

Using Blackbox AI to generate base code

This chapter covers

- Establishing and managing persistent user sessions in Flask-based applications
- Implementing database functionality to track question sets
- Crafting effective prompts for troubleshooting technical errors
- Applying separation of concerns to improve application architecture
- Building a session-based user interface for practice tests

Let's continue our exploration of generative AI tools by building a real web application. Learning to use these tools effectively is an art, and finding the right balance between AI-generated code and human judgment and logic is key to purposeful development.

We will walk through developing our HAM radio practice exam web app. I'll focus on using Blackbox AI to generate code and accelerate development. However, I will

also intervene at key points to refine the architecture, ensure separation of concerns, inject our own code, and override AI suggestions when needed.

The goal is to demonstrate effective cooperation between human expertise and AI. We use the tools to accelerate development through suggestions, while providing context, direction, and corrections. This approach will produce code that surpasses what AI could achieve alone, while also accelerating development compared to human-only workflow. Let's continue our journey!

AI tools as programming partners

When using generative AI for coding

- Bear in mind that each tool has its strengths—Blackbox AI, GitHub Copilot, and Tabnine offer unique features.
- Let AI create scaffolding and boilerplate code to save time.
- Think of AI as your junior pair programmer, not a replacement.
- Be ready to refactor AI-generated code to fit your project's needs.
- Keep a critical eye—these tools are strong, but they miss context that you understand.
- This is a collaborative process: you set the vision, and AI speeds up the work.
- Try different prompting styles to see what fits your workflow best.

Remember: Your skill in recognizing good code architecture is still essential as AI tools change.

5.1 Application development with generative AI tools

We will work on a beta version of our application. In the beginning, we will use Blackbox AI aggressively. Instead of the smart assistant role, the tool will be the lead programmer. Typically, I wouldn't suggest this approach. However, I want to demonstrate the techniques you can use and the full capability of the tools. We'll generate most of the initial code directly from the tool.

If you're an experienced Python developer, you might say, "I would never do it this way." This is perfectly reasonable. In real life, you'll use some suggestions and reject others. You are the final reviewer in your IDE, and that's how it should be. Let's see how far generative AI can take us. After we get things set up, we will do some heavy refactoring to make this work. Overall, it should be a faster pace of development than it would be without the tools.

If you're following along with and coding this project step by step, these tools cost money. I can understand why you wouldn't want to purchase all three. The good news is you don't have to. In my experience, any of the main tools we've talked about (GitHub Copilot, Tabnine, and Blackbox AI) can do this project individually, from start to finish, using the same techniques. You can still follow along if you prefer using a different tool.

5.2 Setting up the development environment

Here is the development environment I'll use to build this application:

- Visual Studio Code, available at <https://code.visualstudio.com/download> (free)
- DB Browser for SQLite, available at <https://sqlitebrowser.org/dl/> (free)
- Python 3.10.12 (The latest version should work fine.)
- Windows 11
- The GitHub Copilot Extension, available at <https://github.com/features/copilot> (Pro Version, \$10 per month)
- The Blackbox.AI Extension, available at <https://mng.bz/DwBn> (\$1.99 per week)
- The Tabnine Pro Extension, available at <https://www.tabnine.com/install/vscode> (\$12 per month)
- WSL running Ubuntu 22.04 LTS

One of the things I love about Python development is its cross-platform capabilities. I'm using Windows 11. If you're following along, you can use Linux or Mac instead, and the instructions are the same. I chose the SQLite browser to work with our database because it works equally well on all three platforms.

The GitHub repository for this code is available at <https://mng.bz/lZ86> if you want to download the code and check it out. Now that we're all set up, let's jump right into developing this application.

5.3 Developing core features

Here, we develop the core of the application. We'll get a rough version of our HAM radio practice test app working by the end of this chapter. I start with the database. A dataset is available for download from the following GitHub repository: <https://mng.bz/Bz80>.

A note about the HAM radio test: Unlike tech certifications, the questions *and answers* to the test are publicly available. So, we will take this database of questions for the technician exam and use it for our application.

5.3.1 Creating the database

In this step, there isn't much for generative AI to do. We will download a CSV from the GitHub repo and place it in a /data folder. Here is the direct link to the file: <https://mng.bz/dWYN>.

You can download the file with your browser or use a tool such as CURL or wget to download it directly. The file is separated by commas, and its format is shown in table 5.1.

Table 5.1 Format of our questions data. It maps directly to a database table with this information.

id	correct	question	a	b	c	d
T1A01	2	Question?	possible answer	possible answer	possible answer	possible answer

We create a new database and a table in that database for these questions. We can start by opening the DB Browser for SQLite. Click on New Database (figure 5.1).

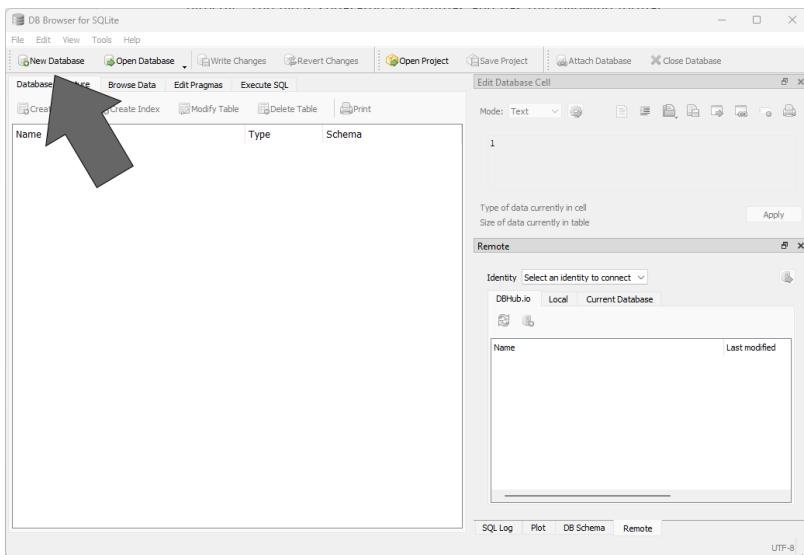


Figure 5.1 The DB Browser main screen. We'll create a new SQLite database here.

In the next screen (figure 5.2), we can specify our file name and click Save.

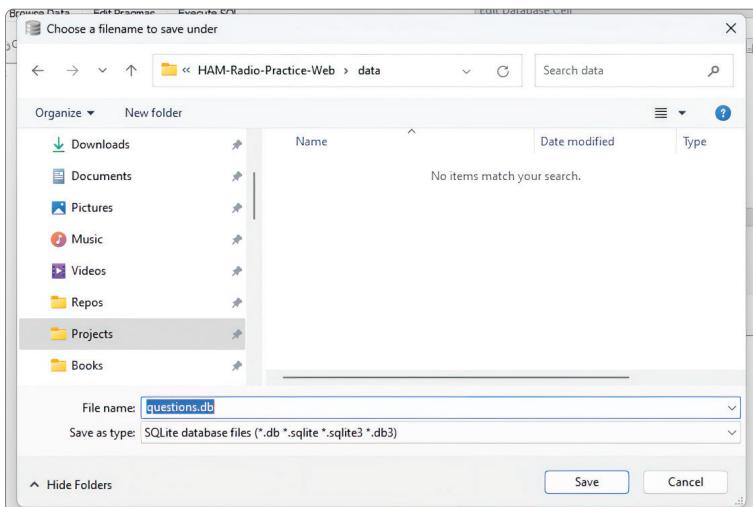


Figure 5.2 Saving a new database as the `questions.db` file

You will see a screen pop up that says, “Edit table definition.” We will skip that, so click Cancel in that window.

Next, go to File -> Import -> Table from the CSV file (figure 5.3).

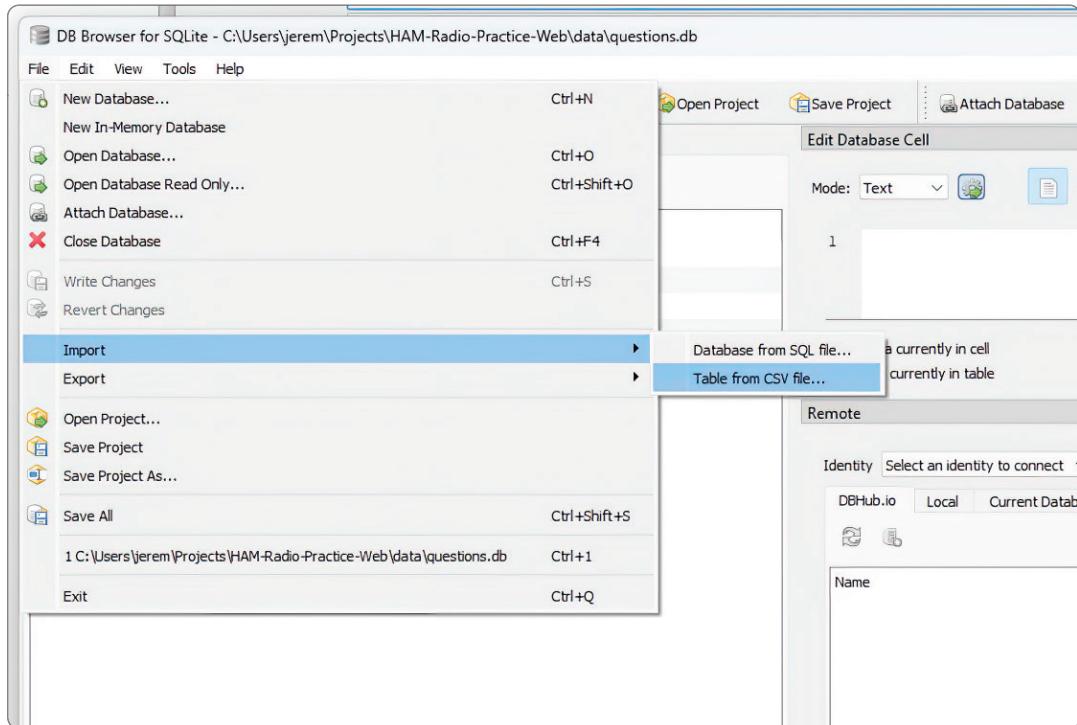


Figure 5.3 Importing a CSV file into the database. We'll create a table from a CSV file.

We select the file we downloaded from the repository, and you will get an Import CSV file window on the screen. Make sure the following options are selected (figure 5.4).

- Change “Table name” to questions.
- Check “Column names” in the first line.
- Enter , as “Field separator.”
- Enter “ as “Quote character.”
- Select UTF-8 for “Encoding.”
- Check “Trim fields.”

