Cross-Platform Development

Overview and QR

Middle Earth Journeys is an exercise companion app that aims to encourage users to get out and partake in physical activity. It provides a more engaging way for less motivated individuals to get fit than conventional running apps by allowing users to follow in the footsteps of J. R. R. Tolkien's Middle Earth heroes. Users can select from two different quest options and explore the land of Middle Earth whilst walking, running, or cycling. The application tracks user movement giving performance metrics such as distance covered, average pace and time spent. Then, depending on the difficulty the user selected, it calculates how far through the quest and whereabouts in Middle Earth they are. The quests are long and therefore are designed to be tackled in 'sections', each time the user resumes a quest, a section is created. When they pause the quest the section results (distance covered, time etc) are added to the total quest statistics. If they have travelled the required distance in middle earth, they will discover one of a plethora of landmarks. These landmarks are accompanied by videos, images and descriptive text which informs users about the location.

The only real feature that has changed since the initial design is the real-world map. The map is accessible and the user can still see their route. However, now it is only viewable *after* they have completed a section not *during*. The original idea of having a live map that could show the users route in real-time proved to be too costly to implement.

App Showcase

Video

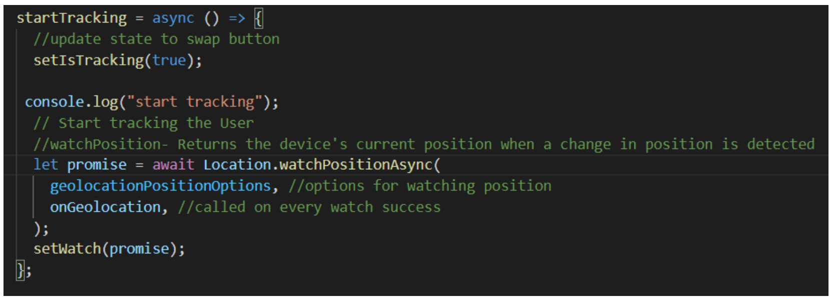
Video link - https://youtu.be/xYiqavLFaRQ

The working version of the application can be seen in the video above. In this video, you can see all of the core features working exactly as intended. However, there are a few irregularities in the video which do need explaining. When showcasing the example of an active quest the total quest time in the performance tab is showing an incorrect time (not matching with the combined section times). This is due to manually deleting sections from the section fire store and then not also updating the quest documents total time. The reason for deleting sections from an existing quest and not simply showing a new quest was to quickly showcase the landmark functionality, the first landmark is found when a user travels 20km (2km real world on human difficulty). To prevent me from having to walk this distance, whenever the pause button is clicked 1k is added to the distance. Sections were removed from the quest to get the distance down to 10km meaning only one section needed to be added to find the landmark. If a quest were started from scratch, it would be clear that the time does correctly update and follow the section time.

The other ‘issue’ shown in the video is the differences between the stopwatch time and the section-time the user is given. This is due to the stopwatch tracking time from when the start and stop buttons are clicked whereas the section-time utilises GPS timestamps and only tracks movement time. It may be beneficial in future iterations of the app to utilise the stopwatch time as the main recorded time for easier usability. Users would be much clearer on their times if it were developed this way.

