🎨 Create a Custom Symfony Flex Recipe in 30 Minutes

laurentmn Follow 8 min read · Oct 25, 2025

Symfony Flex changed how we install bundles.
Instead of manually editing configuration files, Flex automates the setup.
But what if you want to create your own recipe?
In this article I will show you how to build one from scratch.

Not a Medium member yet? Click here to access this article for free!

Illustrations generated with Adobe Firefly AI.

. . .

🎯 What Is a Flex Recipe?

A Flex recipe is a set of instructions that tells Symfony how to configure a package automatically. When you install a bundle, Flex reads the recipe and applies changes to your project. These changes include:

- Creating configuration files
- Updating environment variables
- Registering routes
- Adding service definitions

Think of it as a blueprint. You define what happens during installation, and Flex executes it.

🔧 Prerequisites

You need a working Symfony project with Flex installed. This example uses Symfony 6.4, but the concepts apply to newer versions.

Create a test project:

```
symfony new recipe-demo --version=6.4
cd recipe-demo
```

📝 Recipe Structure

Recipes live in a `manifest.json` file. This file contains all instructions for Flex. The basic structure looks like this:

```
{
    "bundles": {},
    "copy-from-recipe": {},
    "env": {},
    "gitignore": [],
    "composer-scripts": {}
}
```

Each key serves a specific purpose. Let's explore them one by one.

🏗️ Building a Notification Bundle Recipe

We'll create a recipe for a fictional notification bundle. This bundle sends alerts via email, SMS, or Slack. The recipe will configure everything automatically.

Step 1: Create the Bundle Package

First, create a new directory for your bundle:


```
mkdir -p custom-packages/notification-bundle
cd custom-packages/notification-bundle
```

Create a `composer.json` file:

```
{
    "name": "acme/notification-bundle",
    "type": "symfony-bundle",
    "description": "Multi-channel notification system",
    "require": {
        "php": ">=8.1",
        "symfony/framework-bundle": "^6.4"
    },
    "autoload": {
        "psr-4": {
            "Acme\\NotificationBundle\\": "src/"
        }
    }
}
```

Create the bundle class at `src/AcmeNotificationBundle.php`:

```
<?php

namespace Acme\NotificationBundle;
use Symfony\Component\HttpKernel\Bundle\Bundle;
class AcmeNotificationBundle extends Bundle
{
}
```

Step 2: Create the Flex Recipe

Create a `recipe` directory in your bundle:

```
mkdir recipe
```

Create `recipe/manifest.json`:

```
{
    "bundles": {
        "Acme\\NotificationBundle\\AcmeNotificationBundle": ["all"]
    },
    "copy-from-recipe": {
        "config/": "%CONFIG_DIR%/"
    },
```

```
    "env": {
        "NOTIFICATION_EMAIL_FROM": "noreply@example.com",
        "NOTIFICATION_SLACK_WEBHOOK": "",
        "NOTIFICATION_SMS_API_KEY": ""
    },
    "gitignore": [
        ".notification-cache/"
    ]
}
```

Let's break down each section:

bundles: Registers the bundle in `config/bundles.php`. The `["all"]` means it loads in all environments.

copy-from-recipe: Copies files from the recipe to your project. The `%CONFIG_DIR%` placeholder points to your `config/` directory.

env: Adds variables to `.env` and `.env.local.php` files. These define default values.

gitignore: Adds entries to `.gitignore`. The bundle creates a cache directory we don't want in version control.

Step 3: Add Configuration Files

Create `recipe/config/packages/acme_notification.yaml`:

```yaml
acme_notification:
    channels:
        email:
            enabled: true
            from: '%env(NOTIFICATION_EMAIL_FROM)%'
            transport: 'smtp'

        slack:
            enabled: false
            webhook_url: '%env(NOTIFICATION_SLACK_WEBHOOK)%'

        sms:
            enabled: false
            api_key: '%env(NOTIFICATION_SMS_API_KEY)%'
            provider: 'twilio'

    default_channel: 'email'
    retry_failed: true
    max_retries: 3
```

This configuration file defines how the bundle works. Users can modify these values after installation.

