
Microservice
Architecture
for EE Addons
EECONF
2017
DENVER

Julian Fann

TECHNICAL LEAD

Vector Media Group

julian@vectormediagroup.com
github.com/julianfann
@booleanfann

vectormediagroup.com


vectormediagroup.com

What this talk is

Overview of microservice architecture

What can we learn from its best practices?

How can we apply those to EE addons and site builds?

What this talk is not

A primer on cloud services

How to build and deploy “actual microservices”

what is

Microservice Architecture?

Definition

...the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API

—Martin Fowler, ThoughtWorks

Definition

An architecture that structures an application as a set of lean, loosely coupled, collaborating services.

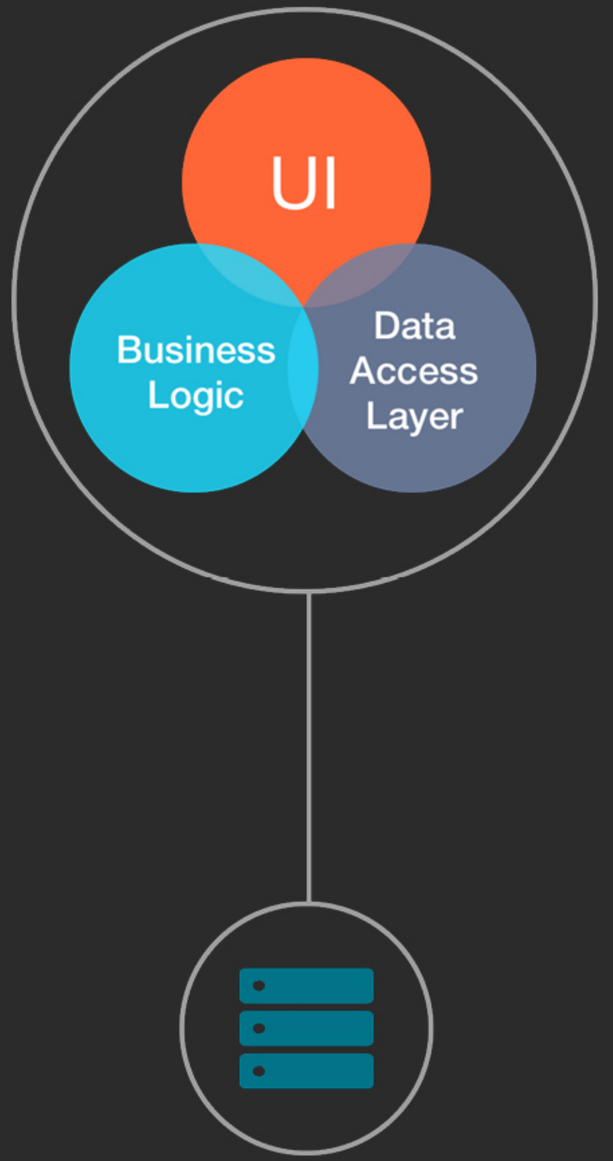
Principles

- A large application is broken into several small services
- Services are organized around capabilities
- Services communicate using APIs
- Each service is small, encapsulated and focused
- Services are easy to replace


Monolithic
vs
Microservices

Monolithic

A large software application, that has grown over time and that is hard to maintain and evolve.



