Realisation - PhishGuard

Authors: Roko Mladinić, Daniella Namuli, Simeon Markov & Zed Minabowan

Table of contents

Introduction

This document describes the realization phase of the PhishGuard game. In this phase, we translated our game designs into a working application. We developed the technical components of the project such as the login page, admin dashboard, game structure and interactive elements. By following the design document and using the selected technologies, we were able to create a functional game.

Purpose of the project

The purpose of this project is to create an interactive and educational tool that helps non-technical users learn how to recognize phishing emails. Our goal is to build a web-based game where users can practice identifying phishing attempts in a safe way. At the start of the project, we had several ideas related to cybersecurity awareness, but we decided to focus specifically on phishing because it is one of the most common threats users faces. By narrowing our scope to phishing emails only, we could spend more time making the scenarios realistic, the feedback clear and the overall experience more enjoyable.

Project Structure

Commented [MS1]: @Markov,Simeon S. Refine this section

This section explains the architecture of our project and its usability.

- System architecture

It's a web-based application built with Nuxt (full-stack JavaScript framework) combined with the Vue framework which follows a component-based architecture. The project is structured to separate logic, interface and data handling, allowing the application to be easy to maintain and extend.

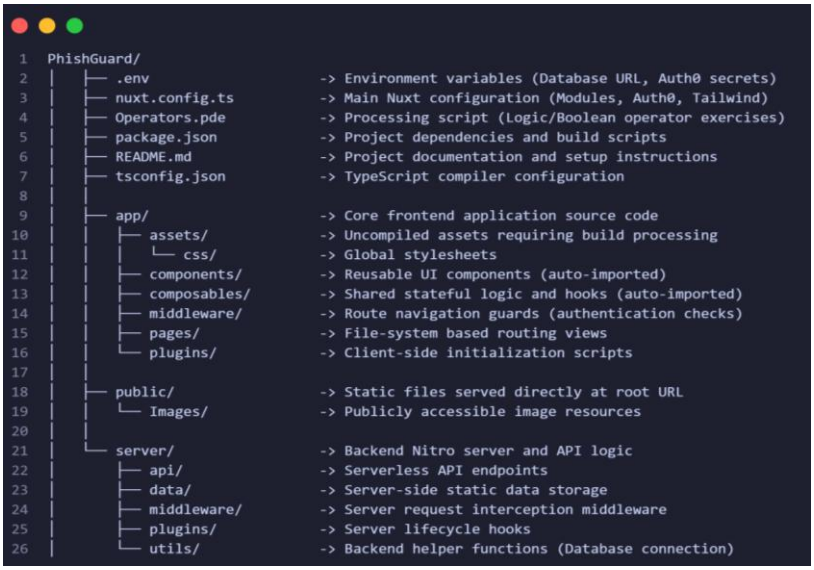
- Backend

The backend logic is handled using JavaScript and TypeScript within the Nuxt framework and application logic. Authentication is handled through external authentication provider (Auth0), which already incorporates all the security measures needed (Rate limiting, forgery tokens, input sanitization, CORS), while users data is cached to the local database, ensuring if the provider fails the users would still be able to access the application.

- Database

A PostgreSQL (relational) database is used to store user data, game sessions, phishing scenarios...

- Application's architecture



```
PhishGuard/
|   ├── .env                    -> Environment variables (Database URL, Auth0 secrets)
|   ├── nuxt.config.ts          -> Main Nuxt configuration (Modules, Auth0, Tailwind)
|   ├── Operators.pde           -> Processing script (Logic/Boolean operator exercises)
|   ├── package.json            -> Project dependencies and build scripts
|   ├── README.md               -> Project documentation and setup instructions
|   ├── tsconfig.json           -> TypeScript compiler configuration
|   |
|   ├── app/                    -> Core frontend application source code
|   |   ├── assets/             -> Uncompiled assets requiring build processing
|   |   |   └── css/            -> Global stylesheets
|   |   ├── components/         -> Reusable UI components (auto-imported)
|   |   ├── composables/        -> Shared stateful logic and hooks (auto-imported)
|   |   ├── middleware/         -> Route navigation guards (authentication checks)
|   |   ├── pages/              -> File-system based routing views
|   |   └── plugins/            -> Client-side initialization scripts
|   |
|   ├── public/                 -> Static files served directly at root URL
|   |   └── Images/             -> Publicly accessible image resources
|   |
|   └── server/                 -> Backend Nitro server and API logic
|       ├── api/                -> Serverless API endpoints
|       ├── data/               -> Server-side static data storage
|       ├── middleware/         -> Server request interception middleware
|       ├── plugins/            -> Server lifecycle hooks
|       └── utils/              -> Backend helper functions (Database connection)
```

