



# *Generating a software backend with Tabnine*

---

## **This chapter covers**

- Creating and maintaining persistent sessions for test-takers
- Implementing backend functionality in Flask with database integration
- Using Tabnine to generate code solutions and fix errors
- Crafting effective prompts to troubleshoot technical challenges
- Building an interactive interface that maintains user state

In chapter 5, we set up our practice test application using Blackbox AI for the database and question selection. In this chapter, we'll create a user-friendly experience—the session management and user interface components—which will make our question database interactive.

We'll use Tabnine as our AI coding assistant. It will help us fix bugs, add new features, and simplify the development process. But before we move on, let's review our progress with the user stories:

- *User Story 1*—As a developer, I want to create a question pool database:
  - *Task 1.1*—Choose an appropriate database technology (SQL or NoSQL) based on the structure and volume of data
  - *Task 1.2*—Design a table/collection with fields `question_id`, `question_text`, `options`, and `correct_option`
- *User Story 2*—As a developer, I want to implement a question selector:
  - *Task 2.1*—Develop a module to randomly select 35 questions from the question pool database for each test
  - *Task 2.2*—Implement an algorithm to ensure true randomness in question selection and prevent the same question from appearing more than once in a single test
  - *Task 2.3*—Test the module for randomness and verify it doesn't repeat questions within a single test

With these basics covered, we can focus on key user experience needs: managing sessions well, ensuring questions persist, and making a user-friendly interface for test-takers. By the end of this chapter, we'll have an application that maintains state across browser sessions and gives users smooth testing experience.

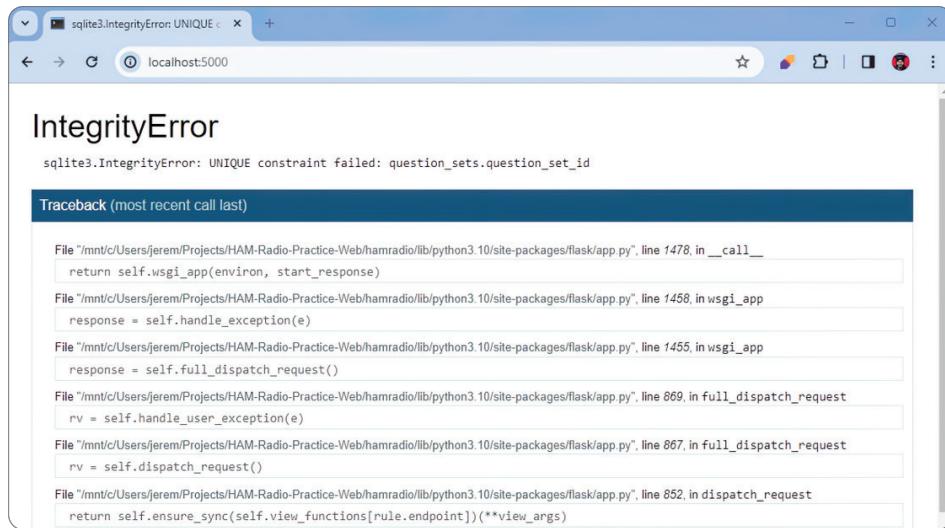
Before creating a cohesive UI, we need to complete some work on the backend. We have the following objectives to think about:

- We must verify whether a session is created on the first visit to the web page.
- We must ensure that a set of questions is added to the pool, that the session persists, and that we can step through the test with the same set of questions.
- Our application should keep the session in the browser in case the user closes it.
- The user should be able to destroy the session at any time and restart the test.

We'll tackle these problems by using Tabnine as our guide to improve our application and get it closer to our requirements. We'll create a session that persists in the application and assign our question set to it. By doing so, we can ensure that the questions persist as the student goes through them. The session will also remain active if the user closes the browser window by accident.

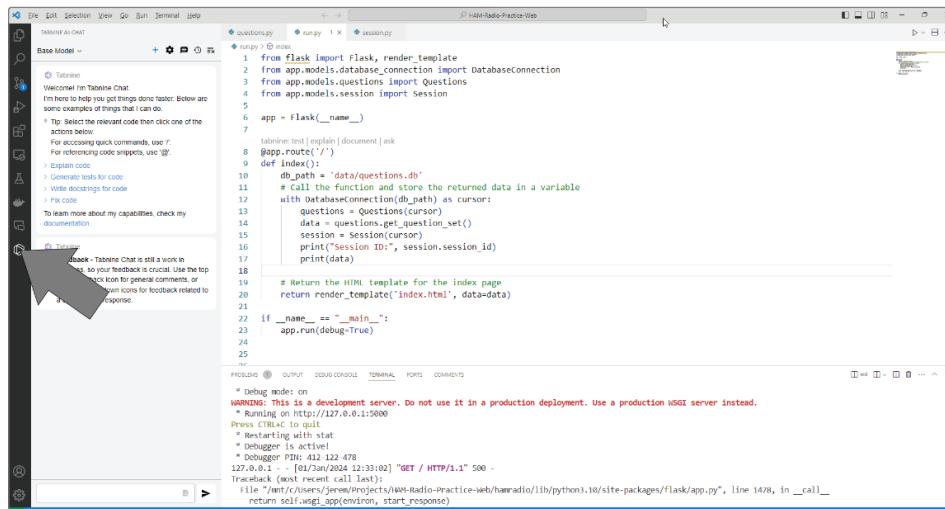
## 6.1 **Creating a session and our first bug**

In the last chapter, we launched our application and loaded it into a web browser. This initiated a test session, creating a unique ID and pulling a set of 35 random questions. We wrapped up the chapter upon verifying that the process was successful. However, if we open the application a second time, here's what we'll see (figure 6.1).



**Figure 6.1 A SQLite error that comes up when we load the application a second time**

We know the application worked the first time we loaded it. Now we see an error, and it's clear that it has something to do with creating a new session. Because we closed the browser and restarted, the application should start a new session. Let's see if Tabnine can help us with this problem. We'll load up Tabnine chat in Visual Studio Code by clicking on the Tabnine icon in the activity bar, as shown in figure 6.2.



**Figure 6.2 Opening Tabnine chat in our activity bar. The chat window opens to the left of the main IDE window.**

A chat window opens, enabling us to communicate with the system just as we've done using other tools (figure 6.3).

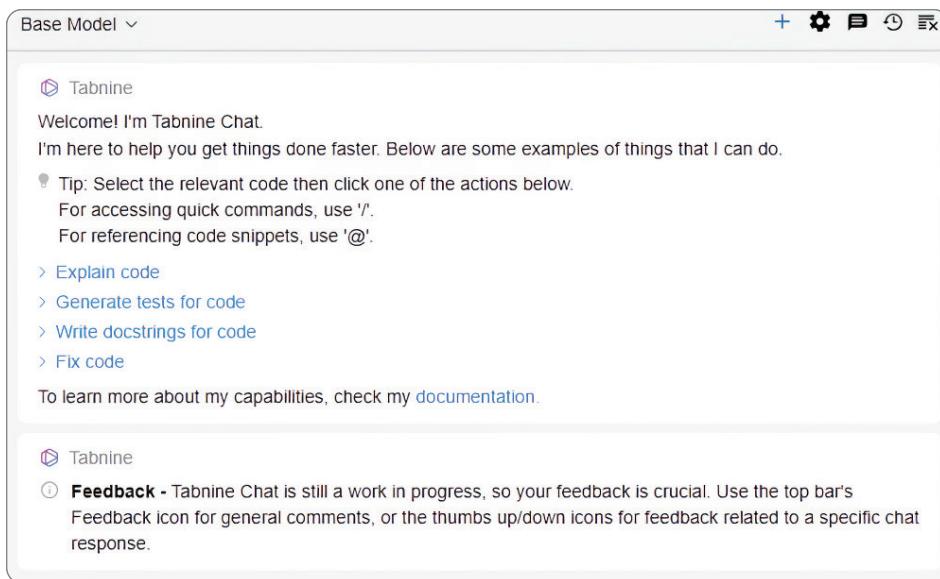


Figure 6.3 The Tabnine Chat window supporting simple conversational interaction

Let's copy the error from our browser and ask Tabnine chat if it can assist us. Here's the first prompt we'll send to the chat window:

JM I'm getting the following error:  
IntegrityError  
`sqlite3.IntegrityError: UNIQUE constraint failed: question_sets.  
question_set_id`  
Can you help with this?