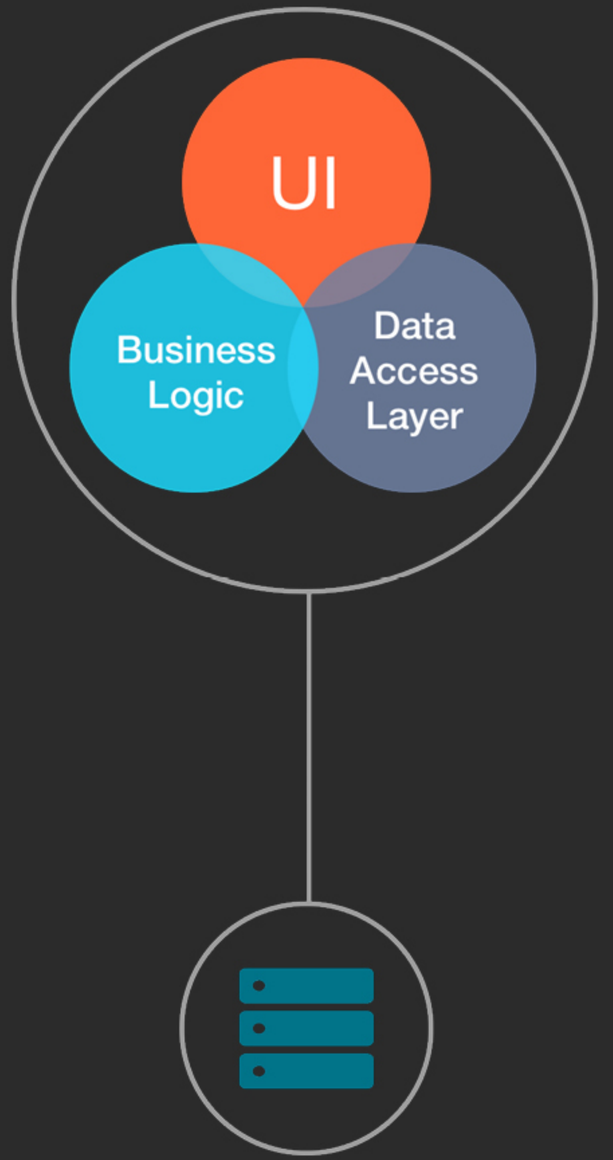
Monolithic

Business logic is highly coupled

Refactor required to make small changes

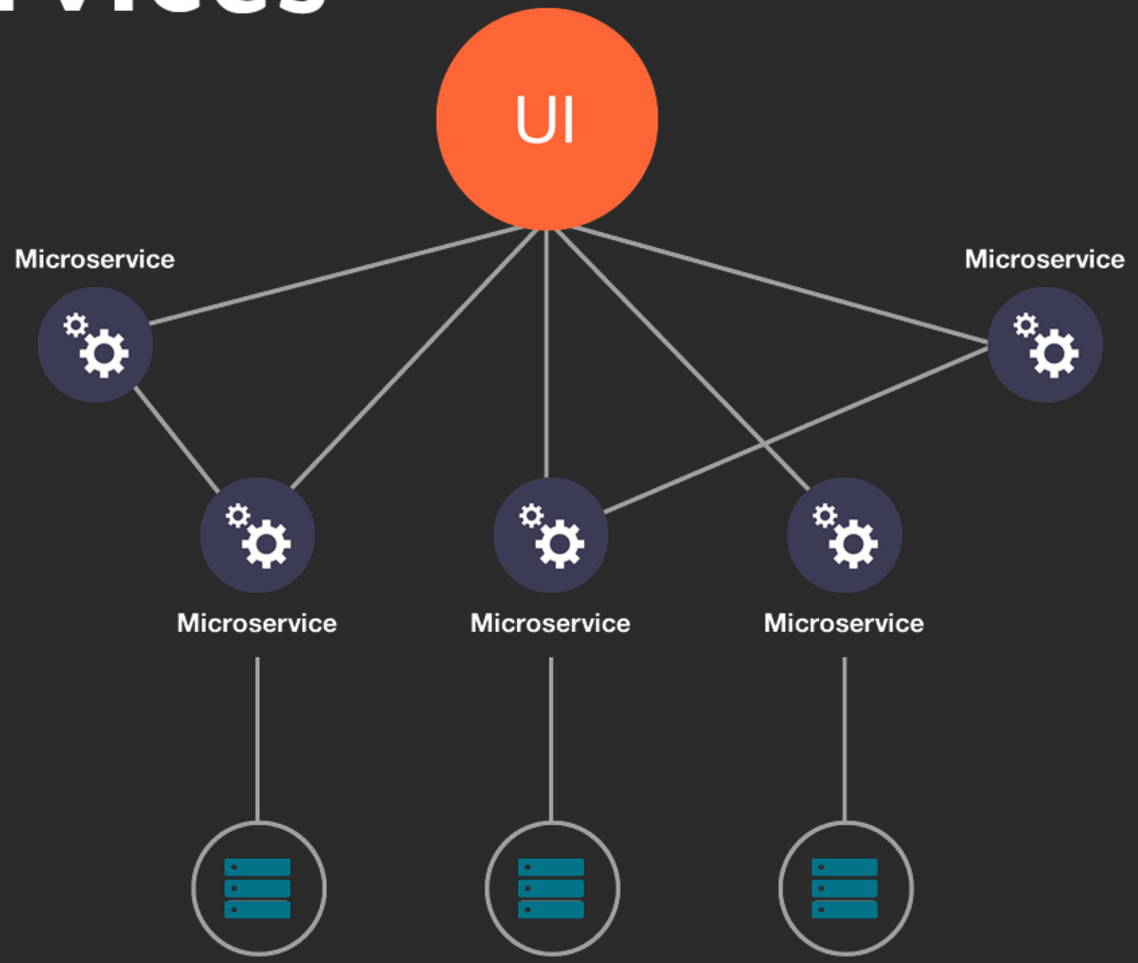
Difficult to build and deploy changes

Failure in one area cascades to the whole application



Microservices

Application broken into independent, small services



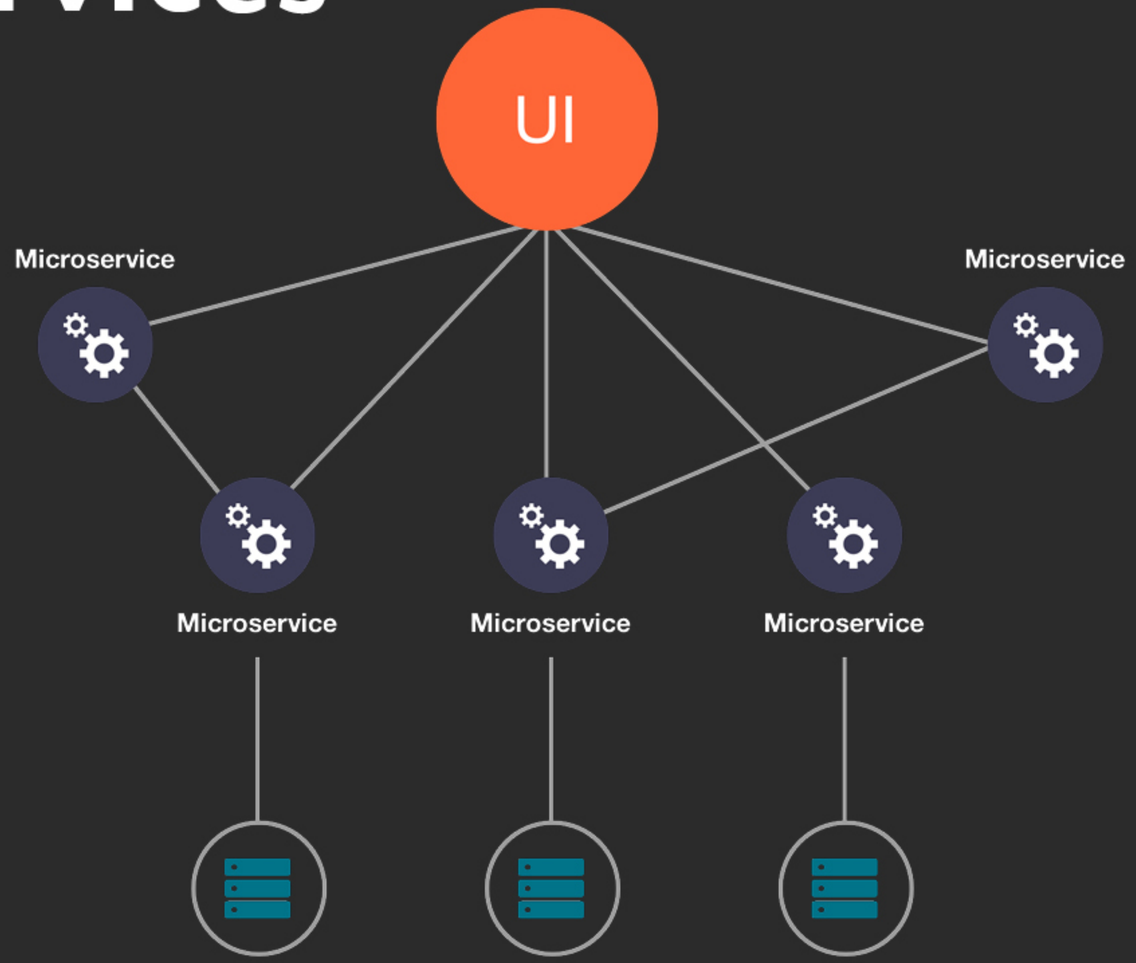
Microservices

Business logic adheres to API contracts.

