4 Coding the first version of our application

This chapter covers

- Extracting software requirements from a design document
- Setting up a Python virtual environment
- Creating code stubs to lay out the application structure
- Organizing a Flask web application
- Running a simple Flask app for the first time

In this chapter, we're diving headfirst into the wild world of developing software with AI. By now, you've gotten a taste of what AI can do, and we're just getting started. We will use AI extensively in this chapter, and by the end, you'll be even more comfortable with integrating these tools into your workflow.

We begin our journey by building a useful application for HAM radio test preparation. We'll explore ChatGPT and Gemini and show how each can assist us as we build our project. In the following chapters, we'll use these and other new tools. As you progress, you'll develop a clearer sense of how and when to use the tools.

Creating software is like building a house. The foundation is the first step; you can't start without it. Building the rest of the house will be a struggle if the foundation doesn't meet the requirements (figure 4.1). If you don't have the time to be thoughtful and do it right, you won't have the time to fix it later.

Figure 4.1 The foundation is the most crucial part of your project. Generated with Midjourney, a generative AI image-building tool.

We will use generative AI to build a solid foundation for our application. We'll turn abstract design concepts into code. We're also going to focus on stubbing parts of our application. *Stubbing* is when you create a simplified piece of code, known as a "stub," that acts as a stand-in for functional code. Once you've built several stubs, you can connect them and run the application. Then, you fill in functionality to make each piece functional.

You've probably done this countless times in your career, but now there is the power of generative AI to help speed things along. Once you're comfortable using these tools for assistance, it will become a part of your application creation routine. Let's load up our tools and get started.

4.1 Stubbing: Building the skeleton of your application

Stubbing is a fundamental technique in software development where simplified placeholder versions of code components are created before implementing the full functionality. It is like building the frame of a house before adding the walls, plumbing, and electrical systems. The stubs provide a way to test the overall structure and flow of an application early on, without getting bogged down in the details of individual components. Some of the benefits of stubbing are

- *Early integration*—Stubbing allows you to integrate different parts of your application sooner, identifying potential compatibility problems early in the development process.
- *Parallel development*—Team members can work on different components simultaneously, using stubs to simulate dependencies.
- *Testability*—Stubs provide a controlled environment for testing, where specific components can be isolated and evaluated for functionality.
- *Faster iteration*—By focusing on the overall structure first, it is possible to iterate faster and make significant changes without rewriting large portions of code.

4.1.1 A simple code example

Let's say we're building an application that needs to connect to a database. Instead of immediately writing the complex database connection code, we can create a stub:

```
class DatabaseManager:
    def __init__(self):
        pass

    def connect_to_database(self):
        # TODO: Implement this method to connect to the database.
        print("Connecting to the database...") # Placeholder
        return True # Simulate successful connection

    def fetch_data(self, query):
        # TODO: Implement this method to fetch data from the database.
        print("Fetching data...") # Placeholder
        return [] # Simulate empty result set
```

In this example, `connect_to_database()` and `fetch_data()` are stubs. They don't actually connect to a database or fetch data. Instead, they print placeholder messages and return dummy values. This feature allows us to test the parts of our application that use the `DatabaseManager`, even without an active, fully functional database connection.

As we develop the application, we can gradually replace these stubs with real implementations. Such an iterative approach makes a development process more manageable and less prone to errors.

By starting with stubs, you create a working skeleton of your application, which enables you to test, refine, and build on a solid foundation. This is where generative AI tools can be incredibly helpful, quickly generating these initial stubs based on your specifications. Let's gather our requirements and build stubs from them.

Stubbing strategically

Effective stubbing creates a solid foundation:

- Create stubs for a feature before implementing it.
- Use generative AI to suggest method signatures and class structures.

- Start with empty functions to establish interfaces between components.
- Consider implementing one feature at a time rather than creating all stubs upfront.
- Use TODOs to document intent for each stub.
- Ensure that your stubbed application runs before adding functionality.
- Remember YAGNI (you aren't gonna need it): don't create stubs for features you might not implement.

4.2 Extracting requirements from the design

When building new software, the clarity and precision of project requirements are pivotal. Getting the requirements right is critical as they often determine whether a software project meets its deadlines or faces significant delays. Requirements always change. Also, they're frequently misinterpreted because we tend to grab the requirements and get to work. There is a lot of room for error here, so if we rush, we can get in trouble. Because generative AI tools make the requirements gathering process easier and faster, we can spend more time working on those requirements and getting them right.

The process of extracting requirements is not merely listing what the software should do, but it is about understanding and interpreting all stakeholders' goals, needs, and constraints. With generative AI, this is even more critical. Although adaptive, these models require a nuanced understanding of the desired outcomes to function optimally. Without precise, well-defined requirements, the tools will help you build a bad project faster.

Let's use generative AI to turn this design document into a set of requirements. The approach is straightforward, and it is helpful to go through it step by step. To extract the requirements using tools, I decided to use ChatGPT and Gemini.

Prompt engineering for requirements extraction

When using AI to extract requirements

- Start with clear, specific prompts that define the scope.
- Ask for structured output (e.g., numbered requirements, categorized lists).
- Request identification of potential pitfalls for each requirement.
- Use follow-up prompts to drill deeper into specific areas.
- Always validate the output with stakeholders or domain experts.
- Remember that AI can hallucinate requirements that seem plausible but aren't really needed.

Here is a three-step process that works well:

1. Ask for key requirements.
2. Convert to user stories.
3. Create stubs based on those stories.

4.2.1 Step 1: Extract the requirements using ChatGPT

Our design document is brief, so we could easily review it and extract the main requirements visually. It would only take a few minutes to complete. But what if the project and documents were larger and more complex? We're using ChatGPT to do this to save time. Moreover, ChatGPT helps with accuracy as it can catch errors or surface something you've missed.