App Elements

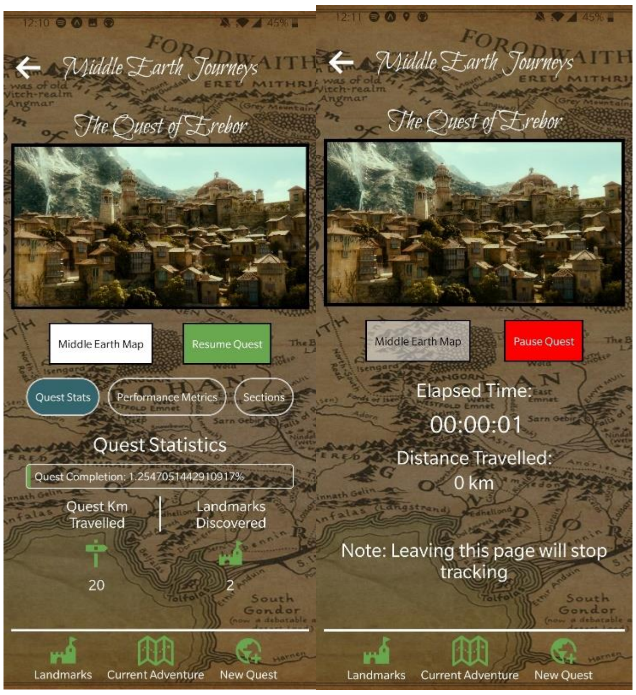
The first element to be discussed will be the main geolocation aspect of the application. Geolocation was implemented through the Expo location package. As seen below when the start button is clicked a function is called which sets a tracking variable to true and utilises Expo locations ‘watchPositionAsync()’ function. This function continuously watches the users position and calls the ‘onGeolocation’ function whenever there is a location update. ‘onGeolocation’ captures the users current coordinates and adds them to a tracking array object which is used later on to map the route. Options are also passed into the watch function which specifies when location updates are checked, for this app the location is checked whenever the user moves one meter. This is to provide as high an accuracy as possible when mapping the route. The watch function returns a promise which is used when the user clicks the stop button to halt geolocation tracking, ensuring updates do not continue when the user is done.



```
startTracking = async () => {
  //update state to swap button
  setIsTracking(true);

 console.log("start tracking");
  // Start tracking the User
  //watchPosition- Returns the device's current position when a change in position is detected
  let promise = await Location.watchPositionAsync(
    geolocationPositionOptions, //options for watching position
    onGeolocation, //called on every watch success
  );
  setWatch(promise);
};
```

By setting a tracking variable the elements on the screen can be adapted to reflect that we are now tracking. The two images below show what the app looks like when we start and stop a quest.

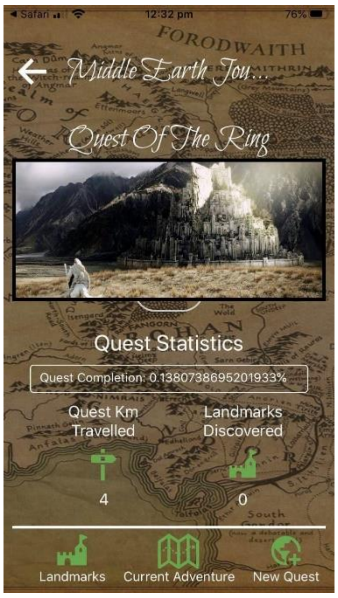


The interface for the quest is split into three separate tabs which each show a different aspect of quest information. By splitting the quest details into separate tabs we can provide the user with all the data they need without overwhelming them. The user can pick exactly which information they want on-screen at a given time. As Nayebi et al., (2012) discuss; simplicity, size and format of text must be considered to provide good usability. If the tabs were not used there would be far too much text on the screen which would make it potentially unreadable.

When a quest is resumed these tabs make way for a stopwatch and distance counter that allows users to focus on the exercise itself and removes any distracting elements. They are only shown important information in a clear readable manner. The start and stop buttons are brightly coloured and showcase a call to action. User's eyes are drawn to these elements as they are the most standout features on the page.

The implementation of the geolocation feature demonstrates at least three of the 'Five E's' of usability - efficient, effective, engaging, error-tolerant and easy to learn (Quesenbery, 2011). The feature is effective in working out the user's location and engaging with the bright colours, images, and icon use. It is also easy to learn, all users have to do is click start/stop, all partitions of the page are signposted.

