Skills Project

Sistemas Web - práctica

Aingeru García Blas
Joseba Gómez Rodriguez

Índice

1. Fase 1

1 Estructura del proyecto

```
❯ tree -I ".vscode|bin|node_modules|icons/|badges/"
.
├── LICENSE
├── README.md
├── app.js
├── config
│   └── config.js
├── doc
│   └── Enunciado.pdf
├── package-lock.json
├── package.json
├── public
│   ├── css
│   │   ├── leaderboard.css
│   │   ├── skillTemplate.css
│   │   └── style.css
│   ├── data
│   │   ├── badges.js
│   │   ├── badges.json
│   │   ├── skills.js
│   │   └── skills.json
│   ├── electronics
│   ├── images
│   ├── index.html
│   ├── js
│   │   ├── confetti.js
│   │   ├── leaderboard.js
│   │   ├── main.js
│   │   └── skillTemplate.js
│   ├── leaderboard.html
│   └── skill-template.html
├── routes
│   ├── index.js
│   └── users.js
├── scripts
│   ├── ImageDownloader.js
│   ├── download_badges.js
│   ├── download_icons.js
│   ├── harvesters
│   │   └── ImageHarvester.js
│   └── scrapers
│       ├── BadgesScrapper.js
│       ├── Scraper.js
│       └── SkillScraper.js
└── views
    ├── error.ejs
    └── index.ejs

14 directories, 32 files
```

Hemos organizado el proyecto en varias carpetas y módulos básicos para para realizar las tareas de scraping, recolección (harvesting) y exportación de datos, así como la lógica del Frontend para mostrar lo requerido por el enunciado. Veamos varios de los componentes clave:

config/

La carpeta `config/` tiene los archivos de configuración (uno por ahora, pero hemos pensado en la siguiente entrega), donde definimos rutas, endpoints y otros parámetros que utilizan los módulos de scraping y harvesting.

---

scripts/

scrapers/

Esta carpeta contiene el conjunto de clases que hemos diseñado para realizar tareas de scraping utilizando `cheerio`. La clase base es `Scraper`, de la cual heredan otras clases especializadas. Cada scraper tiene un enfoque particular para extraer datos específicos según el contexto. Por ejemplo, `SkillScraper` hereda de `Scraper` y extrae información sobre skills (como el texto, iconos asociados, etc.).

- Clase `Scraper`: se encarga de establecer la conexión al endpoint y de cargar el HTML. Utiliza librería `cheerio` como scraper.
- Clase `SkillScraper`: extiende `Scraper` para procesar datos específicos de skills. Implementa métodos como `harvestSkills`, que sirve para identificar y extraer datos de los elementos SVG (ID, texto, y links a iconos...etc). También hemos provisto de funciones para exportar las habilidades en formato JSON.
- Clase `BadgeScraper` Misma idea que SkillScraper, pero para badges. Con su export, capacidad de recolectar/harvestear de lo proporcionado por el scraper etc.

---

harvesters/

La carpeta `harvesters/` contiene clases que funcionan como recolectores de datos ya scrapeados. Por ejemplo:

- Clase `ImageHarvester`: se centra en la descarga de imágenes desde los endpoints, con URLs generadas en función de los datos scrapeados. Básicamente se encarga de almacenar los iconos en el directorio que corresponda (múltiples imágenes en paralelo). La parte interesante es que es configurable, y se adapta al scraper que le pasemos.

`Downloaders:`

- `download_icons.js`: Configura y ejecuta el flujo de scraping y descarga de iconos.
- `download_badges.js`: Lo mismo, pero para las badges.

Ejemplo de flujo de ejecución

Para un proceso de recolección/harvesteo de iconos, tendremos el siguiente flujo:

1. `ImageHarvester` se inicializa con el endpoint de iconos.
2. `setupScraper` configura `SkillScraper` para obtener las skills.
3. Por último, el método `harvestImages` descarga los iconos (en ruta especificada en `config`).

Documentos HTML

- `index.html` - Página de inicio y visualización de habilidades.
- `leaderboard.html` - Página que presenta una tabla de rangos.
- `skill-template.html` - Página de plantilla para mostrar información algo más detallada de los skills.

`index.html`

Árbol de skills principal, con elementos SVG, que generamos dinámicamente desde `main.js`. Su núcleo es un contenedor donde presentamos los SVG y una caja de descripción para visualizar información sobre cada skill cuando el usuario pasa el cursor sobre ella.

`leaderboard.html`

`leaderboard.html` muestra una tabla con explicaciones de rangos que hemos scrapeado de un `README.md` de la cuenta de Github del gran Obijuan, se muestran datos de rangos, puntos, iconos distintivos etc. Cada fila incluye información sobre el rango, el iconos, por ejemplo. Los datos de esta tabla los generamos dinámicamente en `leaderboard.js`.

skill-template.html

skill-template.html es una plantilla para visualizar detalles de cada skill individual. Muestra información de una skill específica; su descripción, icono, lista de tareas y recursos. También se permite que el usuario envíe evidencias de cumplimiento de la skill. El archivo incluye:

- skillTemplate.js para generar dinámicamente los contenidos.
- confetti.js efectos visuales de confeti cuando todas las tareas de una habilidad están completadas.

