Comprehensive Guide to Debugging with Laravel Telescope

Laravel Telescope is an elegant debug assistant that provides insight into requests, exceptions, jobs, queries, and more. Here's how to leverage it effectively in your Dockerized Laravel application.

1. Installation & Setup

Install Telescope in your Laravel app:

```
composer require laravel/telescope
```

Publish assets and run migrations:

```
php artisan telescope:install
php artisan migrate
```

Update your `.env`:

```
TELESCOPE_ENABLED=true
```

2. Docker Integration

Add Telescope service to your `docker-compose.yml`:

PROF

```
services:
  # ... existing services ...

  telescope:
    build:
      context: .
      dockerfile: docker/php/Dockerfile
    container_name: laravel_telescope
    command: php artisan telescope:watch
    volumes:
      - ./src:/var/www
    depends_on:
      - mysql
      - redis
```

3. Key Telescope Features Deep Dive

Request Monitoring

Telescope records all incoming requests with:

- Full request/response payloads
- Headers
- Session data
- User authentication status
- Timeline of application events

Example use case: Identify slow requests by sorting by duration.

Database Queries

Telescope shows:

- All executed SQL queries
- Bindings (click "Show bindings")
- Query duration
- Duplicate queries
- Slow queries (highlighted in red)

Pro tip: Click the "Explain" button to see the query execution plan.

Jobs & Queues

Monitor:

- Job payloads
- Status (pending, processed, failed)
- Execution time
- Exceptions
- Retry attempts

PROF

Example debugging:

```
// Tag important jobs for easier filtering
dispatch(new ProcessPodcast($podcast))->tags(['audio-processing']);
```

Scheduled Tasks

View:

- Command output
- Runtime duration
- Exit code

- Next run time

Mail Preview

Intercept and inspect:

- Raw email content
- Headers
- Attachments
- Recipients

Notifications

See:

- Notification channels used
- Recipients
- Notification content

Cache Operations

Monitor:

- Cache hits/misses
- Stored values
- Tags usage
- Forget operations

Redis Commands

Track all Redis operations with:

- Command type (GET, SET, etc.)
- Keys accessed
- Execution time

PROF

4. Advanced Configuration

Customize `config/telescope.php`:

```
return [
    'storage' => [
        'database' => [
            'connection' => env('DB_TELESCOPE_CONNECTION', 'mysql'),
            'chunk' => 1000,
        ],
    ],

    'watchers' => [
        Watchers\CacheWatcher::class => [
```

```
            'enabled' => env('TELESCOPE_CACHE_WATCHER', true),
            'hidden' => [
                'password',
                'token',
            ],
        ],
        // Other watchers...
    ],

    'ignore_paths' => [
        'nova-api*',
        'telescope*',
    ],
];
```

Filter Sensitive Data

```
Telescope::filter(function (IncomingEntry $entry) {
    if ($entry->type === 'request') {
        return !Str::is('*/admin/*', $entry->content['uri']);
    }

    return true;
});
```

5. Practical Debugging Workflows

Debugging Slow Requests

1. Open Telescope dashboard
2. Sort requests by duration
3. Click into slow request
4. Analyze:
    - Database queries tab
    - Timeline tab
    - Redis operations

Identifying N+1 Problems

1. Look for repeated similar queries
2. Check query counts in the "Queries" tab
3. Use the "Duplicate Queries" filter

Job Failure Analysis

1. Navigate to "Jobs" section
2. Filter by failed status

3. Examine:
    - Exception stack trace
    - Job payload
    - Timeline of attempts

Memory Leak Detection

1. Monitor memory usage in request details
2. Look for steadily increasing values
3. Correlate with specific queries/jobs

6. Telescope in Production

Prune old entries:

```
php artisan telescope:prune --hours=48
```

Set up pruning automatically:

```
// In app/Console/Kernel.php
protected function schedule(Schedule $schedule)
{
    $schedule->command('telescope:prune --hours=48')->daily();
}
```

Limit access with authorization:

```
// In AppServiceProvider
Telescope::auth(function ($request) {
    return in_array($request->user()->email, [
        'admin@example.com'
    ]);
});
```

PROF

7. Integrating with Other Tools

Log Aggregation

```
Telescope::tag(function (IncomingEntry $entry) {
    return $entry->type === 'request'
        ? ['status:'.$entry->content['response_status']]
        : [];
});
```

Slack Notifications for Errors

```
// In AppServiceProvider
Telescope::night();

// Or custom callback
Telescope::filterExceptions(function ($entry) {
    return $entry->isReportableException();
});
```

8. Custom Watchers

Create a custom watcher for your domain:

```
php artisan make:watcher DeploymentWatcher
```

```
namespace App\Watchers;

use Laravel\Telescope\Telescope;
use Laravel\Telescope\Watchers\Watcher;

class DeploymentWatcher extends Watcher
{
    public function register($app)
    {
        $app->events->listen('deployment.started', [$this,
'recordDeployment']);
    }

    public function recordDeployment($deployment)
    {
        Telescope::recordDeployment(DeploymentEntry::make($deployment));
    }
}
```

Register in `config/telescope.php`:

```
'watchers' => [
    App\Watchers\DeploymentWatcher::class,
    // ...
]
```

9. Performance Considerations

For high-traffic apps:

```
// Sample only a percentage of requests
Telescope::filter(function (IncomingEntry $entry) {
    return random_int(0, 100) <= 20; // Sample 20% of requests
});
```

Use separate database:

```
DB_TELESCOPE_CONNECTION=telescope
```

Configure in `config/database.php`:

```
'telescope' => [
    'driver' => 'mysql',
    'url' => env('DATABASE_URL'),
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => 'telescope',
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
    // ...
],
```

10. Security Best Practices

PROF

1. Always restrict access in production:

```
Telescope::auth(function ($request) {
    return app()->environment('local') ||
           $request->user()->isAdmin();
});
```

2. Filter sensitive data:

```
Telescope::hideRequestParameters([
    'password',
    'credit_card',
]);
```

3. Use HTTPS for Telescope routes in production

Example Debugging Session

Problem: API endpoint responding slowly

1. Open Telescope dashboard
2. Filter to the problematic route
3. Examine the request:
    - Note high memory usage (247MB)
    - See 42 database queries
4. Switch to Queries tab:
    - Identify duplicate queries loading user relationships
5. Solution:

```
// Eager load relationships
return User::with(['posts', 'comments'])->get();
```

Telescope provides unparalleled visibility into your Laravel application's inner workings. By leveraging its comprehensive monitoring capabilities, you can quickly identify and resolve performance bottlenecks, debug errors, and understand your application's behavior in depth.