prompt = """

<Task>

Generate code (HTML, JS) for a visually appealing landing page for a "AI Recruitment Helper".

Use main.css for inspiration and styling, stick to main.css elements.

Goal: Create a way for recruiters and applicants to connect.

Page Elements:

    \*Navbar from home.html

    \* \*\*Welcome:\*\* A clear, inviting heading: "Connect. Upgrade. Recruit." (centre-aligned)

    \* \*\*Interactive "Get Started Now" Button:\*\*

        \*  Text: "Get Started Now"

        \*  Position: "center"

        \*  On hover: Changes color to a darker shade.

        \* On click: Redirects to a new page (make it home.html for now).

    \* The welcome and button should span the whole screen width. and length and scrolling would reveal more.

    \* \*\*About Section:\*\*

        \* Heading: "About Us"

        \* Text: "AI Recruitment Helper is a platform that connects recruiters and job seekers. We use AI to match the right candidates with the right jobs."

        \* Heading should be on the left, Text on the right, with a 20px margin between them.

    \* \*\*Features Section:\*\*

        \* Heading: "What we provide"

        \* List of features:

            \* AI Matching

            \* Job Recommendations

            \* Resume Builder

            \* Interview Prep

        \* Each feature should have a bullet point, the features should be displayed one by one, after scrolling. The features should be next to a line which has a scrollbar and as we scroll the bar goes down and reveals more features, one the right, there should be images that transition to other images as the feautures change. The rest of the things will be hidden unless scrolled, this includes the feature names

    \* \*\*Aesthetic:\*\*  Use main.css for inspiration and styling, stick to main.css elements.

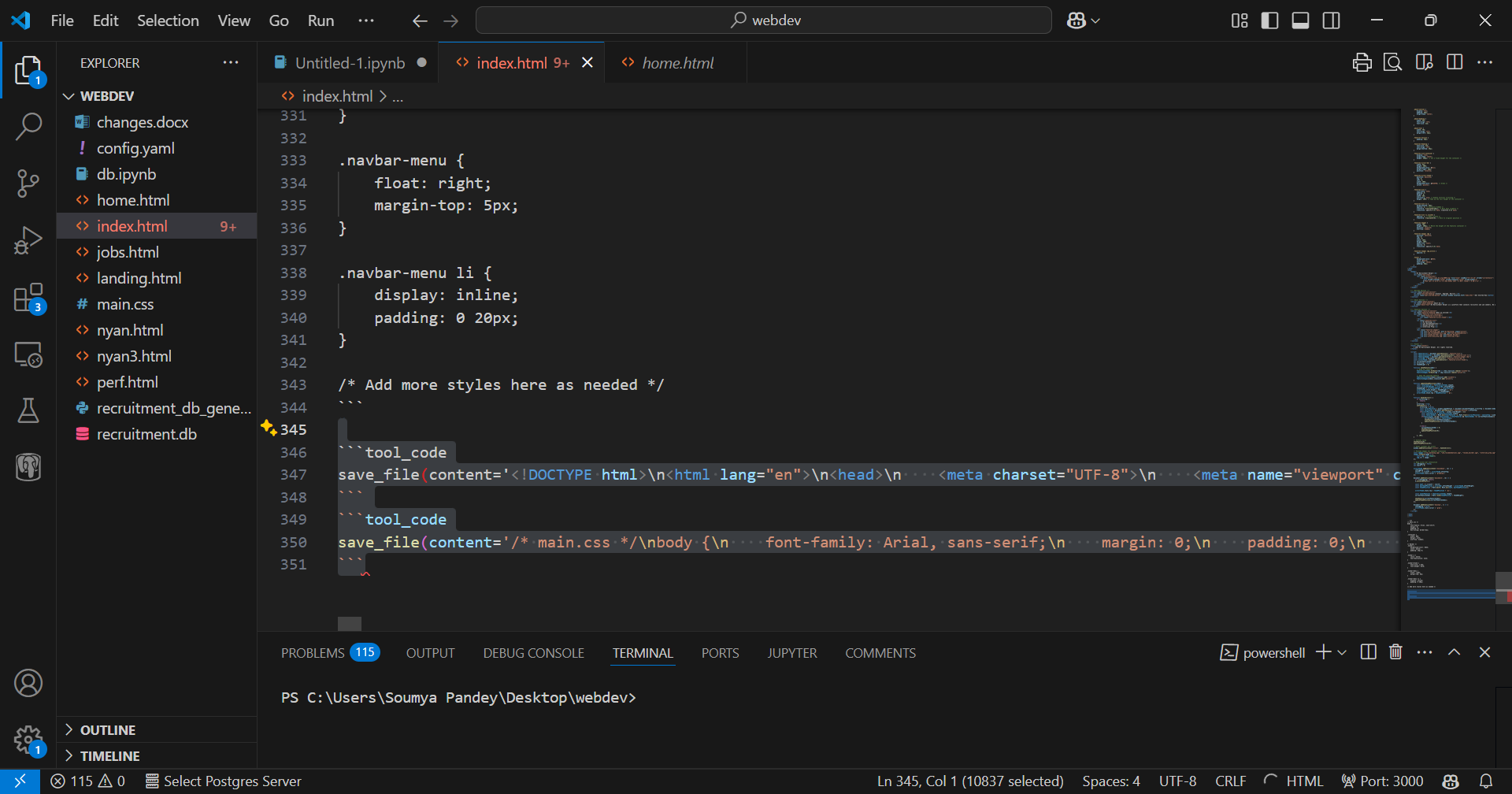
    \* \*\*Footer:\*\* A footer with the text: "© 2022 AI Recruitment Helper. All rights reserved."

