# Hands-on Lab: Software Documentation Using Generative AI

**Estimated time needed:** 30 minutes

As a software developer, you may be very comfortable with coding. However, coding skills alone do not suffice in the real world. The code you create is more often than not used and maintained by many people. To ensure that your code is readable and understandable, you need to include software documentation. While this activity is of pivotal importance, it is time-consuming. You can avail the help of Generative AI to generate the documentation.

Please remember the prompts that you feed to Generative AI are like a conversation with a subject matter expert and the consecutive prompts depend on the previous prompts used and the response received. Change the wording if required to get a specific desired result. The example showcases one possible chat conversation to attain the objective.

## Learning objectives

At the end of the lab, you will be able to:

1. Use Generative AI to generate documentation for the code you have written.
2. Learn how documentation is written for JavaScript.

## Prerequisites

It would be good if you know how to code in JavaScript, as this lab is based on creating documentation for JavaScript code.

# Set up chat title and prompt instruction

When the chat session opens, a new chat conversation begins. Give the chat conversation an appropriate title. This will help you revisit the chat conversation. It is a good practice to segregate the conversations topically as this will help you continue the conversation at a later point.

Also provide a prompt instruction which is specific to the conversation in this particular lab. Let's get started with the task of creating software documentation for the JavaScript code.

> Please note Generative AI is an evolving field. As you attempt the labs, your experience and output might be different than what is seen here.


# Tool capability to translate

You are already comfortable coding in JavaScript. You have created the code in JavaScript. You now need to include documentation in the code. You decide to optimally utilize your time by using Gen AI to write the documentation.

Firstly, you need to determine if the Gen AI tool has the capability to do the required software documentation for JavaScript. In the prompt type:

```
I have code written in JavaScript. Can you help me add documentation to the code?
```

If the GenAI model has the ability to add software documentation to the code, you will get an affirmative response in line with what is shown in the image below. JavaScript is one the most commonly used languages and this task of creating software documentation for JavaScript code will be doable. But if there are other newer or scarcely used languages, you will need to use your judgment to perceive the confidence in the response. Based on these responses you can decide on continuing to use the tool either entirely or as a support tool. Iterative prompts can be used to ensure the correctness of the responses.

▼ Click here to view the sample response generated

Absolutely! To create effective software documentation for your JavaScript code, you can follow a standard format that includes several key components. Below is a general outline you can use, along with an example to illustrate how to document a simple JavaScript function.

Documentation Outline

1. **Title**: Name of the function or module.
2. **Description**: A brief overview of what the code does.
3. **Parameters**: A list of parameters the function accepts, including their types and descriptions.
4. **Returns**: What the function returns, including the type.
5. **Examples**: Sample usage of the function.
6. **Notes**: Any additional information, such as exceptions thrown or important considerations.

Example JavaScript Code

Here's a simple JavaScript function that adds two numbers:

```
function addNumbers(a, b) {
    return a + b;
}
```

Documentation for the Code

```
/**
 * @function addNumbers
 * @description Adds two numbers together and returns the result.
 * @param {number} a - The first number to add.
 * @param {number} b - The second number to add.
 * @returns {number} The sum of the two numbers.
 * @example
 * // returns 5
 * addNumbers(2, 3);
 *
 * @example
 * // returns 10
 * addNumbers(4, 6);
 *
 * @notes
 * Make sure both parameters are of type number to avoid unexpected results.
 */
function addNumbers(a, b) {
    return a + b;
}
```

Additional Tips

- **Consistency**: Use a consistent style for your documentation throughout your codebase.
- **Tools**: Consider using tools like JSDoc to automatically generate documentation from your comments.
- **Clarity**: Write clear and concise descriptions to ensure that anyone reading the documentation can quickly understand the purpose and usage of the code.

Feel free to share your specific code, and I can help you document it accordingly!

# Provide the code to Gen AI

Paste the following request along with the code in the prompt and send it. This process will take a few seconds as it needs to process the code and return it with the documentation.

```
Add documentation to the following code.
const express = require('express');
const axios = require('axios');
const app = express();
const port = 3000;
app.get('/weather/:city', async (req, res) => {
  try {
    const city = req.params.city;
    const weatherData = await getWeatherData(city);
    res.json(weatherData);
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch weather data' });
  }
});
async function getWeatherData(city) {
  const apiKey = 'YOUR_API_KEY';
  const apiUrl = `https://api.weatherapi.com/v1/current.json?key=${apiKey}&q=${city}`;
  const response = await axios.get(apiUrl);
  const weatherData = response.data;
  return {
    city: weatherData.location.name,
    temperature: weatherData.current.temp_c,
    condition: weatherData.current.condition.text,
  };
}
app.listen(port, () => {
  console.log(`Server is running on [http://localhost:$%7bport%7d%60]http://localhost:${port}`);
});
```

It will produce an output in JavaScript code with documentation similar to what is shown below.