This dialog will import the CSV file into our SQLite database. Click the “Write changes” button to ensure the database is created with this data. Next, we want to modify the table. Right click on the questions table and select “Modify table.” Then, in the “Edit

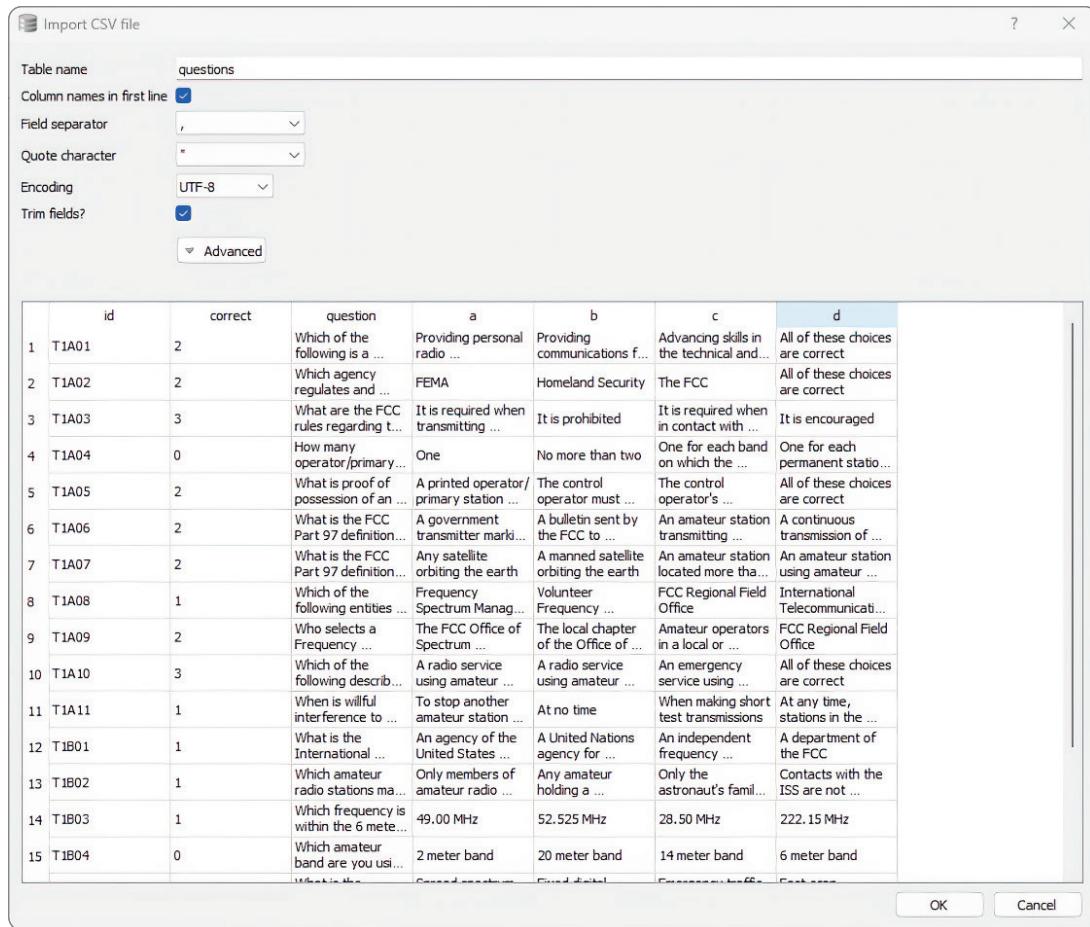


Figure 5.4 This dialog allows you to visually verify the CSV is read properly and the data is formatted the way you expect it.

table” definition screen, make sure PK and U are selected for ID. This selection will ensure that our ID key is treated as a unique and primary key, which will help with indexing and tying this to other tables later (figure 5.5).

Now your database should be up and running, and you can browse the database and see our question data (figure 5.6).

With our database set up, we now need to build the functionality to connect to it. We’re going to do this with one of the generative AI tools in our toolbox.

5.3.2 Connecting to our database

Now that we have a working database preloaded with data, we need to connect to it. We can build something that will open the database and connect to it, then send a

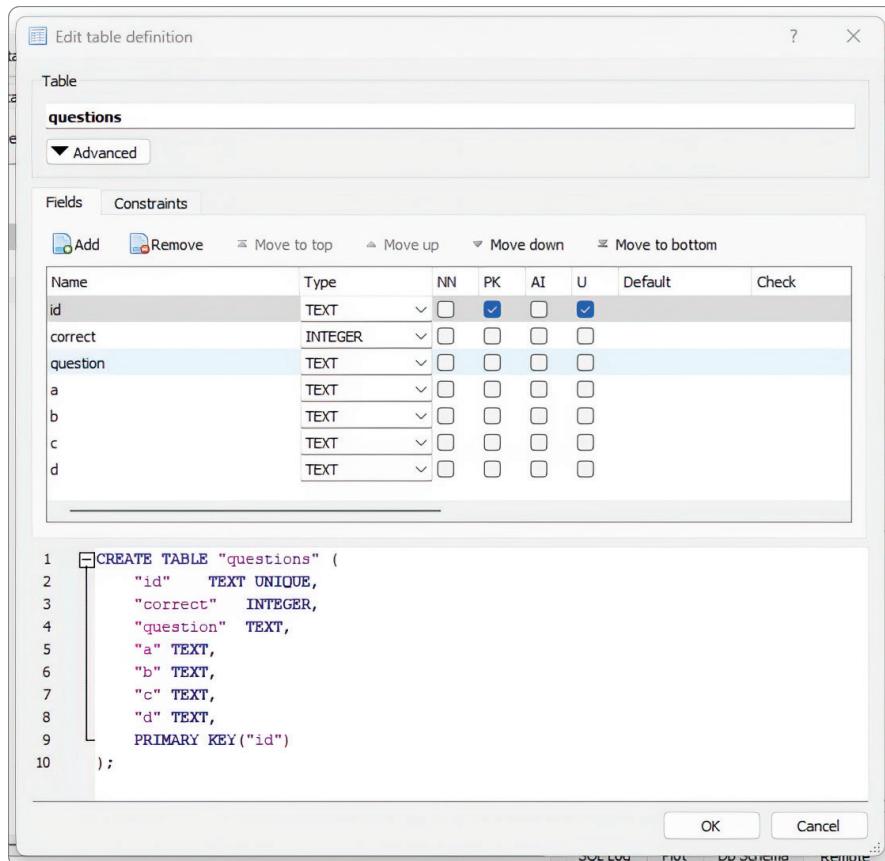


Figure 5.5 In this screen, ensure that the PK and U fields are selected for ID.

	id	correct	question	a
1	T1A01	2	Which of the following is a purpose of the Amate...	Providing personal radio communications for
2	T1A02	2	Which agency regulates and enforces the rules f...	FEMA
3	T1A03	3	What are the FCC rules regarding the use of a ...	It is required when transmitting emergency .
4	T1A04	0	How many operator/primary station license grant...	One

Figure 5.6 In DB Browser, you can visually inspect your tables by clicking Browse Data.

query to it, and finally return data. We’re going to use the Blackbox AI tool to guide us through most of this process.

Let’s start with the file `models/questions` we created in the last chapter. If you remember, it looks like this:

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

    def connect_to_database(self):
        # TODO: Implement database connection logic
        pass

    def update_database(self):
        # TODO: Implement database update logic
        pass

    def validate_and_cleanse_data(self):
        # TODO: Implement data validation and cleansing logic
        pass

    def fetch_questions(self):
        # TODO: Implement logic to fetch questions from the database
        Pass
```

This file is stubbed out and ready for us to write some code. It will look different after we refine the code, but it’s a good start. Let’s begin with the `connect_to_database` method. We will make this method functional:

```
def connect_to_database(self):
    # TODO: Implement database connection logic
    Pass
```

Let’s pretend we’ve never done this before and ask Blackbox to complete the function. Will it work? There are a few approaches we can try. The first approach is one we’ve implemented before. We’ll add a comment and then have Blackbox auto-generate some code based on the comment.

I’ll enter the function and type in the comment:

```
# Connect to a database
```

And this is what Blackbox generates (figure 5.7).

```
def connect_to_database(self):
    # Connect to a database
    # TODO: Implement database connection logic
    < > Accept Accept Word ⌘ + RightArrow ...
    self.db = None # Replace with actual implementation
    pass
```

Figure 5.7 Blackbox showing a vague result from our prompt

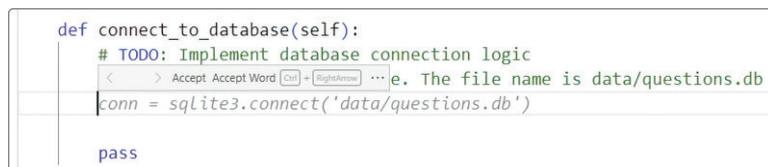
The response isn't very helpful. And if you're this far into the book, you likely already know why. In previous chapters, we learned about context and input. Let's take a look at our code and check what Blackbox sees. By the way, this is pure speculation and an assumption of what the tool looks for. But I do know we're sending the following for context:

- This is written in Python and is part of a class.
- It has four empty functions in the file.
- It seems to have something to do with databases.
- The user wants me to connect it to a database.

We aren't sending Blackbox nearly enough information about what we want. Treat the comment as a prompt. Provide it with clear directions, concise and simple. We can't expect any tool to read our minds (yet). Instead, let's improve our prompt:

JM # connect to a SQLite database. The file name is data/questions.db.

This prompt will tell Blackbox what we want it to do and give specificity. It now knows we have a SQLite database file and want to connect to it. Figure 5.8 shows the outcome.



```
def connect_to_database(self):
    # TODO: Implement database connection logic
    < > Accept Accept Word [Code + RightArrow] ... e. The file name is data/questions.db
    conn = sqlite3.connect('data/questions.db')
    pass
```

Figure 5.8 Code suggestion with more information in the prompt

This result is much better. Blackbox attempts to write code for connecting to an SQLite database. The code looks like it could be correct, so we accept it. Additional code is needed, and we have a couple of options:

- *Go through and fix errors and generate code step by step.*

We can walk through the code and generate lines. I can see errors pop up that give me a clue what to do, as shown in figure 5.9.



```
def connect_to_d "sqlite3" is not defined Pylance(reportUndefinedVariable)
    # TODO: Impl (function) sqlite3: Any
    # connect to View Problem (Alt+F8) Quick Fix... (Ctrl+.)
    self.conn = sqlite3.connect('data/questions.db')

    pass
```

Figure 5.9 Error suggesting we need to install the SQLite3 library

This method works fine but takes a bit more time. It can be slower than coding by hand. But we have another option that will work better—the Blackbox AI chat.

- *Start a chat in the Blackbox AI chat window.*

Since we are generating more than a few lines of code, it may be easier to have a dialog with the tool in the chat window. This way, we can add lots of context to our prompts and get exactly what we want.

You can open the chat window by clicking the Blackbox icon. It will bring up the window and take up around 50% of your IDE space (figure 5.10).

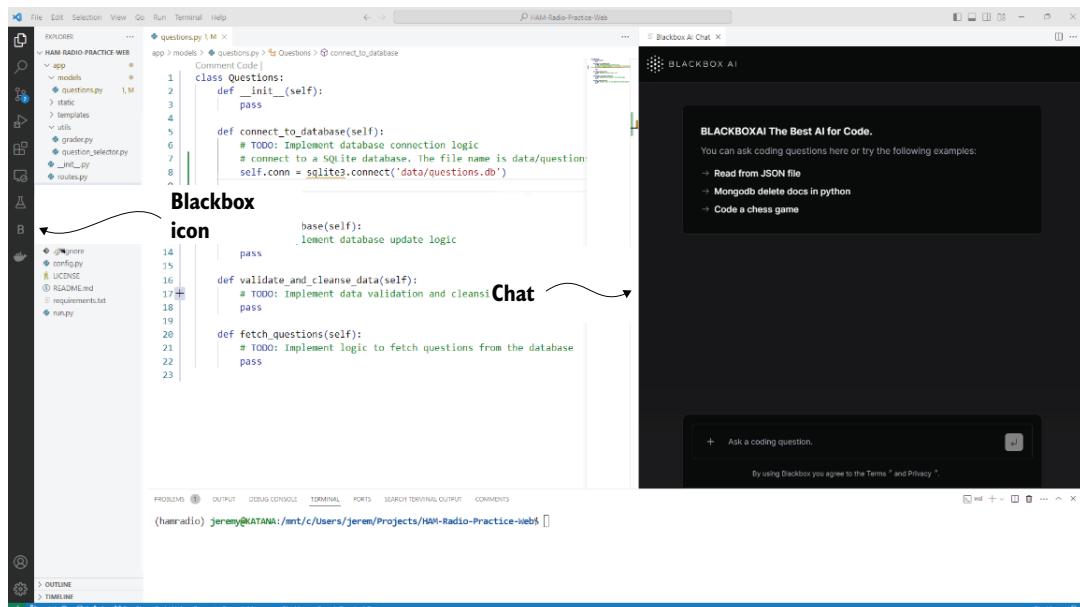


Figure 5.10 The Blackbox AI chat window

This is the approach I like to take when generating larger chunks of code. Later in the chapter, we'll use the comment approach again. For now, let's try to come up with the best prompt to get code to connect to the database. Here is the first one I started with:

JM

I want to connect to a SQLite database in my Flask application. I have a SQLite database located at `data/questions.db`. What is the best way to write this out for `my_connect_to_database(self)` function?

This prompt resulted in a detailed, full explanation of what I needed to do. However, it suggests using SQLAlchemy to connect (figure 5.11).

The screenshot shows the Blackbox AI interface. At the top, there's a logo and the text "BLACKBOX AI". Below that, a message from the AI says: "You can achieve this by using the following code snippet. Here, we're using the Flask-SQLAlchemy extension, which is an easier and more human-friendly way to work with SQLAlchemy." A code editor window titled "python" contains the following code:

```
1 from flask import Flask
2 from flask_sqlalchemy import SQLAlchemy
3
4 app = Flask(__name__)
5 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///data/questions.
6 db = SQLAlchemy(app)
7
8 class User(db.Model):
9     id = db.Column(db.Integer, primary_key=True)
10    username = db.Column(db.String(80), unique=True, nullable=False)
11    email = db.Column(db.String(120), unique=True, nullable=False)
12
13    def __repr__(self):
14        return '<User %r>' % self.username
```

At the bottom of the code editor, there are two buttons: "Continue" and "New Chat".

Figure 5.11 Blackbox AI's first suggestion from our prompt

There's nothing inherently wrong with this suggestion. In fact, it's quite good. And I don't want to use SQLAlchemy in this project. Since this is a dialog with the tool, I can make a simple adjustment and send the following prompt:

JM How can I do this without using SQLAlchemy and connect directly instead?

This will let Blackbox know my intentions. Expressing intent clearly is a good technique for getting better results quickly. Here is how Blackbox responded:

Blackbox response: Sure, you can achieve the same result without using SQLAlchemy by connecting to the SQLite database directly using Python's sqlite3 module. Here's how you can do it:

Python code:

```
import sqlite3

# Establish a connection to the SQLite database
conn = sqlite3.connect('data/questions.db')

# Create a cursor object
cur = conn.cursor()

# Define your query
query = "SELECT * FROM users WHERE id=1"
```

```

# Execute the query
cur.execute(query)

# Fetch the results
results = cur.fetchall()

# Close the connection
conn.close()

# Print the results
for row in results:
    print(row)

```

This code connects to the SQLite database file 'data/questions.db', executes the query to retrieve all rows from the 'users' table where the 'id' is 1, and then fetches and prints the results. Finally, it closes the connection to the database.

You can customize the 'query' variable to fetch data based on your requirements. For example, you can change the condition in the `where` clause, or use different SQL functions.

This output is great. It's exactly what we need to connect to our database. It even has a fetch function there (that we won't use). Let's do one more refinement in this chat and see if we can make it fit into our existing class. Here is the prompt I used:

-  JM Take this code and make it integrate into our questions class. This method will be `connect_to_database` and its only purpose is to create a connection with the database.

With this prompt, we get specific about what we want to do with this code. Blackbox then produced the following response:

-  To integrate this code into our questions class, you can add a method called `connect_to_database` to the class. Here's how you can do it:

Python Code:

```

import sqlite3

class Questions:
    def __init__(self):
        # Connect to the database
        self.connect_to_database()

    def connect_to_database(self):
        # TODO: Implement database connection logic
        # connect to a SQLite database. The file name is data/questions.db
        conn = sqlite3.connect('data/questions.db')

        # Create a cursor object
        cur = conn.cursor()

        # Define your query
        query = "SELECT * FROM users WHERE id=1"

```

```

# Execute the query
cur.execute(query)

# Fetch the results
results = cur.fetchall()

# Close the connection
conn.close()

# Print the results
for row in results:
    print(row)

# Other methods for the class

```

This looks like another great solution that will integrate into our class quite nicely. So, let's add this to our file, and we can focus on testing it next.

5.3.3 Calling our database from the frontend

We can assume our database function works, but we'll need to test it out. We can ask Blackbox how to do this. There is one way to ask: by clicking on the plus sign near a line of code, you can open a chat about it, or you can use a chat window (figure 5.12).

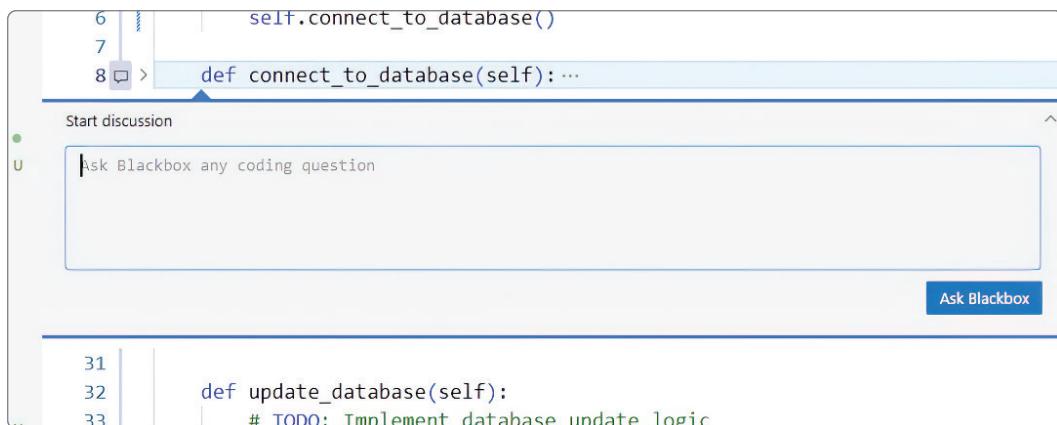


Figure 5.12 You can ask questions directly from the editor window by clicking a plus sign next to a line of code.

I then type in the following prompt:

- JM How can I extract the data from this function and display it on the html index page in Flask?

What follows is the answer I received. It gives us clear steps to run this query and display the results on our front page. Here is the full Blackbox AI output:



To extract the data from the function and display it on the HTML index page in Flask, you can follow these steps:

Create a route in your Flask application for the HTML index page.

In the view function for the HTML index page, import the function that contains the data you want to extract.

Call the function and store the returned data in a variable.

In the HTML template for the index page, use Jinja2 templating to access the data in the variable and display it on the page.

Here is an example of how you could implement these steps:

```
# Import the function that contains the data you want to extract
from my_module import get_data

# Create a route for the HTML index page
@app.route('/')
def index():
    # Call the function and store the returned data in a variable
    data = get_data()

    # Return the HTML template for the index page
    return render_template('index.html', data=data)
<!-- HTML template for the index page -->
<!DOCTYPE html>
<html>
<head>
    <title>Index Page</title>
</head>
<body>
    <h1>Data</h1>
    <ul>
        {% for item in data %}
            <li>{{ item }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

We will accept this result and modify our application. One thing to note. In our `models/questions.py`, we have a query that won't work:

```
# Define your query
query = "SELECT * FROM users WHERE id=1"
```

This query is set to look at the `users` table, which doesn't exist. Our table is named `questions`. Let's change this query to our table name. We know that `SELECT *` is bad. It causes performance problems (e.g., selecting too many fields) and can expose sensitive information. But for now, we'll use `SELECT *` to query our table and refine it later:

```
# Define your query
query = "SELECT * FROM questions"
```

We also need to change the return type of this function:

```
# Print the results
for row in results:
    print(row)
```

As it sits, we are printing the query results to our console. This won't mean much to Flask. Delete the previous lines above and replace them with

```
return results
```

This way, it will pull all the question records from our table when executed and send them to Flask. We can refine the query later.

MODIFYING THE INDEX PAGE

In our query, we got a recommendation for importing our data, but it's a little vague:

```
from my_module import get_data
```

It shows example code rather than the specific names we're working with. It happens sometimes with this tool. I have seen it scan the context of the application and generate specific code, and I have seen it generate code that looks like boilerplate or example code. In this case, we can clarify the code with the name of our class and file. We add this to the top of `run.py`:

```
from app.models.questions import Questions
```

This change will bring in the data class we've just created.

Now, we will add the following code to `app.py` as suggested by Blackbox:

```
@app.route('/')
def index():
    # Call the function and store the returned data in a variable
    data = get_data()

    # Return the HTML template for the index page
    return render_template('index.html', data=data)
```

This code will call a function into the `data` variable and then display it within a template. We know the `data = get_data()` line of code will not work. Because it doesn't match anything we have in that class, I will change it to what it should be. I'll create an instance of our `Questions` class and call the `connect_to_database` method:

```
my_questions = Questions()
data = my_questions.connect_to_database()
```

Next, I need to create our template. We were given the following HTML code, and I'm going to place it in `templates/index.html`:

```
<!-- HTML template for the index page -->
<!DOCTYPE html>
<html>
<head>
    <title>Index Page</title>
</head>
<body>
    <h1>Data</h1>
    <ul>
        { % for item in data %}
            <li>{{ item }}</li>
        { % endfor %}
    </ul>
</body>
</html>
```

I know this project isn't ready to run yet. Figure 5.13 shows an error for the Flask template generation.

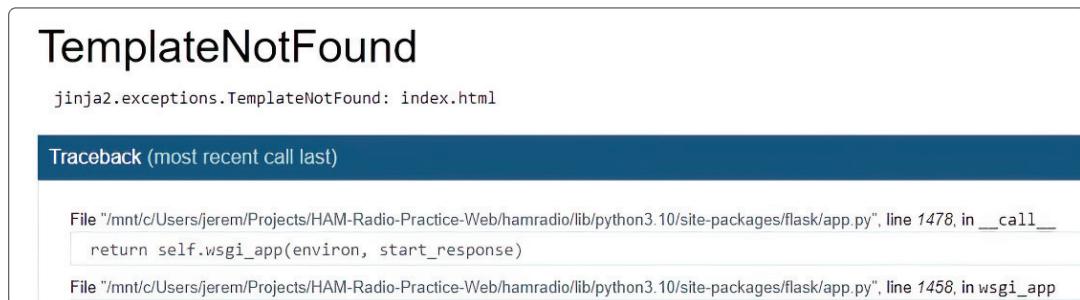


Figure 5.13 We're getting an error with the `render_template` function not being found.

Let's ask Blackbox AI chat what to do:

JM How can I use the `render_template` function in this file so it resolves?

This response I received suggested adding the following to our import statement:

from flask import Flask, render_template

I will add this to our file and see if the application runs. There aren't any errors in our IDE, so let's try it out. I now execute `run.py` and see the error shown in figure 5.14.

What's happening here? This error goes back to the last chapter when we laid out our file structure. According to the Flask documentation, it looks for templates in the `/templates` folder. Yet our templates are in `app/templates` because that's what ChatGPT suggested previously (figure 5.15).

The screenshot shows a code editor window with a Python file open. The code defines a function `index()` that calls `render_template('index.html', data=data)`. A Pylance error message is displayed in a tooltip: `"render_template" is not defined PyLance(reportUndefinedVariable)`. The code editor interface includes line numbers (7-16), a comment code button, and a status bar at the bottom.

```
Comment Code
7  @app.route('/')
8  def index():
9      # Call the function and store the returned data in a variable
10     my_ques
11     data =
12         "render_template" is not defined PyLance(reportUndefinedVariable)
13     (function) render_template: Any
14     # Return View Problem (Alt+F8) No quick fixes available
15     return render_template('index.html', data=data)
16
```

Figure 5.14 Jinja is Flask's templating system. It's showing an error with finding our template.