Let's see how ChatGPT handles reviewing our document and narrowing the requirements. Not only is ChatGPT good at generating text, but it also can interpret and analyze it. It's likely seen tens of thousands of software design documents. It's also seen as many requirement lists. So, it should be able to come up with something reasonably good.

I copied everything from our design document as text. Then, I used the following prompt:

JM What are the key requirements we can obtain from this software design document? {Contents of document}

Granted, it's a basic prompt that's straight to the point. Will it produce something decent or steer us wrong? Here's the output:

From the provided software design document, the following key requirements can be inferred:

1. Question Pool Database
   a. Requirement 1.1: Develop a database, SQL or NoSQL, to store questions, options, and correct answers.
   b. Requirement 1.2: The database should contain fields for `question_id`, `question_text`, `options`, and `correct_option`.
   c. Requirement 1.3: Implement a system to update the database to ensure accuracy over time.
   d. Requirement 1.4: Ensure data validation and cleansing to prevent incorrect or poorly formed data from entering the system.
2. Question Selector
   a. Requirement 2.1: Develop a module to select 35 questions randomly from the database for each test.
   b. Requirement 2.2: Implement an algorithm to ensure true randomness in question selection.
   c. Requirement 2.3: Ensure that the same question does not appear more than once in a single test.
3. User Interface
   a. Requirement 3.1: Develop a user interface, which could be a web, desktop, or mobile application.
   b. Requirement 3.2: The UI should be intuitive, user-friendly, and accessible to cater to a wider audience.

   c. Requirement 3.3: The UI should present questions and answer choices, accept users' answers, and present the score at the end of the test.

4. Grading Component
   a. Requirement 4.1: Develop a component to compare user's answers with the correct answers from the database and calculate the final score in percentage.
   b. Requirement 4.2: Handle cases where the user doesn't answer some questions, considering them as incorrect for scoring purposes.
   c. Requirement 4.3: Handle any inconsistencies or inaccuracies in grading due to discrepancies in the database.

5. Future Considerations
   a. Requirement 5.1: Consider adding functionality for tracking user performance over time and providing performance analytics.
   b. Requirement 5.2: Consider adding features allowing users to focus on specific areas or question types.
   c. Requirement 5.3: Explore the possibility of integrating the application with a larger learning platform.

6. General Requirements
   a. Requirement 6.1: The application should be robust and efficient in design, providing a user-friendly tool to help users prepare for their HAM radio license test.
   b. Requirement 6.2: Consider accessibility needs in all components to ensure the widest possible audience can use the application.
   c. Requirement 6.3: Ensure the effectiveness of study sessions by avoiding frustrations related to poor UI or inaccuracies in question/answer data.

This list might not be exhaustive, and additional requirements may arise as the design and development process progresses. However, these are the identifiable key requirements based on the provided document.

This looks great, and I'm pleased with the result. It provides enough detail to assign tasks to some of these requirements. These are high-level requirements, and they represent our intention well. At this point, it's good practice to bring these back to your stakeholders and validate the data. A quick meeting to make sure you're on the right track will save you time in the long run. There is a chance a generative AI tool hallucinates and produces a result that looks good to you but is complete nonsense. Human reviews are key.

In this case, I'm the stakeholder. Although I am happy with this output, I'd like to get a second opinion. This is a big project, so exploring more options is worth the time. I turn to Google's Gemini for comparison.

4.2.2 Step 2: Gathering requirements from Gemini

I sent the same prompt to Gemini to see how it would interpret our design document.

JM What are the key requirements we can obtain from this software design document? {Contents of document}

The results were unexpected. While ChatGPT and Gemini often produce the same output output, there are times when it's drastically different. This is the nature of generative AI. Each model will interpret input or prompts differently. How they generate output also varies wildly.

You will often encounter similar output across tasks. It even happens with code generation. Why? This is likely because many folks have published code with similar patterns. The model determines the right answer based on how often it has seen it. Whether we admit it or not, we get echoes of the same output because so many developers solve these problems in nearly identical ways. There are only so many ways you can connect to a database!

Don't be thrown off by highly similar outputs from separate tools. It's common. Also, be prepared for things to be far different between tools, which is good. This is what Gemini put out as a response:

From the software design document, the following key requirements can be obtained:

- The application should generate randomized 35-question tests from a predetermined question pool.
- The application should present possible answers to each question.
- The application should provide a score percentage at the end of each test.
- The question pool database should be able to store all the possible questions, along with their multiple-choice options and correct answers.
- The question selector module should be able to use a randomizing algorithm to select 35 questions from the question pool database for each test.
- The User Interface (UI) should be intuitive and user-friendly and should present the questions and answer choices to the user, accept their answers, and present their score at the end of the test.
- The grading component should be able to compare the user's answers with the correct answers from the question pool database to calculate a final score, which will be expressed as a percentage.

Additionally, the software design document also mentions some potential future considerations, such as

1. Tracking user performance over time and providing performance analytics.
2. Allowing users to focus on specific areas or question types.
3. Integrating with a larger learning platform.

These future considerations are not essential requirements for the initial version of the application but could be added over time to increase its utility.

This response looks drastically different from what ChatGPT produced, which is to be expected. Each flavor of GPT has its own personality. Generative tools often produce divergent outputs because the methodology used to generate results differs. Gemini's results are shorter in length, omitting details. It's a level of abstraction higher than the ChatGPT results.