- Frontend

The frontend is built using HTML, Vue and JavaScript, with styling handled through CSS & Tailwind CSS. Vue components are used to create reusable interface elements such as the login page and game screens and that ensures a consistent user experience.

- Development Environment

The project is developed using Visual Studio Code, which supports debugging and extensions for working efficiently with Nuxt, TypeScript and Vue.

Design vs Realisation

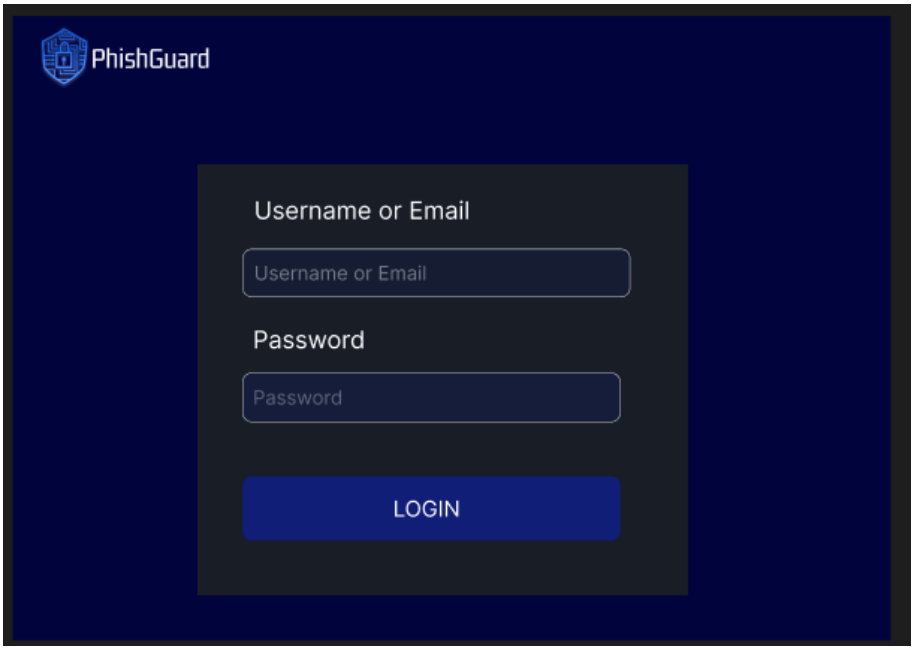
When we look back at our design phase, we see that we've made some changes on the way we designed our pages compared to our final coded version of our pages. In this section, we'll show the original and final versions of our pages and explain what has changed since our initial design.

- **Login Page:**

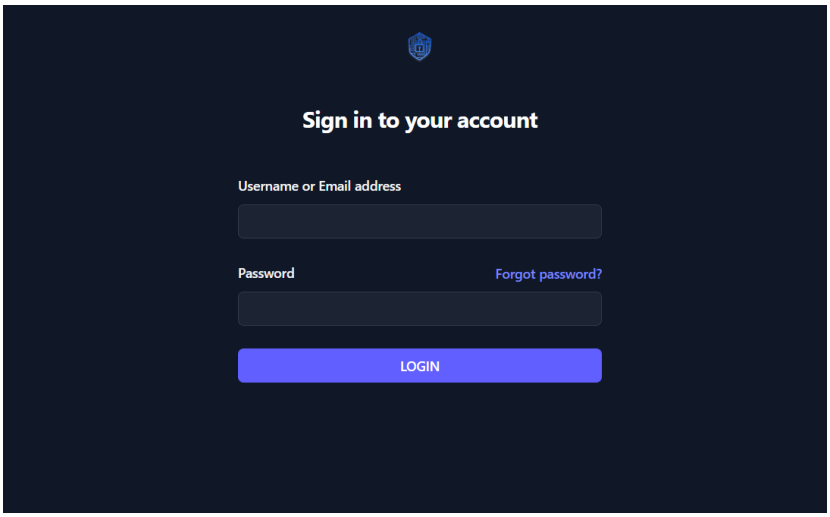
The final Login page does not differ a lot from the initial Figma design. The main difference is that "Sign in to your account" was added to the final design and our game logo was added in the upper middle of the login page. This makes the login page for professional.

