# Chapter 13. Debugging and Testing

Despite developers’ best efforts, no code is ever perfect. You will inevitably introduce a bug that impacts the production behavior of your application or causes an end user distress when something doesn’t operate as expected.

Properly handling errors within your application is critical.[1] However, not every error your application throws is expected—or even catchable. In these circumstances, you must understand how to properly *debug* your application—how to track down the offending line of code so it can be fixed.

Among the first steps any PHP engineer uses to debug their code is the `echo` statement. Without a formal debugger, it’s common to see development code littered with `echo "Here!";` statements so the team can track where things might be broken.

The Laravel framework has made similar functionality popular and easily accessible while working on new projects by exposing a function called `dd()` (short for “dump and die”). This function is actually provided by the Symfony `var-dumper` module and works effectively in both PHP’s native command-line interface and when leveraging an interactive debugger. The function itself is defined as follows:

```
function dd(...$vars): void
{
    if (!in_array(\PHP_SAPI, ['cli', 'phpdbg'], true) &
        header('HTTP/1.1 500 Internal Server Error');
    }

    foreach ($vars as $v) {
        VarDumper::dump($v);
    }

    exit(1);
}
```

The preceding function, when used in a Laravel application, will print the contents of any variable you pass it to the screen and then halt the program's execution immediately. Like using `echo`, it's not the most elegant way to debug an application. However, it is fast, reliable, and a common way developers will debug a system in a hurry.

One of the best ways to preemptively debug your code is through unit testing. By breaking your code down into the smallest units of logic, you can write additional code that automatically tests and verifies the functionality of those units of logic. You then wire these tests to your integration and deployment pipeline and can ensure that nothing has broken in your application prior to deployment.

The open source PHPUnit project makes it simple and straightforward to instrument your entire application and automatically test its behavior. All of your tests are written in PHP, load your application's functions and classes directly, and explicitly document the correct behavior of the application.

---

**NOTE**

An alternative to PHPUnit is the open source Behat library. Whereas PHPUnit focuses on test-driven development (TDD), Behat focuses on an alternative *behavior*-driven development (BDD) paradigm. Both are equally useful for testing your code, and your team should choose which approach to take. PHPUnit is a more established project, though, and will be referenced throughout this chapter.

---

Hands down, the best way to debug your code is to use an interactive debugger. Xdebug is a debugging extension for PHP that improves error handling, supports tracing or profiling an application's behavior, and integrates with testing tools like PHPUnit to illustrate test coverage of application code. More importantly, Xdebug also supports interactive, step-through debugging of your application.

Armed with Xdebug and a compatible IDE, you can place flags in your code called *breakpoints*. When the application is running and hits these breakpoints, it pauses execution and allows you to interactively inspect the state of the application. This means you can view all variables in scope, where they came from, and continue executing the program one command at a time in your hunt for bugs. It is by far the most powerful tool in your arsenal as a PHP developer!

The following recipes cover the basics of debugging PHP applications. You will learn how to set up interactive debugging, capture errors, properly test your code to prevent regressions, and quickly identify when and where a breaking change has been introduced.

## 13.1 Using a Debugger Extension

### Problem

You want to leverage a robust, external debugger to inspect and manage your application so you can identify, profile, and eliminate errors in business logic.

### Solution

Install Xdebug, an open source debugging extension for PHP. Xdebug can be installed directly on Linux operating systems by using the default package manager. On Ubuntu, for example, install Xdebug by using `apt`:

```
$ sudo apt install php-xdebug
```

As package managers can sometimes install an outdated version of the project, you can also install it directly with the PECL extension manager:

```
$ pecl install xdebug
```

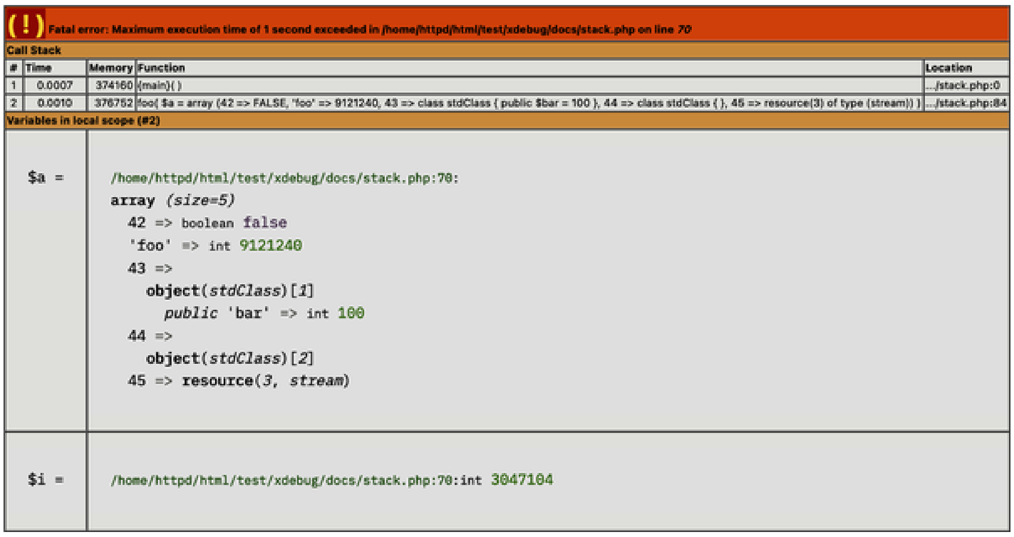
Once Xdebug is live on your system, it will embellish error pages for you automatically, presenting rich stack traces and debugging information to make it easier to identify errors when things go wrong.

### Discussion

Xdebug is a powerful extension for PHP. It empowers you to fully test, profile, and debug your applications in effective ways the language does not support natively. One of the most useful features you get by default with no additional configuration is a vast improvement to error reporting.

By default, Xdebug will automatically capture any errors thrown by your application and expose additional information about the following:

- The call stack (as illustrated in Figure 13-1), including both timing and memory utilization data. This helps you identify exactly when the program failed and where in code the function calls were occurring.
- Variables from the local scope so you don't need to guess what data was in memory when the error was thrown.