▼ Click here to view the sample output generated

Here's how you can add documentation to the provided JavaScript code using a standard format. This includes descriptions for the main server setup, the route handler, and the function that fetches weather data.

Documentation for the Code

```
const express = require('express');
const axios = require('axios');
const app = express();
const port = 3000;
/**
 * @function getWeatherData
 * @description Fetches current weather data for a given city from the weather API.
 * @param {string} city - The name of the city for which to fetch weather data.
 * @returns {Promise<Object>} A promise that resolves to an object containing the city name, temperature, and weather condition.
 * @throws {Error} Throws an error if the API request fails.
 * @example
 * // returns an object with weather data for London
 * getWeatherData('London').then(data => console.log(data));
 */
async function getWeatherData(city) {
  const apiKey = 'YOUR_API_KEY'; // Replace with your actual API key
  const apiUrl = `https://api.weatherapi.com/v1/current.json?key=${apiKey}&q=${city}`;
  const response = await axios.get(apiUrl);
```

```
    const weatherData = response.data;
    return {
      city: weatherData.location.name,
      temperature: weatherData.current.temp_c,
      condition: weatherData.current.condition.text,
    };
  }
  /**
   * @route GET /weather/:city
   * @description Handles GET requests to fetch weather data for a specified city.
   * @param {string} req.params.city - The name of the city provided in the URL.
   * @returns {Object} A JSON response containing the weather data for the specified city or an error message.
   * @example
   * // Example request: GET /weather/London
   * // returns JSON response with weather data for London
   */
  app.get('/weather/:city', async (req, res) => {
    try {
      const city = req.params.city;
      const weatherData = await getWeatherData(city);
      res.json(weatherData);
    } catch (error) {
      res.status(500).json({ error: 'Failed to fetch weather data' });
    }
  });
  /**
   * @function listen
   * @description Starts the Express server and listens for incoming requests.
   * @param {number} port - The port number on which the server listens.
   * @example
   * // Starts the server on port 3000
   * app.listen(port, () => {
   *   console.log(`Server is running on http://localhost:${port}`);
   * });
   */
  app.listen(port, () => {
    console.log(`Server is running on http://localhost:${port}`);
  });
```

Additional Tips

- **API Key**: Remember to replace `'YOUR_API_KEY'` with your actual API key for the weather API.
- **Error Handling**: The error handling in the route is basic; consider logging the error for better debugging.
- **Environment Variables**: For security, consider using environment variables to store sensitive information like API keys instead of hardcoding them.

This documentation will help users understand the purpose and usage of your code, making it easier to maintain and extend in the future. If you have any more specific requests or additional code to document, feel free to share!

> This is assuming that the JavaScript code has been typed properly. If there are minor coding mistakes, Gen AI has the abilitiy to correct them and give the right output.

The documentation may not cover all the aspects you intended to cover. You may need to follow up with specific prompts.

# Variable names casing format

Particular kinds of casing formats are recommended as good coding practice. Camel case has been used in the code here. In camel case, the variable name starts with lowercase and every consecutive word starts with uppercase. For instance, `studentNumber`. Type the following prompt to find which case is better?

```
Let me know which case is better for variable names: camel case or snake case?
```

The output would be similar to what is shown below. You can iteratively prompt for more specific questions you may have.

▼ Click here to view the sample response generated

In JavaScript, the preferred naming convention for variable names is **camel case**. This convention involves writing the first word in lowercase and capitalizing the first letter of each subsequent word, without spaces or underscores. For example:

- `myVariableName`
- `getUserData`

Reasons to Use Camel Case in JavaScript:

1. **Consistency with JavaScript Standards**: Camel case is widely used in JavaScript, especially for variable names, function names, and object properties. Following this convention keeps your code consistent with the broader JavaScript community and libraries.
2. **Readability**: Camel case improves readability by clearly distinguishing between words without the use of underscores or spaces.
3. **Framework Conventions**: Many popular JavaScript frameworks and libraries (like React, Angular, and Vue) adopt camel case for naming components, functions, and variables, making it easier to integrate and collaborate with these tools.

Comparison with Snake Case

- **Snake Case**: This convention uses lowercase letters and underscores to separate words (e.g., `my_variable_name`). While it is common in other programming languages like Python, it is less common in JavaScript.