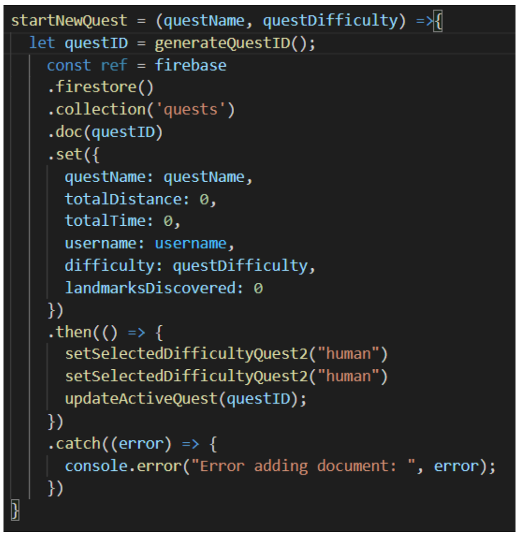
This element was by far the most challenging feature to develop for the application. This was mainly due to issues with the built-in geolocation API which would not work as intended. Instead of calling the 'onGeolocation' function every time the user moved a metre, it was only called once when the start button was pressed. Research found that this was a common issue and unfortunately there was no clear solution. However, luckily Expo had its own location package that did work correctly. Time was spent switching to Expo's location functions (for example they were now asynchronous and accepted different arguments) but once it was complete user location could be tracked successfully. Stylistically there were also problems getting the page content to fit onto multiple screen sizes. A decision was made to wrap the tabs inside a 'ScrollView' element so users could scroll down the page to access more content. This prevented the need to make any items smaller on the page which would have sacrificed usability hugely as hard of sight users would have struggled to read parts of the text. An example of what this page looks like on smaller screens can be seen below – the image shows the page after it has been scrolled down with the tabs being just visible at the top of the page.



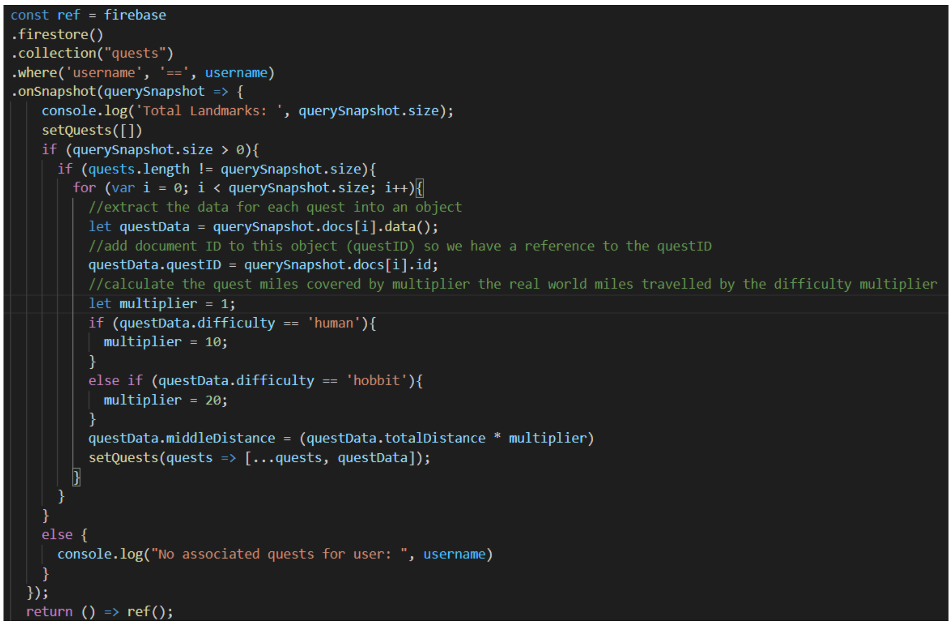
In general terms, the application does work well across multiple screen sizes. 'ScrollViews' are utilised across various pages to make sure elements are accessible regardless of the screen dimensions. The core features are usable on multiple screens but there are a few issues that, given more time (and more prolonged access to different devices), could have been resolved. For example, as seen in the previous image, the title is cut off on smaller devices, an issue that can be fixed by positioning the title more centrally and wrapping it inside a padded view.

