AI Code Generation Code-along

(https://github.com/learn-co-curriculum/phase-2-ai-code-generation-code-along) (https://github.com/learn-co-curriculum/phase-2-ai-cod
generation-code-along/issues/ne

Learning Goals

- Use JSDoc to write function documentation.
- Install the Tabnine extension and create an account.
- Use an AI assistant to generate tests and code.

Introduction

AI models are heavily dependent on their input data. In order to get effective output, we must provide the proper context and information. In this lesson, we will be discussing the following topics:

- Data types review.
- JSDoc.
- Tabnine code generation.

Data Types in JavaScript Review

Recall that JavaScript is a dynamically typed language, i.e., the variables in JS do not contain any information about the type of data they reference. We can even reassign a different value to an existing variable.

```
// the num variable holds a number type value
let num = 2;
// assigned a string type value
num = "2";
```

In statically typed languages like Java, variable types are declared when they are first initialized. This allows various coding tools to analyze the program better and provide improved suggestions or autocompletion. AI tools can also benefit from having more data type information.

Since JavaScript doesn't provide a way to declare variable types we have to use other tools. One such tool is called JSDoc. Let's explore what this tool is and how we can leverage it to improve AI code generation. We will be looking at how to document functions using JSDoc.

JSDoc Overview

JSDoc (https://jsdoc.app/index.html) is a tool for documenting JavaScript code. It allows us to define the data type for various cases such as function parameters, function arguments, and object properties.

The basic JSDoc template for a function looks like this:

```
/** description of the function
 * @param {type} paramOne - parameter description (optional)
 * @param {type} paramTwo - parameter description (optional)
 * @returns {type}        - return value description (optional)
 */
```

Let's look at what each part does in detail:

- The first line always starts with `/**` and describes what the function should do.
- Next, we define all the parameters along with their data types.
    - The `@param` is called a **block tag** (https://jsdoc.app/about-block-inline-tags.html) in JSDoc. They provide detailed information about tl code.
    - The `type` can contain primitive JS types such as `number` or custom types (discussed later in this lesson).
- And finally, we use the `@returns` block tag to define what type of data the function is expected to return.

Here's what the JSDoc would look like for a function that takes in two numbers and returns their sum:

```
/** adds two numbers
 * @param {number} num1
 * @param {number} num2
```

```
  * @returns {number} - The sum of the two numbers.
  */
function add(num1, num2) {
  return num1 + num2;
}
```

You can use the following command to generate the JSDoc documentation page in the `out` directory:

```
jsdoc index.js
```

We will explore a few other JSDoc features later in this lesson. But before that, let’s set up the AI assistant.

Tabnine Set Up

Tabnine is a AI assistant tool that can be used in various code editors and IDEs. They have trained their own model and provides a free plan.

Follow the instructions on **Tabnine’s official page** (https://www.tabnine.com/install/vscode) to install it in VSCode. Once it’s installed, it shoul automatically make AI powered suggestions.

If it asks you to sign up, use your GitHub account to sign up for an account. It will be automatically linked to the VSCode extension.

Codealong

Fork and clone this repo to follow along with the code generation. The repo has the test environment and the JSDoc dependency installed.

Run the following command after cloning the repo:

```
npm install
```

Here’s the problem statement we will be solving:

“Create a function that takes in a list of books and a genre. It should return a new list of books with the specified genre.”

This is an intentionally vague statement so we can explore what additional information we should be aware of when designing the function.

We will modify the following files:

- `index.js` - This is where we will be writing the function code.
- `test/index.test.js` - This is where we will be writing the tests for the function in `index.js` .

We will be following these steps:

1. Define the book structure using JSDoc in `index.js` .
2. Define the function structure using JSDoc in `index.js` .
3. Use the AI assistant to generate tests for the function in `test/index.test.js` .
4. Use the AI assistant to generate the function definition and body in `index.js` .

