SaveDforest Documentation

Introduction

This documentation expands upon my master's thesis (more concretely, upon Appendix D – Technical Manual), focusing on detailing the technical structure and source code of the "*saveDforest*" (v1) game and underlying web-app.
I was only able to cover the front-end and deployment process in the thesis - I couldn't write a full technical breakdown; this document thus aims to fill that gap.

The web-app can be accessed here.

A public repository containing a copy of the front-end project can be found here, and a copy of the back-end project can be found here.

A public repository of the Unity game project is in the works - there's still some stuff that I need to clean up. When that's done, I hope to be able to finish writing its documentation.

As this document focuses mainly on source code, it assumes a certain degree of familiarity with the app; that being said, I've included an overview of its purpose, structure, and data flow below. For a more in-depth explanation of its architecture, conceptual design, development context, evaluation, and other "high-level" related aspects, please refer to my thesis.

A (brief) overview

"saveDforest" is a single-player serious game that promotes environmentally sustainable behaviors that benefit the Borneo rainforest. To achieve this goal, the game relies on fostering empathy towards people and animals directly affected by Borneo's deforestation, as empathy drives prosocial behaviors.
The game contains:

- **Scenarios** that present topics - such as the impact of Borneo's palm oil industry on the forests - through short narratives. The player must make decisions at the end of most scenarios, which can take them to more information about the current topic (mainly through links to external websites and resources), to information unrelated to the topic, to a quiz on the current topic, and/or to a different scenario. There are 10 scenarios, but depending on the player's decisions, some of them may not be shown.
- **Quizzes** that can be answered after completing most scenarios; they're based not only on the scenarios' narratives, but also on information that is available through the external links provided via scenario decisions. There are 8 quizzes; despite most quizzes being optional, the final quiz is mandatory.

cute orangutan artwork done by Gonçalo Sales

- **Points -** awarded through scenario decisions and correct quiz answers - and **badges** - most of which are earned by correctly answering all questions in a quiz. The final badge, however, is only obtained if the player finishes the final mandatory quiz with 5 or more correct answers, and their overall score after the final quiz is equal to or higher than 87 points (roughly around 85% of the highest attainable overall score). The player's final score is stored upon finishing their first playthrough and is displayed on the main-menu as a high score; they are then able to beat their own high score by playing the game again. Additionally, any badges earned during a playthrough persist, allowing players to complete their badge collection through subsequent playthroughs, if they have not yet done so.
- **IRI** (Davis, 1980), **AES** (Paul, 2000) and **SAM** (Bradley & Lang, 1994) **self-report questionnaires** that are answered during players' first playthroughs to measure empathy changes and emotional responses. Both IRI and AES are answered twice: immediately before playing through scenario 1, and after completing scenario 10. SAMs are completed after scenarios 1 through 9 - and if a quiz follows, before said quiz. Therefore, if a player's decisions led them to completing all 10 existing scenarios during their first playthrough, they will have answered 9 SAM questionnaires.

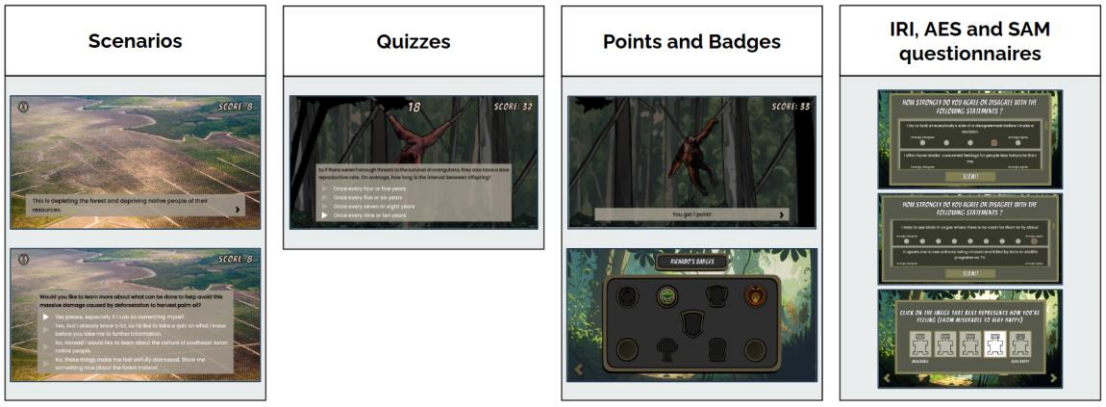


Figure A – The core components of the saveDforest game.

The game is playable via browser as it is supported by an underlying **web-app**:

- Its **front-end** consists of an **Angular single-page app** that embeds the game in Unity WebGL format; it also contains specific components that provide additional information and resources related to the game's topics and the project itself.
- Its **back-end** consists of an **Express.js application server running on top of Node.js**, and a **MongoDB database.**

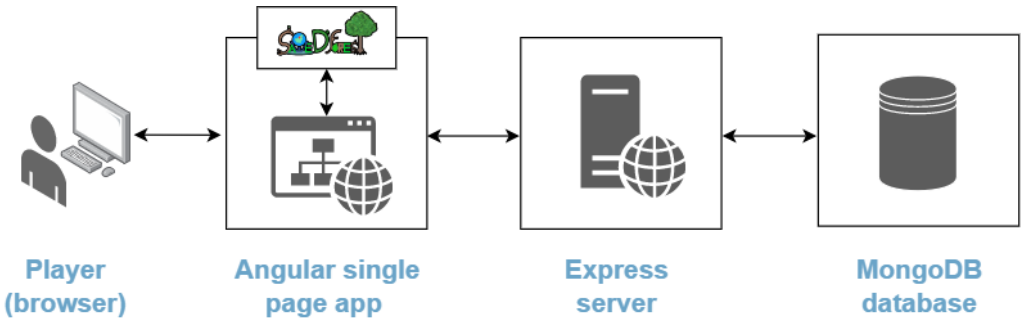


Figure B – An overview of the web-app's architecture.

Generally, whenever the player does an in-game action that involves creating/reading/updating/deleting data from the MongoDB database, the following occurs:

1. The Unity WebGL game calls a specific method in the Angular app (with data potentially being passed as arguments, depending on the in-game action).
2. The Angular app sends an HTTP request to a corresponding API endpoint on the Express.js server (potentially including data in the request body, depending on the in-game action);
3. After validating the request, the Express server delegates an appropriate operation to the MongoDB database;
4. Once the operation completes (whether successful or not), the Express server sends an HTTP response - possibly containing data, depending on the in-game action - back to the Angular app;
5. Upon receiving the response, the Angular app will call a specific method in the game (passing any received data as arguments), so it can proceed accordingly.

In addition to supporting players' game progress and measurements' storage, the web-app also enables authentication-related features: signing up (which involves inputting not only an email and password, but also a username and some demographic information), logging in, logging out, and password resetting.
Adhering to RESTful principles, the server is stateless, meaning no client context is stored on the server between HTTP requests. Instead, a session cookie is used to "maintain" session state.

Some of the players' game data (including username, current score, current scenario, badge indexes, and a number flag signaling if they've already finished their first playthrough) is also stored in the client browser – more concretely, in the IndexedDB - which persists until the player logs out or the session cookie expires. The game can directly access this data, thereby reducing HTTP requests.

The login feature's data flow serves as a good example to briefly illustrate all the web app concepts discussed above:

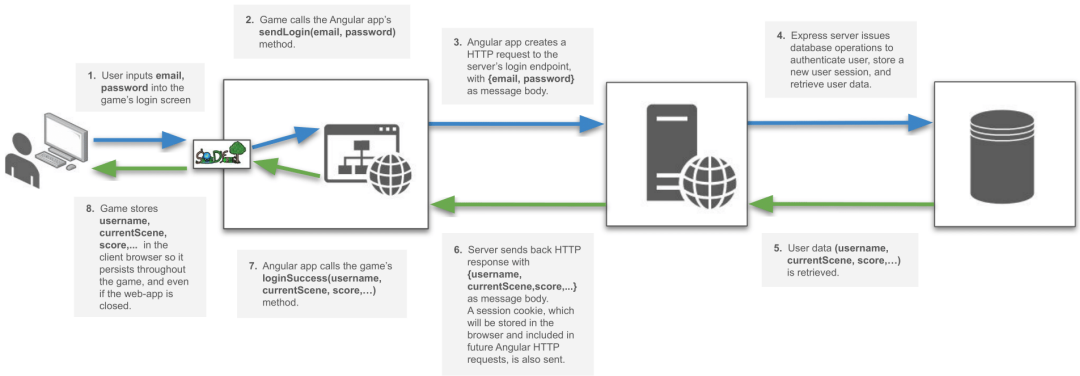


Figure C - An overview of the data flow between the client browser, the Unity WebGL game, the Angular app, the Express server, and the MongoDB database during a successful login.

Index

A - Front-end

A1 - The Angular application

The saveDforest game is embedded within a front-end web app built with Angular. It also serves as an intermediary between the game and the back-end.
Angular is a single-page web app (SPA) framework that revolves around a component-based architecture. Unlike multi-page web apps that navigate between separate pages and require full page refreshes when there is a data update due to user actions, SPAs operate on a single page, dynamically fetching and updating the page's content as needed.
Angular relies on a combination of HTML, CSS, and TypeScript, which extends JavaScript with static type definitions, enabling type checking during compilation and allowing the use of variables and fields with specific data types, akin to languages like Java or C#.
Official Angular documentation is available at https://v13.angular.io/docs. Setup instructions can be found at https://v13.angular.io/guide/setup-local.
A "language service", which provides autocompletion and error checking in code editors, among a few other features, can be found at https://angular.dev/tools/language-service.
If using Visual Studio Code, the free "Rename Angular Component" extension is useful for automatically updating imports and dependencies when renaming files.
The current project was built using Angular CLI v13.3.10 within Visual Studio Code v1.73.1, while being supported by Node.js v16.16.0; that being said, **the application can be run and built without needing to have this particular Angular CLI version globally installed** (or any, for that matter).

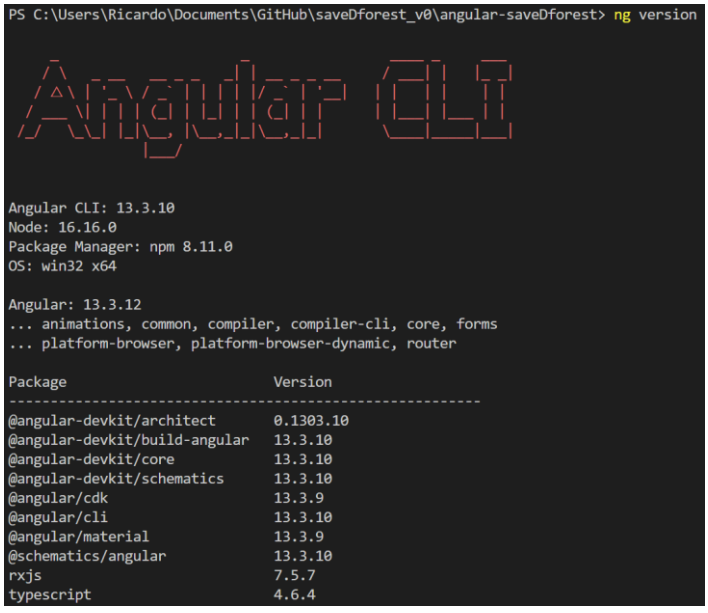


Figure 1 - The Angular CLI version used.

The app can be tested on a local development server (after some modifications) by running the ***npm start*** command, which executes ***ng serve --open***, as it is defined as a script in the **package.json** metadata file. The default URL for the local development server is "http://localhost:4200/".

* For testing the Unity build on the Angular project locally, a non-deployment Unity build must be used; otherwise, the game will be stuck on the loading screen. A local testing build is thus provided in the public repo, named "v29_for_local_testing".
**if Angular CLI v13.3.10 is installed globally, the *ng serve* command can be run directly; otherwise, *npm start* must be used.

The app was tested locally on a couple Android smartphones by port forwarding the URL mentioned above through USB, using Chrome DevTools. Documentation for setting up port forwarding to Android devices through Chrome can be found at https://developer.chrome.com/docs/devtools/remote-debugging/local-server#usb-port-forwarding.

Angular project components

1.1 The root directory

The project's root directory comprises the following directories and files:

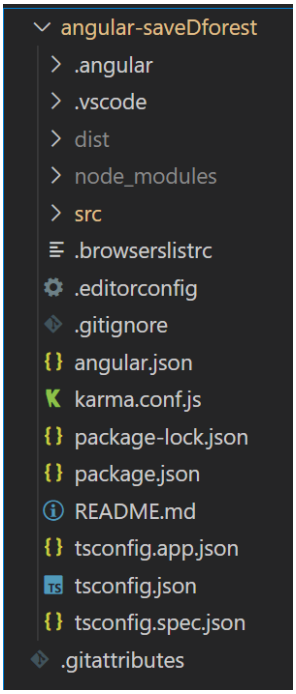


Figure 2 - The root directory of the Angular project.

Important directories and files include:

dist - this is the directory where the resulting angular build is stored in. This directory is not pushed to the project's repository on github.

node_modules - this directory includes installed libraries on which the project depends on. It is not pushed to the project's repo on github due to its size; therefore, when pulling the project from the repository, the command ***npm install*** must be run so all dependencies are installed.

src - this directory contains all assets and source code.

angular.json - workspace configurations are set on this file. A comprehensive guide on this file's structure and content can be found at https://angular.dev/reference/configs/workspace-config.

1.2 The src directory

The **src** directory contains the following:

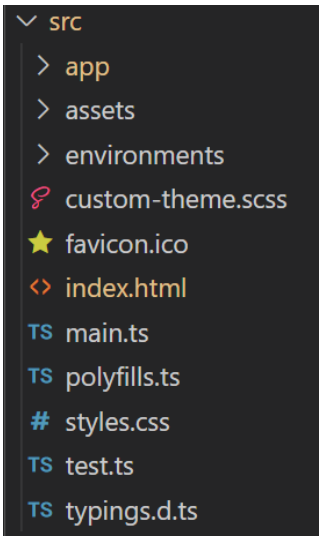


Figure 3 - The src directory of the Angular project.

app - this is where the source code is located.

assets - this is where assets used in the app are located. It includes images, the Yarn Spinner license as a .MD text file, and the generated Unity game build.

favicon.ico - this is the app's tab icon. It's a .PNG image that has been redimensioned and converted to .ICO.

Index.html - this will be the entry point of the app once the Angular build is created. The **<head>** section includes:

- The "preloading" of certain assets;
- Linking to Bootstrap v5.3.0's CSS. Bootstrap is used extensively throughout the project for easily configuring the app's visual components;
- Linking to Google Fonts, so a specific "Roboto" font can be used;

The **<body>** section includes a custom-made script that checks if the user's browser is Bootstrap compatible. If it is, Bootstrap's javascript bundle is linked to normally; if not, the user is advised to update their browser. Without this script, if the user happened to access the app from a non-supported browser, just a blank screen would be shown, with no additional message being provided.

typings.d.ts - this file is used to declare functions/variables from external libraries that do not have native TypeScript support. For this project, it is used to declare Unity WebGL's "createUnityInstance" function, which is responsible for loading the Unity build and creating an instance of the game.

1.3 The app directory

This directory comprises the bulk of the app's logic.