This is a simple prompt, and we aren't giving it much context, so let's see what Tabnine comes up with. Here is the chat response:

The error message suggests that there is a problem with the uniqueness of the `question_sets.question_set_id` column. This could be due to a duplicate entry in the database or a missing entry.

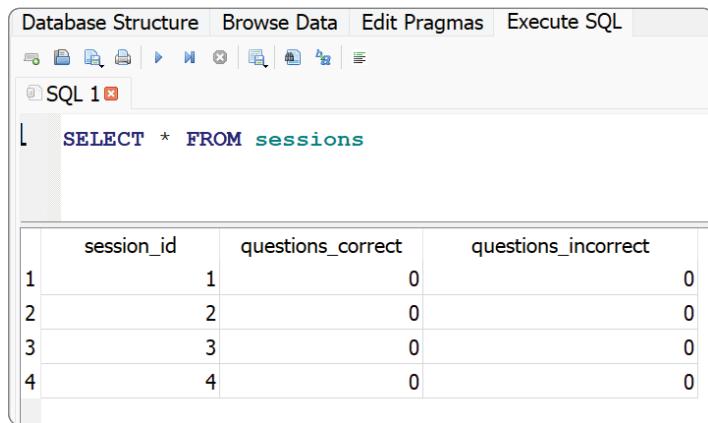
To resolve this issue, you can try the following steps:

- 1 Check the database for any duplicate entries in the `question_sets` table. You can use a SQL query to do this.

- 2 If there are no duplicate entries, you can add the missing entry to the table.
- 3 If the problem persists, you may need to investigate further and try other solutions.

Let me know if these steps help resolve the issue.

This response gives us a good start for troubleshooting. One thing to note: if I refresh the page, a new session is created every time. Things are working as expected for that part of the process (figure 6.4).



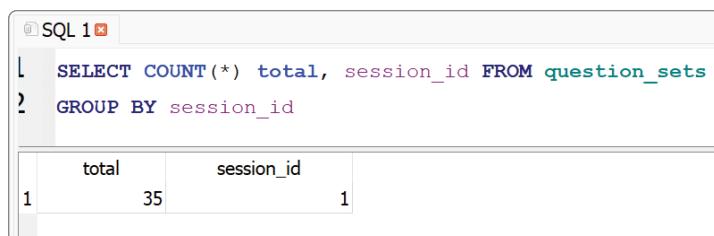
The screenshot shows a SQLite database interface with the following details:

- Toolbar:** Database Structure, Browse Data, Edit Pragmas, Execute SQL.
- SQL Editor:** SQL 1, containing the query: `SELECT * FROM sessions`.
- Result Table:**

	session_id	questions_correct	questions_incorrect
1	1	0	0
2	2	0	0
3	3	0	0
4	4	0	0

**Figure 6.4** The sessions table shows a valid session created every time we refresh the page. The session\_id is incremented properly so we know this is correct.

We know the session is being created, but Tabnine is suggesting that there is a problem with the uniqueness of the `question_sets.question_set_id` column. We can confirm this with a query. If it's working properly, we'll see sets of questions with `session_ids` of 1, 2, 3, and 4 (figure 6.5).



The screenshot shows a SQLite database interface with the following details:

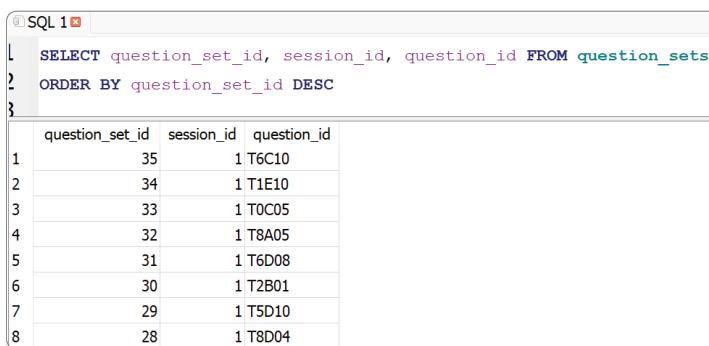
- SQL Editor:** SQL 1, containing the query: `SELECT COUNT(*) total, session_id FROM question_sets GROUP BY session_id`.
- Result Table:**

	total	session_id
1	35	1

**Figure 6.5** We only have one set of 35 questions for session\_id 1. New question sets are not being created in this table.

Tabnine is right in suggesting the problem lies in the `question_sets` table, and it's caused by a unique constraint on the `question_sets.question_set_id`. When we create a new session, it's trying to insert a new set, starting with a `question_set_id` of

1 instead of starting where the last `question_set_id` left off. We can verify that with another query (figure 6.6).



```
SQL 1
SELECT question_set_id, session_id, question_id FROM question_sets
ORDER BY question_set_id DESC
```

	question_set_id	session_id	question_id
1	35	1	T6C10
2	34	1	T1E10
3	33	1	T0C05
4	32	1	T8A05
5	31	1	T6D08
6	30	1	T2B01
7	29	1	T5D10
8	28	1	T8D04

**Figure 6.6** Our numbering ends at 35 in the `question_sets` table, indicating we had one successful set of questions inserted, and no new sets can be created.

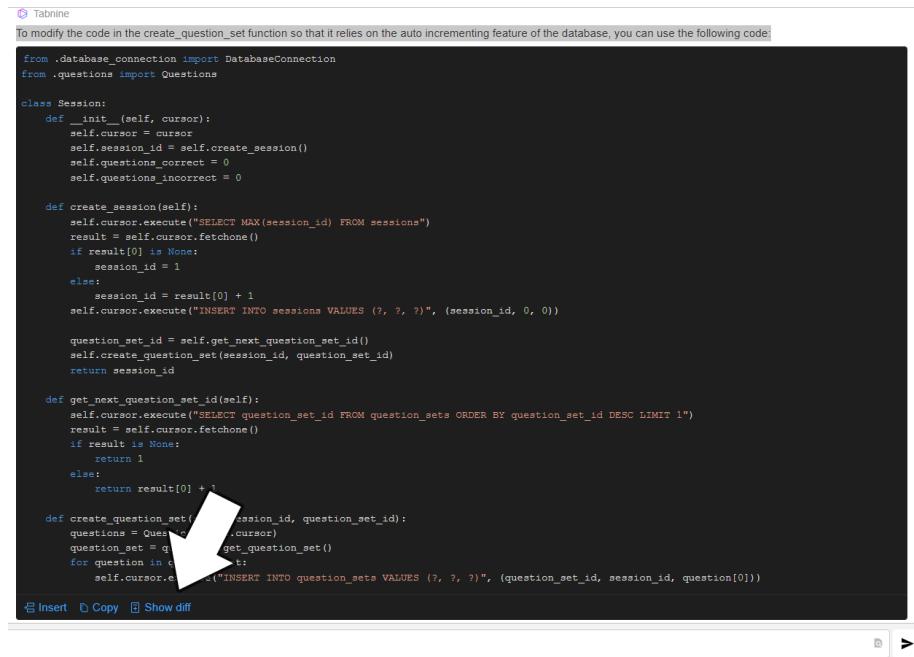
Now we have a theory about our root cause. The `question_set_id` column is supposed to be unique, and we're trying to insert numbers into it, rather than let it auto-increment on its own. Let's see if Tabnine can help us. Here's the next question I'll ask Tabnine Chat:

JM I have a unique constraint on the `question_sets.question_set_id` column. It is also an auto-incrementing column. How can I modify the code in the `create_question_set` function so that it does not insert a new id and instead relies on the auto incrementing feature of the database?

