

Learning and Application of Copilot in APL Block Testing Tool Development

Name	M. Sushitha Reddy
Component	APL
Tool Name	Apl Block Testing Tool
Concepts Learned	HTML, CSS, JavaScript, SQL lite Db, MVC web framework

Table Of Contents

No	Contents	Page
1	Using GitHub Copilot for Learning HTML, CSS, and .NET MVC	3
1.1	Learning HTML with Copilot	3
1.2	Exploring HTML Tags	4
1.3	Learning CSS with Copilot	5
2	Developing .NET MVC Applications with Copilot	7
2.1	Controller	7
2.2	View Setup	8
3	Using Copilot to Set Up a SQLite Database and Save Data	12
3.1	Setting Up the SQLite Database	12
3.2	Saving the test case steps in data base	13
4	Conclusion	14

1. Using GitHub Copilot for Learning HTML, CSS, and .NET MVC

Introduction

- Recently, I explored GitHub Copilot to enhance my learning in web development, particularly focusing on HTML, CSS, and .NET MVC applications. Copilot provided real-time suggestions and solutions, helping me grasp fundamental concepts quickly. This document outlines my experience using Copilot and how it assisted me in streamlining the learning process.

1.1 Learning HTML with Copilot

1. Basic Structure Generation

- One of the first things I tried was creating a basic HTML structure. I started by typing the `<!DOCTYPE html>` tag, and Copilot immediately suggested the entire code, including the necessary `<html>`, `<head>`, and `<body>` tags. This saved time and gave me an idea of the essential components in any HTML document.

provide basic html code with explanation of tags



GitHub Copilot

Sure! The code you provided is written in Razor syntax, which is a combination of HTML and C# code. However, the code you shared is incomplete and ends abruptly. It seems like you were about to write an HTML comment using the `<!--` tag, but it is not closed properly.

To help you understand HTML tags, here's a basic HTML code snippet with explanations for each tag:

```
<!DOCTYPE html>
<!-- The <!DOCTYPE html> tag defines the document type and version of HTML being used (HTML5 in this case) -->
<html>
<head>
  <!-- The <title> tag sets the title of the web page, which is displayed in the browser's title bar or tab -->
  <title>My Web Page</title>

  <!-- The <meta> tag with the name "viewport" helps control the layout on mobile browsers -->
  <!-- The content attribute "width=device-width" sets the width of the page to follow the screen-width of the device -->
  <!-- The content attribute "initial-scale=1.0" sets the initial zoom level when the page is first loaded by the browser -->
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
  <!-- The <h1> tag defines a top-level heading -->
  <h1>Welcome to My Web Page</h1>

  <!-- The <p> tag defines a paragraph of text -->
  <p>This is a paragraph of text.</p>

  <!-- The <ul> tag creates an unordered list -->
  <ul>
    <!-- The <li> tag defines a list item within an unordered or ordered list -->
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
```

```
<!-- The <table> tag creates an HTML table -->
<table>
  <!-- The <tr> tag defines a table row -->
  <tr>
    <!-- The <th> tag defines a table header cell -->
    <th>Name</th>
    <th>Age</th>
  </tr>
  <tr>
    <!-- The <td> tag defines a table data cell -->
    <td>John</td>
    <td>25</td>
  </tr>
  <tr>
    <td>Jane</td>
    <td>30</td>
  </tr>
</table>
</body>
</html>
```

Explanation of the viewport tag:

- `<meta name="viewport" content="width=device-width, initial-scale=1.0">`: This meta tag is crucial for responsive web design. It ensures that the web page is displayed correctly on different devices by setting the width of the page to match the width of the device's screen and setting the initial zoom level to 1.0. This helps in making the web page look good on both desktop and mobile devices.

- Copilot not only completed the code but also taught me the purpose of each section, including meta tags like `viewport`, which I didn't know initially.

