Text Generation

Prism provides a powerful interface for generating text using Large Language Models (LLMs). This guide covers everything from basic usage to advanced features like multi-modal interactions and response handling.

Basic Text Generation

At its simplest, you can generate text with just a few lines of code:

```php
use Prism\Prism\Prism;
use Prism\Prism\Enums\Provider;

$response = Prism::text()
    ->using(Provider::Anthropic, 'claude-3-5-sonnet-20241022')
    ->withPrompt('Tell me a short story about a brave knight.')
    ->asText();

echo $response->text;
```

System Prompts and Context

System prompts help set the behavior and context for the AI. They're particularly useful for maintaining consistent responses or giving the LLM a persona:

```php
use Prism\Prism\Prism;
use Prism\Prism\Enums\Provider;

$response = Prism::text()
    ->using(Provider::Anthropic, 'claude-3-5-sonnet-20241022')
```

```
    ->withSystemPrompt('You are an expert mathematician who explains concepts s
    ->withPrompt('Explain the Pythagorean theorem.')
    ->asText();
```

You can also use Laravel views for complex system prompts:

```php
use Prism\Prism\Prism;
use Prism\Prism\Enums\Provider;

$response = Prism::text()
    ->using(Provider::Anthropic, 'claude-3-5-sonnet-20241022')
    ->withSystemPrompt(view('prompts.math-tutor'))
    ->withPrompt('What is calculus?')
    ->asText();
```

You an also pass a View to the `withPrompt` method.

Message Chains and Conversations

For interactive conversations, use message chains to maintain context:

```php
use Prism\Prism\Prism;
use Prism\Prism\Enums\Provider;
use Prism\Prism\ValueObjects\Messages\UserMessage;
use Prism\Prism\ValueObjects\Messages\AssistantMessage;

$response = Prism::text()
    ->using(Provider::Anthropic, 'claude-3-5-sonnet-20241022')
    ->withMessages([
        new UserMessage('What is JSON?'),
        new AssistantMessage('JSON is a lightweight data format...'),
        new UserMessage('Can you show me an example?')
    ])
    ->asText();
```

Message Types

- `SystemMessage`

- `UserMessage`
- `AssistantMessage`
- `ToolResultMessage`

> **NOTE**
>
> Some providers, like Anthropic, do not support the `SystemMessage` type. In those cases we convert `SystemMessage` to `UserMessage`.

Generation Parameters

Fine-tune your generations with various parameters:

`withMaxTokens`

Maximum number of tokens to generate.

`usingTemperature`

Temperature setting.

The value is passed through to the provider. The range depends on the provider and model. For most providers, 0 means almost deterministic results, and higher values mean more randomness.

> **TIP**
>
> It is recommended to set either temperature or topP, but not both.

`usingTopP`

Nucleus sampling.

The value is passed through to the provider. The range depends on the provider and model. For most providers, nucleus sampling is a number between 0 and 1. E.g. 0.1 would mean that only tokens with the top 10% probability mass are considered.

> **TIP**
>
> It is recommended to set either temperature or topP, but not both.

`withClientOptions`

Under the hood we use Laravel's HTTP client. You can use this method to pass any of Guzzles request options e.g. `->withClientOptions(['timeout' => 30])` .

`withClientRetry`

Under the hood we use Laravel's HTTP client. You can use this method to set retries e.g. `->withClientRetry(3, 100)` .

`usingProviderConfig`

This allows for complete or partial override of the providers configuration. This is great for multi-tenant applications where users supply their own API keys. These values are merged with the original configuration allowing for partial or complete config override.

Response Handling

The response object provides rich access to the generation results:

```php
use Prism\Prism\Prism;
use Prism\Prism\Enums\Provider;

$response = Prism::text()
    ->using(Provider::Anthropic, 'claude-3-5-sonnet-20241022')
    ->withPrompt('Explain quantum computing.')
    ->asText();

// Access the generated text
echo $response->text;

// Check why the generation stopped
echo $response->finishReason->name;

// Get token usage statistics
echo "Prompt tokens: {$response->usage->promptTokens}";
echo "Completion tokens: {$response->usage->completionTokens}";

// For multi-step generations, examine each step
foreach ($response->steps as $step) {
    echo "Step text: {$step->text}";
```

```php
    echo "Step tokens: {$step->usage->completionTokens}";
}

// Access message history
foreach ($response->responseMessages as $message) {
    if ($message instanceof AssistantMessage) {
        echo $message->content;
    }
}
```

Finish Reasons

```php
FinishReason::Stop;
FinishReason::Length;
FinishReason::ContentFilter;
FinishReason::ToolCalls;
FinishReason::Error;
FinishReason::Other;
FinishReason::Unknown;
```

Error Handling

Remember to handle potential errors in your generations:

```php
use Prism\Prism\Prism;
use Prism\Prism\Enums\Provider;
use Prism\Prism\Exceptions\PrismException;
use Throwable;

try {
    $response = Prism::text()
        ->using(Provider::Anthropic, 'claude-3-5-sonnet-20241022')
        ->withPrompt('Generate text...')
        ->asText();
} catch (PrismException $e) {
    Log::error('Text generation failed:', ['error' => $e->getMessage()]);
} catch (Throwable $e) {
    Log::error('Generic error:', ['error' => $e->getMessage()]);
```

```
}
```

Previous page
Configuration

Next page
Streaming Ouput