The screenshot shows a file browser interface with a tree view of a project structure. The root directory is `HAMRadioPracticeTest/`. It contains an `app` folder which further contains `__init__.py`, `routes.py`, `templates` (containing `index.html`, `test.html`, and `results.html`), `static` (containing `css`, `js`, and `img`), `models` (containing `questions.py`), and `utils` (containing `question_selector.py`). There is also a `tests` folder with `__init__.py` and `test_routes.py`. Additionally, there are `config.py`, `run.py`, and `requirements.txt` files at the root level. The interface includes a toolbar with `Copy` and `Edit` buttons.

```
arduinobytes
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 5.15 The file structure ChatGPT suggested for us earlier. It is incorrect according to the common Flask file structure.

ChatGPT is likely displaying best practices or common application layouts. However, Flask applications are structured differently. We could reconfigure Flask to look for templates in the /app/templates folder; however, there's no clear benefit to breaking the Flask convention. I will move the templates and the static folder to our application root instead of storing them in the app folder (figure 5.16).

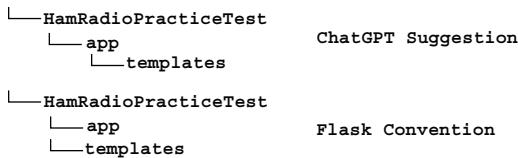


Figure 5.16 We're moving the templates folder to the application root because Flask expects the templates folder at the root by default.

Figure 5.17 shows the new file structure.



Figure 5.17 Our new file structure based on Flask documentation recommendations

With our new file structure in place, I'll re-run the application. Figure 5.18 shows what I see at this stage.

This is the output I'm expecting. The page shows us a few things:

- We can connect to the database.
- We can run a query.
- We can display that data on a page.

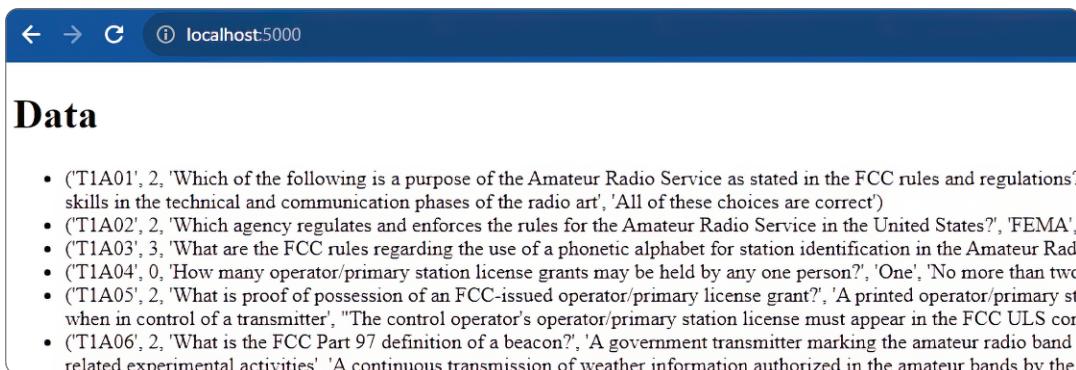


Figure 5.18 The index page showing the results from `connect_to_database()` function, rendered on the screen

This is awesome and a big step forward. We built this to ensure we can connect to a database and display data. But now we need to turn it into something useful for the final product. We will refactor the `connect_to_database()` function to do only one thing—connect to the database.

There's a principle in software development that comes from the original developers of the Unix operating system: *“Do one thing and do it well.”* When a method or software module tries to do too many different things, its complexity increases as it becomes difficult to understand, test, and most importantly maintain. By making your components as simple and lean as possible, you avoid future problems. Code should be easy to understand and work with—especially for future readers, including yourself. Moreover, it should be easy to test.

We'll take that approach with this method. It will connect to the database and pass the cursor back. Nothing more, nothing less. That way, we can quickly connect to the database, perform some actions, and close it.

5.3.4 Refactoring our Questions class

Now, we're going to step back into the hands-on experiences that shaped our understanding. You should never blindly have these tools write an entire application for you. Experience and knowledge still matter. They matter even more in the world of generative AI. You must know what you're building. Here's an example of how we'll tweak things beyond what's generated.

Right now, our `Questions` class has four methods:

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

This is what ChatGPT determined would be best for our class, but it doesn't fit my vision for how it should be built. For one, we aren't updating this database at this time. And validating and cleansing data are important, but we aren't doing that in this phase of the application. We're going for an minimum viable product (MVP) model, meaning we're looking for basic functionality.

Balancing MVP development with best practices

When accelerating development with AI tools

- *Start with core functionality*—Focus on must-have features first.
- *Don't skip database design*—Even for MVPs, proper schema design pays dividends.
- *Defer optimization*—Aim for working code first and performance improvements later.
- *Simplify where possible*—Resist feature creep and overengineering.
- *Maintain separation of concerns*—Even fast development benefits from clean architecture.
- *Keep security in mind*—Some best practices shouldn't be compromised even for speed.
- *Document as you go*—Leave breadcrumbs for your future self in comments.
- *Test critical paths*—Ensure core functionality works as expected.

The sweet spot is creating maintainable code quickly without introducing technical debt that will slow future development.

In the spirit of building a good MVP, here are the methods I'm thinking of:

- `connect_to_database()`
- `close_database()`
- `fetch_questions()`

With these methods fleshed out, we can get the functionality we need to deliver questions to the student.

Here is the `connect_to_database()` function I'll build:

```
def connect_to_database(self):  
    # Connect to a SQLite database. The file name is data/questions.db  
    conn = sqlite3.connect('data/questions.db')  
    # Create a cursor object  
    cur = conn.cursor()  
    # Return the cursor  
    return cur
```

This is a bare minimum connection to our database. We know that we need some correction and error handling here. It's likely you can code this up in your sleep, but let's ask Blackbox for help. For this step, I use Blackbox differently and select the code I'm

asking about. This gives the tool context to work with and avoids generic answers. The selected code becomes part of the prompt.

I'll select the code and ask in Blackbox chat:

JM What error correction should be placed in this function?

I will make sure to check Include Code in Active File, as shown in figure 5.19

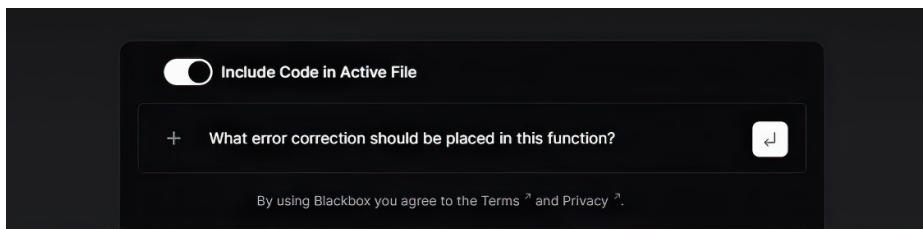


Figure 5.19 Asking a question in Blackbox chat and making sure it's looking at our code

Here is the response from Blackbox. First, it included a suggested code snippet of the entire file:

import sqlite3

```
class DatabaseConnection:

    def __init__(self):
        self.conn = None
        self.cur = None

    def connect_to_database(self):
        # connect to a SQLite database. The file name is data/questions.db
        try:
            self.conn = sqlite3.connect('data/questions.db')
            # Create a cursor object
            self.cur = self.conn.cursor()
        except sqlite3.Error as e:
            print(f>An error '{e.args[0]}' occurred while connecting
to the database.")
        finally:
            if self.conn:
                return self.cur
            else:
                return None

    def close_connection(self):
        if self.conn:
            self.conn.close()
            self.conn = None
            self.cur = None
```

This is exactly what I had in mind when I asked the question. Then, Blackbox follows up with an explanation of the code generated:

 The above code implements the `DatabaseConnection` class with the `connect_to_database` and `close_connection` methods. It includes error handling, and the cursor is stored as an instance variable for later use.

When using the `connect_to_database` method, if there is an error while connecting to the database, it will be printed out. Additionally, if the connection was successful, the cursor will be returned.

When using the `close_connection` method, it will close the connection to the database if it exists.

This is great, and it's what I asked for. You may have noticed something in the generated code, though. In `Questions.py`, the class is named `Questions`:, and in the generated code, we now have class `DatabaseConnection`:. Why?

Looking at this change prompts a great question. Should we be connecting to the database in the `Questions` class at all?

Crafting effective AI prompts for code generation

To get better code from AI tools:

- Be specific about language, frameworks, and versions.
- Include context about your existing code structure and patterns.
- Specify exactly what the function should do and return.
- Mention error handling expectations.
- Include file paths and database details when relevant.
- Break complex requests into smaller, focused prompts.
- Use code comments as inline prompts when working in your IDE.
- For more complex tasks, use the chat interface to maintain conversation context.

Let's compare the flowing prompts:

Weak—"Connect to a database."

Better—"Connect to a SQLite database. The file name is `data/questions.db`."

Best—"Create a method that connects to a SQLite database at `data/questions.db`, handles connection errors, and returns a cursor object."

HUMAN INTERVENTION IS REQUIRED

It's once again time to override the suggestions given by generative AI. ChatGPT and Blackbox did as we asked, but the suggestions still lack the context and ability to read my mind thoroughly. I need to make some design decisions of my own here.

We can reuse this code because there's no reason to put the database open and close functions in every class we create. In fact, our `Questions` class should have *no database connections in it all*.

This follows the basic computer science principle called “separation of concerns.” According to this principle, functions or units of software should be focused on their given task and not overlap or attempt to do too many things. It’s very similar to “Do one thing and do it well.” The `Questions` class should focus on delivering questions, and we should have a separate class to handle database connections.

When to override AI suggestions

Trust your judgment over AI when

- The generated code doesn’t follow separation of concerns principles.
- AI suggests a complex solution when a simpler one would suffice.
- The suggested architecture doesn’t align with your project’s patterns.
- Performance considerations aren’t being addressed (e.g., `SELECT *`).
- The best security practices are being ignored.
- The AI is making assumptions about your project that aren’t accurate.
- The code works but isn’t maintainable long-term.
- You see library or dependency choices that don’t match your stack.

Remember: AI tools can create correct code, but they don’t understand your project’s needs, limits, and future goals. Your ability to detect design flaws early on, before they lead to implementation problems, is your greatest asset.

I’ll create a new file with the code that Blackbox generated for us. Now we’re using a separate class to connect to the database. When we mix the database connection functionality into each of our classes, we risk repeating code. More importantly, we’ll create more code that we must update if we decide to use another database engine, for instance. It’s best to abstract this into its own class for simplicity and avoid these problems.

The database connection pattern

A robust database connection pattern should

- *Isolate connection logic*—Keep database-specific code in one place.
- *Manage resources properly*—Ensure connections are always closed.
- *Support transactions*—Allow for atomic operations.
- *Handle errors gracefully*—Provide meaningful error messages.
- *Be configurable*—Make database paths and credentials configurable.
- *Be thread-safe*—Consider connection pooling for web applications.
- *Follow the single responsibility principle*—Database connection code should do one thing well.

We’ll use this for every database connection we make. Let’s create a new file named `DatabaseConnection.py` in our `Models` directory, and we’ll use it to handle our database connections:

```

import sqlite3

class DatabaseConnection:
    def __init__(self, db_path):
        self.db_path = db_path

    def __enter__(self):
        self.connection = sqlite3.connect(self.db_path)
        return self.connection.cursor()

    def __exit__(self):
        self.connection.commit()
        self.connection.close()

```

This will be our way of connecting to the database, and we can reuse this code in other parts of our application. Let's break it down to understand it completely.

This code defines a context manager for managing database connections using the SQLite3 module in Python:

```
import sqlite3
```

SQLite3 is a database engine that is part of the standard Python library. The following line imports the SQLite3 module:

```
class DatabaseConnection:
```

This line defines a new class named DatabaseConnection:

```
def __init__(self, db_path):
    self.db_path = db_path
```

The following method is the class constructor. It takes a single argument db_path, which is the path to the SQLite database file:

```
def __enter__(self):
    self.connection = sqlite3.connect(self.db_path)
    return self.connection.cursor()
```

The next method is a special method in Python, called when the with statement is used. The with statement in Python is employed to wrap the execution of a block of code that uses methods requiring setup and cleanup steps. The with statement allows you to perform a task and have setup and cleanup performed automatically. That way, you don't have to manually allocate and deallocate resources. We'll design this class to use the methodology. When this class is used as a context manager, __enter__ will be called first if we use the with statement.

In this method, a connection to the SQLite database is established using the sqlite3.connect() function. The connection object is stored in the self.connection attribute. Then, a cursor object is created by calling the cursor() method on the

connection object. This cursor object is returned by this method, allowing it to be used inside the `with` block:

```
def __exit__(self):
    self.connection.commit()
    self.connection.close()
```

This is another special Python method triggered by the `with` statement. When the class is used as a context manager, this method is called last.

The changes made to the database during the `with` block are committed by calling the `commit()` method on the connection object. Then, the connection to the database is closed by calling the `close()` method on the connection object.

To use this class as a context manager, you can do the following:

