ThinkItThru! Final Report

By: Jensen Schmidt, Zachary Travis, Levi Wright

Table of Contents

Introduction

ThinkItThru! is a productivity application designed to help students manage a busy schedule and incentivize hard work. It is common for students to have many deadlines to keep track of, along with their personal/professional goals. They often become stressed from poor time management and develop unhealthy study habits. These difficulties can affect student motivation and their ability to meet deadlines.

ThinkItThru! organizes user-input tasks into a schedule that presents goals for them to focus on today. By focusing on today, the user is presented with a clear way forward to finishing many different goals. The scheduler will consider each task's deadlines and priorities. If there is conflict between the task deadlines and hours of work required, the scheduler will quickly recognize and report it to the user. This gives users the ability to react appropriately to deadlines they are projected to miss.

Completing tasks earns in-game experience points to spend on rewards, driving users to spend time on their tasks. The "Mind Your Garden" gameboard is where users can go to see the "fruits" of their labor.

Technology

What Technologies Were Used, and Why

Brief Overview of the technologies we used:

- Visual Studio Code (IDE and the Live Server Extension by Ritwick Dey)
- Firebase Authentication
- Firebase Cloud Firestore
- Firebase Hosting
- JavaScript, HTML, and CSS languages
- CSS Semantic UI 2.5 Framework
- CSS Bootstrap 5.3 Framework
- GitHub (Site, Desktop, and Visual Studio Code Extension)
- Google Chrome

First on this list is the integrated development environment (IDE) used for this web application. Visual Studio Code utilizes extensions to provide support for almost every type of programming language and text-editor-based utility, often made by both its developer, Microsoft, and its incredibly active community. One of the main extensions we have used with Visual Studio Code is "Live Server," an extension by Ritwick Dey that allows the user to open html pages into the browser on a development local server. This extension has the added

benefit of creating pages that actively reload when changes are made to the source code, allowing for a smooth and expedited debugging process.

The next piece of technology on the list, Firebase Authentication, handles many of our user-verification needs through Firebase's systems. Through enabling the different methods of authentication—for ThinkItThru!, this is email/password and Google authentication—we can rely on Firebase securely storing and handling the passwords for our users instead of directly taking them. Along with the password storage/verification, Firebase also accepts various federated login methods such as Google or Facebook sign-in. This provides a degree of security and freedom for the user to decide the amount of data that is stored about them within the web application.

In fact, Firebase Cloud Firestore handles the secure storage of ThinkItThru!'s user's data. While the current iteration of ThinkItThru! has its security rules set to be open (for testing purposes), the comments present in the system show the rules it adheres to outside of testing:

```
// ADD DATA
//match /users/{userId} {
//  allow write: if request.auth != null && request.auth.uid == userId;
//}

// GET DATA
//match /users/{userId} {
//  allow read: if request.auth != null && request.auth.uid == userId;
//}

//match /users/{userId}/tasks/{taskID} {
//  allow read: if request.auth != null && request.auth.uid == userId;
// allow write: if request.auth != null && request.auth.uid == userId;
//}
```

Figure - Commented Security Rules to Be Implemented

These rules, when active, only allow the verified user who created the data to be able to read that same data. Firebase Firestore is a flexible and scalable NoSQL cloud database using Google's cloud infrastructure. It's intended for professional mobile, web, and server development that keeps data synced across client applications. Through its use of Documents and Collections, Firestore creates an effective database that is scalable for a sizeable user population.

To bring the web application online, ThinkItThru! continues its Firebase trend and defers to Firebase Hosting. This process, while not completely intuitive, provides fast and secure hosting for the web application. This feature assigned a URL to ThinkItThru! and displayed the application based on the files placed in a public folder from our root directory. There are multiple options for deployment; one involves compressing your site data into the "index.html" file through npm, and the other involves simply using multiple files within the public folder (the option that the current iteration of ThinkItThru! took).

Of course, a discussion on ThinkItThru!'s files would not be complete without mentioning the languages used to create the application: HTML, CSS, and JavaScript. These languages are extremely common amongst web applications and serve as the foundational building blocks for ThinkItThru! and its various components. While Firebase handles a lot of the user data, the UI

and boxes for input are created through a mixture of an HTML skeleton and CSS Frameworks, such as Semantic UI for the authentication page and Bootstrap for the rest of the site. Semantic UI and Bootstrap both offer easy-to-use and professional-looking design through class tags that can be input into various HTML elements. Both frameworks ensure that minimal amounts of CSS are written by the developers of ThinkItThru! and allow for a bigger focus on JavaScript functions. The JavaScript of ThinkItThru! handles all the task/objective logic that our system is based around and creates a usable bridge between our database and user input. It also powers our "Mind Your Garden" game by dynamically inserting HTML into the page based on a string for the user's "Gameboard."