Step 4: Add Route Configuration

Create `recipe/config/routes/acme_notification.yaml`:

```
acme_notification_webhook:
    path: /notifications/webhook/{channel}
    controller: Acme\NotificationBundle\Controller\WebhookController::handle
    methods: [POST]

acme_notification_status:
    path: /notifications/status
    controller: Acme\NotificationBundle\Controller\StatusController::index
    methods: [GET]
```

These routes handle incoming webhooks and status checks.

Step 5: Create a Post-Install Message

Create `recipe/post-install.txt`:

```
<bg=blue;fg=white>                                          </>
<bg=blue;fg=white>  Notification Bundle Installed!          </>
<bg=blue;fg=white>                                          </>

Next steps:
  1. Configure your notification channels in config/packages/acme_notification.y
  2. Set environment variables in .env.local:
     - NOTIFICATION_EMAIL_FROM
     - NOTIFICATION_SLACK_WEBHOOK (if using Slack)
     - NOTIFICATION_SMS_API_KEY (if using SMS)
  3. Use the NotificationService in your code:
     $notificationService->send('Hello World', 'email');
Read the documentation at https://github.com/acme/notification-bundle
```

This message appears after installation. It guides users through setup.

🚀 Testing Your Recipe Locally

Symfony Flex looks for recipes in the official repository by default. To test locally, we need a different approach.

Create a Local Repository

Create a `recipes` directory in your project root:

```
cd ../..
mkdir -p recipes/acme/notification-bundle/1.0
```

Copy your recipe there:

```
cp -r custom-packages/notification-bundle/recipe/* recipes/acme/notification-bun
```


nfigure Composer

Edit your project's `composer.json` to add a path repository:

```
{
    "repositories": [
        {
            "type": "path",
            "url": "custom-packages/*"
        }
    ]
}
```

Configure Flex

Create or edit `symfony.lock`:

```
{
    "acme/notification-bundle": {
        "version": "1.0",
        "recipe": {
            "repo": "github.com/symfony/recipes-contrib",
            "branch": "main",
            "version": "1.0"
        }
    }
}
```

For local testing, set the `SYMFONY_ENDPOINT` environment variable:

```
export SYMFONY_ENDPOINT="file://$(pwd)/recipes"
```

This tells Flex to check your local recipes directory.

Install the Bundle

```
composer require acme/notification-bundle:@dev
```

Flex will execute your recipe. You should see:

- A new file at `config/packages/acme_notification.yaml`
- New routes at `config/routes/acme_notification.yaml`
- Environment variables in `.env`
- An entry in `.gitignore`
- Your post-install message

🔄 Advanced Recipe Features

Let's add more functionality to our recipe.

Adding Composer Scripts

Recipes can register Composer scripts. Edit `manifest.json`:

```
{
    "composer-scripts": {
        "cache:clear": "symfony-cmd",
        "notification:test": "script"
    }
}
```

Create a test script at `recipe/bin/test-notification.php`:

```
#!/usr/bin/env php
<?php
require dirname(__DIR__).'/vendor/autoload.php';
echo "Testing notification channels...\n";
echo "Email: Ready\n";
echo "Slack: Not configured\n";
echo "SMS: Not configured\n";
```

