7 Building user interfaces with ChatGPT

This chapter covers

- Using generative AI tools to design effective user interfaces
- Creating structured prompts to get targeted UI guidance and code
- Visualizing application flow using AI-generated diagrams and flowcharts
- Transforming wireframes into functional Flask templates and components
- Implementing responsive design elements with AI-assisted HTML and CSS

Now that our backend is up and running, we are facing another challenge: we need to create an easy-to-use interface, which will turn our database engine into a full test preparation application. Even seasoned developers can find UI design tough, and generative AI tools help tremendously.

AI tools provide a big boost for developers who aren't great at UI. They draw from large collections of best practices, templates, and design patterns used by expert

Flask developers. This chapter shows how AI can connect functional backend code to polished user experiences—even if you don't see yourself as a design expert. By mixing AI suggestions with smart customization, you'll learn to build professional interfaces that boost user engagement, all without needing deep frontend knowledge.

To find a good strategy, we don't need as much existing code for context, so I'd like to start by using chat-based interfaces to ask for a general approach. We can give a bit of information about our application and get general guidance on how to build this frontend. I'm going to utilize several tools we're already familiar with for this task:

- ChatGPT
- Google Gemini (formerly Bard)
- Blackbox AI

Each of these has a web-based chat interface that can be used to work through this problem. It's not installed on our machine, so it has limited access to our code. However, they're also powered by large models that are very robust. Let's see which one comes up with a strategy we like. First, we need to build a great prompt for this.

7.1 Getting our strategy from our AI tools

Before we approach these tools, we should think carefully about our prompt. If we say "generate a frontend for my application" without enough information, the guidance will be too vague. There is a time and a place for brief prompts. In my experience, *brief prompts* are great for focused and specific advice, for problems such as "How do I query this table?" or "How do I debug this error message?" *Detailed prompts* are great for more abstract results, such as "How do I design this frontend?" or "What's a good approach for marshalling data?"

This will be the largest prompt we've used so far. I've given it a lot of thought and trial and error to come up with it. Here is how I want to structure it:

- *Role*—Role-based prompting is something we've done before here, and we want to establish that we want the model to act as a professional Python developer who specializes in Flask. Lately, I've found that establishing a role isn't as important as it used to be. ChatGPT for instance can infer what you want without it, and it's nearly the same. But just to make sure, we'll drop it in there.
- *Objective*—We want to clearly establish our objective. The clearer and more concrete we are, the better the results. We must describe exactly what we want here.
- *Details*—Here we can add some details that will be helpful. These are small but significant toward getting the right result.
- *Data*—I want to describe the data we're working with and what it means. This helps with code generation. If we get code specific to the data, we might not use it, but it will be helpful to give this direction.
- *Functionality requirements*—What do we want this to do? This is a good way to add detail to our objective.

- *Code production*—Here we are specific about what we want to generate. In my experience, this only helps marginally, but it's worth the effort to clarify it.

Here is the initial prompt that I've created and will send to our chat interfaces:

JM

Act as a professional Python Developer specializing in Flask. Create a solution based on this information.

Objective: Develop a front end for a database-driven Flask application for a quiz system. The application will display questions and possible answers to the user, capture their selected answer, store the answer in a database, and then show the next question. Flask has been installed, and the backend is already functional.

Details:

Framework: Flask, with Flask templates for the frontend. The application is already created.

Database: A SQLite Database that provides 35 questions with possible user responses. This database exists.

Frontend: A simple and intuitive interface for users to interact with the quiz. This needs to be created.

Data:

This is an example of a question:

('T1A03', 3, 'What are the FCC rules regarding the use of a phonetic alphabet for station identification in the Amateur Radio Service?', 'It is required when transmitting emergency messages', 'It is prohibited', 'It is required when in contact with foreign stations', 'It is encouraged')

Field 0 is the Question ID.

Field 1 is the id of the correct answer.

Field 2 is the question to be displayed.

Field 3 is a possible answer.

Field 4 is a possible answer.

Field 5 is a possible answer.

Field 6 is a possible answer.

Functionality Requirements:

Iterate through the questions stored in the database.

For each question, display it along with possible answers using Flask templates.

Capture the user's selected answer and store it in the database.

After storing the answer, proceed to the next question until the quiz is complete.

Development Strategy:

Question Management: Implement functionality to add, retrieve, and iterate through questions and their possible answers.

Quiz Interface: Use Flask templates to create the quiz interface. The interface should display one question at a time with options for the user to select an answer.

Answer Handling: Develop a mechanism to capture the user's selected answer, validate it, store it in the database, and then retrieve the next question.

Navigation: Ensure seamless navigation between questions and provide feedback to the user upon completion of the quiz.

Code Production:

Develop routing and template creation for displaying questions and capturing answers.

Implement logic for iterating through questions and storing responses.

Ensure to include error handling and validation for robustness.

Expected Outcome: A fully functional Flask-based quiz application that efficiently manages and displays quiz questions, captures user responses, and navigates through the quiz seamlessly.

I will send this prompt to ChatGPT, Gemini, and Blackbox AI. Let's evaluate the results. We'll start with ChatGPT.

Crafting effective UI prompts

When requesting UI guidance from AI tools,

- Clearly state the purpose of your application and its intended users.
- Identify any technological constraints, such as frameworks or libraries, that may affect the design.
- Describe the data structures that will be used to populate the interface.
- Specify the exact functionality you need to achieve.
- Share your visual preferences, including colors and layout styles.
- Request the specific deliverables you need, such as wireframes, HTML/CSS, or flowcharts.
- For design work, it's best to provide longer and clearer prompts to get better results.
- Treat your prompt like a design brief—more context means a more tailored response.

7.1.1 ChatGPT Results

We've crafted a detailed prompt that's specific to our needs. Here is the output from ChatGPT. Let's review it and see whether it's useful.

Given your objective and details, let's outline a solution for developing the frontend of your Flask application for a quiz system. The solution will cover the backend and frontend components necessary to meet your requirements.

Backend Setup

1. Database Model: Assuming you have a model for the questions, it should look something like this in Flask with SQLAlchemy:

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class Question(db.Model):
    id = db.Column(db.String, primary_key=True)  # Question ID
    correct_answer_id = db.Column(db.Integer)  # Correct answer ID
    question_text = db.Column(db.String)  # Question text
    option1 = db.Column(db.String)  # Option 1
    option2 = db.Column(db.String)  # Option 2
    option3 = db.Column(db.String)  # Option 3
    option4 = db.Column(db.String)  # Option 4
```

2 Fetching Questions: Implement a function to fetch questions one at a time from the database.