( ! ) Fatal error: Maximum execution time of 1 second exceeded in /home/httpd/html/test/xdebug/docs/stack.php on line *70*

Call Stack

| # | Time | Memory | Function | Location |
|---|---|---|---|---|
| 1 | 0.0007 | 374160 | {main}( ) | .../stack.php:0 |
| 2 | 0.0010 | 376752 | foo( $a = array (42 => FALSE, 'foo' => 9121240, 43 => class stdClass { public $bar = 100 }, 44 => class stdClass { }, 45 => resource(3) of type (stream)) ) | .../stack.php:84 |

Variables in local scope (#2)

```
$a =    /home/httpd/html/test/xdebug/docs/stack.php:70:
        array (size=5)
          42 => boolean false
          'foo' => int 9121240
          43 =>
            object(stdClass)[1]
              public 'bar' => int 100
          44 =>
            object(stdClass)[2]
          45 => resource(3, stream)

$i =    /home/httpd/html/test/xdebug/docs/stack.php:70:int 3047104
```

Figure 13-1. Xdebug enriches and formats the information presented when errors occur

Advanced integrations with tools like Webgrind also allow you to dynamically profile the performance of your application. Xdebug will (optionally) record the execution time of every function invocation and record both that time and the "cost" of a function call to disk. The Webgrind application then presents a handy visualization to help you identify bottlenecks in your code to optimize the program as necessary.

You can even pair Xdebug directly with your development environment for step-through debugging. By pairing your environment (e.g., Visual Studio Code) with an Xdebug configuration, you can place breakpoints in your code and literally pause execution when the PHP interpreter hits those points.

**NOTE**

The PHP Debug extension makes integration between Xdebug and Visual Studio Code incredibly straightforward. It adds all of the additional interfaces you'd expect directly to your IDE, in terms of both breakpoints and environment introspection. It's also maintained directly by the Xdebug community, so you can be sure it's in sync with the overall project.

While debugging in step-through mode, your application will pause on a breakpoint and give you direct access to all variables within the program scope. You can both inspect *and modify* these variables to test your environment. Further, while paused in a breakpoint, you have full console access to the application to further identify what might be going on. The call stack is directly exposed, so you can dive deep into which function or method object has led to the breakpoint and make changes where necessary.

While in a breakpoint, you can either step through the program one line at a time or opt to “continue” execution either until the next breakpoint or until the first error thrown by the program. Breakpoints can also be disabled without removing them from the IDE, so you can continue execution as necessary but revisit particular trouble spots later on demand.

---

WARNING

Xdebug is an immensely powerful development tool for any PHP development team. However, it has been known to add significant performance overhead to even the smallest application. Ensure that you are only ever enabling this extension for local development or in protected environments with test deployments. Never deploy your application to production with Xdebug installed!

---

### See Also

Documentation and home page for Xdebug.

## 13.2 Writing a Unit Test

### Problem

You want to validate the behavior of a piece of code to ensure that future refactoring doesn’t change the functionality of your application.

### Solution

Write a class that extends PHPUnit’s `TestCase` and explicitly tests the behavior of the application. For example, if your function is intended to extract a domain name from an email address, you would define it as follows:

```
function extractDomain(string $email): string
{
    $parts = explode('@', $email);

    return $parts[1];
}
```

Then create a class to test and validate the functionality of this code. Such a test would look like the following:

```
use PHPUnit\Framework\TestCase;

final class FunctionTest extends TestCase
{
    public function testSimpleDomainExtraction()
    {
        $this->assertEquals('example.com', extractDomai
    }
}
```

## Discussion

The most important aspect of PHPUnit is how you organize your project. First of all, the project needs to leverage Composer for autoloading both your application code and for loading any dependencies (including PHPUnit itself).[2] Typically, you will place your application code in a *src/* directory within the root of your project, and all of your test code will live in a *tests/* directory alongside it.

In the Solution example, you would place your `extractDomain()` function in *src/functions.php* and the `FunctionTest` class in *tests/FunctionTest.php*. Assuming autoloading is properly configured via Composer, you would then run the test using PHPUnit's bundled command-line tool as follows:

```
$ ./vendor/bin/phpunit tests
```

The preceding command will, by default, automatically identify and run every test class defined in your *tests/* directory via PHPUnit. To more comprehensively control the way PHPUnit runs, you can leverage a local configuration file to describe test suites, file allow lists, and configure any specific environment variables needed during testing.

The XML-based configuration is not often used except for complex or complicated projects, but the project documentation details at length how to configure it. A basic *phpunit.xml* file usable with this recipe or other similarly simple projects would look something like Example 13-1.

**Example 13-1. Basic PHPUnit XML configuration**

```xml
<?xml version="1.0" encoding="UTF-8"?>

<phpunit bootstrap="vendor/autoload.php"
         backupGlobals="false"
         backupStaticAttributes="false"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false">

  <coverage>
    <include>
      <directory suffix=".php">src/</directory>
    </include>
  </coverage>

  <testsuites>
    <testsuite name="unit">
      <directory>tests</directory>
    </testsuite>
  </testsuites>

  <php>
    <env name="APP_ENV" value="testing"/>
  </php>

</phpunit>
```

Armed with the preceding *phpunit.xml* file in your project, you merely need to invoke PHPUnit itself to run your tests. You no longer need to specify the *tests/* directory, as that's now provided by the `testsuite` definition in the application configuration.

Similarly, you can also specify *multiple* test suites for different scenarios. Perhaps one set of tests is built by your development team proactively as they're writing code (*unit* in the preceding example). Another set of tests might be written by your quality assurance (QA) team to replicate user-reported bugs (*regressions*). The advantage of the second test suite is that you can then refactor your application until the tests pass (i.e., the bugs are fixed) while also ensuring that you haven't modified the overall behavior of your application.

You can also ensure that old bugs don't reappear down the line!

In addition, you can choose which test suite runs at which time by passing the optional `--testsuite` flag to PHPUnit when it's run. Most tests are going to be fast, meaning they can be run frequently without costing your development team any additional time. Fast tests should be run as frequently as possible during development to ensure that your code is working and that no new (or old) bugs have crept into the codebase. At times, though, you might need to write a test that is too costly to run very often. These tests should be kept in a separate test suite so you can test around them. The tests remain and can be used before a deployment but won't slow down day-to-day development when standard tests run frequently.

Function tests, like that in the Solution example, are quite simple. Object tests are similar in that you are instantiating an object within a test and exercising its methods. The hardest part, however, is simulating multiple possible inputs to a particular function or method. PHPUnit solves this with data providers.

For a simple example, consider the `add()` function in Example 13-2. This function explicitly uses loose typing to add two values (regardless of their types) together.

