# Chapter 14. Performance Tuning

Dynamically interpreted languages, such as PHP, are known for their flexibility and ease of use but not necessarily for their speed. This is partly because of the way their type systems work. When types are inferred at runtime, it's impossible for the parser to know exactly how to perform a certain operation until it's provided with data.

Consider the following, loosely typed PHP function to add two items together:

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

Since this function does not declare the types for the variables passed in or the return type, it can exhibit multiple signatures. All of the method signatures in Example 14-1 are equally valid ways to call the preceding function.

**Example 14-1. Various signatures for the same function definition**

```
add(int $a,    int $b):   int
                         ❶
add(float $a,  float $b): float
                         ❷
add(int $a,    float $b): float
                         ❸
add(float $a,  int $b):   float
                         ❹
add(string $a, int $b):   int
                         ❺
add(string $a, int $b):   float
                         ❻
```

`add(1, 2)` returns `int(3)` ❶

`add(1.  2.)` returns `float(3)`

`add(1., 2.)` returns `float(3)`

`add(1, 2.)` returns `float(3)`

`add(1., 2)` returns `float(3)`

`add("1", 2)` returns `int(3)`,

`add("1.", 2)` returns `float(3)`

The preceding example illustrates how you can write a single function and PHP internally needs to process it multiple ways. The language doesn't know which version of the function you actually need until it sees the data you provide it, and it will internally cast some values to other types where necessary. At runtime, though, the actual function is compiled to operation code (opcode) that runs on the processor through a dedicated virtual machine —and PHP will need to produce multiple versions of the same function's opcode to handle the differing input and return types.

**NOTE**

The loose typing system leveraged by PHP is one of the things that makes it so easy to learn but also so easy to make fatal programming mistakes. This book has taken care to leverage *strict typing* wherever possible to avoid these exact pitfalls. Review Recipe 3.4 for more on using strict typing in your own code.

With a compiled language, the problem expressed here about loose typing would be trivial—merely compile the program to the multiple forks of opcode and move on. Unfortunately, PHP is more of an interpreted language; it reloads and recompiles your script on demand, depending on how your application is loaded. Luckily, the performance drain of multiple code paths can be combatted with two modern features built into the very language itself: just-in-time (JIT) compilation and the caching of opcode.

## Just-In-Time Compilation

As of version 8.0, PHP ships with a JIT compiler that immediately enables faster program execution and better-performing applications. It does this by leveraging traces of actual instructions passed to the virtual machine (VM) handling the script's execution. When a particular trace is called frequently, PHP will automatically recognize the importance of the operation and gauge whether or not the code benefits from compilation.

Subsequent invocations of the same code will use the compiled byte code rather than the dynamic script, causing a significant performance boost. Based on metrics published by Zend when PHP 8.0 was released, the inclusion of a JIT compiler renders the PHP benchmark suite to be up to three times faster!

The point to remember is that JIT compilation primarily benefits low-level algorithms. This includes number crunching and raw data manipulation. Anything outside of CPU-intensive operations (e.g., graphics manipulation or heavy database integrations) won't see nearly as much benefit to these changes. However, knowing that the JIT compiler exists, you can take advantage of it and use PHP in new ways.

## Opcode Caching

Among the easiest ways to increase performance—indeed, the way the JIT compiler does so—is to cache expensive operations and reference the result rather than doing the operations again and again. Since version 5.5, PHP has shipped with an optional extension for caching precompiled byte code in memory called OPcache.[1]

Remember, PHP is primarily a dynamic script interpreter and will read in your scripts when the program starts. If you stop and start your application frequently, PHP will need to recompile your script to computer-readable byte code in order for the code to execute properly. Frequent starts/stops can force frequent recompiling of scripts and thus slow performance. The OPcache, however, allows you to selectively compile scripts to provide the byte code to PHP before the rest of the application runs. This removes the need for PHP to load and parse the scripts each time!

---

**NOTE**

The JIT compiler in PHP 8 and above can only be enabled if OPcache is also enabled on the server, since it uses the cache as its shared memory backend. However, you do not need to use the JIT compiler to use OPcache itself.

---

Both JIT compilation and opcode caching are low-level performance improvements to the language that you can easily leverage at runtime, but they're not the end of the story. It's also critical to understand how to time the execution of user-defined functions. This makes identifying bottlenecks in business logic relatively easy. Comprehensively benchmarking the application also helps gauge performance changes when deploying to new environments, on new releases of the language, or with updated dependencies down the road.