Define Book Structure

AI suggestions work best when data type and structure information is defined. Since the book object is not defined, we will be creating a simple structure with three fields:

- `title` .
- `author` .
- `genre` .

When defining objects with JSDoc, we have to use the `@typedef` and `@property` block tags. The `typedef` tag defines the data type of the entity we're creating and a `property` tag defines an object key name and its value data type.

Add the following JSDoc definition in your `index.js` file:

```
/** A book object representation.
 * @typedef {Object} Book
 * @property {string} title - The title of the book.
 * @property {string} author - The author of the book.
 * @property {string} genre - The genre of the book.
 */
```

- The `typedef` block tag is defining a `Book` entity with the data type `Object` .

- The `property` block tags are defining the key names and the data type as `string` .

Define the Function Structure

We will use the standard `@params` and `@returns` block tags.

Add the following JSDoc definition and the function signature in your `index.js` file:

```
/** returns a list of books with the given genre.
 * @param {string} genre - The genre of books required.
 * @param {Book[]} books - An array of Book objects.
 * @returns {Book[]} An array of Book objects matching the specified genre.
 */
function filterByGenre(genre, books) {}
```

Notice that we are using `Book[]` as the data type for the `books` param and the return value. A couple things to note here:

1. We can use user-defined data types such as `Book` in other JSDoc definitions.
2. We can use `[]` after any data type to signify that it's an array containing elements of the given data type. In this case, both books and the return value is an array of `Book` objects.

At this point, the AI assistant may try to generate the function body. But you should generally write tests before writing the body of your function

Generate Tests Using AI

There are two ways we can generate the tests for the `filterByGenre` function:

1. Use the AI assistant.
2. Use a chatbot like ChatGPT.

For the first method, you can start the test generation by going to the `test/index.test.js` file and writing the following:

```
// generate tests for the filterByGenre function
describe(...
```

As soon as you type in `describe` it should start suggesting tests. The problem with this method is that the free version of AI assistants don't allow multi-line suggestions or autocompletion.

Note that the `filterByGenre` function is already being imported because we've set up the test environment for you. In your own projects, you would have to export the functions from your main code files (`index.js` in our case) and then import them in your test files.

Let's look at the second method where we use a chatbot to generate the tests. Go to **chat.openai.com** (https://chat.openai.com/) and provid the following prompt:

```
Suggest Chai tests for the following JSDoc function specification:

/** A book object representation.
 * @typedef {Object} Book
 * @property {string} title - The title of the book.
 * @property {string} author - The author of the book.
 * @property {string} genre - The genre of the book.
 */

/** returns a list of books with the given genre.
 * @param {string} genre - The genre of books required.
 * @param {Book[]} books - An array of Book objects.
 * @returns {Book[]} An array of Book objects matching the specified genre.
 */
```

This should provide an output similar to the following:

```
describe("filterByGenre", () => {
  it("should return an empty array if no books match the genre", () => {
    const genre = "Thriller";
    const books = [
      { title: "Book 1", author: "Author 1", genre: "Fantasy" },
      { title: "Book 2", author: "Author 2", genre: "Science Fiction" },
    ];
    const result = filterByGenre(genre, books);
    expect(result).to.be.an("array").that.is.empty;
  });

  it("should return an array of books matching the genre", () => {
    const genre = "Fantasy";
    const books = [
      { title: "Book 1", author: "Author 1", genre: "Fantasy" },
      { title: "Book 2", author: "Author 2", genre: "Science Fiction" },
      { title: "Book 3", author: "Author 3", genre: "Fantasy" },
    ];
    const result = filterByGenre(genre, books);
    expect(result).to.deep.equal([
      { title: "Book 1", author: "Author 1", genre: "Fantasy" },
      { title: "Book 3", author: "Author 3", genre: "Fantasy" },
    ]);
  });

  it("should handle an empty array of books", () => {
    const genre = "Fantasy";
    const books = [];
    const result = filterByGenre(genre, books);
    expect(result).to.be.an("array").that.is.empty;
  });

  it("should return a new array without modifying the original array", () => {
    const genre = "Fantasy";
    const books = [
      { title: "Book 1", author: "Author 1", genre: "Fantasy" },
      { title: "Book 2", author: "Author 2", genre: "Science Fiction" },
      { title: "Book 3", author: "Author 3", genre: "Fantasy" },
    ];
    const result = filterByGenre(genre, books);
```

```
      expect(result).to.deep.equal([
        { title: "Book 1", author: "Author 1", genre: "Fantasy" },
        { title: "Book 3", author: "Author 3", genre: "Fantasy" },
      ]);
      expect(result).to.not.equal(books);
    });
  });
```

