Refactoring Legacy Application by using “Strangler Fig Pattern”

Unblocking Feature Delivery in Legacy Application by using programming technique named by a plant

Agenda

1. Why do we need it at all? What's happened?
2. Finding a way to refactor legacy application
3. What is “Strangler Fig Pattern”?
4. Lets jump into the PHP and Symfony examples
5. Our journey with Strangler Fig Pattern
6. Questions???

Why do we need it at all? What's happened?

Prerequisites

- Your system is a bunch of spaghetti code.
- But it's OK, because your product growth is flattish. So, you can manage this.
- Development routine looks like:
  - ⅓ “fire fighting”
  - ⅓ solving bugs
  - ⅓ trying to implement new features
- New features implementation are slow and unpredictable.
- But somehow you can call it status quo.

How this escalated

- Business did its best, and the investors came in. You never saw so much money.
- All business metrics are skyrocketing. This is good, very good.
- But software metrics also are skyrocketing: systems load, network traffic, infrastructure cost, sentry errors, etc.
- Also we need new features. Enormous amounts and very fast. Actually they should have been done yesterday.

The Outcome

- It’s almost impossible to deliver new features.
- Every feature takes longer to deliver than the previous one. And brings more bugs and demons than the previous one.
- All are unhappy:
  - Developers are unhappy, because the pressure from business is high, code quality poor and accomplishment feeling below zero. Nobody understands the system.
  - Business people are unhappy, because the progress is slow. With a tendency to decline.
  - Investors also should be unhappy, but business still didn’t tell them that feature delivery is almost frozen due to some technological buzz words. So they are happy for now.

We need to find a way to get out from this legacy code trap, and unlock feature delivery.

The impact of poorly designed legacy code*

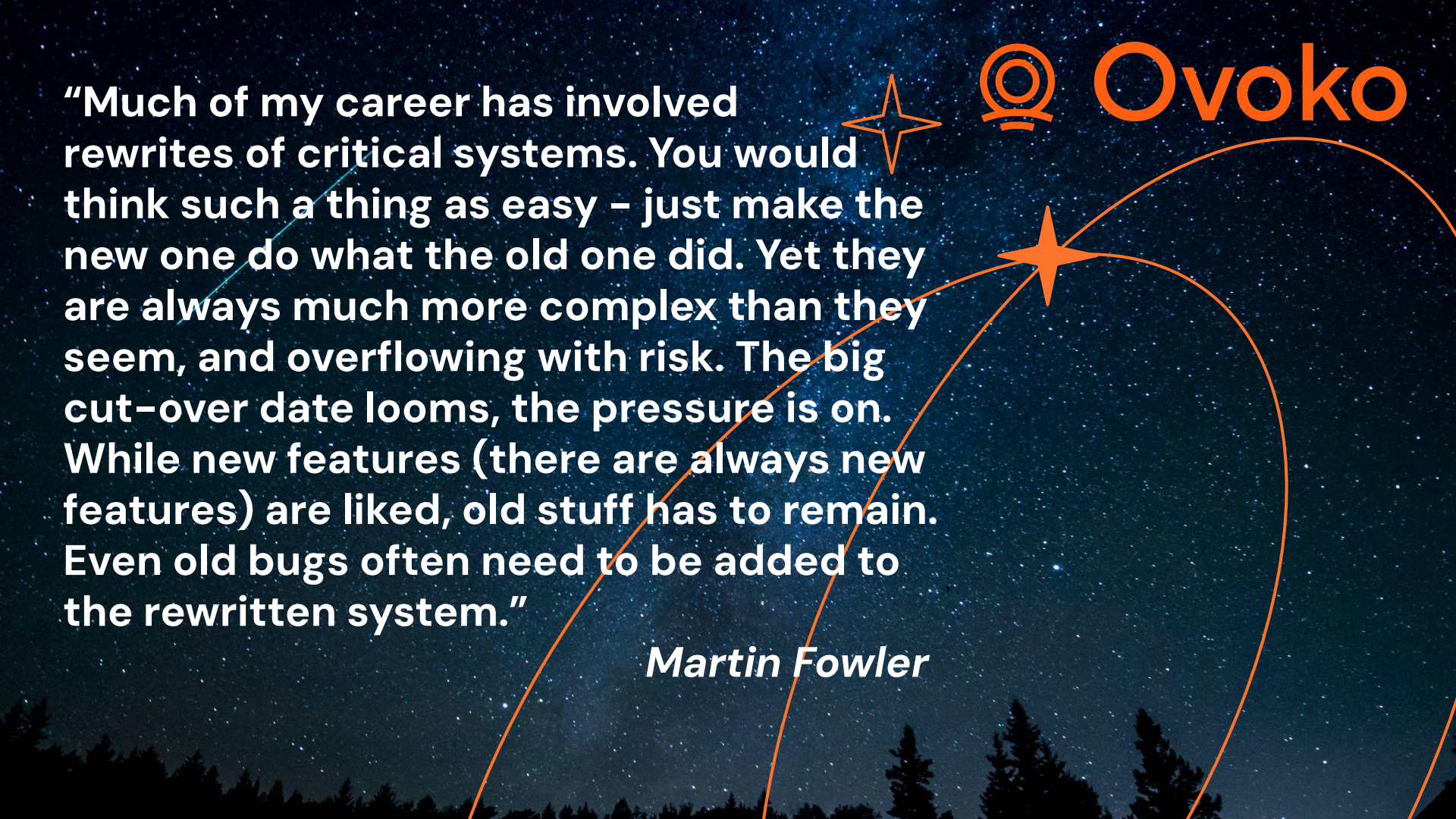
Poorly designed legacy code property	Impact on delivery speed	Impact on developer satisfaction	Impact on maintenance cost	Impact on system stability
Lack of Documentation	Medium	Low	Medium	Medium
“Spaghetti” Code	High	High	High	High
Lack of Modularity	High	High	High	High
Obsolete Technologies	Medium	Low	Low	Medium
Security Vulnerabilities	Low	Low	Low or High	Low or High
Limited Test Coverage	Medium	Medium	Medium	Medium
Tight Coupling	High	Medium	High	High