As demonstrated by both the video and screenshots already presented, the geolocation correctly calculates, tracks, and plot the user's route. However, when the app was published for this report, it appeared that the geolocation no longer worked as intended. The geolocation was still activated as the user's route was adequately plotted, but the distance they travelled was now returning as null ('NaN'). The development version of the app was debugged but surprisingly it worked correctly and therefore it proved extremely difficult to find why the published application was failing. Errors on just the published version were not planned for and as such as the app was only published with a few days to go. This meant that unfortunately there was not enough time to fix the issue. Hopefully, the provided resources highlight enough evidence to show that the geolocation was working and the development version can be launched to consolidate this fact if more proof is needed.

The second element that will be discussed is the utilisation of firebase storage across the application. Firebase has been used to store user accounts and manage authentication as well as store quests with their associated sections and discovered landmarks. The snippet below shows the code for adding a new quest to firebase. A random questID is generated to use as the document ID for the quest. Manually producing a random document ID makes sure the quest is uniquely identifiable and that it can be referenced in other documents. For example, the ID is added to the user's account as a reference to their current active quest (the 'updateActiveQuest()' function seen in the image). This means that when 'current adventure' is clicked in the tab bar, they are taken to the quest with this corresponding ID.

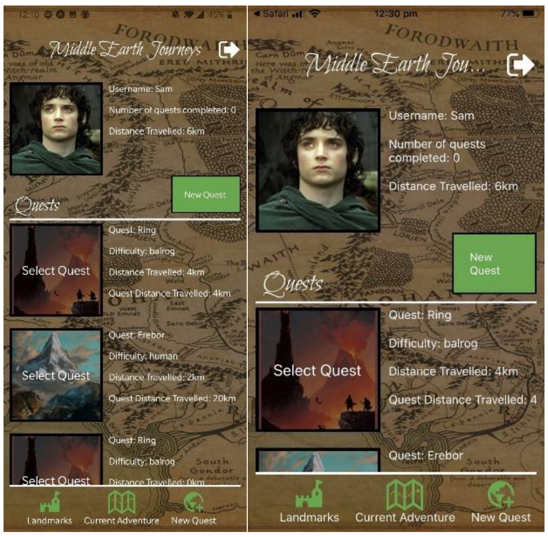

```
startNewQuest = (questName, questDifficulty) =>{
  let questID = generateQuestID();
    const ref = firebase
    .firestore()
    .collection('quests')
    .doc(questID)
    .set({
      questName: questName,
      totalDistance: 0,
      totalTime: 0,
      username: username,
      difficulty: questDifficulty,
      landmarksDiscovered: 0
    })
    .then(() => {
      setSelectedDifficultyQuest2("human")
      setSelectedDifficultyQuest2("human")
      updateActiveQuest(questID);
    })
    .catch((error) => {
      console.error("Error adding document: ", error);
    })
}
```

Firebases 'onSnapshot()' function is utilised to listen to any changes in the quests FireStore. When a new quest is added, all quests which have a username tag equal to the current user are fetched and displayed on the page. This was quite a tricky process to manage as the quests needed to be pushed into a state array which was then mapped over to render the current quests. JavaScript does not allow items to be pushed to an existing state array so a workaround was implemented using the spread operator. This effectively added the quests to a new array and overwrote the state array with this array. A conditional statement was implemented to check that the number of fetched quests differed from the current number of quests. This, combined with setting the state array to null before adding new values, prevented already rendered quests from being re-rendered. The full code for this can be seen below.



```
const ref = firebase
.firestore()
.collection("quests")
.where('username', '==', username)
.onSnapshot(querySnapshot => {
    console.log('Total Landmarks: ', querySnapshot.size);
    setQuests([])
    if (querySnapshot.size > 0){
      if (quests.length != querySnapshot.size){
        for (var i = 0; i < querySnapshot.size; i++){
          //extract the data for each quest into an object
          let questData = querySnapshot.docs[i].data();
          //add document ID to this object (questID) so we have a reference to the questID
          questData.questID = querySnapshot.docs[i].id;
          //calculate the quest miles covered by multiplier the real world miles travelled by the difficulty multiplier
          let multiplier = 1;
          if (questData.difficulty == 'human'){
            multiplier = 10;
          }
          else if (questData.difficulty == 'hobbit'){
            multiplier = 20;
          }
          questData.middleDistance = (questData.totalDistance * multiplier)
          setQuests(quests => [...quests, questData]);
        }
      }
    }
    else {
      console.log("No associated quests for user: ", username)
    }
  });
  return () => ref();
```