**Example 13-2. Simple addition function**

```
function add($a, $b): mixed
{
    return $a + $b;
}
```

Since the parameters in the preceding function can be of different types ( `int` / `int` , `int` / `float` , `string` / `float` , etc.), you should test the various combinations to ensure that nothing breaks. Such a test structure would look like the class in Example 13-3.

**Example 13-3. Simple test of PHP addition**

```
final class FunctionTest extends TestCase
{
    // ...

    /**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected): void
    {
        $this->assertSame($expected, add($a, $b));
    }

    public function additionProvider(): array
    {
        return [
            [2, 3, 5],
            [2, 3.0, 5.0],
            [2.0, '3', 5.0],
            ['2', 3, 5]
        ];
    }
}
```

The `@dataProvider` annotation tells PHPUnit the name of a function within the test's class that should be used to provide data for testing. Rather than writing four separate tests, you've now provided PHPUnit with the ability to run a single test four times with differing inputs and expected outputs. The end result is the same—four separate tests of your `add()` function—but without the need to explicitly write those extra tests.

Given the structure of the `add()` function defined in Example 13-2, you might run afoul of certain type restrictions in PHP. While it's possible to pass numeric strings into the function (they're cast to numeric values before addition), passing non-numeric data will result in a PHP warning. In a world where user input is passed to this function, that kind of issue can and will come up. It’s best to protect against it by explicitly checking the input values with `is_numeric()` and throwing a known exception that can be caught elsewhere.

To accomplish this, first write a new test to *expect* that exception and validate that it’s thrown appropriately. Such a test would look like Example 13-4.

**Example 13-4. Testing the expected presence of exceptions in your code**

```
final class FunctionTest extends TestCase
{
    // ...

    /**
     * @dataProvider invalidAdditionProvider
     */
    public function testInvalidInput($a, $b, $expected)
    {
        $this->expectException(InvalidArgumentExceptior
        add($a, $b);
    }

    public function invalidAdditionProvider(): array
    {
        return [
            [1, 'invalid', null],
            ['invalid', 1, null],
            ['invalid', 'invalid', null]
        ];
    }
}
```

WARNING

Writing tests before changing your code is valuable because it gives you a precise target to achieve while refactoring. However, this new test will fail until you make the changes to the application code. Take care not to commit failing tests to your project’s version control or you will compromise your team’s ability to practice continuous integration!

With the preceding test in place, the test suite now fails as the function does not match the documented or expected behavior. Take time to add the appropriate `is_numeric()` checks to the function as follows:

```php
function add($a, $b): mixed
{
    if (!is_numeric($a) || !is_numeric($b)) {
        throw new InvalidArgumentException('Input must
    }

    return $a + $b;
}
```

Unit tests are an effective way to document the expected and appropriate behavior of your application since they are executable code that also validates that the application is functioning properly. You can test both success *and* failure conditions and even go so far as to mock various dependencies within your code.

The PHPUnit project also provides the ability to proactively identify the percentage of your application code that is covered by unit tests. A higher percentage of coverage is not a guarantee against bugs but is a reliable way to ensure that bugs can be found and corrected quickly with minimal impact to end users.

### See Also

Documentation on how to leverage PHPUnit.

## 13.3 Automating Unit Tests

### Problem

You want your project’s unit tests to run frequently, without user interaction, before any changes to the codebase are committed to version control.

## Solution

Leverage a Git commit hook to automatically run your unit tests *before* a commit is made locally. For example, the `pre-commit` hook in Example 13-5 will automatically run PHPUnit every time the user runs `git commit` but before any data is actually written to the repository.

**Example 13-5. Simple Git `pre-commit` hook for PHPUnit**

```php
#!/usr/bin/env php
<?php

echo "Running tests.. ";
exec('vendor/bin/phpunit', $output, $returnCode);

if ($returnCode !== 0) {
  echo PHP_EOL . implode($output, PHP_EOL) . PHP_EOL;
  echo "Aborting commit.." . PHP_EOL;
  exit(1);
}

echo array_pop($output) . PHP_EOL;

exit(0);
```

## Discussion

Git is by far the most popular distributed version control system available and is also the system used by the core PHP development team. It's open source and highly flexible both in how it hosts repositories and how you can customize workflows and project structures to fit your development cycle.

Specifically, Git allows customization by way of hooks. Your hooks live in the *.git/hooks* directory within your project alongside other information Git uses to track the state of your project itself. By default, even an empty Git repository includes several sample hooks, as shown in Figure 13-2.

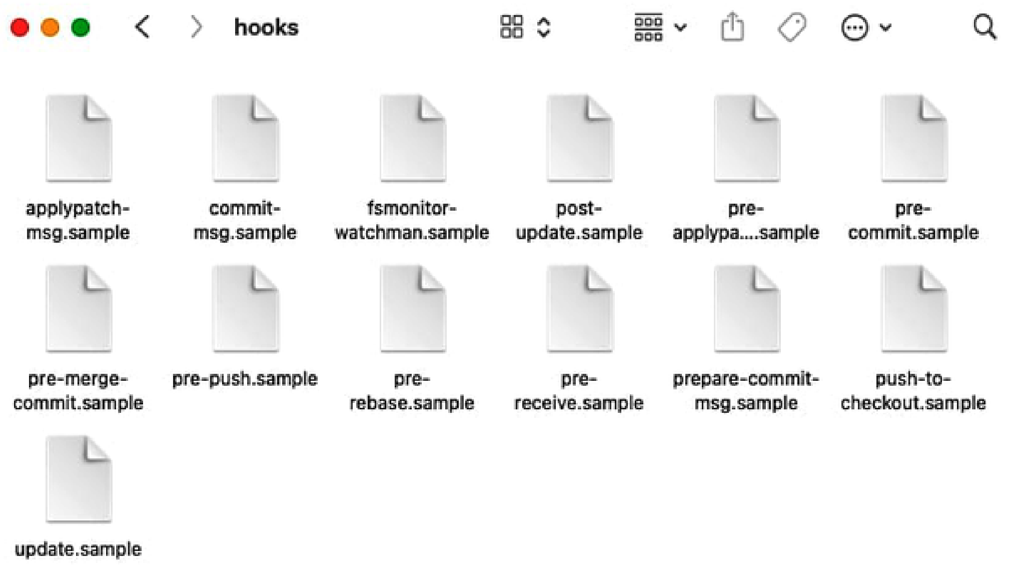


Figure 13-2. Git initializes even an empty repository with sample hooks

Each of the sample hooks is postfixed with a `.sample` extension that will disable it by default. If the sample hooks are ever something you do want to use, merely remove that extension, and the hook will run on that action.