** It’s a subjective interpretation without backed reference data. The results may vary depending on the project.*

In Short

You have to deal with legacy code, because your project is lucky. It survived and is kicking off. Now it’s your job to make that project’s codebase to be stable and scalable enough, in order to unlock feature delivery.

Finding a way to refactor legacy application


Ovoko
“Much of my career has involved rewrites of critical systems. You would think such a thing as easy - just make the new one do what the old one did. Yet they are always much more complex than they seem, and overflowing with risk. The big cut-over date looms, the pressure is on. While new features (there are always new features) are liked, old stuff has to remain. Even old bugs often need to be added to the rewritten system.”
Martin Fowler

There are many possible ways. What to chose?

1. **Strangler Fig pattern.** This pattern used to incrementally replace or refactor a legacy system with a new one. It is typically employed when you have an existing application that is difficult to maintain, extend, or adapt to changing requirements. The primary goal of the Strangler Fig Pattern is to gradually replace the old system while ensuring that it remains functional and reliable during the transition.
2. **Parallel Run.** You create a parallel system that runs alongside the existing one. This parallel system might use a different technology or architecture. It doesn't matter. Data is synchronized between the old and new systems. When the new system is ready, you can switch over to it entirely.
3. **Legacy Adapter.** Create an adapter layer that sits between the legacy system and any new components. This adapter can translate requests and responses between the old and new systems. This approach enables a gradual transition while isolating the legacy system.