```
with DatabaseConnection('path/to/database.db') as cursor:
    cursor.execute('SELECT * FROM some_table')
    results = cursor.fetchall()
```

The `DatabaseConnection` class is used as a context manager in a `with` statement. When the `with` block is exited, the `__exit__()` method of the `DatabaseConnection` class will be called automatically, ensuring that the database changes are committed and the connection is closed properly.

Here is how I refactor the `questions.py` file to use the `database_connection.py` file:

```
class Questions:
    def __init__(self, cursor):
        self.cursor = cursor

    def fetch_data(self):
        self.cursor.execute("SELECT * FROM questions") # Adjust SQL query as needed
        return self.cursor.fetchall()
```

We'll return to `fetch_questions` soon, but now our `Questions` class can utilize a separate database connection and focus on doing its primary job, which is delivering questions.

The art of refactoring AI-generated code

Common refactoring needs for AI-generated code are

- Separating concerns (database access, business logic, presentation)
- Removing hardcoded values and adding configuration
- Improving error handling and edge cases
- Making code more testable through dependency injection
- Ensuring code follows project-specific patterns and standards
- Removing unused or redundant code sections
- Improving variable and function naming for clarity

(continued)

Remember that AI tools often generate happy-path code, which works in ideal conditions but may need hardening for production. The most valuable refactoring typically focuses on

- Exception handling
- Resource management (connections, file handles)
- Configurability
- Testability

5.3.5 *Modifying our entry point (App.py)*

Now that we've changed how our questions are generated and separated, the Questions class from the database connection, our entry point needs to change as well. We need to add some imports. We will import Flask, as well as the files we've created so far. They're named explicitly so app.py knows where to find them:

```
from flask import Flask, render_template
from app.models.database_connection import DatabaseConnection
from app.models.questions import Questions
```

Then, we need to change our index() function to point to our database and then call the database connection class into our cursor:

```
def index():
    db_path = 'data/questions.db'
    # Call the function and store the returned data in a variable
    with DatabaseConnection(db_path) as cursor:
```

Next, we pass the cursor into our questions class and use questions.fetch_data to populate the data variable we use for displaying our data. This is the same data variable that our page rendering will use:

```
questions = Questions(cursor)
data = questions.fetch_data()
```

Here is what our modified App.py looks like in its final form:

```
from flask import Flask, render_template
from app.models.database_connection import DatabaseConnection
from app.models.questions import Questions

app = Flask(__name__)

@app.route('/')
def index():
    db_path = 'data/questions.db'
```

```
# Call the function and store the returned data in a variable
with DatabaseConnection(db_path) as cursor:
    questions = Questions(cursor)
    data = questions.fetch_data()
# Return the HTML template for the index page
return render_template('index.html', data=data)

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

Let's break this down and explain what's happening. This file sets up a single route to serve the root path ('/') of the application. First, we need to import the necessary modules and initialize the Flask application:

```
from flask import Flask, render_template
from app.models.database_connection import DatabaseConnection
from app.models.questions import Questions

app = Flask(__name__)
```

Then we need to set up a route and a handler for the root path. The `@app.route()` is the route the app takes when the root path is requested. Then the `index()` function is called as a route handler for the path:

```
@app.route('/')
def index():
```

Next, we set the path to the SQLite database file (`db_path`) and create a connection to the database using the `DatabaseConnection` class we just created. The connection is managed by a context manager (`with` statement), which ensures that the connection is properly closed after the data has been fetched:

```
db_path = 'data/questions.db'
# Call the function and store the returned data in a variable
with DatabaseConnection(db_path) as cursor:
```

Then, we create an instance of the `questions` class and call its `fetch_data` method to retrieve data from the database. This data is stored in the `data` variable:

```
questions = Questions(cursor)
data = questions.fetch_data()
```

Finally, we render the '`index.html`' template using the `render_template` function and pass the fetched data to the template:

```
# Return the HTML template for the index page
return render_template('index.html', data=data)
```

In this block, we run the Flask application if the script is being executed directly (not imported as a module). The `debug=True` argument tells Flask to enable debugging features, such as displaying detailed error messages in the browser:

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

This is our new entry point, and when we run this application, this is what we'll see in the browser (figure 5.20).

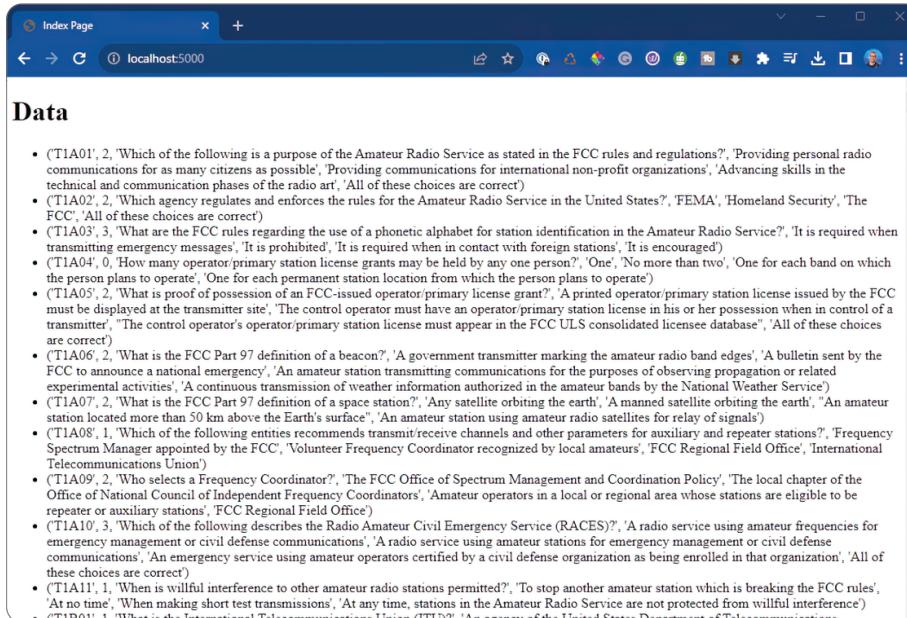


Figure 5.20 The web page displayed at the root of our application. It shows a raw printout of the data returned from our database.

This is a working page, a dump of our questions straight from the database. To the user, this is exactly what we had before, but under the hood, we have more reusable and extensible code that's easier to test. These changes will help us move forward.

Now we'll need to do some refinement to get it closer to our objectives for the application. We'll need to show one question at a time randomly for the student to see and provide an answer for. Let's build out some of that functionality.

5.3.6 Pulling a set of questions

Currently, our application is only showing a dump of questions. We need to extract questions one by one to simulate the test-taking experience. The HAM radio technician

test has 35 questions, so we need to extract 35 questions at random for this test. Then, we'll have our students go through these questions to simulate a test. Let's focus on the function of selecting the questions. We'll start by making it simple.

Our requirements are as follows:

- Pick 35 questions from the database.
- Questions must be unique.
- Attempt to randomize the selection.

We'll attempt to use Blackbox to help us out with this task. The first thing we'll do is open `questions.py` and create a new function:

```
def get_question_set(self):
```

Next, we will highlight this line of code and click `BlackboxAI Code Chat`, as shown in figure 5.21.

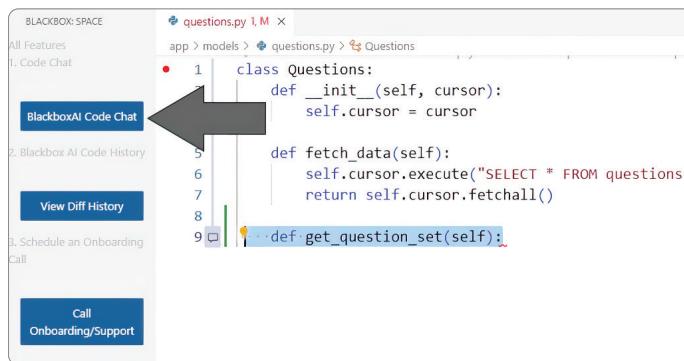


Figure 5.21 The Blackbox AI code chat button

We also make sure the `Include Code in Active View` button is selected and enter our prompt (figure 5.22).

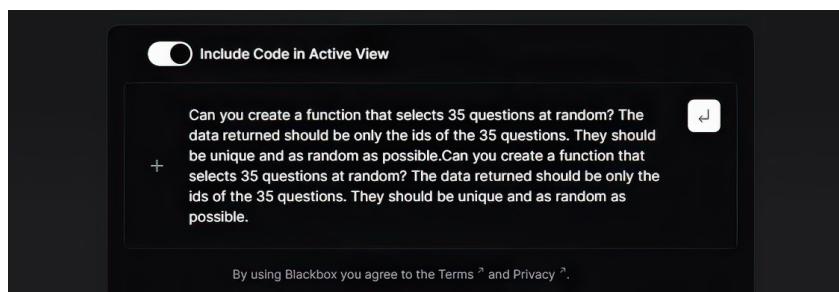


Figure 5.22 The prompt we'll send to Blackbox AI. It is enclosed in the chat window.

Our next prompt is

- JM Can you create a function that selects 35 questions at random? The data returned should be only the ids of the 35 questions. They should be unique and as random as possible.

And here is what Blackbox returns:

-  Sure! Here's an alternative approach to solving this problem using a different data structure called a set. The set data structure allows us to efficiently perform set operations such as unions, intersections, and differences. It also guarantees uniqueness, which is a desirable property for our problem.

Here's a function that achieves the desired result:

```
import random

def get_question_set(self):
    all_questions = self.get_all_questions()
    if len(all_questions) < 35:
        return "There are not enough questions to generate a set."
    question_set = set()
    while len(question_set) < 35:
        question_id = random.choice(list(all_questions))
        question_set.add(question_id)
    return list(question_set)
```

This function works by iteratively adding random question ids to the `question_set` until it contains 35 unique ids. It first checks if there are enough questions available to generate a set of 35 unique ids. If there are not enough questions, it returns an error message. Otherwise, it iterates until the `question_set` contains 35 unique ids.

The time complexity of this function is $O(n)$, where n is the number of questions. The space complexity is also $O(n)$, where n is the number of unique ids in the set.

Note: The function assumes that the `get_all_questions` method returns a set of unique question ids. If the `get_all_questions` method returns a list or any other data structure, you will need to convert it to a set before using it in the `get_question_set` function.

I'll add in the suggested code to our `get_question_set` function. This requires adding in a new function called `get_all_questions()`. However, our `fetch_data` function already provides this service:

```
def get_all_questions(self):
    self.cursor.execute("SELECT * FROM questions")
    return self.cursor.fetchall()
```

This should provide us with what we need. Let's modify our `run.py` file to use our new method. Ideally we can modify this and it should provide us with a set of ids for the "random" questions. Change:

```
data = questions.fetch_data()
```

So, we'll rename it to

```
data = questions.get_question_set()
```

This adjustment should update our application to get the IDs of the 35 questions we need. Let's load it in our browser (figure 5.23).

