointed out that this was pointless and only served to distract the user, thus should be removed on Apple phones.
- The second step of the installation instructions for iOS was not clear.

Which of the following does your child usually do? (Please indicate ALL that apply.)

☐ Expresses anger in an appropriate way (e.g., explains her/his perspective; goes to her/his room to cool down)

☐ Argues, whines or sulks

☐ Becomes verbally insulting, swears, shouts

☐ Threatens

☐ Slams doors, punches walls, makes a mess, destroys property

Migrating to the Real Site

It was time to start thinking about how I would transfer my app to the same hosting platform as the real site. For this, I would need Justin's help. Hoping that he would reply, I sent him an email re-explaining my project and how it would link to the existing WordPress database. Fortunately, he sent me this back:

HI Toby,

I am no app expert and a middling developer at best there are a few rules I have worked by over the years.

One of those is where an API is offered then one should use it. There is usually a good reason that the API exists and probably good reasons for not interacting with the database directly. The only exception to this rule would be if you are only reading the database, and not writing / updating information.

The issue with writing directly is that you go outside of any data input validation and processes the developer, in this case WordPress, have built into their application.

As far a creating the subdomain (for your app I suggest api.taioratrial.net perhaps) that is no issue and setting that up easy enough.

Do you have a public key you can send me and I'll organise SSH access to where the files will need to go? You may want to get a bit of a backgrounder on the hosting platform while I attend to this.
https://kb.sitehost.nz/cloud-containers

Feel free to ask any further questions you may have.

Regards

Justin C. Le Grice

Justin refers to the WordPress REST API that allows external connection to the database. I had mentioned that I had chosen not to use it in my email to him feeling it unnecessary, however this reply was gutting. I knew that adding this API functionality would be extremely tiresome. I thus sent a reply attempting to convince him that this was not needed (and clearing up some other things):

Hi Justin

I'm not sure if you completely understood what I envisaged for this app. It is a progressive web app, meaning that it is completely web-based. I thus imagined simply putting the PWA's files in the same directory as the WordPress site that you are hosting, so that it is hosted in the same place and meaning that I would not need my own public key as it would share the same one as the WordPress site.

From what I understood, this meant that I could connect directly to the database. I made sure to follow suitable security measures such as using prepared statements to insert data, and the data matches the required input type. The only data that I am inserting is into the quiz and survey master results table, and one row per user into the wp_usermeta table. I am in the middle of extensive testing to ensure that all records insert correctly. There have so far been no issues with this.

I understand that if you still need me to use the API. However, I thought that I would make sure by clearing up how my app connects to the database.

Thanks
Toby

Hi Toby,

As long as you are confident that the direct database accesses are not going to break things then lets proceed. The containers are backed up by the hosting company daily, and I have my own separate file based back up (it's WIP, I have to add the database backups to this) that runs 6 hourly. This means we can fairly quickly recover from any mishaps. I may increase the frequency of these over the next few weeks.

Are you able to fire me your public SSH key so I can get you sorted for SSH access to the container?

Regards

Justin C. Le Grice

I still did not know what Justin meant by these SSH keys. I asked Mr Adams, who did not know either. I asked for some further detail and received this reply:

Hi Toby,

So many hosting providers are moving away form FTP to SSH/SFTP (FTP over SSH) access to the site files. Standard FTP is unencrypted transmission of files and is now thought to be an insecure means of transmission.

SSH is an encrypted protocol and considered the standard means of remote communications and file transfer.

It has the added advantage of being able to use certificates for authentication rather than passwords. Have a read of this to fill you in on how these are used. The example given assumes one is using a Linux machine.

If you are using Windows you will need to use the like of PuTTY and PuTTYKeygen to create the key / certificate pair.

Regards

Justin C. Le Grice

I downloaded the software that Justin mentioned and found it quite straightforward to create the key/certificate pair and have sent these to Justin. Therefore, at the due date of this report, the files are about ready to transfer to the live site.