So, I used another source to evaluate our design documents and get a second opinion. If you have multiple tools at your disposal, it helps to cross-check them, even if the output from the first tool is good. A second tool can reveal things you haven't thought of. It can also help validate the core parts of a design. If both models create the same topic or idea, it's more likely that it should be included. That extra bit of research adds value, and it justified the time spent to investigate.

Verifying AI-generated requirements

Critical steps when reviewing AI-generated requirements are

- Cross-reference with original project documentation.
- Validate with stakeholders, especially for technical requirements.
- Look for hallucinated features (things that seem plausible but weren't in your original plan).
- Check for consistency across different sections.
- Ensure all requirements are testable and specific.
- Be wary of overly generic requirements that don't add value.
- Consider how each requirement affects your development timeline.

Always get a second opinion

For crucial generative AI tasks, I duplicate my efforts across tools. We tested model divergence by inserting a similar input into two separate models and obtained different results. This process is a checking mechanism. It seems more likely to be true if you see the same result twice. If the results contradict each other, you can investigate why. Some models will bring up things the other model "didn't think of." These are some reasons why running another tool is worth the time spent.

Which one should we choose?

If we wanted to, we could dig deep into the details and gather two large lists. We could merge the results from ChatGPT and Gemini just like we do with source code. It's best at this point to go back to the stakeholder(s) and do a quick review. Have them help refine and distill a single set of requirements. This is a small, basic project, and I'm the only stakeholder. We're all set to move forward from here.

After careful consideration, I think the ChatGPT-generated list will be easier to work with. It's far more detailed and includes concrete requirements. The document outlines our overall goal well. With enough work, we could have squeezed more detail from Gemini to little benefit.

We'll use this document as a checklist for things our application must do, so the extra information from the ChatGPT output will be helpful. Now that we have requirements in our hands, we're ready to build some stuff.

The power of multiple AI tools

When working with generative AI, using multiple tools can provide significant advantages:

- Different models interpret prompts differently, offering unique perspectives.
- Seeing the same concept repeated across tools increases confidence in its importance.
- Contrasting outputs helps identify gaps or inconsistencies.
- Some tools excel at specific tasks (code generation, analysis, etc.).
- The extra time spent is often worth the additional insights.

Remember that each model has its own personality and strengths. What seems like an inconsistency between tools is in fact an opportunity to get a more well-rounded view of your problem.

4.3 Setting up our development environment

If you want to follow along with this book and create the application (you should), here is how we'll set it up. One great thing about developing in the Python world is the cross-platform functionality. You can run this on Windows, Mac, or Linux, and the setup won't be drastically different. For our demo, I'm using

- Windows 11
- Python 3.11.5
- Visual Studio Code
- GitHub Copilot

You'll need a dedicated directory on your hard drive to serve as the home for our project. I'm naming mine `HAM-Radio-Practice-Web`, but you can name yours whatever you like.

In Visual Studio Code, open the folder as a new project. You can do this by navigating to File > Open Folder or by clicking the Open Folder button in your Explorer panel (figure 4.2).

It's just an empty folder for now, but we'll need to open a command prompt or terminal for this folder. You can open a new command prompt in Windows or use a terminal in Linux or Mac. Visual Studio Code can also include a nice terminal window in the editor, which I prefer. To open a terminal window, in the top menu, select Terminal, then New Window. You can also use Ctrl + Shift + ‘ to open it. When developing in Windows, I prefer using WSL (Windows Subsystem for Linux), with Ubuntu in a VS Code terminal (figure 4.3). This approach allows me to run the powerful Linux commands I prefer and work efficiently using Windows as my desktop.

If you're more comfortable with Windows, you can also use a PowerShell or Command line terminal in VS Code. The commands and usage are very similar. Use whatever works best for you.

