GitHub Copilot for Developers

advanced techniques cookbook

Andrew Scoppa
GitHub Expert Services

Debugging Errors - Debug Invalid JSON

Example Scenario

```
Error: Parse error
----------------------^
Expecting 'STRING', 'NUMBER', 'NULL', 'TRUE', 'FALSE', '{', '[', got 'undefined'
```

Below is the JSON data that caused the error:

```
{
  "location": "San Francisco",
  "current_weather": {
    "temperature": 18,
    "unit": "Celsius",
    "conditions": "Cloudy
  },
  "forecast": {
    "day": "Monday",
    "high": 22,
    "low": 15,
    "precipitation": 10
  }
}
```

Example Prompt

```
Why is my JSON object invalid and how can I fix it?
```

Example Response

```
{
  "location": "San Francisco",
  "current_weather": {
    "temperature": 18,
    "unit": "Celsius",
    "conditions": "Cloudy"
  },
  "forecast": {
    "day": "Monday",
    "high": 22,
    "low": 15,
    "precipitation": 10
  }
}
```

Debugging invalid JSON

Debugging Errors – Handling API Rate Limits

Example Scenario

The following Python code fetches weather data from an external API. If the API has rate limits, requests may fail when limits are exceeded, and your app may need a way to handle these responses gracefully by implementing retry logic.

```python
from flask import Flask, request
import requests

app = Flask(__name__)

WEATHER_API_URL = "https://api.example.com/weather"

@app.route('/get_weather', methods=['GET'])
def get_weather():
    city = request.args.get('city')
    # Simulate an API request to the external weather service
    response = requests.get(WEATHER_API_URL, params={"city": city})
    weather_data = response.json()

    return weather_data
```

Example Prompt

```
How can I handle API rate limits within get_weather().
```

Example Response

```python
import requests
from flask import Flask, request
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.util.retry import Retry

app = Flask(__name__)

WEATHER_API_URL = "https://api.example.com/weather"

def requests_retry_session(
    retries=3,
    backoff_factor=0.3,
    status_forcelist=(500, 502, 504),
    session=None,
):
    session = session or requests.Session()
    retry = Retry(
        total=retries,
        read=retries,
        connect=retries,
        backoff_factor=backoff_factor,
        status_forcelist=status_forcelist,
    )
    adapter = HTTPAdapter(max_retries=retry)
    session.mount("http://", adapter)
    session.mount("https://", adapter)
    return session

@app.route('/get_weather', methods=['GET'])
def get_weather():
    city = request.args.get('city')
    try:
        response = requests_retry_session().get(WEATHER_API_URL, params={"city":
city})
        response.raise_for_status()
        weather_data = response.json()
    except requests.exceptions.RequestException as e:
        return {"error": str(e)}, 500

    return weather_data
```

[Handling API rate limits]

Explore Feature Implementations

Example Scenario

You're assigned the task of creating a new feature that displays the most recently viewed items on a user dashboard

```
[Feature Request] Add "Recent Items" section to user dashboard #753

## Issue Description
Users have requested a "Recent Items" section on their dashboard to quickly
access the most recently viewed documents, projects, or files. This section
should display the ten most recent items, sorted by the last access time.

## Requirements
* Display up to ten items that the user has accessed recently.
* Items must be sorted by the time they were last accessed.
* The solution should minimize database load.
* Data should reflect updates in near real-time.
* The solution should prioritize performance, especially for large datasets of
items.

## Acceptance Criteria
* Dashboard loads the "Recent Items" section within 200ms under normal load.
* The solution must not degrade performance for users with thousands of items.
```

Example Prompt

```
Given the criteria in Issue #753, list options for implementation and tradeoffs for
each option.
```