Summary

- **Use Camel Case** for variable names in JavaScript to align with community standards and improve readability.
- **Use Snake Case** in scenarios where you're working with data formats (like JSON) that may require it, or when integrating with systems that use snake case.

Overall, sticking with camel case for JavaScript variables and functions is the best practice. If you're working in a team or on a project, it's also a good idea to establish and adhere to a consistent naming convention throughout the codebase.

# Change the case

Assume your organization wants to standardize snake case for readability. In the code we have here, there are not too many lines of code. In real-life scenarios, it is rarely the case. Changing the case manually can be very challenging and possibly error-prone. You can use Gen AI to do the same for you. Type the following prompt:

```
Change the case of the variable names to snake case.
```

The output would be similar to what is given below. Notice that all the variable names are retained with just the case changed.

▼ Click here to view the sample response generated

Sure! Below is your JavaScript code with the variable names changed to **snake case**. I've updated the variable names accordingly while keeping the functionality intact.

Updated JavaScript Code with Snake Case Variable Names

```
const express = require('express');
const axios = require('axios');
const app = express();
const port = 3000;
/**
 * @function get_weather_data
 * @description Fetches current weather data for a given city from the weather API.
 * @param {string} city - The name of the city for which to fetch weather data.
 * @returns {Promise<Object>} A promise that resolves to an object containing the city name, temperature, and weather condition.
 * @throws {Error} Throws an error if the API request fails.
 * @example
 * // returns an object with weather data for London
 * get_weather_data('London').then(data => console.log(data));
 */
async function get_weather_data(city) {
  const api_key = 'YOUR_API_KEY'; // Replace with your actual API key
  const api_url = `https://api.weatherapi.com/v1/current.json?key=${api_key}&q=${city}`;
  const response = await axios.get(api_url);
  const weather_data = response.data;
  return {
    city: weather_data.location.name,
    temperature: weather_data.current.temp_c,
    condition: weather_data.current.condition.text,
  };
}
/**
 * @route GET /weather/:city
 * @description Handles GET requests to fetch weather data for a specified city.
 * @param {string} req.params.city - The name of the city provided in the URL.
 * @returns {Object} A JSON response containing the weather data for the specified city or an error message.
 * @example
 * // Example request: GET /weather/London
 * // returns JSON response with weather data for London
 */
app.get('/weather/:city', async (req, res) => {
  try {
    const city = req.params.city;
    const weather_data = await get_weather_data(city);
    res.json(weather_data);
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch weather data' });
  }
});
/**
 * @function listen
 * @description Starts the Express server and listens for incoming requests.
 * @param {number} port - The port number on which the server listens.
 * @example
 * // Starts the server on port 3000
 * app.listen(port, () => {
 *   console.log(`Server is running on http://localhost:${port}`);
 * });
 */
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

Changes Made:

- Changed `getWeatherData` to `get_weather_data`.
- Changed `apiKey` to `api_key`.
- Changed `apiUrl` to `api_url`.
- Changed `weatherData` to `weather_data`.

Note:
While this code now uses snake case for variable names, it's worth noting that using snake case in JavaScript is less common than camel case. If you are working in an environment or with a team that prefers snake case, this change is appropriate. Otherwise, consider maintaining camel case for consistency with JavaScript conventions. If you have any more requests or need further modifications, feel free to ask!

# Including comments

Comments are extremely important for code readability. It is always good coding practice to include comments explaining what the code does. Type the next prompt as:

```
Include comments in the code which will help in readability of the code
```

▼ Click here to view the sample response generated

Certainly! Below is the updated JavaScript code with added comments to enhance readability. These comments explain the purpose of various sections and lines of code, making it easier to understand the flow and functionality.

JavaScript Code with Comments for Readability