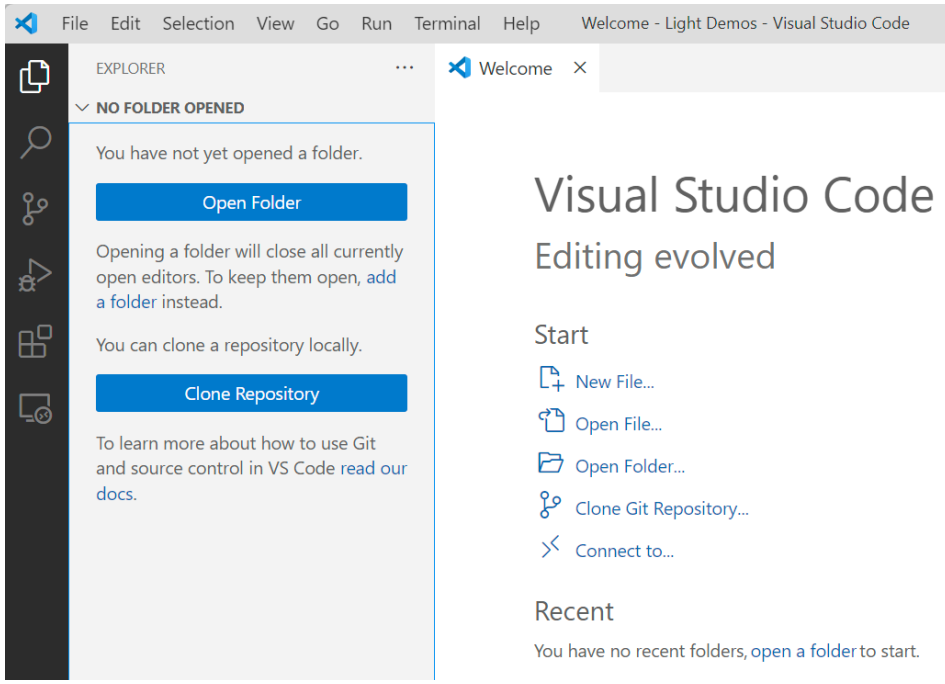

Figure 4.2 Opening a new folder in Visual Studio Code

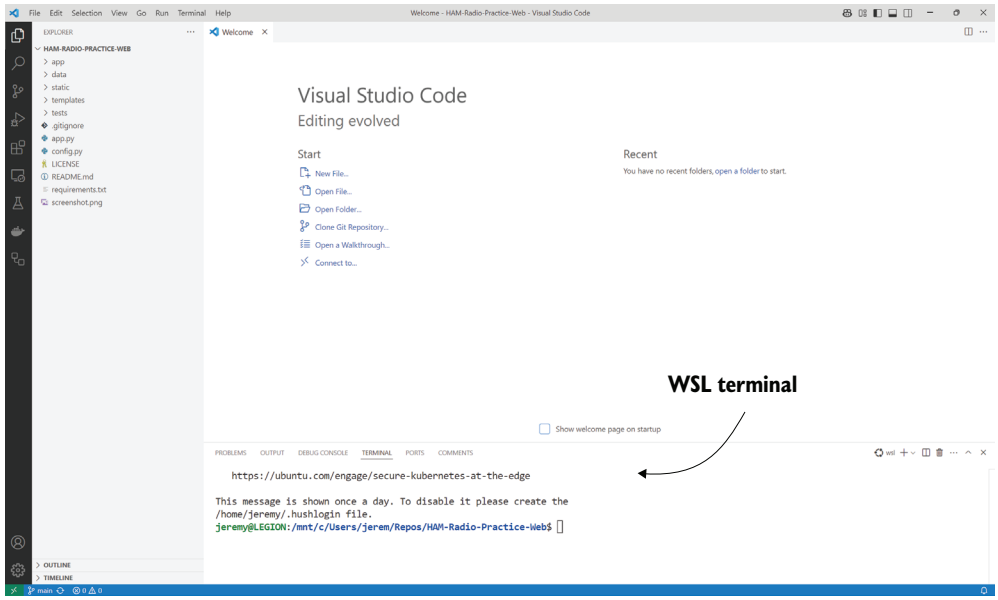

Figure 4.3 A WSL terminal running in Visual Studio Code

CREATING A PYTHON VIRTUAL ENVIRONMENT

Again, we will set up a Python virtual environment. I'll do this for every application we build, as the benefits are worth the time spent on it.

For Linux and Mac systems, the command is

```
python3 -m venv hrpractice
```

I chose `hrpractice` as the name, but you can choose something different. Then, we activate the environment by using

```
source hrpractice/bin/activate
```

You can verify you're in the environment by looking for the name in parentheses in your prompt, shown in figure 4.4.

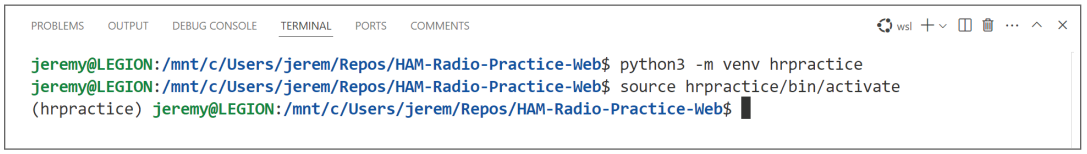

Figure 4.4 Prompt showing the environment is active in Linux/Mac

This prompt shows your environment is active.

In Windows, it's a little different. You still use the same command to create the environment:

```
python -m venv hrpractice
```

Note that on some systems, you may need to specify the exact path to Python in Windows, such as

```
C:\Python311\python.exe -m venv hrpractice
```

To activate, call the activate script in the Includes folder:

```
.\hrpractice\Scripts\activate
```

Of course, `hrpractice` is the name I chose for the environment, so your folder may be different. Now you should see the same environment name in parentheses at your prompt (figure 4.5).

Now, we have our environment set up and ready to go. I will use Visual Studio Code and GitHub Copilot in this chapter. The instructions for installing Copilot can be found at https://mng.bz/OwXK. Now, let's start organizing our Flask application.

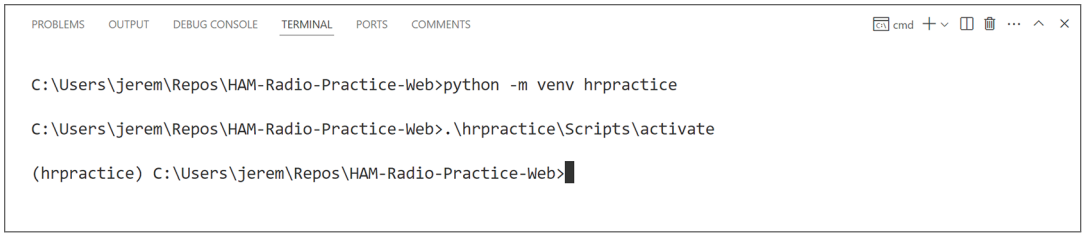

Figure 4.5 Prompt showing the environment is active in Windows

4.4 Flask application structure

I have chosen Flask as the framework for our application. We could easily install Flask, create a single Python file, and be up and running. As we add functionality, we just put more code in the file, save, and restart it. This is fine for small applications built in a hurry. However, if you ever want to expand the application later, it quickly becomes a mess.

How can we structure this application in a nice, organized way? Let's ask ChatGPT. I'll ask my question in the same context (thread) that I used to generate the software documentation. It's important to keep these threads open in ChatGPT if you want it to remember the details of your conversation.