ThinkItThru! is stored on a developer platform called GitHub, a platform meant for developers to create, store, manage, and share their code. This platform has a desktop version for ease-of-use in pulling or pushing changes to the project's repository, and even has Virtual Studio Code integration through an extension that allows you to push or pull from the repository on the platform. The platform uses Git software, providing a plethora of features; for example, access control, bug tracking, software feature requests, task management, version control, continuous integration, etc. As an aside, the browser that ThinkItThru! utilized for most of its testing was Google Chrome. This browser is one of the most commonly used browsers on the internet and appeals to a very wide audience.

Difficulties in Learning, Development, and Implementation

The chosen technologies proved to be straightforward in their initial application and use. Visual Studio Code is very intuitive and can be made incredibly versatile with the extensions it offers. Its file storage process is very straightforward (unlike an IDE like NetBeans), allowing the developers of ThinkItThru! to use it effectively without extensive experience. Google Chrome, the application's chosen browser, had no issues with the Live Server extension from Visual Studio Code or the files made from the IDE. GitHub, the application's chosen platform for distribution and version management, made pushing and pulling changes somewhat intuitive. Thanks to its desktop and Visual Studio Code integration, GitHub enabled ThinkItThru!'s developers to easily update both the main and local repositories with the site's data.

ThinkItThru!'s use of CSS frameworks initially came with its share of issues; however, they have proved fruitful in the application's latest version. Initially, these frameworks presented a steep learning curve in understanding the correct class references to use within various and HTML elements. However, both Semantic UI and Bootstrap offer extensive online support, which aided in the conception of ThinkItThru!. With this framework support in mind, the most challenging aspects of the learning curve were somewhat mitigated over time.

Firebase's suite of features required significantly more time to understand and adjust to. As mentioned in the next subsection, the developers of ThinkItThru! transitioned from using Firebase Realtime Database to Firebase Cloud Firestore, which created unforeseen issues with the subcollections of the database. The authentication feature of Firebase was more straightforward to understand and provided many usable functions for verifying user authentication and status across the site. Firebase also simplified the authentication process

through its session management, allowing users to maintain their verified status between pages.

The biggest difficulties emerged later in the development process, particularly during the integration of ThinkItThru!'s Firestore database with the current JavaScript functions, and further intensified during the deployment of the software to a hosted URL. These difficulties arose from challenges within the Firestore features, mismatches in data that needed to be stored or utilized within JavaScript functions, and issues with security rules that restricted full database functionality. Deployment issues included changes in certain file references when the site was hosted, occasionally causing the site to fail in locating the "newTask.html" file and redirecting to Firebase Hosting's "404.html" file. In other cases, the site would inadvertently revert to using an outdated JavaScript file for the "dashboard.html" file in the "public" folder.

Originally Proposed Technologies. Why were they switched?

Starting with the visual change, ThinkItThru! was originally designed without any CSS frameworks, with the developers of the application opting to style the site using plain CSS instead. This approach proved challenging; it created difficulties in designing a professional-looking application and added a layer of complexity in designing the "tasks" and "objectives" to meet the original requirements. However, the switch to the new user interface necessitated a redesign of the JavaScript functions. Once the JavaScript was reinstated, the new design proved efficient in generating the "tasks" within the user dashboard, as well as designing subsequent pages for the site.

The next change to the application did not prove entirely useful: the switch from Firebase's Realtime Database to Firebase's Cloud Firestore. Both databases, given sufficient time, are excellent choices for managing a web application's user data. However, this change came very late in the development process and introduced unforeseen issues with the "Tasks" information. Due to Firestore's structure, the "Tasks" were organized as subcollections within the user document and required unique identifiers. During the development of ThinkItThru!, the app struggled to effectively iterate through these subcollections within Firebase. This issue emerged near the end of development, and the developers did not have enough time to resolve it. Despite the usability and improvements that Firestore brought to authentication and user data storage, ThinkItThru! might have avoided its major issues had it maintained a consistent database structure throughout its development.