```
def get_next_question(current_question_id=None):
    if current_question_id is None:
        return Question.query.first()
    else:
        current_question = Question.query.filter_by
(id=current_question_id).first()
        return Question.query.filter(Question.id >
current_question.id).first()
```

3 Storing Answers: You'll need a model to store user answers and a function to handle the storage:

```
class UserAnswer(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    question_id = db.Column(db.String, db.ForeignKey('question.id'))
    selected_answer_id = db.Column(db.Integer)
    # Add user reference if needed

def store_user_answer(question_id, selected_answer_id):
    answer = UserAnswer(question_id=question_id,
selected_answer_id=selected_answer_id)
    db.session.add(answer)
    db.session.commit()
```

Frontend Setup

1 Quiz Interface: Use Flask's Jinja templates to create the quiz interface. Here's a simple template for displaying questions:

```
<!-- templates/question.html -->
<html>
<head>
    <title>Quiz Question</title>
</head>
<body>
    <form action="/answer" method="post">
        <h2>{{ question.question_text }}</h2>
        {% for option in question.options %}
```

```
        <input type="radio" name="answer"
value="{{ loop.index }}">{{ option }}<br>
        {% endfor %}
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

2 Routing and Logic: Implement Flask routes to handle displaying questions and capturing answers.

```
from flask import Flask, request, render_template, redirect, url_for

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///yourdatabase.db'
db.init_app(app)

@app.route("/")
def index():
    question = get_next_question()
    return render_template("question.html", question=question)

@app.route("/answer", methods=["POST"])
def answer():
    selected_answer_id = request.form['answer']
    question_id = request.form['question_id']
    store_user_answer(question_id, selected_answer_id)
    next_question = get_next_question(question_id)
    if next_question is None:
        return "Quiz Complete! Thank you for participating."
    else:
        return redirect(url_for('index', question_id=next_question.id))
```

Final Steps

- Make sure to initialize your database with flask `db init`, `flask db migrate`, and `flask db upgrade` commands if you are using Flask-Migrate.
- Test your application thoroughly to ensure that questions are fetched correctly, answers are stored, and the user can navigate through the quiz without issues.
- Implement error handling and validation to manage cases where the user might submit an answer without selecting an option or if there's an issue fetching questions from the database.

This setup should give you a robust starting point for your Flask-based quiz application. Adjustments may be needed based on your specific requirements or if you decide to add more features like user authentication, score tracking, or feedback mechanisms.

This is useful information, and it's laid out in a useful way. Here are a few things I like about the results from ChatGPT. It walks us through a sequence of events, as outlined in figure 7.1.

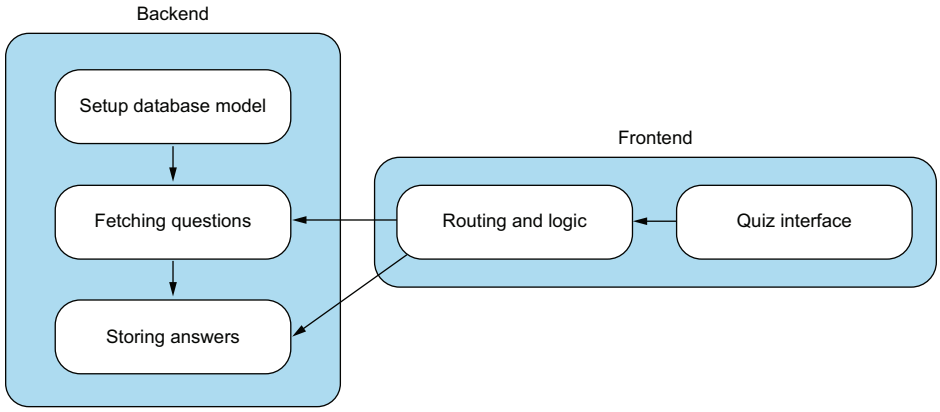

Figure 7.1 How ChatGPT suggested building the frontend. Each piece represents a function that must be created.

ChatGPT prescribes methods to develop this application, many of which we've already talked about. It suggests ways to set up the backend and use SQLAlchemy, which I intentionally avoided in this project. For example, in section 1 of backend design, there is

> Database Model: Assuming you have a model for the questions, it should look something like this in Flask with SQLAlchemy:

It's good advice, but we won't use it. It's not that SQLAlchemy is bad, but it's overkill for what we're trying to do. I skipped through the sections describing how to assemble the backend.

The frontend setup, however, is useful. It prescribes the following steps:

- Create a Jinja template in HTML.
- Implement routing.
- Implement redirects.

I'll ignore most of what ChatGPT has suggested except for the frontend setup. Even though I specified in the prompt that the backend was complete, much of the result was focused on backend and architecture, which I don't need at this point. Let's see what the other tools have to say about it.

7.1.2 Gemini Results

Here is the output from Gemini, exactly as it was returned:

> Flask Quiz Application Frontend
>
> This solution utilizes Flask templates to render the quiz interface and interact with the existing backend for storing user responses.

1 **Dependencies:**

Install Flask-WTF for form handling:

```
pip install Flask-WTF
```

2 **Templates:**

Create a folder named templates in your project directory. Within this folder, create a file named quiz.html.

quiz.html:

```
<!DOCTYPE html>
<html>
<head>
  <title>Quiz App</title>
</head>
<body>
  <h1>{{ question.question }}</h1>
  <form action="/answer" method="POST">
    {% for answer in question.answers %}
      <input type="radio" name="answer" value="{{ answer.id }}"> {{
answer.answer }}<br>
    {% endfor %}
    <br>
    <button type="submit">Submit Answer</button>
  </form>
  {% if message %}
    <p>{{ message }}</p>
  {% endif %}
</body>
</html>
```

3 **Routes and Logic:**

- Import necessary libraries.
- Define routes for displaying the quiz and handling answers.
- In the quiz route, retrieve the next question from the database (implement logic based on your backend).
- Pass the question and answers to the quiz.html template.
- In the answer route, validate the user's selected answer from the submitted form.
- Store the answer in the database (use your existing backend functions).
- Retrieve the next question or display a completion message.

app.py:

```
from flask import Flask, render_template, request, redirect
from flask_wtf import FlaskForm
from wtforms import RadioField, SubmitField
from wtforms.validators import DataRequired

app = Flask(__name__)
```

```
app.config['SECRET_KEY'] = 'your_secret_key'

class AnswerForm(FlaskForm):
  answer = RadioField('Answer', validators=[DataRequired()])
  submit = SubmitField('Submit')

@app.route('/')
def quiz():
  # Retrieve the next question from the database
  question = # your database logic...
  if not question:
    return render_template('quiz.html', message="Quiz completed!")
  form = AnswerForm()
  return render_template('quiz.html', question=question, form=form)

@app.route('/answer', methods=['POST'])
def answer():
  form = AnswerForm()
  if form.validate_on_submit():
    selected_answer = form.answer.data
    # Store answer in the database
    # your database logic...
    return redirect('/')
  return render_template('quiz.html', question=question, form=form)

if __name__ == '__main__':
  app.run(debug=True)
```

4 **Explanation:**

- The `AnswerForm` class defines a radio button field for selecting an answer and a submit button.
- The quiz route retrieves the next question and renders the `quiz.html` template, passing the question details and the form object.
- The answer route validates the submitted form and stores the chosen answer using your existing backend logic.
- Error handling is implemented using `DataRequired` validator for the radio button field.