Make it executable:

```
chmod +x recipe/bin/test-notification.php
```

Creating Configuration Classes

Add a Configuration class for better type safety. Create `src/DependencyInjection/Configuration.php`:

```
<?php

namespace Acme\NotificationBundle\DependencyInjection;
use Symfony\Component\Config\Definition\Builder\TreeBuilder;
```

```
use Symfony\Component\Config\Definition\ConfigurationInterface;
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder(): TreeBuilder
    {
        $treeBuilder = new TreeBuilder('acme_notification');
        $rootNode = $treeBuilder->getRootNode();
        $rootNode
            ->children()
                ->arrayNode('channels')
                    ->children()
                        ->arrayNode('email')
                            ->children()
                                ->booleanNode('enabled')->defaultTrue()->end()
                                ->scalarNode('from')->isRequired()->end()
                                ->scalarNode('transport')->defaultValue('smtp')-
                            ->end()
                        ->end()
                        ->arrayNode('slack')
                            ->children()
                                ->booleanNode('enabled')->defaultFalse()->end()
                                ->scalarNode('webhook_url')->defaultNull()->end(
                            ->end()
                        ->end()
                        ->arrayNode('sms')
                            ->children()
                                ->booleanNode('enabled')->defaultFalse()->end()
                                ->scalarNode('api_key')->defaultNull()->end()
                                ->scalarNode('provider')->defaultValue('twilio')
                            ->end()
                        ->end()
                    ->end()
                ->end()
                ->scalarNode('default_channel')->defaultValue('email')->end()
                ->booleanNode('retry_failed')->defaultTrue()->end()
                ->integerNode('max_retries')->defaultValue(3)->end()
            ->end()
        ;
        return $treeBuilder;
    }
}
```

Create the Extension class at `src/DependencyInjection/AcmeNotificationExtension.php`:

```
<?php

namespace Acme\NotificationBundle\DependencyInjection;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Extension\Extension;
use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;
class AcmeNotificationExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container): void
    {
        $configuration = new Configuration();
        $config = $this->processConfiguration($configuration, $configs);
        $loader = new YamlFileLoader(
            $container,
            new FileLocator(__DIR__.'/../../config')
        );
        $loader->load('services.yaml');
        $container->setParameter('acme_notification.config', $config);
    }
}
```

Adding Services

Create `config/services.yaml` in your bundle:

```yaml
services:
    _defaults:
        autowire: true
        autoconfigure: true

    Acme\NotificationBundle\Service\NotificationService:
        arguments:
            $config: '%acme_notification.config%'
    Acme\NotificationBundle\Channel\EmailChannel:
        tags: ['acme.notification.channel']
    Acme\NotificationBundle\Channel\SlackChannel:
        tags: ['acme.notification.channel']
    Acme\NotificationBundle\Channel\SmsChannel:
        tags: ['acme.notification.channel']
```

Create the main service at `src/Service/NotificationService.php`:

```php
<?php

namespace Acme\NotificationBundle\Service;
use Acme\NotificationBundle\Channel\ChannelInterface;
class NotificationService
{
    private array $channels = [];
    private array $config;
    public function __construct(array $config)
    {
        $this->config = $config;
    }
    public function addChannel(ChannelInterface $channel): void
    {
        $this->channels[$channel->getName()] = $channel;
    }
    public function send(string $message, string $channelName = null): bool
    {
        $channelName = $channelName ?? $this->config['default_channel'];
        if (!isset($this->channels[$channelName])) {
            throw new \InvalidArgumentException("Channel {$channelName} not foun
        }
        $channel = $this->channels[$channelName];
        $config = $this->config['channels'][$channelName] ?? [];
        if (!($config['enabled'] ?? false)) {
            return false;
        }
        return $channel->send($message, $config);
    }
}
```

🎭 Real-World Use Cases

Custom Flex recipes solve specific problems. Here are common scenarios.

Use Case 1: Internal Company Bundles

Your company has a shared authentication bundle. Every new project needs the same configuration. A recipe automates this:

Copies security configuration

Sets up firewall rules

- Adds environment variables for OAuth
- Configures session handling

Developers save 30 minutes per project setup.

Use Case 2: Microservice Communication

You build multiple microservices. Each needs to connect to a message broker. The recipe:

- Installs the Messenger component
- Configures RabbitMQ transport
- Creates default queue bindings
- Adds health check endpoints

Teams avoid configuration drift across services.

Use Case 3: Development Tools

You create a debugging bundle with custom profiler panels. The recipe:

- Registers the bundle only in dev/test environments
- Adds toolbar integration
- Creates log directories
- Sets up demo routes

New team members get debugging tools immediately.