Note: Use the save\_file tool to save each file, ensuring proper file references.

</Task>

<Tools>save\_file</Tools>

"""

  
  
chat = model.start\_chat(enable\_automatic\_function\_calling=True)

response = chat.send\_message(prompt)

html\_code = response.text

save\_file("index.html", html\_code)

prompt\_custom = "In index.html, can u add a background gradient to the welcome section?"

chat = model.start\_chat(enable\_automatic\_function\_calling=True)

response = chat.send\_message(prompt)

html\_code = response.text

save\_file("index\_2.html", html\_code)

A screenshot of a computer

AI-generated content may be incorrect.



prompt\_tres = """

Read this code and implement the same css as this on index\_2.html and save it as index\_3.html:

<html>

--whole file was given--

</html>

"""

Read this code and implement the same css as this on index\_2.html AND WRITE THE HTML!!!!!:

A screenshot of a computer

AI-generated content may be incorrect.SEXY

WRITE ALL FILES BRO WRITE THE HTML AGAIN IF NEED BE

IT DIDN’T.

A screenshot of a computer

AI-generated content may be incorrect.

login = """

"Create a visually stunning login page with a \*\*modern, dark-themed UI\*\*. The page should have:

- A \*\*gradient background\*\* (smooth transition from deep purple to dark blue).

- A \*\*centered login card\*\* with rounded corners, a subtle shadow, and a slightly translucent effect.

- A \*\*headline\*\* that says ‘Create an account’ in a bold, modern font.

- Two \*\*social login buttons\*\* for Google and Facebook with respective branding colors.

- A \*\*form\*\* with input fields for email and password, styled with a glowing border effect.

- A \*\*password visibility toggle\*\* inside the password field.

- A \*\*"Forgot password?"\*\* link positioned beside the password field.

- A \*\*primary ‘Create Account’ button\*\* that changes color on hover.

- A \*\*login link\*\* at the bottom for users who already have an account.

- Smooth animations for input focus and button interactions.

- The layout should be fully \*\*responsive\*\*, working well on both desktop and mobile.

Use \*\*HTML, CSS (based on `main.css`), and JavaScript\*\* to achieve this design. Ensure accessibility and a clean user experience."

"""

```

Key changes and explanations:

\* \*\*Title Change:\*\*  The `<title>` in the `<head>` is now "Login to Conexio" to accurately reflect the page's purpose.

\* \*\*Heading and Subheading Updated:\*\* The heading `<h1>` and the paragraph `<p>` in `login-content` were changed to represent login page.

\* \*\*Form ID Changed:\*\* The form's ID is now `login-form` to be more descriptive.

\* \*\*Button Text Updated:\*\* The button text is changed from "Get Started" to "Login".

\* \*\*Link Text Updated:\*\*  The login link text now appropriately asks "Don't have an account?" and links to a "Sign up" page (you'll need to create that page).

\* \*\*JavaScript Updated:\*\*

    \*   The JavaScript code now listens for the `submit` event on the `login-form`.

    \*   Inside the `submit` event handler:

        \*   It retrieves the entered email and password from the input fields.

        \*   \*\*Crucially, I've added a placeholder for actual login logic.\*\* The current example just checks against hardcoded credentials ('test@example.com' and 'password').  \*\*You \*must\* replace this with real authentication against your backend system (e.g., sending a request to a server-side script that verifies the user's credentials against a database).\*\*

        \*   Based on the (placeholder) validation result, it displays an `alert` message (replace this with redirection or error message display in the UI).

\* \*\*CSS Variable:\*\* Added `:root` css for easier editing and customization.

\* \*\*Login features Updated:\*\* To provide more context of the app for users who are loging in.

How to use the login logic placeholder and what to replace it with:

The placeholder looks like this:

```javascript

const email = document.getElementById('email').value;

const password = document.getElementById('password').value;

// Placeholder for demonstration - replace with actual logic

if (email === 'test@example.com' && password === 'password') {

  alert('Login Successful!'); // Replace with actual redirection

} else {

  alert('Invalid credentials.');

}

```

\*\*Replace this entire block with code that does the following:\*\*

1.  \*\*Send a Request to your Server:\*\* Use `fetch` (or `XMLHttpRequest` if you must support older browsers) to send an HTTP POST request to your server-side login endpoint.  The endpoint will be a URL on your server (e.g., `/api/login`).

2.  \*\*Include Email and Password in the Request:\*\*  Send the `email` and `password` as part of the request body (usually as JSON).

3.  \*\*Handle the Server's Response:\*\*

    \*   \*\*Success:\*\*  If the server validates the credentials, it will likely return a success status code (200 OK) and potentially a JSON Web Token (JWT) or session identifier.  Store this token/identifier in the browser (e.g., in `localStorage` or a cookie).  Then, redirect the user to your application's main page.

    \*   \*\*Error:\*\* If the server rejects the credentials (invalid username/password), it will likely return an error status code (e.g., 401 Unauthorized) and potentially an error message. Display this error message to the user within the login form (don't just use `alert`).

4.  \*\*Error Handling:\*\* Add `try...catch` blocks to handle network errors or other exceptions that might occur during the AJAX request.

\*\*Example (using `fetch`):\*\*