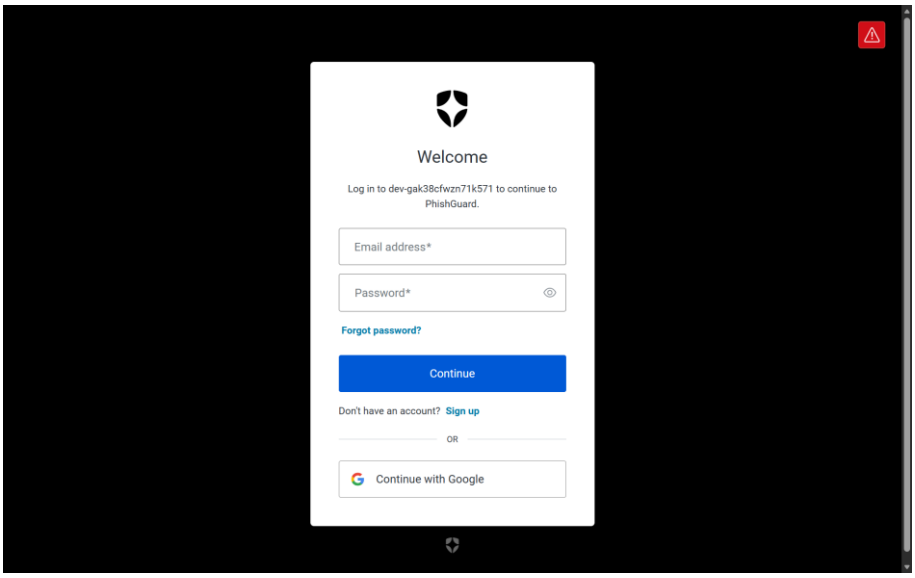
The user has the options to login through the authentication provider and through a custom login view. The reason behind this is that if the external authentication fails, the user data is cached locally to the database.

Our initial figma login page design:



Our final login page designs:



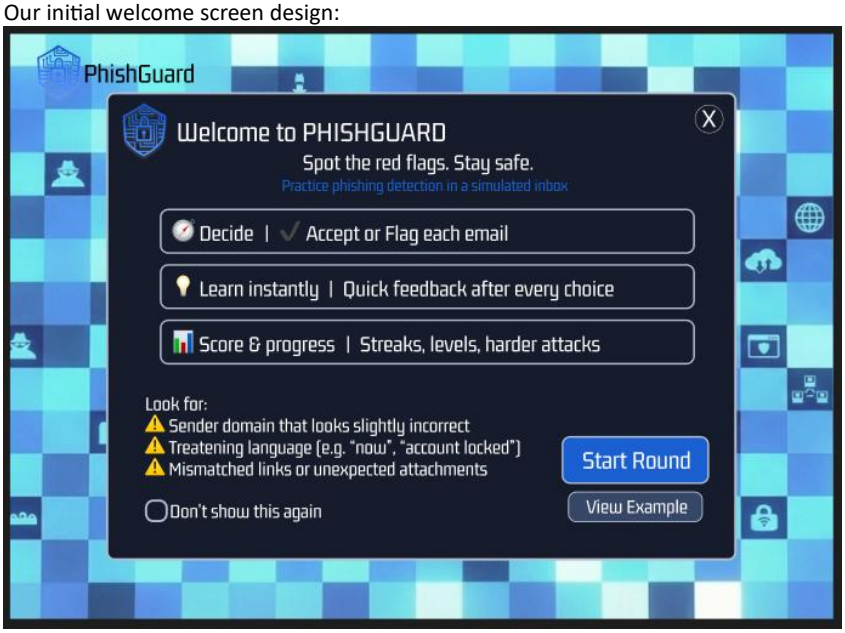


The user has the option to login through the authentication provider and custom login view. The reason behind this is that if the external authentication fails, the user data is cached locally to the database.