1.2 Exploring HTML Tags

- As I progressed, I asked Copilot to help with various HTML tags such as images and links. It is very useful when working with attributes like `src` for images and `href` for links.

explain about HTML tags such as images and links and how to use it in html code



Certainly! Let's discuss the HTML tags for images and links, and how to use them in your HTML code.

Image Tag (``)

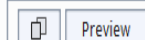
The `` tag is used to embed images in an HTML page. It is an empty tag, meaning it does not have a closing tag. The `` tag has several attributes, but the most important ones are `src` and `alt`.

- `src`: Specifies the path to the image file.
- `alt`: Provides alternative text for the image if it cannot be displayed.

Example:

```

```



Link Tag (<a>)

The <a> tag is used to create hyperlinks, which are clickable links that navigate to other web pages or resources. The <a> tag can contain text, images, or other HTML elements. The most important attribute of the <a> tag is `href`.

- `href`: Specifies the URL of the page the link goes to.

Example:

```
<a href="https://www.example.com">Visit Example.com</a>
```

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Image and Link Example</title>
</head>
<body>
  <h1>Image and Link Example</h1>
  <a href="https://www.example.com" target="_blank">
    
  </a>
  <!-- The image is wrapped in an anchor tag, making it a clickable link. -->
</body>
</html>
```

Explanation:

1. Image Tag ():

- `src`: Specifies the path to the image file.
- `alt`: Provides alternative text for the image if it cannot be displayed.
- `width` and `height`: Specify the dimensions of the image.

2. Link Tag (<a>):

- `href`: Specifies the URL of the page the link goes to.
- `target`: Specifies where to open the linked document. `_blank` opens it in a new tab.

3. Combined Example:

- The image is wrapped in an anchor (<a>) tag, making it a clickable link.

These tags are fundamental for adding images and links to your web pages, enhancing both functionality and user experience.

1.3 Learning CSS with Copilot

1. Styling HTML Elements

- When I began learning CSS, Copilot assisted by suggesting common properties and values for styling HTML elements. I started by creating a basic `style.css` file, and Copilot recommended useful properties like `font-family`, `color`, and `background-color`.

suggest common properties and values for styling HTML elements

GitHub Copilot

Certainly! Here are some common CSS properties and values that you can use to style HTML elements:

Common CSS Properties and Values

1. Color and Background:

- **color:** Sets the text color.

```
color: #333; /* Hex color code */
color: rgb(51, 51, 51); /* RGB color code */
color: red; /* Named color */
```



Preview

- **background-color:** Sets the background color.

```
background-color: #f0f0f0;
background-color: rgba(240, 240, 240, 0.5); /* With transparency */
```



Preview

2. Font and Text:

- **font-family:** Sets the font family.

```
font-family: Arial, sans-serif;
```

This helped me quickly understand how CSS affects the look and feel of a webpage.

2. Learning Flexbox for Layouts

- One challenge I faced was learning modern CSS layouts like Flexbox. Copilot provided immediate suggestions for properties like `display: flex`, `justify-content`, and `align-items`, which allowed me to experiment and understand the Flexbox model more easily.

Explain about CSS layouts : Flexbox with example

GitHub Copilot

Certainly! Flexbox is a powerful layout module in CSS that allows you to design complex layouts with ease. It provides a more efficient way to lay out, align, and distribute space among items in a container, even when their size is unknown or dynamic.

Key Concepts of Flexbox

1. **Flex Container:** The parent element that holds the flex items. It is defined by setting the `display` property to `flex` or `inline-flex`.
2. **Flex Items:** The child elements of the flex container.

Flex Container Properties

- **display: flex;** Defines a flex container.
- **flex-direction:** Defines the direction of the flex items (row, row-reverse, column, column-reverse).
- **justify-content:** Aligns flex items along the main axis (flex-start, flex-end, center, space-between, space-around, space-evenly).
- **align-items:** Aligns flex items along the cross axis (flex-start, flex-end, center, baseline, stretch).
- **flex-wrap:** Controls whether the flex items should wrap or not (nowrap, wrap, wrap-reverse).

Flex Item Properties

- **flex-grow:** Defines the ability for a flex item to grow if necessary.
- **flex-shrink:** Defines the ability for a flex item to shrink if necessary.
- **flex-basis:** Defines the default size of an element before the remaining space is distributed.
- **align-self:** Allows the default alignment (or the one specified by `align-items`) to be overridden for individual flex items.

Here's an example of a simple layout using Flexbox:

```
<!DOCTYPE html>
<html>
<head>
  <title>Flexbox Example</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <style>
    /* Flex container */
    .container {
      display: flex; /* Defines a flex container */
      flex-direction: row; /* Items are placed in a row */
      justify-content: space-between; /* Distributes space between items */
      align-items: center; /* Aligns items vertically in the center */
      height: 100vh; /* Full viewport height */
    }

    /* Flex items */
    .item {
      background-color: #4CAF50; /* Green background */
      color: white; /* White text */
      padding: 20px; /* Padding around the text */
      margin: 10px; /* Margin around the item */
      flex-grow: 1; /* Items will grow to fill the container */
      text-align: center; /* Center text alignment */
    }
  </style>
</head>
<body>
  <div class="container">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>
</body>
</html>
```

Using these suggestions, I was able to build responsive layouts, enhancing my understanding of positioning elements.

Time Comparison: Manual Learning vs. Learning with GitHub Copilot

- Learning the concepts of HTML and CSS manually took me several hours of studying. However, with GitHub Copilot, I was able to grasp both the basic structure html generation and CSS styling much faster, as it provided instant suggestions and templates, saving me significant time and effort needed to understand and implement these concepts.

2. Developing .NET MVC Applications with Copilot

2.1 Controller

- As part of my learning, I built a .NET MVC application. I started by creating controllers, and Copilot suggested complete action methods, including how to return a view. When I typed the public ActionResult Display() method, it provided code that matched MVC best practices.

```
/// <summary>
/// Displays the login role on the view.
/// </summary>
/// <param name="loginRole">The login role object.</param>
/// <returns>The view displaying the login role.</returns>
public ActionResult Display(LoginRole loginRole)
{
    return View(loginRole);
}
```

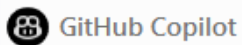
This automatic generation of methods saved me time and helped me follow the correct patterns from the start.

- When I needed to implement a form submission using `[HttpPost]`, Copilot suggested an efficient way to validate the model and handle the flow of data using `TempData` for state persistence.

Example Interaction:

Copilot helped me to develop the following method for handling form submissions and validating the model state:

suggest an efficient way to validate the model and handle the flow of data using `TempData`



To improve the efficiency of model validation and data handling using `TempData`, you can follow these steps:

1. **Consolidate TempData Access:** Access `TempData` only once to avoid redundant operations.
2. **Use Strongly Typed TempData:** Utilize a strongly-typed approach to avoid casting issues.
3. **Simplify Redirect Logic:** Streamline the redirection logic to make it more readable and maintainable.

Here's a refactored version of the `CreateTC` method:

```
[HttpPost]
public ActionResult CreateTC(LoginRole loginRole)
{
    if (ModelState.IsValid)
    {
        TempData["ModelData"] = loginRole;
        // Redirect to the Display action and pass the model
        return RedirectToAction("Display", loginRole);
    }

    // If ModelState is not valid, check if TempData contains previous data
    if (TempData.TryGetValue("ModelData", out var tempData))
    {
        loginRole = tempData as LoginRole;
        return RedirectToAction("Display", loginRole);
    }

