Laravel Package Development

Tinashe Musonza

Laravel Package Development

Tinashe Musonza

This book is for sale at http://leanpub.com/laravel-package-development

This version was published on 2022-01-11

Contents

Introduction

About this book

Who is this book for?

Anyone familiar with the Laravel framework or any PHP framework can make use of this book. Ever installed PHP composer packages or want to create Laravel packages through a Test Driven approach? This book provides that.

How to use this book

I will provide all the code used in this book, but it is beneficial to code along as you go through it to get the most out.

What are we building?

We are going to build a chat package by following a test-driven approach. This package will be installed in any Laravel application and adds the ability for your Laravel models to participate in conversations with each other. For instance, a bot model can chat with a user model and a client model, and so forth. There will be fewer limitations. We will also create a demo application to showcase our package and go over the process of distributing it to end-users.

Requirements

Composer

We are going to be using Composer to manage our PHP dependencies. If you are unfamiliar with Composer, it is a tool that enables you to declare the libraries that your project depends on, and it will install and update them for you. For getting started and installing this great tool, you can visit https://getcomposer.org/.

Git

We are going to be using Git as our version control system. Head over to https://git-scm.com/ for installation instructions and more.

Package setup

Project structure

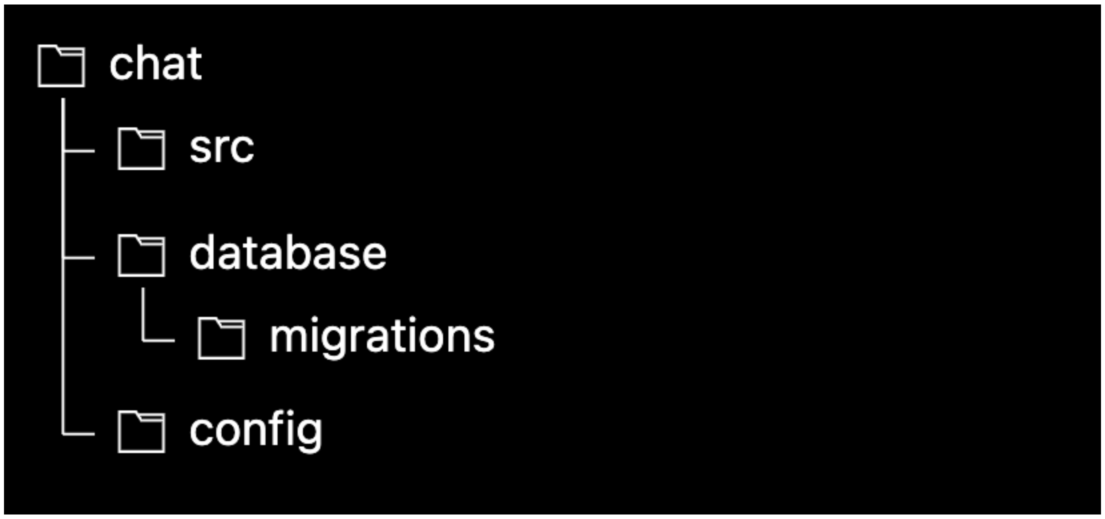
Here we are going to create a composer package. We need a unique identifier across Packagist and/or Github to prefix our package name for composer packages. I'm going to use the name **tashtin** for the vendor. In the end, our app will be referenced as **tashtin/chat** when we install it. While you can use a different name, I would recommend using the same vendor name to make it easier to follow along and debug. You can always search and replace your code when done.

Create a directory named **chat** anywhere on your computer that will hold your chat library. I normally create the libraries under a directory named after the vendor in use. For example, I will have a path like **/Development/tashtin/chat**

Inside the chat directory, create a **src** directory to hold the source code. Also, create a **database** directory at the root of the project, and inside that directory, create a **migrations** directory for database migrations. We will also create a **config** directory to store any configuration files.

```
$ mkdir -p src tests config database/migrations
```

We now have a basic skeleton for our package, and your directory structure should look like the following:



```
chat
├ src
├ database
│ └ migrations
└ config
```

Composer init

The next thing we need to do is initialize our composer package. Open the terminal and navigate to the chat directory.

Let's initialize our composer package by running the following command:

```
$ composer init
```

This command will help you craft your package's composer.json file. You will be prompted to answer a series of questions such as the following:

```
Package name (<vendor>/<name>) :  tashtin/chat
Description []:  Laravel chat package
Author [doe <doe@example.com>, n to skip]:
Would you like to define your dependencies (require) interactively [yes]? no
Would you like to define your dev dependencies (require-dev) interactively [yes]? **\
no**
```

You can skip any of the questions, and you will be able to make updates later. After finishing the setup, you will see a generated file **composer.json** in the directory similar to below:

```
{
  "name": "tasthin/chat",
  "description": "Chat Package for Laravel",
  "type": "library",
  "license": "MIT",
  "authors": [
    {
      "name": "doe",
      "email": "doe@example.com"
    }
  ],
  "minimum-stability": "dev",
  "require": {}
}
```

Composer require-dev

Dev dependencies are libraries that we will use to help us during development. These libraries are not necessary for our package to run in production. In other words, the libraries will only be installed at the root of the package to assist with development. Some of the libraries that we will use for development are PHPUnit and Orchestral Testbench. We will go over each dependency next.

orchestral/testbench

There are two ways you would normally develop Laravel packages. One way would be to create a laravel project and have a directory for all packages in development at the root of the project. Then use PSR-4 to load the package via Laravel project composer.json.
However, we will develop our package independent of any Laravel project and leverage Orchestral Testbench[1], a Laravel testing helper for package development. Orchestral Testbench will allow us to write our package tests as if they existed inside a typical Laravel application.

phpunit/phpunit

PHPUnit is a testing framework for PHP that we will use to write and run tests for our package. You can read more on PHPUnit at https://phpunit.de[2].

We may need additional dev dependencies later. In the meantime, let's install these two. At the time of this writing, I am using Laravel 6, so my library versions might differ from yours. Update your composer.json to the following:

composer.json

```
{
  "name": "tasthin/chat",
  "description": "Chat Package for Laravel",
  "type": "library",
  "license": "MIT",
  "authors": [
    {
      "name": "doe",
      "email": "doe@example.com"
    }
  ],
  "minimum-stability": "dev",
  "require": {},
  "require-dev": {
    "phpunit/phpunit": "^8.0",
    "orchestra/testbench": "^4.0",
    "orchestra/database": "^4.0"
  }
}
```

I have also added orchestra/database library, which will come in handy with our database configuration as if we were in a Laravel application.

[1]https://github.com/orchestral/testbench
[2]https://phpunit.de

Composer require

Another entry in our composer.json file is the `require` key field. This will hold the dependencies needed for our library to be functional in production or for our end users. At this time, we are developing a package for Laravel 6, which requires PHP 7.2 and above. So it makes sense to add the required dependencies in our composer.json.

Go ahead and update `"require": {}` in composer.json file to the following:

```
"require": {
    "php": "^7.2",
    "laravel/framework": "^6.0"
}
```

Now that we have our preliminary dependencies, let's install them by running the following command in the terminal:

```
$ composer install
```

As I stated earlier, all dependencies, including dev, will be installed if we are at the library's root. After a successful install, you should see output similar to the following:

```
Writing lock file
Generating autoload files
```

The lock file being referenced is *composer.lock*. When Composer installs libraries, it writes all of the packages and the exact versions of them downloaded to this file, thus locking the project to those specific versions.

Laravel Packages

Service Provider

Next, we need to create a service provider. Service providers are the connection between our package and Laravel. A service provider is responsible for binding things into Laravel's service container and informing Laravel where to load package resources such as views, configuration, routes, and localization files.
While inside the chat directory in the terminal, create ChatServiceProvider.php inside the src directory:

```
touch src/ChatServiceProvider.php
```

src/ChatServiceProvider.php

```php
<?php

namespace Tashtin\Chat;

use Illuminate\Support\ServiceProvider;

class ChatServiceProvider extends ServiceProvider
{
    protected $defer = true;

    public function boot()
    {
    }

    public function register()
    {
    }
}
```

We have included two methods, `boot()` and `register()`, in our service provider.

Register method

Within the register method, we bind items into the service container. A service container is responsible for managing class dependencies and performing dependency injection. We will specifically bind a service class that our package users will use to interact with the package. More on this later, when we look at facades.

Boot method

This method is called after all other services providers have been registered, meaning you have access to all other services that the framework has registered.
In this method, we will handle things like publishing database migrations and configuration files for our package. It is imperative to make your package configurable as much as possible. At the end of the day, your package users don't have an option to edit/customize your package code that is installed other than making use of the configuration options you provide. We will dive more into this when we look at adding configuration to our package.

Facades

Facades provide a "static" interface to classes that are available in the application's service container. Facades provide an expressive syntax while maintaining more testability and flexibility than the traditional static methods. For instance, we are going to have a Chat facade for our package, which will allow our package users to start a conversation by making the following call:

```
$conversation = Chat::createConversation($participants);
```

The above is not a call to a static method `createConversation`. The Facade base class uses the `__callStatic()` magic-method which is triggered when invoking inaccessible methods in a static context. This will defer calls from your facade to an object resolved from the container. Alternatively, the code above can be written as follows:

```
$chat = new Chat();
$conversation = $chat->createConversation();
```

You can see how it can easily get hard for your package users if you don't use facades. The first piece of code provides a memorable syntax that allows users to use your package without scrambling for documentation at every opportunity. That's not to say you always need facades as there are cons to using facades just like with anything else.

Now that we know what facades are let's create a facade for our package and name it ChatFacade. While in the terminal in the chat directory, run the following commands:

```
$ mkdir src/Facades
$ touch src/Facades/ChatFacade.php
```

Open **src/Facades/ChatFacade.php** and paste the following snippet:

src/Facades/ChatFacade.php

```php
<?php

namespace Tashtin\Chat\Facades;

use Illuminate\Support\Facades\Facade;

class ChatFacade extends Facade
{
    protected static function getFacadeAccessor()
    {
        return 'chat';
    }
}
```

The `getFacadeAccessor()` method returns a string that your container knows about (this string can be any name, we could have named it foo). In other words, looking at the code above, we know that we need to bind chat to an object that we want to use when making calls to our package. Remember, we bind items into the service container in the `register()` service provider method. So let's make the following update to our `register()` method of `src/ChatServiceProvider.php`

```php
public function register()
{
    $this->app->bind('chat', function () {
        return $this->app
            ->make(\Tashtin\Chat\Chat::class);
    });
}
```

We have bound our facade accessor string `chat` to a non-existent `Chat::class`, which we will create later. The idea here is for our package users to interact with only this class when using this package. So when one calls `Chat::send()` laravel Facade base class will help us resolve the ChatAPI object from the container, and in the end, we call the `send()` method in `Chat::class`.

Let's create the `Chat::class`.

```
$ touch src/Chat.php
```

And add the following:

```php
<?php

namespace Tashtin\Chat;

class Chat
{
}
```

This class will be the entry point to our package, so to speak.

Testing

Why Test-Driven Approach?

Many developers distance themselves from Test-Driven Development (TDD) or any testing. There is a notion that writing tests is a waste of time that could be used for actual development. In my experience, I have encountered and written code that I found hard to trust at some point. As a developer, you need to have faith in your code. Otherwise, you find yourself having sleepless nights thinking about a failing code deployment. We are going to use TDD to create our package. I have found Test Driven Development to be of great value in software development, and here are some benefits:

Decoupling

Test-Driven Development forces you to write tests before implementation. This, in turn, forces you to be concise since you must define your input and expectations before writing any code. When your code is derived from tests, you inevitably separate the different pieces as much as possible. The decoupled code is easily maintainable.

Up to date documentation

While it seems like a good idea to only rely on a living document, as time goes on and developers add more features or fix bugs, that document usually becomes neglected. However, enforcing TDD means developers have to add new tests for new features or write regression tests for any bugs. As a result, tests can act as documentation that cannot go out of date.

Refactor with confidence

Have you ever been unable to refactor code because of the fear that something will break? I have seen code that cries out loud for refactoring (including some code I have written in the past). I have sat there and watched classes grow into monster objects with duplicates because refactoring would be like opening a pandora box. With tests, you can refactor code with confidence because you will always run tests after each refactor, and if the test fails, you can immediately revert.

Minimize deployment errors

If you often find yourself afraid to deploy to production because the result is unknown, TDD can help you allay those fears. Writing code without bugs doesn’t mean that it works. Adding new features to a project won’t be a daunting task down the road. I have worked on projects where I have come back a few months or years later to add features. Without written tests, I always had to keep my fingers crossed that nothing breaks. Well-written tests will eliminate this headache as the tests will provide constant feedback that each component is still working.

Software packages get used by many users, and many users contribute to the development of the packages. With tests, it will take less time to go through Pull Requests. You won't waste time on pull requests with failing tests. Rather, the developers will address the issues before you review the code.

Setup tests

How to run PHPUnit tests

To run the PHPUnit, you would type the following in the terminal:

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

If you run this command at the root of our project you will notice a screen similar to the next output because we don't have any tests set up, and we haven't configured our PHPUnit.

```
PHPUnit 7.5.1-11-gedd72a804 by Sebastian Bergmann and contributors.

Usage: phpunit [options] UnitTest [UnitTest.php]
       phpunit [options] <directory>

Code Coverage Options:

  --coverage-clover <file>    Generate code coverage report in Clover XML format
  --coverage-crap4j <file>    Generate code coverage report in Crap4J XML format
  --coverage-html <dir>       Generate code coverage report in HTML format
  --coverage-php <file>       Export PHP_CodeCoverage object to file
  --coverage-text=<file>      Generate code coverage report in text format
                              Default: Standard output
  --coverage-xml <dir>        Generate code coverage report in PHPUnit XML format
  --whitelist <dir>           Whitelist <dir> for code coverage analysis
  --disable-coverage-ignore   Disable annotations for ignoring code coverage
  --no-coverage               Ignore code coverage configuration
  --dump-xdebug-filter <file> Generate script to set Xdebug code coverage filter

Logging Options:

  --log-junit <file>          Log test execution in JUnit XML format to
```

phpunit.xml and phpunit.xml.dist

First, before we write any tests, we need to set up the configuration for PHPUnit. PHPUnit will look for files phpunit.xml or phpunit.xml.dist for configuration, in that order. The idea is to allow developers to check-in phpunit.xml.dist in version control for distribution and keep phpunit.xml for local development, which might include customizations specific to the developer's needs. It's worth noting that PHPUnit will not use both files, phpunit.xml will take precedence.

Let's go ahead and create our configuration file. Run the following command in your terminal.

```
$ touch phpunit.xml
```

For now, we are only creating a configuration file for our local use. We will look at crafting a configuration file for distribution later.
Add the following content to your create file.

phpunit.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit backupGlobals="false"
         backupStaticAttributes="false"
         bootstrap="vendor/autoload.php"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false"
>
    <testsuites>
        <testsuite name="Chat Package Test Suite">
            <directory suffix=".php">./tests/Unit</directory>
        </testsuite>
    </testsuites>
    <filter>
        <whitelist>
            <directory suffix=".php">src/</directory>
        </whitelist>
    </filter>
</phpunit>
```

We are telling PHPUnit where our tests will be located, and for other options, you can read more at https://phpunit.de/documentation.html[3].
Now running the command './vendor/bin/PHPUnit again should give you an output similar to the following:

[3]https://phpunit.de/documentation.html

```
PHPUnit 8.5-g5b88be431 by Sebastian Bergmann and contributors.

Time: 36 ms, Memory: 4.00 MB

No tests executed!
```

Example Test

PHPUnit is now configured; let's create an example test to see if everything is set up correctly. Create a file `tests/Unit/ExampleTest.php` and add the following code

tests/Unit/ExampleTest.php

```php
<?php

namespace Tashtin\Chat\Tests;

class ExampleTest extends \Orchestra\Testbench\TestCase
{
    public function testExample()
    {
        $this->assertEquals(1, 1);
    }
}
```

Run PHPUnit again, and you should see something similar to the below indicating that your test has passed.

```
PHPUnit 7.5.1-11-gedd72a804 by Sebastian Bergmann and contributors.

.                                                       1 / 1 (100%)

Time: 83 ms, Memory: 8.00MB

OK (1 test, 1 assertion)
```

In the result of the PHPUnit run, a dot (.) shows for every test method which passed without any assertion failures. You will also be presented by the number of tests and assertions in the test run. In the event of a test failure, an `F` will be shown instead of a dot.

Let's briefly go over our example test. Our test class is extending `\Orchestra\Testbench\TestCase` which in turn extends `\PHPUnit\Framework\TestCase`. We will require all our tests to extend this class. The TestCase will set up some conditions, including providing us with helper methods to perform our assertions. `$this->assertEquals` is actually part of Assert class that `\PHPUnit\Framework\TestCase` extends.

TestCase

We don't want to keep extending `\Orchestra\Testbench\TestCase` for additional tests that we create because we may need to add our own additional code that our tests depend on. Let's create our own base class that will extend the TestCase that is offered by Testbench. Create a file `tests/TestCase.php` with the following:

```
$ touch tests/TestCase.php
```

tests/TestCase.php

```
<?php

namespace Tashtin\Chat\Tests;

class TestCase extends \Orchestra\Testbench\TestCase
{
}
```

Let's refactor our `tests/Unit/ExampleTest.php` to use this new base class as follows:

tests/Unit/ExampleTest.php

```
<?php

namespace Tashtin\Chat\Tests;

class ExampleTest extends TestCase
{
    public function testExample()
    {
        $this->assertEquals(1, 1);
    }
}
```

Run the test again `$ ./vendor/bin/phpunit`, and you may get the following error:

```
Tashtin\Chat\Tests\TestCase' not found in .../tashtin/chat/tests/Unit/ExampleTest.php
```

The issue here is Composer is not aware of the location of our file when the `vendor/autoload.php` file is generated. This is an easy fix; we need to add an autoload field in our composer.json to tell Composer to autoload our code.

Let's make a few updates to our composer.json file and add the following:

```
"autoload": {
   "psr-4": {
     "Tashtin\\Chat\\": "src/"
   }
},
"autoload-dev": {
    "psr-4": {
      "Tashtin\\Chat\\Tests\\": "tests"
    }
}
```

After the update, please run the following command to include our new class in the autoload class map.