Changes are either invisible or versioned.

Build and deployment of a service is easy and fast.

Failure in one service only impacts part of the application



Examples in the Wild


how can
Microservice Architecture Improve Applications?

Each service has a single responsibility

Services adhere to API contracts
and implementations can be swapped

Internal service use allows for
the creation of robust public APIs

Services can be upgraded independently
with little or no service disruption


how can we use
Microservice Architecture in EE?

Principles

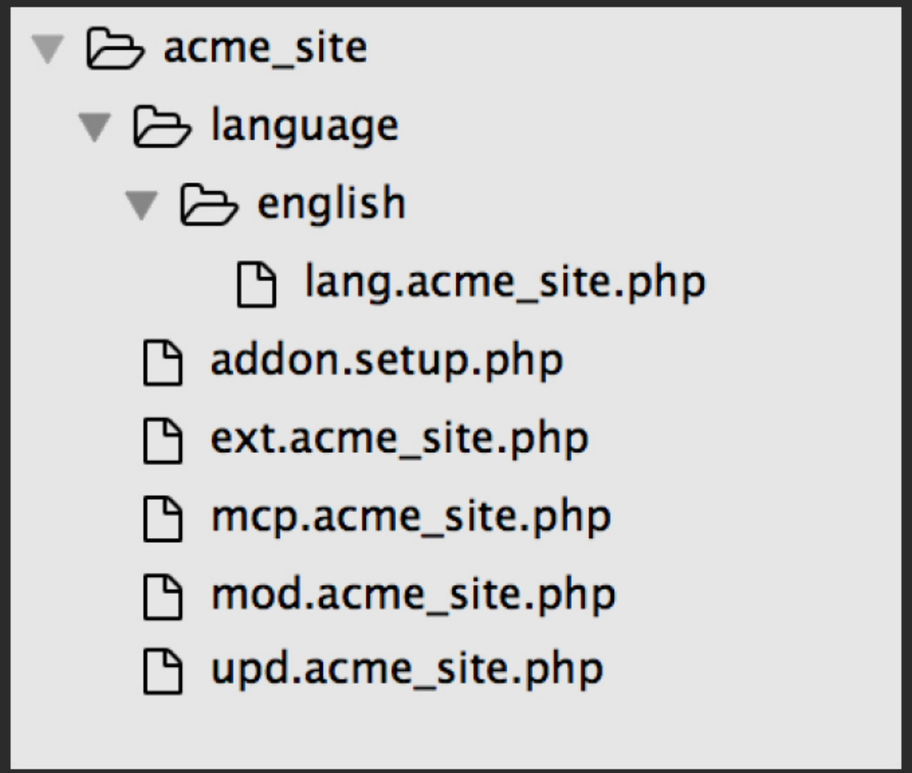
- A large application is broken into several small services
- Services are organized around capabilities
- Services communicate using APIs
- Each service is small, encapsulated and focused
- Services are easy to replace

