How to Use Laravel Broadcasting with Pusher and WebSockets

How to Use Laravel Broadcasting with Pusher and WebSockets

Real-time features like notifications, chats, and dashboards bring apps to life. Laravel Broadcasting makes it simple to broadcast server-side events to connected clients over WebSockets. In this guide, we'll configure broadcasting with Pusher, show how to use Laravel WebSockets as an alternative, create an event, and listen on the frontend with JavaScript.

What is Pusher?

Pusher is a hosted service that provides APIs and infrastructure for real-time communication over WebSockets. Instead of building and scaling your own WebSocket server, Pusher manages the connections, scaling, authentication, and delivery of messages between clients and your Laravel app. It's widely used for chat apps, notifications, live dashboards, and collaborative tools.

When you broadcast an event in Laravel with the `pusher` driver, the event payload is sent to the Pusher service. From there, Pusher instantly pushes the data to all subscribed clients over WebSockets. This makes it possible for hundreds or thousands of users to see updates live without refreshing the page.

Benefits of using Pusher:

- **Zero infrastructure overhead** – No need to maintain your own WebSocket servers.
- **Scalable** – Handles thousands of concurrent connections effortlessly.

- **Reliable delivery** – Provides guaranteed message delivery with retries.
- **Cross-platform support** – Works with web, iOS, and Android clients.
- **Simple integration** – Works seamlessly with Laravel Echo.

While Pusher is a paid service (with a free tier for small projects), it's ideal for apps that want a reliable, managed WebSocket solution without the hassle of scaling servers. If you want a self-hosted alternative, Laravel WebSockets is a great option.

Pusher vs Laravel WebSockets

Not sure whether to use Pusher or Laravel WebSockets? Here's a quick comparison:

Feature	Pusher	Laravel WebSockets
Hosting	Managed SaaS (cloud)	Self-hosted in your Laravel app
Setup	Very easy (just add API keys)	Requires package install + server setup
Scaling	Automatic scaling by Pusher	You scale your own server
Cost	Free tier, then paid monthly	Free (your server costs only)
Best For	Teams who want simplicity and reliability	Teams who want full control and avoid recurring costs

Configure Broadcasting in Laravel

First, install the required packages. If you're using **Pusher**, install the PHP SDK:

```
composer require pusher/pusher-php-serverCode language: Bash (bash)
```

Then, update your `.env` file with your Pusher credentials:

```
BROADCAST_DRIVER=pusher

PUSHER_APP_ID=your-app-id
PUSHER_APP_KEY=your-app-key
PUSHER_APP_SECRET=your-app-secret
PUSHER_APP_CLUSTER=mt1Code language: Bash (bash)
```

For local development without external services, you can use the Laravel WebSockets package as a Pusher-compatible replacement.

Create a Broadcastable Event

Next, generate an event that implements `ShouldBroadcast`. For example, a new chat message:

```
php artisan make:event MessageSentCode language: Bash (bash)
```

```
// app/Events/MessageSent.php
namespace App\Events;

use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;
use App\Models\Message;

class MessageSent implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $message;
```

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

    public function broadcastOn(): array
    {
        return [new PrivateChannel('chat.'.$this->message->chat_id)];
    }
}Code language: PHP (php)
```

This event broadcasts messages to a private chat channel, e.g. `chat.1`. You can scope channels by user, team, or resource ID.

Broadcasting from a Controller

After saving a message, dispatch the event in your controller:

```
// app/Http/Controllers/ChatController.php
namespace App\Http\Controllers;

use App\Models\Message;
use App\Events\MessageSent;
use Illuminate\Http\Request;

class ChatController extends Controller
{
    public function send(Request $request)
    {
        $message = Message::create([
            'chat_id' => $request->chat_id,
            'user_id' => auth()->id(),
            'body' => $request->body,
```

```
        ]);

        broadcast(new MessageSent($message))->toOthers();

        return response()->json(['status' => 'Message sent!']);
    }
}Code language: PHP (php)
```

`broadcast()` sends the event to all connected clients except the sender (`toOthers()`).

Listening on the Frontend

Laravel Echo is a JavaScript library that makes subscribing to channels simple. Install it via NPM:

```
npm install laravel-echo pusher-js --saveCode language: Bash (bash)
```

Configure Echo in your `resources/js/bootstrap.js` or `app.js` file:

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

window.Pusher = Pusher;

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: import.meta.env.VITE_PUSHER_APP_KEY,
    cluster: import.meta.env.VITE_PUSHER_APP_CLUSTER,
    forceTLS: true
});Code language: JavaScript (javascript)
```

Now listen for the event on the frontend:

```
window.Echo.private('chat.1')
    .listen('MessageSent', (e) => {
        console.log('New message:', e.message.body);
        const messages = document.getElementById('messages');
        messages.innerHTML += `<li>${e.message.body}</li>`;
    });Code language: JavaScript (javascript)
```

Whenever a new message is sent to `chat.1`, connected clients instantly receive it without refreshing.

UI Example: Real-Time Chat Box

Here's a minimal Blade view with a real-time chat interface:

```
<div id="chat">
  <ul id="messages"></ul>
  <form id="chat-form" method="POST" action="/chat/send">
    @csrf
    <input type="text" name="body" placeholder="Type a message...">
    <button type="submit">Send</button>
  </form>
</div>

<script>
document.getElementById('chat-form').addEventListener('submit', async
(e) => {
  e.preventDefault();
  const body = e.target.body.value;
  await fetch('/chat/send', {
    method: 'POST',
    headers: {'X-CSRF-TOKEN': '{{ csrf_token() }}','Content-
Type':'application/json'},
    body: JSON.stringify({ chat_id: 1, body })
```

```
  });
  e.target.reset();
});
</script>Code language: PHP (php)
```

This chat form submits messages, which are broadcasted via the event and instantly appended to the message list.

Wrapping Up

With Laravel Broadcasting, Pusher, or Laravel WebSockets, you can implement real-time features like chat, notifications, and dashboards. We explained what Pusher is, compared it with Laravel WebSockets, configured broadcasting, created a `MessageSent` event, listened with Laravel Echo, and built a simple chat UI. This foundation lets you add interactive, live features to your Laravel apps.

What's Next

Explore more real-time and event-driven features with these guides:

- Laravel Events and Listeners: A Complete Guide
- How to Use Laravel Observers for Model Events
- Laravel Horizon vs Queue Workers: Which One Should You Use?