🎨 Environment-Specific Configuration

Recipes can apply different settings per environment. Update your manifest:

```
{
    "bundles": {
        "Acme\\NotificationBundle\\AcmeNotificationBundle": ["all"]
    },
    "copy-from-recipe": {
        "config/": "%CONFIG_DIR%/",
        "config/packages/dev/": "%CONFIG_DIR%/packages/dev/",
        "config/packages/prod/": "%CONFIG_DIR%/packages/prod/"
```

```
    }
}
```


ate `recipe/config/packages/dev/acme_notification.yaml`:

```
acme_notification:
    channels:
        email:
            transport: 'null'
    default_channel: 'email'
    retry_failed: false
```

Create `recipe/config/packages/prod/acme_notification.yaml`:

```
acme_notification:
    channels:
        email:
            transport: 'smtp'
        slack:
            enabled: true
        sms:
            enabled: true
    retry_failed: true
    max_retries: 5
```

Development uses a null transport. Production enables all channels.

🔐 Managing Secrets in Recipes

Never put real credentials in recipes. Use environment variables:

```
{
    "env": {
        "#1": "API keys for notification channels",
        "NOTIFICATION_SLACK_WEBHOOK": "",
        "NOTIFICATION_SMS_API_KEY": "",
        "#2": "For production, set these in .env.local or Symfony secrets"
    }
}
```

Comments in the env section guide users. They start with `#` followed by a number.

Create a `recipe/config/secrets.txt`:

```
To use encrypted secrets in production:
```

```
1. Generate the keys:
   php bin/console secrets:generate-keys
2. Set your values:
   php bin/console secrets:set NOTIFICATION_SLACK_WEBHOOK
   php bin/console secrets:set NOTIFICATION_SMS_API_KEY
3. Commit config/secrets/prod/ to version control
4. Keep config/secrets/prod/prod.decrypt.private.php secret
```

🧪 Testing Recipe Installation

Write tests to verify your recipe works. Create `tests/RecipeTest.php`:

```php
<?php

namespace Acme\NotificationBundle\Tests;
use PHPUnit\Framework\TestCase;
use Symfony\Component\Filesystem\Filesystem;
use Symfony\Component\Yaml\Yaml;
class RecipeTest extends TestCase
{
    private Filesystem $filesystem;
    private string $tempDir;
    protected function setUp(): void
    {
        $this->filesystem = new Filesystem();
        $this->tempDir = sys_get_temp_dir().'/recipe-test-'.uniqid();
        $this->filesystem->mkdir($this->tempDir);
    }
    protected function tearDown(): void
    {
        $this->filesystem->remove($this->tempDir);
    }
    public function testManifestIsValid(): void
    {
        $manifestPath = __DIR__.'/../recipe/manifest.json';
        $this->assertFileExists($manifestPath);
        $manifest = json_decode(file_get_contents($manifestPath), true);
        $this->assertIsArray($manifest);
        $this->assertArrayHasKey('bundles', $manifest);
        $this->assertArrayHasKey('copy-from-recipe', $manifest);
    }
    public function testConfigurationFileExists(): void
    {
        $configPath = __DIR__.'/../recipe/config/packages/acme_notification.yaml
        $this->assertFileExists($configPath);
        $config = Yaml::parseFile($configPath);
        $this->assertArrayHasKey('acme_notification', $config);
        $this->assertArrayHasKey('channels', $config['acme_notification']);
    }
    public function testEnvironmentVariablesAreDefined(): void
    {
        $manifestPath = __DIR__.'/../recipe/manifest.json';
        $manifest = json_decode(file_get_contents($manifestPath), true);
        $this->assertArrayHasKey('env', $manifest);
        $this->assertArrayHasKey('NOTIFICATION_EMAIL_FROM', $manifest['env']);
    }
}
```

Run the tests:

```
./vendor/bin/phpunit tests/RecipeTest.php
```

📤 Publishing Your Recipe

Once tested, publish your recipe to the Symfony recipes repository.

