Debugging with GitHub Copilot

Leverage AI-assisted debugging to identify, understand, and fix issues in your code.

🎯 Overview

GitHub Copilot helps with debugging by: - Explaining error messages - Identifying potential bugs - Suggesting fixes - Generating test cases to reproduce issues - Analyzing stack traces

---

🔍 Understanding Errors

Explain Error Messages

```
Prompt: Explain this error and suggest how to fix it:

NullPointerException at
com.expense.service.ExpenseService.calculateTotal(ExpenseService.java:45)
```

Analyze Stack Traces

```
Prompt: Analyze this stack trace and identify the root cause:

#terminalLastCommand
```

Using Terminal Context

Select error output in terminal, then:

```
Prompt: #terminalSelection

What's causing this error and how do I fix it?
```

🛠️ The /fix Command

Basic Usage

1. Select problematic code
2. Use `/fix` command
3. Review suggested changes
4. Accept or modify

Example

```
// Buggy code - select this
public double calculateAverage(List<Double> values) {
    double sum = 0;
    for (Double value : values) {
        sum += value;
    }
    return sum / values.size(); // Potential division by zero
}
```

```
/fix Handle potential division by zero
```

Copilot suggests:

```
public double calculateAverage(List<Double> values) {
    if (values == null || values.isEmpty()) {
        return 0.0;
    }
    double sum = 0;
    for (Double value : values) {
        sum += value;
    }
    return sum / values.size();
}
```

🐛 Common Debugging Patterns

Pattern 1: Null Pointer Issues

```
Prompt: Review this code for potential null pointer exceptions and add
appropriate null checks:

#selection
```

Pattern 2: Logic Errors

```
Prompt: This method returns incorrect results. Expected: X, Actual: Y
Review the logic and identify the bug:

#file:ExpenseService.java
```

Pattern 3: Performance Issues

```
Prompt: This code is slow with large datasets. Identify performance
bottlenecks and suggest optimizations:

#selection
```

Pattern 4: Concurrency Bugs

```
Prompt: Review this code for thread safety issues:

#selection
```

📊 Adding Debug Logging

Generate Logging Statements

```
Prompt: Add debug logging statements to trace the execution flow in this
method:
```

```
#selection
```

Example output:

```
public Expense processExpense(ExpenseDTO dto) {
    log.debug("Processing expense: {}", dto);

    Expense expense = mapper.toEntity(dto);
    log.debug("Mapped to entity: {}", expense);

    expense = repository.save(expense);
    log.debug("Saved expense with ID: {}", expense.getId());

    return expense;
}
```

🧪 Generate Test Cases for Bugs

Reproduce Bug with Test

```
Prompt: Generate a unit test that reproduces this bug:

When expense amount is negative, the total calculation returns wrong
result.

#file:ExpenseService.java
```

Edge Case Testing

```
Prompt: Generate test cases for edge cases in this method:
- Empty input
- Null values
- Boundary conditions

#selection
```

💡 Debugging Workflow

Step 1: Understand the Error

```
/explain what does this error mean?
```

Step 2: Locate the Issue

```
@workspace where is this variable initialized?
```

Step 3: Analyze the Code

```
Review #file:ExpenseService.java for potential issues with calculateTotal
method
```

Step 4: Fix the Issue

```
/fix
```

Step 5: Verify the Fix

```
Generate a test case to verify this fix works correctly
```

🔧 Practical Exercises

Exercise 1: Fix a NullPointerException

1. Introduce a null bug in your code
2. Run and capture the error
3. Use `#terminalLastCommand` to get Copilot's analysis
4. Apply the suggested fix

Exercise 2: Debug Logic Error

1. Create a method with intentional logic error
2. Describe expected vs actual behavior to Copilot
3. Let Copilot identify and fix the issue

Exercise 3: Add Comprehensive Logging

1. Select a complex method
2. Ask Copilot to add debug logging
3. Use logs to trace execution flow

✅ Best Practices

- **Provide Context**: Include error messages, expected behavior, and actual behavior
- **Use Terminal Integration**: `#terminalSelection` and `#terminalLastCommand` are powerful
- **Verify Fixes**: Always test Copilot's suggested fixes
- **Understand Before Applying**: Use `/explain` before `/fix` to learn
- **Add Tests**: Generate regression tests for fixed bugs

🔗 Related Resources

- Slash Commands - `/fix`, `/explain` commands
- Hash Context Variables - Terminal context variables
- Copilot Limitations - Understanding AI limitations in debugging