Licenses

- Firebase services: https://firebase.google.com/terms
- Semantic UI: https://semantic-ui.com/cla.html
- Bootstrap 5: https://getbootstrap.com/docs/5.0/about/license/
- GitHub License (our repository isn't completely licensed to be open source, would require further steps): https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository
- Visual Studio Code: https://code.visualstudio.com/license

- “Grass and Plants for 2.5D or Topview Games” - Royalty free: https://artisan-studio.itch.io/grass-and-plants-for-25d-or-topview-games
- “100 hand picked pixel flowers” - Royalty free: https://btl-games.itch.io/pixel-art-fauna-asset-pack

Design

Classes

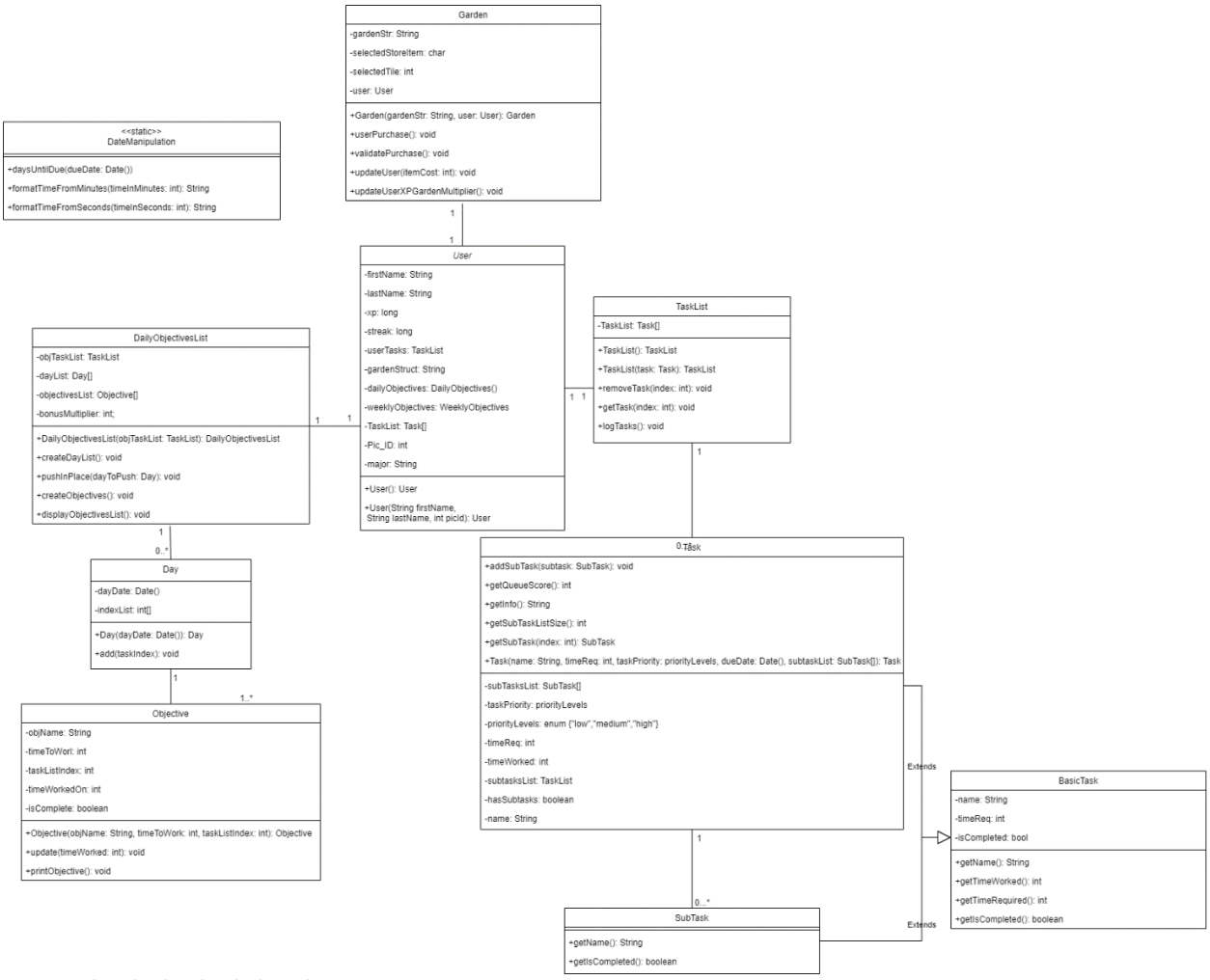

Figure - The ThinkItThru! Class diagram

The following classes are implemented and utilized within the ThinkItThru! site's JavaScript Scripts:

User

The User object contains all necessary information from any hypothetical account on the web page. It contains personal information about the user, such as their first and last name, a picture ID, and their major. It also contains information about their xp such as their Garden layout, set as a String, their earned xp, their streak (unutilized in this build), and their Garden multiplier for their xp (which increases xp gain as the Garden is built). It also contains necessary data structures to access their Tasks for each webpage, such as their TaskList, Daily Objectives List, and Weekly Objectives List (the latter being currently unused).