5 **Running the application:**

Save the files and run the application:

python app.py

This code provides a basic example. You might need to modify it based on your specific backend implementation and desired features like displaying user's score or providing feedback. Remember to replace # your database logic. . . with your actual functions for retrieving questions and storing answers.

The Gemini results start off with suggesting we install `Flask-WTF` to handle the quiz and store responses. It doesn't give any guidance on building the backend, which is good because we've already built it.

What is `Flask-WTF`? It's an extension that integrates the `WTForms` library and handles forms in a simple way for a Flask application. It's certainly something worth considering for our project. Here are the steps Gemini gives us to solve our problem:

- Install dependencies (`Flask-WTF`).
- Create a template.
- Create routes and logic.

It does also give a good explanation of the code, and what's happening, which is useful. We should consider this solution as we implement the interface for the application. Let's see what our third option, Blackbox AI, suggests for this problem.

7.1.3 Blackbox AI results

Here is the output from Blackbox AI, exactly as it was returned:

Here is a possible implementation of the quiz application:
First, let's create the necessary routes and templates.
app.py:

```
from flask import Flask, render_template, request, redirect, url_for
from quiz_db import get_questions, store_answer

app = Flask(__name__)

@app.route('/')
def index():
    return redirect(url_for('quiz'))

@app.route('/quiz', methods=['GET', 'POST'])
def quiz():
    question_id = request.args.get('question_id', 1)
    question = get_questions(question_id)
    if request.method == 'POST':
        selected_answer = request.form['answer']
        store_answer(question_id, selected_answer)
        if question_id < 35:
            return redirect(url_for('quiz', question_id=question_id + 1))
        else:
            return redirect(url_for('results'))
    return render_template('quiz.html', question=question)

@app.route('/results')
def results():
    return render_template('results.html')

if __name__ == '__main__':
    app.run(debug=True)
```

quiz_db.py:

```
import sqlite3

def get_questions(question_id):
```

```
    conn = sqlite3.connect('quiz.db')
    cursor = conn.cursor()
    cursor.execute('SELECT * FROM questions WHERE id=?', (question_id,))
    question = cursor.fetchone()
    conn.close()
    return question

def store_answer(question_id, selected_answer):
    conn = sqlite3.connect('quiz.db')
    cursor = conn.cursor()
    cursor.execute('INSERT INTO answers
(question_id, selected_answer)
VALUES (?, ?)', (question_id, selected_answer))
    conn.commit()
    conn.close()
```

templates/quiz.html:

```
{% extends 'base.html' %}

{% block content %}
    <h1>{{ question[2] }}</h1>
    <form method="post">
        <input type="radio" name="answer"
value="{{ question[3] }}">{{ question[3] }}</input><br>
        <input type="radio" name="answer"
value="{{ question[4] }}">{{ question[4] }}</input><br>
        <input type="radio" name="answer"
value="{{ question[5] }}">{{ question[5] }}</input><br>
        <input type="radio" name="answer"
value="{{ question[6] }}">{{ question[6] }}</input><br>
        <input type="submit" value="Submit">
    </form>
{% endblock %}
```

templates/results.html:

```
{% extends 'base.html' %}

{% block content %}
    <h1>Quiz Results</h1>
    <p>Congratulations! You have completed the quiz.</p>
{% endblock %}
```

templates/base.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Quiz Application</title>
</head>
<body>
```

```
    {% block content %}{% endblock %}
</body>
</html>
```

This implementation uses Flask's routing system to handle different parts of the quiz. The quiz route displays the current question and possible answers using a Flask template. The results route displays the quiz results. The `store_answer` function stores the user's selected answer in the database. The `get_questions` function retrieves the current question from the database. The quiz iterates through the questions using the `question_id` parameter in the URL. The user's selected answer is captured using a form and stored in the database using the `store_answer` function. After storing the answer, the quiz proceeds to the next question. The quiz ends when all questions have been answered, and the user is redirected to the results page.

Out of all the results, the Blackbox one is my favorite. It doesn't bother with any of the database or setup and jumps directly to generating code for the frontend pieces of the application. There are some modifications we can make to the prompt if we aren't seeing the results geared toward the frontend design. But Blackbox created exactly what we're looking for. It suggested the following steps:

- Create routes.
- Create a database connection and query it.
- Create templates.

I'll use a combination of all the advice from each tool but start with the Blackbox results. I'll likely end up pulling some suggestions from each.

REFINEMENT

There are some steps we could take with each of these tools if we want to drill down and get more detail about UI information. For example, in the objective section, we used

JM Objective: Develop a front end for a database-driven Flask application for a quiz system. The application will display questions and possible answers to the user, capture their selected answer, store the answer in a database, and then show the next question. Flask has been installed and the backend is already functional.

We could add further refinement and be more specific:

JM Objective: Develop a visually appealing and user-friendly front-end interface for a database-driven Flask application for a quiz system. The interface should display questions and possible answers to the user in a clear and intuitive manner, capture their selected answer, store the answer in a database, and then show the next question. The design should prioritize simplicity, a light color scheme, and mobile compatibility to ensure a seamless user experience. Flask has been installed, and the backend is already functional.

This small change can drastically alter the output. It's acceptable to start out with abstract requirements, but if you aren't getting what you want, refine your prompt.

Tips for prompt refinement

- *Use specific examples*. Instead of vague ideas, be precise.
- *Quantify whenever possible*. Use numbers, measurements, or statistics.
- *Choose active verbs*. Weak and passive verbs obscure your intent.
- *Remove unnecessary modifiers*. Don't send words (or "tokens" in LLM speak) that don't add value to the request.
- *Be direct*. Avoid hedging or qualifiers

We won't spend more time refining our prompt for each tool. There are some great results we can work with, so let's put those into action.

7.2 Creating our templates

The first thing we'll do is create our templates. Templates allow you to build dynamic HTML with Flask. They're HTML files with placeholders for dynamic content. This allows you to create a layout with HTML and then add placeholders where Python will come in and fill dynamic content. There will be a base html file that creates an entire page, and then each of these templates will make up a portion of that page. Together, we get HTML generated with dynamic content. The templates are compiled by the Jinja2 template engine. This is a very powerful framework, though we'll only be using some basic parts of it. Let's create some template files based on Blackbox AI recommendations.

First, we'll create a file named `quiz.html` in the `/app/templates` folder. Here's what we need this template to do. The template should

- Create an HTML form.
- Provide the question being asked.
- Provide a list of possible answers.
- Have a way to choose the answer.
- Have a submit button.

The suggestion from Blackbox looks like it's providing this information. We can see a content block and the form elements we need. It also includes placeholders for dynamic content that's coming from a question object:

templates/quiz.html