Example Response

```
# Implementation Options and Tradeoffs for "Recent Items" Section

## 1. Client-Side Storage

- Description: Store recent items data in the user's browser using LocalStorage
or IndexedDB.
- Tradeoffs:
  - Pros:
    - Reduces server load by handling data storage and retrieval on the client
side.
    - Immediate updates and access without server communication.
  - Cons:
    - Limited by browser storage capacity.
    - Data lost if the user clears browser storage or switches devices.
    - Security concerns with storing sensitive data on the client side.

## 2. Server-Side Storage with Database Indexing

- Description: Store recent items in the database and use indexing to quickly
retrieve the most recent items.
- Tradeoffs:
  - Pros:
    - Centralized data storage, accessible from any device.
    - Scalable and can handle large datasets efficiently with proper indexing.
  - Cons:
    - Increased database load, especially for frequent access updates.
    - Potential latency in updating the recent items list.

## 3. In-Memory Caching (e.g., Redis)

- Description: Use an in-memory data store like Redis to cache recent items for
fast retrieval.
- Tradeoffs:
  - Pros:
    - Extremely fast data access and updates.
    - Reduces database load by offloading frequent read operations.
  - Cons:
    - Requires additional infrastructure and maintenance.
    - Data may be lost if the cache is cleared or server restarts.
```

Exploring potential feature implementations

Refactoring Code - Improve Code Readability

Common Scenarios

Suggesting improvements to variable names

- Improve the variable names in this function to be descriptive.

Avoiding sequential conditional checks

- Simplify this code. Avoid using if/else chains but retain all function return values.

Reducing nested logic

- Rewrite this code to avoid the nested if/else statements.

Splitting large methods into smaller

- How could the `processOrder` method be refactored to be more useful and easier to maintain?

Improving code readability and maintainability

Refactoring Code - Fixing Lint Errors

Example Scenario

- Fix linting issues with Copilot Chat

Example prompts

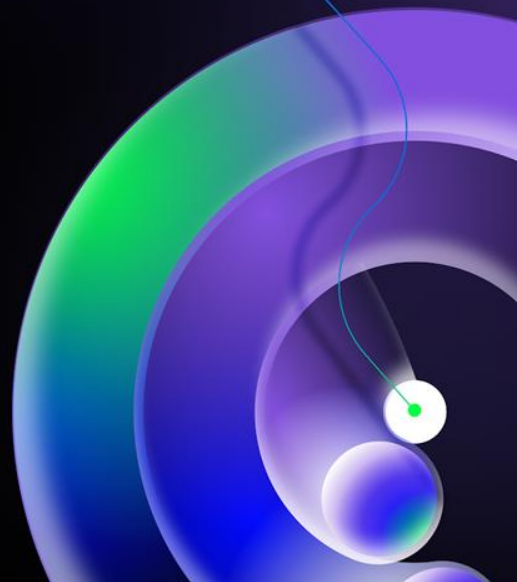
Select all of the code in the editor, then type:
- Fix the lint errors

Specify a particular set of coding guidelines for a language
- Use (PEP8/SonarLint/etc) to fix the lint errors

Drag a local file that defines your coding standards
- Use the attached style guide to fix the lint errors

Fix only a specific type of lint error
- Make sure all functions use snake_case naming style

Fixing lint errors



Refactoring Code - Performance Optimization

Example Scenario

The Python code prints every 1000th prime number up to the 10,000th prime. It includes a timer, and prints how long the code takes to run as the last line of the output. The program takes several seconds to complete.

```python
import time

def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

def every_1000th_prime():
    primes = []
    num = 2
    while len(primes) < 10000:
        if is_prime(num):
            primes.append(num)
        num += 1
    return [primes[i] for i in range(999, 10000, 1000)]

if __name__ == "__main__":
    start_time = time.time()
    primes = every_1000th_prime()
    for prime in primes:
        print(prime)
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Time taken: {elapsed_time:.10f} seconds")
```

Example Prompt

```
Optimize this code for performance
```

Refactoring for performance optimization

Example Response

```python
import time

def sieve_of_eratosthenes(limit):
    sieve = [True] * (limit + 1)
    sieve[0] = sieve[1] = False
    for start in range(2, int(limit**0.5) + 1):
        if sieve[start]:
            for multiple in range(start*start, limit + 1, start):
                sieve[multiple] = False
    return [num for num, is_prime in enumerate(sieve) if is_prime]

def every_1000th_prime():
    limit = 105000  # Higher than needed to ensure 10,000 primes
    primes = sieve_of_eratosthenes(limit)
    return [primes[i] for i in range(999, 10000, 1000)]

if __name__ == "__main__":
    start_time = time.time()
    primes = every_1000th_prime()
    for prime in primes:
        print(prime)
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Time taken: {elapsed_time:.10f} seconds")
```

Refactoring Code - Design Patterns

Example Scenario

This JavaScript code creates 3 Person objects, each with name and age properties, then uses the name person to retrieve and print their age.

