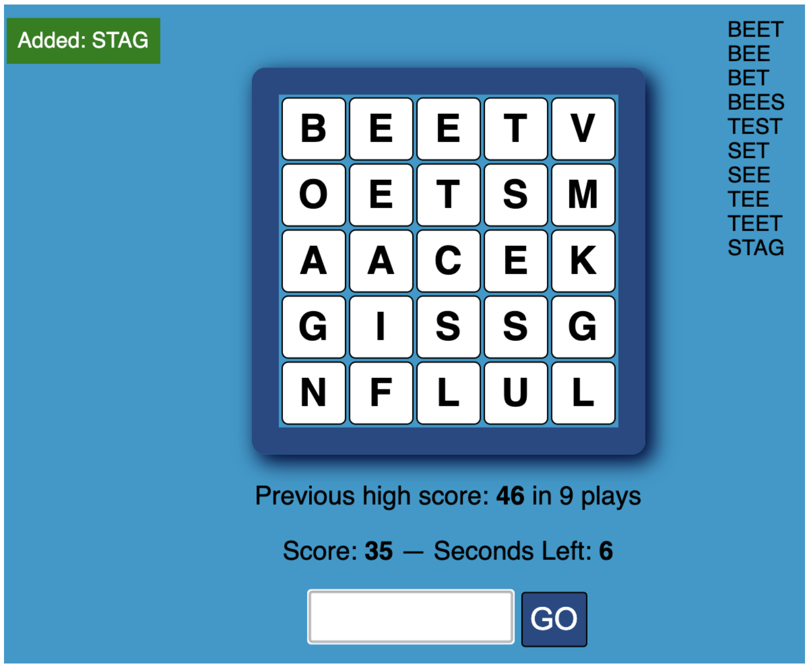
Flask Boggle

Download starter code <../code.zip>

In this exercise, you'll plan, code, and test a Boggle game with a Flask backend and a JavaScript frontend, while practicing testing as you go.



The Game

The goal of the game is to get the highest point total. To gain points, players create words from a random assortment of letters in a 5x5 grid. We will be providing the functionality to generate the grid.

Words can be created from a chain adjacent letters — letters which are horizontal, vertical, or diagonal neighbors of each other. Each tile on the board may only be used once in building a word. We providing code for the logic for this part.

IMPORTANT **Writing Tests**

The focus of this exercise is on testing Flask. At each step, you'll write tests. Don't move forward until you've written tests and they pass!

Step One: Planning and Reading Code

When approaching a new codebase, it's usually best to investigate the classes first (since the rest of the app tends to depend on them).

We've provided two class:

- *WordList* in `wordlist.py`: a simple class for reading a list of words from a file, and providing an API for checking if a word is in the list.
- *BoggleGame*: in `boggle.py`: a more complex class to manage the state of a Boggle game (keeps track of the board, the letters played, the score, and more).

Learn what these classes do and how they work. A good way to feel out a codebase is trying it out in a console:

```
>>> import wordlist
>>> help(wordlist.WordList)    #  *(press 'q' to quit that)*
>>> import boggle
>>> help(boggle.BoggleGame)
```

Reading the docstrings and doctests in those files should give you a good sense of how to use them — you may not even need to read the code.

You can make instances of these classes and experiment with the API for them:

```
>>> game = boggle.BoggleGame()
>>> game.board
<BoggleGame board=HEYXZ.ERSTA.DFSTE.DFDSF.EAGST played_words=set()>

>>> game.check_word_on_board('HEY')
True

>>> game.play_and_score_word('CAT')
1
>>> game.play_and_score_word('HIPPO')
2
>>> game.score
3
```

Step Two: Running and Adding to The Doctests

When you looked at the help for these files, you probably noticed there were doctests. Run them:

```
$ python3 -m doctest -v wordlist.py
$ python3 -m doctest -v boggle.py
```

Add a doctest for the *WordList.check_word* method.

Step Three: Flask and Displaying the Board

Before starting the Flask part, create a virtual environment, activate it, and install the requirements.

Take a look at the `app.py` file. We've provided two routes:

- *homepage()*: returns HTML page where a game can be played.
- *new_game()*: creates a *BoggleGame* instance, and stores it in the global dictionary *games* by a unique, random id (called a *UUID*) (this way, additional routes can find it). It is incomplete (you'll finish it in a later step).

Run the initial integration tests for the Flask app:

```
(venv) $ python3 -m unittest test_app.py
```

Both should pass—but neither test really checks for anything yet.

For the homepage, add to the test to check that correct HTML was returned. You don't to establish that the entire HTML returned matches the entire thing in your test. Instead, find something distinct that appears in this template and check for that.