```
$ composer dump-autoload
```

Rerun the test, and it should pass.

What is psr-4?

Git init

At this point, we can take the opportunity to check our changes into source control. We wouldn't want to lose our work.

Initialize git repository by running the following command in the project root

```
$ git init
```

Once the git repository has been initialized, we need to ignore tracking some files and directories. You don't want to track dependency directories or any configuration that could be only relevant to your system.

Create a .gitignore file:

```
$ touch .gitignore
```

Add the following contents to your .gitignore file:

```
/vendor
composer.lock
phpunit.xml
```

The *composer.lock* should be committed when working on an application. However, for libraries, I tend to ignore that from git. The package consumers will get the versions specified in composer.json anyway.

Run the following commands to add and commit your changes

```
$ git add . && git commit -m "First"
```

Note, I won't repeat the process of adding changes and committing to git. You can do that as you see fit along the way.

In the next section, we are going to start working on our package, following a test-driven approach.

Conversation

The first few tests that we write will have a lot of detail as we have to set up a few items such as database and configurations. So please bear with me as we go through that process slowly.

It creates a conversation

We want to make sure we can create a conversation. Create a file `tests/Unit/ConverationTest.php` that will hold tests related to Conversations

```
$ touch tests/Unit/ConversationTest.php
```

So here is my thinking, starting a conversation requires at least one participant, and you can invite other participants later but let's begin without participants with a method like this:

```
Chat::createConversation();
```

Red, Green

We will write a test that ensures that a conversation gets created and saved in the database.

tests/Unit/ConversationTest.php

```php
<?php

namespace Tashtin\Chat\Tests;

use Chat;

class ConversationTest extends TestCase
{
    public function testCreatesConversation()
    {
        $conversation = Chat::createConversation();
        $this->assertDatabaseHas('conversations', ['id' => 1]);
        $this->assertEquals(1, $conversation->id);
    }
}
```

We have written two assertions:

a)

```
$this->assertDatabaseHas('conversations', ['id' => 1]);
```

We expect to have only one conversation created and use incrementing ids; thus, we check for an id of 1. This assertion is part of the Laravel test helpers. You can check more on these helpers in Laravel documentation https://laravel.com/docs/6.x/database-testing#introduction

b)

```
$this->assertEquals(1, $conversation->id);
```

We also want to return a conversation from our conversation creation method to test that the object returned has an id of 1. We could also test for the returned object type, as we will later see writing more tests.

Run the test `./vendor/bin/phpunit`

```
E.                                                     2 / 2 (100%)

Time: 88 ms, Memory: 8.00MB

There was 1 error:

1) Tashtin\Chat\Tests\ConversationTest::testCreatesConversation
Error: Class 'Chat' not found

/Users/tinashe/Development/tashtin/chat/tests/Unit/ConversationTest.php:11
```

Here we have tried to use the facade **Chat**, but since we aren't in a Laravel application, this will always fail. As an alternative, we can use Orchestra testbench to write our package as if it were in a Laravel application.
Let's update our `tests/TestCase.php` to include the highlighted code.

```
<?php

namespace Tashtin\Chat\Tests;

class TestCase extends \Orchestra\Testbench\TestCase
{
    // markua-start-insert
    protected function getPackageProviders($app)
    {
        return [
            \Tashtin\Chat\ChatServiceProvider::class,
        ];
    }

    protected function getPackageAliases($app)
    {
        return [
            'Chat' => \Tashtin\Chat\Facades\ChatFacade::class,
        ];
    }
     // markua-end-insert
}
```

We are now telling testbench what facades are available and their location by overriding `getPackageAliases()`. We also load our service provider in the `getPackageProviders()`.

Rerun the tests, and you should get the following:

```
E.                                                         2 / 2 (100%)

Time: 100 ms, Memory: 10.00MB

There was 1 error:

1) Tashtin\Chat\Tests\ConversationTest::testCreatesConversation
Error: Call to undefined method Tashtin\Chat\Chat::createConversation()

/Users/tinashe/Development/tashtin/chat/vendor/laravel/framework/src/Illuminate/Supp\
ort/Facades/Facade.php:237
/Users/tinashe/Development/tashtin/chat/tests/Unit/ConversationTest.php:11
```

We are getting somewhere now, and it looks like we are missing the `createConversation()` method in `Tashtin\Chat\Chat::class`, so let's update that

```
<?php

namespace Tashtin\Chat;

class Chat
{
    // markua-start-insert
    public function createConversation()
    {
    }
    // markua-end-insert
}
```

Rerunning the test should throw the following error:

"'bash
There was 1 error:

1. TashtinChatTestsConversationTest::testCreatesConversation

IlluminateDatabaseQueryException: SQLSTATE[HY000] [2002] Connection refused (SQL: select count(*) as aggregate from `conversations` where
(`id` = 1))

/Users/tinashe/Development/tashtin/chat/vendor/laravel/framework/src/Illuminate/Database/Connection.php:664
/Users/tinashe/Development/tashtin/chat/vendor/laravel/framework/src/Illuminate/Database/Connection.php:624
/Users/tinashe/Development/tashtin/chat/vendor/laravel/framework/src/Illuminate/Database/Connection.php:333

~~~
The issue here is `$this->assertDatabaseHas('conversations', ['id' => 1]);` is looki\
ng for a table conversations to assert if the conversation is created but we don't h\
ave a database setup.
Let's add the following function to `tests/TestCase.php`

{line-numbers: false}
```php
<?php

namespace Tashtin\Chat\Tests;

class TestCase extends \Orchestra\Testbench\TestCase
{
    // markua-start-insert
    protected function getEnvironmentSetUp($app)
~~~

```php
    {
        parent::getEnvironmentSetUp($app);

        $app['config']->set('database.default', 'testbench');
        $app['config']->set('database.connections.testbench', [
            'driver' => 'sqlite',
            'database' => ':memory:',
            'prefix' => '',
        ]);
    }
    // markua-end-insert
}
```

The `getEnvironmentSetUp` function allows you to set configuration that you would typically put in Laravel config files or env files.

Rerun the test, and you should get an error that your database has no conversations table.

```
PDOException: SQLSTATE[HY000]: General error: 1 no such table: conversations
```

It's time to set up our database migrations for our package. Create a migration file `database/migrations/create_chat_tables.php`

"'bash
touch touch database/migrations/create_chat_tables.php

~~~php
{format: php, line-numbers: false}
```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateChatTables extends Migration
{
    public function up()
    {
        Schema::create('conversations', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->timestamps();
        });
    }

~~~

```php
    public function down()
    {
        Schema::dropIfExists('conversations');
    }
}
```

Here we created a basic migration for conversations table with just the id and timestamps. We will add additional columns later. Other migrations will be added to this same file.
Now let's tell our TestCase to use the migrations by making the following update.

```php
<?php

namespace Tashtin\Chat\Tests;

// markua-start-insert
require __DIR__ . '/../database/migrations/create_chat_tables.php';

use CreateChatTables;

// markua-end-insert

class TestCase extends \Orchestra\Testbench\TestCase
{
}
```

We want to run migrations each time a test is run, so the best place to do this is in the `setUp()` method of our TestCase. Let's override the setUp method by adding the following function to `tests/TestCase.php`

```php
public function setUp(): void
{
    parent::setUp();
    $this->artisan('migrate', ['--database' => 'testbench']);
    (new CreateChatTables)->up();
}
```

The `setUp()` method is invoked before each test run. It allows you an opportunity to create objects that you will test against. Additionally, we want to be able to reset our database after every test to get predictable results. We can hook into the `tearDown()` method, which runs after every test run. So let's add the following function to `tests/TestCase.php`

```
public function tearDown()
{
    (new CreateChatTables)->down();
    parent::tearDown();
}
```

Rerun the test, and you should get:

“‘bash
There was 1 failure:

1. TashtinChatTestsConversationTest::testCreatesConversation

Failed asserting that a row in the table [conversations] matches the attributes {

“id”: 1
}.

The table is empty.

~~~
Ok, we are getting somewhere here. We created a function to start a conversation, `c\
reateConversation`, but it doesn't do anything yet. We are missing a Conversation mo\
del. Let's update that function to the following:

{line-numbers: false}
```php
public function createConversation()
{
    // markua-start-insert
    return return Conversation::create();
    // markua-end-insert
}
~~~

Rerunning the test gives

“‘bash
There was 1 error:

1. TashtinChatTestsConversationTest::testCreatesConversation

Error: Class ‘TashtinChatConversation’ not found

~~~
Let's create the conversation model.
Create a directory `Models` in the `src` directory, then create `src/Models/Conversa\
tion.php`

{format: bash, line-numbers: false}
```bash
mkdir src/Models && touch src/Models/Conversation.php
~~~

```php
<?php

namespace Tashtin\Chat\Models;

use Illuminate\Database\Eloquent\Model;

class Conversation extends Model
{
}
```

Let's import the Conversation model in src/Chat.php, and we should have our class looking as follows:

```php
<?php

namespace Tashtin\Chat;

// markua-start-insert
use Tashtin\Chat\Models\Conversation;
// markua-end-insert
class Chat
{
    public function createConversation()
    {
        return Conversation::create();
    }
}
```

Rerunning the test we should see it pass.

```
..                                                            2 / 2 (100%)

Time: 159 ms, Memory: 14.00MB

OK (2 tests, 3 assertions)
```

Writing this test took a little longer because we had to go through all the missing setup steps.

Composer Scripts

You may have been tired of typing `./vendor/bin/phpunit` each time you run a test. You can easily shorten the command by creating an alias or by using Composer scripts. For Composer, scripts can be PHP callbacks or command-line executable commands. Let's add a script to shorten our PHPUnit command.

Add the following to your composer.json file:

```
"scripts": {
    "test": "vendor/bin/phpunit"
}
```

Now, whenever you need to run your tests, you can do that by running the command `composer test`

It returns a conversation

Rather than have our package users access the database directly to get a conversation, we want to provide them with a function to do this. In addition, the function allows us to do any transformations to our data before returning the response.

Red, Green

So let's add our next test right below the first:

```php
public function testReturnsConversationById()
{
    $conversation = Chat::createConversation();
    $c = Chat::conversations()->getById($conversation->id);

    $this->assertEquals($conversation->id, $c->id);
}
```

The first line of code creates the conversation, and we have already tested that it works.
In the second line, I want to provide users with an expressive syntax to do any operations related to conversations by chaining the operations with `Chat::conversations()`, which will return a service class as we don't want to do everything in the Conversation model.

Running the tests should result in the following failure `Error: Call to undefined method Tashtin\\Chat\\Chat::conversations()`.

Add the following function to src/Chat.php

```php
public function conversations()
{
    return $this->conversationService;
}
```

Run the test again

"'bash
There was 1 error:

1. TashtinChatTestsConversationTest::testReturnsConversationById

ErrorException: Undefined property: TashtinChatChat::$conversationService

~~~
Let's add the missing Conversation service.

{format: bash, line-numbers: false}
```bash
mkdir src/Services && touch src/Services/ConversationService.php
~~~

Add the following snippet to the Conversation service:

```php
<?php

namespace Tashtin\Chat\Services;

use Tashtin\Chat\Models\Conversation;

class ConversationService
{
    protected $conversation;

    public function __construct(Conversation $conversation)
    {
        $this->conversation = $conversation;
    }
}
```

Now let's use the service in our `src/Chat.php` class.

```php
<?php

namespace Tashtin\Chat;

use Tashtin\Chat\Models\Conversation;
// markua-start-insert
use Tashtin\Chat\Services\ConversationService;
// markua-end-insert

class Chat
{
    // markua-start-insert
    protected $conversationService;

    public function __construct(ConversationService $conversationService)
    {
        $this->conversationService = $conversationService;
    }
    // markua-end-insert

    public function createConversation()
    {
        return Conversation::create();
    }
    public function conversations()
    {
        return $this->conversationService;
    }
}
```

Rerun the test.

"'bash
There was 1 error:

1. TashtinChatTestsConversationTest::testReturnsConversationById

Error: Call to undefined method TashtinChatServicesConversationService::getById()

~~~
Let's add the function to `Tashtin\Chat\Services\ConversationService` to return the \
conversation when a conversation id is provided

{line-numbers: false}
```php
public function getById($conversationId)
{
    return $this->conversation->find($conversationId);
}
~~~

Rerun the tests, and you should see green.

Refactoring

While we are at it, let's take an opportunity to do a minor refactoring. In our `createConversation`, we are referencing the Conversation model. Let's move that task to the ConversationService by making the following updates.

TashtinChatChat.php

```
<?php

namespace Tashtin\Chat;

// markua-start-delete
use Tashtin\Chat\Models\Conversation;
// markua-end-delete
use Tashtin\Chat\Services\ConversationService;
```

```php
class Chat
{
    protected $conversationService;

    public function __construct(ConversationService $conversationService)
    {
        $this->conversationService = $conversationService;
    }

    // markua-start-delete
    public function createConversation()
    {
        return Conversation::create();
    }
    // markua-end-delete

    public function conversations()
    {
        return $this->conversationService;
    }
}
```

Add the function to start a conversation to the Conversation service.

src/Services/ConversationService.php

```php
<?php

namespace Tashtin\Chat\Services;

use Tashtin\Chat\Models\Conversation;

class ConversationService
{
    protected $conversation;

    public function __construct(Conversation $conversation)
    {
        $this->conversation = $conversation;
    }

    // markua-start-insert
    public function createConversation()
    {
        return $this->conversation->create();
    }
    // markua-end-insert

    public function getById($conversationId)
    {
        return $this->conversation->find($conversationId);
    }
}
```

Running the tests will show the following failures

```
Error: Call to undefined method Tashtin\Chat\Chat::createConversation()
```

We have moved the `createConversation()` function, so we need to update our tests as well

```
<?php

namespace Tashtin\Chat\Tests;

use Chat;

class ConversationTest extends TestCase
{
    public function testCreatesConversation()
    {
        // markua-start-delete
        $conversation = Chat::createConversation();
        // markua-end-delete
        // markua-start-insert
        $conversation = Chat::conversations()->createConversation();
        // markua-end-insert
        $this->assertDatabaseHas('conversations', ['id' => 1]);
        $this->assertEquals(1, $conversation->id);
    }

    public function testReturnsConversationById()
    {
        // markua-start-delete
        $conversation = Chat::createConversation();
        // markua-end-delete
        // markua-start-insert
        $conversation = Chat::conversations()->createConversation();
        // markua-end-insert
        $c = Chat::conversations()->getById($conversation->id);

        $this->assertEquals($conversation->id, $c->id);
    }
}
```

Rerun the tests, and we should see green.

Model Factories - generate test data

We have used the following code `$conversation = Chat::conversations()->createConversation();` in the test for getting a conversation by id. However, there is a better way to make sure a conversation is in the database and test the function that retrieves the conservation. We are going to use factories for this. Factories allow us to insert records into our database before executing tests. Let's create `database/factories/ModelFactory.php`

```
mkdir database/factories && touch database/factories/ModelFactory.php
```

database/factories/ModelFactory.php

```
<?php
use Faker\Generator as Faker;
use Tashtin\Chat\Models\Conversation;

$factory->define(Conversation::class, function (Faker $faker) {
    return [];
});
```

Now let's make updates to our `testReturnsConversationById()` test

```php
<?php

namespace Tashtin\Chat\Tests;

use Chat;
// markua-start-insert
use Tashtin\Chat\Models\Conversation;
// markua-end-insert

class ConversationTest extends TestCase
{
    // ...

    public function testReturnsConversationById()
    {
        // markua-start-delete
        $conversation = Chat::conversations()->createConversation();
        // markua-end-delete
        // markua-start-insert
        $conversation = factory(Conversation::class)->create();
        // markua-end-insert
        $c = Chat::conversation()->getById($conversation->id);

        $this->assertEquals($conversation->id, $c->id);
    }
}
```

Rerun the test, and you will get the following:

```
1) Tashtin\Chat\Tests\ConversationTest::testReturnsConversationById
InvalidArgumentException: Unable to locate factory with name [default] [Tashtin\Chat\
\Models\Conversation].
```

We need to tell our tests where to find the database factories. The Orchestra test bench will help us here. Let's make updates to the setUp method of our `tests/TestCase.php`.

```
public function setUp()
{
    parent::setUp();
    $this->artisan('migrate', ['--database' => 'testbench']);
    // markua-start-insert
    $this->withFactories(__DIR__ . '/../database/factories');
    // markua-end-insert
    (new CreateChatTables)->up();
}
```

Rerun the tests, and they should pass.

It can create a conversation with a participant

We want to be able to create a conversation and add participants right away. The best way would be to pass an array of participant models when starting a conversation, allowing the addition of multiple participants. For instance, something like this:

```
$participants = [$model1, $model2];
$conversation = Chat::createConversation($participants);
```

We don't have any models to use as participants, and we don't want to spin up a Laravel application to test this. Also, the package should work for any model that the user provides. We need to set up models that we can use as helpers for our tests. The test helpers are not limited to models; they can be factories, migrations, or anything.

Let's create a directory to store our test helpers.

```
mkdir tests/Helpers
touch tests/Helpers/Models.php
```

We are going to store all our test helper Models in `Models. php` as below:

```
<?php

namespace Musonza\Chat\Tests\Helpers\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected $table = 'mc_users';
}

class Client extends Model
{
    protected $table = 'mc_clients';
    protected $primaryKey = 'client_id';
}

class Bot extends Model
{
    protected $table = 'mc_bots';
    protected $primaryKey = 'bot_id';
}
```

I have chosen `User`, `Client`, and `Bot` models to represent potential models that users may have. You can add as many models as you like. I have also intentionally specified primary keys other than the default `id` because our package users can have different primary keys, and we want to make sure the package will remain functional in that situation. Let's use our test models in our test case:

```
<?php

use Chat;
use Tashtin\Chat\Models\Conversation;
// markua-start-insert
use Musonza\Chat\Tests\Helpers\Models\Client;
use Musonza\Chat\Tests\Helpers\Models\Bot;
use Musonza\Chat\Tests\Helpers\Models\User;
// markua-end-insert

class ConversationTest extends TestCase
{
    // ...
    // markua-start-insert
    public function testCreatesConversationWithParticipants()
    {
        $clientModel = factory(Client::class)->create();
        $botModel = factory(Bot::class)->create();
        $userModel = factory(User::class)->create();

        $participants = [$clientModel, $botModel, $userModel];
        $conversation = Chat::conversations()->createConversation($participants);

        $this->assertCount(3, $conversation->participants);
    }
    // markua-end-insert
}
```

Running the tests will result in the following error:

```
1) Tashtin\Chat\Tests\ConversationTest::testCreatesConversationWithParticipants
Error: Class 'Musonza\Chat\Tests\Helpers\Models\Client' not found
```

This error looks similar to the failure we had before with Composer autoloading. If you assumed so, you are correct. `tests/Helpers/Models.php` is not a class; therefore won't be included in the class map by the autoloader. We need to tell Composer to autoload our new file for development. We can do so by adding a files field under autoload-dev in composer.json as follows:

```
"autoload-dev": {
    "psr-4": {
        "Tashtin\\Chat\\Tests\\": "tests"
    },
    "files": [
        "tests/Helpers/Models.php"
    ]
}
```

After the addition, run the following command:

```
composer dump-autoload
```

Run the tests again

```
1) Tashtin\Chat\Tests\ConversationTest::testCreatesConversationWithParticipants
InvalidArgumentException: Unable to locate factory with name [default] [Musonza\Chat\
\Tests\Helpers\Models\Client].
```

We need to add database factories for the test models. However, since these are not real models for the package, I would like to separate them from actual package factories by adding them under test helpers.

```
mkdir tests/Helpers/factories
touch tests/Helpers/factories/ModelFactory.php
```

Add the following content to your helpers `ModelFactory.php`:

tests/Helpers/factories/ModelFactory.php

```php
<?php

use Faker\Generator as Faker;
use Musonza\Chat\Tests\Helpers\Models\Bot;
use Musonza\Chat\Tests\Helpers\Models\Client;
use Musonza\Chat\Tests\Helpers\Models\User;

$factory->define(User::class, function (Faker $faker) {
    static $password;

    return [
        'name'           => $faker->name,
        'email'          => $faker->unique()->safeEmail,
        'password'       => $password ?: $password = bcrypt('secret'),
        'remember_token' => 'some_random_string',
    ];
});

$factory->define(Client::class, function (Faker $faker) {
    return [
        'name'           => $faker->name,
    ];
});

$factory->define(Bot::class, function (Faker $faker) {
    return [
        'name'           => $faker->name,
    ];
});
```

Now we need to tell our TestCase where to locate these factories. Go ahead and update your `tests/TestCase.php` class.