This should give Tabnine enough context to assume what I'm looking for, which is a way to keep auto-incrementing the ID. Tabnine returns with a bunch of new code to attempt to solve our problem. The results are shown in figure 6.7. But I'm hesitant to copy and paste a giant chunk of code into my file. If it breaks something, trying to find out which change caused the breakage could be a cat and mouse game. So instead, I'll use a cool Tabnine feature—the “Show diff” feature—to display which portions of code have changed. In figure 6.7, you can also see the button at the bottom of the code window returned in the chat.

The term “diff” refers to a tool used to compare two sets of data. It’s usually two text files. If you have two documents and want to compare them, you can quickly see what changed. A diff highlights both additions and subtractions of text and displays those changes on the screen. This is useful for things such as tracking modifications to code and merging conflicts.

By only showing the differences (diff) of the changes, we can get a better idea of what needs to change. This is a personal preference. If you’ve ever copied and pasted a large amount of code you don’t fully understand, you’ve likely run into problems later when something breaks. I would rather understand the changes exactly and implement them myself, and this feature makes it much easier to do so (figure 6.8).



To modify the code in the `create_question_set` function so that it relies on the auto incrementing feature of the database, you can use the following code:

```

from .database_connection import DatabaseConnection
from .questions import Questions

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

    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))

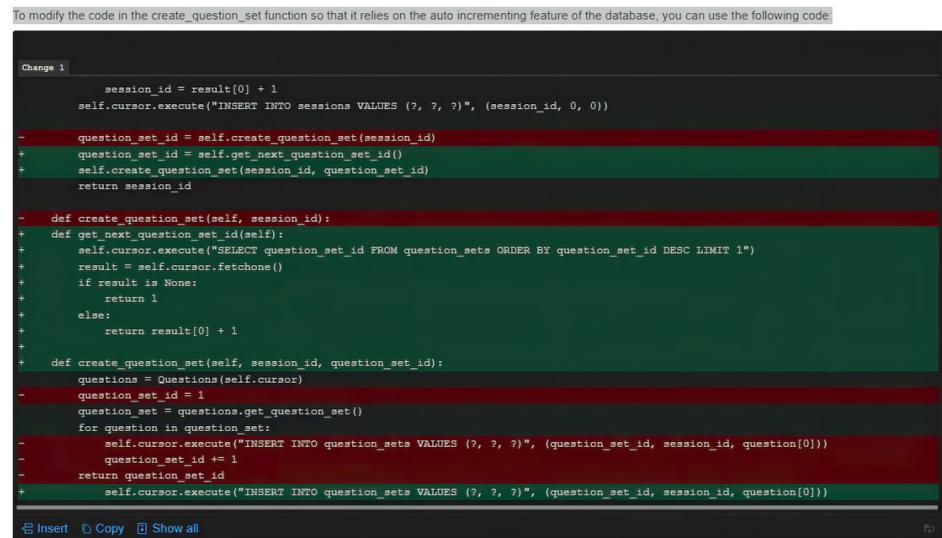
        question_set_id = self.get_next_question_set_id()
        self.create_question_set(session_id, question_set_id)
        return session_id

    def get_next_question_set_id(self):
        self.cursor.execute("SELECT question_set_id FROM question_sets ORDER BY question_set_id DESC LIMIT 1")
        result = self.cursor.fetchone()
        if result is None:
            return 1
        else:
            return result[0] + 1

    def create_question_set(self, session_id, question_set_id):
        questions = Questions(self.cursor)
        question_set = questions.get_question_set()
        for question in question_set:
            self.cursor.execute("INSERT INTO question_sets VALUES (?, ?, ?)", (question_set_id, session_id, question[0]))

```

**Figure 6.7** This is the code Tabnine has generated to fix our problem. Rather than copy and paste everything in the window, I'll use “Show diff” to see which code is changed.



To modify the code in the `create_question_set` function so that it relies on the auto incrementing feature of the database, you can use the following code:

```

Change 1
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)
+     question_set_id = self.get_next_question_set_id()
+     self.create_question_set(session_id, question_set_id)
return session_id

-     def create_question_set(self, session_id):
+     def get next question set id(self):
+         self.cursor.execute("SELECT question_set_id FROM question_sets ORDER BY question_set_id DESC LIMIT 1")
+         result = self.cursor.fetchone()
+         if result is None:
+             return 1
+         else:
+             return result[0] + 1
+
+     def create_question_set(self, session_id, question_set_id):
        questions = Questions(self.cursor)
-     question_set_id = 1
        question_set = questions.get_question_set()
        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
+             self.cursor.execute("INSERT INTO question_sets VALUES (?, ?, ?)", (question_set_id, session_id, question[0]))

```

This code gets the next question set ID from the `question_sets` table by ordering by `question_set_id` in descending order and taking the next value. It then calls the `create_question_set` method with the session ID and the next question set ID.

**Figure 6.8** The diff view of the proposed changes showing exactly which lines of code need to be modified to implement Tabnine's suggestions

And here's another great feature of Tabnine. If you click Insert, the changes will be inserted into your code, and you can go step by step through each change. You can

- Accept current change
- Accept incoming change
- Accept both changes
- Compare changes

The Insert feature is shown in figure 6.9.

```

Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<< Current (Current Change)
def create_question_set(self, session_id):
=====
tabnine: test | explain | document | ask
def get_next_question_set_id(self):
    self.cursor.execute("SELECT question_set_id FROM question_sets ORDER BY question_set_id DESC LIMIT 1")
    result = self.cursor.fetchone()
    if result is None:
        return 1
    else:
        return result[0] + 1
  
```

**Figure 6.9** You can insert the code as part of the diff process and accept changes within the IDE.

### Debugging with AI tools

While AI coding assistants can offer solutions to bugs, remember that they're not infallible. The most effective approach combines AI suggestions with your own critical thinking:

- Show errors directly to AI tools for targeted help.
- Use diff views to understand exactly what changes are being proposed.
- Implement changes incrementally and test after each change.
- Be prepared to refine prompts when initial suggestions don't fully solve the problem.
- Remember that sometimes the AI will fix symptoms rather than root causes.