4. **Big Bang Rewrite**. While the Strangler Fig Pattern is about gradual replacement, the Big Bang Rewrite involves rebuilding the entire system from scratch and replacing it all at once. I guess, no need for more words here.
   "The only thing a Big Bang rewrite guarantees is a Big Bang!" *Martin Fowler*
5. **Microservices**. You can break your application down into smaller, loosely coupled microservices. This approach allows you to gradually replace or rebuild specific parts of your system while leaving the rest intact. Back in the days this was the most escalated way of splitting the monolith. By the way, for breaking the monolith, you can also use Strangler Fig Pattern
6. **Feature Toggles (Feature Flags)**. If the mess in a code is not too big, such technique may be just enough for gradual legacy modules replacement. Feature Toggles are not a pattern, but rather a programming technique that is useful in many software development situations.
7. **Incremental Improvement**. Yes, you read it right. Instead of focusing on a complete system replacement, you can incrementally improve the existing system by addressing specific pain points and technical debt over time.
8. ...

Choose “Strangler Fig Pattern”, when:

1. You want to modernize your legacy application without disrupting existing users and business processes. You can’t use turn on/off button.
2. You lost hope in your legacy code. There is no chance to fix it, only to replace it.
3. There are plenty of new features and other business requests. And you want them to be done in a new codebase.

What is “Strangler Fig Pattern”?

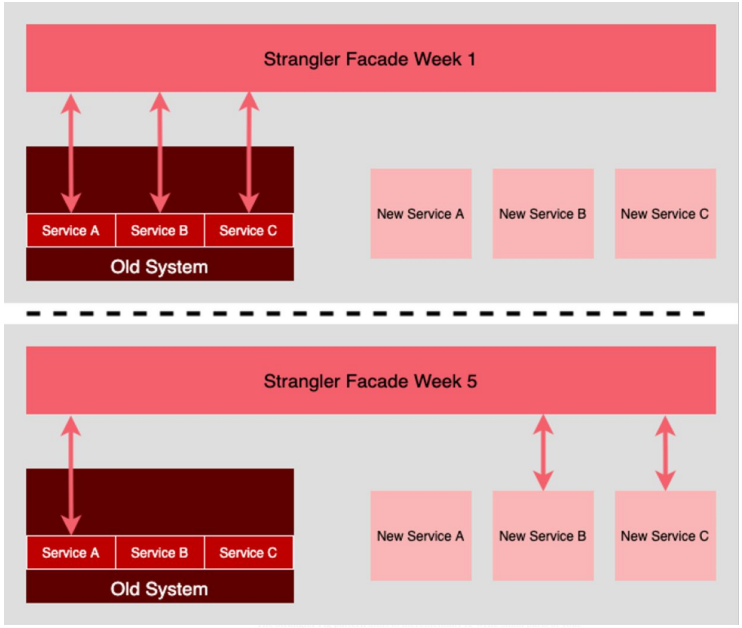
Strangler Fig Pattern

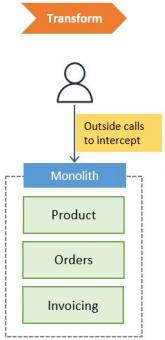
The Strangler pattern is one in which an “old” system is put behind an intermediary facade. Then, over time external replacement services for the old system are added behind the facade.

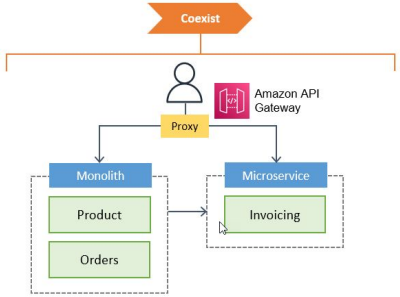
The facade represents the functional entry points to the existing system. Calls to the old system pass through the facade. Behind the scenes, services within the old system are refactored into a new set of services. Once a new service is operational, the intermediary facade is modified to route calls that used to go to the service on the old system to the new service. Eventually, the services in the old system get "strangled" in favor of the new services.