```
public function setUp(): void
{
    parent::setUp();
    $this->artisan('migrate', ['--database' => 'testbench']);
    $this->withFactories(__DIR__ . '/../database/factories');
    // markua-start-insert
    $this->withFactories(__DIR__.'/Helpers/factories');
    // markua-end-insert
    (new CreateChatTables)->up();
    (new CreateTestTables)->up();
}
```

Rerunning the tests will fail as we don't have migrations for our test tables.

```
1) Tashtin\Chat\Tests\ConversationTest::testCreatesConversationWithParticipants
Illuminate\Database\QueryException: SQLSTATE[HY000]: General error: 1 no such table:\
 mc_clients (SQL: insert into "mc_clients" ("name", "updated_at", "created_at") valu\
es (Vance Considine MD, 2019-12-02 05:04:54, 2019-12-02 05:04:54))
```

Let's add a file that will store our test helper migrations. These would be separate from the migrations that we will publish with the package.

```
touch tests/Helpers/migrations.php
```

Add the following to the file:

tests/Helpers/migrations.php

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateTestTables extends Migration
{
    public function up()
    {
        Schema::create('mc_users', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });

        Schema::create('mc_clients', function (Blueprint $table) {
            $table->increments('client_id');
            $table->string('name');
            $table->timestamps();
        });

        Schema::create('mc_bots', function (Blueprint $table) {
            $table->increments('bot_id');
            $table->string('name');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('mc_users');
        Schema::dropIfExists('mc_clients');
        Schema::dropIfExists('mc_bots');
    }
}
```

Like we did earlier with our package migrations, we need to let our TestCase know where to locate these migrations.

```
<?php

namespace Tashtin\Chat\Tests;

require __DIR__ . '/../database/migrations/create_chat_tables.php';
// markua-start-insert
require __DIR__.'/Helpers/migrations.php';
// markua-end-insert

use CreateChatTables;
// markua-start-insert
use CreateTestTables;
// markua-end-insert
```

```
class TestCase extends \Orchestra\Testbench\TestCase
{
    public function setUp(): void
    {
        parent::setUp();
        $this->artisan('migrate', ['--database' => 'testbench']);
        $this->withFactories(__DIR__ . '/../database/factories');
        $this->withFactories(__DIR__.'/Helpers/factories');
        (new CreateChatTables)->up();
        // markua-start-insert
        (new CreateTestTables())->up();
        // markua-end-insert
    }

    public function tearDown(): void
    {
        (new CreateChatTables)->down();
        // markua-start-insert
        (new CreateTestTables())->down();
        // markua-end-insert
        parent::tearDown();
    }
}
```

In addition, we also added code to run our test migrations.

Run the tests again

```
There was 1 error:

1) Tashtin\Chat\Tests\ConversationTest::testCreatesConversationWithParticipants
PHPUnit\Framework\InvalidArgumentException: Argument #2 of PHPUnit\Framework\Assert:\
:assertCount() must be a countable or iterable
```

We aren't doing anything with the participants that we are attempting to add to the conservation. It looks like we need an additional database table to keep track of our participants.

Let's head over to our migrations file `database/migrations/create_chat_tables.php` and make the following updates:

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateChatTables extends Migration
{
    public function up()
    {
        Schema::create('conversations', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->timestamps();
        });

        // markua-start-insert
        Schema::create('participation', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->bigInteger('conversation_id')->unsigned();
            $table->bigInteger('messageable_id')->unsigned();
            $table->string('messageable_type');
            $table->timestamps();
            $table->unique(
                ['conversation_id', 'messageable_id', 'messageable_type'],
                'participation_index'
            );
            $table->foreign('conversation_id')
                ->references('id')
                ->on('conversations')
                ->onDelete('cascade');
        });
        // markua-end-insert
    }

    public function down()
    {
        // markua-start-insert
        Schema::dropIfExists('participation');
        // markua-end-insert
        Schema::dropIfExists('conversations');
    }
}
```

We are adding a table named `participation` to keep track of our participants in a conversation. For each table entry, we need to know the conversation associated with the participant; thus, we have the `conservation_id` column. Additionally, we need to store the identifier for the participant as `messageable_id`. The participant could be a `bot`, `user`, or any model of our choice. Since the participant can be of any model type, we need to store `messageable_type` to help us resolve the correct participant. `messageable_type` would ideally be the full namespace of your class model. For instance, `App\User`, `App\Bot`, and so on. Lastly, we create a unique index, `participation_index`, to ensure we don't duplicate participants in a conversation.

As we mentioned earlier, our users are going to be working with multiple models. It would be nice for us to give those models characteristics of participants via Traits and Polymorphic relationships. If this sounds confusing, don't worry, we will cover that soon. Once we have our tests, it will be easier to refactor our code. So, in the meantime, for lack of a better term, we will use a primitive approach to add participants to a conversation.

Ok, let's modify our createConversation function in `src/Services/ConversationService.php` to the following:

```php
public function createConversation($participants = [])
{
    $participation = [];
    $conversation = $this->conversation->create();

    foreach ($participants as $participant) {
        $participation[] = [
            'conversation_id' => $conversation->getKey(),
            'messageable_id' => $participant->getKey(),
            'messageable_type' => get_class($participant),
        ];
    }

    $this->participation->insert($participation);

    return $conversation;
}
```

We create the conversation, loop through our participants, and add them to our participant table. You may have noticed that I have used `getKey()` on the models instead of explicitly specifying the primary key. Doing so allows this code to work for any model primary key, be it `id`, `user_id`, `bot_id`, and so on.

Running the tests will throw the following error:

```
ErrorException: Undefined property: Tashtin\Chat\Chat::$participation
```

We need to create a participation model. Let's do that.

```
touch src/Models/Participation.php
```

```php
<?php

namespace Tashtin\Chat\Models;

use Illuminate\Database\Eloquent\Model;

class Participation extends Model
{
    protected $table = 'participation';

    public function conversation()
    {
        return $this->belongsTo(Conversation::class, 'conversation_id');
    }
}
```

Also, we need to add the participation relationship to our conversation model:

src/Models/Conversation.php

```php
<?php

namespace Tashtin\Chat\Models;

use Illuminate\Database\Eloquent\Model;

class Conversation extends Model
{
    public function participants()
    {
        return $this->hasMany(Participation::class);
    }
}
```

Now let's use our Participation model in `ConversationService.php`.
The beginning of your class should look as follows:

```php
<?php

namespace Tashtin\Chat\Services;

use Tashtin\Chat\Models\Conversation;
use Tashtin\Chat\Models\Participation;

class ConversationService
{
    protected $conversation;
    protected $participation;

    public function __construct(
        Conversation $conversation,
        Participation $participation
    ) {
        $this->conversation = $conversation;
        $this->participation = $participation;
    }

    // ...
}
```

Run the tests again and you should see them passing.

It can add participants to a conversation

In addition to adding participants during conversation creation, we need to add additional participants after the conversation has been created. Add the following test case to `tests/Unit/ConversationTest.php`

```php
public function testAddUserToConversation()
{
    $conversation = factory(Conversation::class)->create();
    $userModel = factory(User::class)->create();
    Chat::conversations()
        ->setConversation($conversation)
        ->addParticipants([$userModel]);
    $this->assertCount(1, $conversation->participants);
}
```

We create a conversation and a participant using the factories that we set up earlier. We pass a specific conversation that we want to work with to `setConversation` and chain the `addParticipants` method.

Run the tests

```
1) Tashtin\Chat\Tests\ConversationTest::testAddUserToConversation
Error: Call to undefined method Tashtin\Chat\Services\ConversationService::setConver\
sation()
```

Let's create the `setConversation` function in `src/Services/ConversationService.php`

```
public function setConversation(Conversation $conversation): self
{
    $this->conversation = $conversation;
    return $this;
}
```

Run the test again

```
1) Tashtin\Chat\Tests\ConversationTest::testAddUserToConversation
Error: Call to undefined method Tashtin\Chat\Services\ConversationService::addPartic\
ipants()
```

Next, we add the `addParticipants` function

```
public function addParticipants(array $participants)
{
    $participation = [];

    foreach ($participants as $participant) {
        $participation[] = [
            'conversation_id' => $this->conversation->getKey(),
            'messageable_id' => $participant->getKey(),
            'messageable_type' => get_class($participant),
        ];
    }

    $this->participation->insert($participation);
}
```

Rerun the tests, and you should get green.
We have a little problem here; it appears we have some code duplication. If you remember earlier, we added the same functionality to add participants on conversation creation. Let's do a minor refactoring to our `createConversation` use our new `addParticipants` function as follows:

ConversationService::createConversation

```
public function createConversation($participants = []): Conversation
{
    $this->conversation = $this->conversation->create();
    $this->addParticipants($participants);

    return $this->conversation;
}
```

Rerun the tests, and you should see them passing.

It can remove a single participant from a conversation

Now that we can add a participant to a conversation, we need to create functionality to reverse that operation. Think of a group chat in which we would want to give participants the ability to leave a conversation or for administrators to remove participants.

Let's add a testcase for removing participants from a conversation.

ConversationTest::testRemoveUserFromConversation

```
public function testRemoveUserFromConversation()
{
    $conversation = factory(Conversation::class)->create();
    $userModel = factory(User::class)->create();

    Chat::conversations()
        ->setConversation($conversation)
        ->addParticipants([$userModel]);
    $this->assertCount(1, $conversation->participants);

    Chat::conversations()
        ->setConversation($conversation)
        ->removeParticipants([$userModel]);
    $this->assertCount(0, $conversation->fresh()->participants);
}
```

We have created a conversation and added a participant. Then we verify that we have a count of one participant. A new method removeParticipants will be used to remove users from a conversation. You may also have noticed we have used `$conversation->fresh()`. The `fresh` method will re-retrieve the model from the database so we can have tests on the affected model.

Run the tests

```
1) Tashtin\Chat\Tests\ConversationTest::testRemoveUserFromConversation
Error: Call to undefined method Tashtin\Chat\Services\ConversationService::removePar\
ticipants()
```

ConversationService::removeParticipants

```
public function removeParticipants(array $participants)
{
    foreach ($participants as $participant) {
        $this->participation->where([
            'conversation_id' => $this->conversation->getKey(),
            'messageable_id' => $participant->getKey(),
            'messageable_type' => get_class($participant),
        ])->delete();
    }
}
```

We have essentially written a query to check for the passed participant in the participation table and delete the entry. We will refactor this later on to make use of model relationships. However, for now, this is sufficient to get our test passing.

In the meantime, let's run our tests, and we should see green.

It can return conversation participants

One thing our package users would want is to tell who is participating in a conversation. We could use this information to send notifications to participants when a message is received or when a new participant joins.

Let's add a test to make sure we can return participants in a conversation. We will add a user to a conversation and assert the number of conversations returned in our test. We will also assert that we indeed receive an object of type Conversation.

```
public function testReturnsParticipantConversations()
{
   $conversation = factory(Conversation::class)->create();
   $participant = factory(User::class)->create();
   Chat::conversations()
       ->setConversation($conversation)
       ->addParticipants([$participant]);

    $this->assertInstanceOf(
        Conversation::class, $participant->conversations()->first()
    );
}
```

Running the test above will fail. We have one problem with our code at this point. We can not directly figure out what conversations a participant belongs to. If we only had one type of participant model, we could easily add a relationship that links to Conversations to that model. However, our participants can be of any model type, a perfect use case for polymorphic relationships.

Refactor to Polymorphic relationship

In short, a polymorphic relationship allows the target model to belong to more than one type of model using a single association. In our case, the Conversation model is our target model. Now, remember, we don't have any knowledge of where our package users will store their participant Models in their application. So we need to provide them with a way to give their models an ability to participate, which gives them the polymorphic relationship to query any conversations they may have. To get around this issue, we can use traits, a mechanism for code reuse across classes.

Create the Traits folder to store our traits and then create a trait that will allow models to participate in conversations.

```
mkdir src/Traits && touch src/Traits/Messageable.php
```

src/Traits/Messageable.php

```php
<?php

namespace Tashtin\Chat\Traits;

use Illuminate\Database\Eloquent\Relations\MorphMany;
use Tashtin\Chat\Models\Participation;

trait Messageable
{
   public function participation(): MorphMany
   {
       return $this->morphMany(Participation::class, 'messageable');
   }
}
```

If you recall, we introduced the participation model to keep track of our participants; hence we can create a relationship between participation and our *messageable* models. So, if we have a User model that is *messageable*, calling $user->participation will return all entries of the model's participation. Let's use the trait in our test models by making an update to `tests/Helpers/Models.php` as below.

tests/Helpers/Models.php

```php
<?php

namespace Musonza\Chat\Tests\Helpers\Models;

use Illuminate\Database\Eloquent\Model;
// markua-start-insert
use Tashtin\Chat\Traits\Messageable;
// markua-end-insert

class User extends Model
{
   // markua-start-insert
   use Messageable;
   // markua-end-insert
   protected $table = 'mc_users';
}

class Client extends Model
{
   // markua-start-insert
   use Messageable;
   // markua-end-insert
   protected $table = 'mc_clients';
   protected $primaryKey = 'client_id';
}

class Bot extends Model
{
   // markua-start-insert
   use Messageable;
   // markua-end-insert
   protected $table = 'mc_bots';
   protected $primaryKey = 'bot_id';
}
```

Let’s run our tests to see where we are and if our changes had any negative impacts.

```
There was 1 error:

1) Tashtin\Chat\Tests\ConversationTest::testReturnsParticipantConversations
BadMethodCallException: Call to undefined method Musonza\Chat\Tests\Helpers\Models\U\
ser::conversations()
```

Our trait at this time only returns a participation model and is missing a method to return conversations.

We are going to add a function that returns conversations for our participants. Your messageable trait should look as below:

src/Traits/Messageable.php

```php
<?php

namespace Tashtin\Chat\Traits;

use Illuminate\Database\Eloquent\Relations\MorphMany;
use Illuminate\Support\Collection;
use Tashtin\Chat\Models\Participation;

trait Messageable
{
   public function participation(): MorphMany
   {
       return $this->morphMany(Participation::class, 'messageable');
   }

   public function conversations(): Collection
   {
       return $this->participation->pluck('conversation');
   }
}
```

We have used the pluck method since we only need to get the conversation from participation.

Rerun the tests, and you should see green.

It can create a direct conversation between two participants

Our participants may need to participate in private one on one conversations. It would not be good if our package exposes private conversations to users who aren't part of the conversation or allows additional participants to join a direct conversation.

```
public function testCreateDirectConversation()
{
   [$participant1, $participant2] = factory(User::class, 2)->create();
   $conversation = Chat::conversations()
       ->createConversation(
            [$participant1, $participant2],
            ['direct_message' => true]
        );

   $this->assertTrue($conversation->direct_message);
}
```

Run the test

```
There was 1 error:

1) Tashtin\Chat\Tests\ConversationTest::testCreateDirectConversation
Failed asserting that false is true.
```

A field `direct_message` is needed on the conversation model. We need to update our conversations table migration to include the column.

```
Schema::create('conversations', function (Blueprint $table) {
   $table->bigIncrements('id');
   // markua-start-insert
   $table->boolean('direct_message')->default(false);
   // markua-end-insert
   $table->timestamps();
});
```

By default, we will have `direct_message` set to false.

Now when we create a conversation, we will specify whether it's a direct message or not

```
public function createConversation($participants = [], $data = []): Conversation
{
    $this->conversation = $this->conversation->create($data);

    $this->addParticipants($participants);

    return $this->conversation;
}
```

Running the tests, we get the following failure:

"‘bash
IlluminateDatabaseEloquentMassAssignmentException: Add [direct_message] to fillable property to allow mass assignment on [TashtinChatModelsConversation].

~~~
The error is self-explanatory; Laravel is protecting us from mass assignment to our \
database table. Let's go ahead and whitelist by adding a fillable property to our Co\
nversation model.

{format: php, line-numbers: false}
```php
<?php

namespace Tashtin\Chat\Models;

use Illuminate\Database\Eloquent\Model;

class Conversation extends Model
{
// markua-start-insert
   protected $fillable = ['direct_message'];
// markua-end-insert

// … rest of the code here
}
~~~

Rerunning the tests, we should see green. While we have added an option to create a direct message, we don’t have safeguards that prevent us from adding users to existing direct messages or re-creating direct messages for the same pair of participants. Also, we may need to throw an exception if two participants are not specified when creating a direct message. We will address that later. We will refactor and address our issue, passing simple or primitive values like `direct_message`. Primitive values may make our code less readable and hard to validate.

It can make a private conversation

Our users will need to create private conversations. A good use case would be closed social network groups or something that works similarly to private Slack channels.

As always, let's start by crafting our testcase.

```
    public function testCreatePrivateConversation()
    {
        [$participant1, $participant2] = factory(User::class, 2)->create();
        $conversation = Chat::conversations()
            ->createConversation(
                [$participant1, $participant2],
                ['is_private' => true]
            );

        $this->assertTrue($conversation->is_private);
    }
```

Running our test fails:

```
1) Tashtin\Chat\Tests\ConversationTest::testCreatePrivateConversation
Failed asserting that null is true.
```

Let's add the column `is_private` to our Conversation model migration.

Conversation migration

```
Schema::create('conversations', function (Blueprint $table) {
    $table->bigIncrements('id');
    $table->boolean('direct_message')->default(false);
    $table->boolean('is_private')->default(false);
    $table->timestamps();
});
```

In addition, we will make the column fillable in our Conversation model to prevent mass assignment exceptions.

Conversation model fillable entry

```
protected $fillable = [
    'direct_message',
    'is_private',
];
```

Rerunning the tests, and you should get green.

What's next? What if we wanted to revert and make a private conversation public? We will work on this when we add functionality to update a conversation.

It can create a conversation with details

We want to offer our package users the ability to create conversations and specify additional details. You may want to have a conversation with properties like name, description, topic, and so on. Since we can not cover all the properties in the realm of possibility, we can also provide an additional column blob for our users to use as they see fit.

```
    public function testCreateConversationWithDetails()
    {
        $data = [
            'name' => 'PHP',
            'description' => 'PHP Channel',
            'topic' => 'Topic here',
            'extra' => [
                'field1' => 'field1 content',
                'field2' => 'field2 content',
                'field3' => 'field3 content',
            ]
        ];

        $participant= factory(User::class)->create();
        $conversation = Chat::conversations()
            ->createConversation([$participant], $data);

        $this->assertSame($data['name'], $conversation->name);
        $this->assertSame($data['description'], $conversation->description);
        $this->assertSame($data['topic'], $conversation->topic);
        $this->assertSame($data['extra'], $conversation->extra);
    }
```

We had already added a second parameter with conversation data/details to the ConversationService function `createConversation` when we worked on creating a direct message. At this point, users can pass in any additional fields, so we may want to whitelist the accepted fields at some point. As I mentioned earlier, you can see how passing primitives can get out of hand in terms of validation.