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}

var people = {};

function addPerson(name, age) {
  people[name] = new Person(name, age);
}

function getAge(name) {
  if (people[name]) {
    console.log(name + " is " + people[name].age + " years old.");
  } else {
    console.log("Person not found.");
  }
}

addPerson("Ana", 30);
addPerson("Mario", 25);
addPerson("Louise", 40);

getAge("Mario");
```

Refactoring to implement a design pattern

Example Prompt

What design patterns could improve this code? Don't show me code examples.

Example Response

Copilot suggests the following patterns:

- Singleton pattern: to ensure that there is only one instance of the people object.
- Factory pattern: to create a factory for creating Person objects.
- Module pattern: to encapsulate the people object and related functions within a module to avoid polluting the global namespace.

Refactoring Code - Data Access Layers

Example Scenario

This Python code connects to a database, retrieves a user record, and returns the user data. However, it fails to abstract the database connection logic and uses a hardcoded query that's vulnerable to SQL injection.

```python
import sqlite3

def get_user_by_id(user_id):
    conn = sqlite3.connect('database.db')
    cursor = conn.cursor()
    cursor.execute(f"SELECT display_name FROM users WHERE id = {user_id}")
    user = cursor.fetchone()
    conn.close()
    return user
```

Example Prompt

How can I improve this code to make it safe and easier to update and expand?
List possible improvements but don't show revised code.

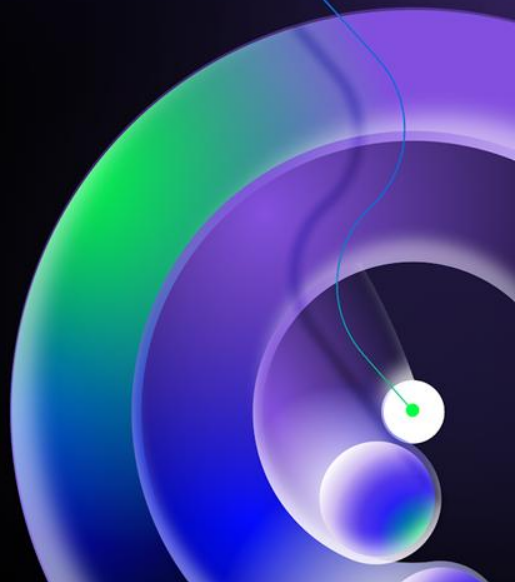
Example Response

Copilot makes several suggestions for improving the code:

- Abstract the database connection details.Use parameterized queries to prevent SQL injection.

- Use a context manager to ensure that the database connection is properly closed if an error occurs.

- Add error handling.



Refactoring data access layers

Refactoring Code - Decoupling Business Logic

Example Scenario

In this JavaScript code, the Counter component configures the business logic for the component and also defines UI code for using the component in a web page.

```
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  const increment = () => {
    if (count >= 10) {
      alert('Count cannot exceed 10');
      return;
    }
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
};