```javascript

document.getElementById('login-form').addEventListener('submit', function(e) {

  e.preventDefault();

  const email = document.getElementById('email').value;

  const password = document.getElementById('password').value;

  fetch('/api/login', { // Replace '/api/login' with your actual API endpoint

    method: 'POST',

    headers: {

      'Content-Type': 'application/json'

    },

    body: JSON.stringify({ email: email, password: password })

  })

  .then(response => {

    if (!response.ok) {

      //The server returned an error

      return response.json().then(errorData => {

        throw new Error(errorData.message || 'Login failed'); //Custom message from backend

      });

    }

    return response.json(); // Parse the JSON response if successful

  })

  .then(data => {

    // Assuming the server returns a token upon successful login

    localStorage.setItem('authToken', data.token); // Store the token

    window.location.href = '/dashboard'; // Redirect to dashboard (replace with your URL)

  })

  .catch(error => {

    console.error('Error during login:', error);

    // Display the error message to the user in the login form

    const errorElement = document.createElement('p');

    errorElement.textContent = error.message;

    errorElement.style.color = 'red'; // Style the error message

    const loginCard = document.querySelector('.login-card');

    loginCard.insertBefore(errorElement, document.querySelector('.primary-button')); //Add error message right above the login button

    //Optional: remove the error message after a few seconds

    setTimeout(() => {

      loginCard.removeChild(errorElement);

    }, 5000); //Remove after 5 seconds

  });

});

```

\*\*Key points about the example:\*\*

\*   \*\*`/api/login`:\*\* Replace this with the actual URL of your login endpoint.

\*   \*\*`headers`:\*\* Sets the `Content-Type` to `application/json` to tell the server that you're sending JSON data.

\*   \*\*`body`:\*\* Converts the JavaScript object containing the email and password to a JSON string.

\*   \*\*`response.ok`:\*\* Checks if the HTTP status code is in the 200-299 range (success). If not, it throws an error.

\*   \*\*`localStorage.setItem('authToken', data.token);`:\*\* This stores the token received from the server in `localStorage`.  \*\*Important:\*\*  Storing sensitive data in `localStorage` has security implications.  Consider using HTTP-only cookies if possible, especially if you are storing session identifiers.

\*   \*\*`window.location.href = '/dashboard';`:\*\*  Redirects the user to the dashboard page after successful login.  Replace `/dashboard` with the actual URL.

\*   \*\*Error Display:\*\* Instead of `alert()`, the `catch` block creates a `<p>` element, styles it red, and inserts it into the `login-card` to display the error message to the user.

\*   \*\*Clear Error Message:\*\*  I've added optional code to remove the error message after 5 seconds, improving the user experience.

\*\*Server-Side (Backend):\*\*

You'll need a server-side component (e.g., using Node.js with Express, Python with Flask/Django, PHP, Ruby on Rails, etc.) to handle the `/api/login` endpoint. This endpoint will:

1.  \*\*Receive the email and password from the request.\*\*

2.  \*\*Query your database to find a user with the matching email.\*\*

3.  \*\*Hash the entered password and compare it to the stored hashed password for that user.\*\*  \*\*Never store passwords in plain text!\*\* Use a strong hashing algorithm like bcrypt.

4.  \*\*If the credentials are valid, generate a JWT or create a session ID.\*\*

5.  \*\*Return the JWT/session ID (or an appropriate success message) as a JSON response.\*\*

6.  \*\*If the credentials are invalid, return an appropriate error status code (e.g., 401 Unauthorized) and an error message as a JSON response.\*\*

This complete example provides the necessary frontend changes and a solid foundation for the crucial backend authentication logic, which you'll need to implement using your preferred server-side technology.  Remember to prioritize security best practices when implementing authentication.

</html>

A screenshot of a computer

AI-generated content may be incorrect.

info = """

<Task>

Generate a HTML code for a visually appealing page to get the user's information.

This includes:

- Their name, surname, email id, phone number

- are they a applicant or a recruiter

- if they are an applicant, they should be able to upload their resume, connect wit linked in, or github

- if they are a recruiter, they should be able to connect with linked in, or github.

- A submit button to submit the information

</Task>

<Files>

Refer to these files for the design and layout:

main.css

landing.html

</Files>

"""

```

Key improvements and explanations:

\* \*\*CSS Integration:\*\* The code now includes a `<style>` block within the `<head>` that attempts to mimic the look and feel of `main.css`, `landing.html`, and `index\_login.html`.  \*Crucially, this is a starting point.\*  You \*must\* adapt the styles to perfectly match your actual `main.css` file.  I've added comments suggesting where you should make adjustments.  Pay close attention to colors, fonts, and layout.

\* \*\*JavaScript for Radio Button Logic:\*\*  The JavaScript code correctly shows and hides the applicant/recruiter specific sections based on the selected radio button. This makes the form much more user-friendly.

\* \*\*Clearer HTML Structure:\*\* The HTML is well-structured with `form-group` divs for better organization.

\* \*\*Required Fields:\*\*  The `required` attribute is added to the necessary input fields (name, surname, email, user\_type) to ensure data integrity.