In the meantime, let us update our Conversation migration to the following:

```
Schema::create('conversations', function (Blueprint $table) {
    $table->bigIncrements('id');
    $table->boolean('direct_message')->default(false);
    $table->boolean('is_private')->default(false);
    $table->string('name')->nullable();
    $table->string('description')->nullable();
    $table->string('topic')->nullable();
    $table->string('extra')->nullable();
    $table->timestamps();
});
```

Run the tests, and you may get the following failure:

"'bash
There was 1 error:

1. TashtinChatTestsConversationTest::testCreateConversationWithDetails

ErrorException: Array to string conversion

~~~
The failure is caused by attempting to insert the `extra` field, an array. We can so\
lve this by adding a `$cast` attribute to our Conversation model:

```php
<?php

namespace Tashtin\Chat\Models;

use Illuminate\Database\Eloquent\Model;

class Conversation extends Model
{
    // ...

    protected $casts = [
        'direct_message' => 'boolean',
~~~

```
        'is_private'        => 'boolean',
        'extra'           => 'array',
    ];

    // ...
}
```

I have also added **`direct_message`** and **`is_private`** to cast them to the appropriate types.

Run the tests, and we should see green.

Value Objects - gain control over primitives

I mentioned about issues involved when using primitive values. We can still have our functions accept primitive values, but it can be beneficial to convert those values to value objects before using them. What are value objects? These are immutable types that are distinguishable only by the state of their properties.

Let's create a Value Object to use for our conversation data.

```
mkdir src/ValueObjects && touch src/ValueObjects/ConversationData.php
```

src/ValueObjects/ConversationData.php

```
namespace Tashtin\Chat\ValueObjects;

use Illuminate\Support\Collection;
use Tashtin\Chat\Models\Conversation;

class ConversationData
{
    /**
     * @var Collection
     */
    protected $data;

    public function __construct(array $input)
    {
        $this->data = collect($input);
    }

    public function getData()
    {
        return $this->data->toArray();
    }
}
```

Here we just pass the properties/input data to `ConversationData` value object. We are not doing any validation or filtering for supported properties yet. Let's see if we can refactor to use this class in our ConversationService.

Import the class ConversationData at the top of your class

```
use Tashtin\Chat\ValueObjects\ConversationData;
```

Update the `createConversation` function to look as below:

```
public function createConversation($participants = [], $data = []): Conversation
{
    $conversationData = (new ConversationData($data))->getData();

    $this->conversation = $this->conversation
        ->create($conversationData);

    $this->addParticipants($participants);

    return $this->conversation;
}
```

Run the tests, and you should see green. Currently, our `getData` function in our value object returns all the input data passed. Let's work on filtering out unsupported properties that the users may pass.

Update `ConversationData` class to the following:

```
<?php

namespace Tashtin\Chat\ValueObjects;

use Illuminate\Support\Collection;
use Tashtin\Chat\Models\Conversation;

class ConversationData
{
    // ...

    public function getSupportedProperties(): array
    {
        return app(Conversation::class)->getFillable();
    }

    public function filtered(): Collection
    {
        return $this->data->filter(function ($val, $key){
            return in_array($key, $this->getSupportedProperties());
        });
    }

    // ...
}
```

We have added `getSupportedProperties` method to return our supported properties. In our case, we have limited this to our model's fillable properties.

Now our **`createConversation`** function would look as follows:

```
public function createConversation($participants = [], $data = []): Conversation
{
    $conversationData = (new ConversationData($data))
        ->filtered()
        ->toArray();

    $this->conversation = $this->conversation
        ->create($conversationData);

    $this->addParticipants($participants);

    return $this->conversation;
}
```

Rerun the tests, and you should get green.

Refactoring Tests

Yes, we can refactor tests. So before we move on, let's do a minor refactoring to our tests. It seems we have repetitive code when we create participants before each test. In addition, we have just named them **`$participant1`**, **`$participant2`**, and so on. It would be nice if we have easy-to distinguish names, especially when we move on to writing messaging code.

With that said, I would like to create participant models in the base testcase before each test.

tests/TestCase.php

```
<?php

namespace Tashtin\Chat\Tests;

require __DIR__ . '/../database/migrations/create_chat_tables.php';
require __DIR__.'/Helpers/migrations.php';

use CreateChatTables;
use CreateTestTables;
use Musonza\Chat\Tests\Helpers\Models\User;
use Musonza\Chat\Tests\Helpers\Models\Bot;
use Musonza\Chat\Tests\Helpers\Models\Client;

class TestCase extends \Orchestra\Testbench\TestCase
{
    // ...
    protected $userAlpha;
    protected $userBravo;
    protected $userCharlie;
    protected $userDelta;

    protected $botAlpha;
    protected $botBravo;
    protected $botCharlie;
    protected $botDelta;

    protected $clientAlpha;
    protected $clientBravo;
    protected $clientCharlie;
    protected $clientDelta;

    public function setUp(): void
    {
        // ...
        [
            $this->userAlpha,
            $this->userBravo,
            $this->userCharlie,
            $this->userDelta,
        ] = $this->createUserModels(4);
        [
            $this->botAlpha,
            $this->botBravo,
            $this->botCharlie,
            $this->botDelta,
        ] = $this->createBotModels(4);
        [
            $this->clientAlpha,
            $this->clientBravo,
            $this->clientCharlie,
            $this->clientDelta,
        ] = $this->createClientModels(4);
    }

    // ...
    public function createUserModels($count = 1)
    {
        return factory(User::class, $count)->create();
    }

    public function createBotModels($count = 1)
    {
        return factory(Bot::class, $count)->create();
    }

    public function createClientModels($count = 1)
    {
        return factory(Client::class, $count)->create();
    }

    // ...
}
```

We now have functions to create our helper test models. Unchanged code has been omitted for brevity. Let’s update our **tests/Unit/ConversationTest.php** to make use of these changes.

tests/Unit/ConversationTest.php

```php
<?php

namespace Tashtin\Chat\Tests;

use Chat;
use Tashtin\Chat\Models\Conversation;

class ConversationTest extends TestCase
{
    // ...

    public function testCreatesConversationWithParticipants()
    {
        $participants = [$this->userAlpha, $this->clientAlpha, $this->botAlpha];
        $conversation = Chat::conversations()->createConversation($participants);
        // ...
    }

    public function testAddUserToConversation()
    {
        // ...
        Chat::conversations()
            ->setConversation($conversation)
            ->addParticipants([$this->userAlpha]);
        // ...
    }

    public function testRemoveUserFromConversation()
    {
        // ...
        Chat::conversations()
            ->setConversation($conversation)
            ->addParticipants([$this->userAlpha]);
        // ...

        Chat::conversations()
            ->setConversation($conversation)
            ->removeParticipants([$this->userAlpha]);
        // ...
    }

    public function testReturnsParticipantConversations()
    {
        $conversation = factory(Conversation::class)->create();
        Chat::conversations()
            ->setConversation($conversation)
            ->addParticipants([$this->userDelta]);

        $this->assertInstanceOf(
            Conversation::class, $this->userDelta->conversations()->first()
        );
    }

    public function testCreateDirectConversation()
    {
        $conversation = Chat::conversations()
            ->createConversation([$this->userAlpha, $this->userBravo], [
                'direct_message' => true,
            ]);

        // ...
    }

    public function testCreatePrivateConversation()
    {
        $conversation = Chat::conversations()
            ->createConversation([$this->userAlpha, $this->userBravo], [
                'is_private' => true,
            ]);

        // ...
    }

    public function testCreateConversationWithDetails()
    {
       // ...

        $conversation = Chat::conversations()
            ->createConversation([$this->userAlpha], $data);

        // ...
    }
}
```

Run the tests, and you should see them passing.

Next, we are going to move on to sending messages in a conversation. We still need to address some issues in our conversation-related code, but it will become more apparent after implementing messaging and other features.

Messaging

Create messages

It can send a message

Now let’s get to the fun part of actually sending messages.

Let’s create a new test file to add our messaging-related tests.

```
$ touch tests/Unit/MessageTest.php
```

tests/Unit/MessageTest.php

```php
<?php

namespace Tashtin\Chat\Tests;

use Chat;

class MessageTest extends TestCase
{
    /** @test */
    public function it_can_send_a_message()
    {
        $conversation = Chat::conversations()
            ->createConversation([$this->userAlpha, $this->userBravo]);

        Chat::messages()
            ->body('Hello')
            ->from($this->userBravo)
            ->to($conversation)
            ->send();

        $this->assertEquals($conversation->messages->count(), 1);
        $this->assertEquals(
            'Hello',
            $conversation->messages->first()->body
        );
    }
}
```

Run the tests

```
There was 1 error:

1) Tashtin\Chat\Tests\MessageTest::it_can_send_a_message
Error: Call to undefined method Tashtin\Chat\Chat::messages()
```

Let’s add a MessageService to contain all our messaging-related code as we did with conversations.

```
$ touch src/Services/MessageService.php
```

src/Services/MessageService.php

```php
<?php

namespace Tashtin\Chat\Services;

class MessageService
{
}
```

src/Chat.php

```php
<?php

namespace Tashtin\Chat;

use Tashtin\Chat\Services\ConversationService;
// markua-start-insert
use Musonza\Chat\Services\MessageService;
// markua-end-insert

class Chat
{
    protected $conversationService;
    // markua-start-insert
    protected $messageService;
    // markua-end-insert

    public function __construct(
        ConversationService $conversationService,
        // markua-start-insert
        MessageService $messageService
        // markua-end-insert
    ) {
        $this->conversationService = $conversationService;
        // markua-start-insert
        $this->messageService = $messageService;
        // markua-end-insert
    }

    public function conversations(): ConversationService
    {
        return $this->conversationService;
    }

    // markua-start-insert
    public function messages(): MessageService
    {
        return $this->messageService;
    }
    // markua-end-insert
}
```

Running the tests again

```
There was 1 error:

1) Tashtin\Chat\Tests\MessageTest::it_can_send_a_message
Error: Call to undefined method Tashtin\Chat\Services\MessageService::body()
```

Let's add the method to set the message body:

src/Services/MessageService.php

```
<?php

namespace Tashtin\Chat\Services;

class MessageService
{
    protected  $body;

    public function body(string $body): self
    {
        $this->body = $body;

        return $this;
    }
}
```

Rerun the tests

```
1) Tashtin\Chat\Tests\MessageTest::it_can_send_a_message
Error: Call to undefined method Tashtin\Chat\Services\MessageService::from()
```

Let’s add the method to set the sender

```
<?php

namespace Tashtin\Chat\Services;

use Illuminate\Database\Eloquent\Model;

class MessageService
{
    protected  $body;
    // markua-start-insert
    protected $sender;
    // markua-end-insert

    public function body(string $body): self
    {
        $this->body = $body;
        return $this;
    }

    // markua-start-insert
    public function from(Model $sender): self
    {
        $this->sender = $sender;
        return $this;
    }
    // markua-end-insert
}
```

Rerun the tests

```
1) Tashtin\Chat\Tests\MessageTest::it_can_send_a_message
Error: Call to undefined method Tashtin\Chat\Services\MessageService::to()
```

src/Services/MessageService.php

```php
<?php

namespace Tashtin\Chat\Services;

use Illuminate\Database\Eloquent\Model;
use Tashtin\Chat\Models\Conversation;

class MessageService
{
    protected  $body;
    protected $sender;
    // markua-start-insert
    protected $conversation;
    // markua-end-insert

    public function body(string $body): self
    {
        $this->body = $body;
        return $this;
    }

    public function from(Model $sender): self
    {
        $this->sender = $sender;
        return $this;
    }

    // markua-start-insert
    public function to(Conversation $conversation): self
    {
        $this->conversation = $conversation;
        return $this;
    }
    // markua-end-insert
}
```

Run the tests

```
1) Tashtin\Chat\Tests\MessageTest::it_can_send_a_message
Error: Call to undefined method Tashtin\Chat\Services\MessageService::send()
```

Add the following function to the end of MessageService class.

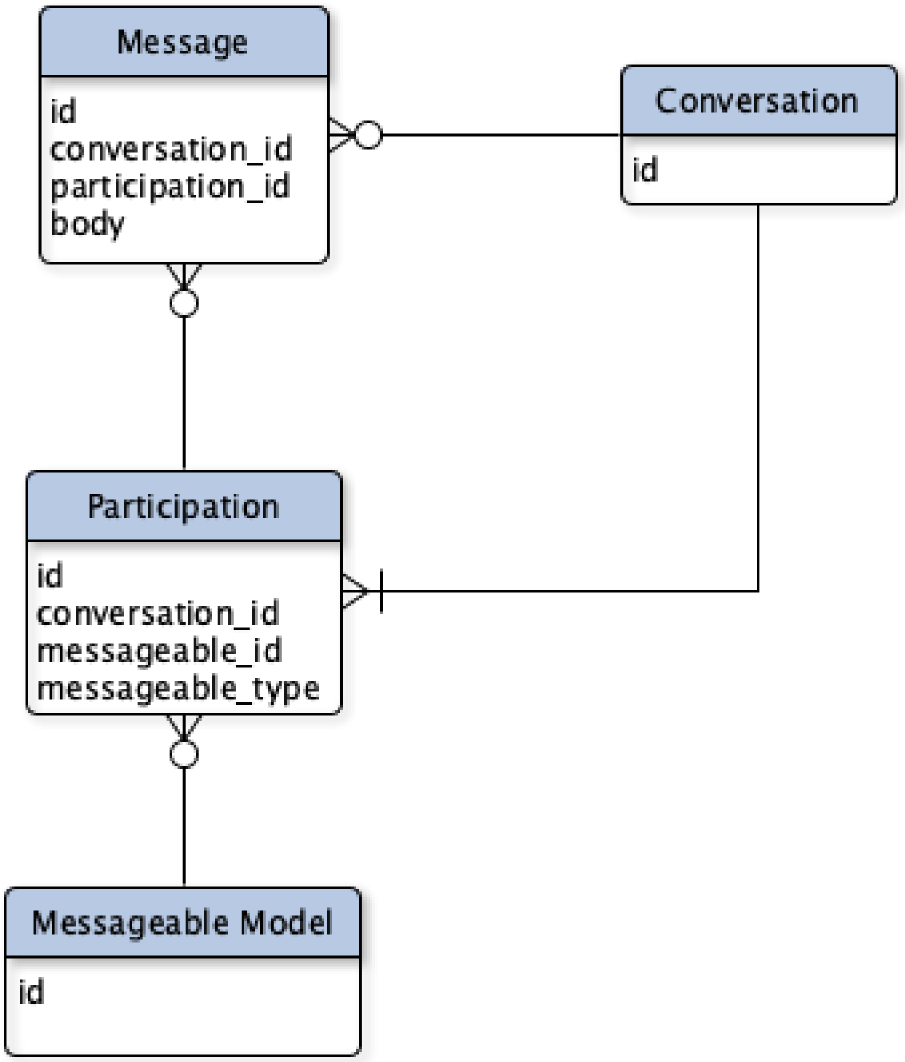
```
public function send()
{
}
```

We are not doing anything with the message, so the test will still fail.

Let's pause for a second and think of how we would store messages for our conversations.

We can't possibly store a message for each user in a table entry as it would create unnecessary duplication. So how will we reference the message for the sender? How can we mark the message as read or deleted?

The following is a diagram of how we will set up our messages.



We can store the message sender's participation id for each message. That will allow us to resolve the sender (messageable Model). So let's set up our `send` method.

```
public function send()
{
    return Message::create([
        'body'             => $this->body,
        'participation_id' => $this->conversation
            ->participantFromSender($this->sender)->id,
        'conversation_id' => $this->conversation->getKey()
    ]);
}
```

```
1) Tashtin\Chat\Tests\MessageTest::it_can_send_a_message
Error: Class 'Tashtin\Chat\Services\Message' not found
```

Let's create the **Message** model

```
touch src/Models/Message.php
```

src/Models/Message.php

```php
<?php

namespace Tashtin\Chat\Models;

use Illuminate\Database\Eloquent\Model;

class Message extends Model
{
    protected $fillable = [
        'body',
        'participation_id',
        'conversation_id',
    ];

    // Link sender via participation
    public function participation()
    {
        return $this->belongsTo(Participation::class, 'participation_id');
    }

    public function conversation()
    {
        return $this->belongsTo(Conversation::class, 'conversation_id');
    }
}
```

Import the model in your MessageService class by adding `use Tashtin\Chat\Models\Message;` at the top.

Rerun the tests, and you will get an error containing the following:

```
Caused by
PDOException: SQLSTATE[HY000]: General error: 1 no such table: messages
```

Let's head over to our migrations and set up our messages table migration by making the following additions to the `up` and `down` functions.

database/migrations/create_chat_tables.php

```php
public function up()
{
    // ...
    Schema::create('messages', function (Blueprint $table) {
        $table->bigIncrements('id');
        $table->text('body');
        $table->bigInteger('conversation_id')->unsigned();
        $table->bigInteger('participation_id')->unsigned()->nullable();
        $table->timestamps();

        $table->foreign('participation_id')
            ->references('id')
            ->on('participation')
            // We want to keep the messages when the sender is deleted
            ->onDelete('set null');

        $table->foreign('conversation_id')
            ->references('id')
            ->on('conversations')
            ->onDelete('cascade');
    });
}
```

```php
public function down()
{
    Schema::dropIfExists('messages');
    // ...
}
```

We need to obtain the participation id from the conversation and sender. So we will call `participantFromSender` on the conversation model.

Run the tests

```
1) Tashtin\Chat\Tests\MessageTest::it_can_send_a_message
BadMethodCallException: Call to undefined method Tashtin\Chat\Models\Conversation::p\
articipantFromSender()
```

Add the following function to `Conversation` model

Conversation::participantFromSender

```php
public function participantFromSender(Model $sender)
{
    return $this->participants()->where([
        'conversation_id'  => $this->getKey(),
        'messageable_id'   => $sender->getKey(),
        'messageable_type' => get_class($sender),
    ])->first();
}
```

Rerunning the tests, we get a failure:

```
1) Tashtin\Chat\Tests\MessageTest::it_can_send_a_message
Error: Call to a member function count() on null
```

We have this assertion

```php
$this->assertEquals($conversation->messages->count(), 1);
```

We need to add messages relationship to the Conversation model

```php
<?php

namespace Tashtin\Chat\Models;

use Illuminate\Database\Eloquent\Model;

class Conversation extends Model
{
    // ...
    public function messages()
    {
        return $this->hasMany(Message::class, 'conversation_id');
    }
    // ...
}
```

Run the tests again and we should see green.

We have our tests passing, but it doesn't mean our code is fully functional. So what happens if we don't provide the message body or conversation and still attempt to send a message? Our code with throw an error. Instead, we should perform validations and throw exceptions to our package users.

```
/** @test */
public function it_validates_message_body_presence()
{
    $this->expectException(\InvalidArgumentException::class);

    $conversation = Chat::conversations()
        ->createConversation([$this->userAlpha, $this->userBravo]);

    Chat::messages()
        ->from($this->userBravo)
        ->to($conversation)
        ->send();
}
```

Running the test fails

```
Failed asserting that exception of type "Illuminate\Database\QueryException" matches\
 expected exception "InvalidArgumentException"
```

Let's update our send message function

```
public function send(): Message
{
    //markua-start-insert
    if (!strlen($this->body)) {
        throw new \InvalidArgumentException('Message body not set');
    }
    //markua-end-insert

    return Message::create([
        'body'             => $this->body,
        'participation_id' => $this->sender->getKey(),
        'conversation_id' => $this->conversation->getKey(),
    ]);
}
```

Rerun the tests, and we should see green. Let's add additional tests for other data that we expect. We want to make sure the sender and recipient/conversation are specified.