Now I have inserted the code suggested by Tabnine. Here is what the final `session.py` code looks like now:

```

from .database_connection import DatabaseConnection
from .questions import Questions

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

    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))

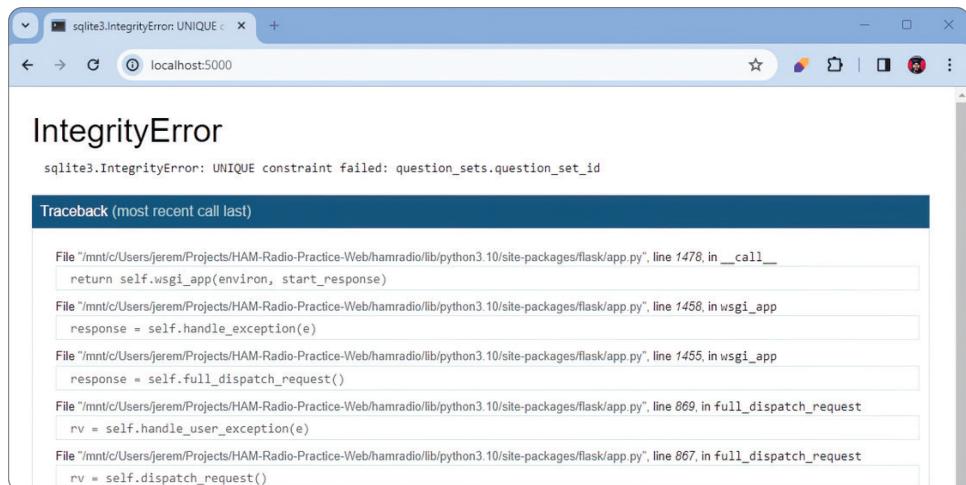
        question_set_id = self.get_next_question_set_id()
        self.create_question_set(session_id, question_set_id)
        return session_id

    def get_next_question_set_id(self):
        self.cursor.execute("SELECT question_set_id
        FROM question_sets ORDER BY question_set_id DESC LIMIT 1")
        result = self.cursor.fetchone()
        if result is None:
            return 1
        else:
            return result[0] + 1

    def create_question_set(self, session_id, question_set_id):
        questions = Questions(self.cursor)
        question_set = questions.get_question_set()
        for question in question_set:
            self.cursor.execute("INSERT INTO question_sets
VALUES (?, ?, ?)", (question_set_id, session_id, question[0]))

```

We're now creating a new `session_set_id` based on the last ID value. I will refresh the page. If this fix is valid, we should see a new session created and no errors. However, after refreshing the window, I see the same error again (figure 6.10).



**Figure 6.10** The same error as before—a unique constraint violation in the same field

We're getting the same error, but why? I can take a guess at it. You may have caught this error as well if you looked closely at the code above. Let's examine our code in the `create_session` function:

```
question_set_id = self.get_next_question_set_id()
self.create_question_set(session_id, question_set_id)
```

This code was modified to get the next ID in the series and then create a question set with that. However, remember that the question set is 35 questions, which means that in our `create_question_set` method, we attempt to insert that new ID 35 times!

```
def create_question_set(self, session_id, question_set_id):
    questions = Questions(self.cursor)
    question_set = questions.get_question_set()
    for question in question_set:
        self.cursor.execute("INSERT INTO question_sets
VALUES (?, ?, ?)", (question_set_id, session_id, question[0]))
```

This is certainly not going to work, meaning we can only use that new ID value one single time—then it's going to break again. To verify this, let's do a quick check of the database. Sure enough, there's only one record inserted with a `question_set_id` of 36, and a `session_id` of 6. On the next insert, we violated the constraint by trying to insert 36 as the next `question_set_id` value (figure 6.11).

	question_set_id	session_id	question_id
1	36	6	T7B02
2	35	1	T6C10
3	34	1	T1E10
4	33	1	T0C05
5	32	1	T8A05

**Figure 6.11** Our method inserted record 36 correctly. However, as we iterate through the question set to add new questions, we're still inserting 36 for the `question_set_id` value.

Tabnine took our request literally but didn't fully understand the problem. The output looks correct, but it's not functional. It's generating a new incremented ID *one time*. However, we need to do that each time we add a new question. This is an example of why we shouldn't rely on output of the tools too much. Not only are they fallible, but we may not frame the prompt to express exactly what we need. Code that "looks right" can be wrong, and we already know this as developers. It's best to understand the code being generated.

We need to modify this piece of code to accomplish our goal:

```
for question in question_set:
    self.cursor.execute("INSERT INTO question_sets
VALUES (?, ?, ?)", (question_set_id, session_id, question[0]))
```

We can easily make that happen, but let's ask Tabnine for help. This time, we'll craft our idea into a query for the chat to interpret. Here's my next prompt for Tabnine:



In our `create_question_set` function, we have an error. It is attempting to add new records into `question_sets`; however, `question_set_id` is an auto-incremented field. How do we modify the code within the `create_question_set` method so that we utilize the auto-increment functionality?

This should provide additional detail and context to solve our problem. Here is the answer I received:



To modify the code within the `create_question_set` method so that it utilizes the auto increment functionality, you can modify the code as follows:

```
def create_question_set(self, session_id):
    questions = Questions(self.cursor)
    question_set = questions.get_question_set()
    for question in question_set:
        self.cursor.execute("INSERT INTO question_sets
VALUES (DEFAULT, ?, ?)", (question_set_id, session_id, question[0]))
```

In this modified code, we are using the `DEFAULT` keyword in the SQL statement to let the database automatically assign a value to the `question_set_id` column.

I'll save you some time—this method did not work. That's because, as far as I know, the `DEFAULT` keyword doesn't work in SQLite. It's once again time for human intervention. I know how the auto-increment flag is supposed to work, and I am familiar enough with SQLite databases to know that if you insert a null value into an auto-incremented field, it will generate a new value. I'll give Tabnine an informal, conversational prompt:



Default doesn't work here, can I just insert a null value into the database?

It immediately returns an answer that will work for me (figure 6.12).

Jeremy Morgan

Default doesn't work here, can I just insert a null value into the database?

Tabnine

Yes, you can insert a null value into the database by using the SQL `NULL` value. You can modify the code as follows:

```
def create_question_set(self, session_id):
    questions = Questions(self.cursor)
    question_set = questions.get_question_set()
    for question in question_set:
        self.cursor.execute("INSERT INTO question_sets VALUES (NULL, ?, ?)", (session_id, question[0]))
```

In this modified code, we are using the SQL `NULL` value as the value for the `question_set_id` column.

**Figure 6.12** I asked Tabnine an informal question and suggested a solution. It responded with a solution that will work.

## Crafting effective code prompts

The quality of AI code assistance depends heavily on your prompting approach:

- Start with clear, specific descriptions of what you want to accomplish.
- Include relevant context about your codebase and frameworks.
- For complex problems, try conversational, informal prompts.
- When you have a potential solution in mind, don't be afraid to steer the AI by suggesting an approach.
- Follow up with clarifying questions when suggestions aren't working.

Now let's change our insert line from this:

```
self.cursor.execute("INSERT INTO question_sets VALUES  
(DEFAULT, ?, ?)", (question_set_id, session_id, question[0]))
```

to the following line of code, which simply uses "None" as the datatype to insert. By inserting a null value, SQLite will automatically create the `question_set_id` and increment it properly:

```
self.cursor.execute("INSERT INTO question_sets VALUES  
(null, ?, ?)", (session_id, question[0]))
```

I'm sure Tabnine would have generated a similar solution first, with enough attempts. Sometimes, if you know the answer or feel strongly about an idea, it's good to steer the AI into a certain direction. We refresh the page and see the window displayed in figure 6.13.