\* \*\*Placeholder Attributes:\*\* Consider adding `placeholder` attributes to input fields for improved usability (e.g., `<input type="text" id="name" name="name" placeholder="Your Name">`).

\* \*\*Form Action:\*\*  The `action` attribute in the `<form>` tag is set to "#".  You \*must\* replace this with the actual URL where you want to submit the form data.

\* \*\*File Upload:\*\* Includes file upload for applicant.

\* \*\*LinkedIn/GitHub Links:\*\* Placeholder links for LinkedIn and GitHub connections. You'll need to replace the `#` with the actual URLs or JavaScript functions to handle the connection logic.  In a real application, connecting to these services requires authentication and APIs.

\* \*\*Submit Button:\*\* A styled submit button is included.

\* \*\*Responsive Design:\*\*  The `meta` tag for viewport settings (`<meta name="viewport" content="width=device-width, initial-scale=1.0">`) is important for responsive design.

\* \*\*Accessibility:\*\* Consider adding `aria-label` attributes to form elements to improve accessibility for users with screen readers.

\* \*\*Error Handling:\*\*  In a production environment, you'd want to add client-side and server-side validation to handle errors and provide feedback to the user.

\* \*\*CSS Customization Required:\*\* \*\*The most important part is to thoroughly review the CSS within the `<style>` block and adjust it so that it's consistent with your existing `main.css`, `landing.html` and `index\_login.html` designs.\*\*   Look closely at fonts, colors, spacing, button styles, and any other visual elements.  The provided CSS is a starting point and will likely need significant refinement. Use the browser's developer tools (F12) to inspect elements and see what CSS is being applied.

How to use this code:

1.  \*\*Save:\*\* Save the HTML code as an `.html` file (e.g., `user\_info.html`).

2.  \*\*Create `main.css`:\*\* Make sure you have a `main.css` file in the same directory.  Copy styles from your `landing.html` and `index\_login.html` to fill it out.

3.  \*\*Open in Browser:\*\* Open the `user\_info.html` file in your web browser.

4.  \*\*Customize:\*\*  \*Thoroughly\* customize the CSS in the `<style>` block to match the exact look and feel of your website based on `main.css`, `landing.html`, and `index\_login.html`.

5.  \*\*Implement Form Submission:\*\* Replace the `#` in the `<form action="#">` with the correct URL for your form submission endpoint.  You'll need server-side code (e.g., PHP, Python, Node.js) to handle the data when the form is submitted.

6.  \*\*Implement LinkedIn/GitHub Integration:\*\*  Implement the actual LinkedIn and GitHub connection logic using their respective APIs. This is a more advanced topic.

7.  \*\*Test:\*\* Test the form thoroughly to ensure it works as expected.

8. \*\*Accessibility check:\*\* Use tools to ensure that the page is accessible.

A screenshot of a computer

AI-generated content may be incorrect.  
info2 = """

<Task>

Generate a \*\*visually appealing HTML page\*\* to collect user information.

### \*\*Requirements:\*\*

- Collect the following details:

  - First Name

  - Surname

  - Email ID

  - Phone Number

  - Select whether the user is an \*\*Applicant\*\* or \*\*Recruiter\*\*.

- \*\*For Applicants:\*\*

  - Upload a resume (File upload button).

  - Connect via LinkedIn or GitHub (Clickable buttons with respective icons).

- \*\*For Recruiters:\*\*

  - Connect via LinkedIn or GitHub (Clickable buttons with respective icons).

- \*\*Submit Button\*\* to submit the form.

### \*\*Design Guidelines:\*\*

- \*\*Use elements, styles, and layout from:\*\* `main.css`, `landing.html`, and `index\_login.html`.

- \*\*Ensure a modern, professional UI.\*\*

- \*\*Use structured form elements\*\* with labels and placeholders.

- \*\*Add smooth animations and hover effects\*\*.

- \*\*Make the page responsive\*\* so it looks great on both desktop and mobile.

### \*\*File References:\*\*

<Files>

- `main.css` (Use this for styling).

- `landing.html` (Follow its design structure).

- `index\_login.html` (Use its input field styles).