Archivos JavaScript

- main.js - Renderizado y control de skills en index.html.
- leaderboard.js - Generación de la tabla de rangos en leaderboard.html.
- skillTemplate.js - Tareas y evidencias en skill-template.html.
- confetti.js - Efecto visual.

main.js

Creación y visualización de las skills dentro de index.html. Al cargarse, monta los elementos SVG en base a los datos de skills (scrapeados + harvest) y aplica colores específicos (en función del progreso, etc.). Funciones principales:

- buildSkills - Crea el contenido SVG dinámicamente.
- applyHexagonColors - Aplica colores a cada skill según el estado almacenado en localStorage.

leaderboard.js

Maneja la carga y generación dinámica de la tabla de rangos en leaderboard.html. Extrae datos de los rangos y crea filas para mostrar los iconos, nombres y rangos de puntos (intervalos puestos aleatoriamente, por ahora). Función clave:

- loadBadgeTable - Genera cada fila en la tabla (a partir de los datos de rangos).

skillTemplate.js

skillTemplate.js se encarga de gestionar las tareas y la presentación de una skill en skill-template.html. Genera la interfaz donde los usuarios pueden ver tareas, agregar evidencia, etc. Funciones destacadas:

- renderSkillTemplate - genera la interfaz de la habilidad a partir de su ID y datos.
- handleEvidenceSubmit - enviar evidencia de cumplimiento.
- updateHexagonColor - Cambia el color del hexágono de la skill.

`confetti.js`

`confetti.js` implementa efecto de confeti cuando un usuario completa todas las tareas de una skill en `skill-template.html`.

1 Sobre la integración completa de datos

Debido a la falta de acceso a datos durante el proceso de scraping, hemos configurado el frontend del proyecto de manera que, cuando estos datos estén disponibles y contemos con una gestión de datos más granular y segura desde el backend, podamos integrar la lógica completa de manera adecuada. Ahora mismo, algunos elementos del proyecto están `hardcodeados` en esta entrega para simular la funcionalidad y permitir el desarrollo y visualización básica. Ya que en el enunciado aparecen varios de ellos y, sin embargo, aún no es posible gestionarlo.

2. Fase 2

2 Estructura - adiciones

2 Backend

Como se puede observar, la mayor parte del trabajo se ha centrado en el **backend**. Hemos dejado lo relacionado a la primera fase, aunque en algunos casos es han hecho algunos cambios. Hemos incluido:

- **Modelos**: Hemos implementado los modelos principales para gestionar los recursos que servimos a través del API REST. Utilizamos `Mongoose` como interfaz ORM.
- **Servicios de autenticación**: Un servicio muy simplista dedicado a manejar la autenticación de usuarios. También utilizamos sesiones de `express`.
- **Rutas**: La lógica se ha programado directamente en los `routers`, a pesar de ser conscientes que la lógica de negocio debería residir en servicios o controladores independientes. Esto se debe a **falta de tiempo**.

También hemos añadido las **views** requeridas, e incluído alguna más que considerábamos correspondía (por ejemplo, `navbar`). Se ha hecho más cómodo trabajar con views que durante la primera fase, aunque EJS esté relativamente obsoleto ya que hoy en día es casi todo renderizado en el cliente (frameworks como REACT, Angular etc), se hace más ameno y organizado que únicamente con HTML y JS.

Mencionar también, que en algunos casos, para evitar desarrollar sistemas de soporte que se requerían para algunas features, hemos decidido **extender las APIs** de ciertos recursos, para poder admitir requests desde el cliente. Lo hemos utilizado para casos concretos y evitar tener que rehacer todo de cero, con lo que este enfoque no ha sido común y la mayor parte del desarrollo se ha realizado mediante la **sinergia entre views, routers y modelos**.

Hemos hecho un **diseño responsivo** en la mayor parte del proyecot y nos habría gustado hacer una entrega dockerizada, al menos con algo básico para poder expresar bien la diferencia entre frontend y backend. Es decir, por un lado un contenedor con **NGINX** que simplemente sirva los archivos estáticos y, que estos, desde el cliente y mediante JS se comuniquen con el `endpoint` oportuno del API del backend, el cual estaría en otro contenedor separado. Pero, como se ha hecho mención anteriormente, no nos ha sido posible por falta de tiempo.

Sistema de migración de datos

Hemos desarrollado un pequeño sistema de **migración de datos** que se ejecuta automáticamente si la base de datos está vacía. Este sistema **puebla la base de datos** con información inicial, la idea es disponer de un setup rápido, tanto para nosotros como para la persona que lo tenga que probar.

La información inicial de los usuarios está almacenada en el archivo `db/data/users.json`.