BasicTask

BasicTask is the parent class of both Task and SubTask. While SubTask became simplified in design, we have not restructured the class hierarchy otherwise. BasicTask holds information on a Task's name, time required to work on it, work time completed on the Task, and a Boolean value for if the Task is completed.

Task

Task adds more information to the BasicTask constructor, such as a list of SubTasks, the priority of the Task, and the Task's due date. Aside from holding info, Task can return information on the SubTask List or add a new one. The Object can also return calculations such as the time remaining to work, the queue score of the Task, and the days until the Task is due.

TaskList

TaskList holds an array of Tasks that can be changed through its methods. A Task can be added, in which case queue scores are compared to push a Task in place, removed, or retrieved for the array stored in the TaskList object.

DailyObjectives

DailyObjectives acts similarly to TaskList, in that it holds an array of Objectives (like Tasks) and has methods to push them in place after fetching a User's TaskList by chronological order instead of by queue score. Tasks are organized into Days (by the Date that the Task is due) stored in a dayList object. Each Day in the dayList contains references to Objectives which store the name of the Task, the time needed to work, and the index reference for the TaskList derived from the User object, which is copied into DailyObjectives as objTaskList. By calling createDayList() and createObjectives() in its constructor after being fed a TaskList, DailyObjectives filters out what Tasks need to be worked on today and assigns a specific amount of time from the algorithm in createObjectives().

Day

Day is created by and used with the DailyObjectives constructor. It contains the referenced day's Date and a list of the indices of all Tasks referenced by all Objectives due on the same Date. Simply put, each Day contains an array of the same indices referenced by that Day's Objectives.

Objective

Objective acts similarly to Task, in that it stores the name of a corresponding Task and a reference to the index of the Task on the user's TaskList. It also contains a goal for the user of the site to work on that Task for in a day and has its own counter for minutes of work done on that day. It also has its own Boolean to keep track of if the Objective has been completed, even if the Task it references is not finished.

Garden

Garden tracks a User and their garden String, which is a simple String variable that can be digested by the function updateScreen(garden) to create the User's Garden structure. It also tracks the selected store item and selected tile index when they are clicked on by the user. When a tile is selected, it begins a userPurchase() which uses the selected item to purchase and the selected tile. It validates that the User has enough xp for the purchase, updates the Garden's garden String and the User's xp, and updates the screen.

DateManipulation

DateManipulation is a static object used for calculations and formatting. It contains calculations to find the number of days left until a Task is due based on a Date object and the current time. It also formats time from the number of minutes (used to display the amount of time a Task has been worked on) as well as from the number of seconds (used to display the timer) to use on the Dashboard and Total Task Layout pages.

Database Tables

The Firestore database has changed significantly from our initially planned ERD diagrams. Instead, each user is placed in the "users" collection of documents, with each user retaining a document labeled with their unique userId from Firebase. Each user document retains several fields: **email**, **exp**, the **gameboard** string, **lastLogin**, their **major**, their **name**, and their **streak** (for logging in consistently). Each task is stored into the database as a document and is given a unique taskId. The document contains various fields about each task: the **dueDate,** the task's **name**, the task's **priority**, the total amount of time the user estimates to work on the task in **taskMinutes,** and the total amount of time worked on the task in **totalTimeWorked.**