- **Welcome screen**

Our final welcome screen has been improved since our original Figma design. We improved the visual aspect of this screen and didn't change anything for the informative parts of it.

Our initial welcome screen design:

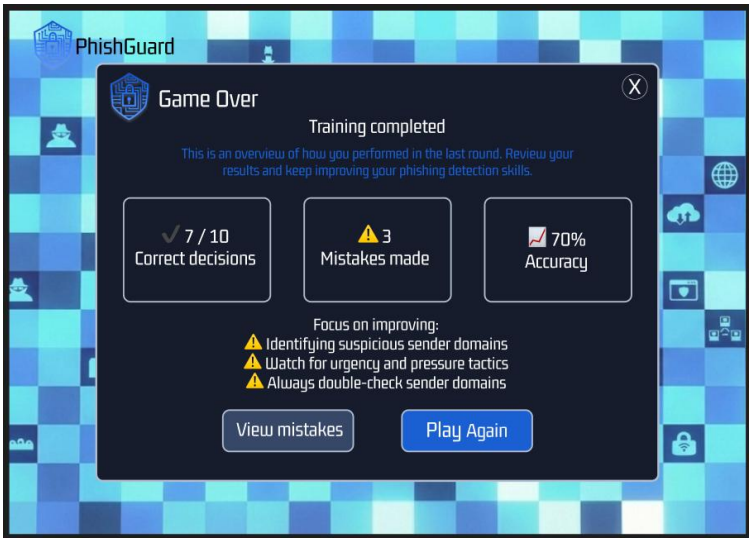


Our final welcome screen design:

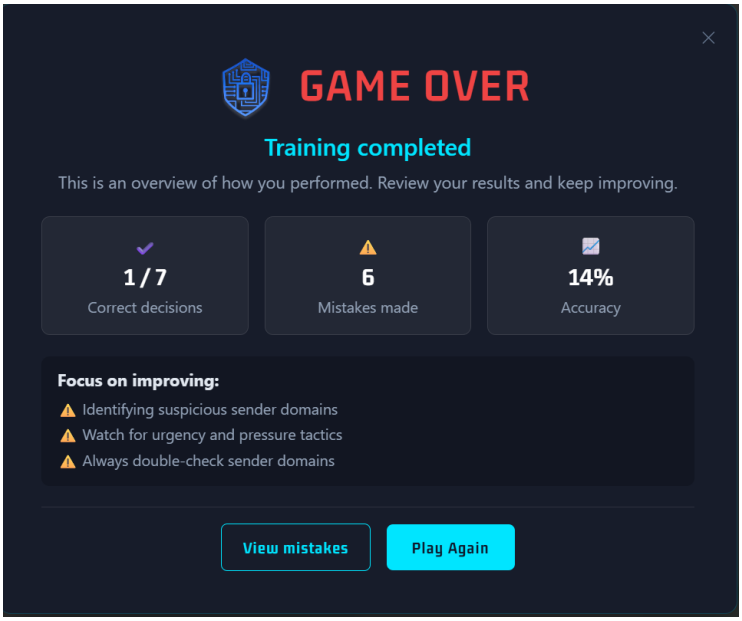
- **Game over screen**

Our game over screen has also been updated and improved. Once again, we only improved the visual aspects of this page to make it look better. The information we originally planned to put on this screen stayed the same.

Our initial figma game over screen design:



Our final game over screen design:



- **Database**

One additional ‘users ’table was added for caching the user data from the external auth provider in case it fails.

Column	Data Type	Constrains	Description
id	INT	Primary Key	Unique identifier for each record
email	CHARS	Unique	Users fetched email from auth provider
name	CHARS	NOT NULL	Users fetched name from auth provider
picture	CHARS	NULLABLE	User’s picture
authid	CHARS	NOT NULL	User’s id fetched from auth provider
last_login	TIMESTAMP	NOT NULL	Last login of user
Created_at	TIMESTAMP	NOT NULL	When the user registered
Password_hash	CHARS	NOT NULL, UNIQUE	User’s password hashed

Code documentation

Commented [MS2]: @Markov,Simeon S.

