**תיק מתכנת**

**General description**

The site includes games that come to help the user learn and expand his knowledge about cloud computing, with the help of a trivia game with a variety of questions on the subject.

In addition, the website has the following screens: home screen, login screen, registration screen, leaderboards screen, game screen, user’s profile screen, statistics screen, manager screen and about us screen.

**The target audience**

For people who want to learn about the whole subject of cloud computing, the site appeals to all types of the population in that it is easy to operate and includes a method of learning in a playful way that makes learning fun and flowing.

**Maintenance guide**

In this section we will explain what are the requirements of software and hardware that must exist in order to use \ continue to develop the system.

**Hardware and Software requirements:**

Before I mentionwhat are the requirements of software and hardware that must exist in order to use \ continue to develop the system, It is important to note that the entire program run in Google Collab.

In Google Collab we work with Python language.

To install all the libraries and tools that we need to run the site, we added requirements.txt file that include all the libraries and tools in the Google Collab.

To implement the backend we use the Flask library, and for implementation of the site in independent window we use with ngrok tool.

For connecting the database and external files we use Mount to the Google Drive and Firebase library.

**Database environment:**

Our system connects to Firebase Realtime Database. in order to maintain the DB, we must allow one common Firebase access to every single team member. that way, the whole team will be able to work at the same DB at the exact same time.

**External files:**

In the Google Collab we are importing an external files from Google Drive for the design and logic of the site.  
We use with HTML, CSS and JavaScript files for the whole design and logic of the site.  
Following the use of Flask combined with jinja 2, in the HTML files we import, part of the code will be written in Python.

**Order of running the entire application:**

* Run the requirements.txt file in the Google Collab (This will run all the libraries and tools needed to run the code) and wait for it to finished.
* Run the cell with the entire code and wait for it to finished.
* Click on the link that the Collab brings you and it will enter to the site.

**Backend.py**

* Importing the firebase REST-API library.
* Database class \_\_init\_\_ gets the db\_url and initializes a db connection with it.
* Database class - get\_data gets a collection name and implements the GET operation which retrieves the collection from the firebase database.
* Database class – update\_data gets a collection, target and data and implements UPDATE operation which updates the collection at target with the data.
* Database class – delete\_data gets a collection and a key of document and implements a DELETE operation which removes the document with the key from the collection.
* Database class – post\_data gets a collection and a data and implements POST operation which adds new data to the given collection.

**Collab Main Cell – server.py:**

**Initialization:**

This cell is basically the server of our application, which is written in python using FLASK framework.  
First we initialize the app and its secret key, and configure the running of the app using NGROK, which provides us a way to run the application on the internet.   
Then we initialize the database connection with our backend module and provide it the connection string from the firebase database.

**Body:**

We define all the routes of our application using @app.route(‘url\_path’) and the method to be invoked in each of the routes. Therefore, each route has a method corresponding to it, right below it in the code, and each method contains the logic to be applied in each route.

**@app.route(‘/’) –** home page route in which loads the index.html file.

**@app.route(‘/<nickname>’) –** home page route which displays the nickname of the user which is logged-in.

**@app.route(‘/play/<game>’) –**  trivia page route which gets all the question from the database and sends these questions to the playtrivia.html file in the question\_list argument, and displays the playtrivia.html file. If the user is not logged-in, he cannot access the play trivia page.

**@app.route(‘/play/<game>’,methods = [‘POST’]) –**  the game\_end method is invoked when the user clicked on finish game. game\_end method saves the information about the currently played game which contains score, answers, array of scores and a Boolean about broken record and this information is updated and saved in the current session and in the DB. Finally, the user is redirected to the game\_statistics page.

**@app.route(‘/statistics) –** game statistics page route which displays a graphs about the currently played game.

**@app.route(‘/register’) –** registration page route in which we load the register.html file.

**@app.route(‘/profile’) –** profile page route which loads the MyProfile.html file and sends the currently logged-in users information which contains highscore, nickname, username.

**@app.route(‘/profile’, methods=[‘POST’]) –** profileChanges() method is invoked when the user submits the updates of his information which contains username, password and nickname. The user with the new data is updated in the database and the page refreshes.

**Def score(element)-**  converts the element’s highscore to int. it is used in the leaderboard method.

**@app.route(‘/leaderboard’) –** leaderboard page route which loads the leaderboard.html file and sends the reversely sorted list of users to this file (which means we can access the list of users in the leaderboard.html file).

**@app.route(‘/about’) –** about us page route in which we load the AboutUs.html file.

**@app.route(‘/logout) –** clears the current session the redirects the user to the home page in the index route.

**@app.route(‘/login’) –** login-page route in which we load the login.html file.

**@app.route(‘/login’,methods=[‘POST’]) –** checks the user’s username and password and redirects the user to the home page with his nickname shown if the user’s details are correct, otherwise shows an error label on the same page.