The following recipes describe both the timing/benchmarking of userland application code and how to leverage the language-level opcode cache to optimize the performance of your application and environment.

## 14.1 Timing Function Execution

### Problem

You want to understand how long it takes a particular function to execute in order to identify potential opportunities for optimization.

### Solution

Leverage PHP's built-in `hrtime()` function both before and after function execution to determine how long the function took to run. For example:

```php
$start = hrtime(true);

doSomethingComputationallyExpensive();

$totalTime = (hrtime(true) - $start) / 1e+9;

echo "Function took {$totalTime} seconds." . PHP_EOL;
```

### Discussion

The `hrtime()` function will return the system's built-in high-resolution time, counting from an arbitrary point in time defined by the system. By default, it returns an array of two integers—seconds and nanoseconds, respectively. Passing `true` to the function will instead return the total number of nanoseconds, requiring you to divide by `1e+9` to convert the raw output back to human-readable seconds.

A slightly fancier approach is to abstract the timing mechanism into a decorator object. As covered in Chapter 8, a decorator is a programming design pattern that allows you to extend the functionality of a single function call (or a whole class) by wrapping it in another class implementation. In this case, you want to trigger the use of `hrtime()` to time a function's execution without changing the function itself. The decorator in Example 14-2 would do exactly that.

**Example 14-2. A timed decorator object for measuring function call performance**

```
class TimerDecorator
{
    private int $calls = 0;
    private float $totalRuntime = 0.;

    public function __construct(public $callback, priva

    public function __invoke(...$args): mixed
                                       ❶

    {
        if (! is_callable($this->callback)) {
            throw new ValueError('Class does not wrap a
        }

        $this->calls += 1;
        $start = hrtime(true);
                                       ❷


        $value = call_user_func($this->callback, ...$ar
                                       ❸


        $totalTime = (hrtime(true) - $start) / 1e+9;
        $this->totalRuntime += $totalTime;

        if ($this->verbose) {
            echo "Function took {$totalTime} seconds."
                                       ❹

        }

        return $value;
                                       ❺
    }

    public function getMetrics(): array ❻
    {
        return [
            'calls'   => $this->calls,
            'runtime' => $this->totalRuntime,
            'avg'     => $this->totalRuntime / $this->c
        ];
    }
}
```

❶ The `__invoke()` magic method makes class instances callable as if they were functions. Using the `...` spread operator will capture any arguments passed in at runtime so they can be passed later to the wrapped method.

❷ The actual timing mechanism used by the decorator is the same as that in the Solution example.

❸ Assuming the wrapped function is callable, PHP will call the function and pass all necessary arguments thanks to the `...` spread operator.

❹ This implementation of the decorator can be instantiated with a verbosity flag that will also print runtimes to the console.

❺ Since the wrapped function might return data, you need to ensure that the decorator returns that output as well.

❻ As the decorated function is itself an object, you can directly expose additional properties and methods. In this case, the decorator keeps track of aggregate metrics that can be retrieved directly.

Assuming the same `doSomethingComputationallyExpensive()` function from the Solution example is the function you want to test, the preceding decorator can wrap the function and produce metrics as shown in Example 14-3.

**Example 14-3. Leveraging a decorator to time function execution**

```
$decorated = new TimerDecorator('doSomethingComputatio

$decorated(); ❶
```

```
var_dump($decorated->getMetrics());
```
❷

❶ Since the decorator class implements the `__invoke()` magic method, you can use an instance of the class as if it were a function itself.

❷ The resulting metrics array will include a count of invocations, the total runtime (in seconds) for all invocations, and the average runtime (in seconds) across all invocations.

Similarly, you can test the same wrapped function multiple times and pull aggregate runtime metrics from all invocations as follows:

```
$decorated = new TimerDecorator('doSomethingComputatior

for ($i = 0; $i < 10; $i++) {
    $decorated();
}

var_dump($decorated->getMetrics());
```

Since the `TimerDecorator` class can wrap any callable function, you can use it to decorate class methods just as easily as you can native functions. The class in Example 14-4 defines both a static and an instance method, either of which can be wrapped by a decorator.