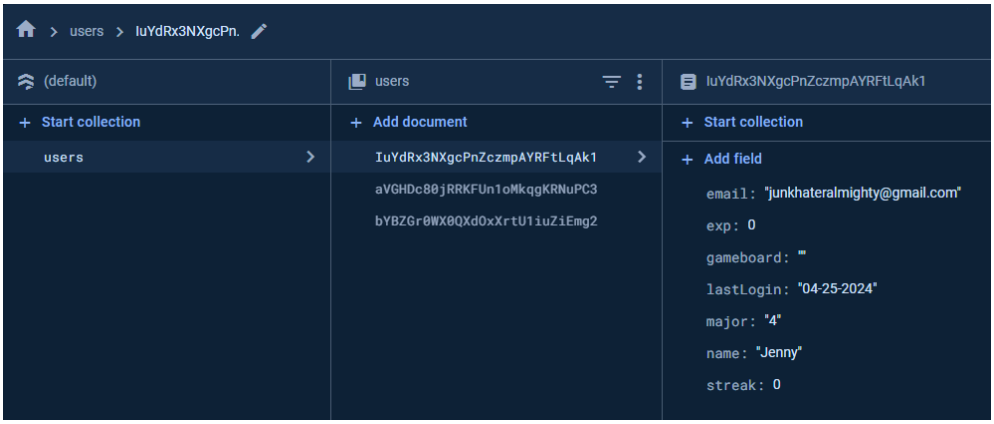

Figure - The User Collection and Documents

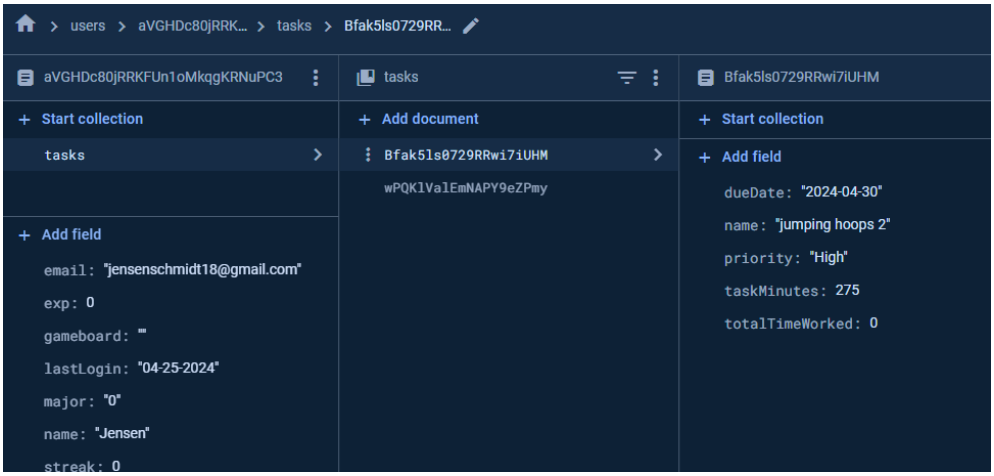

Figure - The Task Collections and Documents

Script Files

The following script files are utilized throughout the web application to provide the current functionalities of ThinkItThru!.

app.js

This JavaScript file is intended for the index.js page, and the authentication/registration needs of ThinkItThru!. It imports many of the authentication methods from the CDN version of Firebase 10.11.0 and imports several of the Firestore database functions from the CDN version of Firebase 10.11.0. The file initializes the **auth** authorization object and the **db** database object from the Firebase methods, while also creating a variety of **const** variables from the necessary HTML elements by their various ID values. **onAuthStateChanged()** works to identify when a user's authentication state changes and is helpful in determining when a user is unknown, has signed up, when a user has verified their email, or when a user has logged into the system. There are many event listeners for the different buttons present throughout the registration/authorization form, each intended to call functions based on their press: **signUpButtonPressed, logOutButtonPressed, loginButtonPressed, needAnAccountButtonPressed, haveAnAccountButtonPressed, resendButtonPressed,**

resetPasswordButtonPressed, loginWithGoogleButtonPressed, updateUserProfileButtonPressed. The buttons dealing with sign up, logout, and login are self-explanatory; **signUpButtonPressed** accepts the auth object, email.value from the input field, and the password.value from the input field to pass to an asynchronous method imported from Firebase, **createUserWithEmailAndPassword()**. Once it passes the information to **userCredential**, it will use that variable and attempt another asynchronous function from firebase, **sendEmailVerification(userCredential.user).** Once firebase receives email confirmation, the function will use **setDoc()** with a reference to the "users" collection in Firestore to establish all the necessary values for a new user in ThinkItThru!. Any errors garnered in the process will be sent to the console and the **formatErrorMessage()** function. The **loginButtonPressed** function uses an asynchronous method imported from Firebase, **signInWithEmailAndPassword()**, that accepts **auth, loginEmail.value, loginPassword.value**, as parameters. These values are then passed to firebase, and if they authenticate, the function will direct the user to "dashboard.html;" otherwise, a specified error and **formatErrorMessage()** are displayed. The **logOutButtonPressed** function attempts an asynchronous function using an imported Firebase method, **signOut(),** accepting the **auth** object as its only parameter. The rest of the values in the function reset the values of the registration/login screen inputs.

The **loginWithGoogleButtonPressed** and **updateUserProfileButtonPressed** functions operate a little differently. With the former function, several imported Firebase methods are used: **GoogleAuthProvider()** and **signInWithPopup().** Through these two methods, the user's **auth** object is passed to the Google Provider for authentication in an asynchronous method; when authenticated, the user is taken to the dashboard. For the **updateUserProfileButtonPressed**, an asynchronous method is called to update the users document within the "users" collection with user-input values.