In the case of automated testing, you want the `pre-commit` hook explicitly and should create a file with that name containing the contents of Example 13-5. With the hook in place, Git will always run this script before it commits the code.

The `0` exit status at the end of the script tells Git everything is OK and it can continue with the commit. Should any of your unit tests fail, the `1` exit status will flag that something went wrong, and the commit will abort without modifying your repository.

If you are absolutely sure you know what you're doing and need to override the hook for any reason, you can bypass the hook by adding the `--no-verify` flag when committing your code.

**WARNING**

The `pre-commit` hook is run entirely on the client side and lives outside your code repository. Every developer will need to install the hook individually. Aside from team guidelines or company policy, there isn't an effective way to enforce that the hook is being used (or that someone isn't bypassing it with `--no-verify`).

If your team is using Git for version control, there's a very good chance you're also using GitHub to host a version of your repository. If so, you can leverage GitHub Actions to run PHPUnit tests on GitHub's server as part of your integration and deployment pipeline.

Running tests locally helps protect against accidentally committing a *regression* (code that reintroduces a known bug) or other error to the repository. Running the same tests in the cloud provides even greater functionality as you can do so across a matrix of potential configurations. Developers will typically run only a single version of PHP locally, but you can run your application code and tests in containers on the server that are leveraging various versions of PHP or even different dependency versions.

Using GitHub Actions to run tests also provides the following benefits:

- If a new developer hasn't yet set up their Git `pre-commit` hook and commits broken code, the Action runner will immediately flag the commit as broken and prevent that developer from accidentally releasing a bug into production.
- Using a deterministic environment in the cloud protects your team against "well, it worked on my machine" issues, where code works in one local environment but then fails in a production environment that has a different configuration.
- Your integration and deployment workflow should build new deployment artifacts from every commit. Wiring this build process to your tests ensures that every build artifact is free from known defects and is, in fact, deployable.

### See Also

Documentation on customizing Git by using hooks.

## 13.4 Using Static Code Analysis

### Problem

You want to leverage an external tool to ensure that your code is free from as many errors as possible before it ever runs.

## Solution

Use a static code analysis tool like PHPStan.

## Discussion

PHPStan is a static code analysis tool for PHP that helps minimize errors in production code by flagging them for correction long before you've shipped your application. It's best when used with strict typing and helps your team write more manageable and understandable applications.[3]

Like many other development tools, PHPStan can be installed into your project via Composer with the following:

```
$ composer require --dev phpstan/phpstan
```

You can then run PHPStan against your project, analyzing both your application code and your tests directly. For example:

```
$ ./vendor/bin/phpstan analyze src tests
```

By default, PHPStan runs at level 0, which is the absolute loosest level of static analysis possible. You can specify a higher level of scanning by passing the `--level` flag at the command line with a number greater than 0. Table 13-1 enumerates the various levels available. For well-maintained, strictly typed applications, a level 9 analysis is the best way to ensure quality code.

Table 13-1. PHPStan rule levels

| Level | Description |
|---|---|
| 0 | Basic checks for unknown classes, functions, or class methods. Will also check number of arguments in function calls and any variables that are never defined. |
| 1 | Checks for possibly undefined variables, unknown magic methods, and dynamic properties retrieved through magic getters. |

| Level | Description |
|---|---|
| 2 | Validates unknown methods on all expressions and validates functional documentation (docblocks in code). |
| 3 | Checks return types and property type assignment. |
| 4 | Checks for dead code (e.g., conditionals that are always false) and unreachable code paths. |
| 5 | Checks argument types. |
| 6 | Reports on missing type hints. |
| 7 | Reports partially incorrect union types.[a] |
| 8 | Checks for any method calls or property access on nullable types. |
| 9 | Strict checks on usage of `mixed` typing. |

[a] For examples of union types, see the discussion of Example 3-9.

Once you've run the analysis tool, you can work on updating your application to fix basic flaws and validation errors. You can also automate the use of static analysis, similarly to the way you automated testing in Recipe 13.3 to ensure that the team is running analyses (and fixing identified errors) regularly.

## See Also

The PHPStan project home page and documentation.

## 13.5 Logging Debugging Information

### Problem

You want to log information about your program when things go wrong so you can debug any potential errors later.

### Solution

Leverage the open source Monolog project to implement a comprehensive logging interface within your application. First install the package by using Composer as follows:

```
$ composer require monolog/monolog
```

Then wire the logger into your application so you can emit warnings and errors whenever necessary. For example:

```
use Monolog\Level;
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

$logPath  = getenv('LOG_PATH')  ?? '/var/log/php/error.
$logLevel = getenv('LOG_LEVEL') !== false
            ? Level::from(intval(getenv('LOG_LEVEL')))
            : Level::Warning;

$logger = new Logger('default');
$logger->pushHandler(new StreamHandler($logPath, $logLe

$log->warning('Hello!');
$log->error('World!');
```

### Discussion

The easiest way to log information from PHP is via its built-in `error_log()` function. This will log errors either to the server error log or to a flat file as configured in *php.ini*. The only problem is that the function explicitly logs *errors* in the application.

A result is that any content logged by `error_log()` is treated as an error by any system parsing the log file. This can make it difficult to disambiguate between true errors (e.g., user login failures) and messages logged for debugging purposes. The intermingling of true errors and debugging statements can make runtime configuration difficult, particularly when you want to turn off certain logging in various environments. A workaround is to wrap any calls to `error_log()` in a check for the current logging level as shown in Example 13-6.

**Example 13-6. Selectively logging errors with `error_log()`**

```
enum LogLevel: int
                          ❶

{
    case Debug   = 100;
    case Info    = 200;
    case Warning = 300;
    case Error   = 400;
}

$logLevel = getenv('LOG_LEVEL') !== false
                          ❷

            ? LogLevel::from(intval(getenv('LOG_LEVEL')
            : LogLevel::Debug;

// Some application code ...
if (user_session_expired()) {
    if ($logLevel >= LogLevel::Info) {
                          ❸

        error_log('User session expired. Logging out ..
    }

    logout();
    exit;
}
```

❶ The easiest way to enumerate logging levels is with a literal `enum` type in PHP.

The log level should be retrievable from the system environment. If

The log level should be retrievable ❷ from the system environment. If it's not provided, then you should fall back on a sane, hardcoded default.

Whenever you invoke `error_log()`, you'll need to explicitly check the current logging level and decide whether or not to actually emit the error.