Public Packages

Submit a pull request to `symfony/recipes` or `symfony/recipes-contrib`:

1. Fork the repository
2. Create a directory: `vendor-name/package-name/version`
3. Add your `manifest.json` and files
4. Submit the PR

The Symfony team reviews all contributions. They check:

- Valid JSON syntax
- Correct file paths
- No sensitive data
- Clear documentation

For Private Packages

Host recipes on your own server:

```
recipes/
  acme/
    notification-bundle/
      1.0/
        manifest.json
        config/
      1.1/
        manifest.json
        config/
```

Set the endpoint in your projects:

```
export SYMFONY_ENDPOINT="https://recipes.yourcompany.com"
```

Point Composer to your repository:

```
{
    "repositories": [
        {
            "type": "composer",
```

```
            "url": "https://packages.yourcompany.com"
        }
    ]
}
```


Debugging Recipe Problems

Recipes can fail. Here's how to troubleshoot.

Enable Flex Logging

Set verbose output:

```
composer require acme/notification-bundle -vvv
```

This shows each step Flex takes.

Check the Lock File

Open `symfony.lock` and find your package:

```
{
    "acme/notification-bundle": {
        "version": "1.0",
        "recipe": {
            "repo": "github.com/symfony/recipes-contrib",
            "branch": "main",
            "version": "1.0"
        }
    }
}
```

The `recipe` key shows which recipe Flex used.

Validate JSON Syntax

Bad JSON breaks recipes. Validate your manifest:

```
php -r "json_decode(file_get_contents('recipe/manifest.json'));"
```

No output means valid JSON. Errors appear for problems.

Test File Copies

Check if files copied correctly:

```
ls -la config/packages/acme_notification.yaml
ls -la config/routes/acme_notification.yaml
```


sing files mean wrong paths in `copy-from-recipe`.

🎓 Best Practices

Follow these rules for better recipes.

Keep recipes simple: One recipe per bundle. Don't try to configure the entire application.

Use sensible defaults: Set environment variables to working values. Users can change them later.

Document everything: Add comments in configuration files. Explain what each setting does.

Version carefully: Update the recipe version when you change the manifest. This prevents conflicts.

Test in clean projects: Install your bundle in a fresh Symfony project. Check that everything works.

Respect user configuration: Don't overwrite existing files. Let users customize their setup.

Minimize dependencies: Keep recipes focused on your bundle. Don't configure third-party services.

🏁 Summary

Custom Symfony Flex recipes automate bundle installation. You define the configuration once, and every project gets it automatically.

The key components are:

- `manifest.json` with installation instructions
- Configuration files in the recipe directory
- Environment variables for secrets
- Post-install messages for guidance

Recipes save time and reduce errors. Teams stay consistent across projects. New developers get running faster.

Start with a simple recipe. Add features as you learn. Test thoroughly before publishing. Your future self will thank you.

notification bundle example showed a complete workflow. You can pt it to any bundle. The patterns stay the same.

Now build your own recipe. Automate your setup. Make your tools easier to use.

. . .

Found this article useful?
Show your support by clapping 👏, subscribing 🔔, sharing to social networks ⓕ 𝕏

💬 Which concrete subject do you want me to expose next time?
Share your suggestions or questions in the comments!

🔗 Follow me here on Medium and join my mailing list for more in-depth content about Symfony!

Symfony | Web Development | Software Development | Technology | Programming

Written by laurentmn

467 followers · 119 following

Follow

IT PHP/Symfony Expert, Daddy of 4 children, technology enthusiast, keen reader of books on personal development, entrepreneurship, productivity, neuro sciences.

Responses (2)

Athosch

What are your thoughts?

Tac Tacelosky
1 day ago

…rticle, thanks. You can consolidate a few steps by extending AbstractBundle and putting some …guration in the Bundle class itself.