These tests covers the following scenarios:

- No books match the provided genre.
- The genre matches a book in the list.
- The book list input is empty.
- The book list input is not modified.

For complex functions, AI assistants and chatbots may miss edge cases. You should always double and triple check the generated test scenarios.

Note that **Chai** (https://www.chaijs.com/) is the assertion library we are using. This is already set up and imported in the test file. If you use another library like Jest or Vitest, substitute the library name in the chatbot prompt.

Now you can add the generated tests in the `test/index.test.js` file under the comment `Your tests here` . Run the following command afterwards:

```
npm test
```

You should see failing tests in the console.

The final `test/index.test.js` file should look like this:

```
require("./helpers.js");

const { expect } = require("chai");
```

```
// Your tests here
describe("filterByGenre", () => {
  it("should return an empty array if no books match the genre", () => {
    const genre = "Thriller";
    const books = [
      { title: "Book 1", author: "Author 1", genre: "Fantasy" },
      { title: "Book 2", author: "Author 2", genre: "Science Fiction" },
    ];
    const result = filterByGenre(genre, books);
    expect(result).to.be.an("array").that.is.empty;
  });

  it("should return an array of books matching the genre", () => {
    const genre = "Fantasy";
    const books = [
      { title: "Book 1", author: "Author 1", genre: "Fantasy" },
      { title: "Book 2", author: "Author 2", genre: "Science Fiction" },
      { title: "Book 3", author: "Author 3", genre: "Fantasy" },
    ];
    const result = filterByGenre(genre, books);
    expect(result).to.deep.equal([
      { title: "Book 1", author: "Author 1", genre: "Fantasy" },
      { title: "Book 3", author: "Author 3", genre: "Fantasy" },
    ]);
  });

  it("should handle an empty array of books", () => {
    const genre = "Fantasy";
    const books = [];
    const result = filterByGenre(genre, books);
    expect(result).to.be.an("array").that.is.empty;
  });
  it("should return a new array without modifying the original array", () => {
    const genre = "Fantasy";
    const books = [
      { title: "Book 1", author: "Author 1", genre: "Fantasy" },
      { title: "Book 2", author: "Author 2", genre: "Science Fiction" },
      { title: "Book 3", author: "Author 3", genre: "Fantasy" },
    ];
    const result = filterByGenre(genre, books);
    expect(result).to.deep.equal([
      { title: "Book 1", author: "Author 1", genre: "Fantasy" },
      { title: "Book 3", author: "Author 3", genre: "Fantasy" },
    ]);
    expect(result).to.not.equal(books);
  });
});
```

Define the Function Body

Go back to the `index.js` file and start typing the following in the `filterByGenre` function body:

```
return ...
```

The AI assistant should suggest the entire function body immediately. Press `tab` to accept the suggestion.

If you run `npm test` now, all of the tests should pass!

Conclusion

We have learned how to document our code and use that specification to quickly generate code using AI assistants. Although this is very convenient, you should exercise caution when generating complex code with AI.

AI assisted code suggestion is primarily useful for generating boilerplate code or pure functions with clearly defined inputs, outputs, and constraints.