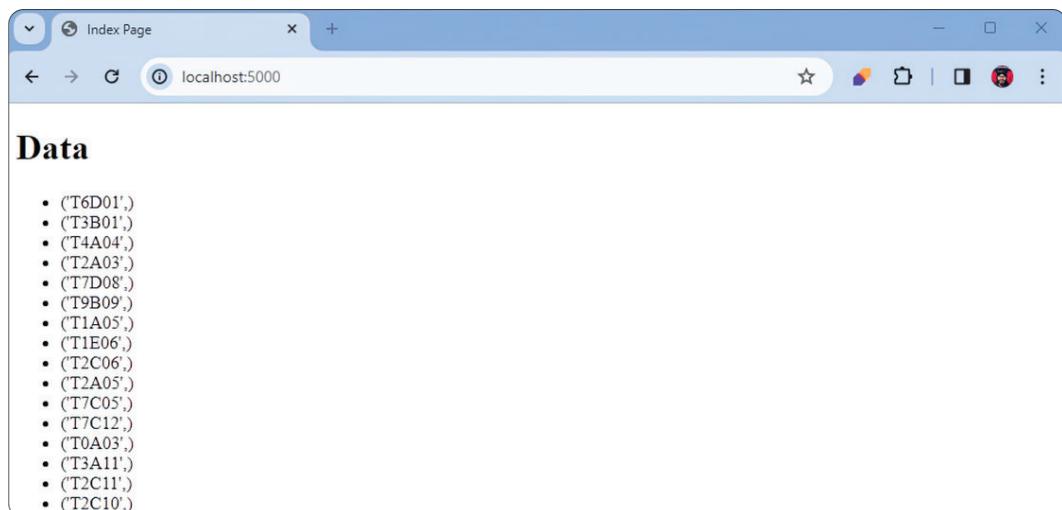


Figure 6.13 This is our makeshift success screen. It shows the IDs of the 35 questions inserted into our database.

With this screen, we know we're inserting records in our table again without an error. I'll refresh it a few times and create multiple sessions. We can then validate it by looking at our database again. We can count the number of questions and group them by session\_id. There should be 35 questions in each set, except for the session that failed (figure 6.14).

	total	session_id
1	35	1
2	1	6
3	35	20
4	35	29
5	35	30
6	35	31
7	35	32
8	35	33

**Figure 6.14** We can see a list of sessions here with 35 questions each. Session 6 failed on insert, so there is only one question available.

Now that we have that bug squashed, we need to move on to a better way to create our sessions. Right now, just loading the website up creates a new session. Let's put a user interface on it so the user can create the session. Then, we'll add functionality to handle the questions.

## 6.2 Creating an index page

So far, our application can start a session, from a technical standpoint, but it's a little rough. We must reload the page to start a session, and we can't do anything after that except view a few QuestionIDs in a list. Let's change that.

In our `run.py`, we have a line of code that produces our index page:

```
return render_template('index.html', data=data)
```

This loads up a file located in `/templates`, and it looks like this:

```
<!-- 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>

```

This file doesn't do much more than list every item in our data array. As it's not incredibly useful, let's create a function to start a new session with the click of a button. This gives the user control over when to start a session. Then we can store that session, and the user can go on to answer questions. Let's ask Tabnine to help us out with that. Here is the prompt I sent:

-  JM How can I add a button to this page that will start our session for us? In `run.py`, our question session is started with `"data = questions.get_question_set()"`. I would like to add a form with a button to call that function. How would I code this?

In this prompt, I'm attempting to be as specific as possible about what I want to do, without revealing too much about how it's done. I don't want to give the solution away but ask for one. Here is the output:

-  To add a button to the `index.html` page that starts the question session, you can add the following code to the `index.html` file:

```

<form action="/" method="POST">
    <input type="submit" value="Start Question Session">
</form>

```

This code adds a form with a submit button to the `index.html` page. When the user clicks the submit button, the form will submit to the root URL (`/`), which is where the `index.py` file is located.

In the `index.py` file, you can add the following code to handle the form submission:

```

@app.route('/', methods=['POST', 'GET'])
def index():
    # Check if the form has been submitted
    if request.method == 'POST':
        # Start the question session
        #

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

```

This code checks if the request is a `POST` request. If it is, it starts the question session. Otherwise, it returns the `index.html` template with the `data` variable.

You can replace the code in the `index.py` file that retrieves the question data with the following code:

```

def index():
    # Start the question session

```

```

with DatabaseConnection(db_path) as cursor:
    questions = Questions(cursor)
    data = questions.get_question_set()
    session = Session(cursor)
    print("Session ID:", session.session_id)

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

```

This code starts the question session before retrieving the data, which ensures that the session ID is generated before the data is retrieved.

Let's put this suggestion to test. I'll follow the instructions precisely, and we'll see how the result turns out.

In `index.html`, below this line

```
<h1>Data</h1>
```

add

```

<form action="/" method="POST">
    <input type="submit" value="Start Question Session">
</form>

```

In the `run.py` file, change

```
@app.route('/')
```

to the suggested line

```
@app.route('/', methods=['POST', 'GET'])
```

Under this line of code, within the function

```
def index():
```

add the following:

```

# Check if the form has been submitted
if request.method == 'POST':
    # Start the question session
    # ...

```

Then we can indent the rest of the code, as shown in figure 6.15.

Now we know that a session will only be created if we ask for it. This is a great suggestion; however, I have a feeling it's not quite complete. If you are following along, you'll notice Tabnine catches a small problem here, as shown in figure 6.16.

```

# Check if the form has been submitted
if request.method == 'POST':
    # Start the question session
    #
    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.get_question_set()
        session = Session(cursor)
        print("Session ID:", session.session_id)
        print(data)

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

```

**Figure 6.15** The remainder of the question creation code should be indented so it is only executed if this is a POST command.

The screenshot shows a code editor with Python code. Line 11 contains an error: `if request.method == 'POST':` is not indented under the `if` block. A yellow exclamation mark icon is next to the line. A tooltip box titled "Quick Fix" appears, listing several suggestions to fix the error, such as adding imports for `urllib`, `socketserver`, `http.client`, and `wsgiref.util`. At the bottom of the tooltip is a blue button labeled "Fix with Tabnine". The rest of the code is correctly indented.

```

9 def index():
10     # Check if the form has been submitted
11     if request.method == 'POST':
12         Quick Fix
13         Add "from urllib import request"
14         Add "from socketserver import _RequestType"
15         Add "from http.client import REQUEST_ENTITY_TOO_LARGE"
16         Add "from http.client import REQUEST_HEADER_FIELDS_TOO_LARGE"
17         Add "from http.client import REQUEST_TIMEOUT"
18         Add "from http.client import REQUEST_URI_TOO_LONG"
19         Add "from http.client import REQUESTED_RANGE_NOT_SATISFIABLE"
20         Add "from urllib.request import Request"
21         Add "from urllib.robotparser import RequestRate"
22         Add "from wsgiref.util import request_uri"
23         Fix with Tabnine
24             return render_template('index.html', data=data)
25

```

**Figure 6.16** Tabnine catches an error and proposes several changes.

I'll select the “Fix with Tabnine,” and it suggests changing this line of code

```
from flask import Flask, render_template
```

to

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

That way, we can import the `request` method from `Flask`. Since you're a Python developer, you've probably already spotted our next problem, where `data` isn't yet defined, in this line of code:

```
return render_template('index.html', data=data)
```

The code will return a “referenced before assignment” error with Python. So, let’s add the following line of code above it:

```
else:  
    data = None
```

This way, if a `POST` is sent, it will create a new session and populate data; if it’s a `GET` (simply loading the web page), nothing will be sent. Let’s open `index.html` and modify it to handle an empty variable.

Say we don’t know anything about Flask. How do we check for empty strings? Let’s find out.

In `index.html`, highlight the following code:

```
<ul>  
    {%- for item in data %}  
        <li>{{ item }}</li>  
    {%- endfor %}  
</ul>
```

And then, we’ll ask Tabnine for some help. Here’s the prompt I sent:

JM In the selected code, I want to check to see if data is empty. If it is, render nothing. If it’s not empty, render the selected code.