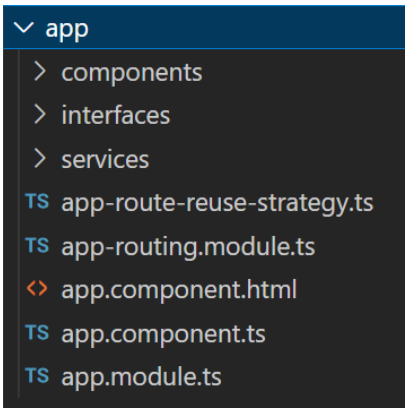


Figure 4 - The app directory of the Angular project.

components - components, which define the different app views, are stored in this directory. For further information on components and their structure, refer to 1.3.1 The component directory.

interfaces - Angular interfaces, which define custom data types, are stored in this directory. For further information on interfaces, refer to 1.3.2 The interface directory.

services - services, classes that handle specific tasks, are stored in this directory. For further information on services, refer to 1.3.3 The service directory.

app-routing.module.ts - module classes organize related components, directives, and services into cohesive units, making the app more modular and manageable. This module in particular specifies "routes" that map URL paths to components, enabling navigation within the app and deep-linking. 5 routes are configured:

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'howToPlay', component: HowToPlayComponent },
  { path: 'otherResources', component: OtherResourcesComponent },
  { path: 'resetPwdToken/:token/:id', component: ResetPwdComponent}
];
```

Figure 5 - The routes defined in app-routing.module.ts.

app-route-reuse-strategy.ts - A native router service handles route-switching; by default, navigating between routes requires components to be constantly recreated and destroyed (e.g., when navigating from route A to route B, route A's associated component is destroyed and route B is created. If the user returns to route A, route B's component is destroyed and route A's component is recreated, and so and so on), which is incompatible with the embedded Unity game, since it would cause the game instance to be destroyed every time the user would leave the root ('') route (associated with the home component that embeds the game), leading to significant delays and possible loss of state. This router behavior was thus modified by creating a custom implementation of the native **RouteReuseStrategy** abstract class through the **AppRouteReuseStrategy** class, found in this file. In this implementation, based upon the one

available at https://www.angulararchitects.io/en/blog/sticky-routes-in-angular-2-3/, routes are stored in a map when they're deactivated and retrieved again as they are requested.

Even though I opted for reusing all routes, this class can be easily configured so only the root ('') route is reused by modifying the **shouldDetach** method, which defines whether or not a currently activated route should be stored:

```
shouldDetach(route: ActivatedRouteSnapshot): boolean {
    return true;
}
```

```
shouldDetach(route: ActivatedRouteSnapshot): boolean {
    return route.routeConfig!.path == ''
}
```

Figure 6 - On the left, all routes being reused. On the right, only the root route is reused.

I tested the performance of both solutions by creating two builds: one with all routes being reused, and another one only reusing the root route. Both builds were served through Render and accessed using Google Chrome on a laptop, first navigating between all routes a couple times in succession, then letting the app go idle on the 'about' route, before navigating back to the 'otherResources' route. When comparing both builds' performance with Chrome DevTools and Task Manager, it was evident that:

- Navigation between routes that had already been visited before was a few milliseconds faster in the build reusing all routes;
- Memory usage was roughly the same for both builds;
- After letting the app go idle and navigating back to the 'otherResources' route, the build only reusing the root route had to request images from the hosting server once again, requiring more data to be transferred through the network. This fact weighed heavily on decision to reuse all routes, since free bandwidth usage on Render is limited.

Certain elements in the **about** and **home** components need to be explicitly resized whenever there is a window resize event; these events are thus handled by event listeners. However, while these component's routes are deactivated, their elements are not correctly resized since they are not rendered on screen. To optimize performance and prevent memory leaks, the event listeners are disabled when these routes are deactivated, and re-enabled when they are reactivated. Likewise, since these routes are not recreated upon returning to them, and because components lack a native lifecycle hook for route reactivation, the resizing methods must be explicitly called once they're reactivated to account for any resizing event that occurred while they were deactivated.

The **store** method stores routes to be reused and disables the aforementioned components' resize event listeners:

```
store(route: ActivatedRouteSnapshot, handle: DetachedRouteHandle): void {
    this.handlers[route.routeConfig!.path!] = handle;
    if(handle!=null){
      const routePath = route.routeConfig!.path;
      if(routePath == 'about' || routePath == ''){
        (this.handlers[routePath] as any).componentRef?.instance.disableResizeEventListener();
      }
    }
}
```

Figure 7 - The store method.

This method is invoked twice when navigating between two previously stored routes: once for the route being deactivated, and once for the route being reactivated. For the route being reactivated, the handle parameter will be null; for the route being deactivated, it will not. If the route being deactivated corresponds to the 'about' or root route, its component's "disableResizeEventListener" method is called to disable the resize event listener.

The **retrieve** method is responsible for retrieving stored routes when requested, and also invokes the aforementioned components' **"onRouteReactivation"** method, triggering a resize and re-enabling the resize event listener:

```
retrieve(route: ActivatedRouteSnapshot): DetachedRouteHandle|null {
    if (!route.routeConfig) return null;
    var routePath = route.routeConfig!.path;
    if(routePath == 'about' || routePath == ''){
      (this.handlers[routePath] as any).componentRef?.instance.onRouteReactivation()
    }
    return this.handlers[route.routeConfig.path!];
}
```

Figure 8 - The retrieve method.

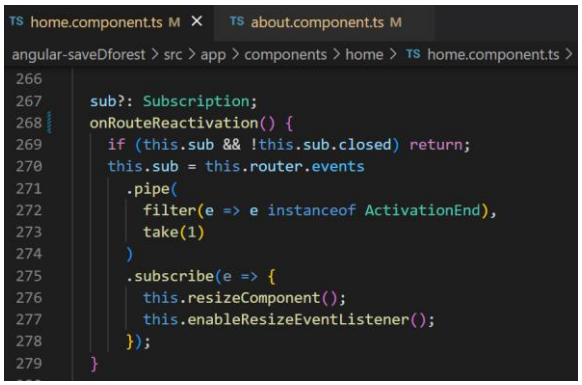


Figure 9 - The onRouteReactivation method, present in both the home component and the about.

The retrieve method can be invoked several times per reactivated route; to avoid invoking "onRouteReactivation" multiple times, I opted for a similar approach to one recommended by an Angular team developer, at https://github.com/angular/angular/issues/43251; on "onRouteReactivation", the aforementioned components subscribe to the router service's events (by subscribing to the **router.events** observable which emits them). The **"ActivationEnd"** event, in particular, is emitted when the router has finished (re)activating a route. The **"take(1)"** operator is used to ensure that only the first "ActivationEnd" event emitted is acted upon.

***The current implementation of "onRouteReactivation" makes it so that there is a component resize every time the routes are reactivated, even if there hasn't been any uncaught resize event.**

app.component.html - this file defines a global structure template for the app. It currently looks like this:

```
<app-navbar></app-navbar>
<router-outlet></router-outlet>
<app-footer></app-footer>
```

Figure 10 - The app.component.html.

The **<app-navbar>** element refers to the navbar component. **<router-outlet>** is a router directive, referring to whichever route is currently active. **<app-footer>** refers to the footer component.

app.component.ts - this is the app's root component class. It acts as a linker for the global structure template.

app.module.ts - this is the app's root module. It defines components, imports and some service providers used. Most definitions are automatically updated when using Angular CLI, but the aforementioned AppRouteReuseStrategy had to be explicitly declared as a provider of a RouteReuseStrategy to replace the router service's default behavior:

```
providers: [{provide: RouteReuseStrategy, useClass: AppRouteReuseStrategy}],
```

Figure 11 - The AppRouteReuseStrategy being declared as a provider of RouteReuseStrategy.

1.3.1 The component directory

A component defines an app view and is specified through three main parts:

- a **.ts** class with a **@Component** decorator, that defines the component's behavior and logic;
- a **.html** (a.k.a **template**), that defines the component's structure and layout;
- a **.css** style sheet, which defines the visual appearance of the component's template.

Besides identifying a **.ts** class as a "component class", the @Component decorator also acts as metadata that links said class with the corresponding template and style sheet.

Specific Angular directives (such as **ngIf** and **ngClass**) and data binding markups can be used in template files to dynamically modify HTML elements based on the logic defined in the component classes. Likewise, HTML elements and user-made changes (e.g., filling in a login form) can be accessed and propagated to component classes through data binding.

Component classes can implement several lifecycle hooks to intercept components' lifecycle events, such as **ngOnInit,** which is triggered when a component is initialized.
Components can access service instances by having them injected into the class constructor.

A new component can be created by executing ***npm run generate-component <component_name>*,** which invokes the corresponding Angular CLI command, as defined in **package.json**. By default, when generating a component, a spec.ts file is created for unit testing. Since unit testing is not implemented, the **--skip-tests** flag is set to skip their creation.

To create a new route by associating a new component to a URL path, add it to the **app-routing-module.ts**'s **routes** in the app-directory. To add the new route to the navbar, access the **navbar.component.html**, copy a **<li class="nav-item">** , paste it below the last one, and modify the **id** and **router-link** properties.

Besides the root component in the src directory, there are currently 8 components in their own dedicated directories:

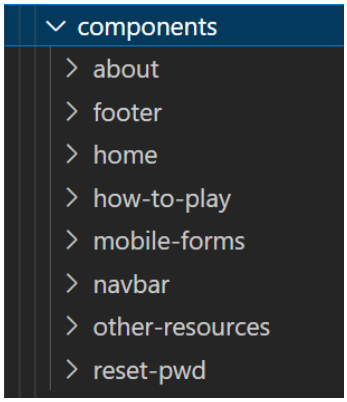


Figure 12 - The component directory.

about - this component defines a view that contains information about the project itself. It includes an image with text overlaid on top of it that needs to be programmatically resized when the browser window is resized.
The overlay itself and the image are referenced by the component class (via the @ViewChild property decorator), being scaled whenever a resize event is caught by a listener. This listener is enabled/disabled based on the component's associated route, as explained in the **app-route-reuse-strategy.ts** of the 1.3 The app directory.
The overlay text, on the other hand, is declared as a variable in the template, and is resized through .css.

footer - this component defines the app's footer, which contains the project's associated faculties and institutions logos. It is fixed at the bottom of the screen through a Bootstrap class. On mobile devices, the footer itself is hidden in landscape mode on the root ('') route through the **ngIf directive**, so the game canvas can expand in height. A non-fixed div, about half the height of the navbar, is instead set and shown at the bottom of the game canvas by the **home** component. Alternatively, the footer with the logos could be modified to serve this purpose as well (by making it non-fixed), but since some logos are trimmed down on some devices in landscape mode, this option makes the route look more aesthetically pleasing.

```
<div *ngIf="!(isLandscapeMode && router.url === '/' )" class="container-fluid fixed-bottom">
```

Figure 13 - Hiding the footer on the root route when in landscape mode.

The **ngClass** directive is used to change the footer's background color depending on the current router route, or not show it at all in the password reset route; this directive dynamically appends a class named after the current route to the div, which is accessed through css.

```
<div [ngClass]="{'home': router.url === '/',
'about': router.url === '/about',
'howToPlay': router.url === '/howToPlay',
'otherResources': router.url === '/otherResources',
'resetPwd': router.url.includes('resetPwdToken') }" class="logo-container" >
```

Figure 14 - Changing the color of the footer, depending on the route.

*Several .css breakpoints are used so that all logos retain their proportions whilst keeping a similar scale between them, no matter the device scale, but there's still a lot of room for improvement as some logos are trimmed down while in landscape mode on some mobile devices.

home - this component embeds the Unity game. It is associated with the root ('') route.
To embed the game into the component, the code in the index.html that is generated when a Unity game build is created was modified, following a similar approach to the one described at https://blog.devgenius.io/deploying-unity-2020-3-2-webgl-project-using-angular-11-312cb33e6fbe

The game build is loaded and instanced through Unity's **createUnityInstance** function, which is invoked as this component is initialized. A "config" object with various settings - such as the location and name of the files to be loaded - is passed to the function. Additionally, as this component is initialized, the **navbar service** is invoked to toggle its boolean BehaviorSubject property to false, a change subscribed to by the **navbar** component, which in turn disables the navbar items, thus preventing route navigating. This step is crucial because if the user navigates to another route while the game build is loading, it will crash and stop loading.

While createUnityInstance is running, a custom loading bar is displayed on top of the HTML canvas that contains the game. Unity builds are exported with a template loading bar that reflects a “progress” value. However, this value does not accurately represent the actual progress; when a user accesses the app for the first time, the loading bar will gradually go up as expected until it reaches 90%. This value indicates that all game files have been downloaded. Afterwards, until the game is effectively started, it will remain at 90%, as if it is stuck. This would not be so bad if such behavior remained consistent; the problem is that on subsequent accesses, since the build files are cached by the browser, the progress bar initializes at 90%, and doesn’t move until the game starts, which can take several seconds. To avoid this misleading behavior, a custom loading view is shown instead, featuring an animated .GIF and messages that fade in and out through a CSS animation, and no mention of the progress’ state.

After the game has finished loading, if the app is being accessed through a mobile device, two special event listeners are set for **touchmove** and **mousemove** events. This is done so the user can scroll in landscape mode to adjust the canvas to their mobile device’s screen, since by default the HTML unity container, which contains the game canvas, will consume all touch inputs (including scroll events). The touchmove listener makes it so that if the user does a scroll motion -through a touchmove event- , the HTML game container stops consuming events, and afterwards, when a tap happens, triggering a mousemove event, it starts consuming them again. There is however a situation in which scrolling must be fully consumed by the container: when the user selects its nationality from a dropdown list upon signup. For this reason, when the user interacts with the nationality dropdown, opening it, the game issues a call to disable the aforementioned touchmove event listener, and once the dropdown is closed (being destroyed from the scene), the game issues another call to enable it. There are other scrolling elements in the game, including other dropdowns, but the nationality dropdown is the only one that really requires this specific approach.

Figure 15 - Preventing or allowing the game container to consume scrolling events.

In addition to the aforementioned event listeners, there are several others that handle function calls invoked by the game. These functions include:

- **isMobile**, which is called when the game's TitleScreenScene is loaded. If the app is running on a mobile device, an instruction is sent to the game via SendMessage for the MobileController game object to invoke the ConfigureForMobile method of its attached script. Additionally, an event listener is added to trigger the mobile-forms service's promptForm method whenever a game input field is interacted with;
- **openLink**, called by multiple game scenes, when the user chooses a dialog option that allows them to get more information. It attempts to open a new browser tab, but for scenes where multiple links are opened, it can fail if the user's browser is configured to block popups; if this happens, SendMessage is once again used to show a warning canvas to the user, prompting them to disable popup blocking. This will only happen once per app load; if the user chooses to continue playing without disabling popup blocking, the warning canvas will not be shown again until the app is restarted;
- Functions that invoke service methods responsible for handling HTTP requests and responses to and from the back-end (ex: **sendSignup**, **sendLogin**, etc.).

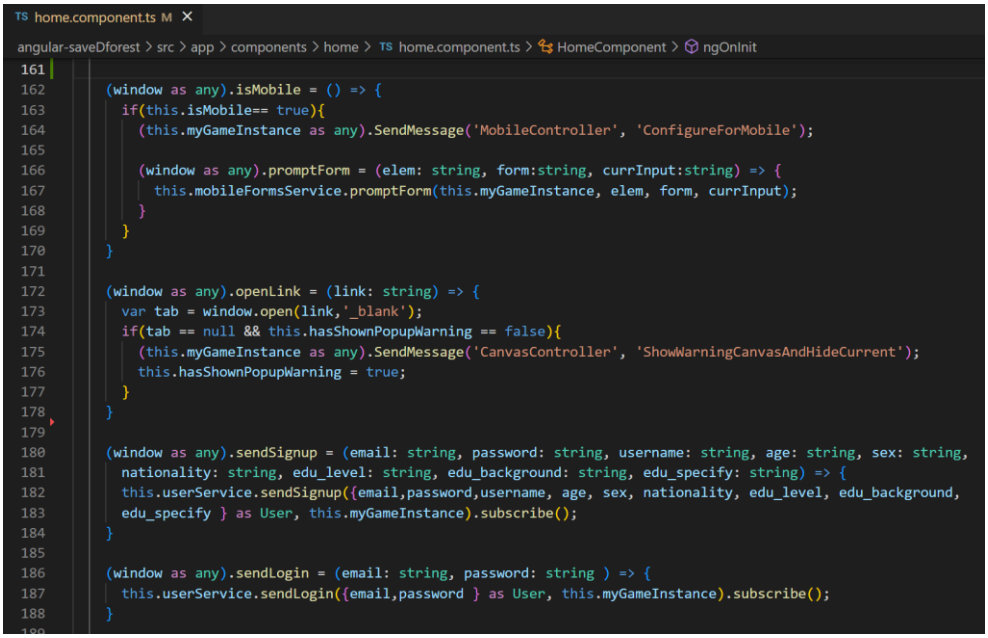


Figure 16 - Event listeners that handle calls issued by the game.

The HTML canvas element which contains the game is also resized through event listeners so the game's aspect ratio remains constant.
The game is originally configured with a resolution of 1200x600, giving it a 2:1 aspect ratio. This aspect ratio must be kept so all game elements are scaled and positioned correctly - for instance, the loading screen canvas's "easter egg" tree is placed in front of a static background image in a specific position so it appears to be a "part" of the background image; if the aspect ratio changes, then the tree's position and/or scale in relation to the background image will also change, potentially breaking the aforementioned illusion. Moreover, changes in aspect ratio might cause game images to become overtly distorted.

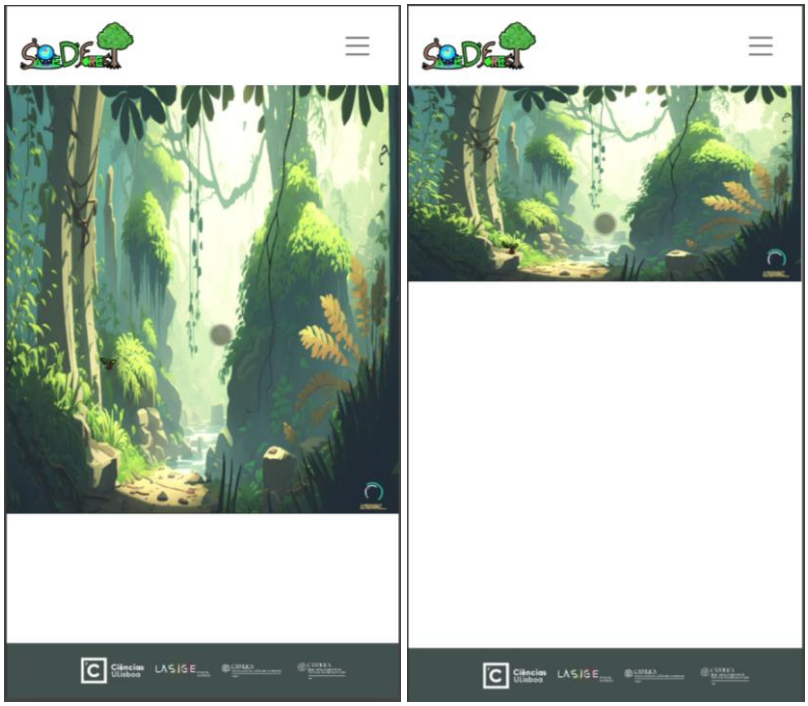


Figure 17 - On the left, the game canvas is set to fit to the width of the device, with the aspect ratio of 2:1 not being preserved. On the right the 2:1 aspect ratio is enforced.

These problems could be solved at the expense of increased complexity and performance; take the easter egg tree for example: its new position/scale would need to be set according to the canvas's current width/height, which is only known by Angular, not the Unity game build itself. In order to simplify things, my approach focuses on keeping the aspect ratio constant, as mentioned before. For this, the canvas is adjusted through specific resizing methods on initialization and whenever there is a screen resizing event.
These methods depend on the type of device the app is being accessed by.

If the device is a desktop or equivalent, the Window interface's resize event is listened to. It triggers the **adjustCanvasDesktop** method, which adjusts the canvas size based on whether its current width exceeds the available browser window width minus a margin of 100 pixels. If it does, it resizes the canvas to fit within the window width, with the height of the canvas being constrained to half of its width. If it doesn't, it calculates a new size based on the window height and the footer height.

