**Validation and Full Demo**

**Validation and Full Demo**

**GameSAT**

Andris Docaj

Mostofa Adib Shakib

Ayush Dhumal

Scott McFarlane

Shree Shrestha

1. **Project Description**

GameSAT is an android device app which intends on making learning fun. This aim is achieved by transforming learning into an enjoyable play, where the user’s competitive aspects are challenged. The main intention of GameSAT is to prepare high school students for the verbal section of the SAT exam, or undergraduate students for the verbal section of the GRE exam.

Our app will first require the user to create an account so that their progress can be traced. Once an account has been created the user is then presented with the option of playing or training. The playing option allows the user to play a word game, which challenges the user’s vocabulary comprehension of words. In addition to this, the user can also play a passage game, where randomly chosen short passages, obtained from various sources, are presented followed by questions. The questions will be related to the passage and will test the user’s reading comprehension, and their understanding of words in context. The play mode will start the user with 5 points and the maximum number of points that the game allows is 30. If the score reaches zero, the game is over. The game will have 3 levels, and each level will challenge the user by imposing a time limit on the answer window. This time limit will reduce as higher levels are reached. Level 1 will give the user 30 seconds to answer a question, followed by 20 seconds in level 2, and finally only 10 seconds for level 3. If the user misses a question, they will be penalized, but a correctly found answer will lead them to a reward. The users will have the option to skip questions they find too difficult and avoid a point penalty.

In training mode no scoring is kept, and no time limit exists. The training is available for words individually but also for passages of text. The training mode is strictly designed for free learning, without any pressure. In this mode the questions are randomly chosen, but if a question is missed it will show up more often than others, until the user correctly selects for a particular number of times.

1. **UML diagrams**
   1. **Use Case diagrams**

The first possible scenario the user of our app will experience is that of logging-in or creating a new profile. The use case diagram for this situation looks as follows,

Chart

Description automatically generated

Figure 2: Use Case for logging-in.

If the user is already registered, then they will access the **Log-in** functionality. On the other hand, if the user is new, the New User extension point activates the **Create Profile** functionality, which prompts the user to create a new profile.

The user upon logging in can decide to play a game, in which case the diagram capturing this scenario will look as follows,

Diagram, schematic

Description automatically generated

Figure 3: Use case for Game Mode.

In **Game Mode** the user has access to a **Help** functionality which provides the user with the rules of the game. The user can also **Skip** questions they find difficult. Moreover, a **Timer** exists for each question given to the user. The user will also be able to track their score through the **Game Score**. The **Level** which the user is in, will also be available. In addition to this, the user can see where their score ranks by observing the **Global Score**. In **Game Mode** the user can choose to play a **WordGame** or a **PassageGame**.

If the user selects to train instead of play, then the use case diagram will look as follows,

Diagram

Description automatically generated

Figure 4: Use Case for Train Mode.

In **Train Mode** the user can access the **Help** functionality which will provide the user with information regarding this aspect of the app. In addition, the user can choose to train word vocabulary by selecting **WordTrain**, or passage comprehension by selecting **PassageTrain**.

* 1. **Class diagrams**

The general class diagram looks as follows,



Figure 5: Class diagram for GameSAT.

All the objects are connected through association, so they know about each other’s presence. The user will communicate with the GameMain object, and the registration of the user’s data will look as follows,

A picture containing diagram

Description automatically generatedFigure 6: GameMain object.

GameMain presents the user with a display and acquires the user’s information. If the user is new, then Registration stores the user’s data. The user can also observe the GlobalRank where the top 10 scores are recorded.

The user can further provide GameMain with the choice on whether to play or train. This case is illustrated below,

Diagram

Description automatically generated

Figure 7: Play Mode options.

The userModeChoice in play mode takes the user’s choice and determines whether to go in Game Mode or in Train Mode. If the Game Mode option is selected, we will have the following,

Diagram

Description automatically generated

Figure 8: Game Mode options.