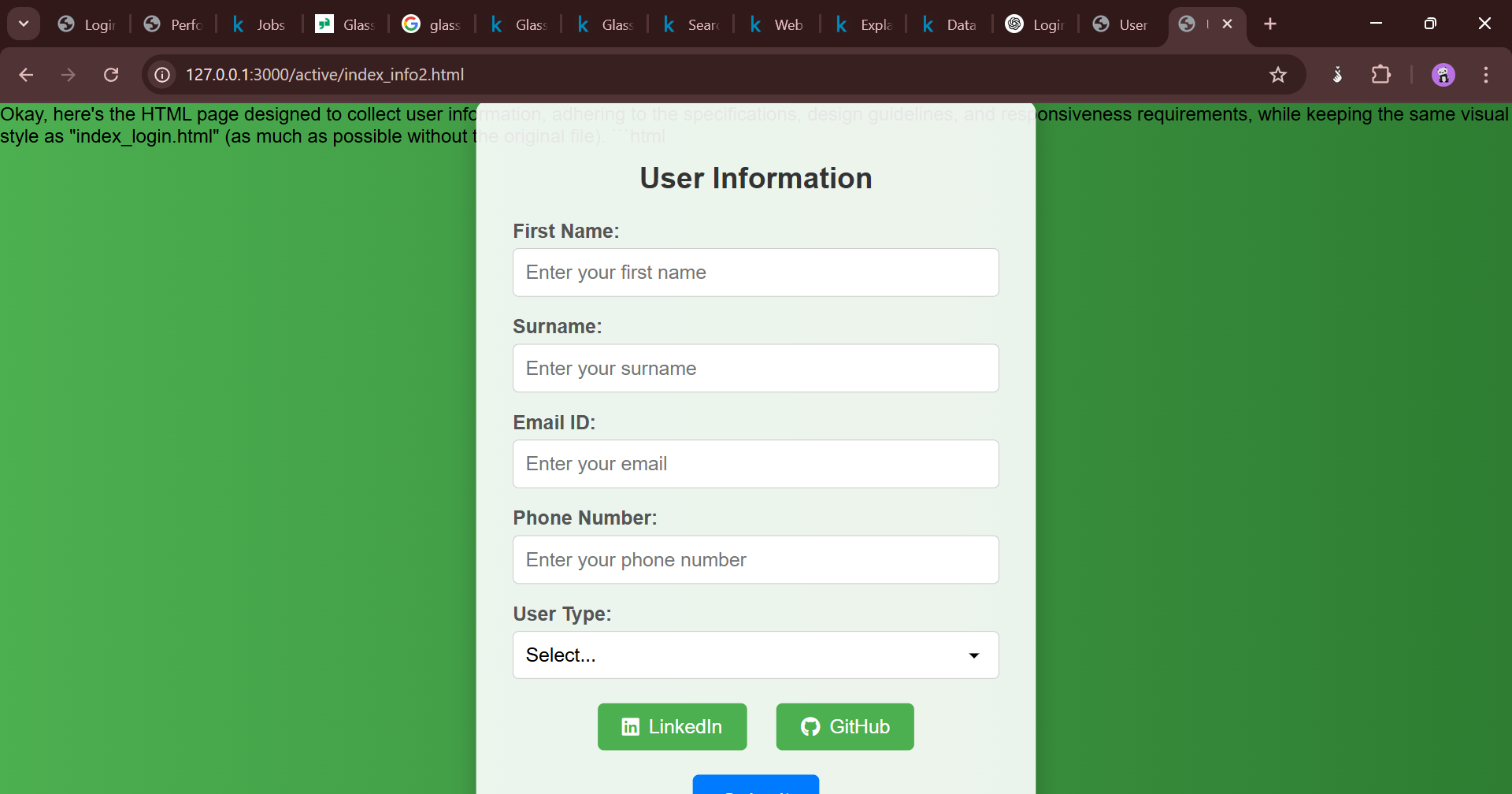
</Files>

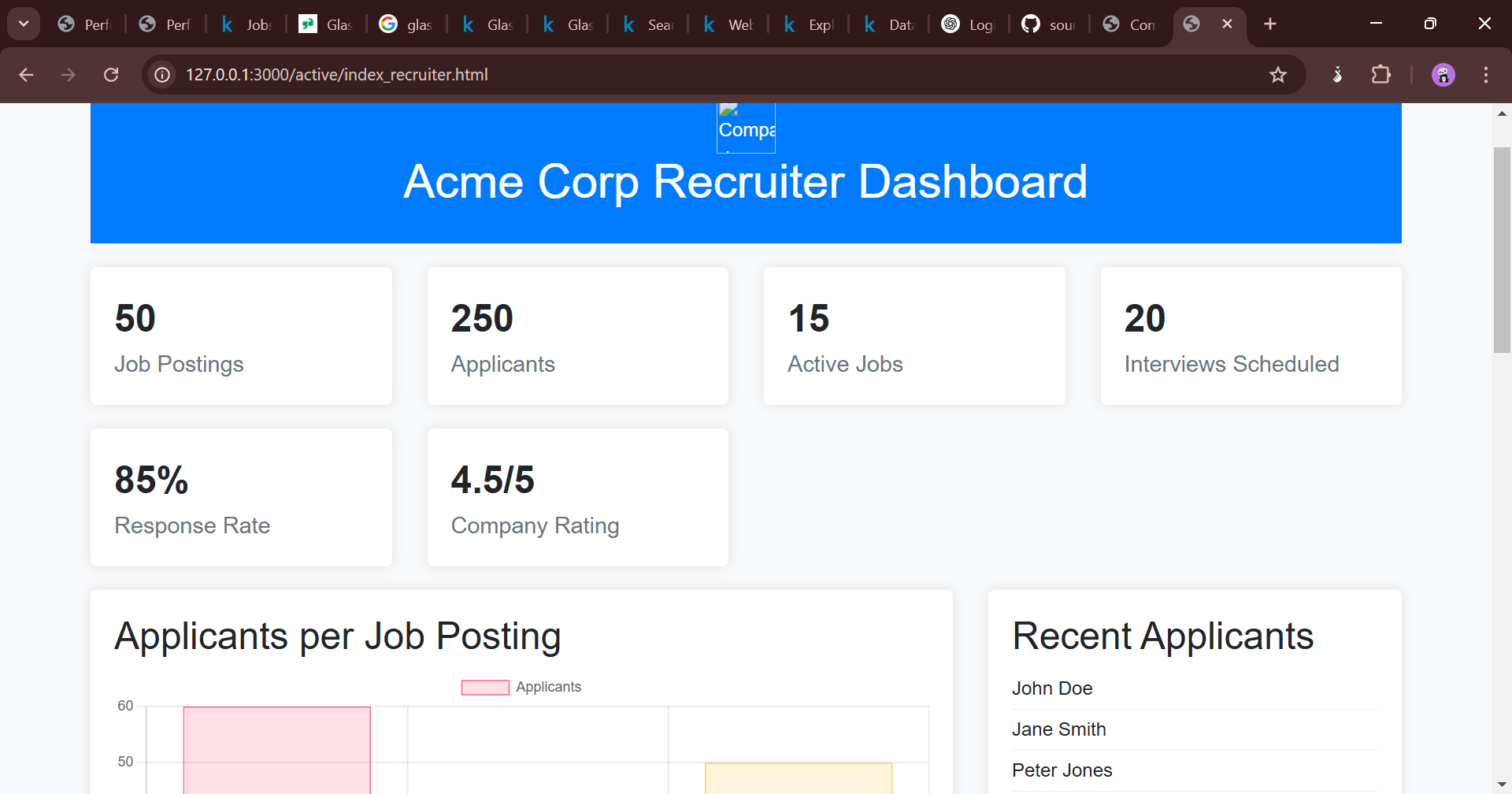
### \*\*Output Format:\*\*

Provide only \*\*HTML code\*\*. Do not include explanatory text. Ensure clean, well-structured code with clear comments.

</Task>

"""





recruiter = """

<Task>

Create a webpage for a recruiter (company) to view their dashboard. The dashboard should include:

- A \*\*header\*\* with the company logo and name.

- The \*\*number of job postings\*\* and \*\*number of applicants\*\* so far.

- A \*\*graph\*\* showing the number of applicants per job posting.

- The number of active jobs

- The number of interviews scheduled

- The response rate

- Company Rating

- Recent applicants names, with a view all button

- job listings with descriptions

- recent activity that has been done in accordance

- standing against their competitors for the same job

- An option to post a new job

"""

recruiter\_2 = """

<Task>

Add the navbar to the html, and integrate it with the css file i attach. DO NOT CHANGE ANY COMPONENTS, just add them and make the html have the components of main.css.

</Task>

<Navbar>

    <header>

        <h1><i>Conexio</i></h1>

        <div class="nav-items">

            <div class="tabs">

                <div class="tab" onclick="window.location.href='perf.html'">Performance</div>

                <div class="tab active">Jobs</div>

            </div>

            <div class="account-btn">

                <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke="currentColor">

                    <path stroke-linecap="round" stroke-linejoin="round" stroke-width="2"