Using a map function to render the quests ensured there was a uniform, tabular look to the account page. This made it clear to distinguish each quest and what their related statistics were. Again the ScrollView element was utilised to fit all the information on the page regardless of screen size (the two below screenshots show the page on different devices). As Hoehle and Venkatesh (2015) state, “presenting a large body of content on mobile devices is problematic because the application interface is overloaded with information”. The layout of this page tries to counter that by displaying only a limited amount of information associated with each quest and each element is adequately spaced out from one another. White borders are also used to divide the quest section and the tab navigation at the bottom of the page. As well as this, only a maximum of three quests are viewable on the page at any one time. All of these usability considerations reduce interface overload and keep the application in line with the ‘5 E’s’ that were previously discussed.



Username: Sam
Number of quests completed: 0
Distance Travelled: 6km
Quest: Ring
Difficulty: balrog
Distance Travelled: 4km
Quest Distance Travelled: 4km
Quest: Erebor
Difficulty: human
Distance Travelled: 2km
Quest Distance Travelled: 20km
Quest: Ring
Difficulty: balrog
Distance Travelled: 0km

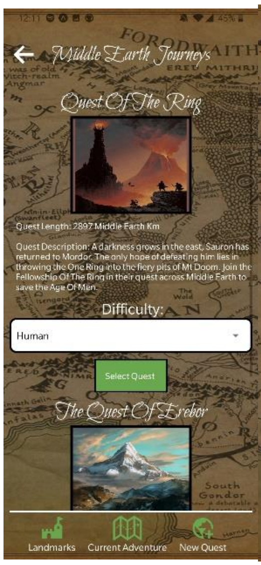
Username: Sam
Number of quests completed: 0
Distance Travelled: 6km
Quest: Ring
Difficulty: balrog
Distance Travelled: 4km
Quest Distance Travelled: 4
Quest: Erebor

It is worth noting that due to time constraints the middle earth map is not fully functioning. The intention was to create a serious of images with pin drops on them to mark where the user was and what locations they had found. Depending on how far through the quest they were would depend on which image they would be shown. Unfortunately, I ran out of time to implement this feature and only one of the images was created. Luckily, this image provides a good reflection of the intended end goal.

<u>Reflection</u>

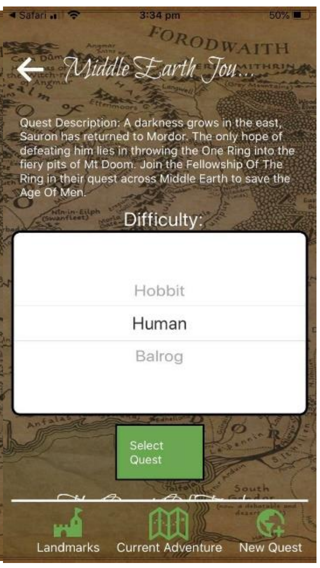
When utilising cross-platform development (CPD) the speed of development is relatively quick. The app can be written once using a language that supports multiple platforms meaning separate versions of the app do not need to be created for different devices (Nitecki, 2019). This was most apparent during the publishing of Middle Earth Journeys; Expo provides different builds of your published app to release on a multitude of app stores. This ties into arguably the biggest advantage of CPD, portability. Applications can be used across a wide range of devices and due to the small size of mobiles the app can be utilised on the go. When testing the application's geolocation the phone could be taken outside to properly test its capabilities. With CPD you can utilise the smaller form factor of phones to do a variety of tasks that previously would have been challenging on a web application. It is in these conditions that the CPD framework thrives.