One of the last interesting functions of app.js involves the **resetPasswordButtonPressed** function. It attempts to call an asychronous method imported from Firebase, **sendPasswordResetEmail(),** which takes the **auth** object and an input email as parameters. Depending on whether the email works for this function, the application will report a success and send the email or the application will report an error/failure.

The remaining methods do several smaller tasks: **needAnAccountButtonPressed** and **haveAnAccountButtonPressed** switch between the registration and login screens, **resendButtonPressed** re-calls a **sendEmailVerification()** method from firebase, and the **forgotPasswordButtonPressed** displays the forgotten password screen.

infoScript.js

This JavaScript file contains multiple essentially connected objects referenced by all pages of the site: the User, BasicTask, Task, SubTask, and TaskList. A User will contain a TaskList, which contains an array of Tasks. Each Task contains an array of SubTasks. Both Tasks and SubTasks are children of BasicTask, which defines a super constructor which contains basic information. Each object contains its own methods for fetching and changing data and calculations associated with that data (see Objects section above).

dashboard.js

This JavaScript file contains functions related to buttons pressed by the user of the site to update the User object and the site user's screen. It contains an array of timers and has the function **toggleTimer()**, which updates whether or not the timer display on the page is running. It also contains **deleteTask()** and **completeTask()** which, in our current build, both remove the Task card from the user's screen and remove the correlating Task from the User object.

dashboardHTMLUpdate.js

This JavaScript file is related to the dashboard functionality for the site, and contains functions called by HTML elements to calculate necessary data, update the User object, and display the newly updated state of the User (**displayTasks()**, **displayXP()**). The first lines of the file create and update a User object, then display the XP and Tasks of the User to update the site. It does this by calling **displayTasks()**, which fetches the Objective and Task List of the User object and updates the site HTML to display the Tasks within the Objectives List by dynamically generating a string of HTML.

game.js

The JavaScript file is dedicated to initializing the Garden by creating a default User, calling **updateScreen()** from gardenScript.js, and displaying the User's updated XP with **displayXP()**. This function defined here simply puts the User's updated xp from all purchases into the html of the website to update the site user's screen.

gardenScript.js

This JavaScript file contains the Garden object, which is initialized in **game.js** with the User object for the Mind Your Garden game, as well as a few functions that work with this garden that are called in the **gameboard.html** buttons from the site. The function **updateStoreItem()** is called when a user of the site clicks on an item from the shop at the top of their screen. This updates the Garden's **selectedStoreItem** to remember the character related to the item the user plans on purchasing. To purchase the item and add it to their Garden, the user will click on a tile in the Garden which calls **beginPurchase()**. It uses the item and tile selected to update the Garden's garden String that stores the Garden structure. From there, **updateScreen()** can be called to dynamically update the page's HTML and display the new state of the Garden to the user.

dailyTasks.js

This JavaScript file contains the DailyObjectivesList, Day, and Objective objects which work in tandem to create an array of Daily Objectives that display to the user on the dashboard page. They contain references to the User object and user's TaskList and contain their own storage and logic to reference specific Tasks from the TaskList and create a specific daily goal for the user of the site to accomplish (see Objects section above).

totalTaskLayoutHTMLUpdate.js

This JavaScript file is run upon the loading of the Total Task Layout page of the site. It creates a default user, adds Tasks and SubTasks, and updates the HTML of the website to reflect the User's

current state. It updates the HTML through **displayTasks()**, which dynamically creates HTML Task tiles, and **displayXP()**, which simply updates the "xp" div to reflect the User's xp. This file also contains functions to update the User: such as **changeUserSubTask()**, which updates whether or not a User's Task's SubTask is completed; **addTimeToUserTask()**, which updates the time worked on the Task when a user "clocks out" of working on a Task; **awardUser()**, which adds xp to the User based on the number of minutes worked on a Task when a user "clocks out"; and **removeTaskFromUser()**, which removes a Task from the User's object when a user deletes or completes a Task.

newTask.js

At the beginning of this file, many Firebase methods get imported from Firebase authentication, Firestore, and the basic Firebase application feature. The Firebase configuration setup gets instantiated again in **firebaseConfig.** After which, the Firebase application object is initialized with the database and authentication objects. The **onAuthStateChanged()** checks to ensure the user is verified and currently authenticated, then pulls the new elements from the forms in the HTML page. The method then tries to call an asynchronous method to add a collection in the user document, "tasks," where the data from the forms will be stored. **fetchAllTasks()** is a function leftover from testing, with the developers intending it to iterate through each task document of a user and output the fields to the console.

timeScript.js

This JavaScript file contains the DateManipulation static object, used to calculate days remaining to complete a Task with **daysUntilDue()** as well as formatting time into Strings with **formatTimeFromMinutes()** and **formatTimeFromSeconds()**.

firebase.js

This JavaScript file is intended to be input into any of the HTML files utilizing Firebase functionality (so, eventually every HTML page). It imports **intializeApp()** and **getAnalytics()** from CDN links to the Firebase 10.11.0 versions of Firebase's features. f**irebaseConfig** initializes the configuration for ThinkItThru!'s Firebase project (with all the necessary API keys and references for functionality). After this, the Firebase project is initialized as **app** through the imported Firebase method **initializeApp();** Then, the analytics object is initialized for the application.