```
// Importing required modules
const express = require('express'); // Express framework for building web applications
const axios = require('axios'); // Axios for making HTTP requests
const app = express(); // Creating an instance of an Express application
const port = 3000; // Defining the port number for the server
/**
 * @function get_weather_data
 * @description Fetches current weather data for a given city from the weather API.
 * @param {string} city - The name of the city for which to fetch weather data.
 * @returns {Promise<Object>} A promise that resolves to an object containing the city name, temperature, and weather condition.
 * @throws {Error} Throws an error if the API request fails.
 * @example
 * // returns an object with weather data for London
 * get_weather_data('London').then(data => console.log(data));
 */
async function get_weather_data(city) {
  const api_key = 'YOUR_API_KEY'; // Replace with your actual API key
  // Constructing the API URL using the city name and API key
  const api_url = `https://api.weatherapi.com/v1/current.json?key=${api_key}&q=${city}`;
  // Making a GET request to the weather API
  const response = await axios.get(api_url);
  const weather_data = response.data; // Extracting the data from the response
  // Returning an object with relevant weather information
  return {
    city: weather_data.location.name, // City name
    temperature: weather_data.current.temp_c, // Current temperature in Celsius
    condition: weather_data.current.condition.text, // Weather condition description
  };
}
/**
 * @route GET /weather/:city
 * @description Handles GET requests to fetch weather data for a specified city.
 * @param {string} req.params.city - The name of the city provided in the URL.
 * @returns {Object} A JSON response containing the weather data for the specified city or an error message.
 * @example
 * // Example request: GET /weather/London
 * // returns JSON response with weather data for London
 */
app.get('/weather/:city', async (req, res) => {
  try {
    const city = req.params.city; // Extracting the city name from the request parameters
    const weather_data = await get_weather_data(city); // Fetching weather data for the city
    res.json(weather_data); // Sending the weather data as a JSON response
  } catch (error) {
    // Handling errors and sending a 500 status with an error message
    res.status(500).json({ error: 'Failed to fetch weather data' });
  }
});
/**
 * @function listen
 * @description Starts the Express server and listens for incoming requests.
 * @param {number} port - The port number on which the server listens.
 * @example
 * // Starts the server on port 3000
 * app.listen(port, () => {
 *   console.log(`Server is running on http://localhost:${port}`);
 * });
 */
app.listen(port, () => {
  // Logging a message to indicate that the server is running
  console.log(`Server is running on http://localhost:${port}`);
});
```

Key Comments Added:

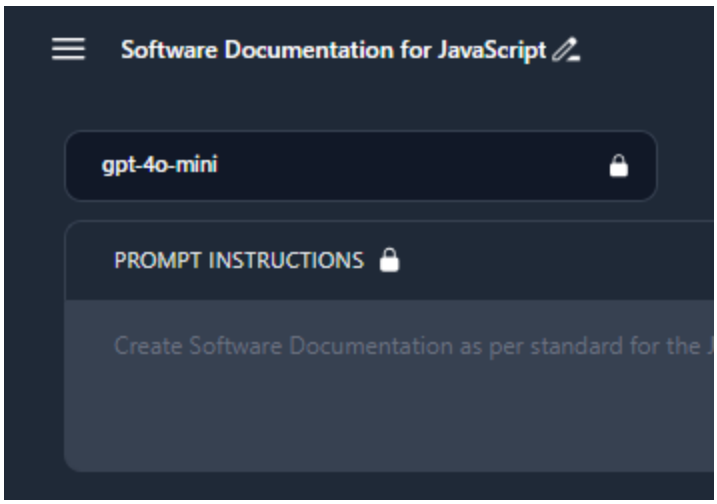
- Descriptive comments were added at the beginning of each section to explain the purpose of imported modules, server initialization, and function definitions.
- Inline comments were added within functions to clarify the purpose of specific lines, such as constructing the API URL, making requests, and handling responses.
- Comments were included to explain error handling and the server's listening behavior.

These comments will help other developers (or your future self) understand the code's functionality and flow more easily. If you have any further requests or need additional modifications, let me know!

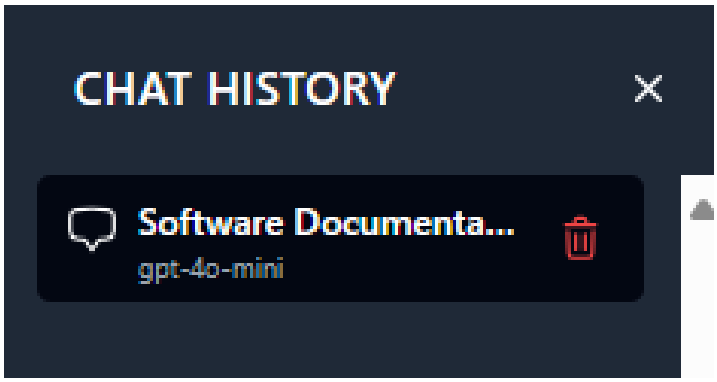
# Conclusion

Congratulations! You have created a complete documentation for a JavaScript code in such little time. If you are happy with the output produced, you may copy the code and use it. If you want to improvise, you may ask additional relevant questions. Generative AI depends on external sources to supplement its responses with more facts and realism. However, it is your prerogative as a software developer to ascertain the accuracy of the output.

You can always visit the conversation during the live session using the history option by clicking on the main menu on top-left.




Then, choose the chat by title among all the labs listed.




## Author(s)

Rama
Lavanya