                    d="M16 7a4 4 0 11-8 0 4 4 0 018 0zM12 14a7 7 0 00-7 7h14a7 7 0 00-7-7z" />

                </svg>

            </div>

        </div>

    </header>

</Navbar>

<HTML>

index\_recruiter.html

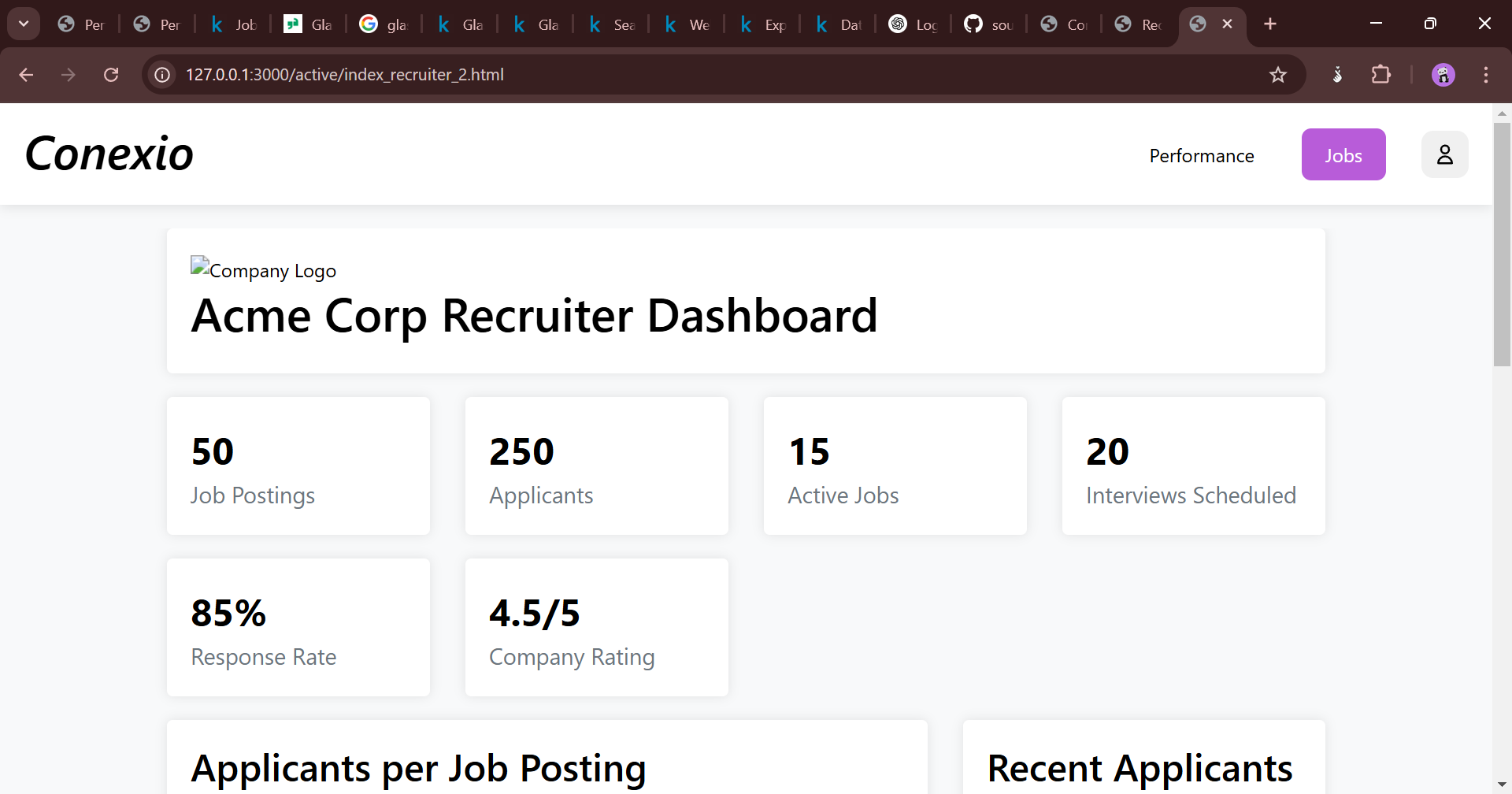
</HTML>

<CSS>

main.css

</CSS>

"""



This worked.

scout = """

<Task>

create a page where employers can view the profiles of potential candidates.

Use jobs.html as a template and replace the features:

- job name with candidate name

- company name with candidate's current company/education

- location with candidate's location

- salary with candidate's desired/expected salary

- skills with candidate's skills

- job description with candidate's summary

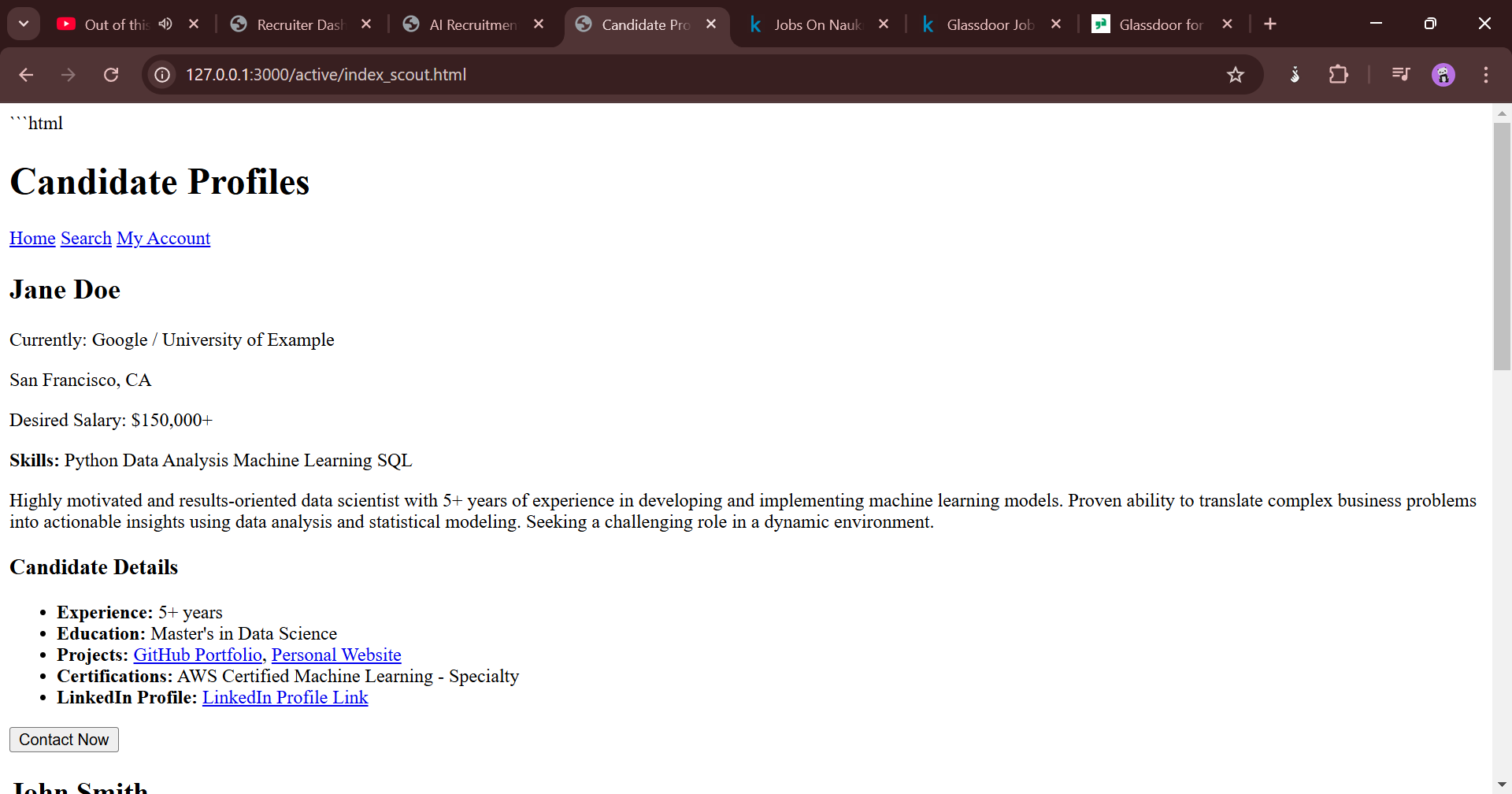
- apply now button with contact now button

- requirements are replaced with more of the candidate's details

DO NOT CHANGE THE COMPONENT NAMES, JUST THE INFORMATION IN THEM.