2 Breve especificación de la API

En este documento, no vamos a entrar en detalle en la implementación concreta de cada uno de los métodos ni en el código maquetado en EJS, existe algún caso que podría destacar más que otros, pero consideramos que el peso de ésta fase lo tienen la implementación de los recursos del API, los cuales son bastante autoexplicativos (en cuanto a concepto).

Hemos incluido algún `endpoint` extra, con lo que listamos brevemente la forma final del API del proyecto.

`Admin`

- **GET /dashboard**: Renderiza el panel de administración con el nombre del usuario.
- **GET /badges**: Muestra todas las insignias ordenadas por `bitpoints_min`.
- **GET /badges/edit/:id**: Renderiza la página para editar una insignia específica.
- **POST /badges/edit**: Actualiza o elimina una insignia según la acción recibida.
- **POST /badges/add**: Agrega una nueva insignia con los datos proporcionados.
- **POST /badges/delete/:id**: Elimina una insignia específica.
- **GET /users**: Muestra una lista de usuarios con sus roles.
- **POST /change-password**: Cambia la contraseña de un usuario específico.

`Skills`

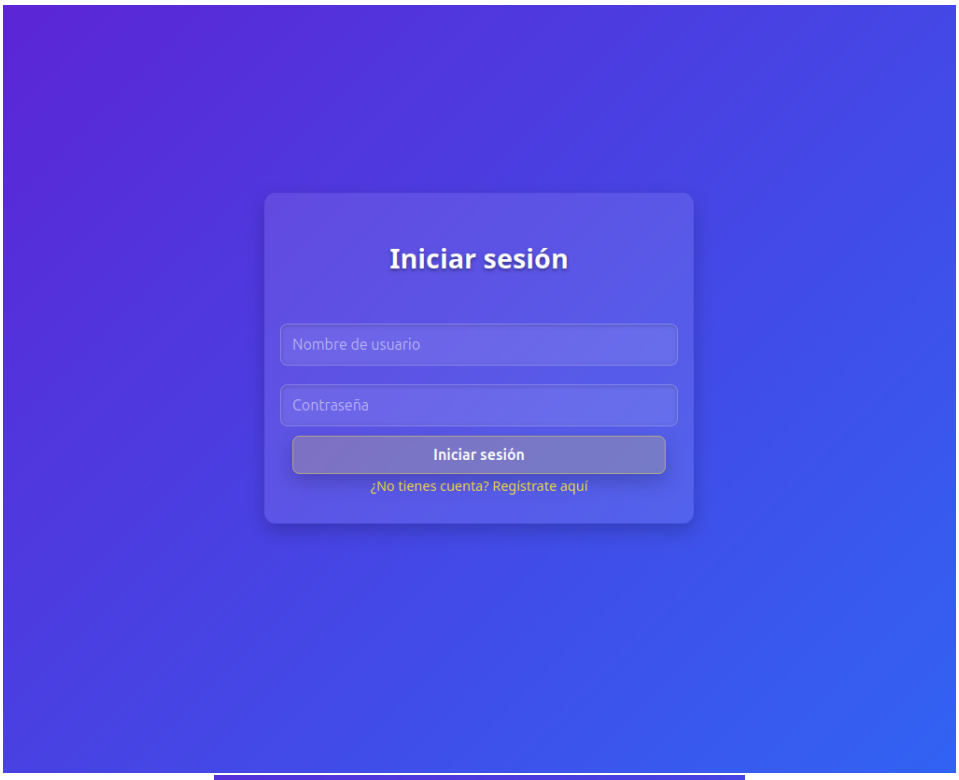
- **GET** /: Redirige al árbol de skill `electronics` (por defecto).
- **GET /:skillTreeName**: Renderiza el árbol de skill con *notificaciones* del usuario en skills.
- **GET /:skillTreeName/add**: Render del formulario para agregar una skill.
- **GET /:skillTreeName/all**: Devuelve el skill y sus evidencias relacionadas.
- **GET /:skillID/get-by-id**: Obtiene una skill por su `ID` y si está completada por el usuario.
- **GET /:skillTreeName/view**: Renderiza una skill específica y sus evidencias no verificadas.
- **GET /:skillTreeName/edit/:skillID**: Renderiza el formulario para editar una skill.
- **GET /:skillId/evidences**: Devuelve las evidencias relacionadas con una skill.
- **POST /:skillTreeName/add**: Agrega una nueva skill con los datos proporcionados.
- **POST /evidence/:evidenceId/verify**: Verifica una evidencia y la marca como completada si corresponde.
- **POST /:skillTreeName/:skillId/submit-evidence**: Envía una evidencia para una skill específica.
- **POST /:skillTreeName/delete/:skillID**: Elimina una skill específica.
- **POST /:skillTreeName/edit/:skillID**: Actualiza una skill específica.
- **POST /evidence/:evidenceId/approve**: Aprueba y marca una evidencia como completada.
- **POST /evidence/:evidenceId/reject**: Rechaza y elimina una evidencia.