**Example 14-4. Simple class definition for testing decorators**

```
class DecoratorFriendly
{
    public static function doSomething()
    {
        // ...
    }

    public function doSomethingElse()
    {
        // ...
        }
    }
```

Example 14-5 shows how class methods (both static and instance-bound) can be referred to as callables at runtime in PHP. Anything that can be expressed as a callable interface can be wrapped by a decorator.

**Example 14-5. Any callable interface can be wrapped by a decorator**

```
$decoratedStatic = new TimerDecorator(['DecoratorFrienc
                                  ❶

$decoratedStatic();
                                  ❷


var_dump($decoratedStatic->getMetrics());

$instance = new DecoratorFriendly();

$decoratedMember = new TimerDecorator([$instance, 'doSc
                                  ❸

$decoratedMember();
                                  ❹


var_dump($decoratedMember->getMetrics());
```

❶ A static class method is used as a callable by passing an array of both the names of the class and its static method.

❷ Once created, the decorated static method can be called as any other function would be and will produce metrics the same way.

❸ A method of a class instance is used as a callable by passing an array of the instantiated object and the string name of the method.

❹ Similar to the decorated static method, a decorated instance method can then be called as any other function would be to populate metrics within the decorator.

Once you know how long a function takes to run, you can focus on optimizing its execution. This might involve refactoring the logic or using an alternative approach to defining an algorithm.

The use of `hrtime()` originally required the HRTime extension to PHP but is now bundled as a core function by default. If you're using a version of PHP older than 7.3 or a prebuilt distribution where it was explicitly omitted, the function itself might be missing. In that event, you can either install the extension yourself via PECL or leverage the similar `microtime()` function instead.[2]

Rather than counting seconds from an arbitrary point in time, `microtime()` returns the number of microseconds since the Unix epoch. This function can be used in place of `hrtime()` to gauge function execution time as follows:

```php
$start = microtime(true);

doSomethingComputationallyExpensive();

$totalTime = microtime(true) - $start;

echo "Function took {$totalTime} seconds." . PHP_EOL;
```

Regardless of whether you use `hrtime()` as in the Solution example or `microtime()` as in the preceding snippet, ensure that you're consistent with the way you read out the resulting data. Both mechanisms return notions of time at different levels of precision, which could lead to confusion if you mix and match on any output formatting.

### See Also

PHP documentation on `hrtime()` and `microtime()`.

## 14.2 Benchmarking the Performance of an Application

### Problem

You want to benchmark the performance of your entire application so you can gauge changes (e.g., performance regressions) as the codebase, dependencies, and underlying language versions evolve.

## Solution

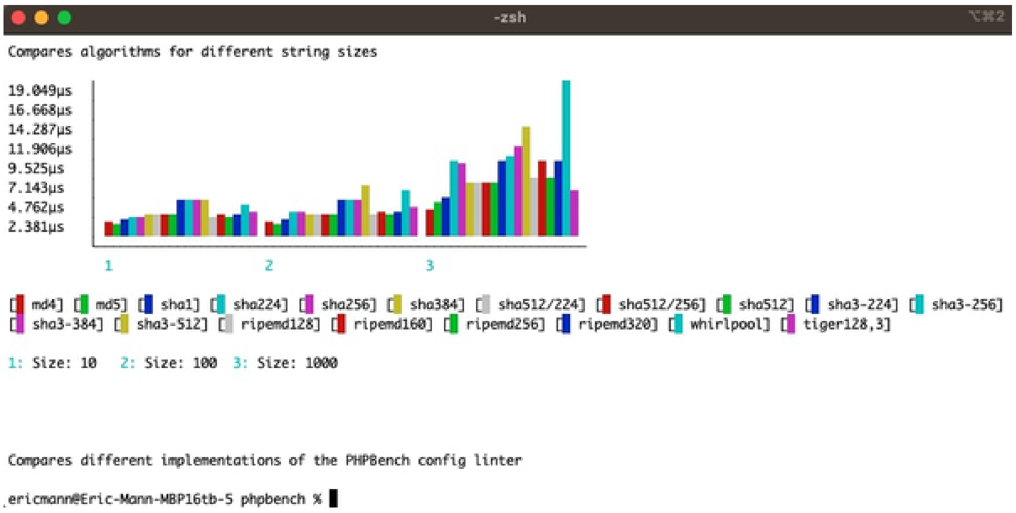
Leverage an automated tool like PHPBench to instrument your code and run regular performance benchmarks. For example, the following class is constructed to test the performance of all available hashing algorithms over various string sizes.[3]

```
/**
 * @BeforeMethods("setUp")
 */
class HashingBench
{
    private $string = '';

    public function setUp(array $params): void
    {
        $this->string = str_repeat('X', $params['size']
    }

    /**
     * @ParamProviders({
     *     "provideAlgos",
     *     "provideStringSize"
     * })
     */
    public function benchAlgos($params): void
    {
        hash($params['algo'], $this->string);
    }

    public function provideAlgos()
    {
        foreach (array_slice(hash_algos(), 0, 20) as $a
            yield ['algo' => $algo];
        }

    }

    public function provideStringSize() {
        yield ['size' => 10];
        yield ['size' => 100];
        yield ['size' => 1000];
    }
}
```