In Game Mode the function userGameChoice applies the user’s preference to play word games or passage games. If the user selects WordGame, then questions determined by the Question object will be presented to the user. The Level object will keep track of the levels, and the time each level allows to answer a question. The WordGame object will also keep track of the score, penalties, and bonuses of the user. The same features exist if the user selected to play PassageGame, with the only difference being that the Question object would present the user with questions involving passages instead of singular words.

If the user decided to train instead of play, we would have the following diagram,

Diagram

Description automatically generated

Figure 9: Train Mode options.

In train mode the user is informed through the userTrainChoice function that they can train with words or with passages. If the user selects to train with words, then WordTrain will inform the Question object of this and provide the user with word questions. On the other hand, PassageTrain will provide the user, through its interaction with the Question object, with passage questions.

* 1. **Sequence diagrams**

Diagram

Description automatically generatedThe sequence diagram that illustrates the logging in of the user, will look as follows,

Figure 10: Sequence diagram for logging in.

The user informs the GameMain object through the userRegStatus if they are new or active users. If the user is new, then the function isUserNew informs the Registration of this fact, and the user’s data is stored using storeUserData.

On the other hand, if the user decides to play a game the sequence diagram will look as follows,

Diagram

Description automatically generated

Figure 11: Sequence diagram for play mode.

The user makes their choice available to GameMain, and GameMain informs PlayMode of the user’s choice using userModeChoice. Afterwards, GameMain informs GameMode on the user’s preference of which game they like to play. If the user selects to play WordGame, then questions particular to this game will be selected and displayed. On the other hand, if the user decides to play a passage game, then PassageGame will select and display questions particular to this game.

Diagram

Description automatically generatedThe sequence diagram for the train mode,

Figure 12: Sequence diagram for train mode.

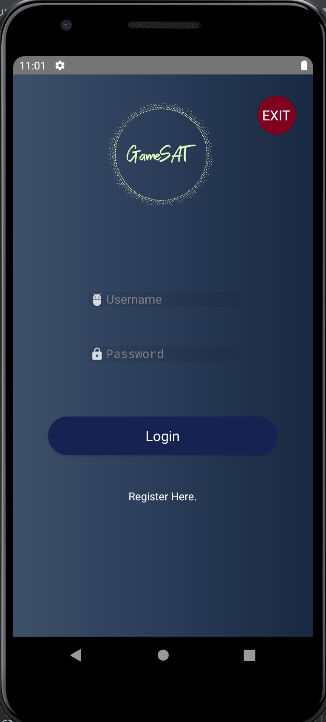
is very similar to that of the GameMode. PlayMode is informed of user’s choice through userModeChoice, and afterwards GameMain informs TrainMode using userTrainChoice on the user’s preference for word training or passage training. If the user decides to train on words, then WordTrain will select and display word questions. If the user wants to train with passages, then PassageTrain will select and display passage questions.

1. **Implementation Overview**

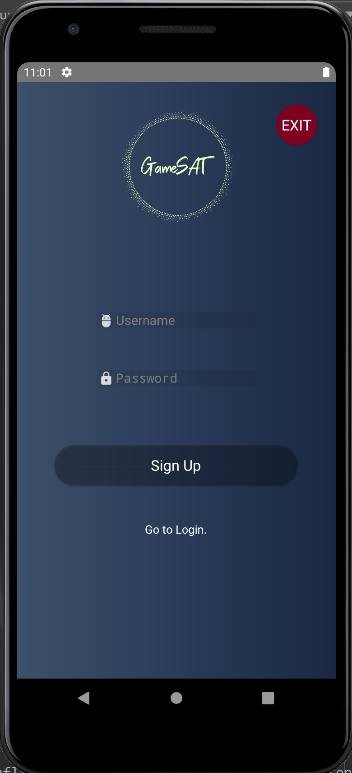
The implementation of the project was quite enjoyable because none of our group members had much experience with android development, and the project provided us with the opportunity to learn new things. For this project we had a GitHub repository where edits could be managed. Two of our team members will handle the question database part, while two others will handle the app design part. The final of the group member acts as a mediator helping both groups with their issues, and with development. The tools we used for the project are **Android Studio**, **XAMPP**, **Java**, and **PHP**.