Users

- **GET /login**: Renderiza la página de inicio de sesión.
- **GET /register**: Renderiza la página de registro.
- **GET /dashboard**: Renderiza el panel principal del usuario autenticado.
- **GET /logout**: Cierra la sesión del usuario.
- **GET /leaderboard**: Renderiza el ranking de usuarios por puntuación.
- **POST /login**: Inicia sesión con las credenciales proporcionadas.
- **POST /register**: Registra un nuevo usuario con las credenciales proporcionadas.

2 Showcases

A modo de showcases, veamos algunas imágenes del resultado final:



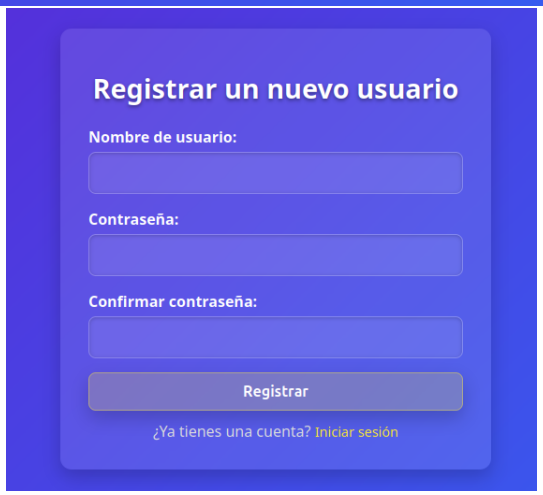


Figura 1: Login y registro de usuarios

Figura 2: Vista de usuario de todas las skills

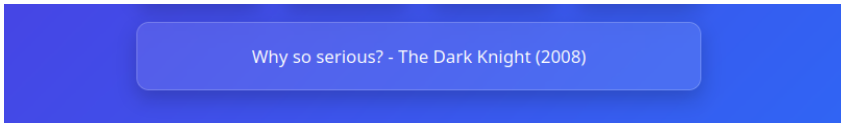


Figura 3: Descriptionbox al pasar el cursor por una skill

Figura 4: Responsivo + fixed menu (navbar) al scrollear.

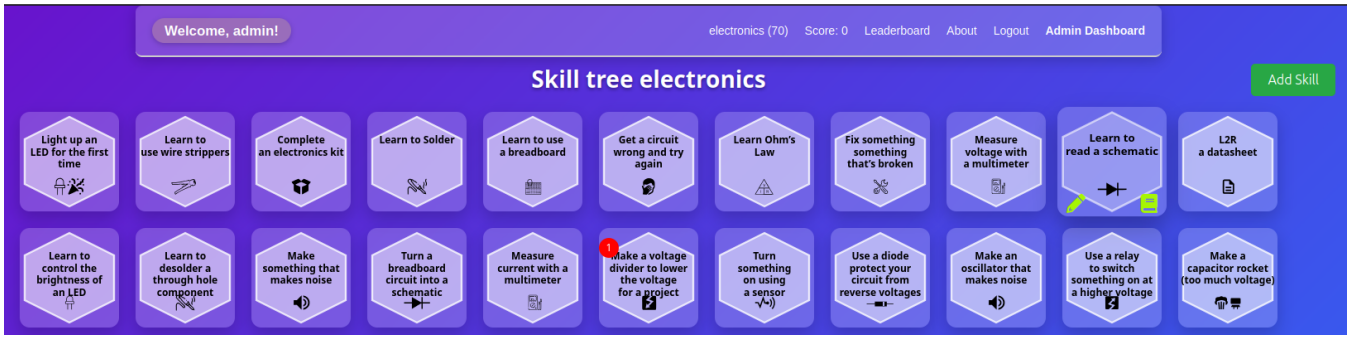


Figura 5: Vista de admin de las skills (no se ve el cursor en el screenshot)