Tabnine understood my request and produced the following solution:

 {%- if data %}  
 <ul>  
 {%- for item in data %}  
 <li>{{ item }}</li>  
 {%- endfor %}  
 </ul>  
{%- endif %}

Tabnine suggests some template code that will render if data is populated and do nothing if it isn’t. Perfect. Let’s run the application again. As you can see in figure 6.17, we have what we’re expecting, which is a title and a button to create a new session.



Figure 6.17 Our screen successfully loaded with our new button.

Now that the screen is loaded up, let's click the button! It should load up our question set (figure 6.18).

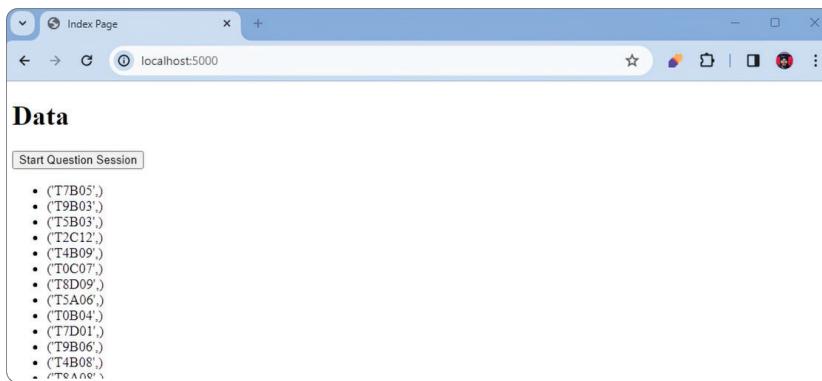


Figure 6.18 Our page loaded correctly with a new question set.

Awesome. We can now start a session. However, this session is only active if the browser window is open. We need to find a way to create a session for each user so that the application maintains state for them.

### 6.2.1 Persisting the session

We want to persist this session for a user so that if they close the browser by accident, the session will still be available. The first thing we'll do is display the session ID in the browser so we can make sure that we're working with the same session.

In `templates/index.html`, let's change the following code. We have an `h1` tag displaying "Data," and we're going to add a title and then add a display of our session ID. Let's change

```
<h1>Data</h1>
```

to look like this:

```
<h1>Ham Radio Practice Test</h1>
{%
  if session_id %
    <h3>Session: {{session_id}}</h3>
  %
  endif %
}
```

We are saying if a `session_id` exists, we'll display it on the page. If it doesn't exist yet, nothing will happen. Then we need to modify our `run.py` file to populate this variable in our template. Find this piece of code:

```
return render_template('index.html', data=data)
```

Let's change that to pass in our session ID:

```
return render_template('index.html', data=data, session_id=session.session_id)
```

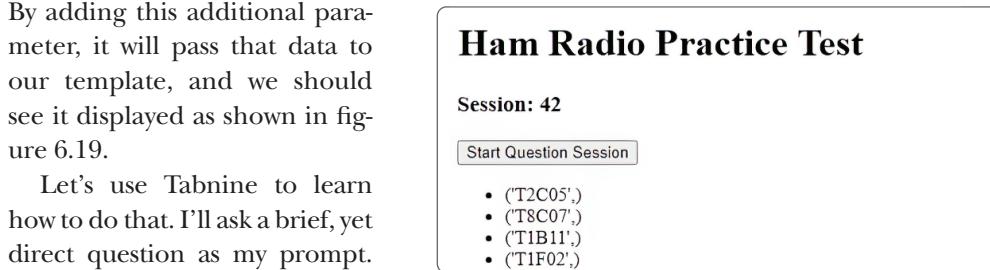
By adding this additional parameter, it will pass that data to our template, and we should see it displayed as shown in figure 6.19.

Let's use Tabnine to learn how to do that. I'll ask a brief, yet direct question as my prompt. I'm deliberately using a conversational approach here as it works well with Tabnine.

Here's my first prompt:

JM I want to store a session id in a cookie. I want flask to create this cookie based on the session id from the database. How can I build that?

Although it's basic, this prompt clearly states what I would like Tabnine to do. Tabnine gives a detailed response to the question with a full code example (figure 6.20).



The screenshot shows the Tabnine AI interface. At the top, there is a header "Ham Radio Practice Test" and a sub-header "Session: 42". Below this is a button labeled "Start Question Session". A list of four items follows: • ('T2C05,'), • ('T8C07,'), • ('T1B11,'), • ('T1F02,'). The main content area contains a user prompt from "Jeremy Morgan" asking about storing a session ID in a cookie using Flask. Tabnine responds with a detailed code example:

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

app = Flask(__name__)

@app.route('/', methods=['POST', 'GET'])
def index():
    # Check if the form has been submitted
    if request.method == 'POST':
        # Start the question session
        # ...
    else:
        # Render the index template
        return render_template('index.html', data=Questions.get_all(), session_id=session.sid)
```

**Figure 6.20** The detailed response for our prompt. Tabnine explains exactly how to create and store a cookie based on our request.

Let's follow Tabnine's advice here. It suggests that we use the `Session` object and call the `session.sid` object, then store it as a cookie. We can use the “Show diff” feature to see what needs to change (figure 6.21).

```

Change 1
-from flask import Flask, render_template, request
+from flask import Flask, render_template, request, session
+from app.models.database_connection import DatabaseConnection
+from app.models.questions import Questions
+from app.models.session import Session

Change 2
     questions = Questions(cursor)
     data = questions.get_question_set()
     session = Session(cursor)
+    session.create_session()
+    print("Session ID:", session.session_id)
+    print(data)
+    # Set the session ID as a cookie
+    session.sid = session.session_id
+
     else:
         data = None
+
     # Get the session ID from the cookie
+    session.sid = request.cookies.get('session_id')
+
     if not session.sid:
+
         # Create a new session if the cookie is not set
+        session.create_session()
+
     # Return the HTML template for the index page
     return render_template('index.html', data=data, session_id=session.session_id)

Insert Copy Show all

```

**Figure 6.21** The “Show diff” feature does a diff on the file, highlighting which lines of code we need to add to our project.

Find this section of code in `run.py`:

```

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

```

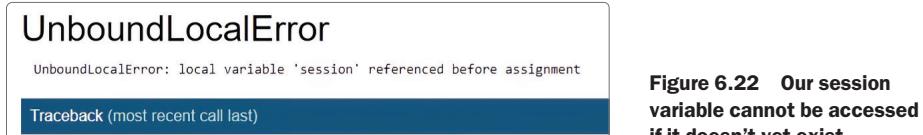
Let’s remove the print statements. We no longer need to print out the session ID in the console. After, we create the session variable. Let’s add

```

session.create_session()
# Set the session ID as a cookie
session.sid = session.session_id

```

Then, in our `else` block, we can add the code to get the session ID. If it doesn’t exist, we’ll create a new session automatically. Now that we’ve added the changes, let’s refresh our page. We see an error as shown in figure 6.22.



**Figure 6.22** Our session variable cannot be accessed if it doesn’t yet exist.

This error is another reminder that we can't blindly copy and paste code from these tools, and some bit of intervention is usually needed. We can't access session if it hasn't been created. We need to refactor this application anyway. Right now, it takes a `POST` to create a new session. It's an extra step that relies on the user clicking a button to start. We know the user wants to take a practice test, so there's no need for the extra step in creating it. Let's change the application to start a new test session when the user visits the page for the first time. Then we'll persist the session.

## 6.2.2 Refactoring session creation

At this time, we can only start a new question session by pressing a button and sending a `POST` to the application to create it. We can't access the session globally because the database connection is only made after that `POST`, as shown in our code:

```
# Check if the form has been submitted
if request.method == 'POST':
    # Start the question session
    #
    db_path = 'data/questions.db'
    # Call the function and store the returned data in a variable
    with DatabaseConnection(db_path) as cursor:
```

Let's change this so the database connection is loaded every time our index page is called. And we'll create the session when we load it up. How can we do that? We'll remove the logic for handling a `POST` and put all the creation into our index method. This is what our index method looks like now:

```
@app.route('/', methods=['POST', 'GET'])
def index():
    db_path = 'data/questions.db'
    # Call the function and store the returned data in a variable
    with DatabaseConnection(db_path) as cursor:
        session = Session(cursor)
        questions = Questions(cursor)
        data = questions.get_question_set()
        session.sid = request.cookies.get('session_id')
        return render_template('index.html', data=data, session_id=session.
    session_id)

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

Now we open the database every time and create a session. However, we need to add logic around checking the cookie for an existing session. Right now, we have the following line of code that gets our session ID; still, it needs to be modified to create a session if none is found:

```
session.sid = request.cookies.get('session_id')
```

Let's instead look for this value and create a session if it doesn't exist. Here is our updated code:

```
with DatabaseConnection(db_path) as cursor:
    session = Session(cursor)
    our_session = request.cookies.get('session_id')
    if our_session is None:
        questions = Questions(cursor)
        data = questions.get_question_set()
    session.session_id = session.create_session()
```

In the previous conversation with Tabnine, we didn't create a cookie. Let's do that and add it into a part of our session creation process. Let's ask Tabnine how to create a cookie value with Flask. Here is the prompt I used:

 JM How can I create a cookie with Flask and store our session id in a cookie. I want to do that after calling `session.session_id = session.create_session()` and store that generated id in a cookie with the value `session_id`.

Tabnine comes back with suggestion that changes how we respond to the request, so we need to remove this code:

 return render\_template('index.html',
 data=data, session\_id=session.session\_id)

It has to be replaced with code that will create a response based on our rendered template. We also need to set a cookie with the `response.set_cookie` method. Here is the new code:

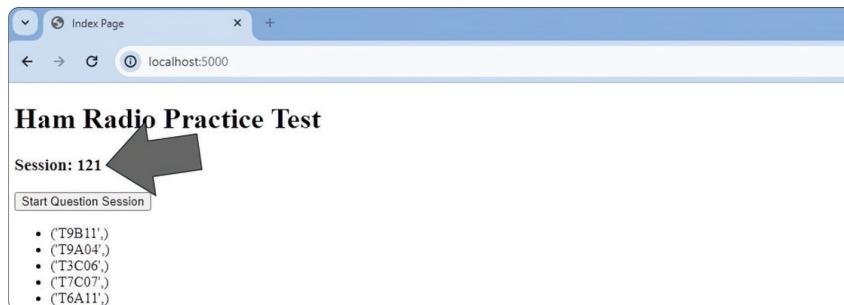
```
# Set the session ID as a cookie
response = make_response(render_template
    ('index.html', data=data, session_id=session.session_id))
response.set_cookie('session_id', str(session.session_id))
return response
```

This change will set the cookie every time the page is loaded. Now we'll create an `else` block for what to do when the cookie value exists, and we just want to render the page from the existing session:

```
else:
    # we have a cookie value and existing id
    session.session_id = our_session
    questions = Questions(cursor)
    data = questions.get_question_set()
    response = make_response(render_template('index.html', data=data,
        session_id=session.session_id))
    return response
```

Now when we run the application, every time we refresh the page, it keeps the existing value of our session. However, with each refresh, we're getting a different set of

questions in our questions set. But the ID we're displaying is the same. We know that the cookie is being set properly; however, our method is still generating more questions (figure 6.23).



**Figure 6.23 Our questions get refreshed, but the `session_id` stays the same.**

We'll need to refactor that method so it works with an ID, and instead of generating a new question set every time, it will use an existing one that we've tied with an ID.

### Verify, then trust

Always thoroughly test AI-generated code before considering it complete:

- Test edge cases, not just the happy path.
- Verify functionality after implementing each suggestion.
- Look for unintended side effects in related functionality.
- Database operations require extra scrutiny—check that database state is what you expect.
- Remember: Code that looks right can still be functionally wrong.

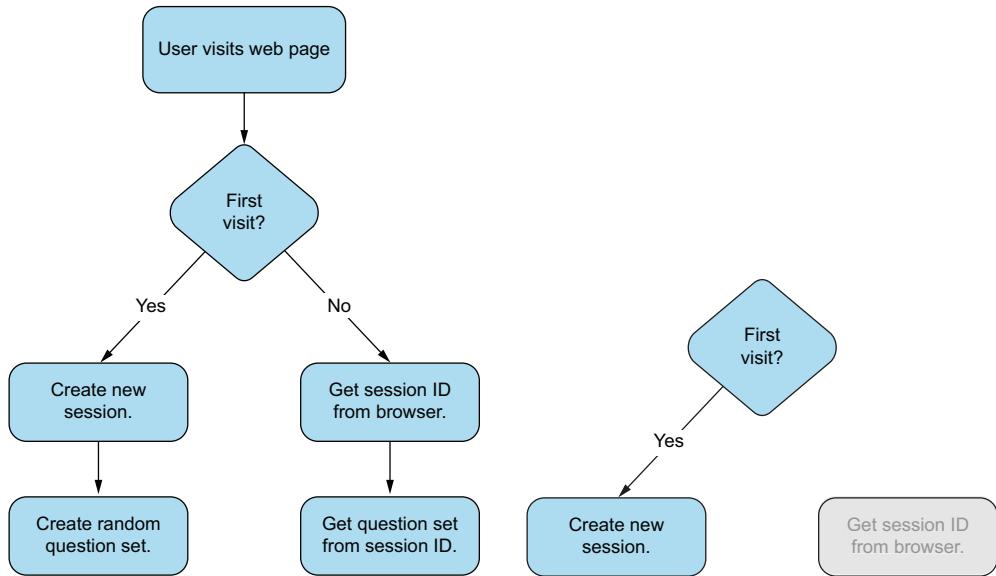
### 6.2.3 Refactoring our question set method

What exactly is happening in our application right now? The first time you load up the web page, it checks whether a session ID is stored in a cookie. This is how the application determines if it's the user's first time visiting the page. If no session ID is present, the application creates a new session and a new question set from randomly selected questions. If there is an ID present, then it continues with that existing session. Per our design specification, we should then load the question set that is associated with that session ID. This flow is shown in figure 6.24.

In our case, we're still getting a random question set every time. This is because our method looks like this right now:

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

A random question set gets drawn every time the user visits the web page, essentially making the “first visit” check pass through and create a new session every time (figure 6.25).



**Figure 6.24** The initial flow of our website. The application looks for a cookie value and handles the session accordingly.

**Figure 6.25** Because the method is being called without specifying our session ID, the first visit will always create a new session.

The method has no way of knowing whether we’ll have an ID, but this is a simple fix. We can inject the session ID into the method, and if it exists, we’ll use the existing question set instead of generating a new one.

Let’s open `questions.py` and modify our method parameters. Change

```
def get_question_set(self) :
    to
    def get_question_set(self, session_id=None) :
```

We’ll also change the logic of our method so that if a session ID is present, we’ll query that instead. Our logic currently looks like this:

```
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)

```

We'll add in our logic to check. Since we aren't changing the return type, this in-place refactoring should be easy to test. We'll change the method to include logic to test for the ID, and if the session exists, we'll pull that set of questions for it.