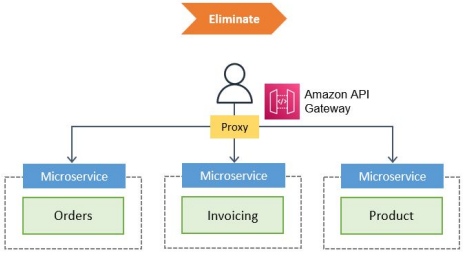
redhat.com

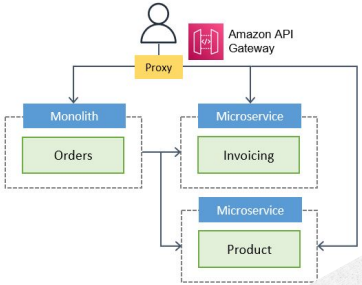
They all say it should look something like this

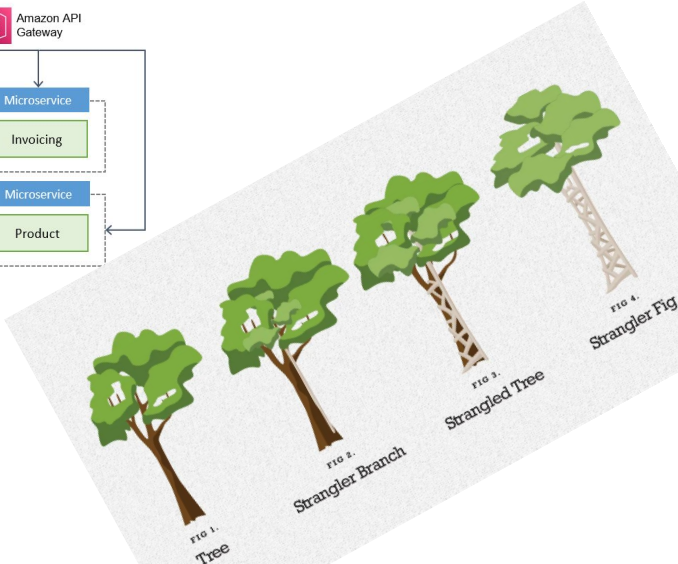


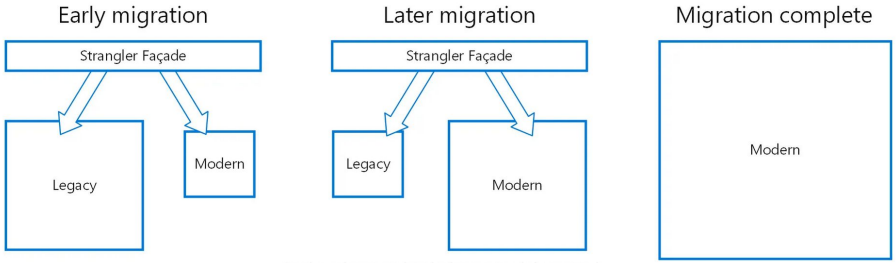




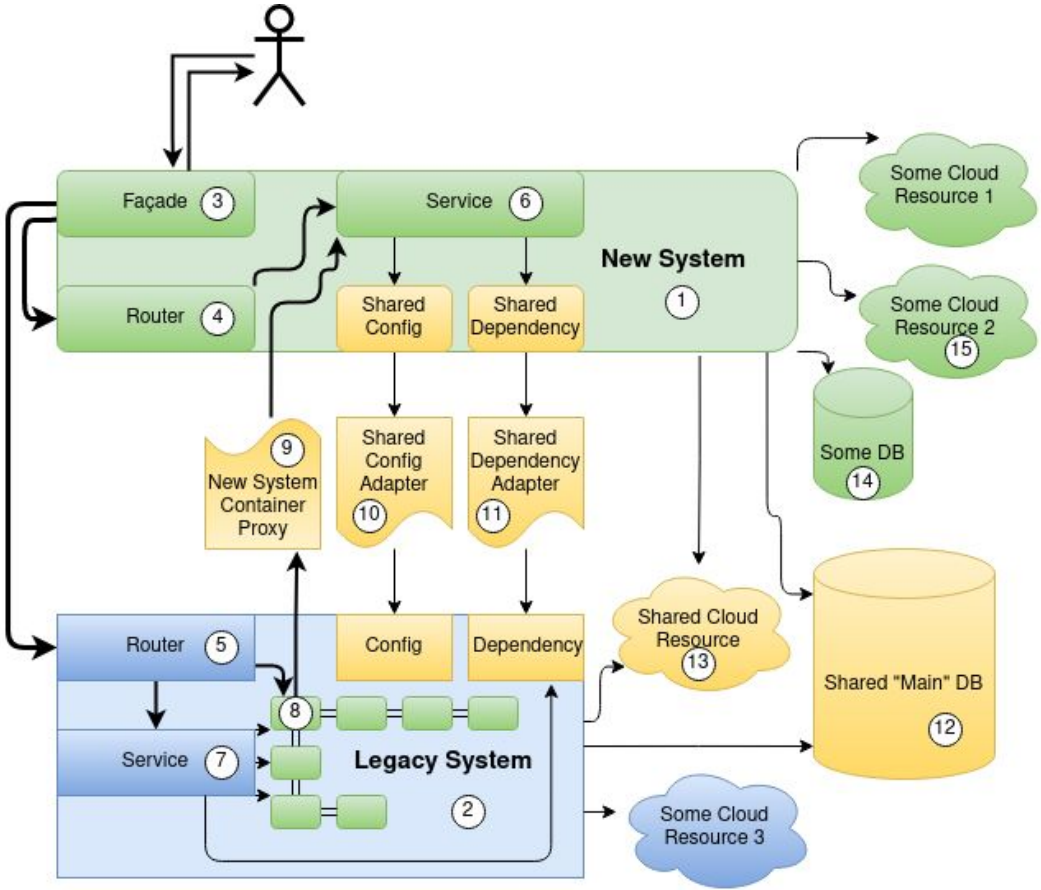








But in reality it looks something like this



1. New application.
2. Legacy application.
3. Façade, which choses where to redirect requests, old or new application. It usually happens to be built inside a new application.
4. New application router.
5. Legacy application router.
6. One of the new application service.
7. One of the legacy application service.
8. New application service injected inside old application code.
9. In order to access new app service inside the legacy app, we need a proxy service to new app container.
10. Usually many configs are coupled with legacy system. We need some adapter in order to access them.
11. Another shared dependency adapter.
12. The old one (but not good one) shared DB.
13. Some shared cloud infrastructure.
14. Some DB used by only one application.
15. Some cloud resource used by only one application.

It’s not only about the code

The Strangler Fig Pattern principles are often used in a non-software context, in a physical world, when high scale changes should not disrupt current flow.

As an example, the process of street gas lamps changing to electric ones in the late 19th century, is accurately reflecting this pattern. First parallel electric lines were installed, and only after full switch, old one gas lamps were removed.
I’d say, it is possible that currently we see a similar transformation with electric cars. While infrastructure for electric cars are built in parallel with internal combustion engine cars infrastructure, this can be the first parallelisation step, if we suppose that combustion engine cars will disappear from general society in a far future.

Lets jump into the PHP and Symfony examples

Migrating an Existing Application to Symfony

Strangler Fig Pattern is an “official” way to move your legacy application to Symfony. So we won’t go deep into the details, because it’s well documented (docs). But here are main preparation step takeaways, slightly modified by my personal interpretation:

1. **Wisely select your library versions.** Symfony, its bundles, other dependencies and whole legacy application should be compatible in terms of PHP version, composer packages, etc. Sometimes first you’ll need to upgrade your legacy application a bit, in order to work with SFP.
2. **Get rid of the “global” state.** Actually, try to get rid of as many such buzzwords as possible: global, static, shared, etc. Sooner or later they'll be pain in the ass.
3. **Cover critical paths with end-to-end tests.** You’ll be replacing blocks of legacy logic with overridden Symfony based logic. That definitely requires a safety net.