Here is the prompt I entered:

JM This will be a Flask application. Show the best folder structure and file name suggestions for this application.

In this case, the model already has the context of our application, so it should be able to layout a decent structure. Let's see what it generates.

Output from ChatGPT

Assuming you are building a relatively standard Flask application, the folder structure presented in figure 4.6 could be used:

Here's a brief overview of what each part does:

- `app/_init_.py`—This file initializes your Flask application.
- `app/routes.py`—This file defines the routes (URLs) for your application and the logic associated with them.
- `app/templates/`—This folder contains HTML templates that your application will render, which in this case is an index (landing) page, a test page for taking tests, and a results page for showing the scores.
- `app/static/`—This folder contains static files such as CSS for styling, JavaScript for interactivity, and any images your application might use.
- `app/models/questions.py`—This file defines the structure of your question model and interacts with your database.

```
arduino                                                    Copy    Edit
HAMRadioPracticeTest/
│
├── app/
│   ├── __init__.py
│   ├── routes.py
│   ├── templates/
│   │   ├── index.html
│   │   ├── test.html
│   │   └── results.html
│   ├── static/
│   │   ├── css/
│   │   ├── js/
│   │   └── img/
│   ├── models/
│   │   └── questions.py
│   └── utils/
│       └── question_selector.py
│
├── tests/
│   ├── __init__.py
│   └── test_routes.py
│
├── config.py
├── run.py
└── requirements.txt
```

Figure 4.6 File structure as suggested by ChatGPT

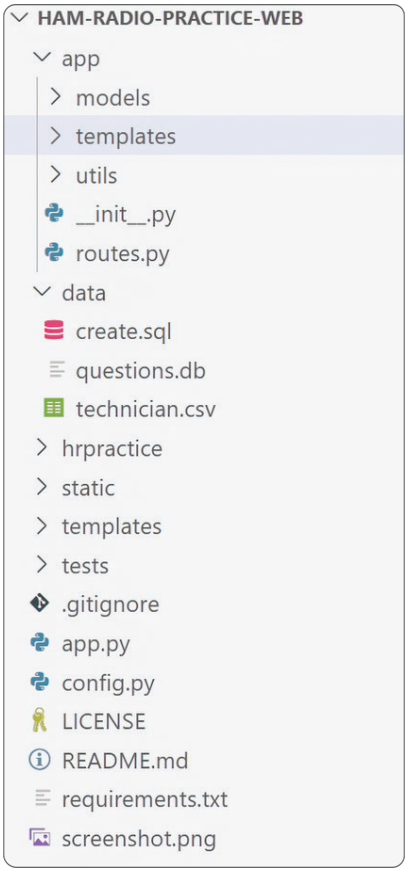

Figure 4.7 Our application in Visual Studio Code

- `app/utils/question_selector.py`—This file contains the logic for selecting the random questions for each test.
- `tests/`—This folder contains all test files, such as tests for your routes, question selection logic, grading logic, and similar.
- `config.py`—This file contains configuration variables for your Flask application (e.g., database URIs, secret keys).
- `run.py`—This file is used to run the application.
- `requirements.txt`—This file lists all Python dependencies that your Flask application needs to run.

This setup offers a good starting point and can be further expanded or refined based on your specific needs. It's a solid approach, and there's every reason to explore it further. Let's follow the outline by creating the folders and then use text files in place of working scripts. Figure 4.7 shows our project.

Our project is laid out and ready for code implementation. Now that we have established a file structure, we need to create a code structure to match it. Generative AI can help us with this as well. You can ask ChatGPT to generate a Python script to create this folder structure:

JM Create a Python script to create this structure. Create the folders and blank files in each folder where they are stored.

It saves you the hassle of making it all yourself.

Using generative AI for application structure

Remember, when using AI for application architecture

- AI tools excel at suggesting standard project structures.
- Request different options to compare approaches.
- Always question AI's suggestions against your specific needs.
- Use AI suggestions as a starting point, not a final decision.
- Ask for reasoning behind structural decisions.

For Flask applications specifically, AI can help organize

- Routes and controllers
- Templates and static assets
- Database models and migrations
- Testing structure

4.5 Stubbing out our application

We're ready to stub out our application. We talked about stubbing earlier in the chapter. Now let's put it into action.

For example, our `DatabaseManager` class could look like this:

```
class DatabaseManager:
    def __init__(self):
        pass
    def connect_to_database(self):
        # TODO: Implement this method to connect to the database.
        pass
    def update_database(self, data):
        # TODO: Update the database with the given data.
        pass
    def validate_and_cleanse_data(self, data):
        # TODO: Validate and cleanse the given data.
        pass
    def fetch_questions(self):
        # TODO: Fetch questions from the database.
        pass
```

These stubs can be organized in your files, and those files are placed in appropriate folders. This way, you can preemptively organize your code. Here is how I've chosen to generate stubs for this application, outlined in figure 4.8. I do this in three steps:

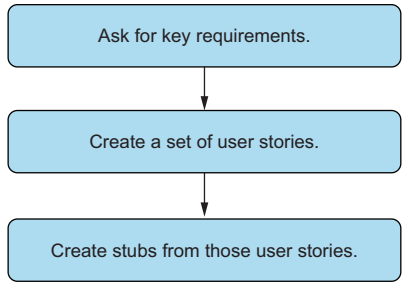

Figure 4.8 The three-step prompt process

1. *Requirements extraction*—Ask AI to identify key requirements from your design document.
2. *User story creation*—Have AI convert those requirements into user stories with roles and objectives.
3. *Stub generation*—Request that the AI create class and method stubs organized by feature.