To run the default preceding example benchmark, first clone PHPBench, then install Composer dependencies, and finally run the following command:

```
$ ./bin/phpbench run --profile=examples --report=exampl
```

The resulting output, once the benchmark completes, will resemble the chart in Figure 14-1.



```
-zsh

Compares algorithms for different string sizes

19.049µs
16.668µs
14.287µs
11.906µs
9.525µs
7.143µs
4.762µs
2.381µs

1                    2                    3

[🟥 md4] [🟩 md5] [🟦 sha1] [🟦 sha224] [🟪 sha256] [🟨 sha384] [⬜ sha512/224] [🟥 sha512/256] [🟩 sha512] [🟦 sha3-224] [🟦 sha3-256]
[🟪 sha3-384] [🟨 sha3-512] [⬜ ripemd128] [🟥 ripemd160] [🟩 ripemd256] [🟦 ripemd320] [🟦 whirlpool] [🟪 tiger128,3]

1: Size: 10   2: Size: 100   3: Size: 1000


Compares different implementations of the PHPBench config linter

ericmann@Eric-Mann-MBP16tb-5 phpbench %
```

Figure 14-1. Output metrics from PHPBench’s example hashing benchmark

## Discussion

PHPBench is an effective way to gauge performance benchmarks of user-defined code in a variety of situations. It can be used in development environments to judge the performance level of new code, and it can also be integrated directly into continuous integration environments.

PHPBench’s own GitHub Actions configuration runs a full benchmarking suite of the application itself with every pull request and change. This allows the project maintainers to ensure that the project continues performing as expected with each change they introduce across a broad matrix of supported versions of PHP.

Any project aiming to include automated benchmarks must first start with Composer.[4] You need to leverage Composer autoloading so PHPBench knows where to grab classes from, but once that’s set up, you can build your project however you want.

Assume you're building a project that leverages value objects and hashing to protect sensitive data they store. Your initial *composer.json* file might look something like the following:

```
{
    "name": "phpcookbook/valueobjects",
    "require-dev": {
        "phpbench/phpbench": "^1.0"
    },
    "autoload": {
        "psr-4": {
            "Cookbook\\": "src/"
        }
    },
    "autoload-dev": {
        "psr-4": {
            "Cookbook\\Tests\\": "tests/"
        }
    },
    "minimum-stability": "dev",
    "prefer-stable": true
}
```

Naturally, your project code will live in a *src/* directory and any tests, benchmarking or otherwise, will live in a separate *tests/* directory. For the sake of benchmarking alone, you'll want to create a dedicated *tests/Benchmark/* directory to keep track of both namespaces and filterable code.

Your first class, the one you want to benchmark, is a value object that takes in an email address and can be easily manipulated as if it were a string. But when it dumps its contents to a debugging context, like `var_dump()` or `print_r()`, it automatically hashes the value.

---

**WARNING**

Email is a common enough format that even hashing the data won't be enough to protect it from a truly dedicated attacker. The illustrations in this recipe are meant to demonstrate how data can be *obfuscated* using a hash. This should not be considered a comprehensive security tutorial.

---

Create the class defined in Example 14-6 as *ProtectedString.php* in your new *src/* directory. This class has a lot to it—primarily several implemented magic methods to ensure that there is no way to *accidentally* serialize the object and get at its internal value. Instead, once you instantiate a `ProtectedString` object, the only way to get at its contents is with the `::getValue()` method. Anything else will return the SHA-256 hash of the contents.