1. **Functionalities Implemented**

The first functionality we implemented was to record and register the user’s data. The initial screen presented to the user is the following,



Upon clicking the registration button, the user is taken to the following sign-up screen,



For this project we decide to use **XAMPP** for our database management, using the **Apache** server and **MariaDB**. The following picture illustrates what we are currently running,

Graphical user interface, application

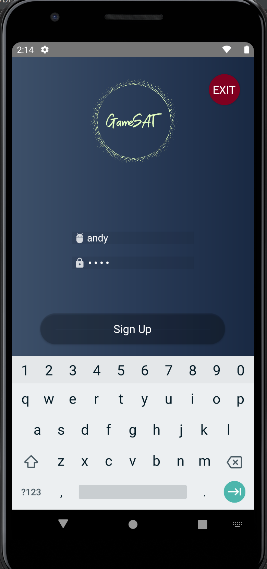
Description automatically generated

The initial database, called logindb, looks as follows,

A picture containing text, screenshot, monitor, indoor

Description automatically generated

Presently there are no users, but if we register a user through our sign-up page as follows,

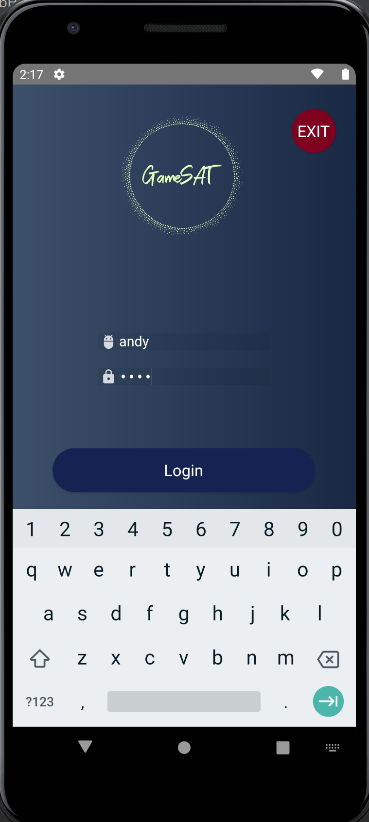


Then our database will register the user, and this can be proved from the following pic,

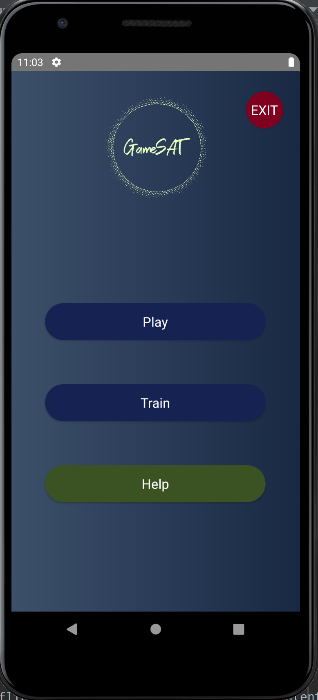
A picture containing text, screenshot, indoor, monitor

Description automatically generated

Our user’s data is now stored into the login database, and this data can be used to access the full functionality of our app. Upon entering the data in the log-in screen as follows,



The user will be greeted with the following welcome screen,



The **Help** button will provide the user with the following information,

A picture containing text, electronics, monitor, computer

Description automatically generated

The **Help** activity is made up of two buttons as shown below,

Text

Description automatically generated

The exit button closes the whole app by calling **finishAffinity**, followed by a **System** exit. While the **Back** button, takes the user to the welcome screen.

The signup code employs php to interact with **XAMPP** as follows,

Text

Description automatically generated

If the user has inputted some data for the username and password, then this data is put into the database located at a particular IP location. If the operation of putting this data is successful, then an intent to move from the signup activity to the login activity is started, the data entered by the user is put under a label, and the signup activity is finished.