This methodical approach helps

- Ensure alignment between design and implementation
- Create a logical organization for your code
- Identify missing components early
- Establish clear interfaces between system components
- Produce documentation that maps to your actual code

Although the process involves several steps, it's a technique that's worked well for me. I won't show the output of the first two steps here for brevity and as we've already reviewed the requirements. So, I'll show you the three prompts I used, and we can look at our final output.

JM What are the key requirements we can obtain from our software design document?

JM Rewrite this as a set of user stories in SCRUM. Each headline will serve as a story, and the requirements will become tasks for the appropriate story.

JM Rewrite this as a set of features organized in groups. Create a list of stubs and the classes they would go in to design the application.

I discovered these steps through experimentation and found that if you can distill the requirements and put them in stories, it makes it easier for ChatGPT to create useful stubs. After running these prompts in the same order, we'll come up with a decent start for stubbing. Here is the output from ChatGPT:

Features:

Database Management Feature

- Manage Question Pool Database.

- Update, validate, and cleanse data.

Question Selection Feature

- Randomly select 35 questions for each test.
- Ensure true randomness and no repetition within a single test.

User Interface Feature

- Present questions and answer choices to users.
- Display final score at the end of the test.
- Be user-friendly and accessible.

Grading Feature

- Compare selected answers with correct answers.
- Calculate and display the final score as a percentage.
- Handle unanswered questions.

Enhancement Feature

- Performance analytics.
- Focused study on specific areas or question types.
- Integration with a larger learning platform.

Robustness and Efficiency Feature

- Ensure robustness and efficiency in all components.
- Address accessibility needs.
- Collect and address user feedback.

Stubs and Classes:

1. Class: `DatabaseManager`
   a. Stub: `connect_to_database()`
   b. Stub: `update_database()`
   c. Stub: `validate_and_cleanse_data()`
   d. Stub: `fetch_questions()`
2. Class: `QuestionSelector`
   a. Stub: `randomize_questions()`
   b. Stub: `fetch_single_test()`
3. Class: `UserInterface`
   a. Stub: `display_questions()`
   b. Stub: `display_final_score()`
   c. Stub: `receive_user_input()`
   d. Stub: `validate_user_input()`
4. Class: `Grader`
   a. Stub: `compare_answers()`

   b. Stub: `calculate_final_score()`
   c. Stub: `handle_unanswered_questions()`
5. Class: `EnhancementManager`
   a. Stub: `track_user_performance()`
   b. Stub: `focus_study_area()`
   c. Stub: `integrate_learning_platform()`
6. Class: `ApplicationManager`
   a. Stub: `ensure_robustness()`
   b. Stub: `ensure_efficiency()`
   c. Stub: `address_user_feedback()`
   d. Stub: `ensure_accessibility()`

This setup organizes the features and associated stubs within appropriate classes, laying the foundation for a structured and modular development approach. The classes and stubs can be further refined and detailed during the actual implementation phase.

When looking at the output, we can see things are organized well. But how can we fit these stubs into our application with the structure we asked for earlier? Here's the prompt I used:

JM Consider the file structure you provided for this application. Arrange these stubs into the appropriate files and merge them in with the existing structure. Provide the output.

This prompt takes advantage of our conversational history, which is important because it's treated like a human conversation. This allows ChatGPT to act as if it remembers the file structure it suggested. And it works well. Figure 4.9 shows the new file structure.

This structure is more useful than the previous one. It shows files and folders; some of our stub methods can be included in different files. What we need to do now is build this structure out. It should be noted that this will be a *suggestion* for our application. We are not forced to use this suggestion by any means. Remember, this isn't a source of truth but *a representation of what the model thinks a software file structure looks like.* I don't want to build an application manager or enhancement manager, so I'll take some liberties with this design and remove them. Figure 4.10 shows the final folder structure.

Now that we have our file structure established, I'll fill in the methods in the classes as placeholders. Here's an example of `questions.py`:

```
class Questions:
    def connect_to_database(self):
        pass

    def update_database(self):
        pass
```

```
    def validate_and_cleanse_data(self):
        pass

    def fetch_questions(self):
        pass
```

```
HAMRadioPracticeTest/
│
├── app/
│   ├── __init__.py   # Initialization code for the app.
│   ├── routes.py     # Define routes for handling user requests and responses.
│   ├── templates/
│   │   ├── index.html   # Main page (possibly for starting a test or seeing options).
│   │   ├── test.html    # Page for showing the randomized test questions.
│   │   └── results.html # Page to display the score/results after the test.
│   ├── static/
│   │   ├── css/
│   │   ├── js/
│   │   └── img/
│   ├── models/
│   │   └── questions.py  # Store the structure and logic related to questions and database.
│   │       - connect_to_database()
│   │       - update_database()
│   │       - validate_and_cleanse_data()
│   │       - fetch_questions()
│   ├── utils/
│   │   ├── question_selector.py  # Logic for randomizing and fetching questions for a test.
│   │       - randomize_questions()
│   │       - fetch_single_test()
│   │   ├── grader.py             # Logic for grading the test.
│   │       - compare_answers()
│   │       - calculate_final_score()
│   │       - handle_unanswered_questions()
│   │   ├── enhancement_manager.py # Logic for future enhancements like focused study or inte
│   │       - track_user_performance()
│   │       - focus_study_area()
│   │       - integrate_learning_platform()
│   │   └── application_manager.py # Logic to ensure the app's robustness, efficiency, and ac
│   │       - ensure_robustness()
│   │       - ensure_efficiency()
│   │       - address_user_feedback()
│   │       - ensure_accessibility()
│
├── tests/
│   ├── __init__.py
│   └── test_routes.py  # For testing the routes.
│
├── config.py  # Configuration settings for the app.
├── run.py     # Script to start the application.
└── requirements.txt  # List of required libraries/packages.
```