```
{% extends 'base.html' %}

{% block content %}
    <h1>{{ question[2] }}</h1>
    <form method="post">
        <input type="radio" name="answer"
value="{{ question[3] }}">{{ question[3] }}</input><br>
        <input type="radio" name="answer"
value="{{ question[4] }}">{{ question[4] }}</input><br>
        <input type="radio" name="answer"
```

```
value="{{ question[5] }}">{{ question[5] }}</input><br>
        <input type="radio" name="answer"
value="{{ question[6] }}">{{ question[6] }}</input><br>
        <input type="submit" value="Submit">
    </form>
{% endblock %}
```

Next, we need to create a results template, located at `templates/results.html`. We'll use the Blackbox suggestion for this one as well. However, you can quickly see that it doesn't provide any useful results. It simply states that the quiz is complete. We'll fix that later, though.

templates/results.html

```
{% extends 'base.html' %}

{% block content %}
    <h1>Quiz Results</h1>
    <p>Congratulations! You have completed the quiz.</p>
{% endblock %}
```

Finally, we have our base html, which is the wrapper for the main web page. This is what provides the full-page experience including header and footer, and similar. Blackbox suggests naming this `templates/base.html`; however, we already have an `index.html` that serves this purpose. Let's look at what `base.html` looks like:

templates/base.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Quiz Application</title>
</head>
<body>
    {% block content %}{% endblock %}
</body>
</html>
```

Instead of adding another file with a similar purpose, we'll use our existing `index.html`. However, we need to change the `extends` statements in each of our template files to reflect that. Then we'll need to insert our `content` into `index.html`.

In the template files `quiz.html` and `results.html`, we'll change

```
{% extends 'base.html' %}
```

to

```
{% extends 'index.html' %}
```

And we will add our new `Index.html` in a content block, though it won't do anything for us just yet. We will be coming back to this file and making some modifications. Here's what the `index.html` file looks like:

```
<!-- HTML template for the index page -->
<!DOCTYPE html>
<html>
<head>
    <title>Index Page</title>
</head>
<body>

    <h1>Ham Radio Practice Test</h1>
    <!-- main content-->
    {% block content %}{% endblock %}

    {% if session_id %}
    <h3>Session: {{session_id}}</h3>
    <button onclick="deleteCookieAndRefresh()">End This Session</button>
    <script>
        function deleteCookieAndRefresh() {
        // Client-side JavaScript to delete the cookie and refresh
        document.cookie = "session_id=;
expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
        window.location.reload();
        }
    </script>
    {% endif %}
    {% if data %}
    <ul>
        {% for item in data %}
            <li>{{ item }}</li>
        {% endfor %}
    </ul>
    {% endif %}
</body>
</html>
```

With our templates created, we will wire up the application. This guidance was important for where to go next. Generative AI tools can be great for guidance in areas you aren't familiar with or helping get you unstuck from a problem.

We won't go through every step involved as the point of this book isn't a step-by-step tutorial for building an application, but rather techniques we can use to improve the application. Let's look at how we can use AI to get ideas for our user interface.

Using AI tools throughout the design process

Different AI tools excel at various stages of UI development:

- *Early conceptualization*—Use chat-based LLMs to explore layout possibilities and design patterns.

(continued)

- *Flow visualization*—Request Mermaid or other diagram formats to map user journeys.
- *Wireframing*—Ask for text-based wireframes before committing to specific designs.
- *HTML/CSS generation*—Once the design is settled, request implementation code.
- *Troubleshooting*—If your implementation has problems, share screenshots or code snippets for suggestions.
- *Documentation*—Generate clear descriptions of your UI components for team members or users.
- *Consider tool strengths*—Some platforms excel at code generation, while others provide better conceptual guidance.

7.3 Describing the flow of our application

Right now, our application is running, but the user interface is as minimal as possible. That's fine with me. It doesn't need to be complex. However, I am open to suggestions to make it look nicer, and possibly easier to use. Figure 7.2 shows what it looks like right now.

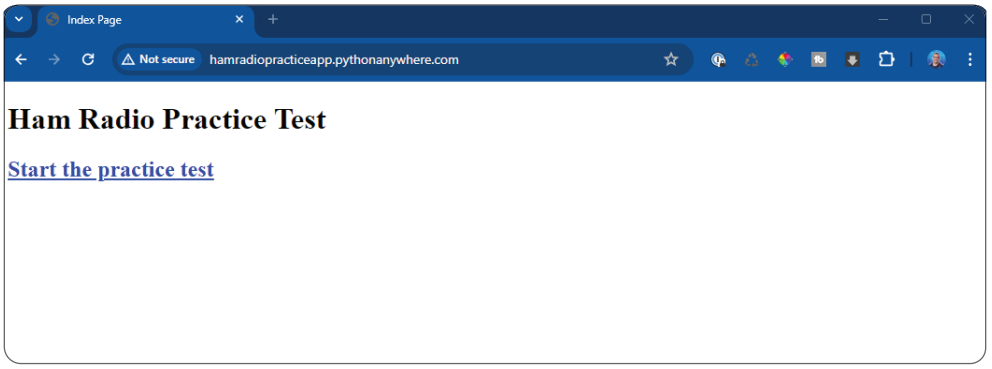

Figure 7.2 The entrance page to our HAM radio practice test

When you start a test, it doesn't get much better. It looks like a web page from the 1990s. I'm not much of a designer, but I'm confident we can use generative AI to make it look better and possibly easier to use (figure 7.3).

The first thing we should do is think about the flow and describe it somehow. It's a simple flow, but I want to avoid the tunnel vision that developers sometimes suffer from. We think it's perfect and easy to use, until someone else tries to use it and hates it. Figure 7.4 shows what our flow looks like right now.