Our method now looks like

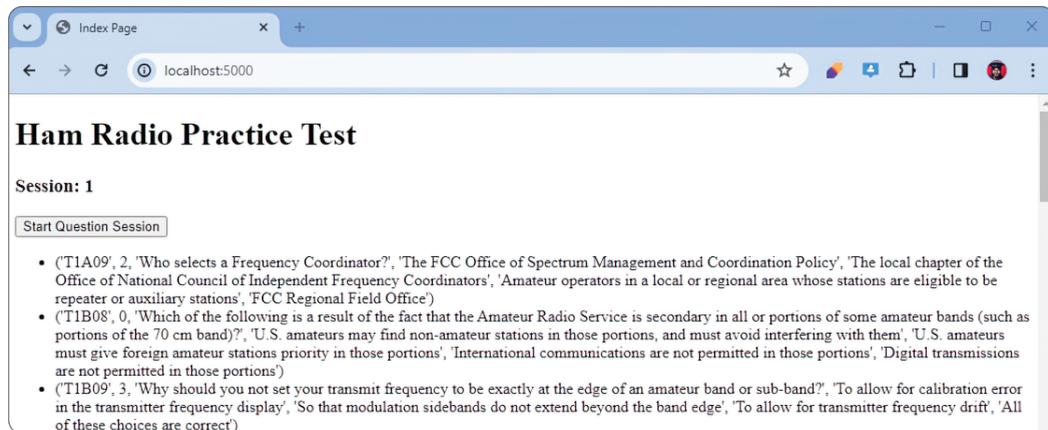
```

def get_question_set(self, session_id=None):

    if (session_id is None):
        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)
    else:
        # Get a list of questionIDs
        self.cursor.execute("""WITH question_set_info AS (
            SELECT DISTINCT question_id
            FROM question_sets
            WHERE session_id = ?
        )
        SELECT
            questions.id,
            questions.correct,
            questions.question,
            questions.a,
            questions.b,
            questions.c,
            questions.d
            FROM questions
            JOIN question_set_info
            ON questions.id = question_set_info.question_id;"""
        , (session_id,))
        result = self.cursor.fetchall()
        if result is None:
            return None
        else:
            return list(result)

```

We can now reload the browser and test the functionality. No matter how many times we refresh it, the values stay the same. I deleted the sessions table and restarted it. It now shows a single session and will not change the questions no matter how many times I refresh it (figure 6.26).



**Figure 6.26 Upon the first visit, a session is created, and no matter how many times we refresh it, the session persists, and questions stay the same.**

We refactored our method, and we're getting the functionality we were looking for in the design. Next, we need to remove the Start Question Session button because it's no longer needed. We'll also create a method to end the session.

#### 6.2.4 Creating an end session function that ends the test

Let's create a button that will quickly get rid of our cookie, in case we want to end the test and start over. I want the solution to be as simple as possible. I'm thinking a simple JavaScript method can do it. We'll use a casual, conversational prompt to check this:

JM I would like to add a button and delete the cookie and refresh the page. What's the best way to do that with JavaScript?

Here's the answer Tabnine came up with:

HTML To add a button in HTML, you can use the following code:

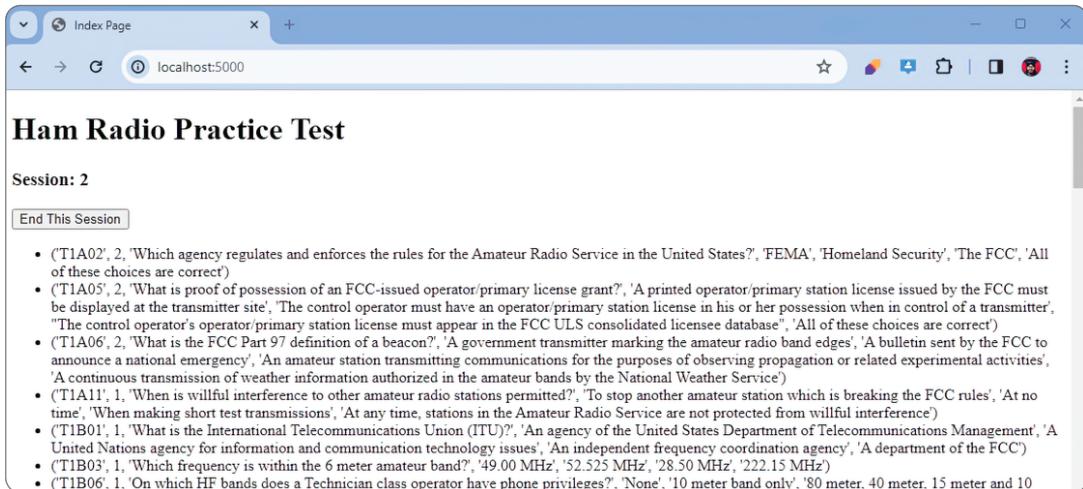
```
<button onclick="deleteCookieAndRefresh()">Delete Cookie and Refresh</button>
```

To delete the cookie and refresh the page using JavaScript, you can use the following code:

```
function deleteCookieAndRefresh() {
    // Client-side JavaScript to delete the cookie and refresh
    document.cookie = "session_id=; expires=Thu, 01 Jan 1970 00:00:00 UTC;
path=/;";
    window.location.reload();
}
```

Make sure to add the JavaScript code inside the `<script>` tags.

This suggestion is perfect, so in `index.html`, I'll add it to the code (figure 6.27). I did change the wording of the button, however, to reflect its function. When I click the button, it refreshes with a new session and new questions.



**Figure 6.27 Clicking the End This Session button refreshes the page with a new session.**

This is exactly what we intend for this application. Now comes the challenging part—taking this blob of questions and making a usable quiz from them.

### AI as an accelerator, not an autopilot

The most productive use of AI coding tools balances automation with your own expertise:

- Use AI to accelerate routine coding tasks and generate initial solutions.
- Apply your domain knowledge to evaluate whether suggestions match your architectural goals.
- Be ready to take manual control when the AI doesn't fully understand context.
- Treat AI tools as collaborative assistants rather than autonomous developers.
- Recognize that AI can help find solutions, but you're still responsible for understanding them. You can't hold AI accountable.

We built a strong backend for our app. It effectively manages random question selection and keeps user sessions active. During our development with Tabnine, we faced real coding challenges. We resolved these problems with Tabnine, showing how useful AI can be in development.

By combining AI suggestions with our development skills, we made a more reliable app that maintains state across user sessions. This is crucial for any web-based testing

platform. We learned that generative AI tools can speed up development, while still requiring critical thinking and expertise. The key point is to find the right balance. Use AI to enhance your workflow, but don't rely on it completely. As we keep building our app in the next chapters, we'll refine this approach to using AI tools in various development situations.

## Summary

- Critical evaluation of AI-generated code is key. Always use diff views to see proposed changes. Also, think about how the generated code fits your current architecture before implementing it.
- Refining your prompts iteratively boosts AI assistance quality. Start with clear descriptions. Add context when needed. Use casual language to guide the AI toward your solution.
- Implementing database session management needs careful thought. Consider constraints, auto-increment features, and cookie persistence. These keep state across user interactions.
- Testing after every implementation stage stops cascading errors. Verify functionality after each code change. Focus on database operations to ensure correct state management.
- Separation of concerns in web applications leads to cleaner architecture. For example, session creation, question retrieval, and user interface components can be separate, testable units.
- User experience considerations should guide development choices. For instance, automatically creating sessions is better than asking for user action when it fits the expected workflow.
- Error handling for edge cases makes applications more robust. Anticipate possible problems such as duplicate entries and set up fallback mechanisms.
- Using AI as a collaborative assistant boosts productivity. Pairing the AI's code generation with your domain knowledge gives the best results.