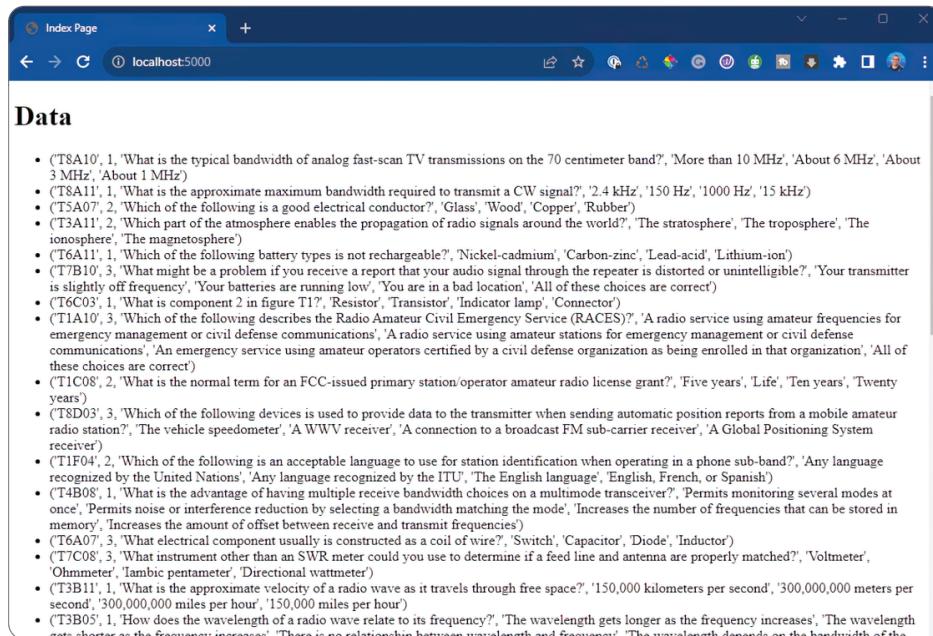


Figure 5.23 Our results from creating a function to randomly grab 35 questions

After loading this up, we can see that it is pulling 35 questions at random. However, we are seeing the full record here, and we only need the ID of each question. The idea here is we can store a set of IDs in a new table and then retrieve the question data from that ID as the student is taking the test. Let's look closer at our `questions` class. I can see that our function `get_all_questions()` is what is pulling the full record:

```
def get_all_questions(self):
    self.cursor.execute("SELECT * FROM questions")
    return self.cursor.fetchall()
```

So, we can change that to only pull IDs. Then we can get a list of IDs and use them as a reference set. We will change this line to

```
self.cursor.execute("SELECT id FROM questions")
```

I like this better because it serves our purpose, but also we never want to `SELECT *` from a table anyway. It can lead to a host of problems and security vulnerabilities.

Effective SQL practices with generative AI

When generating SQL code with AI tools

- *Always review queries.* AI-generated SQL may work but be suboptimal.
- *Avoid `SELECT *`.* Specify exact columns needed to improve performance.
- *Check where clauses.* Ensure proper filtering to avoid table scans.
- *Look for missing indexes.* AI might not know your data volumes or access patterns.
- *Be wary of string concatenation.* Ensure generated code uses parameterized queries.
- *Consider transactions.* Check if operations need to be atomic.
- *Verify error handling.* How does the code handle database exceptions?

SQL is a particular area where AI tools show their limitations. They often suggest functional but inefficient queries that don't account for your specific data patterns and volumes.

This function now selects all IDs from the database, and our `get_question_set()` function will only pull 35 random IDs. Figure 5.24 shows the application's response following a page refresh.

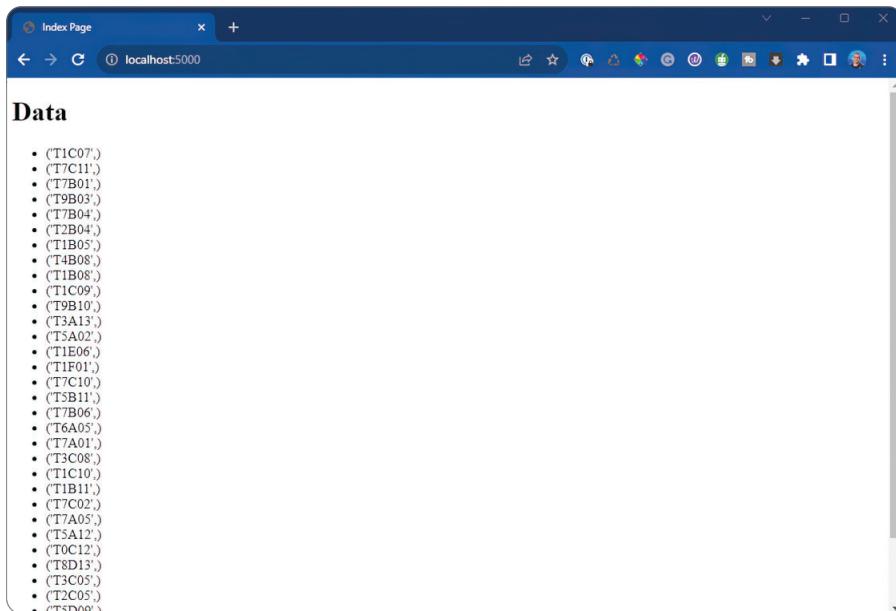


Figure 5.24 Our index page now only shows IDs as output from our query.

This is a step in the right direction. And it completes the functionality I was looking for:

- Pick 35 questions from the database.
- Questions must be unique.
- Attempt to randomize the selection.

Now I just need to create a table that will store these IDs and some session data for our students.

5.3.7 **Creating a test session in the database**

Now that we can pull 35 random questions from the database, we need to establish a test session. Here is what I have in mind. I'd like a table that can store a state for the test session as it's running. This session will run as the students take the test. Here are the high-level requirements:

- Create a session.
- Keep track of the 35 questions.
- Keep track of the correct answer for each.
- Track the answer the student has entered.
- Track the number of correct answers.

We need to create three tables:

- *Sessions*—This table will be instances of tests that a user creates. As they create a test session, it will keep track of the answers answered correctly and incorrectly.
- *Question sets*—For the purposes of a practice test, we need a table of questions grouped together from the main question pool. This will be 35 randomly selected questions.
- *Questions*—This is the table we've already created, the one that contains all the questions from the HAM radio test.

Figure 5.25 shows the database design I'm considering using.

I need to create a `sessions` table that looks like this:

```
"session" (
    "session_id"      INTEGER,
    "questions_correct"   INTEGER,
    "questions_incorrect"  INTEGER,
)
```

We'll open DB Browser for SQLite and click Create Table (figure 5.26).

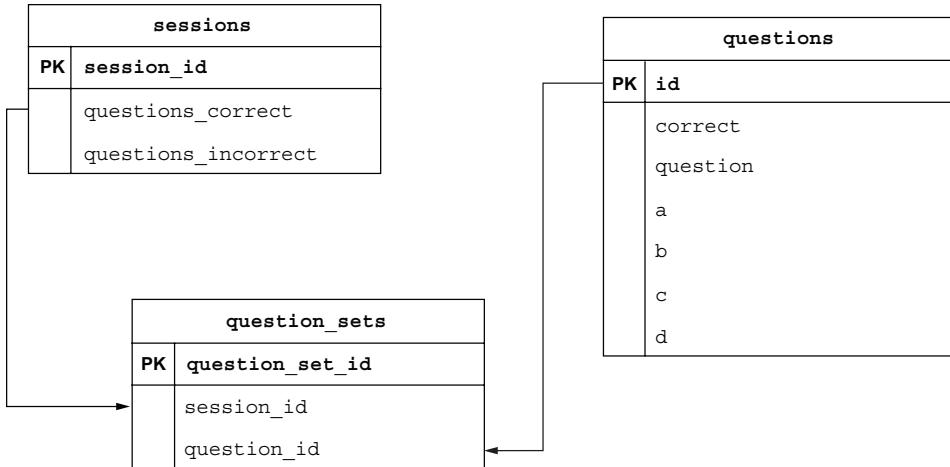


Figure 5.25 Our proposed database design that contains our sessions, question sets, and questions from the test

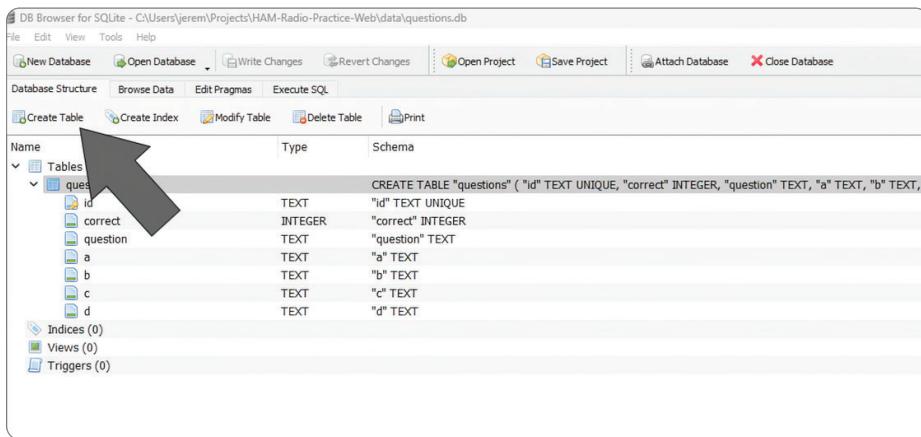


Figure 5.26 DB Browser for SQLite main screen

Then, we'll configure our session table using the dialog box in figure 5.26. We want to create each field, and in this case, they're all integer types. The `session_id` field is a primary key that's auto incremented, so we'll check the PK and AI boxes for that field (figure 5.27).

Next, we create a question set table that looks like the table in figure 5.27. This is also a set of integer fields, with `question_set_id` being a primary key that's auto incremented. The `session_id` field will link to our session ID, and the `question_id` will link to a question in the main question pool (figure 5.28).

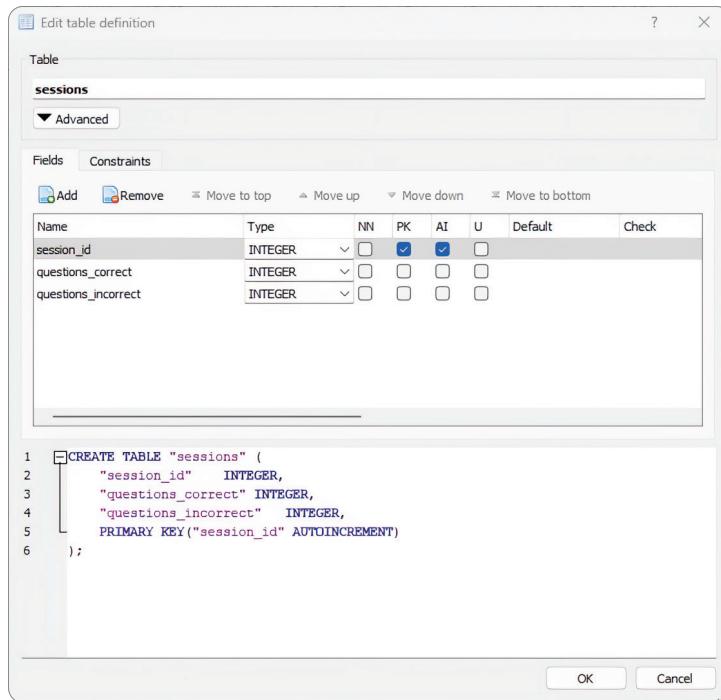


Figure 5.27 Creating table dialog box for session table

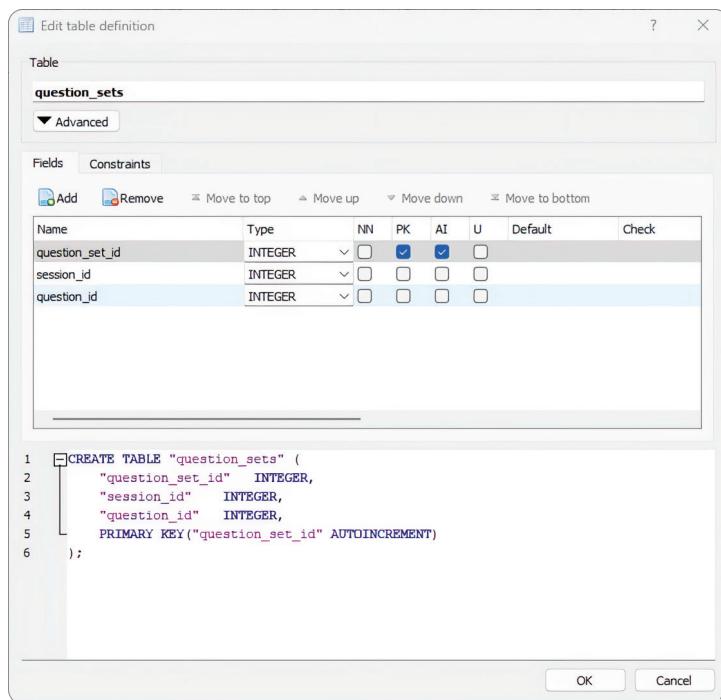


Figure 5.28 Our question set table in SQLite. We'll enter the fields you see in the top window and save them as the question_sets table.

This step completes our database structure. The design is minimal, but it will efficiently propel us to that point of MVP quickly. Make sure to click Write Changes in the top bar of DB Browser (figure 5.29).