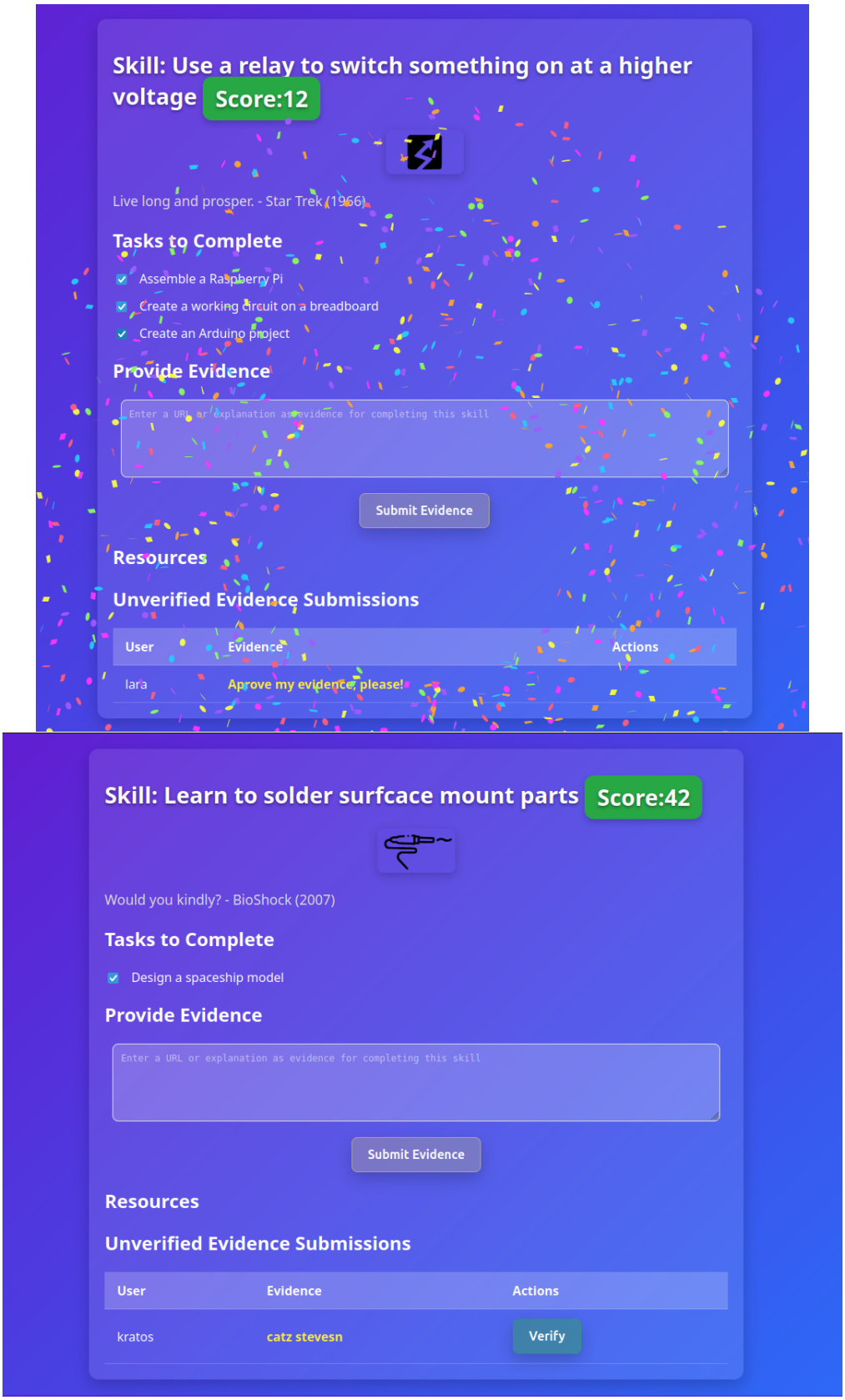


Figura 6: Vista skill con confetti - skill user view

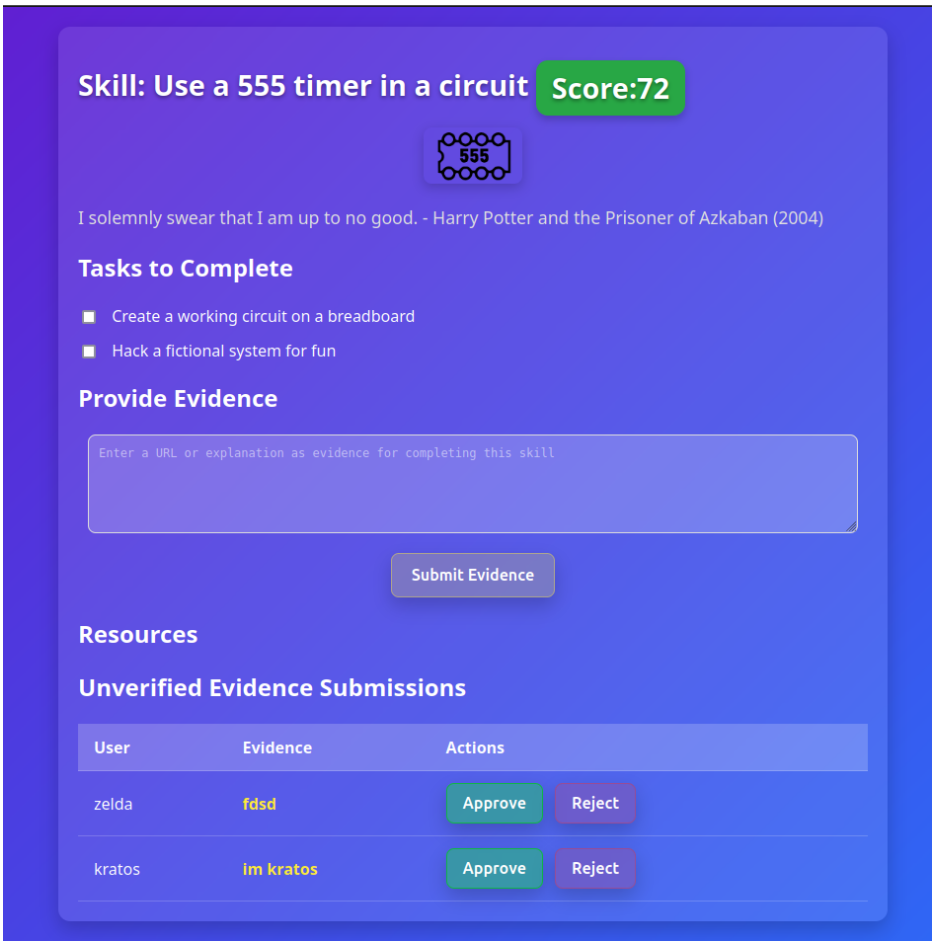


Figura 7: Vista de admin skill

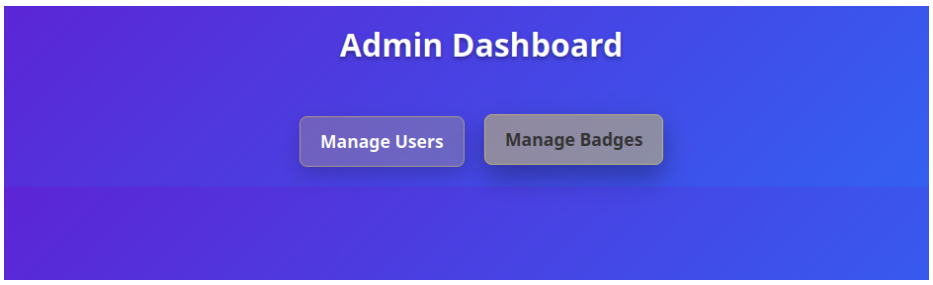


