# Guide to Using GitHub Copilot for Backend Development

In this section, we will explore how to use GitHub Copilot for backend development. Follow the tasks step by step to learn about various Copilot features.

## Task 1: Autocomplete a Line

Autocompletion is a common tool provided by IDEs. Copilot enhances this feature by using the context and code style around.

Go to API/Controller/Math.cs. Delete the `// TODO Task 1` comment. Start typing `if` and wait for a suggestion. Copilot should suggest something like `if (a < 0) return BadRequest("Invalid input");`.

## Task 2: Autocomplete Full Code Based on Context

Copilot can generate entire functions or even classes based on the context.

Go to API/Controller/Math.cs. Delete the `// TODO Task 2` and the `throw new System.NotImplementedException();` comment. Press `CTRL+Enter`. Accept a suggestion of the factorial implementation that you like, e.g., `Accept suggestion 1`.

## Task 3: Generate Code with Prompt

Sometimes there might be no context for Copilot to generate code. In such cases, you can provide context via prompts.

Go to API/Controller/Math.cs. Delete the `// TODO Task 3`. Open the Copilot Chat and reference the file using `#file:Math.cs` in the message. Prompt for method implementation like `Add a get method that implements Greatest Common Divisor (GCD)`. Copy and paste the suggested code snippet or add additional prompts to fine-tune the result.

## Task 4: Inline Code Improvement / Refactoring

You can prompt Copilot for completions without using the side chat, which is handy for quick and integrated suggestions.

Go to API/Controller/Math.cs. Delete the `// TODO Task 4` and select the whole `IsPrimeBad` function. Open the inline chat using `Command+I`. Prompt `Can you suggest a more efficient implementation for the IsPrimeBad method?`. Copilot will suggest changes. If you are happy with them, accept the changes by clicking `Accept`.

## Task 5: Commenting Documentation

Copilot can help generate comments and documentation.

Go to API/Controller/Math.cs. Delete the `// TODO Task 5` and select everything (`CTRL+A`). Open the inline chat as in Task 4 and simply write `/doc`. If this generates broken code, `Discard` the suggestion and use the side chat again and prompt `#file:Math.cs /doc`. Copy and paste the generated code.

## Task 6: Adapting Code Style

GitHub Copilot uses the context of the code in your current file and project to adapt its suggestions to your coding style. It does not learn or remember your coding style across different sessions or projects.

Go to API/Controller/Task.cs. Delete the `// TODO Task 6` and open the inline chat. Prompt code generation for a delete method like `implement a delete method`. You will see that the generated code adapted the style of commenting each line.

## Task 7: Find and Fix a Bug

Sometimes you just can't see the forest for the trees. Bugs might be caused by a single typo. Copilot provides an extension (also possible via prompting) to help you identify a bug and also provides a solution.

Go to API/Utils/Dijkstra.cs. The chat should already select the active file automatically for the context. Use the tool `@workspace /fix`. Copilot will find the bug and also tell you how to fix it.

## Task 8: Error Logs to Find a Bug

As we have seen before, Copilot can review our code and find and fix bugs. But what if we have an unknown bug and only some error logs?

Open your terminal and run the API service:

```
cd API
dotnet run
```

We see some errors that are easy to understand and fix. But let's assume we have no idea where this bug is coming from. In the Copilot chat, use the terminal extension @terminal /explain with the additional prompt what is causing the error? to ask Copilot. With that answer, we can fix our error.

We might also add log files or copy-paste errors from any source. The extension helps to add context without the need for copying. Keep in mind that the context provided is limited.

## Task 9: Explaining Code and Solution

Sometimes you want to know what some class or algorithm is doing, especially if it is not clear what something might be.

Go to API/Utils/Magic.cs. You might understand the code, but do you know what this is? The side chat should already detect the file `Utils.cs` since it is open in the active editor tab. Use the extension `@workspace /explain` and Copilot will tell you what algorithm this implementation might be. "Psst... it's Floyd-Warshall."

Now let's go to API/Entities/TodoTask.cs. Again, the chat should already select the file automatically for the context. Prompt for an explanation of the attributes, this time without using the extension, e.g., `explain the attributes`. You will see that you get a clean explanation of all the attributes.

## Task 10: Editorconfig

Copilot uses the `.editorconfig` file by default when generating code. This ensures that the generated code adheres to the coding style and formatting rules specified.

Go to API/Utils/HammingDistance.cs. Then prompt `implement a static hamming distance class` on the chat. You can see that the opening `{` are placed at the end of the line instead of newline. Copilot knows it because this is defined in the `.editorconfig`.

## Task 11: Generate Tests

Copilot can add missing tests, add more test cases, or even generate tests from scratch.

Go to Tests/Controller/TaskTest.cs. In the chat, use the extension `@workspace /tests #file:Task.cs #file:TodoTask.cs`. Copilot will then automatically generate tests for the controller. Depending on the version and IDE, you can accept these tests directly which will create the test file on its own.

## Task 12: Code Review

Copilot can also review your code to find potential bugs or things to improve.

Go to API/Utils/Dijkstra.cs. Use the prompt `Review my Dijkstra class` to get a review.