export default Counter;
```

Decoupling business logic from UI components

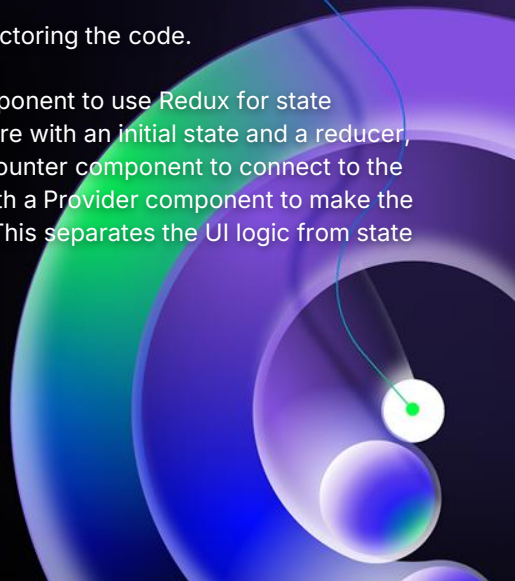
Example Prompt

Use Redux to separate the UI code from the Counter component in this code.

Example Response

Copilot provides step-by-step instructions for refactoring the code.

The suggested changes refactor the Counter component to use Redux for state management. This includes setting up a Redux store with an initial state and a reducer, defining an increment action, and modifying the Counter component to connect to the Redux store. Finally, the application is wrapped with a Provider component to make the Redux store available to the Counter component. This separates the UI logic from state management.



Refactoring Code - Cross-cutting Concerns

Example Scenario

You have a Python project that contains multiple service files in which logging occurs. The information that gets logged is defined within each of the individual service files. If the application is modified or extended in future, this design could lead to inconsistency in the content and style of log entries. You can consolidate and centralize the logging behavior to avoid this being distributed across your project.

Logging in multiple service files

- Consistency of logging as the service is modified or extended

- Consolidate and centralize logging behavior

- Service files
- `main.py`
- `logging_config.py`
- `order_service.py`

Example Prompt

`@workspace Refactor this project by using` **Aspect-Oriented Programming (AOP)** `to avoid logging that has cross-cutting concerns`

Example Response

To refactor this project using Aspect-Oriented Programming (AOP) to handle logging as a cross-cutting concern, you can use a library like `aspectlib`.

1. `pip install aspectlib`
2. Create new file `logging_aspect.py`
3. Apply the aspect to the service methods in `main.py`
4. Copilot then tells you to remove the logging code from all of the service classes, and gives you the code for the revised files. For example, this is the revised version of `order_service.py`

Handling cross-cutting concerns

Refactoring Code - Simplify Inheritance Hierarchies

Example Scenario

This C# code has a deep inheritance hierarchy where each class builds on the previous one.

```
public class Employee
{
    public string Name { get; set; }
    public int EmployeeID { get; set; }
}

public class Manager : Employee
{
    public int NumberOfTeams { get; set; }
}

public class SeniorManager : Manager
{
    public string Region { get; set; }
}

public class Director : SeniorManager
{
    public int NumberOfDepartments { get; set; }
}

public class VicePresident : Director
{
    public string Division { get; set; }
}

public class President : VicePresident
{
    public string Company { get; set; }
}
```

Simplifying complex inheritance hierarchies

Example Prompt

Refactor this code so that classes only inherit the properties of the Employee class. All of the other properties that each class currently inherits should instead be declared independently in each class definition.

Example Response

```
public class Employee
{
    public string Name { get; set; }
    public int EmployeeID { get; set; }
}

public class Manager : Employee
{
    public int NumberOfTeams { get; set; }
}

public class SeniorManager : Employee
{
    public int NumberOfTeams { get; set; }
    public string Region { get; set; }
}

public class Director : Employee
{
    public int NumberOfTeams { get; set; }
    public string Region { get; set; }
    public int NumberOfDepartments { get; set; }
}

public class VicePresident : Employee
{
    public int NumberOfTeams { get; set; }
    public string Region { get; set; }
    public int NumberOfDepartments { get; set; }
    public string Division { get; set; }
}

public class President : Employee
{
    public int NumberOfTeams { get; set; }
    public string Region { get; set; }
    public int NumberOfDepartments { get; set; }
    public string Division { get; set; }
    public string Company { get; set; }
}
```

Refactoring Code - Fixing Database Deadlocks

Example scenario

The following SQL snippet updates one row of a table, then performs an operation that takes several seconds, then updates another row in the same table. This is problematic because the transaction locks the `id = 1` row for several seconds before the transaction completes, releasing the lock. If another transaction starts during this time that performs a similar operation, but locks the `id = 2` row first, so that it can update the row, and then attempts to lock the `id = 1` row, then both transactions will be left waiting for the other to complete, resulting in deadlock.