Otherwise, we inform the user that all fields are required, as in,

Text

Description automatically generated

The login on the other hand takes the information sent by signup, as follows,

Text

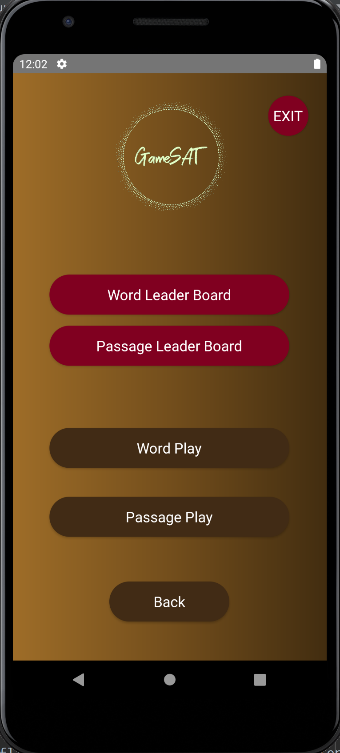
Description automatically generated

So, we first take the data put in place by the signup, before it finished, and check that the username and password of this data are not null, then we interact through php with the **XAMPP** database, to make sure that this username and password match those in the **XAMPP** database. If this is the case, then we start an intent to move to the Welcome screen and finish the login activity. Also, as can be seen in the following pic, the exit button finishes all activities on the stack and calls a **System** exit. This format is followed for all the exit buttons present in all the activities of our app. If the login was unsuccessful usually due to the user not inputting all the information, then a message prompting the user to enter all the required information is displayed, through the **Toast** feature.

Text

Description automatically generated

If the user decides to play the game, then the following play screen will be presented to the user,



The code that enables the functionality of the play screen looks as follows,

Text

Description automatically generated

The back button takes the user to the **Welcome** screen using intents. Similarly, the play word button takes the user to the **WordPlayQuestion** object, while the play passage button takes the user to the **PassagePlayQuestion** object. On the other hand, the leader board is a single object,

Text

Description automatically generated

And, both buttons **bWordLeaderBoard**, and **bPassLeaderBoard** send the user to the **LeaderBoard** activity. But, in addition to creating an intent, we also put extra information on the intent, this way if the **wordBoardSelect** value caught in the **LeaderBoard** activity is 1, then we display the word game leader board, but if the value is 0, then we display the passage game leader board.

The following code illustrates the previous point,

Text

Description automatically generated

In addition to this, the leader board uses a recycler view to display the username and the game completion time measured in milliseconds.

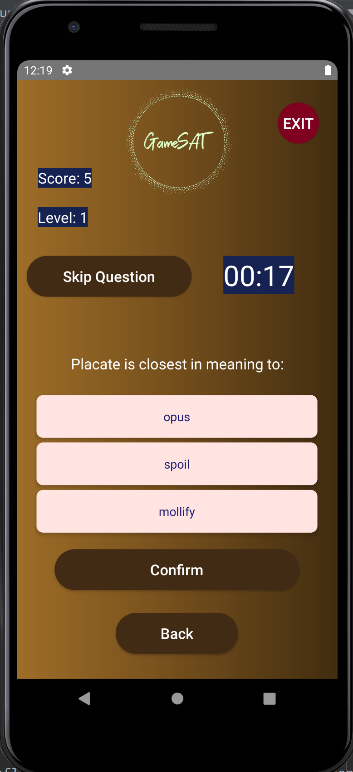
As can be seen from the following code,

Text

Description automatically generated

We take the users from either a word completion timetable, or a passage completion timetable, using a cursor. But we limit the number of users displayed to 5, and this aspect is controlled by the **displayCount** variable.

If the user decides to play a word game, the following screen will be displayed,



The default level is 1, the initial score is 5, and the timer starts from 30 seconds. The user can **skip** questions without a point penalty. The user can enter the answer by selecting any of the choices provided and pressing confirm. Once **Confirm** is pressed the timer stops, and the user selection is checked for correctness. If the user decides to go back, they be taken back to the play screen, and will lose all progress, so a warning is displayed. The code for the **Back** button looks as follows,

Text

Description automatically generated

We use an alert dialog to warn the user and give them a pause before completely deciding to leave using the **Back** button. The code for the **Skip** button looks as follows,

Text

Description automatically generated

The function **updateWordQuestionCorrectVal** exists in the database helper object and assigns each correct value attribute the value of 1. The correct value attribute is part of the question properties, and is used to determine which questions were skipped, or missed. This way later we can gather these questions in a list, using this attribute, and we can use them for training purposes. The question index is the current question index, a random number which is used to select question objects in an array list. While the previous index is needed to avoid having the same question repeating twice. The timer needs to be cancelled when a question is skipped, because a new timer needs to start, for the question displayed by the **showNextWordPlayQuestion** function.