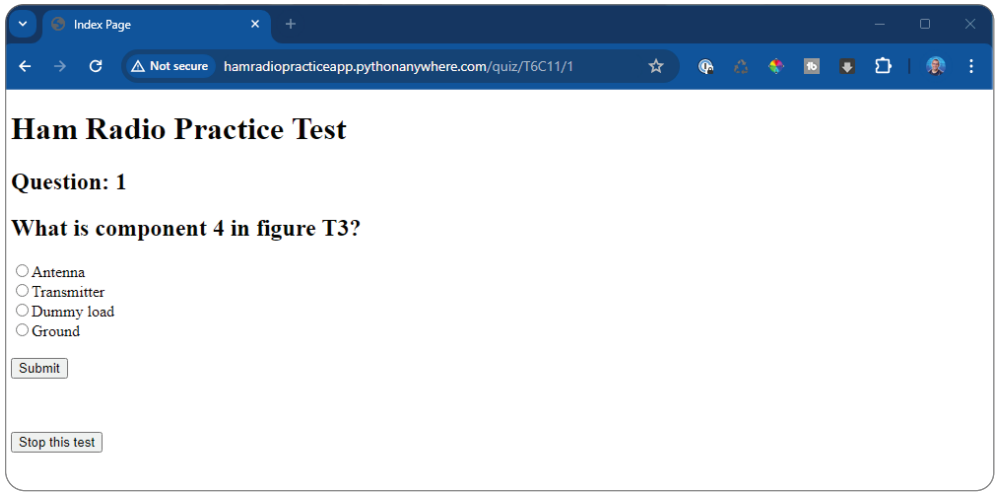

Figure 7.3 The question interface for our practice test

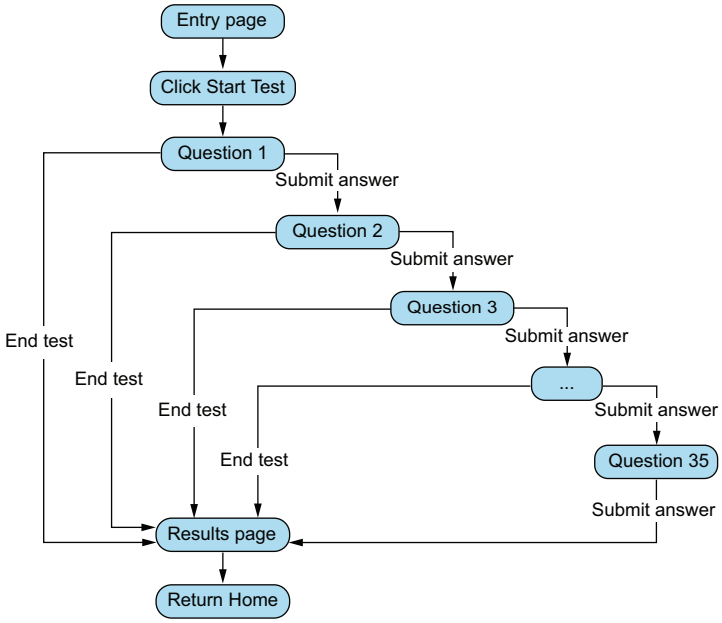

Figure 7.4 Our application's workflow

Figure 7.4 is a graphical representation of the flow. We start at the entry page, and our only choice is to take a test. Once we start the test, our choice is either to answer the question or end the test. Either way we're routed to a results page. The document

shows we can end the test at any time. This is a nice little flowchart that I created in around 30 seconds. I'll show you how I did that.

ASK CHATGPT FOR A MERMAID FLOWCHART BASED ON A DESCRIPTION

I could put the chart together by hand, but I want to save time. Here's a way I generate charts quickly when needed. I deliberately kept the description for the prompt as crude as possible to show how well ChatGPT handles the task. Remember, natural language processing, or NLP, is a core duty of ChatGPT, and it does well.

Here is the prompt I sent:

JM

```
Please create a Mermaid diagram for this process:
Entry Page (start the practice test)
    click start test
A page presents question 1
    Choice: submit answer or end test
    if user chooses to submit answer this continues from question 1–35
    user can exit the test any time, and is sent to results page
    after question 35 (finished test)
    user is sent to results page
results page
    Results page is shown with the results
    And a link to return home
```

If you're unfamiliar with Mermaid, it's a neat markup language like Markdown for building flowcharts. There are several tools for converting this text to diagrams. ChatGPT can generate Mermaid charts easily, and here's what it produced based on the previous prompt :

```
graph TD
    A[Entry Page (start the practice test)] –> B[Click Start Test]
    B –> C[Question 1]
    C –> |Submit Answer| D[Question 2]
    D –> |Submit Answer| E[Question 3]
    E –> |Submit Answer| F[...]
    F –> |Submit Answer| G[Question 35]
    C –> |End Test| H[Results Page]
    D –> |End Test| H
    E –> |End Test| H
    F –> |End Test| H
    G –> |Submit Answer| H
    H –> I[Results Page]
    I –> J[Return Home]
```

Even if you've never seen Mermaid before, you can make sense of the answer by looking at it. It can get pretty complex though, and one good resource is this Mermaid cheat sheet (https://mng.bz/rZMB) that shows many different options.

After reviewing this Mermaid chart, I imported it into Excalidraw (https://excalidraw.com/), and it created this diagram instantly, the exact one you see in figure 7.4. There are several tools that support Mermaid inputs, including Draw.io, and I highly recommend learning about Mermaid and the tools that use it.

CREATING A DOCUMENT BASED ON THIS CHART

I mentioned that I used a very crude description for my prompt. It happened to work; all of us have experienced that feeling that these tools read our mind and provide exactly what we ask for. We've also seen it go wrong. I was lucky this time, but it often requires tweaking or more contextual information. I decided to create an accompanying document with more detail I can refer to later. Also, it is something I can give to a UI designer, with more detail and in the format they expect.

So, here's what I sent as a prompt to ChatGPT, in the same conversation as the chart was drawn, for context:

JM Write out a document describing this flow that would make sense to a UI expert.

The prompt is short and sweet in this case, and here's what ChatGPT came up with:

Entry Page:

Step 1: Start Practice Test:

- Users land on the Entry Page where they can begin the practice test by clicking the "Start Test" button.

Question Pages:

Step 2.

Question 1:

After clicking "Start Test," users are taken to Question 1. Users are presented with a choice:

- **Submit Answer:** Upon submitting their answer, users proceed to the next question.
- **End Test:** Users can choose to end the test at any time, which will take them to the Results Page.

Question 2 to Question 34:

This process repeats from Question 2 through to Question 34:

- **Submit Answer:**
  - Users proceed to the subsequent question.
- **End Test:**
  - Users can end the test at any point, directing them to the Results Page.

Question 35 (end of test):

On reaching Question 35, users have the same options:

- **Submit Answer:**
  - Submitting the answer for Question 35 will direct users to the Results Page.
- **End Test:**
  - Users can still choose to end the test at this point, which will also direct them to the Results Page.

Results Page:

View Results: Users are presented with their test results after either completing all questions or choosing to end the test early. - From the Results Page, users have the option to return to the Home Page.

Return Home:

Return to Entry Page: After viewing their results, users can click a "Return Home" button to go back to the Entry Page, where they can start a new test if desired.

This document lays out the steps in a linear fashion. You can gather the flow of the application from this response, and this document is a great start. It will need some refinement with the UI expert to make sure it communicates intentions well.

You should never expect that these tools will generate the final product every time. In fact, you don't want to rely on finished products from LLMs and should always add a personal touch to whatever you generate. Just like with coding, these tools should augment your work, rather than replace you (even if they could).

Now let's make this application look nicer. The 1990s look isn't working well for this. I'm no designer myself, but I can use ChatGPT or other LLMs to generate some CSS and HTML for me that should make this site look great.