How to Deploy or Build the Current Application

Downloading the Code and Setting Up Firebase

Necessary software, prior to using the app:

- Node.js, which contains node package manager (npm)
- Visual Studio Code, with extensions for JavaScript, HTML, CSS, and Live Server by Ritwick Dey (for testing and debugging)

The first step for a user to build this project is to navigate to our **GitHub** repository, lwright249/ThinkItThru. This holds our files for ThinkItThru!'s old build, our current build, **node_modules/json** files (these were automatically generated through the process of building the current version of ThinkItThru!), and all the previous documentation surrounding the development of the project and its use in our college course at APSU, CSCI 4805 (the Computer Science Capstone). The user will want to prepare a place to install this repository, though they only need to install the "CurrentBuild," "node_modules," "package-lock.json," and "package.json" folders and files within a central "ThinkItThru" folder.

Now that the site's files are installed, the user will need to establish the necessary initial Firebase functionalities within the code. Firstly, the user will need to **Create a Firebase** project (outlined in this documentation: https://firebase.google.com/docs/web/setup). The user will need to navigate to https://firebase.google.com and sign into Google with their Google account. Once this is done, the user can click the message "Go to console" at the top right of the Firebase homepage; the "Go to console" text should be blue and clickable. Once in the Firebase console, click **Add project**. If the user is prompted, they need to review the Firebase terms and accept them. Firebase will ask if the user would like to set up analytics for this project—this is not required for the function of this project, however the user will need to delete several imports and object instantiations in the "firebase.js" file later should they decide to not use it.

At this point, the user should be able to see and click **Create project**. During this process, Firebase will provision the resources for this project and the user will be taken to the homepage of the Firebase console. In the center of the Firebase console's project overview page, the user should see a Web icon **(</>)**. After clicking this, the user will be asked for the app's nickname and optional hosting. The user should provide a nickname and check the box regarding hosting, allowing for easier setup later. Once the nickname is given, the user should click **Register app** and skip the instructions for the Firebase SDK installation (this app utilizes the Content Delivery Network—CDN—method for configuration). The user should follow subsequent prompts to install the Firebase CLI tool utilizing the node package manager tool (npm). After this, the user should skip the deployment stage and proceed to the Firebase console homepage.

Installing the SDK and Initializing Firebase

First, the user will need to set up the necessary Firebase functions to work with the app. ThinkItThru utilizes two big mechanisms of Firebase: Firebase Cloud Firestore and Firebase Authentication. To set these utilities up, the user should navigate to the left side of the Firebase console and find the word "Build." Once the user clicks this, they will open a submenu with a variety of Firebase tools to choose from. The user should first click **Firestore Database** and click **Create Database** in the ensuing interface. Following the on-screen instructions and accepting defaults, the user should set up the database security rules in **Test mode** (**however, should the user wish to continue this project, they must input the reading/writing security rules included in the project files**. After creating the blank database, the user will need to go back to the **Build** menu on the side of the console and click **Authentication**. In the UI, the user will need to then click **Get Started**. After the console finishes loading, the user needs to navigate to the **Sign-in method** tab and click **Email/Password** and only click to enable Email/Password in the next pane. Once the user is back at the **Sign-in method** tab, they need to click **Add new provider** and **Google.** In the next screen, the user must click to enable Google sign-in, which will open a menu for the user to input their project's public-facing name and the required support email the user would like to use. After saving this functionality and being brought back to the **Sign-in method** tab, the user is ready to look at the project's configuration data.

Modifying the Code to Incorporate the User's Firebase Project

Navigating to the leftmost menu and up to the **Project Overview,** the user should click the gear icon, then click **Project settings.** Once in this menu, in the **General** tab, the user should scroll all the way to **Your Apps** and continue scrolling down until the **SDK setup and configuration** section. The user should click the **CDN** radial option and scroll to view the code that Firebase presents. The user should copy the **const firebaseConfig = {...}** code so that they can update the firebase configuration in the following step.