The problem with Example 13-6 isn't the use of an `enum`, nor is it the fact that you need to dynamically load the logging level from the environment. The problem is that you have to explicitly check the logging level prior to every invocation of `error_log()` to ensure that the program *actually should* emit an error. This frequent checking leads to a lot of spaghetti code and makes your application both less readable and more difficult to maintain.

A seasoned developer will realize the perfect solution here would be to wrap all of the logging logic (including log level checking) in a functional interface to keep the application clean. That's absolutely the right approach, and the entire reason the Monolog package exists!

**NOTE**

While Monolog is a popular PHP package for application logging, it is not the only package available. Monolog implements PHP's standard Logger interface; any package implementing the same interface can be dropped into your application in place of Monolog to provide similar functionality.

Monolog is far more powerful than merely printing strings to an error log. It also supports channels, various handlers, processors, and logging levels.

When instantiating a new logger, you first define a channel for that object. This allows you to create multiple logging instances side by side, keep their contents separate, and even route them to different means of output. By default, a logger needs more than a channel to operate, so you must also push a handler onto the call stack.

A handler defines what Monolog should do with any message passed into a particular channel. It could route data to a file, store messages in a database, send errors via email, notify a team or channel on Slack of an issue, or even communicate with systems like RabbitMQ or Telegram.

**NOTE**

Monolog also supports different formatters that can be attached to various handlers. Each formatter defines how messages will be serialized and sent out to the defined handler—for example, as a one-line string, a JSON blob, or an Elasticsearch document. Unless you're working with a handler that requires data in a specific format, you'll likely be just fine with the default formatter.

A processor is an optional extra step that can add data to a message. For example, the IntrospectionProcessor will automatically add the line, file, class, and/or method from which the log call was made to the log itself. A basic Monolog setup to log to a flat file with introspection would look something like Example 13-7.

**Example 13-7. Monolog configuration with introspection**

```
use Monolog\Level;
use Monolog\Logger;
use Monolog\Handler\StreamHandler;
use Monolog\Processor\IntrospectionProcessor;

$logger = new Logger('default');
$logger->pushHandler(new StreamHandler('/var/log/app.lo
$logger->pushProcessor(new IntrospectionProcessor());

// ...

$logger->debug('Something happened ...');
```

The last line of Example 13-7 invokes your configured logger and sends a literal string through the processor to the handler you've wired in. In addition, you can optionally pass additional data about the execution context or the error itself in an array as an optional second parameter.

Even without the additional context, if this entire code block lives in a file called */src/app.php*, it will produce something resembling the following in the application log:

```
[2023-01-08T22:02:00.734710+00:00] default.DEBUG: Somet
```

```
[] {"file":"/src/app.php","line":15,"class":null,"call
```

All you needed to do was create a single line of text ( `Something happened ...` ), and Monolog automatically captured the event timestamp, the error level, and details about the call stack thanks to the registered processor. All of this information makes debugging and potential error correction that much easier for you and your development team.

Monolog also abstracts away the burden of checking the error level on each call. Instead, you define the error level at play in two locations:

- When registering a handler to the logger instance itself. Only errors of this error level or higher will be captured by the handler.
- When emitting a message to the logger channel, you explicitly identify the error level attributed to it. For example, `::debug()` sends a message with an explicit error level of `Debug` assigned to it.

Monolog supports the eight error levels listed in Table 13-2, all illustrated by the syslog protocol described by RFC 5424.

Table 13-2. Monolog error levels

| Error level | Logger method | Description |
|---|---|---|
| `Level::Debug` | `::debug()` | Detailed debugging information. |
| `Level::Info` | `::info()` | Normal events like SQL logs or information application events. |
| `Level::Notice` | `::notice()` | Normal events that have greater significance than informational messages. |
| `Level::Warning` | `::warning()` | Application warnings that could become |

| Error level | Logger method | Description |
|---|---|---|
| | | errors in the future if action is not taken. |
| `Level::Error` | `::error()` | Application errors requiring immediate attention. |
| `Level::Critical` | `::critical()` | Critical conditions impacting the operation of the application. For example, instability or lack of availability in a key component. |
| `Level::Alert` | `::alert()` | Immediate action is required because of a key system failure. In critical applications, this error level should page an on-call engineer. |
| `Level::Emergency` | `::emergency()` | The application is unusable. |

Through Monolog, you can intelligently wrap error messages in the appropriate logger method and determine when these errors are actually sent to a handler based on the error level used when creating the logger itself. If you instantiate a logger only for `Error` -level messages and above, any calls to `::debug()` will *not* result in a log. The ability to discretely control your log output in production versus development is vital to building a stable and well-logged application.

## See Also

Usage instructions for the Monolog package.

## 13.6 Dumping Variable Contents as Strings

### Problem

You want to inspect the contents of a complex variable.

### Solution

Use `var_dump()` to convert the variable into a human-readable format and print it to the current output stream (like the command-line console). For example:

```
$info = new stdClass;
$info->name = 'Book Reader';
$info->profession = 'PHP Developer';
$info->favorites = ['PHP', 'MySQL', 'Linux'];

var_dump($info);
```

The preceding code will print the following to the console when run in the CLI:

```
object(stdClass)#1 (3) {
  ["name"]=>
  string(11) "Book Reader"
  ["profession"]=>
  string(13) "PHP Developer"
  ["favorites"]=>
  array(3) {
    [0]=>
    string(3) "PHP"
    [1]=>
    string(5) "MySQL"
    [2]=>
    string(5) "Linux"
  }
}
```

## Discussion

Every form of data in PHP has some string representation. Objects can enumerate their types, fields, and methods. Arrays can enumerate their members. Scalar types can expose both their types and values. It's possible for developers to get at the inner contents of any variable in any of three slightly different but equally valuable ways.

First, `var_dump()` as used in the Solution example directly prints the contents of a variable to the console. This string representation details the types involved, the names of fields, and the value of interior members directly. It's useful as a quick way to inspect what lives within a variable, but it isn't much use beyond that.

---

**WARNING**

Take care to ensure that `var_dump()` doesn't make its way into production. This function does not escape data and could render unsanitized user input to your application's output, introducing a serious security vulnerability.[4]

---

More helpful is PHP's `var_export()` function. By default, it also prints the contents of any variable passed in, except the output format is itself executable PHP code. The same `$info` object from the Solution example would print as follows:

```
(object) array(
   'name' => 'Book Reader',
   'profession' => 'PHP Developer',
   'favorites' =>
  array (
    0 => 'PHP',
    1 => 'MySQL',
    2 => 'Linux',
  ),
)
```

Unlike `var_dump()`, `var_export()` accepts an optional second parameter that will instruct the function to *return* its output rather than print it to the screen. This results in a string literal that represents the contents of the variable being returned, which could then itself be stored elsewhere for future reference.