7.3.1 Creating an overall design

Our current look is raw HTML without any backgrounds, special fonts, or CSS involved. Even if this were an internal product, we want it to look better than this. I'm certain we'll stick with a basic one-column layout, but I want to try some new ideas first. I'll take the previous results describing the UI layout and add them to implement my prompt. I will keep this in the same conversation on ChatGPT. This maintains context, and recently, ChatGPT has introduced memory as a feature, so it keeps track of all your conversations and results. By working within this context and keeping the conversation in a thread, you can improve results. It also helps you, the person using it, remember what has transpired throughout the thread. I'll add this to my initial prompt thread. Here is the next prompt I will use:

JM Generate layout concepts for a HAM radio Practice Test website targeting beginners who are not tech-savvy. The website should feature a clean, uncluttered, and professional design, optimized for desktop use.

Key Requirements:

Multiple-Choice Quiz Format: Present questions in a clear, easy-to-navigate multiple-choice format.

Results Display: Provide a clear and neat display of the results upon completion of the test.

User-Friendly Interface: Ensure the layout is intuitive and straightforward to use, accommodating users with minimal technical experience.

Design Aesthetic:

Uncluttered and professional appearance

Simple, clean lines and ample white space

Consistent use of a professional color scheme

Description of UI: [UI Description from above]

A quick note before we go on. You may have noticed that I mentioned targeting beginner users who are not tech savvy. You may be thinking, "What? HAM radio operators who aren't tech savvy? That doesn't make sense." It's nothing personal toward the audience. By adding this instruction into the prompt, I'm making sure that the interface will be easy to use for folks who are beginners. You can never assume the skillset of those using your site, so it's best not to exclude those who don't use the internet much. Let's see the result:

Entry Page:

Header:

Logo: Top-left corner.

Navigation: Home, About, Contact (simple, minimal links).

Professional color scheme (e.g., shades of blue and grey).

Main Content:

Title: "HAM Radio Practice Test"

Large, bold, and centered.

- **Introduction:** Brief description of the test.
- **Start Test Button:** Prominent, centered button labeled "Start Test".

Footer:

Simple footer with links to privacy policy and terms of service.

Question Pages:

Each question page should follow a consistent format:

Header:

Logo and minimal navigation.

- **Question Section:**

  Question Number: "Question 1 of 35"

  Large, bold, and centered.
- **Question Text:** Clearly displayed, large font.

Multiple-Choice Options:

Radio buttons for selection.

Clear, spaced out, and easy to read.

- **Actions:**
  - **Submit Answer Button:** Prominent and centered.
  - **End Test Button:** Subtle but accessible, perhaps in a lighter shade.

Results Page:

Header:

Logo and minimal navigation.

Main Content:

Title: "Your Results"

Large, bold, and centered.

- **Score Display:** Clear and prominent display of the user's score.

Detailed Breakdown:

Number of correct and incorrect answers.

Option to review the answers.

- **Actions:**
  - **Return Home Button:** Prominent button to take users back to the Entry Page.
  - **Start New Test Button:** Optional, for quick retakes.
- **Design Aesthetic:**
  - **Color Scheme:** Professional colors like blue, grey, and white.
  - **Typography:** Clean, sans-serif fonts for readability.
  - **White Space:** Ample spacing around elements to avoid clutter.
  - **Consistent Elements:** Uniform button styles, headings, and text alignment.

This is great information. It's the beginning of a style guide you may have seen in enterprise development. It's a great set of rules and basic guidance for designing our application. But there's a bonus that ChatGPT has given us: wireframes. Wireframes are the barebone visualization of an interface. They convey what each page or section of an application will look like. They usually don't contain things like colors or graphics. The intention is simply to show where elements live in the layout. It shows things such as text inputs, buttons, and dividers and where they live on the page.

Wireframes are helpful for UI/UX folks (user interface/user experience). They often move these elements to make the interface easier to use. They're also helpful for programmers, as most of us can put together HTML well enough to create these pages and add each element in its proper place.

I didn't ask for wireframes in my initial prompt, I didn't know that was a possibility. But now that we know ChatGPT can generate wireframes, that can be something we can

add to our toolbox with future prompts. Here are the wireframes ChatGPT generated for us.

Entry Page

The first wireframe will be our entry page. This is the very first page a user sees when they enter our application. We can see a dialog drawn out showing how we should lay this out. There is a header with our logo, home, about, and contact links, as well as a footer with Privacy Policy and Terms. In the center is our welcome text and a single link to start the test (Figure 7.5).

Entry page

plaintext Copy code

```
+-----------------------------------+
| LOGO       Home  About  Contact   |
+-----------------------------------+
|                                   |
|       HAM Radio Practice Test     |
|                                   |
|   Welcome to the HAM Radio        |
|   Practice Test. Click below to   |
|   start your test.                |
|                                   |
|           [Start Test]            |
|                                   |
+-----------------------------------+
|      Privacy Policy | Terms       |
+-----------------------------------+
```

Figure 7.5 This is a text wireframe diagram of the entry page. It is rough but gives us a good idea of what the page should look like.

Question Page

Our question page is the primary page for answering quiz questions. Notice the LOGO, Home, About, and Contact are still here. These are what common elements we can carry on throughout the application, and I'll show how to do that. This page is repeated for each question, the only change being the inner content itself. This page will be repeated 35 times, for each question. There is an "end test" button so the user can stop the test any time, as we've asked for in our requirements (figure 7.6).

Question page

```plaintext
+-----------------------------------+
| LOGO       Home  About  Contact   |
+-----------------------------------+
|                                   |
|         Question 1 of 35          |
|                                   |
|   What is the frequency of...     |
|                                   |
|   ( ) Option A                    |
|   ( ) Option B                    |
|   ( ) Option C                    |
|   ( ) Option D                    |
|                                   |
|   [Submit Answer]    [End Test]   |
|                                   |
+-----------------------------------+
```

Figure 7.6 The wireframe for the question portion of the quiz. This is repeated for each question.

RESULTS PAGE

This is the final page containing the test results. Users will get here after completing the test, or if they choose to exit the test early. The results are tallied and displayed on the screen. They also have the option to return home or start a new test (figure 7.7).

Results page

```plaintext
+-----------------------------------+
| LOGO       Home  About  Contact   |
+-----------------------------------+
|                                   |
|            Your Results           |
|                                   |
|   Score: 28/35                    |
|                                   |
|   Correct Answers: 28             |
|   Incorrect Answers: 7            |
|                                   |
|   [Return Home]   [Start New Test]|
|                                   |
+-----------------------------------+
```

Figure 7.7 This is our results page. We'll likely add in the percentage as that's a key part of our quiz.

The initial wireframes provided aren't much different than what we have now but some added details that are important. I will use the context of this conversation to use ChatGPT to generate some code for us, based on these wireframes.