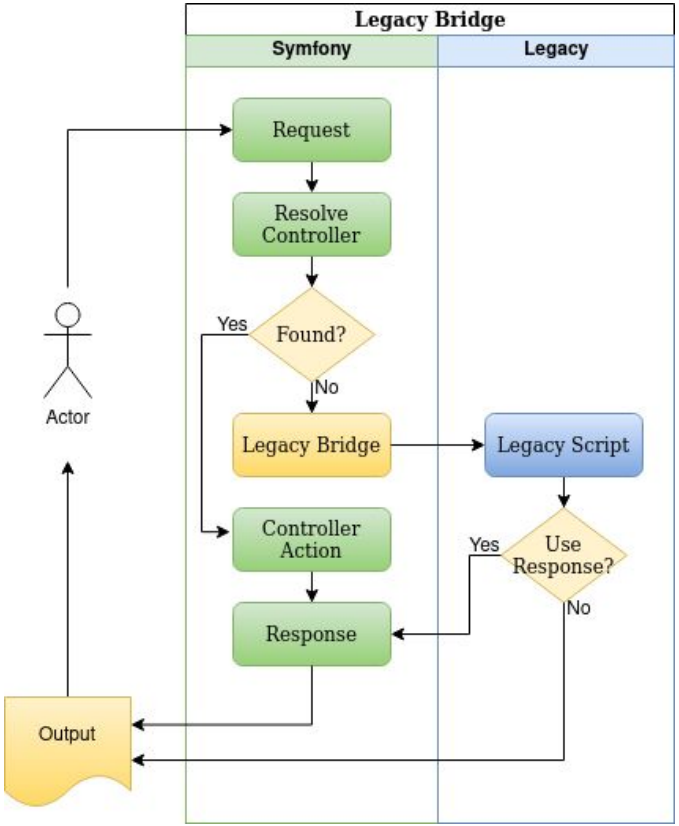
Two major approaches

No matter which approach you'll choose, first create a Front Controller script. In short - all requests should be redirected to ***public/index.php***. And your Symfony application should be the first one handling all requests. This is a part of the standard Symfony installation process. Then choose your destiny:

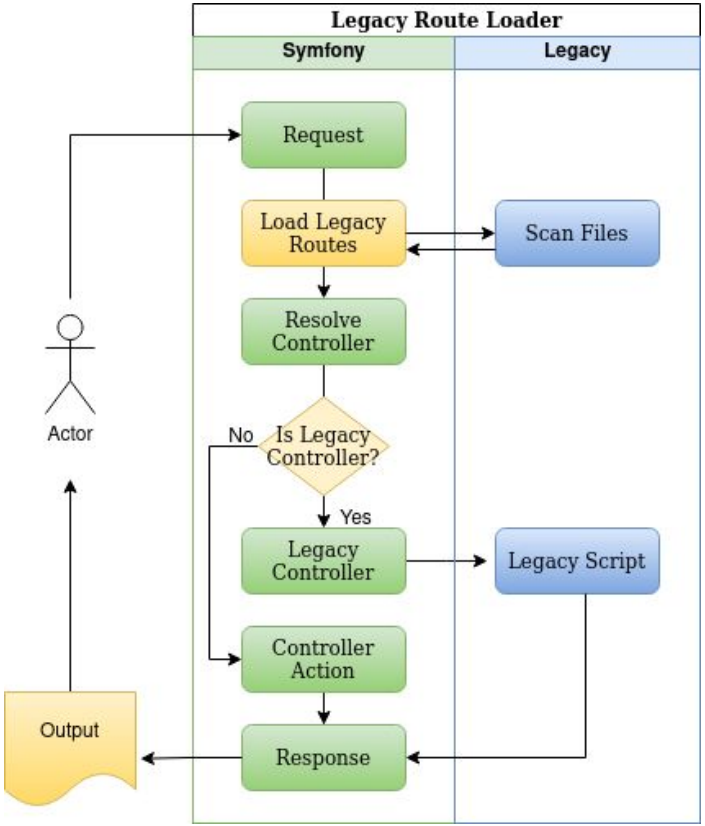
1. **Legacy Bridge,** which leaves the legacy application untouched and allows migrating it in phases to the Symfony application. This is a bit more decoupled way. If a Symfony route is found, then the legacy application is never touched.

2. **Legacy Route Loader**, where the legacy application is integrated in phases into Symfony, with a fully integrated final result. A bit more coupled approach. Legacy script file acts as a controller action logic inside a Symfony application.