```
BEGIN TRANSACTION;
UPDATE my_table SET value = 'Some value' WHERE id = 301;
-- Simulate a process taking 5 seconds:
WAITFOR DELAY '00:00:05';
UPDATE my_table SET value = 'Another value' WHERE id = 127;
COMMIT TRANSACTION;
```

Example prompt

Enter this follow-up prompt:

```
How can I rewrite this code to reduce the chance of deadlock to a minimum while not adversely affecting performance. Tell me anything else I can do to avoid a deadlock. Note: I don't want to add error handling at this time.
```

Example response

Copilot gives the following advice:

- **Order of Access:** Ensure that all transactions access tables and rows in the same order.
- **Keep Transactions Short:** Minimize the time a transaction holds locks.
- **Use Appropriate Isolation Levels:** Use the lowest isolation level that meets your requirements.

Here is a revised version of your code:

```
BEGIN TRANSACTION;
-- Update the rows in a consistent order
UPDATE my_table SET value = 'Another value' WHERE id = 127;
UPDATE my_table SET value = 'Some value' WHERE id = 301;
-- Simulate a process taking 5 seconds:
WAITFOR DELAY '00:00:05';
COMMIT TRANSACTION;
```

[Fixing database deadlocks or data integrity issues]

Documenting Code - Explaining Legacy Code

Example scenario

Consider the following COBOL code. If you're a Python developer (as an example), you might not be familiar with COBOL, so you could ask Copilot Chat to explain the code to you.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INSERT-RECORD.

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-STATUS-FLAGS.
    05 WS-DB-STATUS     PIC X(2).
       88 WS-SUCCESS    VALUE "00".
    05 WS-SQLCODE       PIC S9(9) COMP.
    05 WS-ERROR-MSG     PIC X(50).

LINKAGE SECTION.
01  LS-PARAMETERS.
    05 LS-PERSON-RECORD.
       10 PERSON-ID     PIC 9(6).
       10 PERSON-NAME   PIC X(50).
       10 PERSON-AGE    PIC 9(3).
    05 LS-RESULT        PIC X.
       88 SUCCESS       VALUE 'T'.
       88 FAILED        VALUE 'F'.

PROCEDURE DIVISION USING LS-PARAMETERS.
    PERFORM INSERT-AND-VALIDATE
    GOBACK
    .

INSERT-AND-VALIDATE.
    EXEC SQL
        INSERT INTO persons (id, name, age)
        VALUES (:PERSON-ID, :PERSON-NAME, :PERSON-AGE)
    END-EXEC

    IF SQLCODE = 0
        EXEC SQL COMMIT END-EXEC
        SET SUCCESS TO TRUE
    ELSE
        EXEC SQL ROLLBACK END-EXEC
        SET FAILED TO TRUE
        STRING "DB Error: " SQLCODE
            DELIMITED BY SIZE
            INTO WS-ERROR-MSG
        DISPLAY WS-ERROR-MSG
    END-IF
    .
```

Example prompt

You can ask for an explanation of the COBOL code from the point of view of a Python developer.

```
I'm a Python developer. Help me better understand this code. Please provide analogies.
```

Example response

> **Note**
>
> The following response is an example. Copilot Chat responses are non-deterministic, so you may get a different response from the one shown here.

Copilot provides a high-level overview of the code, before walking through the various operations and explaining them in a way that Python developers can understand. It uses analogies to make the explanation more accessible. The response might start with the following:

```
Let me explain this COBOL code using Python analogies:

High-Level Overview
This is similar to a Python function that inserts a person's record into a
database with error handling.
```

```python
# In Python, the COBOL divisions would be like:

# IDENTIFICATION DIVISION = module/file name
# PROGRAM-ID = function name
def insert_record():
    pass

# DATA DIVISION would be like declaring these variables:
class WorkingStorage:
    db_status = ""  # WS-DB-STATUS
    sqlcode = 0     # WS-SQLCODE
    error_msg = ""  # WS-ERROR-MSG

# LINKAGE SECTION is like function parameters
class PersonRecord:
    def __init__(self, person_id: int, person_name: str, person_age: int):
        self.person_id = person_id    # PERSON-ID
        self.person_name = person_name # PERSON-NAME
        self.person_age = person_age   # PERSON-AGE
        self.success = False          # LS-RESULT
```