```
adjustCanvasDesktop() {

  if(this.canvasRef.offsetWidth>window.innerWidth -100){
    this.canvasRef.style.width= window.innerWidth - 150 + "px";
    this.canvasRef.style.height = (window.innerWidth - 150)/2 + "px";
  }else{
    var footerHeight = window.innerHeight;
    const pixelHeight = (footerHeight * 8) / 100;
    var heightFinal = window.innerHeight-pixelHeight- 150;
    this.canvasRef.style.width= 2* heightFinal + "px";
    this.canvasRef.style.height = window.innerHeight-pixelHeight- 150 + "px";
  }
}
```

Figure 18 - The method that adjusts the game canvas, if accessing via a desktop or similar.

If running on a mobile device, the VisualViewport interface is relied on instead, since its resize event triggers when the browser's address bar is expanded or minimized, which the Window's resize event does not. It will trigger the **adjustCanvasMobile** method, which adjust the canvas based on the current orientation of the device after the resize is triggered.

If in portrait mode, the canvas is set to fill the width of the browser window, with its height being half of the window's width. This orientation is almost never ideal; the canvas ends up being too small and much of the screen's space is wasted. For this reason, users are advised to rotate their devices if in portrait mode while the CreateUnityFunction is running.

On the other hand, if the device is in landscape mode:

- If half of the window's width is equal to or smaller than its height, the game canvas is adjusted based on a width constraint: it can fill the window's width completely, with its height either not filling it (if the height is small) or fitting the window perfectly;
- If the width of the window divided by 2 is greater than its height, the game canvas needs to be adjusted based on height: let the canvas fill the window's height, and set the width to be twice the height value. In this case, since the window's width is not totally filled, the template's root div is set with a black background color, to create the illusion that there's two vertical black bars serving as padding, akin to the pillarboxing effect.

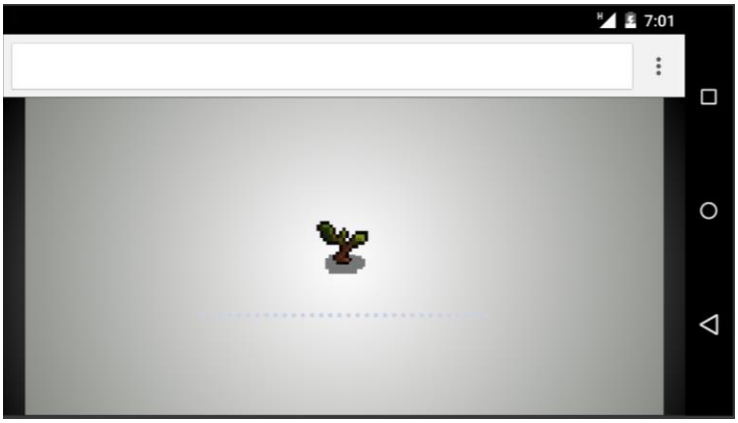


Figure 19 - The pillarboxing effect in landscape mode, on a device wherein the game canvas is adjusted based on height.

```
case "landscape-primary":
case "landscape-secondary":

  this.isPortrait= false;
  this.rootDivRef!.style.backgroundImage = "radial-gradient(circle, rgb(255 255 255 / 94%), rgb(0, 0, 0))";
  var LANDSCAPE_WIDTH;
  var LANDSCAPE_HEIGHT;

  if(window.innerWidth/2 <= window.innerHeight){
    LANDSCAPE_WIDTH = window.innerWidth;
    LANDSCAPE_HEIGHT = ((LANDSCAPE_WIDTH)/2);
  }else{
    LANDSCAPE_HEIGHT = window.innerHeight;
    LANDSCAPE_WIDTH = LANDSCAPE_HEIGHT * 2;
  }

  this.canvasRef.style.width = `${LANDSCAPE_WIDTH}px`;
  this.canvasRef.style.height = `${LANDSCAPE_HEIGHT}px`;
  this.containerRef.style.width = `${LANDSCAPE_WIDTH}px`;
  this.containerRef.style.height = `${LANDSCAPE_HEIGHT}px`;
  break;
```

Figure 20 - The logic behind the screen resizing in landscape mode.

In any case, in landscape mode, the footer (controlled by the **footer** component) is hidden so the canvas can expand in height. An additional div which enables users to scroll down "beyond" the canvas bottom end is shown instead, so they can fit the game to their screen for a "pseudo-fullscreen" experience. It also serves as padding for mobile screens on which the height of the canvas ends up being smaller than the browser window's height (akin to the letterboxing effect).

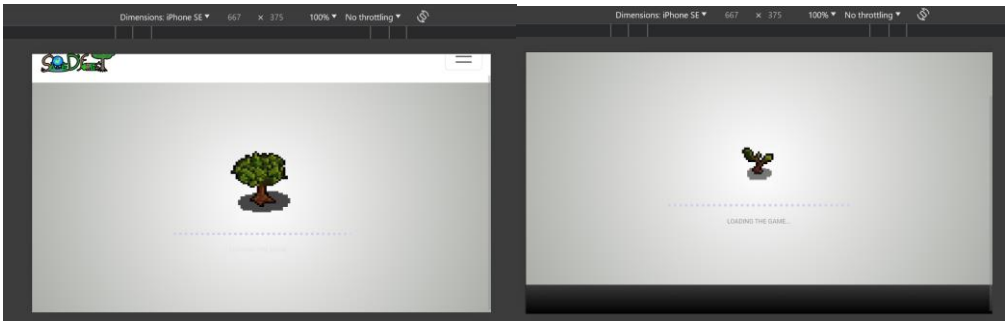


Figure 21 - On the right, there is no div being enabled. On the right, the same device, but with the aforementioned padding effect created by the div.

I had initially implemented an in-game button which set the game in fullscreen mode, but ultimately realized this approach was incompatible with the Angular implementation: besides the aforementioned aspect ratio issues, on mobile, when in fullscreen mode, neither the MatDialogs opened by mobile-forms service mobile-forms nor the mobile keyboard were overlaid on top of the game, as required.

how-to-play - this simple component displays a user manual/guide image for the game. As the name implies, it is associated with the 'howToPlay' route. There is no additional logic in the component class, it merely acts as a linker for the template and .css files.

mobile-forms - this component defines a modal dialog view that pops up on mobile devices when users interact with the game's input fields. These input fields, implemented with Unity's TMP_InputField class v3.0.6, don't trigger a mobile keyboard when interacted with in WebGL builds. A pre-release that adds this feature and is compatible with the chosen Unity editor can be installed, which I tried to, but by doing so all game objects using TMP classes ended up misconfigured, so I looked out for another approach.
I have tried to use one of the several in-game keyboard plugins available online (the one I attempted to use is available for free at https://github.com/kou-yeung/WebGLInput), but realized it clashed with the Angular web-app that embeds the game, throwing a lot of errors. Ultimately, I decided to take advantage of the fact that the game is embedded in the aforementioned Angular app and use a similar approach to the one suggested by a dev at stack-overflow, relying on javascript. The Angular Material library was thus used to define "MatDialogs" on which the user can type and that automatically prompt the mobile keyboard. MatDialogs are opened by the **mobile-forms service**; when a MatDialog is opened, an instance of this component is created within it. The instance is injected with both a **MatDialogRef**, to control the MatDialog it is contained in, and a **MAT_DIALOG_DATA** token, so it can access data passed to the MatDialog- in our case string parameters to be embedded into the template. Both the **element** (ex:"email") and **form** (ex: "login") strings parameters are embedded using interpolation, whereas the **current input** string is embedded and synced with the component class through the **ngModel** directive; additionally, depending on the element parameter, the type of the input (ex: numerical-only input) is restricted through data binding.

Figure 22 - The mobile-forms component class.

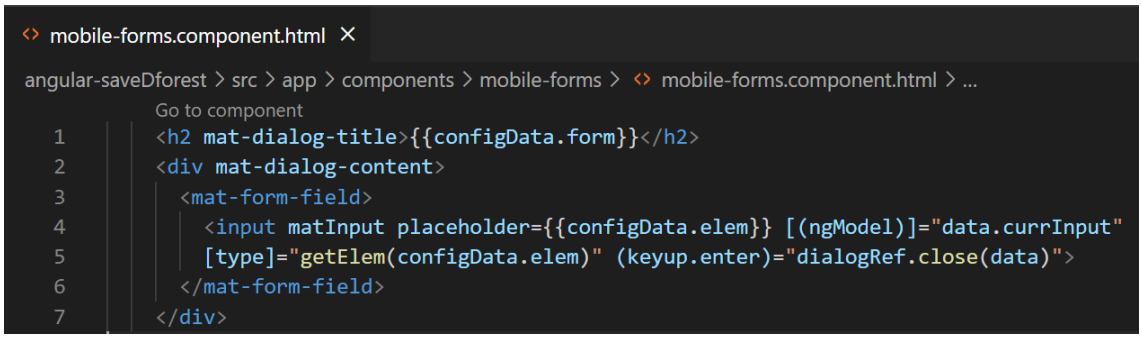


Figure 23 - The mobile-forms component html.

When a dialog is closed, either by pressing the “enter” key or by touching outside of the modal view, the current input string is passed back to the service and the component instance is destroyed.

navbar - this component defines the app’s navigation bar which allows users to navigate between different routes. Bootstrap is used to build the layout, and route navigating is enabled by using the **routerLink** directive for each navbar item pointing to a route.
The navbar items must be disabled as the game build is being loaded by the “**CreateUnityInstance”** function inside the **home** component, because if the user navigates to another route as this function is running, the game will stop and crash.
To synchronize the navbar’s state with the home component’s, a **navbar service** is used. This service includes an observable to which the navbar’s component class subscribes to; the navbar component is thus notified when the game build is/isn’t loading in the home component, reflecting this state on a boolean property of its own, **showNavItems**. By using the **ngClass** directive in the template, the navbar items’ class is modified, either disabling or enabling them according to showNavItems’ state.

```
export class NavbarComponent implements OnInit {
  showNavItems = false;

  constructor(private navbarService: NavbarService) { }

  ngOnInit(): void {
    this.navbarService.showNavItemsSubject.subscribe(show => this.showNavItems = show);
  }
}
```

Figure 24 - The navbar component class.