    // If no valid data in TempData, return the view with the current model
    return View(loginRole);
}
```



Preview



Through Copilot, I learned how to:

- Validate the form using `ModelState.IsValid`.
- Use `TempData` to store and retrieve the model data between requests.
- Redirect the user to another action while passing the model as a parameter.

This interaction not only helped me understand the logic behind form handling but also saved me time by automatically suggesting best practices for dealing with post requests and state management.

- In this way, I utilized GitHub Copilot extensively to develop controllers in my web tool, including handling form submissions, model validation, and state management. Copilot not only helped me in understanding of how controllers interact with views but also provided real-time suggestions for improving code quality and adhering to best practices. This made the development process smoother and more efficient, allowing me to focus more on improving the functionality of the tool rather than getting stuck down in repetitive coding tasks.

Time Comparison: Manual Learning vs. Learning with GitHub Copilot

- Manually developing a controller in a .NET MVC application took me around 2 hours, as I had to follow the structure and syntax carefully. However, with GitHub Copilot's real-time suggestions and code completion, I was able to complete the task in about 45 minutes, significantly reducing the time spent on repetitive tasks and common patterns.

2.2 View Setup

- As I continued to develop my .NET MVC application, I utilized Copilot to assist in creating the necessary views alongside controllers. One important lesson I learned was how to create a form for submitting test case data, such as `TestCaseName`, `TestCaseSteps`, and `TestCaseFile` with the help of copilot. It also provided useful suggestions for setting up the form using Razor syntax and Bootstrap for styling, making the process smoother.
- *Example Interaction:*

For creating the form in the view, I used Copilot's suggestions to structure the layout using HTML and Razor helpers for form controls. This is the code I generated with Copilot's guidance:

Setting the View Title and Layout

```
@using CeatingDashboard.BusinessObjects
@{
    ViewBag.Title = "CreateTC";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<!DOCTYPE html>
```

What I Learned:

- Copilot suggested the use of `@model` to bind the `LoginRole` model to the view.
- `ViewBag.Title` allows setting the title of the page dynamically, which is then displayed in the browser tab.
- The `Layout` directive defines the shared layout file that contains the common structure of the website (e.g., header, footer, etc.).

Adding HTML Structure and Meta Tags

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Create New Testcases</title>

  <!-- Favicon -->
  <link rel="shortcut icon" href="~/Content/images/favicon.ico">

  <!-- Bootstrap CSS -->
  <link href="~/Content/css/bootstrap.min.css" rel="stylesheet" type="text/css" />

  <!-- Font Awesome CSS -->
  <link href="~/Content/font-awesome/css/font-awesome.min.css" rel="stylesheet" type="text/css" />

  <!-- Custom CSS -->
  <link href="~/Content/css/style.css" rel="stylesheet" type="text/css" />
</head>
```

What I Learned:

- Copilot helped me understand the importance of using meta tags like `charset` and `viewport` to ensure proper character encoding and responsiveness.
- The `<title>` tag is dynamically set using `ViewBag.Title`.
- Copilot suggested including descriptive meta tags to enhance SEO and provide relevant information about the page.

Building the Form for Test Case Details

Create a Razor view that includes a form for collecting user input on test case details such as test case name and test case steps

 GitHub Copilot

Certainly! Below is an example of a Razor view that includes a form for collecting user input on test case details, specifically the test case name and test case steps.

```
@using (Html.BeginForm("SecondView", "CreateTestCases", FormMethod.Post, new { id = "contact-form", role = "form" }))
{
<div class="controls">
  <div class="row">
    <div class="col-md-6">
      <div class="form-group">
        @Html.LabelFor(m => m.TestCaseName, "TestCase Name *")
        @Html.TextBoxFor(m => m.TestCaseName, new { @class = "form-control", placeholder = "Enter the Name of the Testcase *", required = "required" })
        @Html.ValidationMessageFor(m => m.TestCaseName)
      </div>
    </div>
    <div class="col-md-6">
      <div class="form-group">
        @Html.LabelFor(m => m.TestCaseSteps, "No. of steps *")
        @Html.TextBoxFor(m => m.TestCaseSteps, new { @class = "form-control", placeholder = "Enter No. of Steps in Testcase *", required = "required" })
        @Html.ValidationMessageFor(m => m.TestCaseSteps)
      </div>
    </div>
  </div>
</div>
```