Explaining legacy code

Documenting Code - Explaining Complex Logic

Example scenario

In the C# code below, we have a method which fetches data, retries if there's an error, and updates a status label. You might want to explain, in comments in the code, how the method works and how it handles retries and cancellations.

```
private static readonly HttpClient _client = new HttpClient();

public async Task<string> FetchDataFromApiWithRetryAsync(string apiUrl,
CancellationToken cancellationToken, int maxRetries, int cancellationDelay, Label
statusLabel)
{
    var retryCount = 0;
    using var cts =
CancellationTokenSource.CreateLinkedTokenSource(cancellationToken);

    while (retryCount < maxRetries)
    {
        try
        {
            cts.CancelAfter(cancellationDelay);
            return await FetchDataFromApiAsync(cts.Token, statusLabel);
        }
        catch (Exception ex) when (!(ex is OperationCanceledException))
        {
            if (retryCount < maxRetries - 1) {
                retryCount++;
                int delay = (int)Math.Pow(2, retryCount) * 1000;
                await Task.Delay(delay, cancellationToken);
                UpdateStatusLabel($"Retrying ({retryCount}/{maxRetries})...",
statusLabel);
            } else {
                throw new Exception($"Failed to fetch data after {maxRetries}
retries: {ex.Message}", ex);
            }
        }
    }

    throw new OperationCanceledException("The operation was cancelled.",
cts.Token);
}
```

Explaining complex algorithms or logic

Example prompt

We can ask Copilot Chat to make this code easier to understand using comments.

`Add comments to this code to make it more understandable`

Example response

> **Note**
> The following response is an example. Copilot Chat responses are non-deterministic, so you may get a different response from the one shown here.

Copilot responds with an overview of the update it suggests, and the code snippet with the comments added:

```
// Shared HttpClient instance for making HTTP requests (best practice for reuse)
private static readonly HttpClient _client = new HttpClient();

/// <summary>
/// Fetches data from an API with retry mechanism and timeout functionality
/// </summary>
/// <param name="apiUrl">The URL to fetch data from</param>
/// <param name="cancellationToken">Token to cancel the operation</param>
/// <param name="maxRetries">Maximum number of retry attempts</param>
/// <param name="cancellationDelay">Timeout duration in milliseconds</param>
/// <param name="statusLabel">UI label to show status updates</param>
public async Task<string> FetchDataFromApiWithRetryAsync(string apiUrl,
CancellationToken cancellationToken, int maxRetries, int cancellationDelay, Label
statusLabel)
{
    var retryCount = 0;
    // Create a linked cancellation token that combines user cancellation and
timeout
    using var cts =
CancellationTokenSource.CreateLinkedTokenSource(cancellationToken);

    while (retryCount < maxRetries)
    {
        try
        {
            // Set timeout for this attempt
            cts.CancelAfter(cancellationDelay);
            return await FetchDataFromApiAsync(cts.Token, statusLabel);
        }
        catch (Exception ex) when (!(ex is OperationCanceledException))
        {
            // If not the last retry attempt, implement exponential backoff
            if (retryCount < maxRetries - 1) {
                retryCount++;
                // Exponential backoff: 2^retryCount seconds (1s, 2s, 4s, 8s,
etc.)
                int delay = (int)Math.Pow(2, retryCount) * 1000;
```

Documenting Code - Syncing Documentation

Example scenario

Imagine a scenario where you have a TypeScript function that retrieves products by category name, but the documentation is out of date.

```
/**
 * Retrieves all products belonging to a specific category.
 *
 * @param categoryId - The unique identifier of the product category
 * @returns Promise that resolves to an array of Product objects
 *
 * @example
 * const products = await getByCategoryName(5);
 * // Returns: [{id: 1, name: "Product 1", categoryId: 5}, ...]
 *
 * @throws {PrismaClientKnownRequestError} If database query fails
 * @see {@link https://www.prisma.io/docs/reference/api-reference/prisma-client-reference#findmany}
 */
async getByCategoryName(categoryId: number, page = 1, pageSize = 10): Promise<Products[]> {
  return await prisma.products.findMany({
    where: { categoryId },
    skip: (page - 1) * pageSize,
    take: pageSize,
    include: {
      reviews: {
        orderBy: { date: 'desc' },
        take: 5,
      },
    },
  });
}
```

Example prompt

You can ask Copilot Chat to update the documentation according to the current code.