Next Steps

Once my PWA has been connected to the real database, it will be important to confirm that this connection runs correctly: it will need to be completely error free (there is no reason that it would not be), but also the inserted results should show up on the admin page (if you remember, these appeared as some random string of characters on my test WordPress site; I suspected that they would appear as normal with the real database but was not sure). Additionally, I will need to further collaborate with Justin and Merlin to find out how to run my Node script (which sends the push notifications) so that it connects to this existing database. Once this is done, we will likely have a slow rollout to ensure that there are no problems, so that we can ensure that frustration will decrease rather than increase.

Summary & Reflection

Despite having a bit more work ahead of me, I am extremely pleased with what I have managed to get done. Faced with obstacles such as unresponsive stakeholders, having to learn entirely new techniques, or just the absolute stupidity of some of the tools I was forced to use, I have managed to create a functional PWA that ticks all the boxes of functionality that we had planned. I have gained so much knowledge and experience not only around digital technologies, but with and working with stakeholders. Once the PWA is fully deployed, I will know that I have assisted in essential research for mental health.

There is no doubt that I will need to keep working on the PWA throughout the rest of the trial. If the app succeeds in boosting engagement, Julia has indicated her desire to continue using similar technologies in future studies. It does not seem that the end of this report signifies the end of my involvement with this topic.

Note to the Marker

In the included USB drive, please find a video titled ‘App’, which shows functionality of the app, and another titled ‘Notification’, which shows a notification being sent from my computer to a user.

Bibliography

This is not a complete list, with many quick fixes not being recorded. The most important sources have been referenced in the report.

https://wordpress.org/support/article/how-to-install-wordpress/
https://wpmudev.com/blog/setting-up-xampp/
https://stackoverflow.com/questions/9854480/using-wordpress-user-password-outside-wordpress-itself
https://davidcoveney.com/575/php-serialization-fix-for-wordpress-migrations/
https://www.php.net/manual/en/function.serialize.php
https://www.geeksforgeeks.org/how-to-use-php-serialize-and-unserialize-function/
https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications
https://superpwa.com/?utm_source=wordpress.org&utm_medium=description-demo
https://www.w3schools.com/tags/att_input_type_checkbox.asp#:~:text=The%20%3Cinput%20type%3D%22checkbox,tag%20for%20best%20accessibility%20practices!
https://www.w3schools.com/php/php_mysql_prepared_statements.asp
https://www.w3schools.com/js/js_output.asp
https://www.wpbeginner.com/wp-tutorials/how-to-add-web-push-notification-to-your-wordpress-site/
https://wordpress.org/support/topic/multiple-choice-answers-were-easily-found-from-source-code/
https://www.codegrepper.com/code-examples/php/how+to+fetch+php+array+in+javascript
https://stackoverflow.com/questions/60084428/failed-to-fetch-data-from-localhost
https://stackoverflow.com/questions/1078118/how-do-i-iterate-over-a-json-structure
https://www.w3schools.com/js/js_json_parse.asp
https://firebase.google.com/docs/cloud-messaging
https://stackoverflow.com/questions/55643149/how-to-store-fetch-api-json-response-in-a-javascript-object
https://stackoverflow.com/questions/45018338/javascript-fetch-api-how-to-save-output-to-variable-as-an-object-not-the-prom
https://stackoverflow.com/questions/6880749/php-ini-production-vs-development/6880824
https://stackoverflow.com/questions/56860675/refresh-page-every-time-pwa-is-opened-pwa
https://stackoverflow.com/questions/66009755/how-to-detect-if-pwa-is-already-installed-on-ios/66012300
https://www.w3schools.com/js/js_json_php.asp
https://www.w3schools.com/js/js_output.asp
https://stackoverflow.com/questions/50929198/open-installed-pwa-from-external-url
https://stackoverflow.com/questions/45605690/migrating-wordpress-site-serialized-data
https://stackoverflow.com/questions/3020751/can-javascript-connect-with-mysql
https://stackoverflow.com/questions/45018338/javascript-fetch-api-how-to-save-output-to-variable-as-an-object-not-the-prom
https://www.puttygen.com/