```
/** @test */
public function it_validates_message_sender_presence()
{
    $this->expectException(\InvalidArgumentException::class);

    $conversation = Chat::conversations()
        ->createConversation([$this->userAlpha, $this->userBravo]);

    Chat::messages()
        ->body('Hello')
        ->to($conversation)
        ->send();
}

/** @test */
public function it_validates_message_conversation_presence()
{
    $this->expectException(\InvalidArgumentException::class);
    Chat::messages()
        ->body('Hello')
        ->from($this->userBravo)
        ->send();
}
```

Run the tests and it fails

```
1) Tashtin\Chat\Tests\MessageTest::it_validates_message_sender_presence
Failed asserting that exception of type "Error" matches expected exception "InvalidA\
rgumentException". Message was: "Call to a member function getKey()
```

Let’s add a check for the message sender

```
public function send(): Message
{
    if (!strlen($this->body)) {
        throw new \InvalidArgumentException('Message body not set');
    }

    //markua-start-insert
    if (!$this->sender) {
        throw new \InvalidArgumentException('Message sender not set');
    }
    //markua-end-insert

    return Message::create([
        'body'             => $this->body,
        'participation_id' => $this->sender->getKey(),
        'conversation_id' => $this->conversation->getKey(),
    ]);
}
```

Run the tests again

```
1) Tashtin\Chat\Tests\MessageTest::it_validates_message_conversation_presence
Failed asserting that exception of type "Error" matches expected exception "InvalidA\
rgumentException". Message was: "Call to a member function getKey()
```

```
public function send(): Message
{
    if (!strlen($this->body)) {
        throw new \InvalidArgumentException('Message body not set');
    }

    if (!$this->sender) {
        throw new \InvalidArgumentException('Message sender not set');
    }

    //markua-start-insert
    if (!$this->conversation) {
        throw new \InvalidArgumentException('Message recipient not set');
    }
    //markua-end-insert

    return Message::create([
        'body'             => $this->body,
        'participation_id' => $this->sender->getKey(),
        'conversation_id' => $this->conversation->getKey(),
    ]);
}
```

It can specify the message type

Right now, it seems our package will only support regular text messages. For example, we would like to send images, video links, or URL links like s3 bucket URLs. However, we will still use the `body` field and specify a message **`type`**to achieve this. It will be up to the package users to figure out how to process and store the media sent in messages. All that the package will do is store the generated URL.

Let's write the test for that.

```
/** @test **/
public function it_can_specify_message_type()
{
    $conversation = Chat::conversations()
        ->createConversation([$this->userAlpha, $this->userBravo]);

    $message = Chat::messages()
        ->type('image')
        ->body('http://example.com/my-cool-image')
        ->from($this->userBravo)
        ->to($conversation)
        ->send();

    $this->assertEquals('image', $message->type);
}
```

Run the tests

```
1) Tashtin\Chat\Tests\MessageTest::it_can_specify_message_type
Error: Call to undefined method Tashtin\Chat\Services\MessageService::type()
```

```
<?php

// ...

class MessageService
{
    // ...
    protected $type = 'text';
    // ...

    // ...
    public function type(string $type): self
    {
        $this->type = $type;
        return $this;
    }
    // ...
}
```

```
1) Tashtin\Chat\Tests\MessageTest::it_can_specify_message_type
Failed asserting that null matches expected 'image'.
```

We need to add the field `type` to our message model that we will use to track our message types. Add the following line to **message** model migration.

```
$table->string('type')->default('text');
```

In addition, your message model **`$fillable`** property should look as follows:

```
protected $fillable = [
    'body',
    'participation_id',
    'conversation_id',
    'type',
];
```

Run the tests and we should see green.

It returns a message given the id

```
/** @test */
public function it_returns_a_message_given_the_id()
{
    $conversation = Chat::conversations()
        ->createConversation([$this->userAlpha, $this->userBravo]);

    $message = Chat::messages()
        ->body('Hello')
        ->from($this->userBravo)
        ->to($conversation)
        ->send();

    $m = Chat::messages()->getById($message->getKey());

    $this->assertEquals($message->getKey(), $m->getKey());
}
```

Run the tests…

```
1) Tashtin\Chat\Tests\MessageTest::it_returns_a_message_given_the_id
Error: Call to undefined method Tashtin\Chat\Services\MessageService::getById()
```

```
public function getById($id)
{
    return Message::findOrFail($id);
}
```

Run the tests and you should see green.

It creates participant message references

We want a way to tell how many new messages a user has…

How many messages are in a conversation…

We will use this to also track read and unread messages…delete messages…

```
/** @test */
public function it_creates_participant_message_references()
{
    $conversation = Chat::conversations()
        ->createConversation([$this->userAlpha, $this->userBravo]);

    Chat::messages()
        ->body('Hello 1')
        ->from($this->userBravo)
        ->to($conversation)
        ->send();

    Chat::messages()
        ->body('Hello 2')
        ->from($this->userAlpha)
        ->to($conversation)
        ->send();

    Chat::messages()
        ->body('Hello 3')
        ->from($this->userAlpha)
        ->to($conversation)
        ->send();
    $this->assertEquals(3, $conversation->getMessages($this->userAlpha)->count());
    $this->assertEquals(3, $conversation->getMessages($this->userBravo)->count());
    $this->assertEquals(0, $conversation->getMessages($this->userCharlie)->count());
}
```

```
1) Tashtin\Chat\Tests\MessageTest::it_creates_message_notification
BadMethodCallException: Call to undefined method Tashtin\Chat\Models\Conversation::g\
etMessages()
```

Add **getMessages** ...

```
public function getMessages(Model $participant)
{
}
```

Run the tests...

```
1) Tashtin\Chat\Tests\MessageTest::it_creates_message_notification
Error: Call to a member function count() on null
```

```
public function getMessages(Model $participant)
{
        // one message => multiple participants
        return $this->messages()
            ->join(
                'participant_messages',
                'participant_messages.message_id',
                '=',
                'messages.id'
            )
            ->join(
                'participation',
                'participation.id',
                '=',
                'participant_messages.participation_id'
            )
            ->where('participation.messageable_type', get_class($participant))
            ->where('participation.messageable_id', $participant->getKey())
            ->get();
}
```

Run the tests

```
Caused by
PDOException: SQLSTATE[HY000]: General error: 1 no such table: participant_messages
```

Let's create the migration for **`participant_messages`**

Add the following to the bottom of the **`up()`** function in *database/migrations/create_chat_tables.php*

```
Schema::create('participant_messages', function (Blueprint $table) {
    $table->bigIncrements('id');
    $table->bigInteger('message_id')->unsigned();
    $table->bigInteger('participation_id')->unsigned();
    $table->timestamps();

    $table->index(['participation_id', 'message_id'], 'participation_message_index');

    $table->foreign('message_id')
        ->references('id')
        ->on('messages')
        ->onDelete('cascade');

    $table->foreign('participation_id')
        ->references('id')
        ->on('participation')
        ->onDelete('cascade');
});
```

To the beginning of the **`down()`** function add:

```
Schema::dropIfExists('participant_messages');
```

Run the tests

```
1) Tashtin\Chat\Tests\MessageTest::it_creates_message_notification
Failed asserting that 0 matches expected 3.
```

When we send a message, we aren't making an entry in **`participant_messages`** for each participant yet. So let's work on that.

We can take advantage of laravel eloquent model events for this. According to Laravel documentation, eloquent models dispatch several events, allowing you to hook into the following moments in a model's lifecycle: retrieved, creating, created, updating, updated, saving, saved, deleting, deleted, restoring, restored, and replicating.

```
<?php

// ...

class Message extends Model
{
    // ...

    public static function boot()
    {
        parent::boot();

        self::created(function($message){
            ParticipantMessages::make($message);
        });
    }

    // ...
}
```

Let’s also create the `ParticipantMessages` model

```
touch src/Models/ParticipantMessages.php
```

```
<?php

namespace Tashtin\Chat\Models;

use Illuminate\Database\Eloquent\Model;

class ParticipantMessages extends Model
{

}
```

Re-run the tests

```
Error: Call to undefined method Tashtin\Chat\Models\ParticipantMessages::make()
```

Let’s create the method

```php
<?php

namespace Tashtin\Chat\Models;

use Illuminate\Database\Eloquent\Model;

class ParticipantMessages extends Model
{
    //markua-start-insert
    public static function make(Message $message)
    {
        $data = [];

        // create a message entry for each participant
        foreach ($message->conversation->participants as $participation) {
            $data[] = [
                'message_id'       => $message->getKey(),
                'participation_id' => $participation->id,
            ];
        }

        self::insert($data);
    }
    //markua-start-insert
}
```

Run the tests again and you should see green.

Read / Delete messages

We want to tell if a message is read or deleted by a user.

It can mark a message as read

Add a test case to MessageTest.php

```
/** @test */
public function it_can_mark_a_message_as_read()
{
    $conversation = Chat::conversations()
        ->createConversation([$this->userAlpha, $this->userBravo]);

    $message = Chat::messages()
        ->body('Hello')
        ->from($this->userBravo)
        ->to($conversation)
        ->send();

    Chat::messages()->markRead($message, $this->userAlpha);

    // We need to know the participation entry for the our participant
    // in the conversation
    $participation = $this->userAlpha->participation
        ->where('conversation_id', $conversation->id)
        ->first();

    $this->assertDatabaseHas('participant_messages', [
        'message_id' => $message->id,
        'participation_id' => $participation->id,
        'is_seen' => 1, // we have marked the message as read / seen
    ]);
}
```

Run the tests

```
1) Tashtin\Chat\Tests\MessageTest::it_can_mark_a_message_as_read
Error: Call to undefined method Tashtin\Chat\Services\MessageService::markRead()
```

Let’s add a function to mark messages as read.

```
public function markRead(Message $message, Model $model)
{
    ParticipantMessages::join(
        'participation', 'participation.id', '=', 'participation_id'
    )
        ->where('participation.messageable_id', $model->getKey())
        ->where('participation.messageable_type', get_class($model))
        ->where('message_id', $message->getKey())
        ->firstOrFail()
        ->update(['is_seen' => 1]);
}
```

Run the tests

```
1) Tashtin\Chat\Tests\MessageTest::it_can_mark_a_message_as_read
Illuminate\Database\Eloquent\MassAssignmentException: Add [is_seen] to fillable prop\
erty to allow mass assignment on [Tashtin\Chat\Models\ParticipantMessages]
```

Add a **`$fillable`** property to **`ParticipantMessages`** model

```
protected $fillable = [
    'is_seen'
];
```

Let’s also add the column to our `participant_messages` migration where `0` will represent an unread message, and `1` signifies a read message.

```
$table->enum('is_seen', [0, 1])->default(0);
```

Rerun the tests, and you should see green.

It can tell whether a participating model has read a message

Now that we can mark messages as read wouldn’t be excellent for our package users to tell if a specific participating model has a particular message read. The use case for this may not be apparent right off the bat, but it’s important to understand we are developing this package for users who have unlimited use cases. Therefore, providing more functionality is always better.

So let’s go ahead and set up our testcase in `MessageTest.php`

```php
/** @test */
public function it_can_tell_participant_has_read_message()
{
    $conversation = Chat::conversations()
        ->createConversation([$this->userAlpha, $this->userBravo]);

    $message = Chat::messages()
        ->body('Hello')
        ->from($this->userBravo)
        ->to($conversation)
        ->send();

    $this->assertFalse(Chat::messages()->isRead($message, $this->userAlpha));
    Chat::messages()->markRead($message, $this->userAlpha);
    $this->assertTrue(Chat::messages()->isRead($message, $this->userAlpha));
}
```

Here we have sent a message to participant `$this->userAlpha`, and we will check if the message is read. We also will mark the message as read and test if our functionality works.

Run the tests, and we get the following failure.

```
Error: Call to undefined method Tashtin\Chat\Services\MessageService::isRead()
```

Let's go ahead and create that function in the `MessageService`. This function will query our `participant_messages` table for the participant and the specific message where the column `is_seen` is true.

```php
public function isRead(Message $message, Model $model)
{
        return !!ParticipantMessages::join(
            'participation', 'participation.id', '=', 'participation_id'
        )
            ->where('participation.messageable_id', $model->getKey())
            ->where('participation.messageable_type', get_class($model))
            ->where('message_id', $message->getKey())
            ->firstOrFail()
            ->is_seen;
}
```

Rerun the tests, and we should see green.

It can mark a conversation as read

We can mark messages as read now. What if we want to mark all messages in a conversation as read? I think that's a perfect use case. Like always, let's start by crafting our test case. In `ConversationTest.php`, we are going to add the following:

```php
/** @test */
public function it_can_mark_a_conversation_as_read()
{
    /** @var Conversation $conversation */
    $conversation = Chat::createConversation([
        $this->alpha,
        $this->bravo,
    ])
}
```

Events

What are events?

Events in an application help us with decoupling various implementations. For instance, when a message is sent, we could send push notifications, email notifications, etc., to the recipient. However, we certainly can't cover all use cases in our package, so we will defer the implementation of such code to the end-user. Firing events will allow our users to listen for these events and take necessary action in their applications. It's worth noting that there is no harm in ignoring and not listening for the emitted events for our package users.

Our package events

Here are some events that we are going to add to our package:

ConversationStarted

his event will be fired when a conversation is created, which can help notify participants that they have been added to a group chat.

ParticipantsJoined

When a new participant is added to a conversation, we will fire this event which can be helpful in applications where you want to inform other participants that a new member has joined.

ParticipantsLeft

This event will be fired when participants leave a conversation.

MessageWasSent

This event will be fired each time a message is sent. As a package user, I could listen for this event and send notifications to all the recipients. In addition, I could update my user interface with the new message using web sockets if the event is being broadcasted.

Let's get started writing our events tests by creating a class to hold these tests.

```
touch tests/Unit/EventsTest.php
```

Paste the following code

```php
<?php

namespace Tashtin\Chat\Tests\Unit;

use Tashtin\Chat\Tests\TestCase;

class EventsTest extends TestCase
{
}
```

We will be testing that events are indeed dispatched with expected data. For example, a `ConversationStarted` event makes sense with the conversation data. However, it is essential to include only the necessary data because these events can be used with broadcasting services like Pusher, which impose a message size limit.

ConversationStarted

Go ahead and start with a test for `ConversationStarted` event.

```php
<?php

namespace Tashtin\Chat\Tests\Unit;

// markua-start-insert
use Chat;
use Illuminate\Support\Facades\Event;
use Tashtin\Chat\Events\ConversationStarted;
// markua-end-insert
use Tashtin\Chat\Tests\TestCase;

class EventsTest extends TestCase
{
    // markua-start-insert
    /** @test */
    public function it_dispatches_conversation_started_event()
    {
        Event::fake();

        $conversation = Chat::conversations()->createConversation();

        Event::assertDispatched(ConversationStarted::class,
            function ($event) use ($conversation) {
                return $event->conversation->id === $conversation->id;
        });
    }
    // markua-end-insert
}
```

You notice we are using `Event::fake()` before running our test. This method will prevent all event listeners from executing, allowing us to focus on whether the events were dispatched and inspect the data they receive.

Running the test will result in the below failure.

```
There was 1 failure:

1) Tashtin\Chat\Tests\Unit\EventsTest::it_dispatches_conversation_started_event
The expected [Tashtin\Chat\Events\ConversationStarted] event was not dispatched.
Failed asserting that false is true.
```

Create `Events` directory and `ConversationStarted.php` file for our Event

```
mkdir src/Events && touch src/Events/ConversationStarted.php
```

Add the following to our event

```php
<?php

namespace Tashtin\Chat\Events;

use Tashtin\Chat\Models\Conversation;

class ConversationStarted
{
    /**
     * @var Conversation
     */
    public $conversation;

    public function __construct(Conversation $conversation)
    {
        $this->conversation = $conversation;
    }
}
```

Running the test still fails. So let's go and look at where we are creating a conversation in `ConversationService.php` and fire the event.

```php
public function createConversation($participants = [], $data = []): Conversation
{
    $conversationData = (new ConversationData($data))
        ->filtered()
        ->toArray();

    $this->conversation = $this->conversation
        ->create($conversationData);

    $this->addParticipants($participants);

    // markua-start-insert
    event(new ConversationStarted($this->conversation));
    // markua-end-insert

    return $this->conversation;
}
```

Remember to import the event at the top of the class as `use Tashtin\Chat\Events\ConversationStarted;`.

Running the tests again we should see green.

ParticipantsJoined

We may want to notify when additional participants join a conversation in some cases. For this reason, we will emit a `ParticipantsJoined` event each time we add a participant to a conversation.

Go ahead and add a test for that.

```php
/** @test */
public function it_dispatches_participants_joined_event()
{
    Event::fake();

    $conversation = Chat::conversations()->createConversation();
    Chat::conversations()
        ->setConversation($conversation)
        ->addParticipants([$this->userAlpha]);

    Event::assertDispatched(ParticipantsJoined::class,
        function ($event) use ($conversation) {
            return $event->conversation->id === $conversation->id
                && in_array($this->userAlpha, $event->participants);
    });
}
```

Here we create a conversation and then add a participant. Adding a participant should result in the `ParticipantsJoined` event emission.

```
There was 1 failure:

1) Tashtin\Chat\Tests\Unit\EventsTest::it_dispatches_participants_joined_event
The expected [Tashtin\Chat\Events\ParticipantsJoined] event was not dispatched.
Failed asserting that false is true.
```

Of course, we have not added the code to dispatch the event, so our test fails. We need to dispatch this event after adding participant(s). So let's go to our `ConversationService` class and update `addParticipants` to the following:

```php
public function addParticipants(array $participants)
{
    $participation = [];

    foreach ($participants as $participant) {
        $participation[] = [
            'conversation_id' => $this->conversation->getKey(),
            'messageable_id' => $participant->getKey(),
            'messageable_type' => get_class($participant),
        ];
    }

    $this->participation->insert($participation);

    // markua-start-insert
    event(new ParticipantsJoined($this->conversation));
    // markua-end-insert
}
```

Lastly, let's create the event

```php
<?php

namespace Tashtin\Chat\Events;

use Tashtin\Chat\Models\Conversation;

class ParticipantsJoined
{
    /**
     * @var Conversation
     */
    public $conversation;

    public $participants;

    public function __construct(Conversation $conversation, array $participants)
    {
        $this->conversation = $conversation;
        $this->participants = $participants;
    }
}
```

Remember to import the event at the top of the class by adding `use Tashtin\Chat\Events\ParticipantsJoined;`.

Now run your test suite and you should see green.

It's worth noting that getting passing tests doesn't always mean your code is working as expected. For instance, in our test for `ParticipantsJoined`, we only tested whether the event is emitted. However, we are not checking how many times the event is emitted. Thankfully, Laravel has a handy way to assert how many times an event is emitted.

Here is how you would assert that an event was emitted once

```php
// Assert an event was dispatched once...
Event::assertDispatched(ParticipantsJoined::class, 1);
```

Add that line of code to your `it_dispatches_participants_joined_event` test as following:

```php
/** @test */
public function it_dispatches_participants_joined_event()
{
    Event::fake();

    $conversation = Chat::conversations()->createConversation();
    Chat::conversations()
        ->setConversation($conversation)
        ->addParticipants([$this->userAlpha]);

    Event::assertDispatched(ParticipantsJoined::class,
        function ($event) use ($conversation) {
            return $event->conversation->id === $conversation->id;
    });

    // markua-start-insert
    // Assert an event was dispatched once...
    Event::assertDispatched(ParticipantsJoined::class, 1);
    // markua-end-insert
}
```

Rerun the tests, and you will see a failure

```
There was 1 failure:

1) Tashtin\Chat\Tests\Unit\EventsTest::it_dispatches_participants_joined_event
The expected [Tashtin\Chat\Events\ParticipantsJoined] event was dispatched 2 times i\
nstead of 1 time.
Failed asserting that false is true.
```

It appears our event is being dispatched twice instead of once. We can add participants when we create a conversation, resulting in the event being emitted. At the moment, we are always calling the `addParticipants` function even if we have no participants provided.

Let's make an update to `createConversation` as follows:

```php
public function createConversation($participants = [], $data = []): Conversation
    {
        $conversationData = (new ConversationData($data))
            ->filtered()
            ->toArray();

        $this->conversation = $this->conversation
            ->create($conversationData);

        // markua-start-insert
        if (!empty($participants)) {
        // markua-end-insert
            $this->addParticipants($participants);
        // markua-start-insert
        }
        // markua-end-insert

        event(new ConversationStarted($this->conversation));

        return $this->conversation;
    }
```

Rerun the tests, and you should see green.

ParticipantsLeft

Just like we have added an event for when new participants join a Conversation, we also need to add an event for when participants leave a conversation.

Let's go ahead and create a test for that.