**Example 14-6. Protected string wrapper class definition**

```
namespace Cookbook;

class ProtectedString implements \JsonSerializable
{
    protected bool $valid = true;

    public function __construct(protected ?string $valu

    public function getValue(): ?string
    {
        return $this->value;
    }

    public function equals(ProtectedString $other): boo
    {
        return $this->value === $other->getValue();
    }

    protected function redacted(): string
    {
        return hash('sha256', $this->value, false);
    }

    public function isValid(): bool
    {
        return $this->valid;
    }

    public function __serialize(): array
    {
        return [
            'value' => $this->redacted()
        ];
    }
```

```
        public function __unserialize(array $serialized): v
        {
            $this->value = null;
            $this->valid = false;
        }

        public function jsonSerialize(): mixed
        {
            return $this->redacted();
        }

        public function __toString()
        {
            return $this->redacted();
        }

        public function __debugInfo()
        {
            return [
                'valid' => $this->valid,
                'value' => $this->redacted()
            ];
        }
    }
```

You want to validate the performance of the chosen hashing algorithm. SHA-256 is more than reasonable, but you want to ensure that you benchmark all possible means of serialization for performance so that, if and when you need to change to a different hashing algorithm, you can ensure no performance regressions in the system.

To actually begin benchmarking this class, create the following *phpbench.json* file in the root of your project:

```
    {
        "$schema": "./vendor/phpbench/phpbench/phpbench.sch
        "runner.bootstrap": "vendor/autoload.php"
    }
```

Finally, create an actual benchmark to time the various ways a user can serialize a string. The benchmark defined in Example 14-7 should live in *tests/Benchmark/ProtectedStringBench.php*.

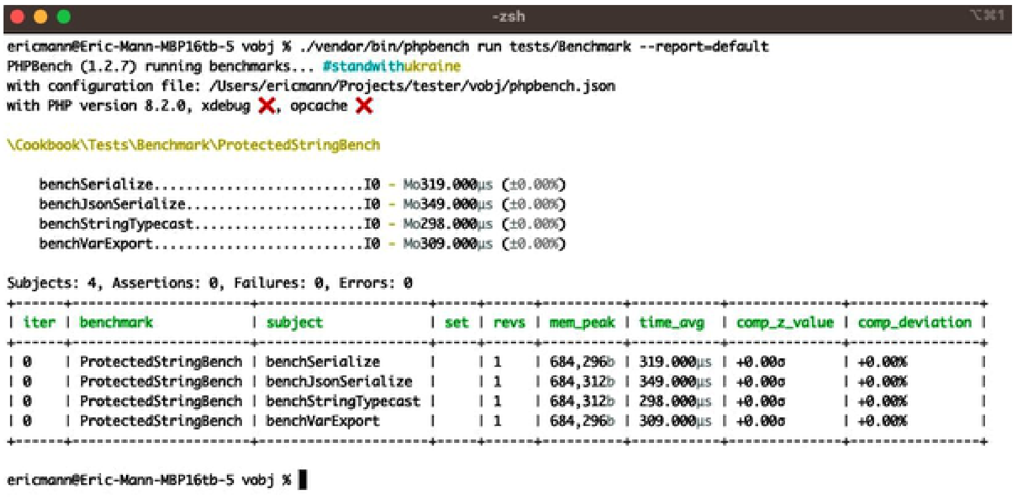
**Example 14-7. Benchmarking the `ProtectedString` class**

```
namespace Cookbook\Tests\Benchmark;

use Cookbook\ProtectedString;

class ProtectedStringBench
{
    public function benchSerialize()
    {
        $data = new ProtectedString('testValue');
        $serialized = serialize($data);
    }

    public function benchJsonSerialize()
    {
        $data = new ProtectedString('testValue');
        $serialized = json_encode($data);
    }

    public function benchStringTypecast()
    {
        $data = new ProtectedString('testValue');
        $serialized = '' . $data;
    }

    public function benchVarExport()
    {
        $data = new ProtectedString('testValue');
        ob_start();
        var_dump($data);
        $serialized = ob_end_clean();
    }
}
```

Finally, you can run your benchmarks with the following shell command:

```
$ ./vendor/bin/phpbench run tests/Benchmark --report=de
```