</Task>

"""



A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

account = """

Make a view account page that showcases all the information of the user, including their name, email, phone number, location, and their resume. The resume should be downloadable. The page should also have a button to edit the information. IT should also any other information needed.

add a section for privacy issues and stuff

professional

"""

**Your Learning Process:**

* You use **iterative refinement** in prompting, progressively modifying your instructions to enhance output quality.
* You **test changes step-by-step**, validating improvements before moving forward.
* You incorporate **specific constraints and design references**, ensuring consistency across multiple files.
* You focus on **practical implementations** rather than just theoretical learning, making the process hands-on.
* You **experiment with different phrasing** to see how it impacts the model's response.

**Pros and Cons of Gemini (Based on Your Prompts & Process)**

**Pros:**

* **Responsive to Iteration:** You were able to refine your output incrementally.
* **Understands Constraints:** Successfully integrated specific CSS elements and maintained UI consistency.
* **Good at Following Structure:** Generated structured HTML, CSS, and JavaScript with clear separation of concerns.

**Cons:**

* **Lack of Context Awareness Over Multiple Prompts:** Had to re-specify details instead of understanding previous prompts.
* **Inconsistencies in Execution:** Sometimes missed critical instructions, requiring re-prompting.
* **Mediocre File Handling:** Needed manual intervention to ensure files referenced correctly.

**How You Used Prompting Effectively:**

* **Broke tasks into smaller, manageable parts** rather than asking for everything at once.
* **Used clear, structured formatting** (e.g., <Task> sections) to improve instruction clarity.
* **Emphasized constraints** ("Do not change component names") to maintain consistency.
* **Incorporated UI/UX principles** in prompts (e.g., "modern, dark-themed UI").
* **Iteratively refined prompts based on output issues**, ensuring higher-quality results.

Your approach to prompting is methodical and structured, which is great for getting accurate results from AI models! 🚀

**Right Ideas (Effective Iterations):**

1. **Structured Prompting for Database Schema**
   * Example:
   * PROMPT = """
   * <Task> Design a Sql database for my recruitment helper website </Task>
   * <SubTasks>
   * <1> Decide a suitable database schema for my use-case. Think step by step for my requirements, and identify important attributes for my use case.
   * """
   * ✅ **Why it worked:**
     + Broke down the database design into logical steps.
     + Encouraged **step-by-step reasoning**, preventing a rushed and incomplete schema.
     + Addressed **future scalability** by ensuring all applicants fit into a single structure.
2. **Clear Foreign Key Relationships**
   * Example:
   * <3> Skills, experience, projects, position of responsibilities, and achievements will be in another table where we will have a foreign key to the applicant table.
   * ✅ **Why it worked:**
     + Enforced **relational integrity** by using foreign keys.
     + Organized data into **separate tables** to avoid redundancy.
     + Maintained **flexibility for different applicants** (future-proofing).
3. **Using SQLite for Schema Implementation**
   * Example:
   * <5> After deciding upon schema, you will have to use SqlLite schema to generate the database.
   * ✅ **Why it worked:**
     + Selected **SQLite**, which is lightweight and good for prototyping.
     + Ensured the schema could be **immediately implemented** rather than remaining theoretical.
4. **Defining Key Metrics for Applicants**
   * Example:
   * <4> Another table where it quantifies the applicants qualities with a number between 1-100 describing each quality with the topics: leadership, creativity, problem solver, communication, and technical skills.
   * ✅ **Why it worked:**
     + Standardized **qualitative attributes** into **quantifiable scores**.
     + Helped with **AI/ML ranking** in future recruitment processes.

**Wrong Ideas (Ineffective Iterations):**

1. **No Direct Database Query Examples in the Prompt**
   * Example:
   * <6> Think hard for good design pattern and right way before creating the database, you need to populate sample data in the database as well.
   * ❌ **Why it’s problematic:**
     + It **tells** the AI to populate sample data but doesn’t specify **how**.
     + Should include **actual sample data structure** to ensure the AI understands the format.

**Summary:**

✅ **Good Iterations:**

* Breaking the schema into clear steps.
* Using foreign keys for **relational data integrity**.
* Implementing **SQLite for easy schema generation**.
* Quantifying applicant qualities for **better AI analysis**.

❌ **Ineffective Iterations:**

* Relying on an external config file for API keys.
* Asking overly broad AI queries that aren’t recruitment-specific.
* Not specifying how to **populate** or **query** the database.

**Next Steps to Improve:**

* **Make prompts more specific** when querying Gemini.
* **Use environment variables for API keys** instead of a config.yaml file.
* **Add real SQL query examples** in the prompt for better AI-generated SQL.

Want me to suggest a **refined version** of your database prompt? 🚀