```php
/** @test */
    public function it_dispatches_participants_left_event()
    {
        Event::fake();

        $conversation = Chat::conversations()
            ->createConversation([$this->userAlpha, $this->userBravo]);

        Chat::conversations()
            ->setConversation($conversation)
            ->removeParticipants([$this->userAlpha]);

        Event::assertDispatched(ParticipantsLeft::class,
            function ($event) use ($conversation) {
                return $event->conversation->id === $conversation->id
                    && in_array($this->userAlpha, $event->participants);
            });

        Event::assertDispatched(ParticipantsLeft::class, 1);
    }
```

Running the test fails. Let's create the event.

```
touch src/Events/ParticipantsLeft.php
```

Add the following code to the class

```php
<?php

namespace Tashtin\Chat\Events;

use Tashtin\Chat\Models\Conversation;

class ParticipantsLeft
{
    /**
     * @var Conversation
     */
    public $conversation;

    public $participants;

    public function __construct(
        Conversation $conversation,
        array $participants
    ) {
        $this->conversation = $conversation;
    }
}
```

Now, look for the code that removes participants from conversations in `ConversationService`. Make updates to `removeParticipants` so that it looks as below

```php
public function removeParticipants(array $participants)
{
    foreach ($participants as $participant) {
        $this->participation->where([
            'conversation_id' => $this->conversation->getKey(),
            'messageable_id' => $participant->getKey(),
            'messageable_type' => get_class($participant),
        ])->delete();
    }

    // markua-start-insert
    event(new ParticipantsLeft($this->conversation, $participants));
    // markua-end-insert
}
```

Run the test suite, and you should see green.

MessageWasSent

Lastly, we will add an event that will be fired each time a message is sent.

Like always, let’s begin by writing a test for that.

```php
 /** @test */
public function it_dispatches_message_was_sent_event()
{
    Event::fake();

    $conversation = Chat::conversations()
        ->createConversation([$this->userAlpha, $this->userBravo]);

    $message = Chat::messages()
        ->body('Hello')
        ->from($this->userBravo)
        ->to($conversation)
        ->send();

    Event::assertDispatched(\Tashtin\Chat\Events\MessageWasSent::class,
        function ($event) use ($message) {
            return $event->message->id === $message->id;
        });

    Event::assertDispatched(MessageWasSent::class, 1);
}
```

This time we are going to pass our message object to the event. That should be enough information for our package users. They will get the conversation or message sender from the message object.

Run the tests, and we get the following error.

```
Time: 1.7 seconds, Memory: 22.00 MB

There was 1 failure:

1) Tashtin\Chat\Tests\Unit\EventsTest::it_dispatches_message_was_sent_event
The expected [Tashtin\Chat\Tests\Unit\MessageWasSent] event was not dispatched.
Failed asserting that false is true.
```

Create the event

```
touch src/Events/MessageWasSent.php
```

Add the following to the class

```
<?php

namespace Tashtin\Chat\Events;

use Tashtin\Chat\Models\Message;

class MessageWasSent
{
    /**
     * @var Message
     */
    public $message;

    public function __construct(Message $message)
    {
        $this->message = $message;
    }
}
```

Now let's emit the event when we send a message. Update the `send` method in `MessageService`

```
 public function send(): Message
{
    if (!strlen($this->body)) {
        throw new \InvalidArgumentException('Message body not set');
    }

    if (!$this->sender) {
        throw new \InvalidArgumentException('Message sender not set');
    }

    if (!$this->conversation) {
        throw new \InvalidArgumentException('Message recipient not set');
    }

    $message = Message::create([
        'body'             => $this->body,
        'participation_id' => $this->conversation
            ->participantFromSender($this->sender),
        'conversation_id' => $this->conversation->getKey(),
        'type' => $this->type,
    ]);

    event(new MessageWasSent($message));

    return $message;
}
```

Rerunning the tests, you should see green.

These events should be sufficient for now. As you develop packages, you will have an opportunity to make additional updates and bump the package version.

In the next chapter, we will look at adding routes to our package. For example, our package users may prefer to call ready-made endpoints, so let's provide that for them.

Routes in packages

We will be adding routes that package users can call to perform all the functionality that we have worked on so far. An example use case would be a mobile app calling these API endpoints/routes. Let's take time to plan how we will organize our routes in the application. In addition to routes, we will need to add controllers and possibly form request classes. So I think we can try to mirror our Laravel applications by creating the following structure:

src/Http

src/Http/Controllers

src/Http/Requests

Our routes will live in the *chat/src/Http/routes.php* file.

Go ahead and create the folders above and the *routes.php* file.

```
$ mkdir src/Http && mkdir src/Http/Controllers && mkdir src/Http/Requests
$ touch src/Http/routes.php
```

Firstly, let's figure out how to get Laravel to recognize our package routes. Add the following to routes.php

src/Http/routes.php

```
<?php

Route::group([
    'middleware' => [],
    'prefix'     => '',
], function () {
    /* Test */
    Route::get('/alive', function () {
        return 'It works';
    });
});
```

We have added a route to test if our routing works. To verify this, let's create a simple test in *tests/Unit/ExampleTest.php*

Red, Green

tests/Unit/ExampleTest.php

```php
<?php

namespace Tashtin\Chat\Tests\Unit;

use Tashtin\Chat\Tests\TestCase;

class ExampleTest extends TestCase
{
    public function testExample()
    {
        $this->assertEquals(1, 1);
    }

    public function testExampleRoute()
    {
        $response = $this->get('alive');
        $this->assertEquals('It works', $response->getOriginalContent());
    }
}
```

Running the test we get a failure similar to the following

```
1) Tashtin\Chat\Tests\Unit\ExampleTest::testExampleRoute
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'It works'
+'<!DOCTYPE html>\n
+<html lang="en">\n
+    <head>\n
+        <meta charset="utf-8">\n
+        <meta name="viewport" content="width=device-width, initial-scale=1">\n
+\n
+        <title>Not Found</title>\n
```

We need to tell Laravel where to find our routes file, and if you recall, the right place to do this is in our `ChatServiceProvider`, precisely the `boot` method. Let's include the `route.php` file.

Your `boot` method should now look as follows.

```
public function boot()
{
    require __DIR__.'/Http/routes.php';
}
```

Rerun the tests, and you should see green. Later on, we will make sure the loading of routes is configurable because some users will want an option to disable the routes.

Unit tests vs Feature tests

While there are different types of tests you could have in an application, we will focus on unit and feature tests. For example, we have been working with unit tests that focus on testing the functionality of small parts of the application. We will now write features tests for testing our package routes - these are tests that verify that an entire application feature works as expected.

Create a folder to store our feature tests:

```
$ mkdir tests/Feature
```

Conversation

We are going to write tests related to conversations in `ConversationControllerTest`

Let's create that file.

```
$ touch tests/Feature/ConversationControllerTest.php
```

Add the following snippet to your class.

POST /conversations

Red, Green

Our first test will verify that we get a successful response from the server when we make an API call. In addition, the test should verify that a conversation is created.

tests/Feature/ConversationControllerTest.php

```php
<?php

namespace Tashtin\Chat\Tests\Feature;

use Tashtin\Chat\Tests\TestCase;

class ConversationControllerTest extends TestCase
{
    public function testStore()
    {
        $this->postJson('/conversations')
            ->assertSuccessful();

        $this->assertDatabaseHas(
            'conversations',
            ['id' => 1]
        );
    }
}
```

Run the tests, and you may notice all your tests passing. At the moment, PHPUnit is only aware of our unit tests only. So let's update the PHPUnit configuration in `phpunit.xml` to be aware of tests in our `Features` folder.

Your `testsuites` section should look as follows:

```xml
<testsuites>
    <testsuite name="Chat Package Test Suite">
        <directory suffix=".php">./tests/Unit</directory>
    </testsuite>
    <testsuite name="Feature">
        <directory suffix=".php">./tests/Feature</directory>
    </testsuite>
</testsuites>
```

Rerun the tests, and you should see the following failure.

```
1) Tashtin\Chat\Tests\Feature\ConversationControllerTest::testStore
Symfony\Component\HttpKernel\Exception\NotFoundHttpException: POST http://localhost/\
conversations
```

So we are missing the route to store a conversation. Add that in your routes file below the test route we added earlier.

src/Http/routes.php

```php
<?php

Route::group([
    'middleware' => [],
    'prefix'     => '',
], function () {
    /* Test */
    Route::get('/alive', function () {
        return 'It works';
    });

    /** Conversations */
    Route::post('/conversations', 'ConversationController@store');
});
```

Now rerun the tests.

```
Conversation Controller (Tashtin\Chat\Tests\Feature\ConversationController)
Expected status code 200 but received 500.
Failed asserting that false is true.
```

We know why this is failing, but the error we get doesn't tell us why. This is because Laravel is handling exceptions that may have occurred. For feature tests, it might be necessary to disable exception handling. Luckily, Laravel has a handy method to do just that. You can call the method `$this->withoutExceptionHandling()` before running your tests.

Let's update our test class real quick by adding a `setUp` method as follows:

```php
public function setUp(): void
{
    parent::setUp();
    $this->withoutExceptionHandling();
}
```

Rerunning the test we get:

```
Illuminate\Contracts\Container\BindingResolutionException: Target class [Conversatio\
nController] does not exist.
```

We can see that the issue is due to a missing controller. Go ahead and create that controller.

```
$ touch src/Http/Controllers/ConversationController.php
```

Add the following to the class

src/Http/Controllers/ConversationController.php

```php
<?php

namespace Tashtin\Chat\Http\Controllers;

class ConversationController
{
    public function store()
    {
    }
}
```

Rerunning the tests, we will get the same error. By default, Laravel recognizes controllers that are stored under the `App\Http\Controllers` namespace. Any other namespaces will have to be specified by the end-user. All our controllers are going to be in the same namespace, so let's make the following update to our routes file

src/Http/routes.php

```php
<?php

Route::group([
    'middleware' => [],
    'prefix'     => '',
    // markua-start-insert
    'namespace'     => 'Tashtin\Chat\Http\Controllers',
    // markua-end-insert
], function () {
    /* Test */
    Route::get('/alive', function () {
        return 'It works';
    });

    /** Conversations */
    Route::post('/conversations', 'ConversationController@store');
});
```

Rerun the tests, and you will still be in a red state because we are not creating the conversation yet.

We already have the functionality to create a conversation, so it should be a matter of calling the function in our controller.

```php
<?php

namespace Tashtin\Chat\Http\Controllers;

// markua-start-insert
use Chat;
// markua-end-insert

class ConversationController
{
    public function store()
    {
        // markua-start-insert
        $conversation = Chat::conversations()->createConversation();
        return response()->json([
            'data' => $conversation->toArray(),
        ]);
        // markua-end-insert
    }
}
```

In the meantime, we are just returning our created conversation as an array. Then, later on, we will look at data transformers for consistent output. Run the tests, and you should see green.

How about if we want to create a conversation with participants and any extra data? If you recall, we added the ability to add laravel model participants. However, when our package users use the endpoints, they ought to pass string inputs. So for that reason, we are going to accept the following as an example input structure:

Sample input

```
{
    "participants": [
        {
            "id": 1,
            "type": "Tashtin\\Chat\\Tests\\Helpers\\Models\\User"
        },
        {
            "id": 2,
            "type": "Tashtin\\Chat\\Tests\\Helpers\\Models\\Bot"
        }
    ],
    "data": {
        "name": "PHP Channel",
        "description": "This is our test channel"
    }
}
```

So when adding participants, we will specify an identifier and the participant type.

Let's take a stab at writing the test. We could add the assertions in the test we created earlier, but it's best to limit the number of assertions in a test case.

Red, Green

Create a new test `testStoreWithParticipants`

```
public function testStoreWithParticipants()
{
    $participants = [
        [
            'id' => $this->userAlpha->getKey(),
            'type' => $this->userAlpha->getMorphClass()
        ],
        [
            'id' => $this->botAlpha->getKey(),
            'type' => $this->botAlpha->getMorphClass()
        ],
    ];

    $payload = [
        'participants' => $participants,
        'data'         => [
            'name' => 'PHP Channel',
            'description' => 'This is our test channel'
        ],
    ];

    $this->postJson(route('conversations.store'), $payload)
        ->assertSuccessful()
        ->assertJson([
            'data' => $payload['data'],
        ]);

    $this->assertDatabaseHas('participation', [
        'messageable_id'   => $this->userAlpha->getKey(),
        'messageable_type' => $this->userAlpha->getMorphClass(),
    ]);

    $this->assertDatabaseHas('participation', [
        'messageable_id'   => $this->botAlpha->getKey(),
        'messageable_type' => $this->botAlpha->getMorphClass(),
    ]);
}
```

Running the tests we get

```
[{
    "data": {
        "name": "PHP Channel",
        "description": "This is our test channel"
    }
}]

within response JSON:

[{
    "data": {
        "updated_at": "2020-08-08 00:24:23",
        "created_at": "2020-08-08 00:24:23",
        "id": 1
    }
}].
```

Let’s head to our controller and handle the input data that we are passing.

```php
public function store(Request $request)
{
    $participantModels = [];

    foreach ($request->input('participants', []) as $participant) {
        $participantModels[] = app($participant['type'])->find($participant['id']);
    }

    $conversation = Chat::conversations()
        ->createConversation(
            $participantModels,
            $request->input('data', [])
        );

    return response()->json([
        'data' => $conversation->toArray(),
    ]);
}
```

Run the tests, and you should see green. This code works, but it does look ugly to an extent. Now that we are green, we can refactor with confidence.

Refactor

Our `createConversation` function expects the first parameter to be an array of Models, but we are just passing an array of participants with `id` and `type`. We need to resolve the input data to participant models. Hence we have looped through the `participants` input. Since we pass our participants' class names, we can resolve the models from the laravel container by using the `app()` helper and chaining the `find` method to get a specific participant.

While we can handle our input and validation in our controller method, I will opt for a "form request". These are custom classes that we can use to handle validation and transform our input.

Create the form request class by running the following command in the terminal:

```
$ touch src/Http/Requests/StoreConversation.php
```

Add the following code snippet

src/Http/Requests/StoreConversation.php

```php
<?php

namespace Tashtin\Chat\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class StoreConversation extends FormRequest
{
    public function authorized()
    {
        return true;
    }

    public function rules()
    {
        return [
            'participants'        => 'array',
            'participants.*.id'   => 'required',
            'participants.*.type' => 'required|string',
            'data'                => 'array',
        ];
    }

    public function participants()
    {
        $participantModels = [];
        $participants = $this->input('participants', []);
        foreach ($participants as $participant) {
            $participantModels[] = app($participant['type'])
                ->find($participant['id']);
        }

        return $participantModels;
    }
}
```

Laravel form requests expect `authorized` and `rules` functions that check whether the request is authorized and perform input validation. While creating a conversation with participants is not a requirement, we have made sure if any participant data is provided, it should have an `id` and `type`.

While we are at it, we can also add a method to get conversation data from our request like so:

```
public function conversationData()
{
    return  $this->input('data', []);
}
```

Now let’s use the form request in our controller.

src/Http/Controllers/ConversationController.php

```
<?php

namespace Tashtin\Chat\Http\Controllers;

use Chat;
use Tashtin\Chat\Http\Requests\StoreConversation;

class ConversationController
{
    public function store(StoreConversation $request)
    {
        $conversation = Chat::conversations()
            ->createConversation(
                // markua-start-delete
                $request->input('participants'),
                // markua-end-delete
                // markua-start-insert
                $request->participants(),
                // markua-end-insert
                // markua-start-delete
                $request->input('data', [])
                // markua-end-delete
                // markua-start-insert
                $request->conversationData()
                // markua-end-insert
            );

        return response()->json([
            'data' => $conversation->toArray(),
        ]);
    }
}
```

Now let's update the `store` method in our controller to the following:

```
public function store(StoreConversation $request)
{
    $conversation = Chat::conversations()
        ->createConversation(
            // markua-start-delete
            $request->input('participants'),
            // markua-end-delete
            // markua-start-insert
            $request->participants(),
            // markua-end-insert
            // markua-start-delete
            $request->input('data', [])
            // markua-end-delete
            // markua-start-insert
            $request->conversationData()
            // markua-end-insert
        );

    return response()->json([
        'data' => $conversation->toArray(),
    ]);
}
```

Run the tests again and you should see green.

GET /conversations/<id>

Red, Green

Next, let's add an endpoint to return a conversation. Given a conversation id, we want to return a specific conversation.

For instance `GET /conversations/1`

Add the following test case:

```
public function testShow()
{
    $conversation = factory(Conversation::class)->create();

    $this->getJson("/conversations/{$conversation->id}")
        ->assertSuccessful()
        ->assertJsonStructure([
            'data',
        ]);
}
```

Running the tests we get

```
Symfony\Component\HttpKernel\Exception\MethodNotAllowedHttpException: The GET method\
 is not supported for this route
```

Add the missing route to our route group

```
Route::get('/conversations/{id}', 'ConversationController@show');
```

Add the controller method that accepts conversation id for showing the conversation underneath the `store()` method. We already know how to get a conversation by id. Essentially, we are just using our package in these routes.