Figura 8: Admin dashboard

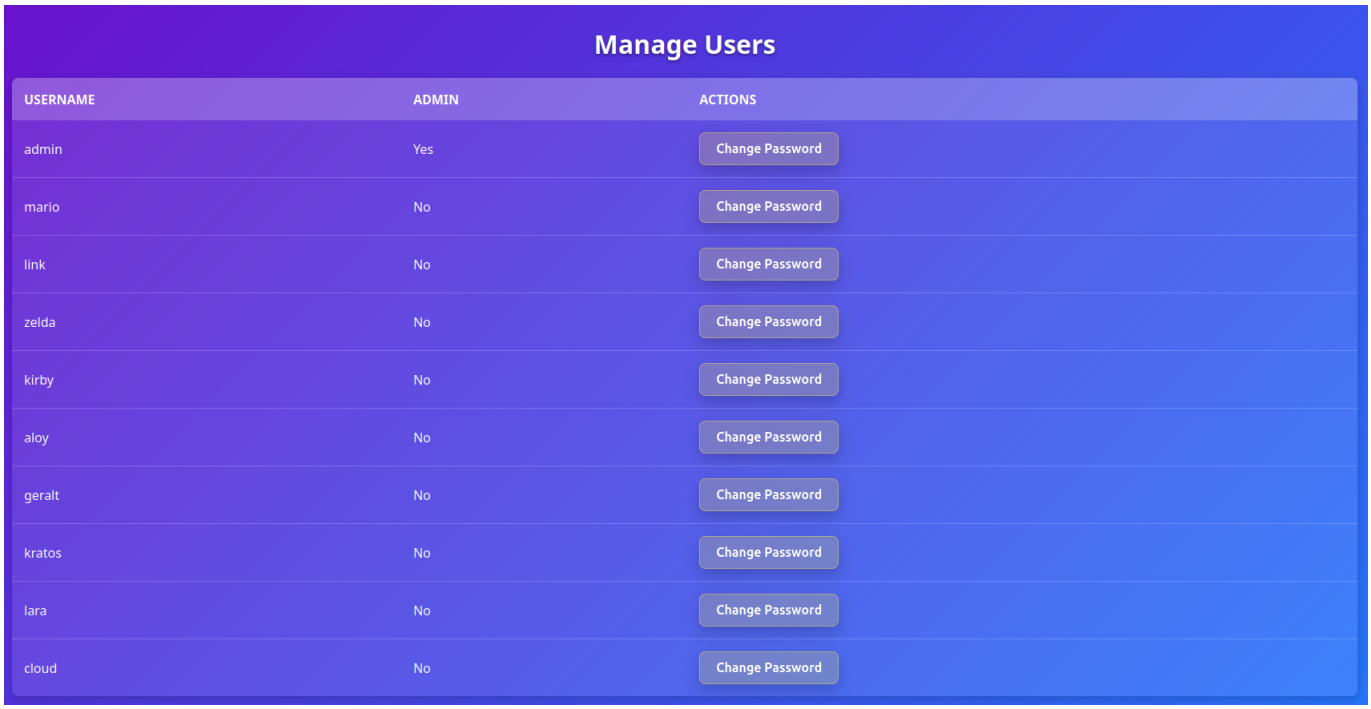


Figura 9: Admin - manage users

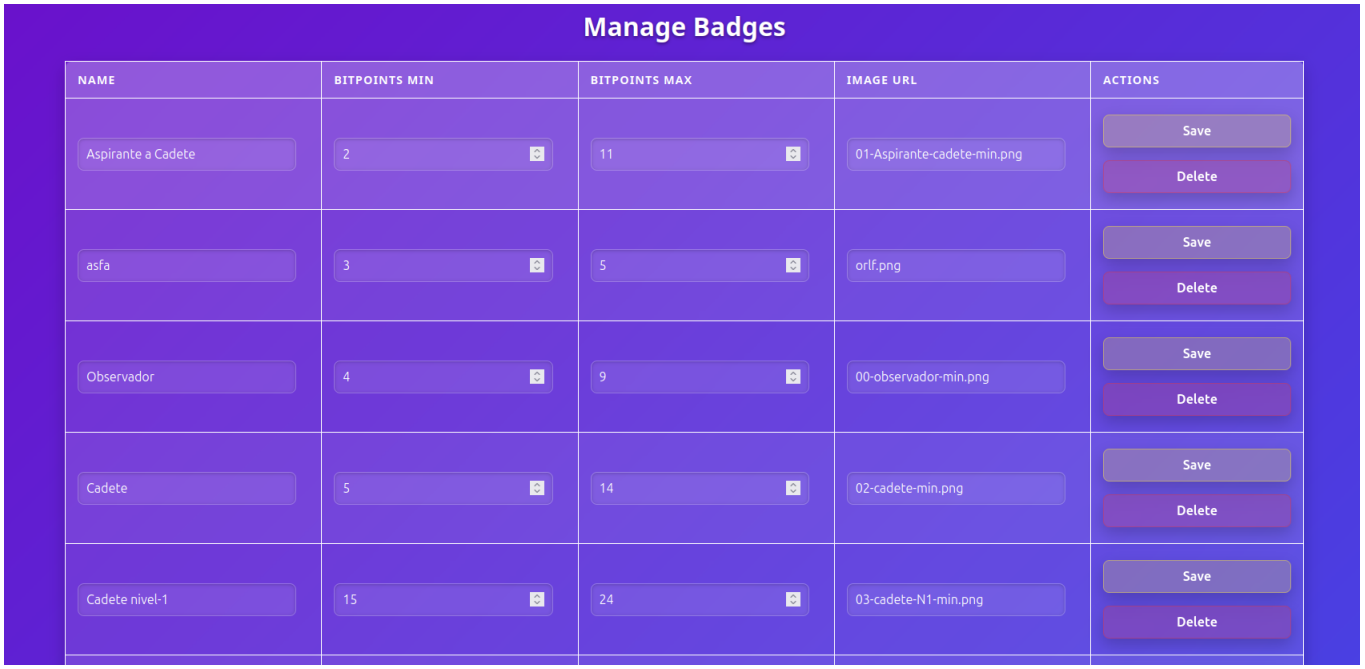


Figura 10: Admin - manage badges