What I Learned:

- Copilot suggested using `Html.BeginForm` to create the form, specifying the action (`SecondView`), controller (`CreateTestCases`), and form method (`Post`).
- Inside the form, Copilot guided the use of `@Html.LabelFor`, `@Html.TextBoxFor`, and `@Html.ValidationMessageFor` to link form fields with model properties (`TestCaseName`).
- Bootstrap classes like `form-control` and `col-md-6` were suggested by Copilot for responsiveness and styling.
- This structure made it easy to build a form that directly interacts with the backend, with proper form validation messages displayed dynamically.

Adding File Upload Input with Validation

```
@using (Html.BeginForm("Index", "Contact", FormMethod.Post, new { id = "contact-form", role = "form" }))
{
    <div class="row">
        <div class="col-md-12">
            <div class="form-group">
                @Html.LabelFor(m => m.TestCaseFile, "DB FileName *")
                <div class="input-group">
                    @Html.TextBoxFor(m => m.TestCaseFile, new { @class = "form-control", placeholder = "Enter Filename to Load Testcase *", required = "required" })
                    <span class="input-group-btn">
                        <button type="submit" class="btn btn-success ml-3">Browse</button>
                    </span>
                </div>
                @Html.ValidationMessageFor(m => m.TestCaseFile)
            </div>
        </div>
    </div>
</div>
}
```

What I Learned:

- Copilot helped me structure a file upload input field using `TextBoxFor` and provided the `input-group` for file browsing functionality.
- Copilot also recommended using validation helpers like `@Html.ValidationMessageFor` to handle validation errors for required fields like `TestCaseFile`.
- This provided an effective way to manage file inputs within the form while ensuring validation feedback.

Submit Button for Final Submission

```
<div class="row">
    <div class="col-md-12">
        <input type="submit" class="btn btn-success btn-send pt-2 btn-block" value="Submit Data">
    </div>
</div>
```

What I Learned:

- Copilot suggested adding a submit button with Bootstrap classes (`btn`, `btn-success`, and `btn-block`) to create a fully styled and responsive submission button.
- This allows the user to submit all the form data to the backend for further processing.

Through Copilot's guidance, I was able to structure the entire form for creating test cases with proper validation, Bootstrap styling, and Razor syntax.

Time Comparison: Manual Learning vs. Learning with GitHub Copilot

- Setting up the above view manually in a .NET MVC application will took me around 3.5 hours, as I had to structure the HTML and apply Razor syntax from scratch. With GitHub Copilot, I was able to complete the view setup in about 30 minutes.

3. Using Copilot to Set Up a SQLite Database and Save Data

- While building the backend logic for the APL Block Testing Tool, I needed to set up a database using SQLite to store and retrieve test case steps. Copilot assisted me in writing the C# code for this database setup and data saving functionality. Below is a breakdown of the code Copilot helped me generate and how I learned each aspect of it.

3.1 Setting Up the SQLite Database



What I Learned:

- *Database Connection:* Copilot suggested how to use `SQLiteConnection` to establish a connection with the SQLite database. It also provided the correct connection string to access the database file.
- *Table Creation:* The code creates a table `TestCasesSteps` if it doesn't exist already. Copilot helped me understand how to write SQL queries in C# using `SQLiteCommand`.
- *Command Execution:* Using `ExecuteNonQuery()`, Copilot guided me to execute the SQL command for creating the table.

This allowed me to set up the database structure without manual intervention, ensuring the table exists before saving any test case data.

Time Comparison: Manual Learning vs. Learning with GitHub Copilot

- Writing a method to set up an SQLite database for storing test case steps manually took me around 2 hours, as I had to research the correct syntax and ensure the database connection and queries were properly configured. With GitHub Copilot, I was able to complete the task in about 1 hour, as it provided useful code suggestions and pre-written patterns, speeding up the process and reducing errors.