```
public function show($id)
{
    $conversation = Chat::conversations()->getById($id);

    return response()->json([
        'data' => $conversation->toArray(),
    ]);
}
```

Run the tests and you should see green.

PUT /conversations/<id>

Our next expectation is to call an endpoint to update specific conversation details.

Go ahead and set up the test as follows:

```
public function testUpdate()
{
    $conversation = factory(Conversation::class)->create();

    $payload = ['data' => ['name' => 'New Title']];

    $this->putJson("/conversations/{$conversations->id}"), $payload)
        ->assertSuccessful()
        ->assertJson([
            'data' => $payload['data'],
        ]);
}
```

Run all tests to make sure we get red.

Add the route to update a conversation to our routes file

```
Route::put('/conversations/{id}', 'ConversationController@update');
```

Now let's add the implementation to update the conversation to our controller:

```
public function update(Request $request, $id)
{
    $conversation = Chat::conversations()->getById($id);
    $conversation->update($request->input('data'));

    return response()->json([
        'data' => $conversation->toArray(),
    ]);
}
```

Additionally, import the `Request` class as `use Illuminate\Http\Request;` at the top of your class.

It's worth noting that hardcoding routes in our tests could result in brittle tests. For instance, if we update the routes prefix, we would need to update all our tests; otherwise, they will fail. Instead, laravel has a handy helper to generate route links dynamically. All you need is to specify a unique name for each route.

Update your routes to the following:

```
/** Conversations */
Route::post('/conversations', 'ConversationController@store')
    ->name('conversations.store');
Route::get('/conversations/{id}', 'ConversationController@show')
    ->name('conversations.show');
Route::put('/conversations/{id}', 'ConversationController@update')
    ->name('conversations.update');
```

Now in our tests, we can make the following updates

Named routes in tests

```
<?php
    // ...
class ConversationControllerTest extends TestCase
{
    // ...
    public function testStore()
    {
        // ...
        $this->postJson(route('conversations.store'))
        // ...
    }

    public function testStoreWithParticipants()
    {
        // ...
        $this->postJson(route('conversations.store'), $payload)
        // ...
    }

    public function testShow()
    {
        // ...
        $this->getJson(
            route('conversations.show', ['id' => $conversation->getKey()])
        )
        // ...
    }

    public function testUpdate()
    {
        // ...
        $this->putJson(
            route('conversations.update', ['id' => $conversation->getKey()]),
            $payload
        )
        // ...
    }
}
```

Use named routes, any change to the location of the route, your tests will keep referencing the correct path.

Participation

```
$ touch tests/Feature/ConversationParticipantsControllerTest.php
```

POST /conversations/<id>/participants

Red, Green

We would like to add participants by making a post request like:

```
POST /conversations/1/participants
```

```php
<?php

namespace Tashtin\Chat\Tests\Feature;

use Chat;
use Tashtin\Chat\Models\Conversation;
use Tashtin\Chat\Tests\Helpers\Models\Client;
use Tashtin\Chat\Tests\Helpers\Models\User;
use Tashtin\Chat\Tests\TestCase;

class ConversationParticipantsControllerTest extends TestCase
{
    public function setUp(): void
    {
        parent::setUp();
        $this->withoutExceptionHandling();
    }

    public function testStore()
    {
        $conversation = factory(Conversation::class)->create();
        $payload = [
            'participants' => [
                [
                    'id' => $this->userAlpha->getKey(),
                    'type' => $this->userAlpha->getMorphClass()],
                [
                    'id' => $this->clientAlpha->getKey(),
                    'type' => $this->clientAlpha->getMorphClass()
                ],
            ],
        ];

        $this->postJson(
            route(
                'conversations.participation.store',
                [$conversation->getKey()]
            ),
            $payload
        )->assertStatus(200);

        $this->assertCount(2, $conversation->participants);
    }
}
```

Run the tests

```
There was 1 error:

1) Tashtin\Chat\Tests\Feature\ConversationParticipantsControllerTest::testStore
Symfony\Component\Routing\Exception\RouteNotFoundException: Route [conversations.par\
ticipation.store] not defined.
```

Add the missing route

```
/* Conversation Participation */
    Route::post(
        '/conversations/{id}/participants',
        'ConversationParticipantsController@store'
    )->name('conversations.participation.store');
```

Go ahead and set up the `ConversationParticipantsController`

```
 1  <?php
 2
 3  namespace Tashtin\Chat\Http\Controllers;
 4
 5  use Chat;
 6  use Illuminate\Http\Request;
 7
 8  class ConversationParticipantsController
 9  {
10      public function store(Request $request, $conversationId)
11      {
12          $participantModels = [];
13          $participants = $request->input('participants', []);
14
15          foreach ($participants as $participant) {
16              $participantModels[] = app($participant['type'])
17                  ->find($participant['id']);
18          }
19
20          $conversation = Chat::conversations()->getById($conversationId);
21          Chat::conversations()
22              ->setConversation($conversation)
23              ->addParticipants($participantModels);
24
25          return response($conversation->participants);
26      }
27  }
```

Lines **12** - **17** we have already seen when we created a conversation with participants earlier.

19
We get the conversation from the database using the passed conversation id.

21
Use our existing functionality to add participants to a conversation.

Running the tests we get green. This was enough to get our tests passing, so we can move on to refactoring.

Refactor

Let's start our refactoring process

First, let's create a form request to handle our input

```
$ touch src/Http/Requests/StoreParticipation.php
```

```php
<?php

namespace Tashtin\Chat\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class StoreParticipation extends FormRequest
{
    public function authorized()
    {
        return true;
    }

    public function rules()
    {
        return [
            'participants'        => 'required|array',
            'participants.*.id'   => 'required',
            'participants.*.type' => 'required|string',
        ];
    }

    public function participants()
    {
        $participantModels = [];
        $participants = $this->input('participants', []);

        foreach ($participants as $participant) {
            $participantModels[] = app($participant['type'])
                ->find($participant['id']);
        }

        return $participantModels;
    }
}
```

Now update

src/Http/Controllers/ConversationParticipantsController.php

```php
<?php

namespace Tashtin\Chat\Http\Controllers;

use Chat;
use Tashtin\Chat\Http\Requests\StoreParticipation;

class ConversationParticipantsController
{
    public function store(StoreParticipation $request, $conversationId)
    {
        $conversation = Chat::conversations()->getById($conversationId);
        Chat::conversations()
            ->setConversation($conversation)
            ->addParticipants($request->participants());

        return response($conversation->participants);
    }
}
```

Run the tests again and we should see green.

DELETE /conversations/<id>/participants

Red, Green

We would like to remove participants by making a post request like:

```
DELETE /conversations/1/participants
```

```php
public function testDestroy()
{
    $conversation = Chat::conversations()
        ->createConversation([$this->userAlpha, $this->clientAlpha]);

    $this->assertCount(2, $conversation->participants);

    $payload = [
        'participants' => [
            [
                'id' => $this->userAlpha->getKey(),
                'type' => $this->userAlpha->getMorphClass()
            ],
        ],
    ];

    $this->deleteJson(
        route(
            'conversations.participation.destroy',
            [$conversation->getKey()]
        ),
        $payload
    )->assertStatus(200);

    $this->assertCount(1, $conversation->fresh()->participants);
}
```

Add route

```
Route::delete(
    '/conversations/{id}/participants',
    'ConversationParticipantsController@destroy'
)->name('conversations.participation.destroy');
```

Run the tests

```
There was 1 error:

1) Tashtin\Chat\Tests\Feature\ConversationParticipantsControllerTest::testDestroy
Symfony\Component\Debug\Exception\FatalThrowableError: Call to undefined method Tash\
tin\Chat\Http\Controllers\ConversationParticipantsController::destroy()
```

Let’s add the method to the controller

```
public function destroy(Request $request, $conversationId)
{
    $participantModels = [];
    $participants = $request->input('participants', []);

    foreach ($participants as $participant) {
        $participantModels[] = app($participant['type'])
            ->find($participant['id']);
    }

    $conversation = Chat::conversations()->getById($conversationId);
    Chat::conversations()
        ->setConversation($conversation)
        ->removeParticipants($participantModels);

    return response($conversation->participants);
}
```

Refactor

We have the same logic for resolving participants from requests. At this step, we can afford to refactor that and reduce some duplication.
First, let's create a form request for removing participants.

```
$ touch src/Http/Requests/DeleteParticipation.php
```

src/Http/Requests/DeleteParticipation.php

```
<?php

namespace Tashtin\Chat\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class DeleteParticipation extends FormRequest
{
    public function authorized()
    {
        return true;
    }

    public function rules()
    {
        return [
            'participants'        => 'required|array',
            'participants.*.id'   => 'required',
            'participants.*.type' => 'required|string',
        ];
    }

    public function participants()
    {
        $participantModels = [];
        $participants = $this->input('participants', []);

        foreach ($participants as $participant) {
            $participantModels[] = app($participant['type'])
                ->find($participant['id']);
        }

        return $participantModels;
    }
}
```

Update the `destroy` method in `ConversationParticipantsController` to the following:

```
public function destroy(DeleteParticipation $request, $conversationId)
{
    $conversation = Chat::conversations()->getById($conversationId);
    Chat::conversations()
        ->setConversation($conversation)
        ->removeParticipants($request->participants());

    return response($conversation->participants);
}
```

Run the tests again and you should still see green.

Now let's get back to our form requests. We are going to work on removing duplication for the `participants()` method. Remember, as long as we are in the green state of our tests we can afford any refactoring. There are a few ways you can achieve re-using code across classes. You could create a base class to extend from or you can create a trait. I will opt for a trait.

```
$ touch src/Traits/ResolvesParticipants.php
```

src/Traits/ResolvesParticipants.php

```php
<?php

namespace Tashtin\Chat\Traits;

trait ResolvesParticipants
{
    public function participants(array $participants = [])
    {
        $participants = request()->input('participants', $participants);
        $participantModels = [];

        foreach ($participants as $participant) {
            $participantModels[] = app($participant['type'])
                ->find($participant['id']);
        }

        return $participantModels;
    }
}
```

Now we will update our form requests.

For instance, update `DeleteParticipation.php` form request to the following:

Use ResolvesParticipants

```php
<?php

namespace Tashtin\Chat\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
// markua-start-insert
use Tashtin\Chat\Traits\ResolvesParticipants;
// markua-end-insert

class DeleteParticipation extends FormRequest
{
    // markua-start-insert
    use ResolvesParticipants;
    // markua-end-insert

    public function authorized()
    {
        return true;
    }

    public function rules()
    {
        return [
            'participants'        => 'required|array',
            'participants.*.id'   => 'required',
            'participants.*.type' => 'required|string',
        ];
    }

    // markua-start-delete
    public function participants()
    {
        $participantModels = [];
        $participants = $this->input('participants', []);

        foreach ($participants as $participant) {
            $participantModels[] = app($participant['type'])->find($participant['id'\
]);
        }

        return $participantModels;
    }
    // markua-end-delete
}
```

Repeat the same for *StoreConversation.php* and *StoreParticipation.php* and run the tests again. You should see green.

GET /conversations/<id>/participants

We also need to create an endpoint to list participants in a conversation.

Red, Green

```
public function testIndex()
{
    $conversation = Chat::conversations()
        ->createConversation([$this->userAlpha, $this->clientAlpha]);

    $this->getJson(
        route(
            'conversations.participation.index',
            [$conversation->getKey()]
        )
    )->assertJsonCount(2);
}
```

Add the route

```
Route::get(
    '/conversations/{id}/participants',
    'ConversationParticipantsController@index'
)->name('conversations.participation.index');
```

Running the tests gives the following error as expected

```
1) Tashtin\Chat\Tests\Feature\ConversationParticipantsControllerTest::testIndex
Symfony\Component\Routing\Exception\RouteNotFoundException: Route [conversations.par\
ticipation.index] not defined.
```

Add the method to list participants to our controllers

```
public function index($conversationId)
{
    $conversation = Chat::conversations()->getById($conversationId);

    return response($conversation->participants);
}
```

Run the tests and you should see green.
You can see that it's really easy to get to a passing state when adding the routes.

Messaging

Next, we are going to add messaging-related routes.

```
$ touch tests/Feature/ConversationMessageControllerTest.php
```

POST /conversations/<id>/messages

Red, Green

tests/Feature/ConversationMessageControllerTest.php

```php
<?php

namespace Tashtin\Chat\Tests\Feature;

use Chat;
use Tashtin\Chat\Tests\TestCase;

class ConversationMessageControllerTest extends TestCase
{
    public function setUp(): void
    {
        parent::setUp();
        $this->withoutExceptionHandling();
    }

    public function testStore()
    {
        $conversation = Chat::conversations()
            ->createConversation([$this->userAlpha, $this->clientAlpha]);

        $payload = [
            'sender' => [
                'id' => $this->userAlpha->getKey(),
                'type' => $this->userAlpha->getMorphClass(),
            ],
            'message' => [
                'body' => 'Hello',
            ],
        ];

        $this->postJson(
            route('conversations.messages.store', $conversation->getKey()),
            $payload
        )->assertSuccessful();

        $this->assertDatabaseHas('messages', [
            'body' => 'Hello',
        ]);
    }
}
```

Run the tests

```
There was 1 error:

1) Tashtin\Chat\Tests\Feature\ConversationMessageControllerTest::testStore
Symfony\Component\Routing\Exception\RouteNotFoundException: Route [conversations.mes\
sages.store] not defined.
```

Add the route to store a sent message

```
/* Messaging */
Route::post(
    '/conversations/{id}/messages',
    'ConversationMessageController@store'
)->name('conversations.messages.store');
```

Running the tests

```
There was 1 error:

1) Tashtin\Chat\Tests\Feature\ConversationMessageControllerTest::testStore
Illuminate\Contracts\Container\BindingResolutionException: Target class [Tashtin\Cha\
t\Http\Controllers\ConversationMessageController] does not exist.
```

Now we add the `ConversationMessageController` to manage our conversation messages

```
$ touch src/Http/Controllers/ConversationMessageController.php
```

src/Http/Controllers/ConversationMessageController.php

```php
<?php

namespace Tashtin\Chat\Http\Controllers;

use Chat;
use Illuminate\Http\Request;

class ConversationMessageController
{
    public function store(Request $request, $conversationId)
    {
        // Resolve participant from type and id
        $participant = app($request->input('sender.type'))
            ->find($request->input('sender.id'));

        $conversation = Chat::conversations()->getById($conversationId);
        $message = Chat::messages()
            ->body($request->input('message.body'))
            ->from($participant)
            ->to($conversation)
            ->send();

        return response($message);
    }
}
```

Run the tests and you should see green

Refactor

Now we can refactor with confidence.

Let’s add a form request

```
$ touch src/Http/Requests/StoreMessage.php
```

src/Http/Requests/StoreMessage.php

```
 1 <?php
 2 
 3 namespace Tashtin\Chat\Http\Requests;
 4 
 5 use Illuminate\Foundation\Http\FormRequest;
 6 use Tashtin\Chat\Traits\ResolvesParticipants;
 7 
 8 class StoreMessage extends FormRequest
 9 {
10     use ResolvesParticipants;
11 
12     public function authorized()
13     {
14         return true;
15     }
16 
17     public function rules()
18     {
19         return [
20             'sender'   => 'required|array',
21             'sender.id'   => 'required',
22             'sender.type'   => 'required',
23             'message'          => 'required|array',
24             'message.body'     => 'required',
25         ];
26     }
27 
28     public function getMessageBody()
29     {
30         return $this->input('message.body');
31     }
32 
33     public function getSender()
34     {
35         $resolvedModels = $this->participants([$this->input('sender')]);
36         return $resolvedModels[0];
37     }
38 }
```

Line **10**

We are going to use the `ResolvesParticipants` trait that we added earlier to resolve our message sender from the request.

We also add some validation rules to make

Line **28**

Added a method to return the message body

Line **33**

We use the `participants()` method in our `ResolvesParticipants` trait. Remember, it also accepts an array of participants. In this case, we have one participant, the sender, so we pass that in an array and get the first entry for the resolved model(s) `$resolvedModels[0]`.

Let's update our controller to use the form request. Your class should look as follows:

src/Http/Controllers/ConversationMessageController.php

```php
<?php

namespace Tashtin\Chat\Http\Controllers;

use Chat;
use Tashtin\Chat\Http\Requests\StoreMessage;

class ConversationMessageController
{
    public function store(StoreMessage $request, $conversationId)
    {
        $conversation = Chat::conversations()
            ->getById($conversationId);
        $message = Chat::messages()
            ->body($request->getMessageBody())
            ->from($request->getSender())
            ->to($conversation)
            ->send();

        return response($message);
    }
}
```

Run the tests again and you should see green.

Next up is adding a route to list participant messages.

GET /conversations/<id>/messages

Red, Green

We would like to get participant messages by making a get request as follows:

```
GET /conversations/1/messages
    ?participant_id=1
    &participant_type=Tashtin\Chat\Tests\Helpers\Models\Client
```

Let’s add that route

```
Route::get('/conversations/{id}/messages','ConversationMessageController@index')
        ->name('conversations.messages.index');
```

Running the tests fails with no index method found in `ConversationMessageController`

Add the method `index` to the `ConversationMessageController`

```
    public function index(GetParticipantMessages $request, $conversationId)
    {
        // Resolve participant from parameters passed
        $participant = app($request->participant_type)
            ->find($request->participant_id);

        $conversation = Chat::conversations()->getById($conversationId);
        $messages = Chat::conversations($conversation)
            ->setParticipant($participant)
            ->getMessages();

        return response(['data' => $messages]);
    }
```

Running the tests we should see green.

Refactor

```
public function index(GetParticipantMessages $request, $conversationId)
{
    $conversation = Chat::conversations()->getById($conversationId);
    $messages = Chat::conversations($conversation)
        ->setParticipant($request->getParticipant())
        ->getMessages();

    return response(['data' => $messages]);
}
```

As you have seen previously, we have created a specific request class to handle our input when we request participant messages. We will use the request class to resolve our participant model. Hence we have added `$request->getParticipant()` to the `setParticipant` call. We have seen the rest of the code already when we set up methods to return participant messages.

Running the tests fails.

```
1) Tashtin\Chat\Tests\Feature\ConversationMessageControllerTest::testIndex
ReflectionException: Class Tashtin\Chat\Http\Controllers\GetParticipantMessages does\
 not exist
```

Let's create the class with the following:

```
<?php

namespace Tashtin\Chat\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Tashtin\Chat\Traits\ResolvesParticipants;

class GetParticipantMessages extends FormRequest
{
    use ResolvesParticipants;

    public function rules()
    {
        return [
            'participant_id'   => 'required',
            'participant_type'   => 'required',
        ];
    }

    public function getParticipant()
```

```php
    {
        $participantDetails = [
            'id' => $this->input('participant_id'),
            'type' => $this->input('participant_type'),
        ];

        $resolvedModels = $this->participants([$participantDetails]);

        return $resolvedModels[0];
    }
}
```

We set the validation rules to expect `participant_id` and `participant_type`. We also re-use our `ResolvesParticipants` trait. Remember, the trait has a method `participants` which accepts multiple participant details. So we pass in the details as an array even though we are only looking to resolve a single participant.

Rerun the tests, and you should see green.

In the next chapter, we will look at an essential aspect of package development: configuration.

Configuration

Configuration is an integral part of software packages. End users will look for ways to configure software packages before they write custom code or extensions. Thus it is crucial to provide configuration for Laravel packages you develop. Things that can be configured include database connections, table names, API credentials, middleware, etc.

For our chat package, we are going to have a configuration for the following items:

- Whether the package should broadcast events, which is helpful for users who want to integrate with tools like Pusher
- Whether we should expose the package routes in the application. Some users may choose not to use the routes that are baked in the package. So we should give them an option to turn that on and off.
- In addition, we would also like to have the path prefix for the routes configurable and add the ability to inject any middleware for those routes.
- Provide a way to configure default pagination parameters. When applications start having many messages, the user needs to perform pagination.
- We will also allow configurable data transformers (more on that in the next section).

Let's go ahead and set up our configuration file.

```
touch config/tashtin_chat.php
```

We will be adding our configuration values to this file. This file will be published to the user's laravel config path, so it's important to give it a name that's less likely to clash with other published config files.

Add the following snippet to the file.