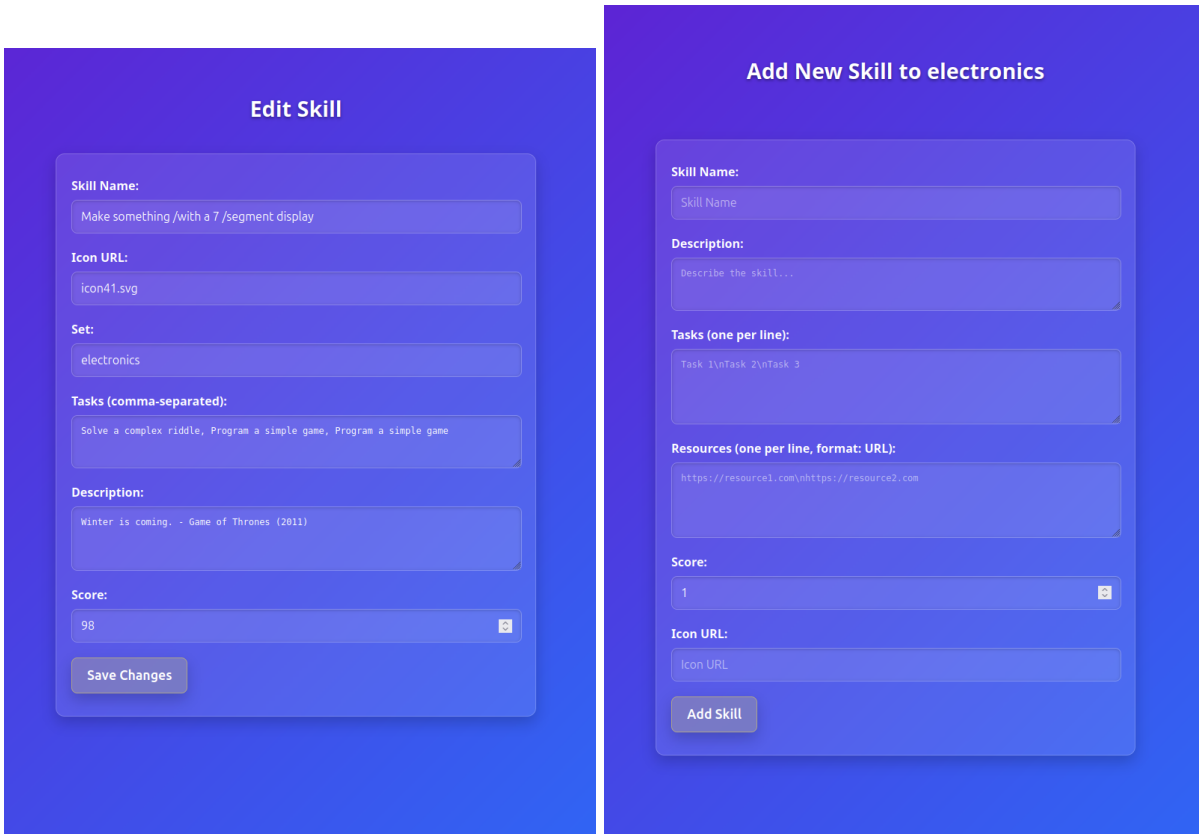


Figura 11: Edit skill

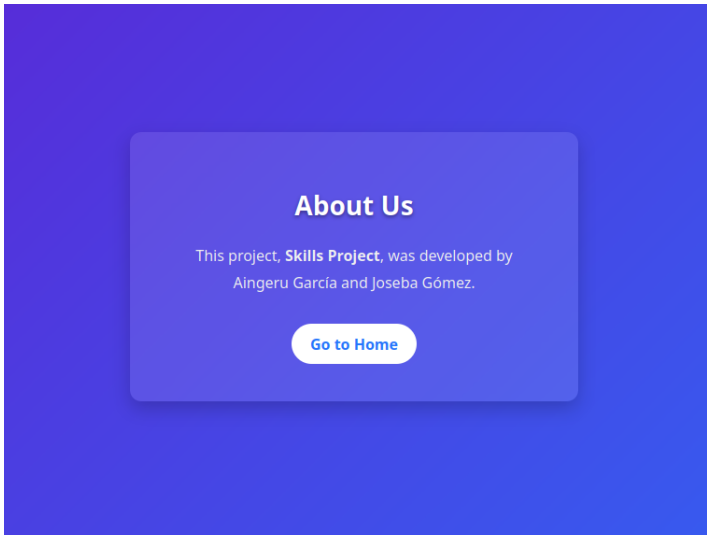


Figura 12: About us

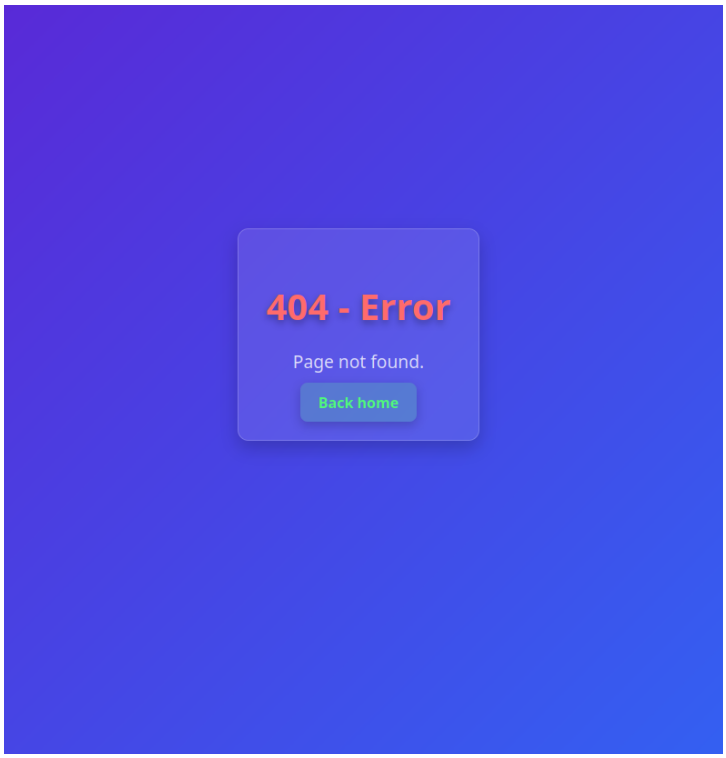


Figura 13: Error page

2 Conclusiones

Ha sido entrentenido realizar el proyecto. Pero, en varias ocasiones nos ha parecido que, tanto la primera fase como la segunda, el enunciado resultaba algo confuso.

26 de Diciembre de 2024