**@app.route(‘/register’,methods=[‘POST’]) –** gets the currently – registering user’s username, passwords nickname and ID, adds a new user with these attributes to the database, saves current user in the current session and logs the user in.

NOTE: in the routes above **methods=[‘POST’] is invoked when the corresponding HTML form is submitted.**

**@app.route(‘/manager’) –** manager() method is invoked. If the user role is ‘admin’, the user is redirected to the manager page – the manager.html is loaded. Otherwise, the user is stays on the home page in the index route.

**@app.route(‘/manager/<qID>’) –** getQuestion() method is invoked. The method gets all the question from the database and sends the currently chosen question as a parameter to the question.html page which is loaded.

**@app.route(‘/manager/<qID>’,methods = [‘POST’]) –** operations(qID) is invoked when the user submits the form containing the questions attributes with the “update” button.  
qID – question number, which is also the question key in the database in the questions collection.  
Moreover, the fields of the form which contain answer options, correct answer, question description and question difficulty are updated in the currently viewed question and the user is redirected to the manager home screen.

Furthermore, if the user clicks on the “delete” button, the question is deleted from the database and the user is redirected to the manager home screen.

**@app.route(‘/manager/allQuestions’) –** all question page route which displays the allQuestions.html file.

**@app.route(‘/manager/addQuestion) –** add question page route which displays the addQuestions.html file.

**@app.route(‘/manager/addQuestion’,methods = [‘POST’]) –** postQuestion() is invoked when the user submits the form containing the questions attributes with the “add” button.  
Moreover, the fields of the form which contain answer options, correct answer, question description and question difficulty are saved in the new question object and the object is posted to the database. The new question is added with the following number of the last question number.

**def SaveGame(id\_reference) -**The `SaveGame` method saves the game information which includes the time, score, right\_answers, array\_answers to a database using a provided `id\_reference`.

**@app.errorhandler(404) –** error page route which displays the 404.html file.

**Requirements file – requirements.txt**

Requirements.txt saves all the needed libraries, frameworks, modules and tools which needed to be downloaded and imported in the initialization part of server.py for the application to run successfully.   
flask\_ngrok tool is used to run the application on the internet.  
flask is used for building the server, defining routes for the pages, sending data to the html pages and applying some logic.  
firebase is the REST\_API library which is used for all the CRUD operations.  
pyngrok is a python wrapper that manages its own binary, making ngrok available via a convenient python-api.  
firebase\_admin allows us to interact with the firebase services from the backend code.

**Playtrivia.html – in this file we write script with code:**

**window.addEventListener('load', function() { ... }) :** This code adds an event listener to the window object for the "load" event. The "load" event is fired when the entire page, including all its resources (such as images and scripts), finishes loading.

**var questionList = {{ question\_list|tojson }} :** This line assigns the value of the `question\_list` variable, which is passed as a context variable from the server-side, to the `questionList` JavaScript variable. The `tojson` filter is used to convert the server-side variable to a JSON string.

**fetchQuestion(questionList) :** This line calls the `fetchQuestion` function, passing the `questionList` as a parameter. It initiates the process of retrieving and populating a random question using the provided question list.

**question.js file**

**fetchQuestion(question\_list):** Fetches a random question from the question\_list array. It updates the question content in the HTML elements, adds event listeners to the answer buttons, and handles the logic for moving to the next question or finishing the game.

**updateButtons():** Clears the event listeners from the answer buttons.

**removeButtonsListners(option):** Removes the event listener from the specified answer button.

**addButtonsListeners():** Adds event listeners to the answer buttons.

**updateQuestionContent(randomQuestion):** Updates the question and options in the HTML elements based on the randomQuestion object.

**checkAnswer(optionNum):** Checks the selected answer against the correct answer. It disables all answer buttons, updates the styling and progress bar based on the selected answer, and increments the score and number of correct answers if the answer is correct.

**clearButtonStyling():** Clears the styling of the answer buttons, hides the answer status, and resets the display of next question and answer buttons.

**updateProgressBar(currentQuestionIndex):** Updates the progress bar by adding the "active" class to the steps corresponding to the answered questions.

**showPopup(CorrectAnswer):** Displays a pop-up with the correct answer.

**closePopup():** Hides the pop-up.

**fetchPost(data):** Performs a POST request with the given data. It constructs the URL dynamically based on the current page's URL and sends the data as JSON. The response is expected to contain a url property to redirect the user to.

**statistics.js**

**convertStringToArray(string):** A utility function that converts a string representation of an array into an actual array. It replaces single quotes with double quotes to ensure valid JSON formatting, parses the JSON string, and returns the resulting array. If an error occurs during parsing, an empty array is returned.