3.2 Saving Test Case Steps

Write a Method to save test case steps to the SQLite database and Ensure the directory for the database exists, create if needed and Open the SQLite connection and save the steps in the table

GitHub Copilot

Certainly! Below is a method that saves test case steps to the SQLite database. It ensures the directory for the database exists, creates it if needed, opens the SQLite connection, and saves the steps in the table:

```
public void SaveTestCasesSteps(IEnumerable<TestCaseStep> steps, string databaseFilePath)
{
    // Ensure the directory for the database exists, create if needed
    var directory = Path.GetDirectoryName(databaseFilePath);
    if (!Directory.Exists(directory))
    {
        Directory.CreateDirectory(directory);
    }

    // Define the connection string for the SQLite database
    string connectionString = $"Data Source={databaseFilePath};Version=3;";

    // Establish a connection to the SQLite database
    using (var connection = new SQLiteConnection(connectionString))
    {
        connection.Open();

        // Define the SQL query to create the table if it doesn't exist
        string createTableQuery = @"
CREATE TABLE IF NOT EXISTS TestCaseSteps (
    StepCount INTEGER PRIMARY KEY AUTOINCREMENT,
    InputsAndOutputs TEXT,
    InputValue TEXT,
    ExpectedValue TEXT
);";

        // Execute the SQL query to create the table

        // Execute the SQL query to create the table
        using (var command = new SQLiteCommand(createTableQuery, connection))
        {
            command.ExecuteNonQuery();
        }

        // Define the SQL query to insert the test case steps
        string insertQuery = @"
INSERT INTO TestCaseSteps (StepCount, InputsAndOutputs, InputValue, ExpectedValue)
VALUES (@StepCount, @InputsAndOutputs, @InputValue, @ExpectedValue);";

        // Insert the test case steps
        foreach (var step in steps.Select((step, index) => new
        {
            StepCount = index + 1,
            InputsAndOutputs = step.InputsAndOutputs,
            InputValue = step.InputValue,
            ExpectedValue = step.ExpectedValue
        }))
        {
            using (var command = new SQLiteCommand(insertQuery, connection))
            {
                command.Parameters.AddWithValue("@StepCount", step.StepCount);
                command.Parameters.AddWithValue("@InputsAndOutputs", step.InputsAndOutputs);
                command.Parameters.AddWithValue("@InputValue", step.InputValue);
                command.Parameters.AddWithValue("@ExpectedValue", step.ExpectedValue);
                command.ExecuteNonQuery();
            }
        }
    }
}
```

Preview

What I Learned:

- **File Path Management:** Copilot provided a solution to check whether the directory where the database file is stored exists. If not, it creates the directory using `Directory.CreateDirectory(directory)`.
- **Table Structure:** For the `TestCaseSteps` table, Copilot introduced the `AUTOINCREMENT` feature for the `StepCount` column, which ensures that each test case step has a unique identifier.
- **Saving Steps:** Copilot suggested using the `SQLiteConnection` class to open a connection to the database and `SQLiteCommand` to execute SQL queries, ensuring that the steps are properly saved in the database.

Time Comparison: Manual Learning vs. Learning with GitHub Copilot

- Writing the `SaveTestCaseSteps` method manually, which involved setting up the database connection, creating the table if it didn't exist, and inserting data, took me around 4 hours due to the need to ensure the SQL queries were correct and the connection was properly handled. With GitHub Copilot, I was able to write this method in about 1.5 hours, as it provided suggestions for SQL queries, connection handling, and the logic for inserting test case steps, making the process more efficient.

Conclusion

In this way, I used Copilot to help with the development of the database layer for the APL Block Testing Tool. Copilot not only provided code suggestions but also assisted in understanding how to manage database connections, execute SQL queries, and handle file paths effectively in C#. This greatly enhanced the efficiency of the development process.