The first part of the show next word function looks as follows,

Text

Description automatically generated

We first clear the buttons, because we do not want the previous question to overlap with the next one. Then before anything we first check if a game ending condition has been reached. If such a condition has not been reached, we determine in which level we are at. In addition to this, the **levelReached** variables ensure that if we reach a level, we do not move away from it, but are only presented from questions from this level. The function **setWordQuestionList** takes in the level variable and picks the right question list for that level. Next, based on the size of the question list, we randomly choose and index that picks a question from the list. And finally, we display the question. The second part of the show next word function looks as follows,

Text

Description automatically generated

After displaying the question, we start the timer, and we set the word question answered variable to false, since we want the confirm button to affect this variable. We then set the text of the confirm button to **Confirm**. On the other hand, if a game ending condition has been reached, the user reaches **0** or the maximum score of **30**, then we finish the play using a finish function.

The countdown function uses the values of the **onTick** method, and has the following form,

Text

Description automatically generated

The on-finish part deals with cases where the time runs out, but no choice has been made. In this case we check the answer regardless.

The update countdown function looks as follows,

Text

Description automatically generated

Using the update countdown, we change the color to red, when the time is less than 5 seconds. The find question index function uses a random number which uses the question list size as the seed to obtain the next question.

Text

Description automatically generated

If the word question has not been answered we determine if any of the radio buttons is checked, and check if the answer selected by the user is correct by using a check answer function. Otherwise, display the next question to keep the game going.

The check answer function looks as follows,

Text

Description automatically generated

We first set the question answered variable to true, since the user reached this stage by selecting the confirm button. Then, we cancel the timer since the user answered the question. Next, we determine how much time the user took to answer the question and add this value to the total completion time of the user. After this we determine if the choice the user made was correct or not. If the choice was correct the user’s score is increased by one. On the other hand, an incorrect choice get’s the score decreased by one, and we mark the question to be used in the training part. Regardless on the user’s choice on whether to answer the question by selection, or letting the timer run out, the question will be displayed by the following function,

Text

Description automatically generated

All radio buttons are initially set to red but depending on which radio button contains the correct answer, their color will change to green, and their text will be different also. Also, if the user score has reached a game ending value, the confirm button will display **Finish** instead of **Next**.

When the game finishes we call on the following function,

Text

Description automatically generated

As can be seen, before we finish the activity to go to the **GameOver** object we put extra information on the intent to determine if the game was won from a word game or passage game. If **wScoreExists** is 1, then the game was won from a word game, otherwise it was won from a passage game. Moreover, we also pass the score, and time of the user as extra information, in case we want to use these values later. Then we finish the activity.

The game over object contains the following code,

Text

Description automatically generated

We first take the extra information coming from either word play or passage play. If **wScoreExists** is 1, we first determine if the game score is different from zero. If the score is positive, this means the user has won, and we insert the **username**, and the user **time** into a table residing the database helper object. If **wScoreExists** is zero, this means that the information is coming from passage play, and we again determine if the user has won or lost the game. If the user has a positive game score, we store their **username** and game **time** in to a passage record table. At the very end we call the show message function to display an appropriate message,

Text

Description automatically generated

So, if the score is 0, we display a game over message, otherwise we congratulate the user on winning.

The passage game is exactly like the word game, the only difference is that we created different tables to keep track of the different questions. The word questions are short, while the passage ones contain a sentence followed by a question. The functionality though between the two is the same.