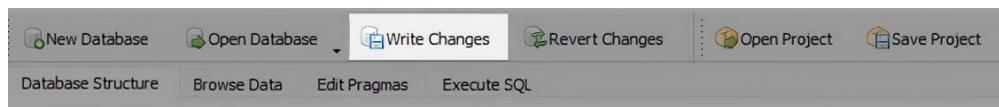


Figure 5.29 The Write Changes button in DB Browser

Your database should now look like the one in figure 5.30.

Name	Type	Schema
Tables (4)		
question_sets		CREATE TABLE "question_sets" ("question_set_id" INTEGER, "session_id" INTEGER, "question_id" INTEGER, PRIMARY KEY("question_set_id" AUTOINCREMENT))
question_set_id	INTEGER	"question_set_id" INTEGER
session_id	INTEGER	"session_id" INTEGER
question_id	INTEGER	"question_id" INTEGER
questions		CREATE TABLE "questions" ("id" TEXT UNIQUE, "correct" INTEGER, "question" TEXT, "a" TEXT, "b" TEXT, "c" TEXT, "d" TEXT, PRIMARY KEY("id"))
id	TEXT	"id" TEXT UNIQUE
correct	INTEGER	"correct" INTEGER
question	TEXT	"question" TEXT
a	TEXT	"a" TEXT
b	TEXT	"b" TEXT
c	TEXT	"c" TEXT
d	TEXT	"d" TEXT
sessions		CREATE TABLE "sessions" ("session_id" INTEGER, "questions_correct" INTEGER, "questions_incorrect" INTEGER, PRIMARY KEY("session_id" AUTOINCREMENT))
session_id	INTEGER	"session_id" INTEGER
questions_correct	INTEGER	"questions_correct" INTEGER
questions_incorrect	INTEGER	"questions_incorrect" INTEGER
sqlite_sequence		CREATE TABLE sqlite_sequence(name,seq)
name	TEXT	"name"
seq	TEXT	"seq"
Indices (0)		
Views (0)		
Triggers (0)		

Figure 5.30 The final database structure, showing all the fields in our three tables

Now, let's move on to the next task—creating code for our test session.

5.3.8 Creating code for the test session

Now we have our database set up. Let's revisit our requirements for our application:

- Create a session.
- Keep track of the 35 questions.
- Keep track of the correct answer for each.
- Track the answer the student has entered.
- Track the number of correct answers.

With our database structure in place, the next step is to add the code that will allow our application to interact with it. Let's see if BlackboxAI chat can help us out. Here is what I'll type in for a prompt:

JM I need to create a class that creates a new session. Here is the SQL used to create the table:

```
CREATE TABLE "sessions" (
    "session_id"    INTEGER,
    "questions_correct"   INTEGER,
    "questions_incorrect" INTEGER,
    PRIMARY KEY("session_id" AUTOINCREMENT)
)
```

Please create a class that will create a new session record, and `session_id` is the primary key, so it must be unique and increment from the last id. Insert zero for the `questions_correct` and `questions_incorrect` fields upon creation.

And here is what it returns—a class named `session.py` that looks like this:

```
import sqlite3

class Session:
    def __init__(self):
        self.session_id = self.create_session()
        self.questions_correct = 0
        self.questions_incorrect = 0

    def create_session(self):
        connection = sqlite3.connect('database.db')
        cursor = connection.cursor()
        cursor.execute("SELECT MAX(session_id) FROM sessions")
        result = cursor.fetchone()
        if result[0] is None:
            session_id = 1
        else:
            session_id = result[0] + 1
        cursor.execute("INSERT INTO sessions
VALUES (?, ?, ?)", (session_id, 0, 0))
        connection.commit()
        connection.close()
        return session_id
```

This code looks pretty good, except that it has database connection functionality in it. We have a `DatabaseConnection` class to handle that for us. I will create a `session.py` in the `/models` folder and save it. Then I will use this file as context and ask it to use our database class. Let's go back to Blackbox AI chat again. I open `session.py` and ask the following:

JM How can I make this class utilize my `DatabaseConnection` class located at `/models/database_connection.py`?

Here is the Blackbox AI response:



You can import the `DatabaseConnection` class in your `Session` class and use it to execute your SQL commands. Here's how you can do it:

Import the `DatabaseConnection` class at the beginning of your `Session` class file:

```
from models.database_connection import DatabaseConnection
```

Modify your `create_session` method to use the `DatabaseConnection` class instead of directly creating a SQLite connection:

```
def create_session(self):
    connection = DatabaseConnection()
    cursor = connection.cursor()
    cursor.execute("SELECT MAX(session_id) FROM sessions")
    result = cursor.fetchone()
    if result[0] is None:
        session_id = 1
    else:
        session_id = result[0] + 1
    cursor.execute("INSERT INTO sessions
VALUES (?, ?, ?)", (session_id, 0, 0))
    connection.commit()
    connection.close()
    return session_id
```

Here's the complete code for your `Session` class:

```
import sqlite3
from models.database_connection import DatabaseConnection

class Session:
    def __init__(self):
        self.session_id = self.create_session()
        self.questions_correct = 0
        self.questions_incorrect = 0

    def create_session(self):
        connection = DatabaseConnection()
        cursor = connection.cursor()
        cursor.execute("SELECT MAX(session_id) FROM sessions")
        result = cursor.fetchone()
        if result[0] is None:
            session_id = 1
        else:
            session_id = result[0] + 1
        cursor.execute("INSERT INTO sessions
VALUES (?, ?, ?)", (session_id, 0, 0))
        connection.commit()
        connection.close()
        return session_id
```

Make sure that your `database_connection.py` file is located in the `/models` directory and contains the necessary code to establish a SQLite database connection.

With this advice, I should be able to utilize my existing database connection, so I'll give it a shot. I updated the code in `session.py`, and now I need to invoke it to test it. I ask Blackbox AI how to do that:

 How can I invoke this from `APP.py` to create a new session?

Sometimes it's good to keep the prompts short and sweet. I only add additional information when it's needed to clarify something. The results it generated are helpful.

 To invoke this from your `APP.py` file, you can simply import the `Session` class and create a new instance of it. Here's how you can do it.

Import the `Session` class at the beginning of your `APP.py` file:

```
from models.session import Session
```

Create a new instance of the `Session` class:

```
session = Session()
```

Use the `session` object to interact with the session. For example, you can print the session ID like this:

```
print("Session ID:", session.session_id)
```

Here's the complete code for your `APP.py` file:

```
from models.session import Session

session = Session()
print("Session ID:", session.session_id)
```

By following these steps, you can invoke the `Session` class from your `APP.py` file and create a new session.

This looks good, and I can give it a shot. However, we do have one problem here. There is more functionality required to create our session. If we remember from our original database design, we have to build a question set and tie it to our session. The question set is unique and tied to each session. The relationship is shown in figure 5.31.

We will add this implementation code; however, we still have work to do to create a session.

5.3.9 Generating a question set

According to our requirements, we need to pull a set of 35 random questions and insert

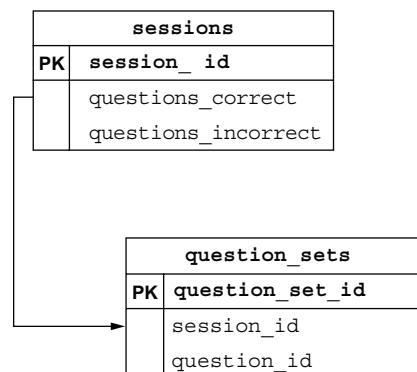


Figure 5.31 The relationship between sessions and question sets in our database

them into our `question_sets` table. It needs to be tied to our session, so we know those 35 random questions are associated with the test the student is taking. I have an idea of how this should work, but let's run it with Blackbox AI to see how it thinks we should accomplish this. The next prompt is longer and more detailed:

- JM In our `create_session()` method, I need to pull 35 random questions from our database. I can do that with the `get_question_set()` method from `questions.py`. I need to take these questions and insert them into the `question_sets` table. Here is the SQL used to create that table:

```
CREATE TABLE "question_sets" (
    "question_set_id" INTEGER,
    "session_id"      INTEGER,
    "question_id"     INTEGER,
    PRIMARY KEY("question_set_id" AUTOINCREMENT)
)
```

In the `create_session()` method, I need to

- Create a new session and store the ID generated
- Create a new `question_set_id`
- Pull 35 questions from `get_question_set()`
- Insert the ID of each question into the `question_sets` table with the `session_id` stored in the same record.

How can I refactor the `create_session` method to accomplish this?

At times, you'll find a longer more detailed prompt can get you closer the right answer faster.

- Here is a refactored version of the `create_session` method that accomplishes the task:

```
class Session:
    def __init__(self):
        self.session_id = self.create_session()
        self.questions_correct = 0
        self.questions_incorrect = 0

    def create_session(self):
        connection = DatabaseConnection()
        cursor = connection.cursor()
        cursor.execute("SELECT MAX(session_id) FROM sessions")
        result = cursor.fetchone()
        if result[0] is None:
            session_id = 1
        else:
            session_id = result[0] + 1
        cursor.execute("INSERT INTO sessions
VALUES (?, ?, ?)", (session_id, 0, 0))
        connection.commit()

        question_set_id =
```

```

        self.create_question_set(session_id, connection, cursor)
        connection.close()
        return session_id

    def create_question_set(self, session_id, connection, cursor):
        question_set_id = 1
        question_set = get_question_set(35)
        for question in question_set:
            cursor.execute("INSERT INTO question_sets
VALUES (?, ?, ?)", (question_set_id, session_id, question.id))
            question_set_id += 1
        connection.commit()
        return question_set_id

```

In this refactored version, I have added a new method called `create_question_set` that handles the creation of the question set and insertion of the questions into the `question_sets` table.

This method takes the `session_id`, `connection`, and `cursor` as parameters. It creates a new `question_set_id`, pulls 35 random questions from the `get_question_set()` method, and inserts the ID of each question into the `question_sets` table with the `session_id` stored in the same record.

The `create_question_set` method is then called within the `create_session` method, after the new session has been created and stored in the `sessions` table. This is a lot of code to digest at once. I copied and pasted this code, and right away, I noticed I must fix some import statements. In `app.py`, change

```
from app.models.questions import Questions
```

to explicitly call our method:

```
from app.models.questions import get_question_set
```

Now, it should be ready to run. I'll start up my application. At this point, the page load will create a new session (we'll fix that soon). I type `python run.py` at the prompt, and it quickly returns this error:

```

Traceback (most recent call last):
  File "/mnt/c/Users/jerem/Projects/HAM-Radio-Practice-Web/run.py", line 4,
in <module>
    from app.models.session import Session
  File "/mnt/c/Users/jerem/Projects/HAM-Radio-Practice-Web/app/models/
session.py", line 2, in <module>
    from .questions import get_question_set
ImportError: cannot import name 'get_question_set' from 'app.models.
questions' (/mnt/c/Users/jerem/Projects/HAM-Radio-Practice-Web/app/models/
questions.py)

```

The problem stems from the code not creating a new instance of `questions`. If we look at the `create_question_set` method, it's not creating a new instance:

```
def create_question_set(self, session_id, connection, cursor):
    question_set_id = 1
    question_set = get_question_set(35)
    for question in question_set:
        cursor.execute("INSERT INTO question_sets
VALUES (?, ?, ?)", (question_set_id, session_id, question.id))
        question_set_id += 1
    connection.commit()
    return question_set_id
```

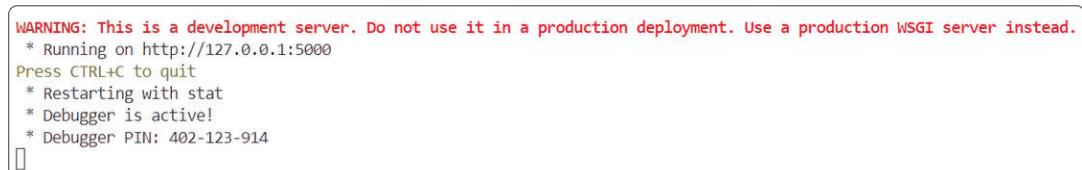
I need to add the following import:

```
from .questions import Questions
```

The new method should look like this:

```
def create_question_set(self, session_id, connection, cursor):
    questions = Questions()
    question_set_id = 1
    question_set = questions.get_question_set(35)
    for question in question_set:
        cursor.execute("INSERT INTO question_sets
VALUES (?, ?, ?)", (question_set_id, session_id, question.id))
        question_set_id += 1
    connection.commit()
    return question_set_id
```

Now I will try to run `python run.py` again, and it loads successfully (figure 5.32).



```
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: 402-123-914
```

Figure 5.32 A successful load of our Flask application

However, once we load up a page in the browser (figure 5.33), I get a `TypeError`.