This command will produce an output similar to that in Figure 14-2, detailing the memory usage and runtime for each of your serialization operations.



```
ericmann@Eric-Mann-MBP16tb-5 vobj % ./vendor/bin/phpbench run tests/Benchmark --report=default
PHPBench (1.2.7) running benchmarks... #standwithukraine
with configuration file: /Users/ericmann/Projects/tester/vobj/phpbench.json
with PHP version 8.2.0, xdebug ❌, opcache ❌

\Cookbook\Tests\Benchmark\ProtectedStringBench

    benchSerialize..........................I0 - Mo319.000μs (±0.00%)
    benchJsonSerialize......................I0 - Mo349.000μs (±0.00%)
    benchStringTypecast.....................I0 - Mo298.000μs (±0.00%)
    benchVarExport..........................I0 - Mo309.000μs (±0.00%)

Subjects: 4, Assertions: 0, Failures: 0, Errors: 0
+------+---------------------+---------------------+-----+------+----------+-----------+--------------+----------------+
| iter | benchmark           | subject             | set | revs | mem_peak | time_avg  | comp_z_value | comp_deviation |
+------+---------------------+---------------------+-----+------+----------+-----------+--------------+----------------+
| 0    | ProtectedStringBench | benchSerialize      |     | 1    | 684,296b | 319.000μs | +0.00σ       | +0.00%         |
| 0    | ProtectedStringBench | benchJsonSerialize  |     | 1    | 684,312b | 349.000μs | +0.00σ       | +0.00%         |
| 0    | ProtectedStringBench | benchStringTypecast |     | 1    | 684,312b | 298.000μs | +0.00σ       | +0.00%         |
| 0    | ProtectedStringBench | benchVarExport      |     | 1    | 684,296b | 309.000μs | +0.00σ       | +0.00%         |
+------+---------------------+---------------------+-----+------+----------+-----------+--------------+----------------+

ericmann@Eric-Mann-MBP16tb-5 vobj %
```

Figure 14-2. Output of PHPBench for value object serialization with hashing

Every element of your application can, and should, have benchmarking built into it. This will drastically simplify testing the performance of your application in new environments—like on new server hardware or under a newly released version of PHP. Wherever possible, take time to wire these benchmarks into continuous integration runs as well to ensure that the tests are run and recorded as frequently as possible.

### See Also

Official documentation for the PHPBench project.

## 14.3 Accelerating an Application with an Opcode Cache

### Problem

You want to leverage opcode caching in your environment to improve the overall performance of your application.

### Solution

Install the shared OPcache extension and configure it in *php.ini* for your environment.[5] As it's a default extension, you merely need to update your configuration to enable caching. The following settings are generally recommended for solid performance but should be tested against your particular application and infrastructure:

```
opcache.memory_consumption=128
opcache.interned_strings_buffer=8
opcache.max_accelerated_files=4000
opcache.revalidate_freq=60
opcache.fast_shutdown=1
opcache.enable=1
opcache.enable_cli=1
```

## Discussion

When PHP is running, the interpreter reads your scripts and compiles your user-friendly PHP code into something that's easy for the machine to understand. Unfortunately, since PHP isn't a formally compiled language, it has to do this compilation every time a script loads. With a simple application, this isn't that much of an issue. With a complex application, it can lead to slow load times and high latency for repeated requests.

The easiest way to optimize an application around this particular issue is to cache the compiled byte code so it can be reused on subsequent requests.

To test and verify the functionality of the opcode cache locally, you can leverage `-d` flags at the command line when starting a script. The `-d` flag sets an explicit override for a configuration value otherwise set (or left to its default) by *php.ini*. Specifically, the command-line flags in Example 14-8 will leverage the local PHP development server to run an application with OPcache *disabled* entirely.

**Example 14-8. Launch a local PHP web server without OPcache support**

```
$ php -S localhost:8080 -t public/ -dopcache.enable_cli
```

Similarly, you can run almost exactly the same command with explicit *enabling* of the opcode cache to directly compare the behavior and performance of your application, as shown in Example 14-9.