For the training part the user will first be presented with the following options,

A picture containing text, electronics, monitor, iPod

Description automatically generated

The code will look as follows,

Text

Description automatically generated

So, we first take all the questions that had a correction value different from **0**, which contained the ones that were **incorrectly** selected, or **skipped**. Then we reset the correction values, so new games can provide their own lists. We deal with two lists of questions in training mode, with the question list of questions that the user needs help with, and a question list of all the questions present, **150** in total for the word game, and **111** for the passage game.

Depending on which list is non-empty we shuffle the questions to provide a random session each time. Then we call the show next question function,

Text

Description automatically generated

As can be seen from the code above, if the train question list, of questions that are skipped or missed, is non-empty, we pick a question from it first. Otherwise, we randomly pick a question from the **all\_questions\_list**, which contains all the questions. Afterwards we display the question, and set the answered question to false, as we did in the play part.

The confirm button is similar looking to the one in the play part,

Text

Description automatically generated

Here we first check if the question has been answered, if it has not, then we determine which button the user has selected, and compare this to the correct choice for the present question using the check answer function. Otherwise, we show the next question in random way.

The answer checking function looks as follows,

Text

Description automatically generated

First, we determine if the user has found the right answer, and if this is so, then we check the correction value to see if this is a question that the user missed or skipped at some earlier point. If this is the case, then we increase the correction value, and then check if the new correction value is less than **3**, meaning that the question should not be displayed more than **twice** to the user. The value of **minRepeats** is **3**. If the new correction value is less than **3**, then we reset the question correction value in **all\_question\_list**, we then clear the training list, and recalculate it to determine what questions should be in it. If the correction value is greater than or equal to **3**, then we reset the correction value to **0** in the **all\_questions\_list**, and clear the training list, and recalculate it again from the **all\_question\_list**. Correction values that have a **0** value will not enter the training list, only those with a positive value. We also determine the size of the training list, so that we can randomly generate a number using this size as the seed. In the end we always show the solution which is a function like that shown previously.

The is word question on the train list function,

Text

Description automatically generated

Checks to see if the present question is in the train list or not. If it is, then we do not add it to the train list, other we do. The passage training part is exactly like the word training part, with the only difference being that the questions are longer in the passage table.

The question object has the following inner structure,

Text

Description automatically generated

We add the questions in the database helper by first creating a table as shown below,

Text

Description automatically generated

We also create a unique index to prevent the same question from being entered repeatedly. The database and its tables are updated by changing the database version number as follows,

Text

Description automatically generated

The questions are added to the database using the following function,

Text

Description automatically generated

We ignore cases that violate the index we created.

And, questions are inserted as follows,

Text

Description automatically generated

The correction values are affected using setter functions. On the other hand, they are reset using the SQL update function as follows,

Text

Description automatically generated

We can get questions in the play and train modes using a function of the following form,

Text

Description automatically generated

We use a cursor that gets obtained by a SQL query, and if the cursor can move to an entry, we add the entry values into a new question object and add the question object to a question list. At this point we can also introduce a level variable and decide if we want to have only particular level question lists. I have only tried to describe the important parts, and the latest version of the code can be examined at development page which has a link included at the end of this report.

1. **Implementation issues**

The SQL integration with android studio was problematic. Sometimes it was more useful to use a raw query using pure SQL, while at other time using the SQL query functions native to android seemed more useful. There was no standard to follow, and when to choose one approach or the other seemed problem dependent.

1. **Member Contribution**

All the group members contributed equally. Ayush was part of the app design group and handled the interface design aspects of the app. While Scott and Shree were part of the SQL design group and handled the SQL question selection and design. Adib handled the SQL question design, the testing of the software and provided guidance between the SQL group and the app design group so there would be no misunderstandings. Andy was part of the app design group and handled the logic design of the app, as well as certain question design features and display design features.

1. **Link to the development page**

* <https://github.com/adocaj/GameSAT>

1. **References**

* <https://gre.economist.com/gre-advice/gre-vocabulary/which-words-study/most-common-gre-vocabulary-list-organized-difficulty>
* <https://codinginflow.com/tutorials/android/quiz-app-with-sqlite/part-1-layouts>
* <https://github.com/VishnuSivadasVS/Login-Signup>