`Update the existing documentation for the getByCategoryName function to reflect the current implementation.`

Example response

> **Note**
>
> The following response is an example. Copilct Chat responses are non-deterministic, so you may get a different response from the one shown here.

Copilot explains it will create the updated documentation for the `getByCategoryName` function. It then provides the updated documentation, which now includes the `page` and `pageSize` parameters, as well as the `PrismaClientKnownRequestError` error that can be thrown.

For example:

```
/**
 * Retrieves all products belonging to a specific category with pagination and recent reviews.
 *
 * @param categoryId - The unique identifier of the product category
 * @param page - The page number for pagination (default: 1)
 * @param pageSize - Number of products per page (default: 10)
 * @returns Promise that resolves to an array of Product objects with their 5 most recent reviews
 *
 * @example
 * const products = await getByCategoryName(5, 2, 20);
 * // Returns: [{id: 1, name: "Product 1", categoryId: 5, reviews: [...5 recent reviews]}, ...]
 *
 * @throws {PrismaClientKnownRequestError} If database query fails
 * @see {@link https://www.prisma.io/docs/reference/api-reference/prisma-client-reference#findmany}
 */
```

Syncing documentation with code changes

Testing Code - Generate Unit Tests

Example scenario

Imagine you have a Python function which validates data before updating the database. You need to create unit tests to ensure the validation performs the way you expect. You can ask Copilot Chat to generate these tests for you.

```
def validate_price(price: float):
  if price <= 0:
    raise ValueError("Price must be greater than 0")
  if price > 1000:
    raise ValueError("Price must be less than or equal to 1000")
```

Example prompt

```
/tests Generate unit tests for this function. Validate both success and failure, and include edge cases.
```

Example response

> **Note**
>
> The following response is an example. Copilot Chat responses are non-deterministic, so you may get a different response from the one shown here.

If this is your first time asking Copilot Chat to generate unit tests, it may ask you if you wish to configure unit tests for your project. For this example, we are assuming you decline this offer.

Copilot will provide a strategy for generating the tests, and then provide the tests themselves.

```
import unittest
from generate_unit_tests import validate_price

class TestValidateProduct(unittest.TestCase):
  def test_price_zero(self):
    with self.assertRaises(ValueError) as context:
      validate_price(0)
    self.assertEqual(str(context.exception), "Price must be greater than 0")

  def test_price_negative(self):
    with self.assertRaises(ValueError) as context:
      validate_price(-10)
    self.assertEqual(str(context.exception), "Price must be greater than 0")

  def test_price_above_limit(self):
    with self.assertRaises(ValueError) as context:
      validate_price(1500)
    self.assertEqual(str(context.exception), "Price must be less than or equal to 1000")

  def test_price_edge_case_zero(self):
    with self.assertRaises(ValueError) as context:
      validate_price(0)
    self.assertEqual(str(context.exception), "Price must be greater than 0")

  def test_price_edge_case_max(self):
    try:
      validate_price(1000)
    except ValueError:
      self.fail("validate_price() raised ValueError unexpectedly!")

if __name__ == '__main__':
  unittest.main()
```

[Writing tests with GitHub Copilot]

Testing - Create Mock Objects

Example scenario

Imagine a website built with TypeScript which displays a list of runners. You have a service which fetches the runners from a database, and the server-side code which uses this service. You want to test the server-side code, but you don't want to make a call to the database. You can ask Copilot Chat to generate a mock object for the service.

```
// SvelteKit example at +page.server.ts
import service from './service';

export async function load({ params }) {
  const runner = await service.getRunner(params.id);

  return {
    runner
  };
}
```

Creating mock objects to abstract layers

Testing - Create Mock Objects

Example prompt

This example assumes you are using vitest for your tests, but will work with other test runners.

```
/tests Create a unit test to ensure the service is called correctly. Mock the service object.
```

If you have an existing test you wish Copilot to use as a model, you can use `#file: path/to/test-file.js` to provide the path to the file, and include this information in the prompt.