7.3.2 Drafting HTML based on our wireframes

Our application uses Flask templating, and ChatGPT is unaware of that because I haven't mentioned it in this chat thread. I have extensive HTML experience, so I'm just going to ask for HTML suggestions, and then wire up the Flask components. It's a personal choice. It will be faster for me, and I won't have to give ChatGPT as much context about the templates themselves. If you're following along and want to have ChatGPT generate Flask templates, I'm sure that will work fine as well.

The first thing I noticed is a common header. It has a Logo and Home, About, and Contact links. I noticed it's common throughout the screens, so it's not something we want to repeat every time. We should make a header out of it. I'll ask ChatGPT for guidance here. Here is my prompt:

JM Create an HTML "Header" for these wireframes, that contains the logo and the top links that will be common for each page.

The results it returns are exactly what I'm looking for. It includes all the HTML that will be used in each page, so I can add it to my Flask header, and then, if I need to modify it, it will only be modified in one place. This saves me time from tracking down this HTML in several spots. It also generates some CSS, which I intend to use in the application anyway. Here is what ChatGPT returned:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>HAM Radio Practice Test</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 0;
        }
        .header {
            display: flex;
            justify-content: space-between;
            align-items: center;
            padding: 10px 20px;
            background-color: #f8f9fa;
            border-bottom: 1px solid #dee2e6;
        }
        .logo {
            font-size: 24px;
            font-weight: bold;
            color: #007bff;
```

```
                text-decoration: none;
            }
            .nav-links {
                display: flex;
                gap: 20px;
            }
            .nav-links a {
                text-decoration: none;
                color: #007bff;
                font-size: 16px;
            }
        </style>
    </head>
    <body>
        <div class="header">
            <a href="#" class="logo">HAM Radio Test</a>
            <div class="nav-links">
                <a href="#">Home</a>
                <a href="#">About</a>
                <a href="#">Contact</a>
            </div>
        </div>
    </body>
    </html>
```

While this may look like a lot of code, it's standard for an HTML header. This is a good example of how ChatGPT and other LLMs often produce an average or something that is very commonly found on the internet. Due to the statistical nature of LLMs, it's producing something that looks like a common HTML header, based on training data. It looks good to me, so I'll implement it into my `Index.html` page, as this is a wrapper around my application. I'll explain that for clarity.

In a Flask application, headers and footers often wrap some kind of inner content. Again, the reason for this is because the header and footer do not change as the pages in the application change. So instead of creating several pages and copying in a header and footer, Flask loads an `Index.html` file with this information and then changes the content in the middle. This design pattern has existed in HTML for decades, and Flask implements it in an easy way.

Our `Index.html` only contains content that's common among pages and swaps out content, depending on the URL (or route) we're using. Here's the code:

```
<!DOCTYPE html>
<html>
<head>
    <title>Index Page</title>
</head>
<body>
    <h1>Ham Radio Practice Test</h1>
    <!-- main content-->
    {% block content %}{% endblock %}
</body>
</html>
```

In this code, it's clear our title stays the same, as well as head and body tags. The H1 heading also stays the same. But everything within the content block can change from page to page. Figure 7.8 illustrates this concept.

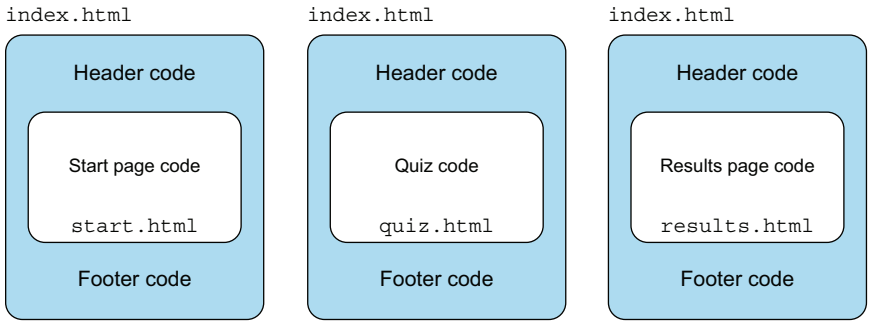

Figure 7.8 The `index.html` is common code within all the pages, but each section of the app has inner code from another html file to complete the page.

With this pattern in mind, I will insert the code into my `index.html` file. Then, within this same conversation (this is important), I will ask ChatGPT to generate the footer code as well. Here is the prompt:

JM Create footer html code with a copyright, year, and name.

ChatGPT then returns a snippet of code that is again fairly standard:

```
    <div class="footer">
        &copy; <span id="year"></span> Jeremy Morgan. All rights
reserved.
    </div>
    <script>
        document.getElementById("year").textContent = new Date().
getFullYear();
    </script>
```

I'll insert it into my application as well, also with the updated CSS for the footer. Our application is already looking cleaner and more professional (figure 7.9).

Now we need to craft some of the HTML for the other pages. I want to have a common box for them, so I'll include that box in my `index.html`. Everything that's inside that box will be dynamic and changed from page to page. Here's the prompt I will use for that:

JM Create html and CSS for an inner box (content wrapper) that will wrap the content for each page. I want it to take up approximately 75% of the screen and be placed between the header and footer.

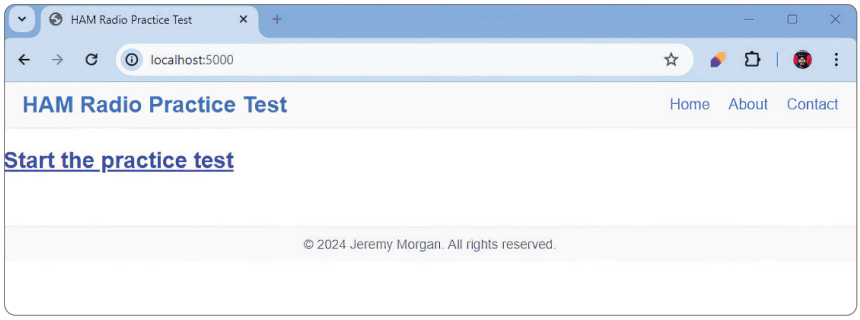

Figure 7.9 The new look of our application with a header and footer and CSS styling

This prompt returns the code that I need, including CSS, to wrap the content and finish out our changes in `index.html`. It returns this HTML:

```
<div class="content-wrapper">
        <!-- Page-specific content goes here -->
</div>
```

It also returns the following CSS:

```
.content-wrapper {
            width: 75%;
            background-color: #ffffff;
            padding: 20px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
            margin: 20px 0;
            flex: 1;
}
```

I'll drop this in, make a few CSS tweaks to it, and we'll examine our page, which now looks even more professional (figure 7.10).

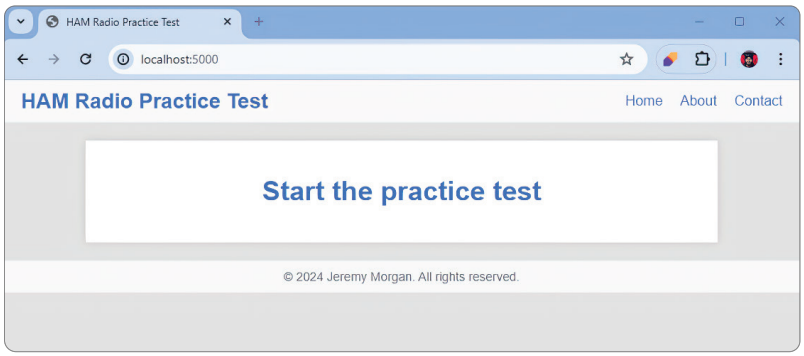

Figure 7.10 The application with additional HTML and CSS used for a cleaner look

To finish this out, I won't' go through each page one by one as it's too repetitive, but I used similar prompts for each page, such as

JM Create the inner HTML for the Question Page according to the wireframe
Create the inner HTML for the Results Page according to the wireframe

I took the generated HTML and inserted it into my Flask templates accordingly. I made a few minor CSS changes, and I'm happy with the result.

7.3.3 The final UI for our application

We generated a workflow diagram and then asked for some wireframes for our application. Next, I asked for HTML to match those wireframes. Though I didn't ask for CSS, ChatGPT generated that as well. Here's what our final application UI looks like. Figures 7.11, 7.12, and 7.13 show the entry, question, and results page, respectively.

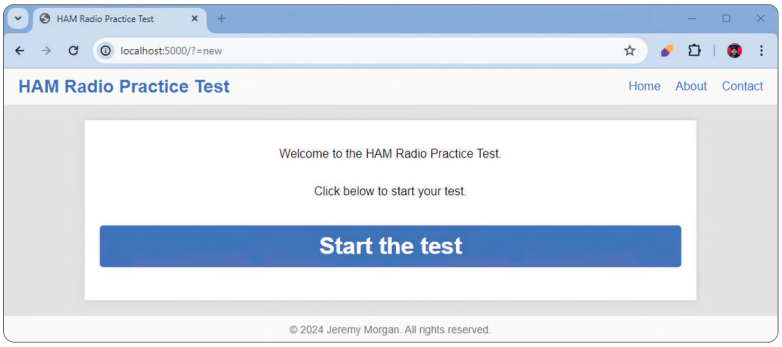

Figure 7.11 The entry page of our application

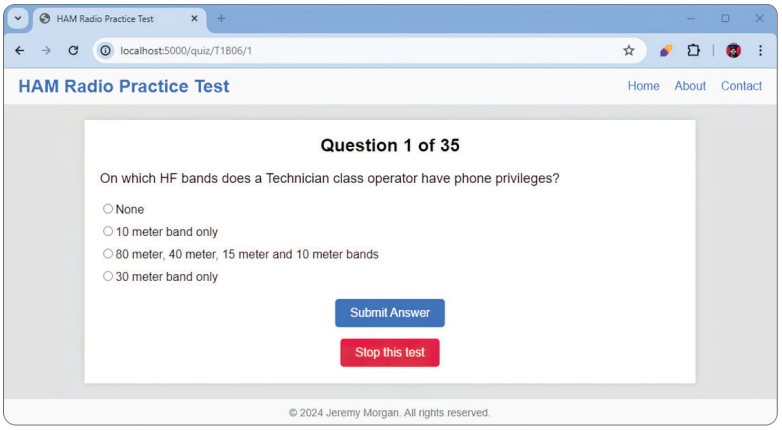

Figure 7.12 The question page of our application

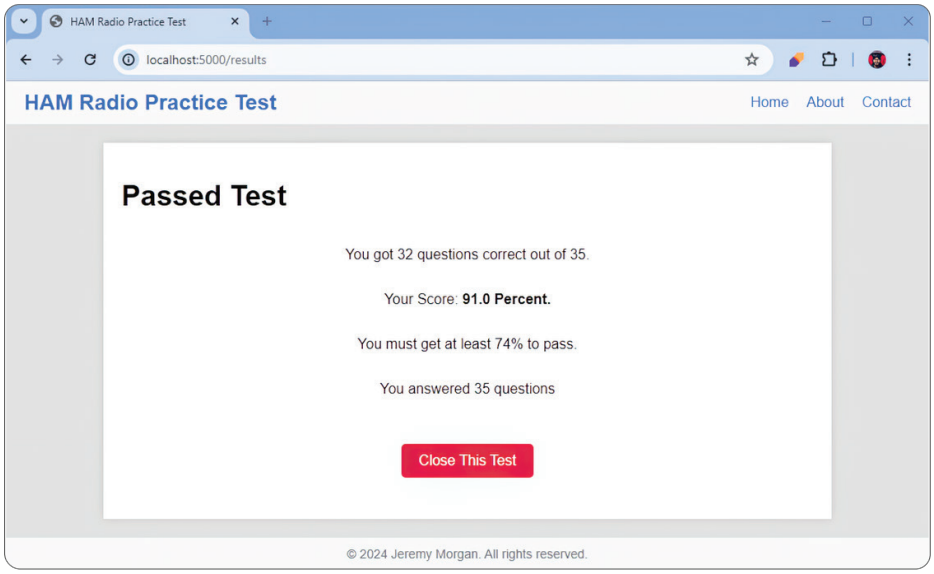

Figure 7.13 The result page of our application

Our finished application now has a clean, professional UI that improves the testing experience without adding complexity. This change shows how AI tools can enhance the design process.

We used chat-based AI interfaces—such as ChatGPT, Google Gemini, and Blackbox AI—to guide our UI development from idea to execution. We learned the quality of AI support relies heavily on how we phrase our prompts. We moved from broad design ideas to specific implementation details. The visualization tools were especially useful, as AI-created Mermaid flowcharts and wireframes helped us clarify user flows before writing code.

By having AI models create the basic HTML and CSS, we refined and integrated it into our Flask templates. This method produced results that usually need many design skills. It shows how generative AI can act as a design partner, speeding up development, while still allowing for the customization that makes an interface effective. As we add features to our application, we will keep building on this foundation. This will illustrate how AI tools can work well with developer insight to create applications that are both functional and visually attractive.

Summary

- Using detailed, specific prompts when requesting UI guidance produces more targeted, useful responses from AI tools.
- AI-generated flowcharts and wireframes can be employed to visualize user journeys before writing code

- You can progress from conceptual designs to implementation by having AI generate customizable template code.
- It is useful to adjust AI prompts incrementally, allowing outputs to remain responsive to the shifting nuances of UI design.
- AI-generated code should be balanced with personal design judgment, making strategic modifications to ensure the interface meets both functional and aesthetic requirements