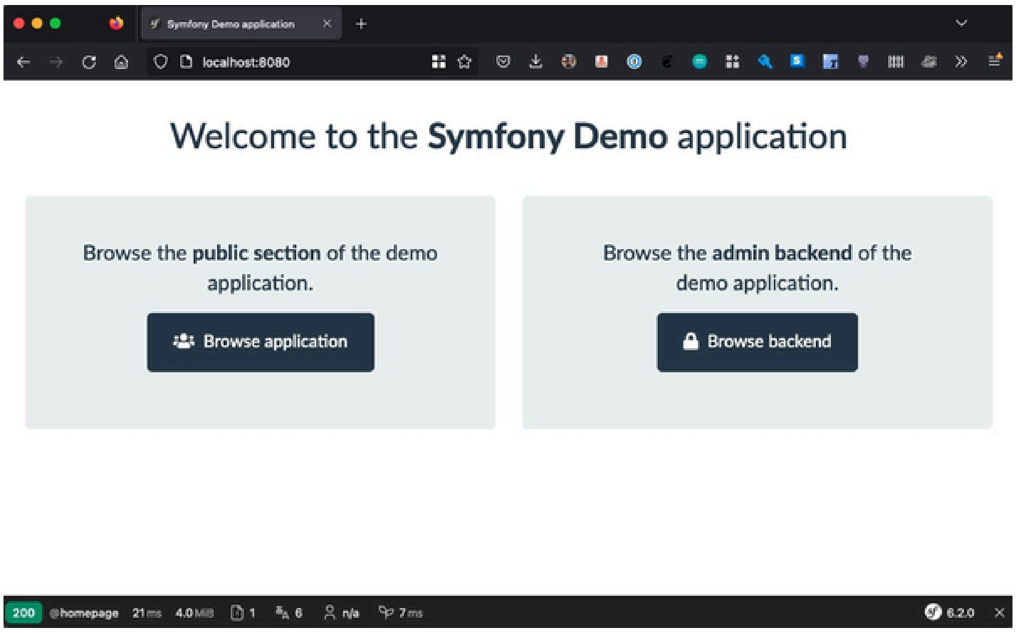
**Example 14-9. Launch a local PHP web server with OPcache support**

```
$ php -S localhost:8080 -t public/ -dopcache.enable_cli
```

To fully demonstrate how this works, take time to install a demo application using the open source Symfony framework. The following two commands will clone a demonstration application to the */demosite/* directory locally and use Composer to install required dependencies:

```
$ composer create-project symfony/symfony-demo demosite
$ cd demosite && composer install
```

Next, use the built-in PHP web server to launch the application itself. Use the command from Example 14-8 to start without opcache support. The application will be available on port 8080 and will look something like Figure 14-3.


Symfony Demo application
localhost:8080
Welcome to the Symfony Demo application
Browse the public section of the demo application.
Browse application
Browse the admin backend of the demo application.
Browse backend
200 @homepage 21 ms 4.0 MiB 1 6 n/a 7 ms
6.2.0
Figure 14-3. Loading page of the default Symfony demo application

The default application is running locally, using a lightweight SQLite database, so it should load fairly quickly. As shown in Example 14-10, you can effectively test the load time with a cURL command in your terminal.

**Example 14-10. Simple cURL command for gauging web application response time**

```
curl -s -w "\nLookup time:\t%{time_namelookup}\
    \nConnect time:\t%{time_connect}\
    \nPreXfer time:\t%{time_pretransfer}\
    \nStartXfer time:\t%{time_starttransfer}\
    \n\nTotal time:\t%{time_total}\n" -o /dev/null \
    http://localhost:8080
```

Without opcode caching enabled, the Symfony demo application loads with a total time of ~0.3677 seconds. This is remarkably fast but, again, the application is being run entirely within the local environment. In production with a remote database, it would likely be slower, but this is a solid baseline.

Now, stop the application and restart it *with* opcode caching enabled using the command defined in Example 14-9. Then rerun the cURL performance test from Example 14-10. With opcode caching, the application now loads with a total time of ~0.0371 seconds.

This is a relatively simple, default application, but a 10 times increase in performance is a massive boost to system performance. The faster an application loads, the more customers your system can service in the same period of time!

## See Also

PHP documentation on the OPcache extension.

[1] The newer JIT compiler released with PHP 8.0 uses OPcache under the hood, but you can still leverage caching *manually* to control the system even if JIT compilation is unavailable.

[2] For more on PECL and extension management, refer to Recipe 15.4.

[3] This particular example is taken from the example benchmarks that ship by default with PHPBench.

[4] For more on initializing a project with Composer, review Recipe 15.1.

[5] OPcache is a shared extension that will not exist if your PHP was compiled with the `--disable-all` flag to disable default extensions. In that case, you have no choice but to either recompile PHP with the `--enable-opcache` flag set or else install a fresh version of the PHP engine that was compiled with this flag set.