```
<li class="nav-item">
  <a [ngClass]="showNavItems ? 'nav-link hover-underline-animation' : 'nav-link disabled'"
  routerLinkActive="active-link" id="how-to-play" routerLink="/howToPlay">HOW TO PLAY</a>
</li>
```

Figure 25 - A nav-item on the navbar html.

other-resources - this component includes several hyperlinks to different organizations so users can access more information on the topics and themes of the game. The view depends entirely on its template and .css files, with the component class merely acting as a linker between them. Bootstrap's grid system is used to display multiple rows with 3 columns each; each column contains a bootstrap card.
On PC (devices with min-width:992px), the **<card-text>** element containing the resource description is hidden through css by setting the opacity to 0, and shown when hovered upon by setting the opacity to 1. The card itself is transformed when hovering, with its scale incrementing a bit. The initial idea was to have the cards show up with only the logos filling up the entire card-body, and when hovered upon, the logos would scale down and the descriptions would be revealed, but that concept was scrapped in favor of this simpler approach.
To add a new resource, copy a ** **and its children and paste it directly below the last one. Modify the **<card-body>** children with the logo, title, and short description of the new resource. There is no need to add new **<rows>**; columns will be automatically wrapped into new rows when needed.

reset-pwd - this component defines the view which users use to change their password. Its associated route, **'resetPwdToken/:token/:id'** is not accessible through the navbar; it is directly linked to the user through an email sent by the back-end. The route parameters **token** and **id** are also generated by the back-end.
The template consists of a form with a password and a "confirm-password" input field, a "rule-container" with the rules for a valid password, and a submit button. **ngClass** is once again used to change the text color of each rule, depending on if the password complies with each or not (which is checked by the component class). Data binding is used to enable/disable the button. Upon submitting a valid password, the **user service's submitNewPassword** method is invoked, creating an HTTP request. The route parameters are also sent through the request, so the server can check in the database if they match an existing password reset request token.

1.3.2 The interface directory

There are currently 3 interfaces in this directory that define the IRI, SAM, and User data types:

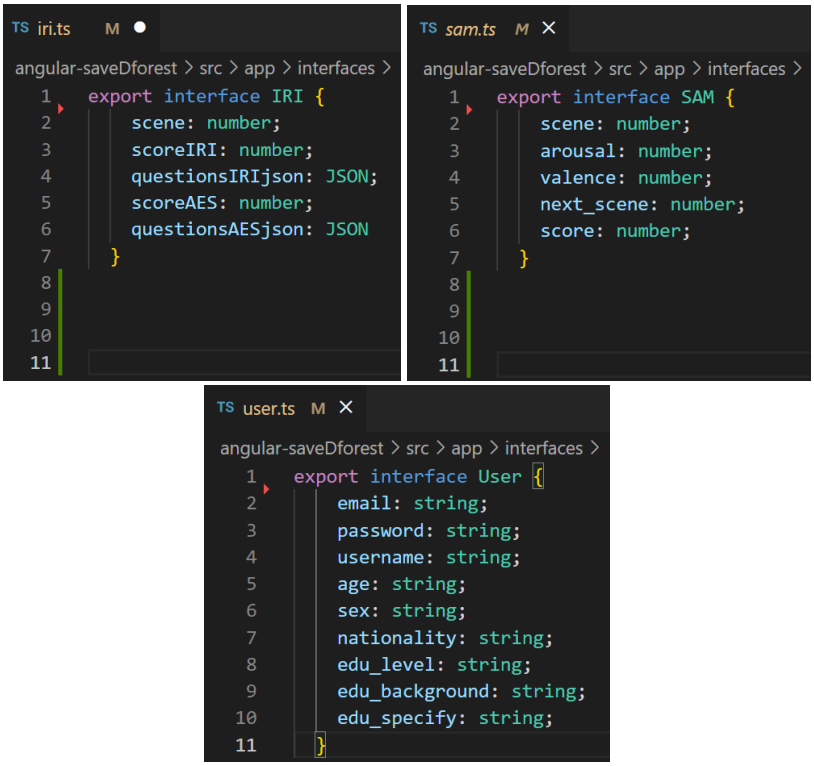


Figure 26 - The defined interfaces.

.

The use of these interfaces is optional; they just help ensure consistency for more complex data that is passed between classes. For instance, notice how by relying on an instanced **User** object parameter, the **sendSignup** method of the **user service** ends up being less verbose and prone to errors:

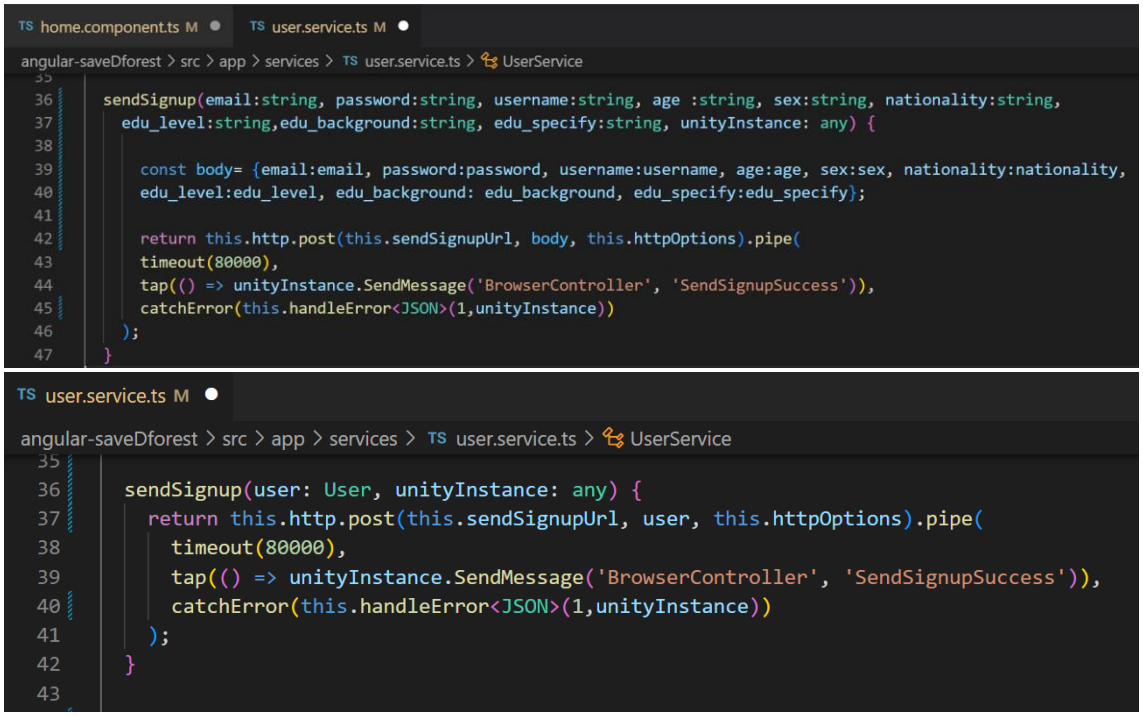


Figure 27 - A comparison between the sendSignup method not using interfaces on the top, and using them on the bottom.

To create a new interface, execute ***npm run generate-interface <interface_name>*** . After being defined, the interface is ready to be referenced and instances of it can be created through other classes, as long the following import statement is used : **import { <interface_name> } from '<interface_filename>'**;

1.3.3 The service directory

The project makes use of several services; each service handles a specific task, facilitating separation of concerns. Services are provided to components that depend on them through dependency injection. We've chosen to provide services at the root level by using the **@Injectable({providedIn: 'root'})** decorator, thus making them available for injection across all components as singletons.

To create a new service, execute ***npm run generate-service <service_name>.***
There are currently 6 services:

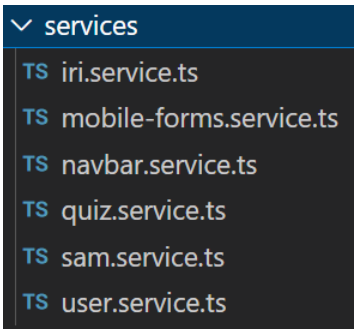


Figure 28 - The different services defined.

iri.service.ts, quiz.service.ts, sam.service.ts, user.service.ts - these 4 services are responsible for creating HTTP requests to the back-end server and handling their responses, thus synchronizing the front-end with the back-end and vice-versa (their "counterparts" being the back-end controllers). These services are injected in the **home** component, with the **user.service** also being injected in the **reset-pwd** component. Most of their methods are invoked by specific event listeners that are triggered by calls from the Unity game in specific situations:

- The **iri.service** handles operations related to the game's IRI scene (IRI and AES submitting in the first playthrough, and score and scene updating in subsequent playthroughs). It makes use of the IRI interface;
- The **quiz.service** handles operations related the game's Quiz scene (score updating and badge submitting);
- The **sam.service** handles operations related to the game's SAM scene (SAM submitting in the first playthrough, and score and scene updating in subsequent playthroughs). It makes use of the SAM interface;
- The **user.service** handles operations related to the game's TitleScreen scene (signup, login, logout, and password reset requesting), and additionally handles new password submitting that's invoked by the reset-pwd component. It makes use of the User interface.

Despite each of these services being responsible for a different set of data/operations, they all share a common structure, having:

- String properties that represent the back-end server API endpoints (as URLs), to which each operation will send its specific HTTP request to. These need to conform to the

rewrite rule that's configured in Render, so the session cookie set by the back-end is not treated by the browser as a third-party cookie;

- A "httpOptions" object property, consisting in a HTTP header with content type set to "application/json", which ensures that requests and responses are formatted in JSON, and a boolean "withCredentials" set to true, so the cookie set by the back-end upon signup or login is included in subsequent requests, identifying the client front-end app as "authenticated";
- The native httpClient service injected in the constructor so HTTP requests can be made;
- Methods that handle the aforementioned operations; upon receiving a response from the server, the methods that handle game related operations will use the Unity **SendMessage** function to propagate relevant information back to the BrowserController object in the current game scene. It will trigger a specific 'Success' method in the attached BrowserManagerScript, allowing flow control to return to the game.
  SendMessage is very limited when it comes to passing parameters - only a single number or string parameters are supported - so string concatenation is used to pass multiple parameters at the same time when required, with each parameter being separated by a comma so they can be parsed and converted in game.
  The **submitNewPassword** method of the **user service** does not contact the game, instead passing the control flow back to the reset-pwd component.
  If the server fails to respond within the specified time or encounters an error, the associated error and operation identifier will be passed to the handleError method;

Figure 29 - The sendLogin method of the userService.

- The handleError method, which categorizes errors based on operation identifier or error code and handles them. SendMessage is used once again, this time to propagate error information into the game, which will be shown in the warning canvas of the current scene.
  For **submitNewPassword** errors, an alert is issued and the error is propagated to the **reset-pwd** component.

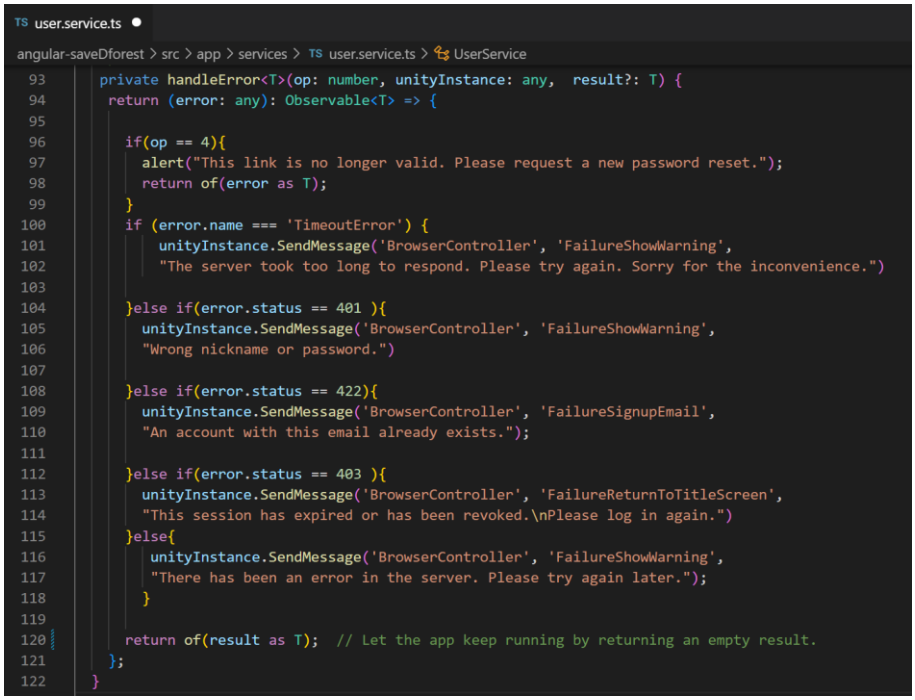


Figure 30 - The handleError method of the userService.

mobile-forms.service.ts - this service is responsible for opening MatDialogs containing instances of the **mobile-forms** component when users interact with a game input field while on a mobile device. The interaction is propagated from the game to the home component, which in turn invokes this service. The input field **element** (ex: “email”), the **form** (ex: “login”), and **current input** are passed as string parameters to it. The service then opens a MatDialog - creating an instance of the **mobile-forms** component, and passing the aforementioned parameters through a **MatDialogConfig** object - and subscribes to the instanced dialog’s **afterClosed** observable, ensuring that once the dialog is closed, the resulting input is propagated by this service to the game’s TitleScreen scene’s MobileController object, which updates the corresponding input field in-game.

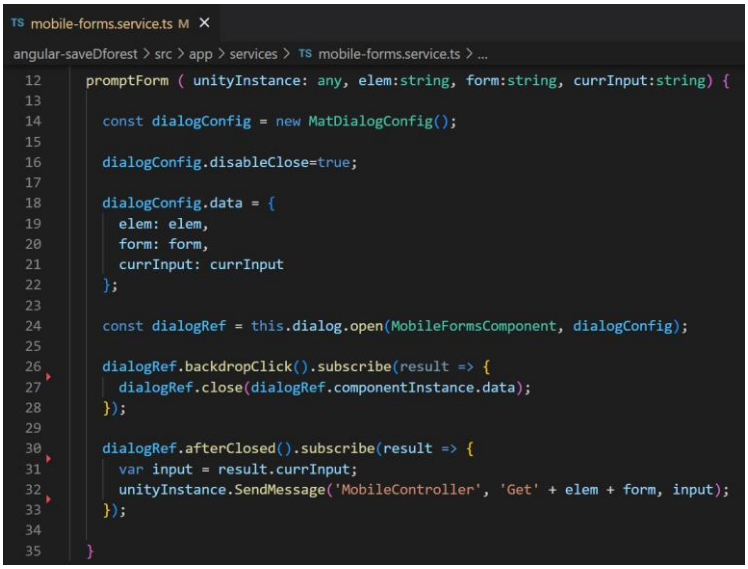


Figure 31 - The prompForm method on the mobile-forms service.

By default, when a touch occurs outside the MatDialog view, the dialog disappears from the screen, but the MatDialog **close** method isn’t invoked, and so the in-game input field is not updated, even if the user has modified the current input. Although this is the expected behavior for a modal window, I decided to alter it to align with the behavior of most mobile web apps’

input fields, where the current input is still updated even when the user touches outside the input field.
To achieve this, the MatDialog is first configured so a touch outside the view (a.k.a backdrop) does not make the dialog disappear (done through **dialogConfig.disableClose=true**), and an additional subscription is added so backdrop touch events invoke the close method.

navbar.service.ts - this is a very simple service for synchronizing the **navbar** component with the **home** component, by "sharing" a BehaviorSubject observable between the two. As the CreateUnityInstance function - which is responsible for loading the game build- is running inside the home component, if the user happens to use the navbar to navigate to another route, the game will stop loading and crash. To prevent this, the navbar items must be disabled while the game loads.
This service has a **showNavItemsSubject** boolean BehaviorSubject that is initialized as true (should the user access the app through any route other than the root ('') route associated with the home component, the navbar must be fully interactable).
When the home component is created, it invokes this service's **toggleNavItems** method, which updates the showNavItemsSubject to false, and invokes it once again to true once again after the loading is complete. The navbar component subscribes to the observable and acts accordingly.

B - Back-end

B1 - The Express server

The front-end communicates with a back-end server built and initialized using the Express framework, which runs on top of the Node.js runtime environment.
Express provides several out-of-the-box features that can facilitate back-end server development and deployment, such as managing endpoints and handling HTTP requests and responses. Furthermore, due to its popularity, there is a wide array of external module libraries that streamline common tasks, such as session management, authentication, or CORS handling.
Official Express documentation is available at https://expressjs.com/en/4x/api.html, and setup instructions can be found at https://expressjs.com/en/starter/installing.html.
An introductory tutorial - which helped kickstart this project - is available at https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs.
Postman was used to test the server endpoints during development.
This project was built using Express v.4.18.2, running on top of Node v.16.16.0. To further streamline the development process, the express-generator library (v.4.16.1) was used to generate a barebones Express application.
The server is initialized by running the command ***npm start***.

Express project components

1.1 The root directory

The project's root directory comprises the following:

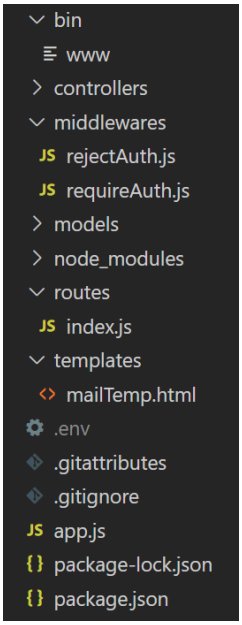


Figure 32 - The root directory of the Express project.

Relevant directories and files include:

bin - this directory is auto-generated by express-generator. It contains a **www** script file that defines the server's startup process; more concretely, it assigns the actual Express application instanced by the **app.js** file as a server, setting up the listening port and other configurations. As I found no need to modify the overall startup process, this file has remained unchanged since its inception. The ***npm start*** command defined in **package.json** is configured to run this file.

controllers - this directory contains controller modules with functions that define the logic for handling HTTP requests and generating appropriate responses. For more information on each controller, refer to 1.3 The controller directory.

middlewares - This directory contains reusable middleware modules that intercept and process the HTTP requests **prior to the controllers**. It currently contains two modules, each with its own specific function**:**

- **requireAuth.js**, which ensures that a user is **authenticated**.
- **rejectAuth.js**, which ensures that a user is **not authenticated.**

For more information on their useage, please refer to 1.3.1 The userController.

models - The modules in this directory define the structure of data stored in the database. To learn more about the models, refer to 1.2 The model directory.

node_modules - similar to the Angular project, this directory contains external libraries and dependencies that the server relies on; however, unlike the Angular project, this directory is actually pushed to the GitHub repository to prevent errors caused by the installation of incorrect versions of some libraries when deployed by the current hosting platform.
*** Do note that I have not included the node_modules directory in the project contained in the public repository.**

routes - this directory contains a single file named **index.js**, which defines all the server routes: endpoints that the front-end can connect to, along with the HTTP requests that can be responded to from said endpoints. Each route is linked to a specific handler function defined by a specific controller:

```
router.post('/signup', user_controller.signup);

router.post('/login',  user_controller.login);

router.post('/logout', user_controller.logout);

router.post('/resetPwdRequest',resetPwdLimiter, user_controller.reset_pwd_request);

router.post('/newPwdSubmit', user_controller.new_pwd_submit);

router.post('/iri', iri_controller.iri_post);

router.post('/updateScoreIRI', iri_controller.update_score_iri);

router.post('/sam', sam_controller.sam_post);

router.post('/updateScoreSAM', sam_controller.update_score_sam);

router.post('/quiz', quiz_controller.quiz_post);

router.post('/quizSetFinalBadge', quiz_controller.quiz_set_final_badge);

module.exports = router;
```

Figure 33 - The different routes defined.

Given that the server's structure is relatively simple, comprising only 11 routes, this approach is currently sufficient. Alternatively, and should the project scale further, this file could potentially be split into multiple smaller route files for better maintainability (for example, a userRoutes.js could describe all user-related routes - login, signup, etc-, while a iriRoutes.js could describe routes strictly related to the IRI, and so on).
For **the '/resetPwdRequest'** endpoint, strict rate-limiting is applied via an instance of the express-rate-limit middleware library to prevent abuse of the password reset functionality. This instance intercepts HTTP requests and limits each IP to 5 password reset requests for every 15 minutes, overriding the limiter instance set up globally (via the **app.js** file, as explained below).

templates - currently, this directory contains a single file (**mailTemp.html**) that defines a template for the emails sent when users request for their password to be reset. To ensure proper rendering across email clients, the CSS is defined inline, meaning the styles are directly embedded within the HTML file itself rather than being linked to an external stylesheet.
Certain fields are defined as placeholder strings (e.g., "{username}", "{reset_url}", "{front_end_url}"); these placeholders are dynamically replaced by actual values when the email is generated via a controller.

Figure 34 - An example of a placeholder string within the mail template.

This email template has been adapted from the open-source "Postmark Transactional Email Templates", available at https://github.com/ActiveCampaign/postmark-templates/tree/main?tab=readme-ov-file.

.env - this file contains environment-specific configuration variables, including settings or data deemed sensitive, regularly changed, and/or highly likely to change, under the syntax "<VARIABLE_NAME>=<value>". Thus, it is not pushed to the deployment repo*; instead, it is manually loaded within the current hosting platform environment. At runtime, the **dotenv** library is used to dynamically load the variables defined in the file, replacing the placeholders (with syntax **"process.env.<VARIABLE_NAME>"**) with their actual values.
***A barebones .env file containing solely the variable names is provided in the public repo.**

app.js - this file describes and instantiates the actual Express application to be started up as a server via the **www** file. It imports and makes use of several external middleware libraries (some of which are also present in other project files):

- **express-rate-limit (v7.5.0) -** this library is used to limit repeated requests to the API endpoints. The limiter instance set up here is applied globally; a stricter instance is configured for the password reset request endpoint (see **routes.js** above for details).

```
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  limit: 100, // Limit each IP to 100 requests per `window` (here, per 15 minutes)
  standardHeaders: true, // Return rate limit info in the `RateLimit-*` headers
  legacyHeaders: false, // Disable the `X-RateLimit-*` headers
})
```

Figure 35 - The global rate limiter instance.

- **CORS (v2.8.5) -** this library facilitates the setup of a CORS mechanism which essentially enables successful communication between the front-end and the back-end server, as for security reasons browsers restrict resource access between apps from different origins (complying with the same-origin policy). Since their subdomains differ, both the front-end and the back-end server are considered different origins; thus, as a result of the aforementioned policy, the server's responses to the front-end's HTTP requests are blocked by default. By setting up a CORS mechanism, the server marks the front-end's origin as "trustworthy", appending specific HTTP headers in responses to incoming requests which inform browsers that communication between the two is allowed. The current configuration is as follows:

```
app.use(cors({
  credentials: true,
  origin: [
    process.env.FRONT_END_URL,
    process.env.FRONT_END_TESTING_URL
  ]
}));
```

Figure 36 - The current CORS configuration.

By setting 'credentials' to 'true', another specific HTTP header is appended, enabling session cookies (configured by express-session, described below) to be included in incoming requests. Likewise, a HTTP header must also be configured in the front-end to successfully include session cookies.

- **express-session (v1.17.3) -** used to generate and manage user sessions so users can authenticate and remain authenticated across successive HTTP requests, provided they're sent from the same client browser.
  Essentially, it works by generating a session upon login/signup, storing it, and setting a session cookie in the client's browser containing the generated session's ID signed with a secret to prevent tampering; this cookie is then sent by the front-end in subsequent requests, allowing the server to identify the corresponding stored session document. By default, this library stores sessions in-memory, which is not suitable for production; to store sessions in the MongoDB database, the connect-mongo (**v4.6.0**) library is used.

```
app.use(session({
  secret: process.env.SESSION_SECRET,
  name: 'saveDforest-Login-Session-Cookie',
  resave: false,
  saveUninitialized: false,
  proxy: true,
  cookie: {secure: true, sameSite: 'strict'},
  store: MongoStore.create({ mongoUrl: process.env.MONGO_URL, stringify:false })
}));
```

Figure 37 - The current express-session configuration.

Seeing as Render.com - the current hosting platform - is behind a reverse proxy, Express must first be configured to trust the proxy via **'app.set('trust proxy', 1)'** in order to successfully identify the cookie as **'secure'** (the recommended setting).
Note as well that **'sameSite'** is set as **'strict'** - the cookie is successfully recognized by the front-end as a first-party cookie; this configuration is only possible because of the rewrite rules set in the front-end.
'MongoStore' refers to the session store connection configured via connect-mongo, with the connection string ('MONGO_URL') being obtained after the database is configured. **'Stringify'** is set to **'false'** so session properties aren't stored as single

JSON string in the database (which is done by default); this approach aligns with the Mongoose Session schema, thus enabling accurate session querying.

- **Mongoose (v7.5.2) -** used to establish a connection with the MongoDB database. It is also used to define schemas and models to manage database documents, as explained in 1.2 The model directory and 1.3 The controller directory.

```
async function main() {
  await mongoose.connect(process.env.MONGO_URL);
}
```

Figure 38 - Mongoose establishes a connection within the main function, which runs when the server is started by the www script.

1.2 The model directory

Each module in this directory defines a specific Mongoose 'schema' outlining the structure, fields and constraints of MongoDB document collections, along with Mongoose 'models' that are compiled from the schemas. These models then serve as an interface for manipulating documents, with each document being represented by an instance of a model.
Schema design was influenced not only by the logical organization of documents and query optimization, but also stakeholder requirements (for instance, for AES and IRI questionnaires, it became clear early on that responses to each single item would need to be stored rather than just their aggregate scores, so results could be better understood and analyzed by psychologists), and front-end constraints (for example, due to Unity's SendMessage - which supports only a limited range of data types -, game badge "index numbers" are concatenated and stored as a single string).
The directory contains the following:

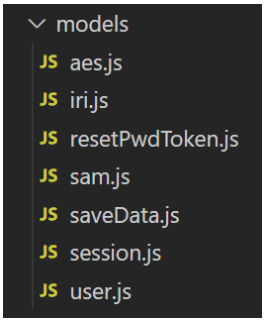


Figure 39 - The model directory.

aes, iri - outline the structure of documents representing AES and IRI questionnaires, respectively. Overall, they share the same properties**,** with the '**scene'** number property indicating whether it is the initial or final AES/IRI- while a value of 0 means it was submitted after the player signs up and clicks play (before scene 1 - corresponding to game scenario 1- is loaded), a value of 10 indicates it was submitted after scene 10, at the end of the game.
The '**AES/IRIQuestions'** defines a map containing each questionnaire item and its corresponding answer given to it by the user.

```
const AESSchema = new Schema({
    scene: {type: Number, required: true},
    scoreAES: {type: Number, required: true},
    user: { type: Schema.Types.ObjectId, ref: 'User', required: true },
    AESquestions: {
        type:Map,
        of: Number
    }},
    {timestamps: { createdAt: true, updatedAt: false }
});
```

Figure 40 – The AES schema; IRI shares a nearly identical structure.

resetPwdToken - defines password reset tokens, stored in the database when a user successfully requests for a password reset mail to be sent. Password tokens are valid for 24 hours.

```
const ResetPwdTokenSchema = new Schema({
    token: { type: String,required: true},
    createdAt: { type: Date, default: Date.now, expires: 86400},
    user: { type: Schema.Types.ObjectId, ref: 'User', required: true, unique: true },
});
```

Figure 41 -The ResetPWDToken schema.

sam - defines SAM questionnaires.

```
const SAMSchema = new Schema({
    scene: {type: Number, required: true},
    arousal: {type: Number, required: true},
    valence: {type: Number, required: true},
    user: { type: Schema.Types.ObjectId, ref: 'User', required: true }},
    {timestamps: { createdAt: true, updatedAt: false }
});
```

Figure 42 -The SAM schema.

saveData – represents users' current game progress.

```
const SaveDataSchema = new Schema({
    user: { type: Schema.Types.ObjectId, ref: 'User', required: true, unique: true },
    currentScene: {type: Number, required: true},
    score: {type: Number},
    badges: {type: String},
    finishedGame: {type: Number},
    maxScore: {type:Number}

});
```

Figure 43 -The SaveData schema.

session – defines user sessions. Unlike the other models, which automatically receive an ID of type 'ObjectId' when stored as documents in the database through Mongoose, session IDs are generated by express-session (and stored via connect-mongo), which sets up a string type ID; for this reason, **'_id'** must be explicitly declared as **'string'** in the schema.
If this isn't done, Mongoose will assume the _id is an ObjectId, which causes a 'no _id found' error when it tries to delete a user's previously started session upon a fresh login from the same user coming from another device/browser.

```
const SessionSchema = new Schema({
    _id: {
        type: String,
        required: true
    },
    expires: { type: Date, required: true },
    session: {
        cookie: {type:Object},
        user: {type: Schema.Types.ObjectId, ref: 'User', required: true},
        isAuthenticated: {type: Boolean}
    }

});
```

Figure 44 -The Session schema.

user – defines users; includes all information submitted upon signing up.

```
const UserSchema = new Schema({
    email: {type: String, required: true, unique: true},
    password: {type: String, required: true},
    username: {type: String, required: true},
    age: {type: String, required:true},
    sex: {type: String, required:true},
    nationality: {type: String, required:true},
    education_level: {type: String, required:true},
    education_background: {type: String},
    education_background_specified: {type: String},
});
```

Figure 45 -The User schema.

1.3 The controller directory

Controllers handle logic related to routes: they're responsible for processing front-end HTTP requests, delegating CRUD database operations to models, and generating appropriate HTTP responses. There are currently 4 controllers, each managing a set of related routes:

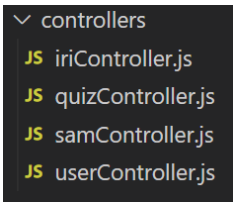


Figure 46 -The controller directory.

1.3.1 The userController

The userController handles user sign-up, login, log out, password-reset requesting, and password-reset submitting routes, with its front-end "counterpart" being the userService.

Upon a valid sign-up request, the **signup** function stores all user submitted data into the database via the User model, with the password being hashed with bcrypt beforehand. Once the user document has been stored, a session is prepared for storage by making use of express-session's 'req.session' object, which in this case is used to set the user's '_id' as a property and the 'isAuthenticated' flag to true. The Session document is automatically stored at the end of the request-response cycle, and a session cookie is also automatically sent to the front-end by express-session.

```
const user = new User ({email: req.body.email, password: req.body.password, username: req.body.username,age: req.body.age,
sex: req.body.sex, nationality: req.body.nationality, education_level: req.body.edu_level,
education_background: req.body.edu_background, education_background_specified: req.body.edu_specify});

const salt = await bcrypt.genSalt(10);
user.password = await bcrypt.hash(user.password, salt);

await user.save()
req.session.user = user._id;
req.session.isAuthenticated = true;
```

Figure 47 - An excerpt from the userController's signup function.

When it comes to a valid user **login** request, it is first verified whether the provided email and password match a User document in the database; if so, the Session collection is then checked to see if the user already has a session on the database (which typically happens when the user had previously logged in through a different browser/device but didn't log out); If it does, it is deleted, as only a single session per user is allowed at any given time, in order to ensure consistency throughout the app. Regardless of whether a prior session existed or not, a new session is established. Afterwards, if there's a SaveData document associated with the user, its contents are retrieved and sent back to the front-end, along with the username found in the User document.

```
const user = await User.findOne({email: req.body.email }).exec();
if(user){
  const validPassword = await bcrypt.compare(req.body.password, user.password);
  if (validPassword) {

    const sessiondb = await Session.findOne({ 'session.user': user._id }).exec();
    if(sessiondb){
      sessiondb.deleteOne();
    }

    req.session.user = user._id;
    req.session.isAuthenticated = true;

    const saveData = await SaveData.findOne({ user: user._id }).exec();
    var currentScene = saveData?.currentScene ?? 0;
    var score = saveData?.score ?? 0;
    var badges = saveData?.badges ?? "";
    var finishedGame = saveData?.finishedGame ?? 0;
    var maxScore = saveData?.maxScore ?? 0;
```

Figure 48 - An excerpt from the userController's login function.

The **rejectAuth** middleware intercepts all incoming login and signup requests prior to the functions to ensure that only non-authorized users can access these features.

```
exports.signup = [rejectAuth, async function(req, res,next) {
```

Figure 49 - Setting up rejectAuth to intercept requests prior to the signup function.

This is done because while the cookie ensures authentication persistence between front-end and back-end, the Unity game does not rely on it to verify authentication status; instead, upon a successful sign-up or login, the game stores user-related data - such as username, score, current scene, and badges - in the browser's IndexedDB, through Unity's built in 'PlayerPrefs' class (with this data being managed and updated throughout the game).
This approach thus enables the game to maintain state efficiently without needing to constantly query the database: for example, if a user logs in, then closes and reopens the app via the same browser (without clearing cookies), they'll retain their logged-in state in-game despite the game being reloaded from scratch, as their data persists in IndexedDB.
However, if a user deletes their browser's IndexedDB and then reopens the app - again, without clearing cookies - , the game will assume the user is not authenticated since there's no user data

on the indexedDB; as a result, the game's main menu will prompt the user to either login or sign-up.
For this reason, and to prevent issues should the above happen, incoming sign-up and login requests are first intercepted by rejectAuth, which will check if the user is still authenticated; if so, it will first destroy the session and remove the cookie on the front-end (do note that 'req.session.destroy' doesn't clear the front-end cookie by itself), before letting the user perform another login/ sign-up request, now unauthenticated.

```
module.exports = function rejectAuth(req, res, next) {
    if(req.session.isAuthenticated === true){
        try{
            req.session.destroy(function(err) {
            res.clearCookie('saveDforest-Login-Session-Cookie', {
                sameSite: "none",
                secure: true,
            });
            console.log("session logging out due to removed playerprefs = " + req.sessionID);
            res.status(403).send({ success: false, message: 'operation failed, user was still authenticated' });
            })
        }catch(err){
            console.log(err);
            return res.status(500).send({ success: false, message: 'server error' });
        }
    }else{
        next();
    }
}
```

Figure 50 - The rejectAuth function.

Ideally, this would be all done on a single request-response cycle, but as far as I'm aware, there is no way to both remove and create a new a session cookie on the same cycle, despite the ability to regenerate a session. A future improvement could involve prompting the user to decide whether they would like to continue with their previous session instead of immediately destroying it.

A successful **logout** destroys the session associated to the user in the database (and the current 'req.session' object as well) and removes the cookie from the front-end; it thus requires the user to be authenticated, which is ensured by the **requireAuth** middleware. This middleware verifies that incoming HTTP requests originate from authenticated users (by checking the 'isAuthenticated' flag of the incoming session, set up after a successful sign-up/login); if so, the request proceeds to the controller. Otherwise, the middleware blocks the request, preventing any further processing and sending a failure response to the front-end, which acts accordingly by deleting the browser's IndexedDB data maintained by the game, and reloading the game build.