ATTENTION **Stop and get a code review**

Step Four: Making an API and Testing It

In this step, you'll build a JSON-returning API route for the app. You'll use this later to build a JS front end.

New Game

The */api-new-game* route isn't complete yet; it doesn't return what the docstring says it should. Complete the route and write a test for it.

HINT **Testing Random Things**

This can be tricky — since both the game id and the board are random, you can't test these in your test. But you can test that:

- the route returns JSON with a string game id, and a list-of-lists for the board
- the route stores the new game in the *games* dictionary

Score Word

Make a new route with a path of */api/score-word*. This should accept a POST request with JSON for the game id and the word. It should check if the word is legal:

- It should be in the word list
- It sound be findable on the board

This route should return a JSON response using Flask's *jsonify* function.

- if not a word: `{result: "not-word"}`
- if not on board: `{result: "not-on-board"}`
- if a valid word: `{result: "ok"}`

A good way to try this out is in your web browser console. As it turns out, we already put into the HTML template for the page to load axios. So, you can try out your new route using AJAX, like:

```
await axios.post("/api/score-word", data=...)
```

Alternatively, you could test this in Insomnia.

Write A Test For It

Then, **write an integration test** in the `test_app.py` file for this route.

Your test function will need to use the */api/new-game* route, since that makes a new game and returns the game id.

One challenge is that the board is randomly chosen, so writing a test to check if a word is on the board would be tricky—unless you can figure out a way to change what the letters are on the board before you try to score the play.

Test all the possible results for */score-word*.

ATTENTION **Stop and get a code review**

Step Five: Writing a Front End

Congratulations! You've got a working, tested app with an API.

In this step, you'll build a front end for the app, using jQuery and axios.

- Finish the JavaScript code for putting the board in the DOM
- Add an event handler for submits on the form
- Send the words to the back end and receive the results
- If not a legal play, display a message in the DOM
- If a legal play, add this word in a bulleted list

You shouldn't need to make any changes to the back end to do this.

Be organized here. You'll add more functionality to the front end after this, so take your time and structure and document this professionally.

ATTENTION **Stop and get a code review**

Step Six: Celebrate!

Congratulations! You've got a working, tested app with integrated front and back ends. Play a game with your pair and thank them for their help.

Further Study

Score Words and Keep Track of Score

Add a feature that keeps track of scores. The *BoggleGame* API already returns the word score each time your submit a word. Add that score and the game score to the JSON returned.

Update the front end so that it when it gets the scores back from the back end, it will update the DOM to show the running game score.

Write and fix tests:

- Update the Flask integration test so that it tests scoring.

Don't Allow Duplicate Words

Right now, a user could win a game by finding one valid word on the board and repeatedly submit that. Oops.

The *BoggleGame* API already keeps track of the words scored so far, and has a helpful method that tells you if a word is duplicate or not.

When a new word is submitted, if the word has already been played, return a new JSON result indicating this is a duplicate.

Update your integration to demonstrate this works.

Update the front end so that if the API returns a duplicate-word result, it should not add that word to the DOM list of words, and should display an error message.

Make Game Timed

Boggle is normally a timed game, where you find as many words as you can in a limited amount of time. Show the amount of time left in the DOM and update that every second. When the timer reaches zero, display a message in the DOM and prevent the player from continuing to submit words.

This step should not involve the backend, so all that code and tests should be good.

Keeping Track of High Score and Number of Plays

We'd like to keep track of:

- Highest score ever for this user
- Number of times this user has finished a game

Update the score every time a word is scored.

Add a new route, */api/end-game*, which the front end should call when the game is over. This should update the high score (if the previous record was broken), update the number of plays, and should return JSON like:: `{score, numPlays, highScore}`.

Write backend tests for this.

Update the front end game-ending function so that this is called, change the end game message to include the score, high score, and number of plays.

Even Further Study

This is a good exercise for continuing to play to try things out.

- Does your game ensure that if users enter a lower-cased word, everything still works and this is considered and listed the same way as the upper-cased word? Write tests to find out, and update the code.
- Improve our CSS. Make this something you'd be proud to show.
- Let users pick a board size when starting a game.
- Make your front-end code object oriented. This will help improve the code and remove the need for global variables. Plus, if you do this well, you could make it possible for the app to show *several* ongoing games at once!
- Your backend has lots of tests — now you could write some for the frontend. Using Jasmine, you can test the UI. This will require some reading and research, but would be an excellent thing to put on your learning list.
- Try to implement a hint feature! Add a button to ask for a hint, which will then show the first couple of characters in a valid word that the user hasn't discovered yet.

Solution

- Flask Boggle Solution
  - Main Solution
  - Further Study Solution