Reply

Sriram Yaladandi he/him
Oct 25

Laurent, I genuinely smiled at how your bundle's post-install message actually gives "next steps" in plain words not just raw instructions. Makes setup feel so much friendlier than the usual tech docs!

1 reply Reply

More from laurentmn

laurentmn

🚀How Domain-Driven Design Change My Mindset About…

Domain-Driven Design (DDD) is an architecture/philophy to organize you…

Oct 29 99 5

laurentmn

⚡ What's New in Symfony 7.4/8.0 (And Why You Should Care)

Symfony 7.4/8.0 brings UUID v7, RFC-compliant caching, and voter metadata. See…

Oct 23 49 2

laurentmn

🎨 Build Better DTOs: Symfony 7.1's New Validation Superpower

Learn to use UniqueEntity on DTOs, commands, and forms in Symfony 7.1….

Oct 16 16 1

laurentmn

💎 Advanced Symfony Security: 12 Secret Management Techniques…

Production-ready secret management for Symfony 7.4: rotation without downtime,…

Nov 5 117 1

See all from laurentmn

All your favorite parts of Medium are now in one sidebar for easy access.

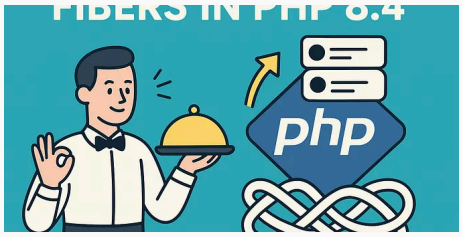
Recommended from Medium

laurentmn

💎 Symfony 7.4's Hidden Gems: Share Directories, Video...

Learn how Symfony 7.4's new share directory, video constraint, and caching HTTP client...

Nov 8 18 1

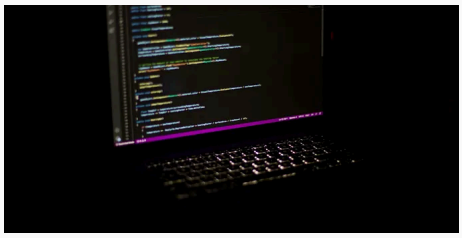


In Level Up Coding by Mathews Jose

🚀 Fibers in PHP 8.4: The Future of Asynchronous PHP

PHP has always been known as a simple, reliable web scripting language. But as the...

Sep 16 97 2

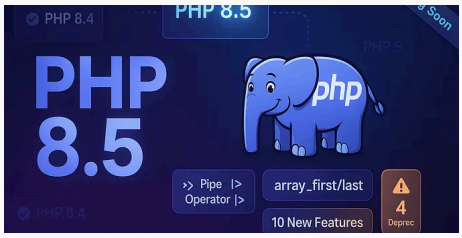


Jakub Skowron (skowron.dev)

Stop Blocking Your PHP: Real Async I/O with Symfony, Fibers,...

Below is a complete, practical guide to asynchronous processing in PHP: from why...

Sep 30 15

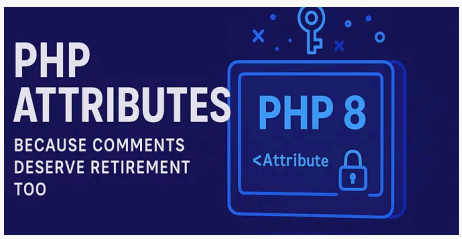


Developer Awam

PHP 8.5 Coming November 2025: 10 New Features and 4...

Discover 10 new features in PHP 8.5 including Pipe Operator, array_first/last, and 4 key...

Oct 28 70 1



Lukasz Lupa

🧠 PHP Attributes: Because Comments Deserve Retirement...

Ivan Vulovic

FrankenPHP vs PHP-FPM (Part 3): CPU, Memory, and the Hidden Co...

Learn how PHP attributes (annotations) work, when to use them, and why they've replaced...

Oct 8 31 2

FrankenPHP vs PHP-FPM: Part 3 explores how they behave when idle, under real...

Aug 4 376 7

All your favorite parts of Medium are now in one sidebar for easy access.

See more recommendations