A third and final alternative is to use PHP's `print_r()` function. Like both of the preceding functions, it produces a human-readable representation of the variable's contents. As with `var_export()`, you can pass an optional second parameter to the variable to return its output rather than print it to the screen.

Unlike both of the preceding functions, though, not all typing information is exposed directly by `print_r()`. For example, the same `$info` object from the Solution example would print as follows:

```
stdClass Object
(
    [name] => Book Reader
    [profession] => PHP Developer
    [favorites] => Array
        (
            [0] => PHP
            [1] => MySQL
            [2] => Linux
        )
)
```

Each function displays a different amount of information pertaining to the variable in question. Which version works best for you depends on how exactly you intend to use the resulting information. In a debugging or logging context, the ability of `var_export()` and `print_r()` to return a string representation rather than printing directly to the console would be valuable, particularly when paired with a tool like Monolog as described in Recipe 13.5.

If you want to export variable contents in a way to easily reimport them into PHP directly, the executable output of `var_export()` would serve you best. If you're debugging variable contents and need deep typing and size information, the default output of `var_dump()` might be the most informative, even if it can't be directly exported as a string.

If you *do* need to leverage `var_dump()` and want to export its output as a string, you can leverage output buffering in PHP to do just that. Specifically, create an output buffer prior to invoking `var_dump()`, then store the contents of that buffer in a variable for future use, as shown in Example 13-8.

**Example 13-8. Output buffering to capture variable contents**

```
ob_start(); ❶

var_dump($info); ❷


$contents = ob_get_clean(); ❸
```

❶ Create an output buffer. Any code that prints to the console after this invocation will be captured by the buffer.

❷ Dump the contents of the variable in question to the console/buffer.

❸ Get the contents of the buffer and delete it afterwards.

The result of the preceding example code will be a string representation of the dumped contents of `$info` stored in `$contents` for future reference. Proceeding to dump the contents of `$contents` itself would yield the following:

```
string(244) "object(stdClass)#1 (3) {
  ["name"]=>
  string(11) "Book Reader"
  ["profession"]=>
  string(13) "PHP Developer"
  ["favorites"]=>
  array(3) {
    [0]=>
    string(3) "PHP"
    [1]=>
    string(5) "MySQL"
    [2]=>
    string(5) "Linux"
  }
}
"
```

### See Also

Documentation on `var_dump()` , `var_export()` , and `print_r()` .

## 13.7 Using the Built-in Web Server to Quickly Run an Application

### Problem

You want to launch a web application locally without configuring an actual web server like Apache or NGINX.

### Solution

Use PHP’s built-in web server to quickly launch a script such that it is accessible from a web browser. For example, if your application lives in a *public_html/* directory, launch the web server from that directory as follows:

```
$ cd ~/public_html
$ php -S localhost:8000
```

Then visit *http://localhost:8000* in your browser to view any file (static HTML, images, or even executable PHP) that resides within that directory.

### Discussion

The PHP CLI provides a built-in web server that makes it easy to test or demonstrate applications or scripts in a controlled, local environment. The CLI supports both running PHP scripts and returning static content from the request path.

Static content could include rendered HTML files or anything from the following standard MIME types/extensions:

```
.3gp, .apk, .avi, .bmp, .css, .csv, .doc, .docx, .flac,
.html, .ics, .jpe, .jpeg, .jpg, .js, .kml, .kmz, .m4a,
.mpg, .odp, .ods, .odt, .oga, .ogg, .ogv, .pdf, .png, .
```

```
.swf, .tar, .text, .tif, .txt, .wav, .webm, .wmv, .xls,
and .zip.
```

**WARNING**

The built-in web server is intended for development and debugging purposes. It should not be used in a production context. For production purposes, always leverage a fully fledged web server in front of PHP. Either NGINX or Apache in front of PHP-FPM are reasonable choices.

In addition, you can pass a particular script as a *router script* to the web server, resulting in PHP directing every request to that script. The advantage to such an approach is that it mimics the use of popular PHP frameworks that utilize routers. The disadvantage is that you need to manually handle routing for static assets.

In an Apache or NGINX environment, browser requests for images, documents, or other static content are served directly without invoking PHP. When leveraging the CLI web server, you must first check for these assets and return an explicit `false` in order for the development server to handle them properly.

A framework router script must then check to see if you're running in CLI mode and, if so, route content accordingly. For example:

```
if (php_sapi_name() === 'cli-server') {
    if (preg_match('/\.(?:png|jpg|jpeg|gif)$/', $_SERVE
        return false;
    }
}

// Continue router execution
```

The preceding *router.php* file could then be used to bootstrap a local web server as follows:

```
$ php -S localhost:8000 router.php
```

The development web server could be made accessible to any interface (available on the local network) by passing `0.0.0.0` instead of `localhost` when invoking it. However, remember that this server is not designed for production use and is not structured in a way to protect your application from abuse by bad actors. *Do not use this web server on a public network!*

### See Also

Documentation on PHP's built-in web server.

## 13.8 Using Unit Tests to Detect Regressions in a Version-Controlled Project with git-bisect

### Problem

You want to quickly identify which commit in a version-controlled application introduced a particular bug so you can fix it.

### Solution

Use `git bisect` to track down the first bad commit in your source tree, as follows:

1. Create a new branch on the project.
2. Write a failing unit test (a test that reproduces the bug currently but, if the bug were fixed, it would pass).
3. Commit that test to the new branch.
4. Leverage `git rebase` to move that commit that introduces your new test to an earlier point in the project's history.
5. Use `git bisect` from that earlier point in history to automatically run your unit tests on subsequent commits to find the first commit where the test failed.

Once you rebase your project's commit history, the hashes of all of your commits will change. Keep track of the *new* commit hash for your unit test so you can properly target `git bisect.` For example, assume this commit has a hash of `48cc8f0` after being moved in your commit history. In that case, as shown in Example 13-9, you would identify this commit as "good" and the `HEAD` (the latest commit) in the project as "bad."