'game_screen': Contains the game's logic and all the interactions with the player. Main content of that file includes:

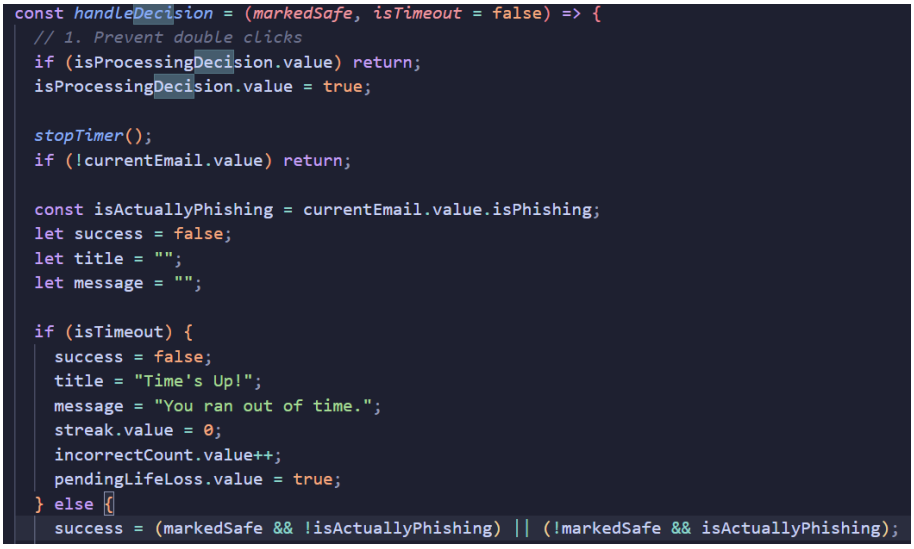
```
// Game Stats
const roundNumber = ref(1);
const lives = ref(5);
const correctCount = ref(0);
const incorrectCount = ref(0);
const streak = ref(0);
const timeLeft = ref(30);
let timerInterval = null;
const feedback = ref(null);
const loading = ref(false);
const error = ref(null);
```

Snapshot of the initialized constants and variables, storing information like lives of the user, remaining time, mistakes...

```
// Fetch scenarios from API
const fetchScenarios = async () => {
  loading.value = true;
  error.value = null;
  try {
    const response = await $fetch('/api/scenarios', {
      query: {
        difficulty: roundNumber.value,
        limit: 10
      }
    });

    if (response.success && response.data) {
      emails.value = response.data.map(email => ({
        ...email,
        redFlags: email.redFlags || [],
        timeLeft: 30, // Initialize individual timer for each email
      }));
      if (emails.value.length > 0) {
        selectedEmailId.value = emails.value[0].id;
      }
    }
  } catch (err) {
    console.error('Failed to fetch scenarios:', err);
    error.value = 'Failed to load emails. Please try again later.';
  } finally {
    loading.value = false;
  }
};
```

Snapshot of the fetching function. It loads the phishing scenarios from a .json file, containing all the information needed per scenario by limiting the number of them per game session. Then it returns the fetched data by setting an individual timer for each scenario. If the operation fails, display an error message to the user.



```
const handleDecision = (markedSafe, isTimeout = false) => {
  // 1. Prevent double clicks
  if (isProcessingDecision.value) return;
  isProcessingDecision.value = true;

  stopTimer();
  if (!currentEmail.value) return;

  const isActuallyPhishing = currentEmail.value.isPhishing;
  let success = false;
  let title = "";
  let message = "";

  if (isTimeout) {
    success = false;
    title = "Time's Up!";
    message = "You ran out of time.";
    streak.value = 0;
    incorrectCount.value++;
    pendingLifeLoss.value = true;
  } else {
    success = (markedSafe && !isActuallyPhishing) || (!markedSafe && isActuallyPhishing);
```

