Testing Overview

To test this code, I would use a combination of unit tests, integration tests, and E2E tests, on the front and backend.

B.N.: While the application is simple at this time, it's important, as the app might grow or extend, to consider testing not just basic functionality, but potential open-ended user input. If open-ended input is an option, this is where edge cases/error-checking would come into being.

Unit Testing

Backend – PHPUnit

- Using PHPUnit Library, create a "Test" class for each Service and Controller that extends TestCase
- In each Test class, we can have a setup() function that instantiates a mockHandler and client to manage all tests for the class
- Create one test function for each functionality in services/controllers.
- Use specific naming conventions that get to precisely what is being tested
- E.g., in TestStationService.php, we would have a function called testProcessStationData (aligning with StationService.php's processStationData). By creating a mockResponse that mirrors the direct response from WMATA's Station List API (i.e., an array including lat, lon, address, etc), we can put that response through processStationData and perform several assertions such as assertIsArray, and assetEquals our expected result (an array with only code, name, and lines)
- This could also be used for Rate Limit testing and Error handling (e.g., if we test a response with code 429, we can test with expectException and expectExceptionMessage functions to verify that the correct error is coming through.)
- PHPUnit Example:

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class EmailTest extends TestCase
{
    public function testCanBeCreatedFromValidEmail(): void
    {
        $string = 'user@example.com';

        $email = Email::fromString($string);

        $this->assertSame($string, $email->asString());
    }
```

Frontend – Vitest

- Using the Vitest library, we can create simple test files with tests for each functionality
- For example, to test loadStationList(), we can create a test that calls the backend for data, and validates that after, stationList contains the expected array.
- Vitest example:

Integration Testing

Backend – Symfony's KernelTestCase Library

- Similar to Unit Testing, we can import this library to test services directly as mock instances

- KernelTestCase provides options to restart the kernel and access the service container anew for each test, resulting in independent testing
- After accessing a new $container, it is as simple as creating a reference to a service, calling a function, and asserting that the result is equal to the expected value
- This can be used the same functionality as unit testing (i.e., testing the return value of processStationData), but directly tests how Services are integrated into the rest of the application
- KernelTestCase Example:

```php
// tests/Service/NewsletterGeneratorTest.php
namespace App\Tests\Service;

use App\Service\NewsletterGenerator;
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;

class NewsletterGeneratorTest extends KernelTestCase
{
    public function testSomething(): void
    {
        // (1) boot the Symfony kernel
        self::bootKernel();

        // (2) use static::getContainer() to access the service container
        $container = static::getContainer();

        // (3) run some service & test the result
        $newsletterGenerator = $container->get(NewsletterGenerator::class);
        $newsletter = $newsletterGenerator->generateMonthlyNews(/* ... */);

        $this->assertEquals('...', $newsletter->getContent());
    }
}
```

E2E Testing

- The final piece of testing is to cover the entire application, E2E. This can be done by quite a few popular softwares, including Cypress and Selenium.
- Taking Cypress for example, we would be able to run the application through Cypress and perform a variety of tests.
- For this application, testing E2E would include verification such as
    - Are the station names contained within the page?
    - Can a user click on the station names?

    - After clicking, can the user see the new data on the next train's ETA?
- All of these are achievable via E2E and would provide a final step to a fully tested application
- One particular set of tests to consider is the variety of return types: as "GetPrediction"(next Train) returns values not just of integers, but also "BRD", "ARR", or even "---", it's important to test that these options are being correctly displayed to the user.

Manual Testing

- As for all applications, I would add manual testing to the formula. Regardless of what automated testing is used, developers and testers alike should manually go into the application and verify that all functionality is working correctly.