Legacy Bridge



Legacy Route Loader



Legacy bridge. The code

```php
<?php
// public/index.php
require dirname( path: __DIR__) . '/vendor/autoload.php';
// Some legacy config load
require dirname( path: __DIR__) . '/legacy/constants.php';

(new Dotenv())->bootEnv(dirname( path: __DIR__) . '/.env');

/**
* The Symfony kernel will always be available globally, allowing you to
* access it from your existing application and through it the
* service container. This allows for introducing new features in
* the existing application.
*/
global $kernel;

$kernel = new Kernel($_SERVER['APP_ENV'], (bool)$_SERVER['APP_DEBUG']);
$request = Request::createFromGlobals();
$response = $kernel->handle($request);

if (false === $response->isNotFound()) {
    // Symfony successfully handled the route.
    $response->send();
} else {
    /**
     * No Symfony route found, legacy app should process the request.
     * Sometimes direct load of legacy bootstrap is enough:
     * require __DIR__ . '/../legacy/bootstrap.php';
     */
    LegacyBridge::handleRequest($request, $response, __DIR__);
}

$kernel->terminate($request, $response);
```

```php
<?php
// src/LegacyBridge.php
namespace App;
// ...
1 usage
class LegacyBridge
{
    /**
     * Map the incoming request to the right file. This is the
     * key function of the LegacyBridge.
     */
    1 usage
    public static function getLegacyScript(Request $request): string
    {
        // Map a route to a legacy script:
        if ($request->getPathInfo() == '/customer/') {
            return __DIR__ . '/../src/customers/list.php';
        }
        // ... etc.
    }

    no usages
    public static function handleRequest(Request $request, Response $response): void
    {
        $legacyScriptFilename = LegacyBridge::getLegacyScript($request);

        // Possibly (re-)set some env vars, etc
        $p = $request->getPathInfo();
        $_SERVER['SCRIPT_NAME'] = $p;
        $_SERVER['SCRIPT_FILENAME'] = $legacyScriptFilename;

        require $legacyScriptFilename;
    }
}
```

Legacy route loader. The code

```php
<?php
// src/Legacy/LegacyRouteLoader.php
namespace App\Legacy;

use Symfony\Component\Config\Loader\Loader;
use Symfony\Component\Routing\Route;
use Symfony\Component\Routing\RouteCollection;

no usages
class LegacyRouteLoader extends Loader
{
    no usages
    public function load($resource, $type = null): RouteCollection
    {
        $collection = new RouteCollection();
        $finder = new Finder();
        $finder->files()->name('*.php');

        /** @var SplFileInfo $legacyScriptFile */
        foreach ($finder->in($this->webDir) as $legacyScriptFile) {
            // This assumes all legacy files use ".php" as extension
            $filename = basename($legacyScriptFile->getRelativePathname(), suffix: '.php');
            $routeName = sprintf( format: 'app.legacy.%s', str_replace( search: '/', replace: '__', $filename));

            $collection->add($routeName, new Route($legacyScriptFile->getRelativePathname(), [
                '_controller' => 'App\Controller\LegacyController::loadLegacyScript',
                'requestPath' => '/' . $legacyScriptFile->getRelativePathname(),
                'legacyScript' => $legacyScriptFile->getPathname(),
            ]));
        }

        return $collection;
    }
}
```

```php
<?php
// src/Controller/LegacyController.php
namespace App\Controller;

use Symfony\Component\HttpFoundation\StreamedResponse;

no usages
class LegacyController
{
    no usages
    public function loadLegacyScript(string $requestPath, string $legacyScript): StreamedResponse
    {
        return new StreamedResponse(
            function () use ($requestPath, $legacyScript): void {
                $_SERVER['PHP_SELF'] = $requestPath;
                $_SERVER['SCRIPT_NAME'] = $requestPath;
                $_SERVER['SCRIPT_FILENAME'] = $legacyScript;

                chdir(dirname($legacyScript));

                require $legacyScript;
            }
        );
    }
}
```