I know why this has gone wrong, and if you're paying close attention, you do as well. Our generated code in `create_session` looks like this:

```
connection = DatabaseConnection()
```

The class requires a `db` path to be injected into it. We could easily write

```
connection = DatabaseConnection('data/questions.db')
```

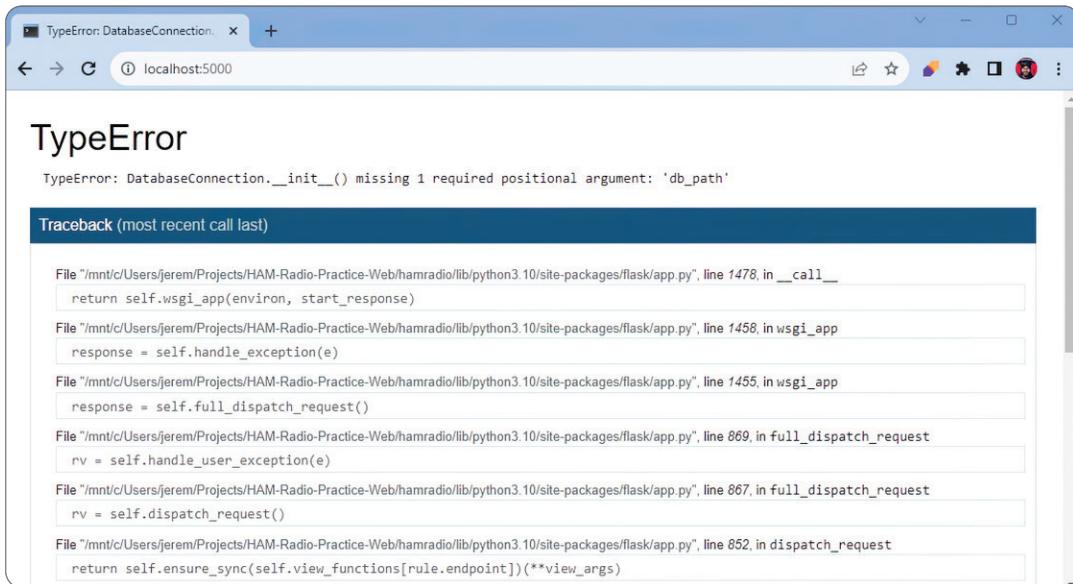


Figure 5.33 `TypeError` from our application.

This code would likely work. However, it is bad practice. If we set the pathname location of our database in multiple places, when the name changes, we must update it multiple times in several files. Let's inject this pathname into the `Session` class instead. That way, we only specify the pathname in one place. Let's open `session.py` and make some small changes.

Change this line of code

```

class Session:
def __init__(self):

```

to

```

class Session:
def __init__(self, cursor):

```

Next, add the following under the definition:

```
    self.cursor = cursor
```

This line will pass in the cursor when you initialize the session. In this manner, we can use the existing database connection throughout each step of the process. Right now, the `create_session` method looks like this:

```

def create_session(self):
    connection = DatabaseConnection()
    cursor = connection.cursor()
    cursor.execute("SELECT MAX(session_id) FROM sessions")
    result = cursor.fetchone()
    if result[0] is None:
        session_id = 1
    else:
        session_id = result[0] + 1
    cursor.execute("INSERT INTO sessions
VALUES (?, ?, ?)", (session_id, 0, 0))
    connection.commit()
    question_set_id = self.create_question_set(session_id, connection,
cursor)
    connection.close()
    return session_id

```

We need to refactor code now so that it uses the cursor we passed in. We make the following changes:

```

def create_session(self):
    self.cursor.execute("SELECT MAX(session_id) FROM sessions")
    result = self.cursor.fetchone()
    if result[0] is None:
        session_id = 1
    else:
        session_id = result[0] + 1
    self.cursor.execute("INSERT INTO sessions
VALUES (?, ?, ?)", (session_id, 0, 0))

    question_set_id = self.create_question_set(session_id)
    return session_id

```

This refactored code will use `self.cursor` instead of creating a new `DatabaseConnection` and `cursor`. We need to refactor `create_question_set` in the same way. Here is what it looks like now:

```

def create_question_set(self, session_id, connection, cursor):
    questions = Questions()
    question_set_id = 1
    question_set = questions.get_question_set()
    for question in question_set:
        cursor.execute("INSERT INTO question_sets
VALUES (?, ?, ?)", (question_set_id, session_id, question.id))
        question_set_id += 1
    connection.commit()
    return question_set_id

```

Here is the new method, using our cursor:

```

def create_question_set(self, session_id):
questions = Questions(self.cursor)
question_set_id = 1

```

```
question_set = questions.get_question_set(35)
for question in question_set:
    self.cursor.execute("INSERT INTO question_sets
VALUES (?, ?, ?)", (question_set_id, session_id, question[0]))
    question_set_id += 1
return question_set_id
```

Now we're ready to load up the page. Once again, we'll start up our application:

```
python run.py
```

I can see immediately there are no errors in my console, which is a good first sign (figure 5.34). We've done a lot of refactoring here!

```
* 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: 400-475-304
```

Figure 5.34 There are no errors present in the console, which means our application started properly.

And now I load up the web page at <http://localhost:5000> and see a list of question IDs (figure 5.35). This is exactly what we want.

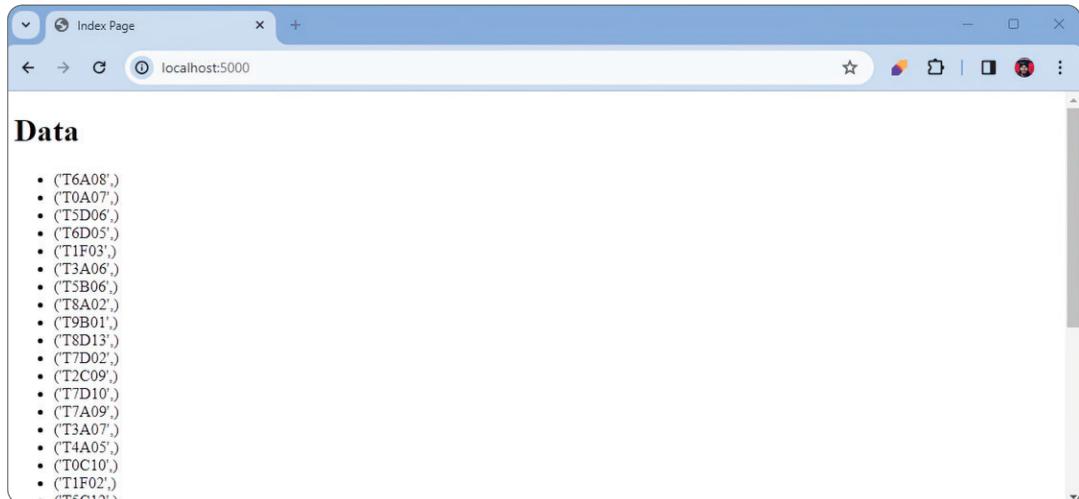


Figure 5.35 Our web browser screen showing IDs

Why are we excited about seeing question IDs? Because this verifies that we've pulled a set of random questions using the following line of code:

```
data = questions.get_question_set()
```

We can assume it worked because we didn't see an error. In fact, we also print out the data variable to our console and can see the same set there, as well as our session ID (figure 5.36).

```
Session ID: 1
[('T6A08'), ('T8A07'), ('T5D06'), ('T6D05'), ('T1F03'), ('T3A06'), ('T5B06'), ('T8A02'), ('T9B01'), ('T8D13'), ('T7D02'), ('T2C09'), ('T7D10'), ('T7A09'), ('T3A07'), ('T4A05'), ('T0C10'), ('T1F02'), ('T5C12'), ('T1E08'), ('T4B04'), ('T8D03'), ('T6D10'), ('T2B02'), ('T0B12'), ('T8A05'), ('T5C02'), ('T7C11'), ('T7C04'), ('T0A02'), ('T1A09'), ('T4B05'), ('T6C10'), ('T2B06'), ('T8A04')]
127.0.0.1 - [13/Dec/2023 16:35:48] "GET / HTTP/1.1" 200 -
```

Figure 5.36 Our console output

This is a promising sign that the implementation succeeded—but further verification is still required.

5.3.10 Verifying our test session was created

As a recap, here are our primary objectives for our MVP. We've been working on the first objective so far:

- Create a session.
- Keep track of the 35 questions.
- Keep track of the correct answer for each.
- Track the answer the student has entered.
- Track the number of correct answers.

In our session creation, we aimed to

- Create a session with an ID
- Pull 35 random questions
- Attach them to our session

We can verify that by looking at our database. First, let's check out the session table. It should have a `session_id` of 1, since it's our first session, and we should have neither correct nor incorrect answers yet because the student hasn't answered any questions. We look at our sessions table (figure 5.37) and see that it's true.

Next, we need to see whether we indeed inserted 35 questions at random into our `question_sets` table. It appears we did. As shown in figure 5.38, I have 35 questions associated with `session_id` 1.

Our initial setup is complete and is working as expected. We can create a new session with a set of questions. We're on the right track, but there's still more to build before the

session_id	questions_correct	questions_incorrect
Filter	Filter	Filter
1	1	0

Figure 5.37 A view of our sessions table showing our first ID and no questions answered

question_set_id	session_id	question_id
Filter	Filter	Filter
1	1	T1A07
2	1	T2A01
3	1	T8D02
4	1	T3B04
5	1	T6A02

Figure 5.38 A view of our question_sets table showing random questions associated with our session_id

application is fully functional. In the next chapter, we’re going to run into a bug in the application we’ve just built. We will use Tabnine as our tool this time to tackle the bug and add more functionality to our application.

5.3.11 Conclusion

We collaborated with Blackbox AI to successfully develop a functioning web application. We used the tool to quickly generate the initial code and advance development. Next, we intervened to refine the architecture for maintainability, ensured proper separation of concerns, added our own code, and overrode AI suggestions when beneficial. This is common practice.

The resulting application meets our core requirements, with a suitable database schema and Python code to manage test sessions. Our application is generating random question sets, creating test sessions, and tracking test state. However, it still needs additional work to be usable.

By combining AI generation with human judgment, we rapidly developed an application foundation. We used AI accelerated suggestions to flesh it out, and then I provided a bit of course correction based on my experience. The process demonstrated the immense value these tools provide, while highlighting the importance of human expertise. The future is a partnership between humans and AI—each playing a vital role. We'll continue this partnership as we build a usable application.

Summary

- Generative AI tools such as BlackboxAI can speed up development. They generate initial code, letting developers focus on refining architecture and improving quality.
- Clear and specific context in prompts is key when using AI coding tools. Vague requests lead to generic or incorrect code, while detailed prompts with clear requirements yield better results.
- Separation of concerns is key in AI-assisted development. We demonstrated it by keeping database connections in dedicated classes instead of mixing them in the application.
- Human intervention is required to adjust AI-generated code. It is essential when the code doesn't follow best practices or meet the application's needs.
- Good database design should come before code generation. Even with AI help, proper schema planning helps avoid technical debt and future problems.
- Context management patterns, such as Python's `with` statement, ensure proper resource handling. These fundamentals are still vital in AI-accelerated development.
- Rather than replacing us, AI should serve as an intelligent partner in our work. It speeds up implementation but requires human judgment for design and architecture choices.
- Testing and checking the database are key with AI-generated code. This aspect helps find problems that might be missed in code reviews alone.