**Example 13-9. Example `git bisect` navigation after rebasing a test case**

```
$ git bisect start
$ git bisect good 48cc8f0
                                         ❶

$ git bisect bad HEAD
                                         ❷

$ git bisect run vendor/bin/phpunit
                                         ❸
```

❶ You must tell Git the first good commit it needs to look at.

❷ Since you don't know for sure where the broken commit is, pass the `HEAD` constant and Git will look at every commit after the good one referenced earlier.

❸ Git can run a specific command for every suspect commit. In this case, run your test suite. Git will continue looking at your project commit history until it finds the first commit where the test suite fails.

Once Git has identified the first bad commit (e.g., `16c43d7`), use `git diff` to see what actually changed at that commit, as shown in Example 13-10.

**Example 13-10. Comparing a known-bad Git commit**

```
$ git diff 16c43d7 HEAD
```

Once you know what's broken, run `git bisect reset` to return your repository to normal operations. At this point, move back to your main branch (and possibly delete the test branch as well) so you can begin correcting the identified bug.

## Discussion

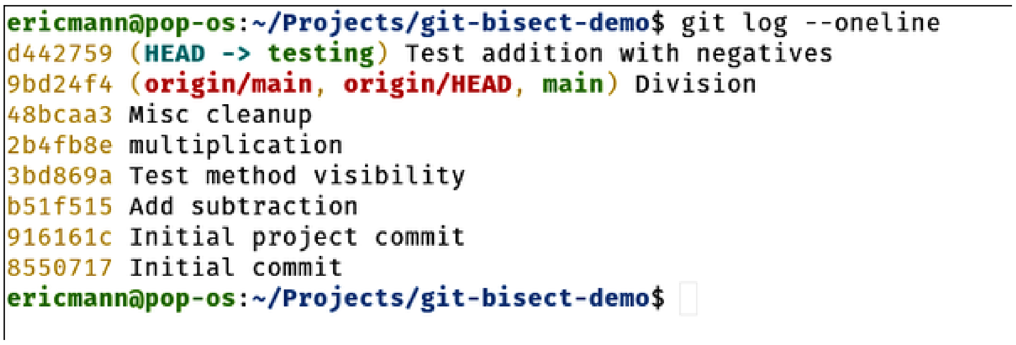
Git's bisect tool is a powerful way to track down and identify a bad commit to your project. It's particularly useful on larger, active projects where there might be several commits between a known-good and known-bad state. With larger projects, it's often cost prohibitive in terms of developer time to iterate through every commit to test its validity on an individual basis.

The `git bisect` command works with a binary search approach. It finds the commit at the midpoint between the known-good and known-bad ones and tests that commit. It then moves closer to the known-good or the known-bad commits based on the output of that test.

By default, `git bisect` expects you to manually test each suspect commit until it finds the "first bad" commit. However, the `git bisect run` subcommand empowers you to delegate this check to an automated system like PHPUnit. If the test command returns a default status of `0` (or success), the commit is assumed to be good. This works well because PHPUnit exits with an error code of `0` when all tests pass.

If the tests fail, PHPUnit returns an error code of `1`, which `git bisect` interprets as a bad commit. In this way, you can fully automate the detection of a bad commit over thousands of potential commits quickly and easily.

In the Solution example, you first created a new branch. This is merely to keep your project clean so you can throw any potential test commits away once you've identified the bad commit. On this branch, you committed a single test to replicate a bug identified in your project. Leveraging `git log`, you can quickly visualize the history of your project, including this test commit, as shown in Figure 13-3.



```
ericmann@pop-os:~/Projects/git-bisect-demo$ git log --oneline
d442759 (HEAD -> testing) Test addition with negatives
9bd24f4 (origin/main, origin/HEAD, main) Division
48bcaa3 Misc cleanup
2b4fb8e multiplication
3bd869a Test method visibility
b51f515 Add subtraction
916161c Initial project commit
8550717 Initial commit
ericmann@pop-os:~/Projects/git-bisect-demo$
```

Figure 13-3. Git log demonstrating a main branch and a testing branch with a single commit

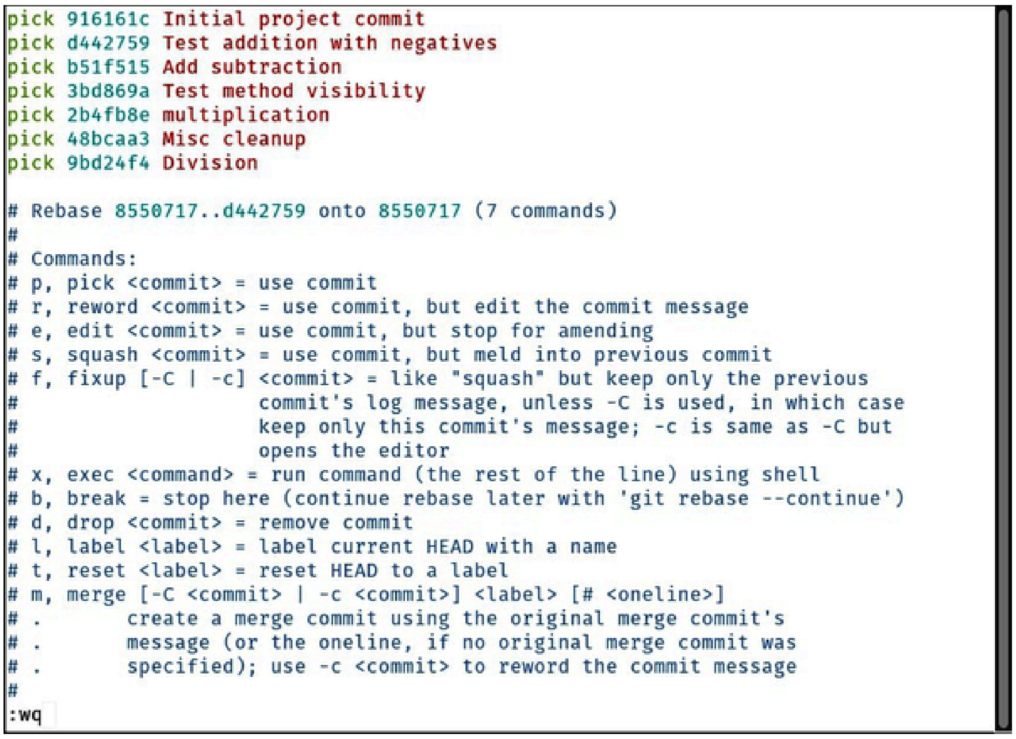
This log is useful as it provides you with the short hash for both your test commit and every other commit in the project. If you know a historical commit that is known to be good, you can rebase your project to move the test commit to *just after* that known commit.