```
exports.logout= [requireAuth, async function (req, res,next){

    try{
      req.session.destroy(function(err) {
        res.clearCookie('saveDforest-Login-Session-Cookie', {
          sameSite: "none",
          secure: true,
        });
        console.log("session successfully logging out = " + req.sessionID);
        res.status(200).send({message: "logged out successfully"});;
      })
    }catch(err){
      console.log(err);
      return res.status(500).send({ success: false, message: 'server error' });
    }

}]
```

Figure 51 - The userController's logout function.

reset_pwd_request handles password reset requests. Assuming the email provided matches one of the users, a reset token is randomly generated, which is then salted, hashed and stored in the database, within a ResetPwdToken document. If a token already existed for the user, it is replaced, to ensure only one valid reset token is stored per user at any given time.

```
const user = await User.findOne({email: req.body.email }).exec();

if(user){
  let resetTokenString = crypto.randomBytes(32).toString("hex");
  const salt = await bcrypt.genSalt(10);
  const hashedResetString = await bcrypt.hash(resetTokenString, salt);
  const token = new resetPwdToken ({token: hashedResetString, createdAt: Date.now(), user: user._id });
  const username= user.username;
  try{
    await token.save()
  }catch(err){
    console.log(err);
    if(err.code === 11000){
      try{
        await resetPwdToken.deleteOne({ user: user._id });
        await token.save();
```

Figure 52 - An excerpt from the userController's reset_pwd_request function, storing a resetPwdToken.

Afterwards, a password reset link - which will take the user to the front-end **'resetPwdToken/:token/:id' route** - is constructed using the un-salted, un-hashed reset token and the user's ID as parameters. The constructed link then populates the email template configured in mailTemp.html**,** along with the user's username, the app's email, and a link to the app's front-end home route (which is inserted in the email's app logo).

```
const link = process.env.FRONT_END_URL + "/resetPwdToken/:" +resetTokenString + "/:" + user._id;
htmlFile = await readFile('templates/mailTemp.html', 'utf8')
const replacements = {
  '{front_end_url}': process.env.FRONT_END_URL,
  '{reset_url}': link,
  '{username}': username,
  '{mail}': process.env.MAIL
};

htmlFile = htmlFile.replace(/{front_end_url}|{reset_url}|{username}|{mail}/g, match => replacements[match]);
```

Figure 53 - An excerpt from the userController's reset_pwd_request function, setting up the mail to be sent.

The nodemailer library is used to send the email; as such, a transporter object must first be configured and authenticated based on the chosen email service and authentication method. Previously, the app used an Outlook account with basic authentication, but that approach stopped working at the end of 2024 due to basic Outlook authentication being deprecated; since integrating Outlook with OAuth proved to be challenging, I opted to switch to Gmail with OAuth 2.0 authentication.
Successfully configuring the transporter to authenticate for Gmail with OAuth 2.0 requires the app to have been previously registered and OAuth to have been configured in the Google Cloud Console, so as to get a client ID and a client secret which must also be exchanged in advance for a refresh token via Google OAuth Playground; for more details, refer to https://www.vanguardbytes.com/how-to-send-emails-with-node-js-using-smtp-gmail-and-oauth2 and https://medium.com/@nickroach_50526/sending-emails-with-node-js-using-smtp-gmail-and-oauth2-316fe9c790a1.
As the app has been previously registered, OAuth has been configured, and the aforementioned variables have all been obtained, the server is able create an OAuth2 client (via googleapis library), that connects to OAuth Playground (the redirect URI) to generate an access token, which is then used by the transporter - along with the other variables - , to authenticate in Gmail's server.

```
const oAuth2Client = new google.auth.OAuth2(process.env.CLIENT_ID, process.env.CLIENT_SECRET, process.env.REDIRECT_URI);
oAuth2Client.setCredentials({ refresh_token: process.env.REFRESH_TOKEN});
const accessToken = oAuth2Client.getAccessToken();

var transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    type: 'OAuth2',
    user: process.env.MAIL,
    clientId: process.env.CLIENT_ID,
    clientSecret: process.env.CLIENT_SECRET,
    refreshToken: process.env.REFRESH_TOKEN,
    accessToken: accessToken,
  }
})
```

Figure 54 - An excerpt from the userController's reset_pwd_request function, setting up the transporter.

With the transporter successfully configured and authenticated, the email is sent through the 'sendMail' function.

```
let info = await transporter.sendMail({
  from: '"saveDforest 🌳" <' + process.env.MAIL + '>',
  to: req.body.email,
  subject: "saveDforest - Forgot your password?",
  text: "Reset password",
  html: htmlFile,
});
```

Figure 55 - An excerpt from the userController's reset_pwd_request function, sending the mail.

new_pwd_submit handles password updating requests coming from the aforementioned **'resetPwdToken/:token/:id'** front-end route. It will only update the user's password if there's a ResetPwdToken document associated with said user in the database, and if the token provided by the request – which the front-end obtains from the route's ':token' parameter- matches the hashed token in the database.

```
const tokenDB = await resetPwdToken.findOne({ user: req.body.id }).exec();
  if(tokenDB){
    const isValid = await bcrypt.compare(req.body.token, tokenDB.token);
    if (isValid) {
      const salt = await bcrypt.genSalt(10);
      password = await bcrypt.hash(req.body.password, salt);
        await User.updateOne(
          { _id: req.body.id },
          { $set: { password: password } }
        );
        await tokenDB.deleteOne();
```

Figure 56 - An excerpt from the userController's new_pwd_submit function.

1.3.2 The iriController

The iriController handles routes related to the game's IRI scene, with its front-end counterpart being the iriService.

During the user's first playthrough, the **iri_post** function receives requests containing IRI and AES item responses, their aggregate scores, and a **"scene" number indicating whether the request was sent at the beginning** (submitted via the first IRI game scene; scene 0) **or end** (submitted via the last IRI scene; scene 10) of the playthrough. When the initial questionnaires are received, in addition to storing said questionnaires, a SaveData document is also created on the database, with its "currentScene" property set to 1, signaling that the user is now proceeding to the first scenario; likewise, when the final questionnaires are received, the SaveData document is updated to indicate the user has finished their first playthrough, along with the

maximum score obtained for the first playthrough; “currentScene” is also set to 1 (thus signaling that the next playthrough’s first IRI game scene is to be bypassed).

```
const IRIobj = JSON.parse(req.body.questionsIRIjson);

const AESobj = JSON.parse(req.body.questionsAESjson);

const iri = new IRI ({scene: req.body.scene, scoreIRI: req.body.scoreIRI,
user: req.session.user, IRIquestions: IRIobj});

const aes = new AES ({scene: req.body.scene, scoreAES: req.body.scoreAES,
user: req.session.user, AESquestions: AESobj});

await iri.save()
await aes.save()
if(req.body.scene == 0){
    const save = new saveData ({user: req.session.user, currentScene: 1});
    await save.save();
}else{
    const save = await saveData.findOne({ user: req.session.user }).exec();

    var maxScore= save.score;

    await saveData.updateOne(
        { user: req.session.user },
        { $set: {currentScene:1, score: 0, finishedGame:1, maxScore : maxScore} }

    );
}
```

Figure 57 - An excerpt from the iriController’s iri_post function.

At the end of each subsequent playthrough, the last IRI game scene (which displays a loading screen upon loading instead of the usual questionnaire UI) delegates a request to the **update_score_iri** function, which compares the submitted score with the previously saved “maxScore” and updates it if the new total score is higher. The function also sets “currentScene” to 1 again, so the next playthrough begins directly with scenario 1 – making it so that **for all subsequent playthroughs, the first IRI game scene is bypassed**.
For both functions, the requireAuth middleware is used to ensure users are authenticated.

```
exports.update_score_iri = [requireAuth, async function(req, res,next){

    try{
        const save = await saveData.findOne({ user: req.session.user }).exec();

        var newMaxScore = save.score > save.maxScore ? save.score : save.maxScore;

            await saveData.updateOne(
                { user: req.session.user },
                { $set: { currentScene: 1,
                    score: 0, maxScore: newMaxScore} }
            );

        res.status(200).send({message: "successfully updated save after finishing playthrough", maxScore: maxScore});
    }catch(err){
        console.log(err);
        return res.status(500).send({ success: false, message: 'error updating data after finihing playthrough' });
    }
}]
```

Figure 58 - The iriController’s update_score_iri function.

1.3.3 The samController

The samController handles routes related to the game’s SAM scene, with its front-end counterpart being the samService.

The rationale behind it is roughly the same as the iriController: the **sam_post function** handles all requests during the user’s first playthrough, receiving and storing SAM responses in the database (each identified by the corresponding scenario “index” number, provided via req.body.scene), while also updating the user’s score with the points won through the scenario

decisions, and setting the next scene to be loaded by updating the “currentScene” field in the associated SaveData document (which is provided via “req.body.nextScene”).

```
const sam = new SAM ({scene: req.body.scene, arousal: req.body.arousal,
valence: req.body.valence, user: req.session.user});

    await sam.save()
    await saveData.updateOne(
        { user: req.session.user },
        { $set: { currentScene: req.body.next_scene,
            score: req.body.score} },
    );
```

Figure 59 - An excerpt from the samController’s sam_post function.

Conversely, the **update_score_sam** simply updates score and scene progression during subsequent playthroughs (and unlike the IRI game scenes, none of the SAM game scenes are bypassed).
Once again, requireAuth middleware ensures users are authenticated.

1.3.4 The quizController

The quizController handles routes related to the game’s quiz scene, with its front-end counterpart being the quizService.

The **quiz_post** function expects requests containing the player’s score after finishing a quiz, the “index” number of the completed quiz (via “req.body.quiz”), and a number flag (via “req.body.badge”, with value 0 or 1) indicating whether the user has won said quiz’s corresponding badge. The user’s associated SaveData document is updated to reflect their progress.

```
if(req.body.badge == 0){
    await saveData.updateOne(
        { user: req.session.user },
        { $set: { score: req.body.score} }
    );
}else{
    const save = await saveData.findOne({ user: req.session.user }).exec();

    var tempBadges ="";
    if(save.badges!= undefined){
        tempBadges = save.badges + " " + req.body.quiz;
    }else{
        tempBadges = req.body.quiz;
    }

    await saveData.updateOne(
        { user: req.session.user },
        { $set: { badges: tempBadges,
            score: req.body.score} }
    );
}
```

Figure 60 - An excerpt from the quizController’s quiz_post function.

quiz_set_final_badge is quite unique, in that it **only handles requests sent if the game has verified that the user has met the requirements to earn the special badge, upon completing the final quiz**. Since a user can earn the special badge (badge 9) without necessarily winning the final quiz badge (badge 8), a “req.body.badge” number flag is also received; a value of 1 indicates the player has earned both badges 8 and 9, while a value of 0 means only the special badge 9 was earned.

```
const save = await saveData.findOne({ user: req.session.user }).exec();

var badgesToAdd = req.body.badge == 1 ? "8 9" : "9";

var tempBadges ="";
if(save.badges!= undefined){
    tempBadges = save.badges + " " + badgesToAdd;
}else{
    tempBadges = badgesToAdd;
}

await saveData.updateOne(
    { user: req.session.user },
    { $set: {score: req.body.score, badges: tempBadges} }
);
```

Figure 61 - An excerpt from the quizController's quiz_set_final_badge function.

Do note that **if the player fails to meet the requirements for the special badge at the end of the final quiz, the game will instead delegate a request to quiz_post**, not quiz_set_final_badge.
The requireAuth middleware enforces authentication once again for both functions.

B2 - The MongoDB database

As explained earlier, the Express.js server delegates CRUD operations to a MongoDB database. MongoDB is characterized as a non-relational database wherein data is stored as JSON documents in binary form (BSON).
Documents are grouped in collections, with documents in each collection generally sharing the same structure/fields, as rows from the same table would in a relational database, but with the difference that documents of the same collection can also have different fields from each other (and data types can vary as well).
The structure of each collection in this project is directly mapped from a Mongoose schema defined by the server. For more information on the schemas, refer to 1.2 The model directory.
Official MongoDB documentation can be found at https://www.mongodb.com/docs/manual/introduction/.
A free-tier MongoDB Atlas cloud database cluster was used for this project; instructions for setting up an Atlas cluster are included in the same introductory tutorial used to kickstart the server project, being available at https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs/mongoose#setting_up_the_mongodb_database . After a successful initial setup, a connection string - which is required by the server to successfully connect to the database, is made available.
The database and its contents can be accessed via a browser GUI, but I personally prefer using the standalone MongoDB Compass GUI for analyzing and querying data, as I find it less cluttered; Compass also features an intuitive data export function which is quite useful for the project at hand and doesn't exist in the browser GUI. Setup instructions and documentation for Compass can be found at https://www.mongodb.com/docs/compass/.

The simplified UML diagram below illustrates the relationships between documents of the collections:

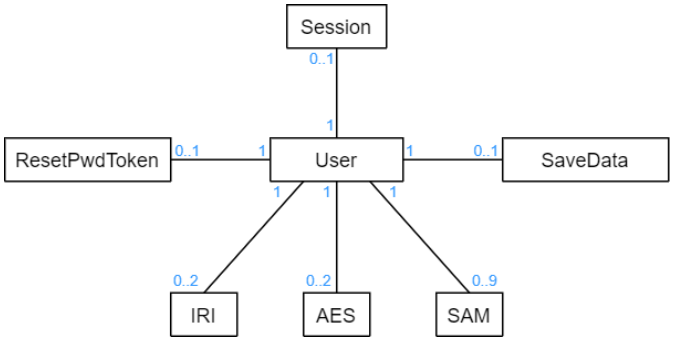
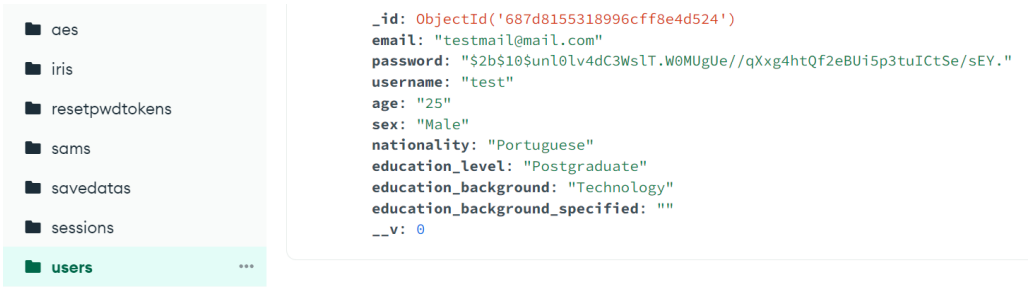
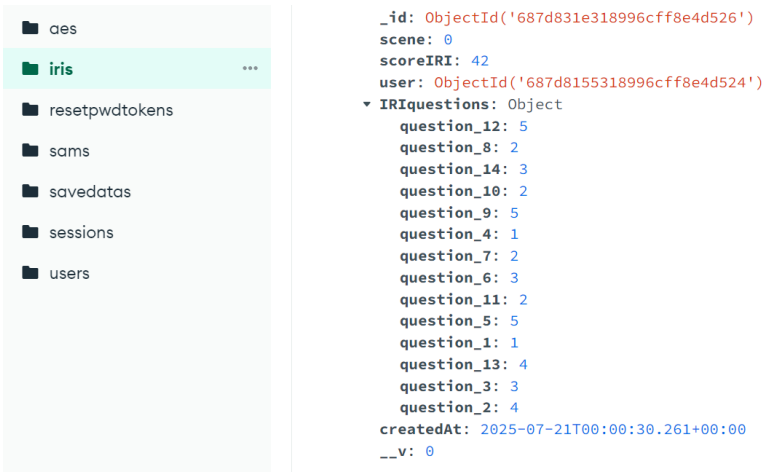
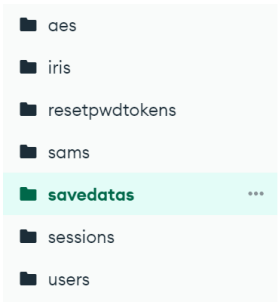


Figure 62 - The relationships between the defined document collections; for example, at any given time, a single user can be associated with up to 9 SAM documents in the database. Likewise, each SAM document is linked to exactly one user.

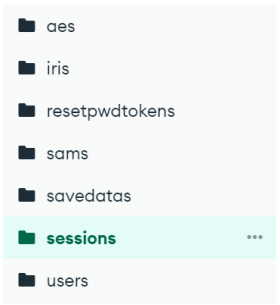
The following images depict a mock document from each collection, all associated with the same user, as displayed in MongoDB Compass:








```
_id: ObjectId('687d831e318996cff8e4d52a')
user: ObjectId('687d8155318996cff8e4d524')
currentScene: 2
__v: 0
score: 7
badges: "3 4"
finishedGame: 1
maxScore: "54"
```


```
_id: "btKg0CodRCNiC8iiPk072-7A_ybXCmUr"
expires: 2025-08-03T23:52:53.677+00:00
session: Object
  cookie: Object
    originalMaxAge: null
    expires: null
    secure: true
    httpOnly: true
    domain: null
    path: "/"
    sameSite: "strict"
  user: ObjectId('687d8155318996cff8e4d524')
  isAuthenticated: true
```

C - Initial setup

This section outlines the steps required to prepare both the front-end and back-end projects, either for running locally or deploying.

Prerequisites

- **Node.js v16.16.0 and npm v8.11.0 must be installed**; Node - along with npm - can be installed globally (via https://nodejs.org/en/blog/release/v16.16.0), but doing so will "lock" the system to these specific versions. For this reason, installing through a Node version manager is recommended, as it enables switching between different Node and npm versions throughout different projects.
  Current Node and npm versions can be checked in a terminal, through the commands *node -v* and *npm -v,* respectively.

- **After Node and npm are set up, local project dependencies must be installed**. For each project, open a terminal on their root directory - I like to use the terminals within VSCode for this. Run ***npm install*** on both to install all external libraries the projects depend on. Do note that both Angular CLI and Express.js will be installed as local dependencies; while the specific Angular CLI version used in the project can also be installed globally, there's no need to: in the project's **package.json** file, script commands can be defined to automatically resolve npm commands (ex: *npm start*) into specific Angular commands (like *ng serve --open*) from the locally installed Angular CLI version. This approach ensures that the version of Angular CLI used is the one defined in the project, thus avoiding any conflicts between the local version and a globally installed one (if any).

- **A MongoDB Atlas Database must be configured, and a connection string to access said database must be obtained**. Instructions for setting up a free-tier Atlas database are included in the same tutorial used to kickstart the server project, available at https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs/mongoose#setting_up_the_mongodb_database. After a successful initial setup, a connection string - which is required by the back-end server to successfully connect to the database - is made available.

- **A Gmail account and OAuth 2.0 credentials must be set up (required only for the password-reset feature)**. To send password reset emails, the back-end server requires valid OAuth credentials to authenticate in Gmail. For instructions on setting up the OAuth credentials refer to https://www.vanguardbytes.com/how-to-send-emails-with-node-js-using-smtp-gmail-and-oauth2 and https://medium.com/@nickroach_50526/sending-emails-with-node-js-using-smtp-gmail-and-oauth2-316fe9c790a1. For more details on how these are used by the back-end server, refer to the description of the reset-pwd-request function, in 1.3.1 The UserController.
  Only the password-reset feature requires the Gmail account and OAuth credentials; without them all other features will still remain functional.

D - Running the web-application locally

Considering that the projects discussed throughout sections A1 and B1 are specifically configured for respectively creating deployment builds and being deployed, a few steps must first be completed before they can be successfully run locally.

1. Local setup

Front-end Angular project

- Embed the "**v29_for_local_testing**" Unity game build files, specifically configured for local testing, and stored in **src/assets/Build** (unlinking the default, deployment-ready game build):

    1. in src/app/components/**home.component.ts**, in the "**ngOnInit()**" method, under **config**, reference the data, framework, and wasm files as properties **dataURL**, **frameworkURL**, and **codeURL**, respectively (these are already set there as comments, simply uncomment them and comment the "v29enabledtest" file assignments).

    2. In the **angular.json** file, under "**build>options>scripts**", reference the "v29_for_local_testing" loader.

Figure 63 - Embedding the game build configured for local testing.

- For all services that make HTTP requests - **iri.service, quiz.service, sam.service,** and **user.service** -, set all server endpoints' URLs to "**http://localhost:3000/<endpointURL>**" (these are already provided as comments, simply uncomment them and comment the deployment endpoints).

Figure 64 – Changing to the local server endpoints (IriService is shown here)

Back-end Express project

- Ensure **.env** file (stored on the root directory) contains valid:
    - FRONT_END_URL (set to “http://localhost:4200”);
    - MONGO_URL (MongoDB Atlas database connection string, obtained in the initial setup);
    - SESSION_SECRET (set to any string, ideally a randomly generated one. MUST NOT be empty);
    - MAIL and OAuth related variables **(required only for the password-reset feature**, obtained in the initial setup).
- In **app.js**, located on the root directory:
    - set express-session to use the local configuration (already set there as comment, simply uncomment it and comment out the deployment one).

Figure 65 - Setting express-session to the local configuration.

2. Launching the front and back-end locally

1. Open a terminal on the root directory of each project;
2. Run ***npm start*** on both.
    - For the Angular project, a local development server will be started on http://localhost:4200. *npm start* executes the Angular CLI’s *ng-serve --open* command (as defined in the package.json metadata file), and so a browser tab will automatically open on the URL mentioned above. While the process is running, any saved changes to project files will trigger an automatic tab refresh, with the modifications done to the app being reflected immediately.
      The front-end can be tested locally on Android phones by port forwarding the URL mentioned above through USB using Chrome DevTools. Documentation for setting up port forwarding to Android devices via Chrome can be found at https://developer.chrome.com/docs/devtools/remote-debugging/local-server#usb-port-forwarding.

- The Express back-end server will be started on http://localhost:3000. Do note that unlike the Angular app, the server does not automatically recompile or refresh as it is running; to make code changes effective, quitting the process and restarting the server again is required.

  *Do also keep in mind that even if everything is successfully configured for the password-reset feature to work as intended, the emails sent by an Express server running locally will not display the app's logo (logo.png) correctly, even if its source is set to http://localhost:4200/assets/logo on mailtemp.html, as the image is not hosted anywhere.

E - Building and deploying the web-application

This section details the steps required for building and deploying a version of the web-app. Do note that the front-end Angular project embeds a deployment-ready Unity game build by default.

1. Configuring and creating the Unity game build

Game scenes required for the Unity build must be set through "**File-> Build Settings**" in the Unity editor interface. This menu is also where remaining build configurations are set and builds are created. The current build settings are as follows:

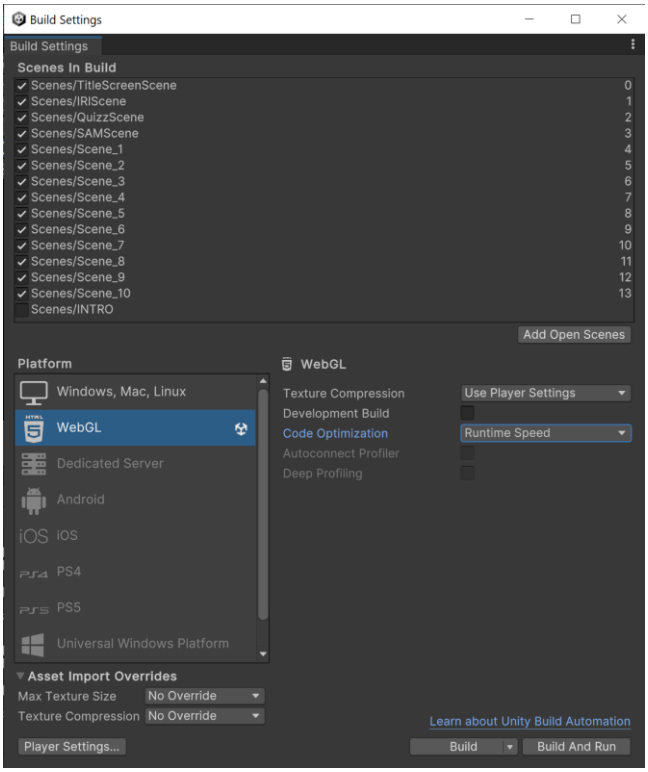


Figure 66 - The game's current build settings.

Even though it is recommended to use the "Disk Size" option for "Code Optimization" when building for mobile web browsers due to it generating smaller sized files, both "Runtime Speed" and "Disk Size" builds end up with nearly the same size for the current project (~1MB of difference), so "Runtime Speed" was chosen to prioritize in-game performance whilst still keeping a "relatively optimal" sized build that can perform decently on mobile browsers.

"**Player Settings**" are also configured via the "Build Settings" menu. There were a few changes done to the default settings; currently, they're as follows:

"**Resolution and Presentation**":

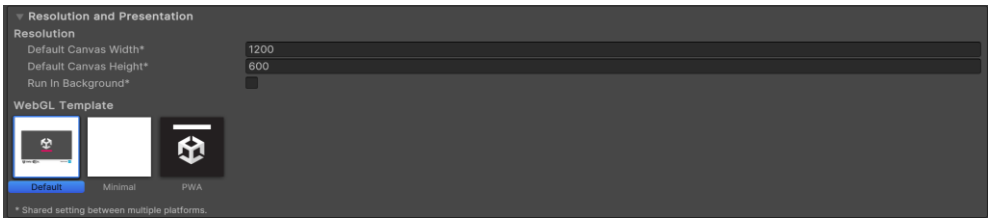


Figure 67 - The game's current resolution settings.

Resolution was set to 1200x600 on PCs as it was found to be an "acceptable" aspect-ratio/resolution for scenario photos, without stretching them too much or turning them blurry.

"**Publishing Settings**":

Figure 68 - The game's current publishing settings.

Since there is no exception catching in the current project and it can cause overhead, "Enable Exceptions" was set to "None".

Brotli compression was chosen since it offers the best compression ratios. Decompression fallback - which enables Unity to automatically embed a JavaScript decompressor into the build, at the cost of disk size and a longer startup time - was set to none. When choosing the compression format and whether to use decompression fallback or not, it is important to check if the hosting server supports them and allows for the appropriate response headers to be configured; for example, the project builds were previously hosted via FTP on a private university server, which could potentially be configured for brotli or gzip serving with no decompression fallback, but since it was impossible to change any server configuration whatsoever, I had to deploy uncompressed builds. Currently there's no such problem for the chosen hosting server.

*** For testing the Unity game on the Angular project locally, a non-deployment build must be generated with decompression fallback ON, since there is currently no response headers configured for the local Angular development server. Otherwise, the game will be stuck on the loading screen.**
For this reason, a build with decompression fallback - named "v29_for_local_testing"- is provided in the public repo containing the Angular project.

To further optimize the deployment build, "Debug Symbols" are also set to off. However, this option should be set to "Embedded" when testing builds, so potential errors are easier to understand and identify.

“**Other Settings -> Configuration**”:

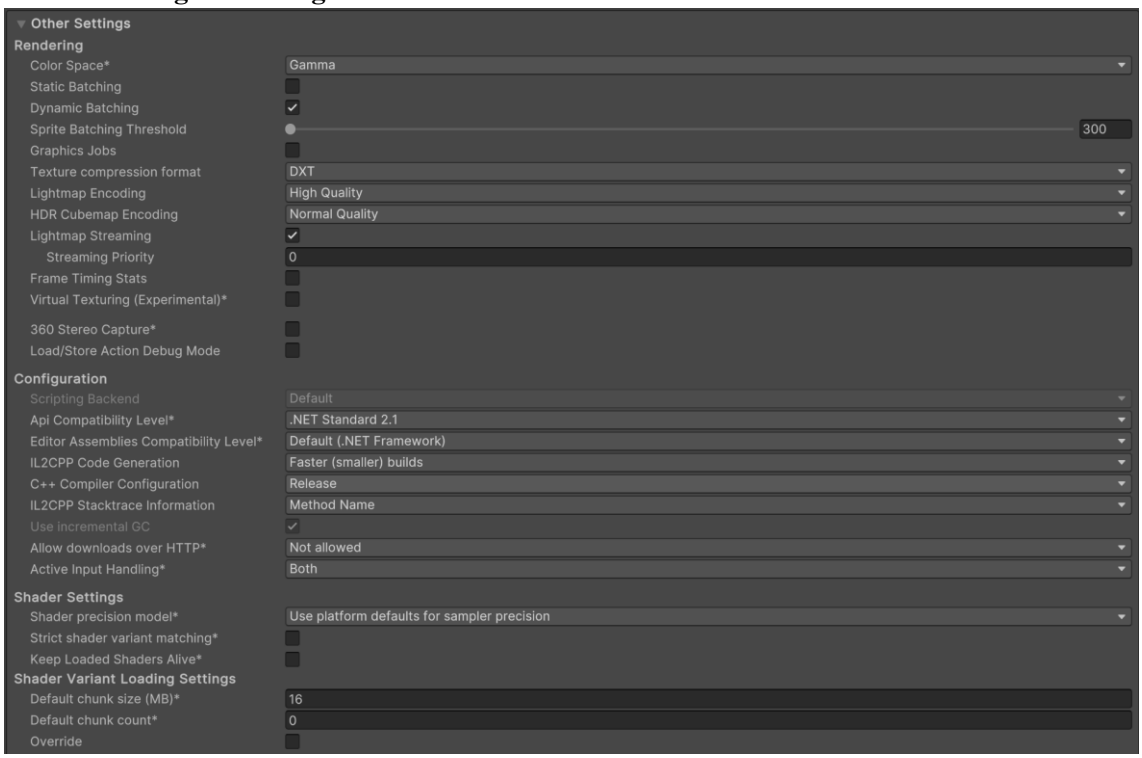


Figure 69 - Other game current settings.

“IL2CPP Code Generation” had to be set to “Faster (smaller) builds” to fix a “null function exception” thrown by Yarn Spinner issue when building for WebGL, as explained in https://github.com/YarnSpinnerTool/YarnSpinner-Unity/issues/163.

As of v2023.1.12f1, resulting builds encompass three directories: “Build”, which includes the main Unity files, such as a .wasm binary file or a javascript file responsible for loading the game build; “Streaming Assets”, which stores video file assets accessed in-game, and “TemplateData”, which holds assets related to the chosen layout template. An index.html file which defines the layout and initializes the game is also generated, but it can be disregarded as all its code has been previously adapted and embedded into the Angular project.

2. Embedding the Unity build into the Angular application

The three aforementioned directories must first be copied to the "**src/assets**" directory of the Angular project.

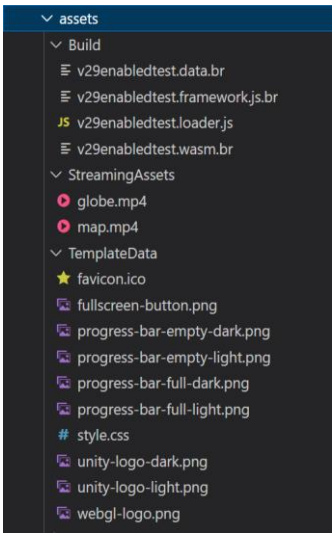


Figure 70 - The src/assets directory of the Angular project.

Then, to link the Unity build into the Angular project:

1. In **home.component.ts**, in the **ngOnInit()** method, under **config**, reference the data, framework, and wasm.files as properties **dataURL**, **frameworkURL**, and **codeURL**, respectively:

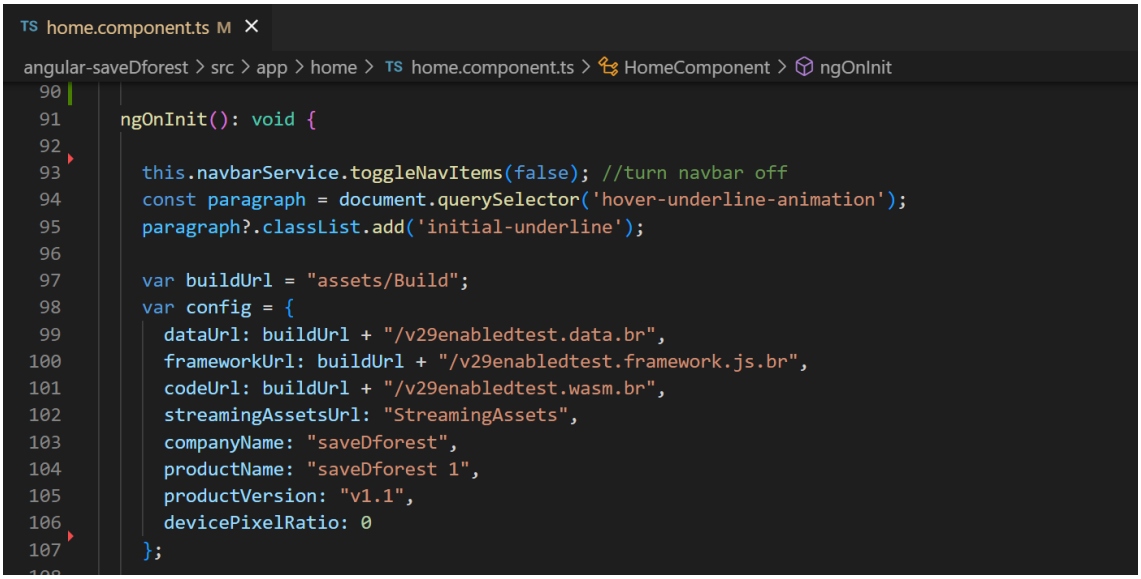


Figure 71 - Referencing different parts of the game built on the home component.

2. In **angular.json**, on **“build>options>scripts”**, reference the loader file:

Figure 72 - Referencing the loader file on the angular.json.

3. Creating the Angular build

An Angular build is created via the ***npm build*** command, which executes ***ng build****. A “**dist/savedforest**” directory will be created (or modified, if it already exists), which includes all assets and resources needed for deployment. The index.html file, which will act as an “entry point” for the build, is also created.

*if Angular CLI v13.3.10 is installed globally, the *ng build* command can be run directly; if not, *npm build* must be used.

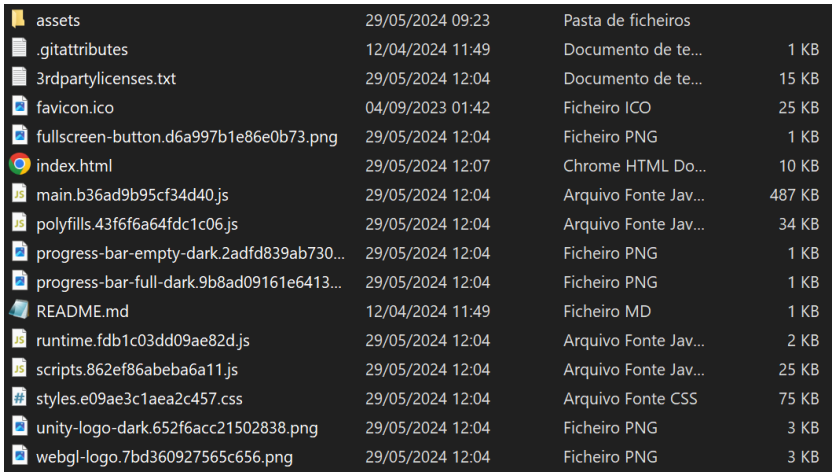


Figure 73 - The dist/savedforest directory.

4. Deploying the Angular build

The Angular build is currently hosted on Render, as a free “Static Site” instance. Render connects to a github repository where only the resulting Angular build is stored in, and deploys it, either automatically - every time there is a new push on the repo-, or manually. If the build is stored on the main branch of the repo (as it currently is), the initial Render configuration is very simple, with no publish directory or build command needed to be specified:

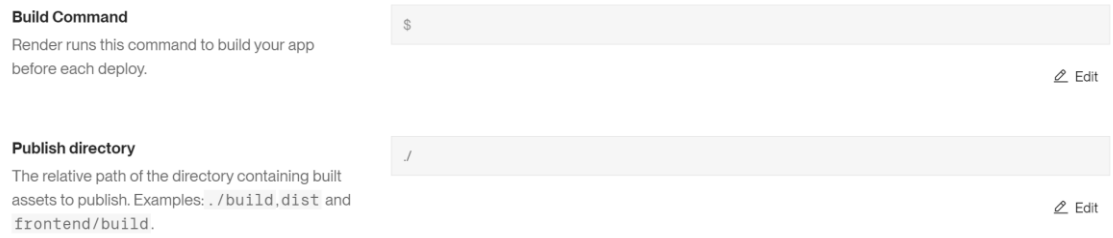


Figure 74 - The build setting on Render.com, for the Angular build.

Render prioritizes the fetching of large resources by default, which can cause smaller resources to not be fetched until the very end. If not dealt with, when users access the app and as files are being fetched, they can be faced with a blank page or a page that appears to be "misconfigured", with resources missing. To avoid this, the angular project's index.html file must be edited so that certain resources such as the app's logo ("logo.png") are fetched before the large Unity files and immediately rendered on the screen via "preloading". For the current build, the following "preloads" are inserted in the <head> element of the index.html:

```
  <meta name="viewport" content="width=device-width, initial-
scale=1">
  <link rel="preload" href="assets/treegrow.gif" as="image">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="preload" href="assets/logo.png" as="image">
  <link rel="preload" href="assets/fcul_logo_white1.png"
as="image">
  <link rel="preload" href="assets/lasige_logo.png" as="image">
  <link rel="preload" href="assets/cat_logo.png" as="image">
  <link rel="preload" href="assets/crcw_logo.png" as="image">
```

*Currently, the loading animation (assets/treegrow.gif) only needs to be rendered in the root (' ') route, but since this asset is preloaded, if the user first accesses the app through a direct link to another route (for example, by first accessing the app through the 'about' route's URL), it will still be fetched. However, this fetch is negligible, since the image is quite small size-wise (~21kb).
The same behavior applies to the logo assets if the root route is first accessed in landscape mode on a mobile device.

In Render, a rewrite rule must be configured to redirect all routing requests to the index.html file, so they can be handled by angular's routing service. Without this, if users were to access routes via direct URLs (by accessing the 'howToPlay' route via https://savedforest-temp-test-2.onrender.com/howToPlay instead of visiting the root (' ') route first and then being redirected through the navbar, for example), the request would fail, and they would be greeted with a 404 error.

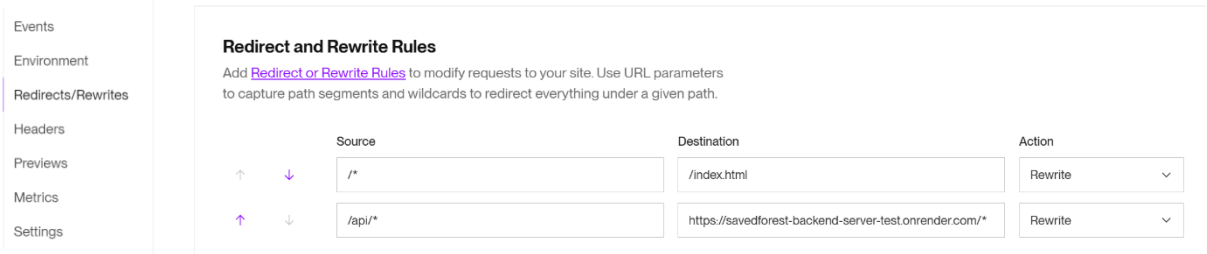


Figure 75 - The rewrite rules set on render.

A second rewrite rule has also been added to address session cookie setting; initially, users could be faced with a never-ending sequence of unauthorized access errors as their game data was sent to the back-end. This would happen because no session cookie would be set by the back-end upon signup/login, due to the browsers blocking it. Users had to manually disable third-party cookie blocking from their browsers to be able to progress in the game.
Both the front-end and back-end are being hosted in Render and thus share the same root domain (onrender.com). This domain is included in the Public Suffix List (PSL) - a list that is automatically loaded by browsers. If a root domain is on the PSL, its subdomains are treated as separate entities. Consequently, despite sharing the same root domain, the front and back-end are not considered in the same-site context, which means that the session cookie set up by the back-end in the front-end is a third-party cookie.
Third party cookies are used by advertisers for web-tracking, so some browsers block them while in incognito mode; Safari and some ad-blockers do it by default as well, and all major browsers are set to block third-party cookies by default by the end of 2024.
There have been several proposed solutions for third-party session cookies to continue to function properly, including Google's "Cookies Having Independent Partitioned State" (CHIPS) solution, which makes use of a new "partitioned" cookie attribute. This solution is already supported by the session management library I use in the back-end; however, as of August 2024, CHIPS is still not compatible with all browsers.
Another more obvious solution for solving this problem would be to acquire a custom domain and set it up so both the back-end and front-end shared it, which would effectively turn the third party-cookie into a first party one; nevertheless, I followed a simple workaround suggested by Render's support team: by setting a rewrite rule that rewrites to the back-end's endpoints, the browser can't detect that the session cookie is being set by the back-end, instead giving the illusion that the cookie is set by the front-end itself (as per official Render documentation on rewrite rules, "The browser can't detect that content was served from a different path or URL").
For this to work however, the URLs that refer to the back-end API endpoints in the services responsible for handling HTTP requests must be configured so they point to the rewrite rule's "source":

```
private sendLoginUrl = "https://savedforest-backend-server-test.onrender.com/login";
private sendSignupUrl = "https://savedforest-backend-server-test.onrender.com/signup";
private sendLogoutUrl = "https://savedforest-backend-server-test.onrender.com/logout/";
private sendResetPasswordRequestUrl = "https://savedforest-backend-server-test.onrender.com/resetPwdRequest";
private sendNewPasswordUrl = "https://savedforest-backend-server-test.onrender.com/newPwdSubmit";
```

```
private sendLoginUrl = "/api/login";
private sendSignupUrl = "/api/signup";
private sendLogoutUrl = "/api/logout/";
private sendResetPasswordRequestUrl = "/api/resetPwdRequest";
private sendNewPasswordUrl = "/api/newPwdSubmit";
```

Figure 76 - On the top, the original server endpoints.
On the bottom, the modification done to the endpoints with the rewrite rule set.

Finally, in order to correctly decompress and serve the Unity build files, several HTTP response headers must be configured, as explained in https://docs.unity3d.com/Manual/webgl-deploying.html and https://docs.unity3d.com/Manual/webgl-server-configuration-code-samples.html; This is also done through a specific tab on Render:

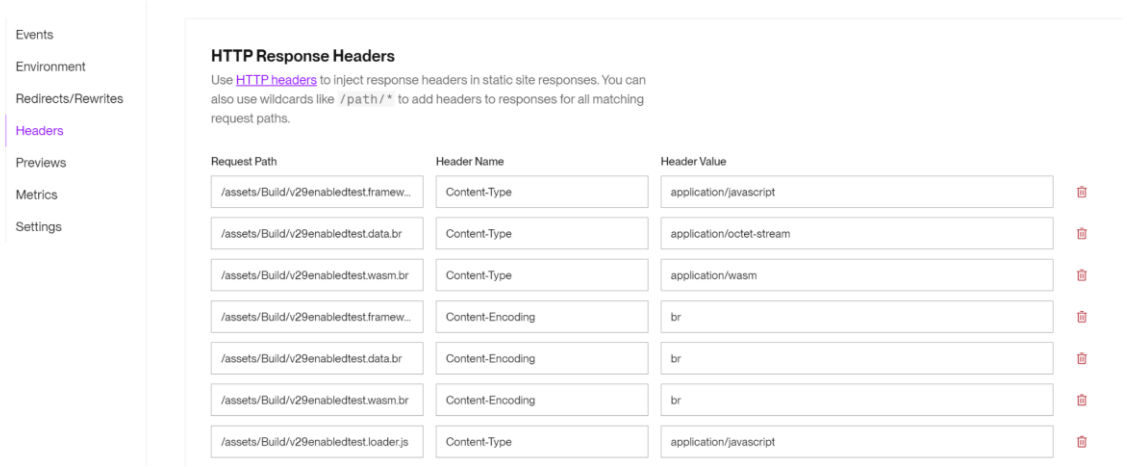


Figure 77 - The HTTP response headers set so the game is correctly served.

5. Deploying the back-end server

The back-end server is also currently being hosted on Render, as a free "Web Service" instance. The whole project (minus the .env file) is pushed to a github repository, which is connected to Render. Free Web Services spin down after 15 minutes of inactivity; as of July 2024, they can take more than 50 seconds to come back up upon receiving a request, and every now and then the server might not even restart. For this reason, timeout issuing and processing was configured on the front-end app, and a mini-game easter egg was added to the game's loading screens. The issuing of a cron job through a free platform (such as https://cron-job.org/en/) could be a potential workaround for this problem; periodic requests could be scheduled so the server would never spin down. However, this approach could contribute towards a bandwidth bottleneck, especially since it is limited in free Render instances.

The .env file, specifically configured for deployment, is added through the "Environment" tab. The project's node version also needs to be specified here, since by default Render services use a different Node version (v20.15.1 as of July 2024) than the one used on this project.

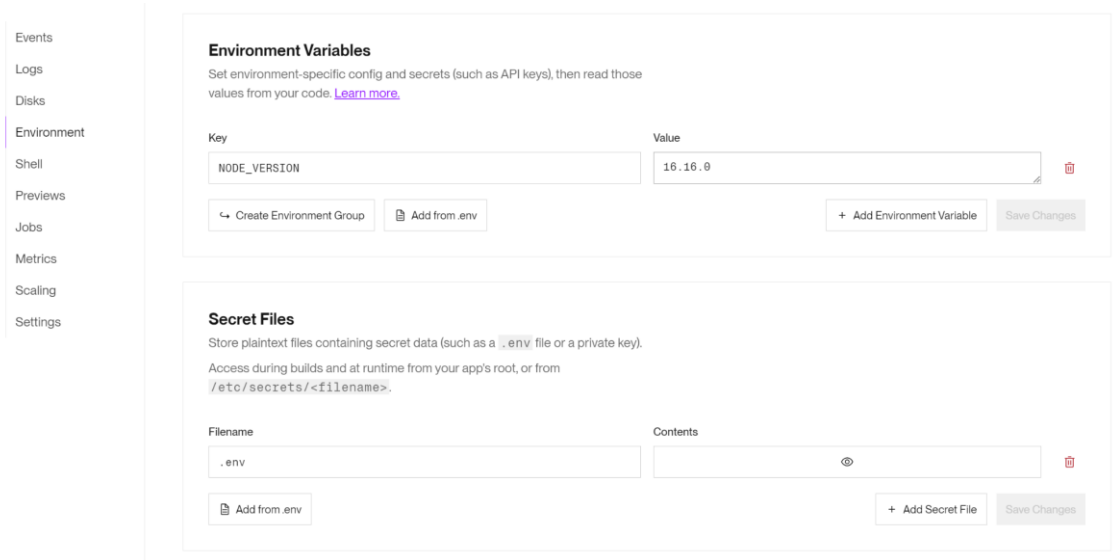


Figure 78 - The current environment tab for the back-end server.

Build and start commands are specified through the "Settings" tab. Initially, the **node_modules** directory that contains installed libraries wasn't pushed to the project's deployment github repository; all libraries were being installed through **npm install**. However, this approach led to

errors due to incorrect library versions being installed. I thus decided to include the node_modules directory in the repo. Despite this change, the bcrypt library requires reinstallation before the app is deployed:

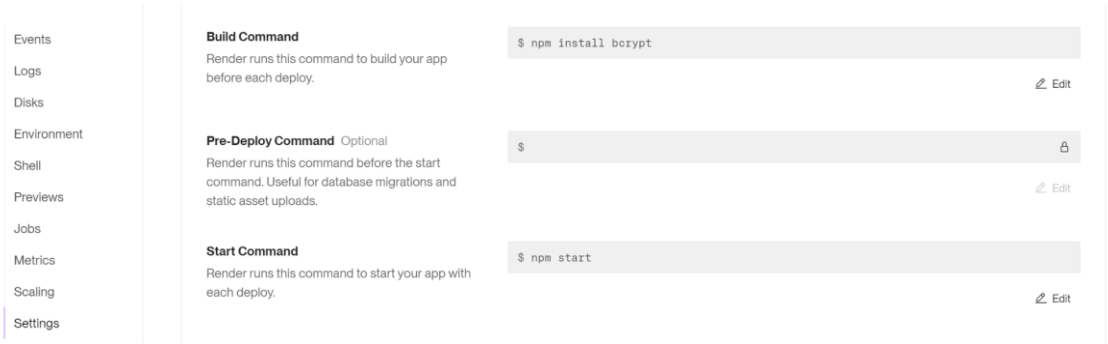


Figure 79 - The current settings tab for the back-end server.