Figure 4.9 The suggested file and stubbing structure from ChatGPT

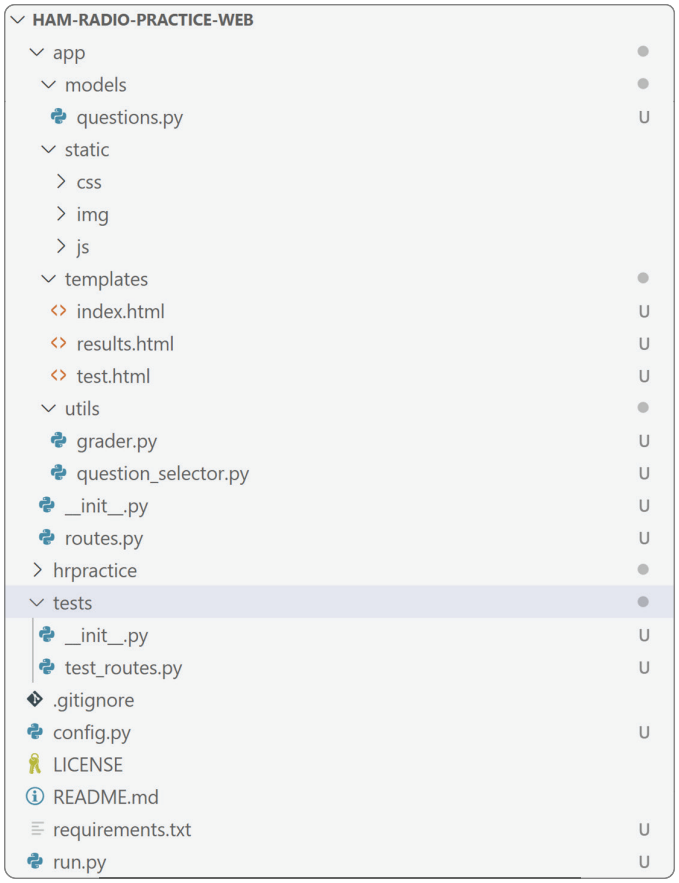

Figure 4.10 Our application file structure

Once the methods are stubbed out, I can define classes and flesh out each method's implementation as I go. This helps with iterative development styles by creating a structure and adding small pieces as we go.

We'll add the empty functions to our other files, too. In Figure 4.9, ChatGPT suggested stub functions for each file, so we'll implement its recommendation. Remember, this isn't set in stone, and we'll change things as we go. After all, we're using generative AI to *assist* us with writing an application, not having it write an application for us.

`models/questions.py`

- `connect_to_database()`
- `update_database()`
- `validate_and_cleanse_data()`
- `fetch_questions()`

utils/question_selector.py

- `randomize_questions()`
- `fetch_single_test()`

utils/grader.py

- `compare_answers()`
- `calculate_final_score()`
- `handle_unanswered_questions()`

We have an initial version of our application created. We have generated a folder structure, inserted some empty files named accordingly, and added stubs for our classes. This will be the foundation or "the bones" of our application. Now, we're ready to initiate the setup process.

Understanding AI limitations in software design

Be aware of these AI limitations when using it for software design:

- AI may suggest popular but inappropriate technologies (like the React example in the chapter).
- Models tend to recommend what they've seen most often, not necessarily what's best.
- Technical suggestions require your critical evaluation.
- AI doesn't understand your specific team's skills or constraints.
- Models can be overconfident in their recommendations.
- Always validate AI suggestions against your actual requirements.
- Remember that *you* are the designer—AI is merely a tool.

4.6 Running our application

We have a file structure and know everything is organized for a Flask application. It's also far from a real application since we can't run it yet. Where can we start? For this problem, I turned to GitHub Copilot chat. I have the project open, and I can ask how to get started.

Here is the prompt I used:

JM This project will be a Flask application. How can I build an invokable Flask application? What must I include in `run.py` so that a basic flask application will run?

I am pretending to know nothing about Flask to see whether it will provide code to get the application running. The results are presented in figure 4.11.

The result looks great. It gives you sample code to put in the application. Then, the code is explained line by line. This is useful if this is a topic or technique you aren't familiar with. As we've discussed many times, it's not advisable to paste in generated