Example response

> **Note**
>
> The following response is an example. Copilot Chat responses are non-deterministic, so you may get a different response from the one shown here.

Copilot provides the plan it will implement and the code text which includes the mocks.

```
import { describe, it, expect, vi, beforeEach } from 'vitest';
import { load } from './+page.server.ts';

// Mock the entire service module
vi.mock('./service', () => ({
  default: {
    getRunner: vi.fn()
  }
}));

// Import the mocked service
import service from './service';

describe('load function', () => {
  const mockRunner = {
    id: '123',
    name: 'Test Runner',
    status: 'active'
  };

  beforeEach(() => {
    // Reset mock before each test
    vi.resetAllMocks();

    // Setup default mock implementation
    (service.getRunner as any).mockResolvedValue(mockRunner);
  });

  it('should fetch runner with correct id', async () => {
    const params = { id: '123' };
    const result = await load({ params });

    expect(service.getRunner).toHaveBeenCalledWith('123');
    expect(service.getRunner).toHaveBeenCalledTimes(1);
    expect(result).toEqual({ runner: mockRunner });
  });
});
```

Creating mock objects to abstract layers

Testing Code - Create E2E tests

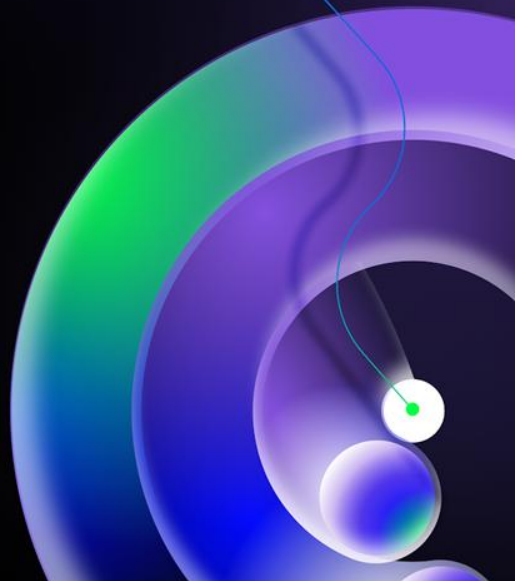
Example scenario

Imagine a React application that displays product details on a webpage. You need to create end-to-end tests to ensure the product details are displayed correctly. You can ask Copilot Chat to generate these tests for you.

```
import React, { useState, useEffect } from 'react';
import PropTypes from 'prop-types';

const ProductDetails = ({ productId = '1' }) => {
  const [product, setProduct] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchProduct = async () => {
      try {
        const response = await fetch(`/api/product/${productId}`);
        if (!response.ok) {
          throw new Error('Product not found');
        }
        const data = await response.json();
        setProduct(data);
        setLoading(false);
      } catch (err) {
        setError(err.message);
        setLoading(false);
      }
    };
```



Creating end-to-end tests for a webpage

Testing Code - Create E2E tests

Example prompt

> **Note**
>
> This example uses Playwright for end-to-end testing, but you can use other frameworks like Selenium or Cypress.

`Using Playwright, generate an e2e test to ensure the product displays correctly.`

If you have an existing test you wish Copilot to use as a model, you can use `#file: path/to/test-file.js` to provide the path to the file, and include this information in the prompt.

Example response

> **Note**
>
> The following response is an example. Copilot Chat responses are non-deterministic, so you may get a different response from the one shown here.

Copilot will respond with an end-to-end test that you can use for your application.

```
import { test, expect } from '@playwright/test';

test.describe('ProductDetails Component', () => {
  test.beforeEach(async ({ page }) => {
    // Set up API mocking
    await page.route('/api/product/1', async (route) => {
      await route.fulfill({
        status: 200,
        contentType: 'application/json',
        body: JSON.stringify({
          name: 'Test Product',
          description: 'This is a test product',
          price: 99.99
        })
      });
    });

    // Mock error response for product ID 999
    await page.route('/api/product/999', async (route) => {
      await route.fulfill({
        status: 404,
        contentType: 'application/json',
        body: JSON.stringify({ error: 'Product not found' })
      });
    });
  });
```

[Creating end-to-end tests for a webpage]