However, CPD does have some drawbacks. Expo does create both IOS and android versions of applications but this does not guarantee that every element in these apps works as intended. Not all elements easily translate across platforms, so time has to be spent finding/configuring elements that do work as intended on all devices. This lack of translation is partly due to developing a native application which is an app that is based on the target platform language. These apps offer good speed and can take advantage of device hardware but this comes at the price of making them hard to develop as fully cross-platform due to them being so closely related to a specific architecture (charkaoui et al., 2014). This is most prominently seen in the difficulty picker that was integrated into the 'New Quest' page. The picker looked perfect on android but on IOS the formatting was completely ruined due to it utilising the operating systems specific picker. The differences between the two platforms can be seen in the image below. It must be noted that the core chunk of code is still usable across platforms meaning that most of the work in getting an app to function across multiple devices is already done for you.



Quest Length: 2897 Middle Earth Km

Quest Description: A darkness grows in the east, Sauron has returned to Mordor. The only hope of defeating him lies in throwing the One Ring into the fiery pits of Mt Doom. Join the Fellowship Of The Ring in their quest across Middle Earth to save the Age Of Men.



Quest Description: A darkness grows in the east, Sauron has returned to Mordor. The only hope of defeating him lies in throwing the One Ring into the fiery pits of Mt Doom. Join the Fellowship Of The Ring in their quest across Middle Earth to save the Age Of Men.

As well as the difference in elements, cross-platform apps have to be well optimised. Limited performance power on mobiles leads to a situation where you have to consider the efficiency of code much more than in a desktop application. CPD cannot be used to develop applications that are extremely processor intensive. This became very clear when the YouTube Iframe was integrated into the landmark page. Although it works it does take a fair amount of time to load the video.

In conclusion, if you want to develop an application for multiple platforms relatively quickly that utilises specific mobile features, such as portability or GPS, and you want to reach as wide an audience as possible, CPD is a perfect approach. The framework can take advantage of platform hardware to perform tasks that previously would not have been feasible with desktop apps (e.g. tracking a run). Where the approach should not be utilised is in more process-intensive tasks, such as performing data analysis, or with apps that require certain pieces of platform hardware to operate. These tasks are more suited to a larger dedicated device.

References

Charkaoui, S., Adraoui, Z. Benlahmar, E.H. (2014) Cross-platform mobile development approaches. In: *2014 Third IEEE International Colloquium in Information Science and Technology (CIST)*, Tetouan, Morocco, 20-22 October. IEEE, 188-191. Available from https://ieeexplore.ieee.org/document/7016616 [accessed 18 May 2021].

Hoehle, H., Venkatesh, V. (2015) Mobile Application Usability: Conceptualization and Instrument Development. *MIS Quarterly,* 39(2) 435-472. Available from https://www.jstor.org/stable/26628361?seq=1#metadata_info_tab_contents [accessed 18 May 2021].

Nayebi, F., Desharnais, J.M, Abran, A. (2012) The state of the art mobile application usability evaluation. In: *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, Montreal, Canada, 29 April – 2 May. IEEE, 1-4/ Available from https://ieeexplore.ieee.org/abstract/document/6334930?casa_token=RH7beTD7JYkAAAAA:iLKNBVfStjMvZHaczGb_iTf1aGQjFgBa6f5kMDqf8qGsW6x0I5Zri0lNx3BlmY-EzG0iD-GemcgZbQ [accessed 19 May 2021].

Nitecki, S. (2019) *Cross-Platform Mobile Apps Development – Pros and Cons.* NetGuru. Available from https://www.netguru.com/blog/cross-platform-mobile-apps-development [accessed 18 May 2021].

Quesenbery, W. (2011) *What does Usability Mean: Looking beyond 'Ease of Use'.* WQ Usability. Available from https://www.wqusability.com/articles/more-than-ease-of-use.html [accessed 18 May 2021].