The user should open **Visual Studio Code** and navigate to the project's folder. In the **CurrentBuild** folder, the user should navigate to the **Scripts** folder, then open **Firebase.js.** After opening the file, the user should replace the firebaseConfig object with the code copied from earlier, allowing the project to user the user's database and authentication. After doing this, the user should save the file and navigate to the **public** folder, to the **Scripts** folder located in it, and do the same modification to the **firebase.js** file in that folder. The first **firebase.js** file was for the current build, and the second **firebase.js** file was for the public-facing deployment of the app. The user then must follow these same steps to modify the **firebaseConfig** objects in the **newTask.js** files from both the **Scripts** folder and the **public** folder.

Deploying the App

At the Firebase console homepage, the user should navigate to the left side of the screen and open the navigation menu. From here, the user should click **Build,** then click **Hosting**. In the UI page that is opened, the user should click **Get Started.** The prompts will ask the user to install the **Firebase CLI** tool, however this was done earlier in these instructions. After clicking **Next**, the user should follow the on-screen instructions of navigating to a terminal and using the command line to log in to Firebase. After logging in, the user will need to use the **cd**

command to navigate to the **CurrentBuild** directory. Once the user has done this, they can use the **firebase init** command to begin the deployment preparation process.

The user should choose to initialize the correct project that is intended for the app, then select **Firestore** and **Firebase Hosting and (optionally) set up GitHub Action deploys** as the features the app will use. In the following menus, the user should keep the default/suggested filenames. When asked if they should overwrite the files, the user should agree. At the end of this, the user should agree to use **public** as the public directory, and should disagree (N) to configure this project as a single-page app. They should also disagree (N) to set up automatic builds and deploys within GitHub in the following message. When asked to overwrite the **public/404.html** file, the user should agree (Y). When asked to overwrite the **public/index.html,** the user should disagree (N). Finally, the user should input this code into the terminal (and from the **CurrentBuild** directory): **firebase deploy --only hosting**. Once the terminal finishes, it will provide the user with a URL that can also be found in the Hosting section of the Firebase console when viewing the app the user specified. The app should be hosted at this point; however, a known issue with network connectivity has occurred when attempting to view the site from a protected campus internet connection.

Known Bugs

ThinkItThru! is still currently in development, with some of its key features requiring more work to become fully operational. However, there are some known bugs outside of the functions in-progress, especially with the introduction of the database and hosting deployment:

- When a user has forgotten their password, an email will be sent no matter what email is input.

This issue appears to be simple and was discovered during a demonstration. It will require a thorough look at the “app.js” and “index.html” files. The developers just need to look through the code and identify where the app bypasses its logic to check the email’s legitimacy before sending the password recovery email.

- The number displaying a user’s XP changes position between “gameboard.html” and other pages of the site.
- Navigation to “gameboard.html” and “dashboard.html” is missing from some navbar sections across the site.

These issues are inherently HTML-related, possibly including issues with JavaScript functionality. ThinkItThru!’s developers would need to look at the navbar sections of each page to confirm uniformity and positioning across the site. Should this approach fail, the developers will investigate the JavaScript code to find any issues with HTML manipulation.

- Depending on the user logging into the application, the deployed site will revert to the dashboard’s original JavaScript file.

Compared to the previous issues, this bug is more complicated. However, the first step towards a solution would be to take the old JavaScript file completely out of the public/CurrentBuild folders, keeping only the updated JavaScript for the program to use. Should

this issue persist, the developers would need to look at the new JavaScript file to understand why the application is refusing to load for different users.

- The current security rules (outside the specified testing rules) do not allow for users to read/write to their own “Tasks” collections

This will require research into the specific syntax that Firebase implements in its Firestore system. There is a plethora of key words to work around this issue, and it would only take some time to research and implement a fix in the logic.

- ThinkItThru!’s Firestore functionality currently does not have an efficient way to iterate through a user’s “Tasks” collections.
- ThinkItThru!’s JavaScript takes some information that is not present within the documents of the “Tasks” collection.

These issues will take some time to research and test using different JavaScript and Firestore functions. Once the correct methods present themselves, these issues will be easy to remedy as the developers modify the JavaScript and “newTasks.html” to accept/use a standardized format for Tasks.

- The dashboard page Task indices may cause the wrong Task to be updated with input from the user for some out-of-order TaskLists

This issue relates to the Objective’s List array indices not quite matching with the TaskList’s indices, which causes an error in the JavaScript and updates the wrong Task Object on the dashboard page. This will require a careful examination of JavaScript files and an update translating and correcting the indices chosen to update.