In Figure 13-3, the test commit hash is `d442759`, and the last known "good" commit is `916161c`. To reorder your project, use `git rebase` interactively from the project's initial commit ( `8550717` ) to move the test commit earlier in the project. The exact command would start as shown in Example 13-11.

**Example 13-11. Interactive `git rebase` to reorder commits**

```
$ git rebase -i 8550717
```

Git will open a text editor and present the same SHA hashes for each possible commit. You want to retain your commit history (so keep the `pick` keywords in place), but move the test commit to just after the known-good commit, as shown in Figure 13-4.



```
pick 916161c Initial project commit
pick d442759 Test addition with negatives
pick b51f515 Add subtraction
pick 3bd869a Test method visibility
pick 2b4fb8e multiplication
pick 48bcaa3 Misc cleanup
pick 9bd24f4 Division

# Rebase 8550717..d442759 onto 8550717 (7 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified); use -c <commit> to reword the commit message
#
:wq
```

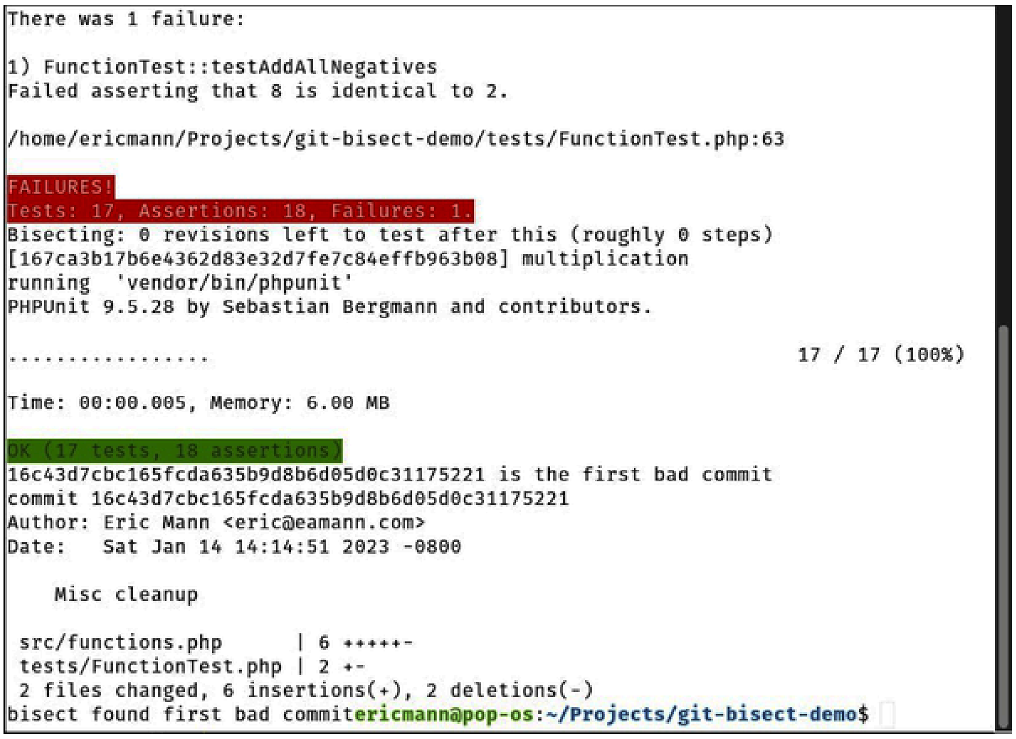
Figure 13-4. Interactive Git rebasing allows for modifying or reordering commits at will

Save the file, and Git will work at reconstructing your project history based on the moved commit. If and when there are conflicts, reconcile them locally first and commit the results. Then leverage `git rebase --continue` to keep moving. Once you're done, your project will be restructured such that the new test case appears immediately after the known-good commit.

**WARNING**

The known-good commit will have the same commit hash, as will any commits that came before it. Your moved commit and all the ones that follow, however, will have new commit hashes applied. Take care to ensure that you're using the correct commit hashes in any subsequent Git commands!

Once the rebase is complete, use `git log --oneline` to again visualize your commit history and reference the *new* commit attributed to your unit test. Then you can run `git bisect` from that commit to the `HEAD` of your project as you did in Example 13-9. Git will run PHPunit on each suspect commit until it finds the first "bad" commit, producing output similar to that in Figure 13-5.



```
There was 1 failure:

1) FunctionTest::testAddAllNegatives
Failed asserting that 8 is identical to 2.

/home/ericmann/Projects/git-bisect-demo/tests/FunctionTest.php:63

FAILURES!
Tests: 17, Assertions: 18, Failures: 1.
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[167ca3b17b6e4362d83e32d7fe7c84effb963b08] multiplication
running  'vendor/bin/phpunit'
PHPUnit 9.5.28 by Sebastian Bergmann and contributors.

.................                                      17 / 17 (100%)

Time: 00:00.005, Memory: 6.00 MB

OK (17 tests, 18 assertions)
16c43d7cbc165fcda635b9d8b6d05d0c31175221 is the first bad commit
commit 16c43d7cbc165fcda635b9d8b6d05d0c31175221
Author: Eric Mann <eric@eamann.com>
Date:   Sat Jan 14 14:14:51 2023 -0800

    Misc cleanup

 src/functions.php       | 6 +++++-
 tests/FunctionTest.php | 2 +-
 2 files changed, 6 insertions(+), 2 deletions(-)
bisect found first bad commitericmann@pop-os:~/Projects/git-bisect-demo$
```

Figure 13-5. Git bisect runs until it finds the first “bad” commit in the tree

Armed with knowledge of the first bad commit, you can view the diff at that point and see exactly where and how the bug crept into your project. At this point, return to the main branch and start prepping your fixes.

It’s a good idea to pull in your new unit test as well.

NOTE

While you could leverage `git rebase` again to move your test commit back to where it belongs, the rebase operation might still leave your project history modified from its former state. Instead, return to `main` and create a *new* branch for actually fixing the bug at that point. Pull in your test commit (perhaps via `git cherry-pick`) and make whatever changes are necessary.

## See Also

Documentation on `git bisect`.

1
For more on error handling, review Chapter 12.

2
For more on Composer, see Recipe 15.1.

3
Strict typing is discussed at length in Recipe 3.4.

4

For more on data sanitization, see Recipe 9.1.