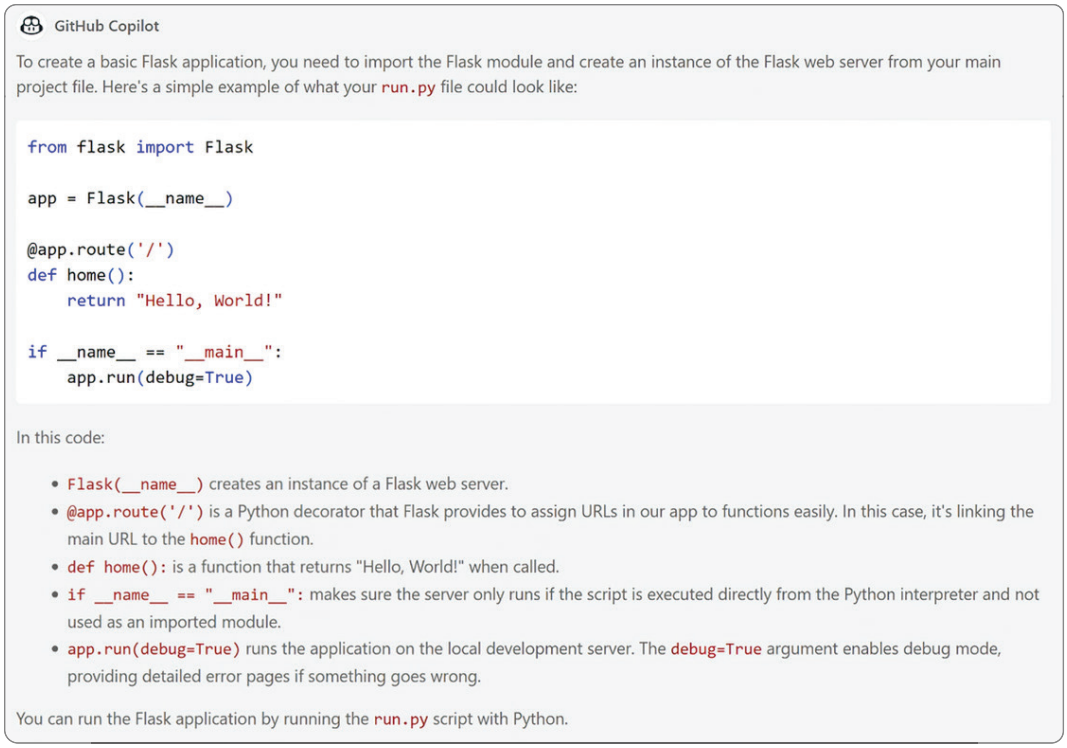

Figure 4.11 A response to our prompt with GitHub Copilot chat

source code you don't understand. It's crucial to understand the code you're working with. I'll paste this into my `run.py` and follow the instructions.

Here's the exact code I added to `run.py`:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Hello, World!"

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

NOTE The instructions do not include installing Flask. You can do this by typing `pip install flask` at your terminal.

I then type `python run.py`, as instructed in the Copilot chat. Figure 4.12 shows the outcome.

```
 * Serving Flask app 'run'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 820-232-364
```

Figure 4.12 A terminal window showing Flask running with no errors

We have a Flask application up and running. Let's go to the URL displayed and check it. We should see a working web page that displays our message, shown in figure 4.13.

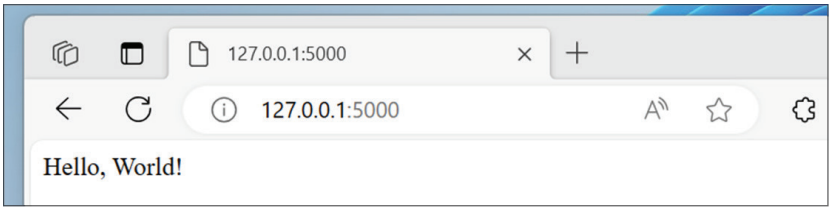

Figure 4.13 A successful "Hello, World!" display from our Flask application

This is exactly what we expect to see from our Flask application. This is our basic "Hello, World!" application. While we're still far from an actual application, this is a great start. We've made a lot of progress in a short amount of time.

AI as an idea generator, not a decision maker

Remember these principles when using AI for software design:

- AI tools are brainstorming partners, not architects.
- The best use is to augment your creativity, not replace it.
- Always apply your domain knowledge and experience.
- Use AI to speed up routine parts of design, freeing time for complex problems.
- Choose what to keep from AI suggestions based on your project's specific needs.
- Combine ideas from multiple AI sessions for optimal results.
- Remember that you make the final decisions—AI is just one input.

In this chapter, we've taken our first practical steps into developing with AI assistance, transforming abstract design concepts into a functional application skeleton. We explored how generative AI tools can accelerate the software development life cycle,

while still preserving developer control and creativity. The journey from requirements to running code demonstrated several key principles:

- *AI as a collaborative partner*—Using multiple AI tools such as ChatGPT and Gemini helped us gather and refine requirements. This approach provided different perspectives, improving our grasp of the problem space. Each tool offered unique insights. AI collaboration shines when seen as a conversation, not just a one-time query.
- *Structure before function*—The AI-suggested file structure and code stubs gave us a strong start. They helped us organize our thoughts before writing any functional code. This stubbing method creates clear interfaces between components. It also prevents the chaos that can happen in quickly developed apps.
- *Human judgment remains essential*—We carefully choose which AI suggestions to use and which to change or ignore. The technical stack recommendations showed that AI often suggests popular technologies such as React.js and Node.js. However, simpler solutions such Flask alone might better suit our needs.
- *From blank page to working application*—One major benefit was overcoming the "blank page syndrome." In a short time, we moved from an idea to a working Flask application. We also established a clear path for adding more features. This can be one of the most valuable ways to use these tools.

As we progress, we'll explore how AI tools can improve various parts of the development process. The skills you've learned in this chapter—creating effective prompts, critically assessing AI outputs, and blending suggestions—set the stage for better AI–human collaboration.

Keep in mind that AI tools are great helpers at many stages of application development. However, you decide how to use their suggestions. Your judgment, creativity, and expertise are key to successful software development. In the next chapter, we'll dig in and start building our application, with our new AI assistants helping us along the way.

Summary

- We can extract requirements from our design by sending the right prompts to ChatGPT.
- ChatGPT can suggest a file structure for our application.
- It is possible to generate stubs and classes with generative AI to structure our application.
- GitHub Copilot will create a starting point for our Flask application.
- You can use multiple AI tools at once for the same process.
- AI tools provide suggestions, but programmers still make key decisions on how to use the output.