Our journey with Strangler Fig Pattern

Ovoko ERP application before SFP introduction

- Currently application already counts for about 8 years.
- System was/is written on the Codeigniter framework:
  - PHP version 7.4
  - PHP files based HTML templates.
  - Multi Tenant system.
  - Application dependant on server filesystem. Not stateless.
  - Framework files committed into the repository, so no availability to safely upgrade the system.
- Application had **~150 000 Codeigniter** based lines of code. Count doesn't include template files, but include framework files, while they are committed.
- Application had **0 Symfony** based lines of code.
- Our ERP system was used by **~1000** scrapyards at the time. Mostly in LT.
- Majority of the delivery time was bug fixing and firefighting.
- New features were delivered very slowly and with new bugs introduced.

Ovoko ERP application after SFP introduction

- Strangler Fig Pattern introduced about 2 years ago.
- Current PHP version 8.1. Current Symfony version 6.1.
- Application is stateless now.
- Application has **~60 000 Codeigniter** based lines of code. Count doesn't include template files, but include framework files, while they are committed.
- Application has **~70 000 Symfony** based lines of code.
- During those 2 years, our legacy codebase shrinked for about **60%**.
- Our ERP system is used by **~3000** scrap yards in 8 Europe countries.
- Majority of the delivery time is new features delivery.
- About 80% of the time developers are working on Symfony codebase.
- Codeigniter codebase is in maintenance mode, and only bug fixes are made there. All new features goes to Symfony codebase.

The pitfalls. There are plenty of them

Authentication. In order to authenticate Symfony requests, you have to have Legacy/Symfony authentication in sync. While this sounds easy, it's not. Their sessions, implementation, firewalls, etc. differ highly. Moreover, legacy systems tend to have highly custom sessions with fully loaded redundant data. This may not be an issue if you have an API based application.

Authorisation. Don't expect that Symfony role-based access control (RBAC) will be the same as your legacy application. Most likely you'll have to make some bridge or adapter for both authorization systems.

Users. A lot of core functionality relies on users. From our trial-error experience, users related functionality should be re-factored at first. This will unlock such features as authentication and authorisation. Those also should be prioritized. Without auth, you'll always try to do some workarounds in order to satisfy both systems requirements.

Tests. Moving legacy code to a new code base is a very risky operation. Mostly, because you are working in a black box. You can never be sure, that new business logic implementation works the same as worked before. Cover critical paths with end-to-end tests.

Database and ORM. You'll have to write ORM DTOs to all your database tables. Unless your application doesn't require it, this is hard, tedious and risky work. Chances your database schema integrity is correct and can be used for DTOs (Entities) auto-generation is very low.

Templates. Again, this is not an issue if you have an API based application. But full stack applications usually have many templates, and those templates have hierarchy. Re-using those templates may be impossible.

Feature Toggles. They are very helpful when working with SFP. While constantly replacing old code with new one, with feature toggle you can easily switch between old and new code states without breaking the system. Also the whole deployment and testing process becomes much smoother and less nervous.

Discipline. While working with SFP, it's very important to have discipline. Task is not easy, it requires high competencies and there are plenty of unknowns, and often you'll want to cut some corners in order to do things faster. But it's a trap. Always have a plan, stick to a plan, but be prepared to change it, when there is no other possibility.

Balancing refactoring and business needs. This is the toughest one. Refactoring, especially at the beginning, is time consuming. You have to build many proof of concepts, write boilerplates for many low level features, and spend a lot of time debating. But when you implement that core functionality, life will become easier. And then you'll reach that point, when building new features and refactoring at the same time is faster, than it was before only building new features on a legacy system.
But at the beginning probably you won't satisfy all business needs. Or, otherwise, refactoring will be stuck. Communication is the key for this problem. You must clearly communicate why refactoring is required, and how it will help for the company, for the product, etc. Just communicate.


Ovoko
Questions???

ovoko.com

career@ovoko.com

+370 601 32921


Vilnius
PHP
Upcoming event dates
2024 March 07
2024 June 06
2024 September 05
2024 December 05


Vilnius
PHP
0x88
Build Stuff
Ovoko