Snapshot of the function **handleDecision()** that handles the game validation (right, wrong answer, time up checkup). The logic is as follows: If the time is up, a message is displayed to the players, notifying them that they ran out of time for this phishing scenario.

```
  if (success) {
    title = "Correct!";
    message = markedSafe ? "Good catch. This email is safe." : "Well done! You spotted the phishing
    streak.value++;
    correctCount.value++;
    pendingLifeLoss.value = false;
  } else {
    title = "Incorrect";
    message = currentEmail.value.educationalMessage || "You missed the signs.";
    streak.value = 0;
    incorrectCount.value++;
    pendingLifeLoss.value = true;
  }
}

feedback.value = { visible: true, isCorrect: success, title, message };
```

Snapshot of the function **handleDecision()** (the previous 2 images come from the same file). If the value of the variable success returns true, the player got the phishing email correctly; he/she will get one point and will receive a success message. If the player gets the scenario wrong, the success variable returns false, incorrect count increases by 1 and one life is deducted from the player with an incorrect message being displayed.

Server / API

seed.ts: The database seeding script.

```
export default defineEventHandler(async (event) => {
    const pool = useDb();
    const client = await pool.connect();

    try {
        await client.query('BEGIN');

        const schema = '"phish-guard"';
        const atTable = '"attacktype"';
        const psTable = '"phishingscenario"';
        const siTable = '"suspiciousindicator"';
        const uTable = '"users"';

        console.log(`Using schema: ${schema}`);


        await client.query(`
            CREATE TABLE IF NOT EXISTS ${schema}.${uTable} (
                "id" SERIAL PRIMARY KEY,
                "email" varchar(255) UNIQUE NOT NULL,
                "name" varchar(255),
                "picture" varchar(500),
                "auth_id" varchar(255) UNIQUE,
                "last_login" TIMESTAMP DEFAULT NOW(),
                "created_at" TIMESTAMP DEFAULT NOW()
            );
        `);
```

This snapshot contains images of overwriting event handler function, where a database connection is established, then a query is started to first check existing tables and then:

```
// 2. LOAD SCENARIOS FROM JSON
const scenariosPath = path.resolve(process.cwd(), 'server/data/scenarios.json');
console.log(`Loading scenarios from: ${scenariosPath}`);

const scenariosContent = fs.readFileSync(scenariosPath, 'utf-8');
const scenarios = JSON.parse(scenariosContent);

for (const scenario of scenarios) {
    const attackTypeId = attackTypeMap.get(scenario.type);
    if (!attackTypeId) {
        console.warn(`Attack type not found: ${scenario.type} for scenario ${scenario.id}`);
        continue;
    }
```

Load the data (available scenarios from the .json file) then iterate over the content mapping the type of the phishing scenario.

```
// UPSERT PhishingScenario
const queryText = `
    INSERT INTO ${schema}.${psTable}
    ("attacktypeid", "attackbody", "difficultylevel",
     "sender", "senderemail", "initials", "subject", "preview",
     "educationalmessage", "hint", "isphishing", "externalid", "timestamp",
     "attackcontext", "attackquestion", "answer", "custom_links")
    VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, NOW(), $13, $14, $15, $16)
    ON CONFLICT ("externalid") DO UPDATE SET
        "attacktypeid" = EXCLUDED."attacktypeid",
        "attackbody" = EXCLUDED."attackbody",
        "difficultylevel" = EXCLUDED."difficultylevel",
        "sender" = EXCLUDED."sender",
        "senderemail" = EXCLUDED."senderemail",
        "initials" = EXCLUDED."initials",
        "subject" = EXCLUDED."subject",
        "preview" = EXCLUDED."preview",
        "educationalmessage" = EXCLUDED."educationalmessage",
        "hint" = EXCLUDED."hint",
        "isphishing" = EXCLUDED."isphishing",
        "attackcontext" = EXCLUDED."attackcontext",
        "attackquestion" = EXCLUDED."attackquestion",
        "answer" = EXCLUDED."answer",
        "custom_links" = EXCLUDED."custom_links"
    RETURNING "id"`;
```

And finally, the snapshot above shows how the data is being inserted into the database with all the necessary columns.

Conclusion

In this realisation phase, the planned design and ideas were successfully turned into a working game with the help of the chosen technologies. This phase showed how the wireframes were applied in practice and how different technologies worked together to create an interactive game. With the core functionality implemented, the project achieved its goal of providing users with an engaging and educational experience.