```php
<?php

// Config values will be returned here as an array
return [];
```

```
touch tests/Feature/ConfigurationTest.php
```

Data Transformers

We implemented an API for our package in the previous section by adding routes. We queried the database and dumped everything out as API response. However, this is bad practice. It’s strongly encouraged to use transformers over just dumping the database data to the world.

What are transformers

A transformer gives you control over what data you output to your API consumers. With transformers, we are confident that sensitive information is not returned by mistake. Not only is it a security measure, but transformers will also help prevent API consumers' code from breaking when things like your database schema change.

While you can build your transformers easily, I recommend Fractal (http://fractal.thephpleague.com/) if you are in the PHP world. With transformers, you are afforded the benefit of serializing your data in a specific way, as well as embedding related resources within each other.

Fractal

I’m going to use a Fractal wrapper https://github.com/spatie/laravel-fractal here. As you can see, we don’t always have to reinvent the wheel. Instead, we can use other packages as dependencies in our packages.

Install the package as a dependency

```
composer require spatie/laravel-fractal
```

Transformers are more about having control. Now, if we force our package users to use our transformers, that wouldn’t be control, would it? So we will make sure adding transformers for use with the package is a configurable process.

However, we will provide a base class for custom transformers to extend. That way, our package consumers don’t have to do much work.

Let’s add the transformers base class.

```php
<?php

namespace Tashtin\Chat\Http;

use League\Fractal;
use League\Fractal\Manager;
use League\Fractal\Pagination\IlluminatePaginatorAdapter;
use League\Fractal\Resource\Collection;
use League\Fractal\Resource\Item;
use League\Fractal\Serializer\ArraySerializer;

abstract class Transformer extends Fractal\TransformerAbstract
{
    public function transformCollection($items, $paginator = null, $meta = null)
    {
        $resource = new Collection($items, $this);

        if ($paginator) {
            $resource->setPaginator(new IlluminatePaginatorAdapter($paginator));
        }

        if ($meta) {
            $resource->setMeta($meta);
        }

        return $this->fractalManager($resource);
    }

    public function fractalManager($resource)
    {
        $fractal = new Manager();

        $fractal->setSerializer(new ArraySerializer());

        if ($includes = request('include', null)) {
            $fractal->parseIncludes($includes);
        }

        return $fractal->createData($resource)->toArray();
    }

    public function transformItem($item)
    {
        $resource = new Item($item, $this);

        return $this->fractalManager($resource);
    }

    abstract public function transform($item);
}
```

To our test helpers, let’s add a directory for our test transformers.

```
mkdir tests/Helpers/Transformers
```

Conversation Transformer

GET /conversations/<id>

We are going to start with a simple conversation transformer.

```
touch tests/Helpers/Transformers/TestConversationTransformer.php
```

Add the following to the class.

```php
<?php

namespace Tashtin\Chat\Tests\Helpers\Transformers;

use Tashtin\Chat\Http\Transformer;

class TestConversationTransformer extends Transformer
{
    public function transform($conversation)
    {
        return [
            'id' => $conversation->id,
            'direct_message' => $conversation->direct_message,
            'is_private' => $conversation->is_private,
            'foo' => 'bar'
        ];
    }
}
```

Our transformer takes a given conversation and returns the fields specified in the array returned. We have also added the key-value `'foo' => 'bar'` to show that we can return anything from our transformer or do any computations.

If we add additional fields to our conversation table, they won't suddenly appear in our API response.

Red, Green

I'm not going to touch the tests we created already because these transformers are optional. So instead, let's start another test class for our transformers.

```
touch tests/Feature/DataTransformersTest.php
```

```
1   <?php
2
3   namespace Tashtin\Chat\Tests\Feature;
4
5   use Tashtin\Chat\Models\Conversation;
6   use Tashtin\Chat\Tests\Helpers\Transformers\TestConversationTransformer;
7   use Tashtin\Chat\Tests\TestCase;
8
9   class DataTransformersTest extends TestCase
10  {
11      public function testShowConversationWithTransformer()
12      {
13          $conversation = factory(Conversation::class)->create();
14          $this->app['config']->set(
15              'musonza_chat.transformers.conversation',
16              TestConversationTransformer::class
17          );
18
19          $this->getJson(route('conversations.show', $conversation->getKey()))
20              ->assertJson([
21                  'data' => [
22                      'id' => $conversation->id,
23                      'direct_message' => $conversation->direct_message,
24                      'is_private' => $conversation->is_private,
25                      'foo' => 'bar',
26                  ]
27              ]);
28      }
29  }
```

Line 14

We tell our configuration what transformer to use for our conversation.

Line 19

Make an API request to get conversation details and assert that the response matches what our data transformer provides.

Run the tests, and you will get a failure similar to the following:

```
--- Expected
+++ Actual
@@ @@
     'extra' => NULL,
     'created_at' => '2020-12-09 02:32:46',
     'updated_at' => '2020-12-09 02:32:46',
-    'foo' => 'bar',
   ),
 )
```

We aren't using our transformer in our `ConversationController` yet. Let's make the following update to the `show` method:

```
    public function show($id)
    {
        $conversation = Chat::conversations()->getById($id);

        // markua-start-insert
        $conversationTransformer = config('tashtin_chat.transformers.conversation');
        if ($conversationTransformer) {
            return fractal($conversation, $conversationTransformer)->respond();
        }
        // markua-end-insert

        return response()->json([
            'data' => $conversation->toArray(),
        ]);
    }
```

Rerun the tests, and you should see green.

It's worth noting that we didn't have to instantiate our transformer class. The helper function `fractal` was able to resolve that for us.

Refactor

We will undoubtedly use the transformer logic that we have written elsewhere. So let's go ahead and extract that to a reusable function.

I'm going to add the function to `Tashtin\Chat\Chat::class`

```
    public function getConversationTransformer()
    {
        return config('tashtin_chat.transformers.conversation');
    }
```

Make the following changes to `ConversationController`

```
    protected $conversationTransformer;

    public function __construct()
    {
        $this->conversationTransformer = Chat::getConversationTransformer();
    }
```

Your `show` method should now look as follows.

```
    public function show($id)
    {
        $conversation = Chat::conversations()->getById($id);

        if ($this->conversationTransformer) {
            return fractal($conversation, $this->conversationTransformer)->respond();
        }

        return response()->json([
            'data' => $conversation->toArray(),
        ]);
    }
```

Now let's move on to the `update` and `store` methods of our `ConversationController`. These methods return a conversation on update or creation. We will have the exact implementation as the `show` method.

POST /conversations

Red, Green

Add the test

```php
public function testStoreConversationWithTransformer()
{
    $this->app['config']->set(
        'tashtin_chat.transformers.conversation',
        TestConversationTransformer::class
    );

    $this->postJson(route('conversations.store', []))
        ->assertStatus(200)
        ->assertJson([
            'data' => [
                'id' => 1,
                'foo' => 'bar',
            ]
        ]);
}
```

Run the tests, and we get a failure because we are not using our transformer yet.

```
  ▯ ---·Expected
   ▯ +++·Actual
   ▯ @@ @@
   ▯      'id' => 1,
   ▯ -····'foo'·=>·'bar',
   ▯    ),
   ▯  )
```

Make the following method to the `ConversationController@store` method.

ConversationController@store

```php
public function store(StoreConversation $request)
{
    $conversation = Chat::conversations()
        ->createConversation(
            $request->participants(),
            $request->conversationData()
        );

    // markua-start-insert
    if ($this->conversationTransformer) {
        return fractal($conversation, $this->conversationTransformer)->respond();
    }
    // markua-end-insert

    return response()->json([
        'data' => $conversation->toArray(),
    ]);
}
```

Rerunning the tests, you should see green.

Refactor

We have an opportunity to do some minor refactoring here. We can avoid repeating ourselves in our code. Let's start with our test suite, `DataTransformersTest`. Rather than setting up our transformer configuration in each test, we can move that to a setup method of our test suite. So go ahead and add the setup method as follows:

DataTransformersTest@setUp

```
public function setUp(): void
{
    parent::setUp();
    $this->app['config']->set(
        'tashtin_chat.transformers.conversation',
        TestConversationTransformer::class
    );
}
```

Remove code snippets that set up the configuration for `TestConversationTransformer` in your test methods.

Run the tests, and you should still get green.

Also, the way we are making assertions for using a transformer is not effective. We want to test that our controllers make use of a given transformer. So let's make the following change to `tests/Feature/DataTransformersTest.php`.

DataTransformersTest@testShowConversationWithTransformer

```php
public function testShowConversationWithTransformer()
{
    /** @var TestConversationTransformer $conversationTransformer */
    $conversationTransformer = app(Chat::getConversationTransformer());
    $conversation = factory(Conversation::class)->create();
    $this->getJson(route('conversations.show', $conversation->getKey()))
        ->assertJson([
            'data' => $conversationTransformer->transform($conversation)
        ]);
}
```

Run the tests, and we should still see green. Instead of hardcoding the transformer item keys, we compare our API response to the entire transformation. While we are at it, make the `$conversationTransformer` a class variable to use in successive tests. Your `setUp` method and `testShowConversationWithTransformer` should now look as following.

```php
public function setUp(): void
{
    // ...
    $this->conversationTransformer = app(Chat::getConversationTransformer());
}

public function testShowConversationWithTransformer()
{

    $conversation = factory(Conversation::class)->create();
    $this->getJson(route('conversations.show', $conversation->getKey()))
        ->assertJson([
            'data' => $this->conversationTransformer->transform($conversation)
        ]);
}
```

Rerun the tests, and we should see green.

Another place to refactor is in our ConversationController. We have repeated the following code.

```php
if ($this->conversationTransformer) {
    return fractal($conversation, $this->conversationTransformer)->respond();
}
```

Instead, let's add a helper function to our controller when returning a single conversation response.

We will add a method called `itemResponse` to take a conversation and check if we have a transformer configured. Otherwise, it will return the conversation response as an array.

ConversationController@itemResponse

```php
private function itemResponse($conversation)
{
    if ($this->conversationTransformer) {
        return fractal($conversation, $this->conversationTransformer)->respond();
    }

    return response()->json([
        'data' => $conversation->toArray(),
    ]);
}
```

Your `store` and `show` methods should now look as follows:

```php
public function store(StoreConversation $request)
{
    $conversation = Chat::conversations()
        ->createConversation(
            $request->participants(),
            $request->conversationData()
        );

    return $this->itemResponse($conversation);
}

public function show($id)
{
    $conversation = Chat::conversations()->getById($id);

    return $this->itemResponse($conversation);
}
```

Run your tests, and you should still see green.

I'm not going to write a test for the `update` method, but you can update it to look as below:

ConversationController@update

```php
public function update(Request $request, $id)
{
    $conversation = Chat::conversations()->getById($id);
    $conversation->update($request->input('data'));
    return $this->itemResponse($conversation);
}
```

Message Transformer

In the previous section, we implemented a way to add transformers for a conversation. We need to do the same for messages in a conversation. So let's go ahead and set up the test and helpers.

```
touch tests/Helpers/Transformers/TestMessageTransformer.php
```

Add the following code to the class.

tests/Helpers/Transformers/TestMessageTransformer.php

```php
<?php

namespace Tashtin\Chat\Tests\Helpers\Transformers;

use Tashtin\Chat\Http\Transformer;

class TestMessageTransformer extends Transformer
{

    public function transform($message)
    {
        return [
            'id' => $message->id,
            'body' => $message->body,
            'conversation_id' => $message->conversation_id,
            'type' => $message->type,
            'read_at' => $message->read_at,
            'created_at' => $message->created_at,
            'direct_message' => $message->direct_message,
            'sender' => $message->sender,
            'foo' => 'bar'
        ];
    }
}
```

As with the conversation transformer, we have only specified the fields we want to be returned for our messages. I also added the key `foo` for the sake of testing.

POST /conversations/<id>/messages

Red, Green

Let’s set up a test for implementing a transformer when a message is created.

DataTransformersTest@testStoreMessageWithTransformer

```
   public function testStoreMessageWithTransformer()
    {
        $conversation = Chat::conversations()
            ->createConversation([$this->userAlpha, $this->clientAlpha]);

        $payload = [
            'sender' => [
                'id' => $this->userAlpha->getKey(),
                'type' => $this->userAlpha->getMorphClass(),
            ],
            'message' => [
                'body' => 'Hello',
            ],
        ];

        $this->postJson(
            route('conversations.messages.store', $conversation->getKey()),
            $payload
        )->assertJson([
            'data' => [
                'body' => 'Hello',
                'foo' => 'bar'
            ]
        ]);
    }
```

Run the tests, and you get a failure.

Add a method that returns message transformer from configuration just like we did with conversation transformer.

Chat@getMessageTransformer

```php
public function getMessageTransformer()
{
    return config('tashtin_chat.transformers.message');
}
```

Next, let's make sure the message transformer is set when we run our tests.

Add the following code to `DataTransformersTest@setUp`.

```php
$this->app['config']->set(
    'tashtin_chat.transformers.message',
    TestMessageTransformer::class
);
```

Finally, head to `Tashtin\Chat\Http\Controllers\ConversationMessageController` and add a class variable `$messageTransformer` as following:

```php
protected $messageTransformer;

public function __construct()
{
    $this->messageTransformer = Chat::getMessageTransformer();
}
```

Now your `ConversationMessageController@store` method should look as follows.

ConversationMessageController@store

```php
public function store(StoreMessage $request, $conversationId)
{
    $conversation = Chat::conversations()->getById($conversationId);
    $message = Chat::messages()
        ->body($request->getMessageBody())
        ->from($request->getSender())
        ->to($conversation)
        ->send();

    if ($this->messageTransformer) {
        return fractal($message, $this->messageTransformer)->respond();
    }

    return response(['data' => $message]);
}
```

Rerun the tests, and you should get green.

Refactor

Now that our tests are passing, we can refactor our code.

Let's add a helper function to check for a message transformer; we don't have code duplication.

ConversationMessageController@itemResponse

```php
private function itemResponse($message)
{
    if ($this->messageTransformer) {
        return fractal($message, $this->messageTransformer)->respond();
    }

    return response()->json([
        'data' => $message->toArray(),
    ]);
}
```

Now use the new method in `ConversationMessageController@store` as follows:

ConversationMessageController@store

```php
public function store(StoreMessage $request, $conversationId)
{
    $conversation = Chat::conversations()->getById($conversationId);
    $message = Chat::messages()
        ->body($request->getMessageBody())
        ->from($request->getSender())
        ->to($conversation)
        ->send();

    return $this->itemResponse($message);
}
```

Rerun the tests, and you should still get green.

GET /conversations/<id>/messages

Red, Green

Now we move on to a route that lists messages in a specified conversation. We would like to apply the message transformer to the response.

So let's go ahead and set up our test in `tests/Feature/DataTransformersTest.php`.

DataTransformersTest@testListMessagesWithTransformer

```php
public function testListMessagesWithTransformer()
{
    $conversation = Chat::conversations()
        ->createConversation([$this->userAlpha, $this->userBravo]);

        Chat::messages()
            ->body('Hello')
            ->from($this->userAlpha)
            ->to($conversation)
            ->send();
        Chat::messages()
            ->body('Hello friend')
            ->from($this->userBravo)
            ->to($conversation)
            ->send();


    $this->getJson(
        route(
            'conversations.messages.index',
            [
                $conversation->getKey(),
                'participant_id' => $this->userAlpha->getKey(),
                'participant_type' => $this->userAlpha->getMorphClass(),
            ]
        )
    )->assertJsonStructure([
        'data' => [
            [
                'id',
                'body',
                'conversation_id',
                'foo',
            ],
        ]
    ]);
}
```

We create a conversation between two participants, `$this->userAlpha` and `$this->userBravo`. Then we have our participants send a message to each other. We will then make an API call to

`/conversations/<id>/messages`, and of course, we have to send `participant_id` and `participant_-type` to resolve the participant context for the messages.

Then for our test assertion, we check that the message items returned in the response match our expected structure.

Run the tests, and you will get an error similar to

```
There was 1 failure:

1) Tashtin\Chat\Tests\Feature\DataTransformersTest::testListMessagesWithTransformer
Failed asserting that an array has the key 'foo'.
```

Head to `src/Http/Controllers/ConversationMessageController.php` and make the following updates to the `index` method.

```php
public function index(GetParticipantMessages $request, $conversationId)
{
    $conversation = Chat::conversations()->getById($conversationId);
    $messages = Chat::conversations($conversation)
        ->setParticipant($request->getParticipant())
        ->getMessages();

    // markua-start-insert
    if ($this->messageTransformer) {
        return fractal($messages, $this->messageTransformer)->toArray();
    }
    // markua-end-insert

    return response(['data' => $messages]);
}
```

Rerun the tests, and you should see green.

Continuous Integration

What is Continuous Integration?

Your package is almost ready for the public. You are concerned about how you will vet contributions from other software developers. You certainly don't want to pull each Pull Request and run the tests on your local machine. You also don't want to spend time on nit-picky comments related to code formatting. Fortunately, there are tools to address these issues and more.

We will solve these issues with continuous integration.

> Continuous integration (CI) is the practice of automating code changes from multiple contributors into a single software project. It's a primary DevOps best practice, allowing developers to frequently merge code changes into a central repository where builds and tests are run. Automated tools are used to assert the new code's correctness before integration. > - https://atlassian.com

Travis CI

There are several continuous integration tools that you can use. In our case, we will use **Travis CI** service, which you can find more information about at https://travis-ci.org/. We will be able to test our package that will be hosted on GitHub. If you don't have a GitHub account yet, head to https://github.com/ and create one for free. It's worth noting that Travis also offers support for Bitbucket, Gitlab, and many other source control providers.

Travis CI is a hosted, distributed continuous integration service used to build and test projects hosted at GitHub. Travis CI automatically detects when a commit has been made and pushed to a GitHub repository using Travis CI. Each time this happens, it will build the project and run tests. **Travis CI** will automatically detect when a commit is made and pushed to your package GitHub repository. Travis CI will run specified tasks like tests for you when that happens. You can even select multiple Laravel or PHP versions to run your tasks against.

Setup

In our case, we will be using Github, but Travis supports other source control providers, as I mentioned above. However, you can find more information in their documentation. For detailed instructions on using Travis with Github, you can refer to https://docs.travis-ci.com/user/tutorial/#to-get-started-with-travis-ci-using-github.

Otherwise, in our repository, we will add the file `.travis.yml` that tells Travis CI what to do.

```
touch .travis.yml
```

Add the following content to `.travis.yml`

```
language: php

matrix:
  include:
    - php: 7.3
      env: ILLUMINATE_VERSION=8.*
php:
  - 7.3

sudo: false

```

```
before_install:
  - composer require "illuminate/support:${ILLUMINATE_VERSION}" --no-update

install: composer update --prefer-source --no-interaction --dev

script: vendor/bin/phpunit --testdox
```

Let's go over the `.travis.yml` file and explain what each section does.

language
This one is self-explanatory. Our project is written in PHP so we specify that language.

matrix

A build matrix allows us to have several multiples that can run parallel. For example, when it comes to package development, the main benefit offered by a build matrix is the ability to run tests against different versions of runtimes or dependencies. For instance, we can specify multiple versions of PHP for our runtimes. We can also specify versions of Laravel that we want to test our package against.

Additionally, we have specified that we want to run PHPUnit tests by placing the command in the `script` section. We also want to ensure that we have all required dependencies (including dev dependencies) installed before running the tests. To do that, we have added `composer update --prefer-source --no-interaction --dev` to the `install` section.

Package distribution

You have finished writing your package. Now you want your code to be easily accessible to end-users. While users can download or clone your code from Github, you want to make it easy for them to install any future updates you make to your code. Need to make your code available to package managers that most frameworks use? In our case, we have Composer as a package manager, and the default package repository for Composer is Packagist.

Packagist

As we have mentioned, Packagist is the default Composer package repository
See https://packagist.org/about#:~:text=Packagist%20is%20the%20default%20Composer,packagist.org%20source%20
Users can easily search for code/packages using keywords. Composer will access the code from Packagist and install it on your system.