**skipped(ctx, value):** A custom function used as an argument in the chart configuration to determine if a data point should be skipped or not. It checks if the current data point is marked as skipped and returns the provided value accordingly.

**down(ctx, value):** Another custom function used as an argument in the chart configuration to determine if a data point is in a downward trend or not. It compares the y-values of two adjacent data points and returns the provided value accordingly.

**monthNames:** An array of month names used to format the x-axis labels in the third chart.

**arrayGames:** Converts the dataset gamesData from the graphElements element into an array of objects.

**arrayGamesScore:** Maps the arrayGames array to extract the "score" property from each object, creating an array of game scores.

**arrayGamesTime:** Maps the arrayGames array to format the "time" property of each object into a string representation of the month and date.

**graphsdata:** An array containing the configuration data for each of the three charts. Each element in the array represents a separate chart and includes labels, datasets, and other chart-specific options.

**new Chart(canvasElement, {...}):** Creates a new instance of the Chart class with the specified canvas element and configuration options. The first chart is a doughnut chart.

**new Chart(canvasElement2, {...}):** Creates a new instance of the Chart class for the second canvas element. This chart is a line chart.

**new Chart(canvasElement3, {...}):** Creates a new instance of the Chart class for the third canvas element. This chart is also a line chart.

Overall, the code sets up three different types of charts (doughnut, line, and line) using Chart.js, populating them with data from the provided datasets, and rendering them on the page.

**allQuestions.html – in this code the script is in the HTML file:**

**initializeApp(firebaseConfig):** This function initializes the Firebase app with the provided configuration (firebaseConfig), which includes the API key, authentication domain, database URL, project ID, storage bucket, messaging sender ID, app ID, and measurement ID. It sets up the app to communicate with Firebase services.

**getFirestore(app):** This function returns a Firestore instance associated with the provided Firebase app (`app`). Firestore is a flexible, scalable NoSQL cloud database provided by Firebase.

**ref(getDatabase()):** This function returns a reference to the default Realtime Database associated with the Firebase app. It's used to access the Realtime Database services.

**get(child(dbRef, 'questions/')).then((snapshot) => { ... }):** This function retrieves data from the Realtime Database at the location `questions/`. It returns a promise that resolves with a snapshot of the data at that location. The snapshot contains methods to access the data.

**$input.addEventListener('input', () => { ... }):** This function adds an event listener to the input field with the ID "txt". It listens for input events, which occur when the user types or modifies the input value.

**qArr.filter((question) => { ... }):** This line filters the `qArr` array based on the condition specified in the callback function. It checks if the lowercase question text includes the lowercase value of the input field. The filtered array is assigned to the `filtered` variable.

**filtered.forEach((question, index) => { ... }):** This function iterates over the `filtered` array and performs operations on each question in the filtered array.

**HTML files:**

**404.html –** page which is displayed when accessing not existing route.

**AboutUs.html –** displays the page with information about the application and the development team.

**MyProfile.html –** displays the page of the logged-in user profile that includes his information.

**addQuestion.html –** displays the manager page to add new question.

**allQuestion.html –** displays the manager page that shows all the existing question.

**gameStatistics.html –** displays the statistic page to show the player the data about the currently played game with graphs.

**Index.html –** displays the home page of the application.  
**leaderboard.html –** displays the leaderboards table page with the users highscore and their ranks.  
**login.html –** displays the login page.

**manager.html –** displays the manager’s home page with his unique features.

**playtrivia.html –** displays the trivia game page with the questions and the current score of the user, and the progress bar.  
**question.html –** displays a page with specific question that selected by the manager with the option of deleting and editing the question.  
**register.html –** displays the register page.

9.Maintence Guide

9.1 Database:

A black text on a white background

Description automatically generated with medium confidenceA screenshot of a computer

Description automatically generated with medium confidence

**We have 3 tables: Users, Games, Questions**

User’s entity attributes: key, highscore, id, level, nickname, password, role and username.

Games’s entity attributes: key, game-info.

Question’s entity attributes: correct, option1, option2 , options3, option4, question and questionlevel.

**The relationships are as follows:**

A User can have multiple(0-\*) Games, represented by the line connecting Games and Users.

Each Game can have one User, represented by the line connecting Users and Games.

Each Game have 10 Questions, represented by the line connecting Games and Questions.

Each Question can be displayed in multiple (0 - \*) Games, represented by the line connecting Question s and Games.

Each Game have 1 game\_info, represented by the line connecting Games and game\_info.

9.2 Deployment Diagrm:

A picture containing text, diagram, screenshot, rectangle

Description automatically generated