Principles

- A large application is broken into several small services
- Services are organized around capabilities
- Services communicate using APIs
- Each service is small, encapsulated and focused
- Services are easy to replace

Break apart larger addons into smaller, focused addons

The Custom Site Module



The Custom Site Module

Consolidates most custom functionality in one place

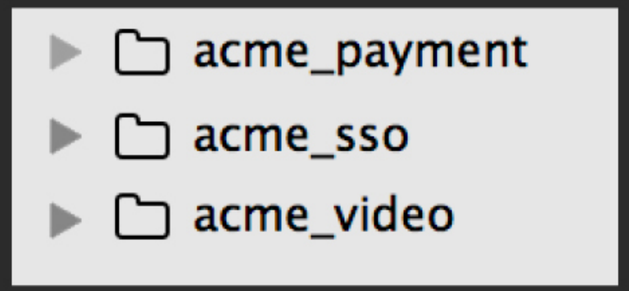
Results in massive mod, mcp, and ext files

Difficult to trace logic chains since unrelated functionality is commingled

One line of code can break the entire addon

Difficult to know exactly what it does at a glance

Create separate addons based on capability



Principles

- A large application is broken into several small services
- Services are organized around capabilities
- Services communicate using APIs
- Each service is small, encapsulated and focused
- Services are easy to replace

Define an addon by its services

What problem does this addon solve?

What individual services do I need to solve it?

How do I define an addon's services?

EE3 Dependency Injection Container

Many parts of EE core functionality are now
exposed using the DI Container

The DI Container is available to any part of an addon

Makes it easy to “new up” a class and its dependencies

Familiar to anyone who has used Laravel or Symfony

```php
use Acme\Payment\Service\Payment;

return array(
    'author'              => 'Acme Corp',
    'author_url'          => 'https://acmecorp.com',
    'description'         => 'Acme payment gateway',
    'name'                => 'Acme Payment',
    'namespace'           => 'Acme\Payment',
    'settings_exist'      => false,
    'version'             => '1.0',
    'services'    => [
        'payment'    => function($addon){
            $apiKey = ee('Config')->get('stripe_api_key');
            $options = ee('Config')->get('stripe_options');

            return new Payment\StripePaymentGateway($apiKey, $options);
        }
    ],
);
```

acme_payment/addon.setup.php

Principles

- A large application is broken into several small services
- Services are organized around capabilities
- Services communicate using APIs
- Each service is small, encapsulated and focused
- Services are easy to replace

```php
public function post_payment_form()
{
    $amount = ee()->input->post('amount');
    $cardToken = ee()->input->post('card_token');

    $response = ee('acme_payment:payment')
                    ->charge([
                        'amount' => $amount
                        'token'  => $cardToken
                    ]);

    if(!$response){
        // throw some error
    }

    // redirect to a thank you page
}
```

acme_payment/mod.acme_payment.php or other_addon/mod.other_addon.php

Principles

- A large application is broken into several small services
- Services are organized around capabilities
- Services communicate using APIs
- Each service is small, encapsulated and focused
- Services are easy to replace

```php
namespace Acme\Payment\Service\Payment;

use Stripe\Stripe;
use Stripe\Charge;

class StripePaymentGateway implements PaymentGatewayInterface {

    private $apiKey;
    private $options;

    public function __construct($apiKey, $options = null){
        $this->apiKey = $apiKey;
        $this->options = $options;

        Stripe::setApiKey($stripeApiKey);

        // Set Stripe options

    }

    public function charge($params = null, $options = null){

        // Check params for
        // required items

        $charge = Charge::create([
            'amount' => $params['amount'],
            'currency' => 'usd',
            'source' => $params['token']
        ]);

        // Check response and
        // return true or false

    }

}
```

acme_payment/Service/Payment/StripePaymentGateway.php

Principles

- A large application is broken into several small services
- Services are organized around capabilities
- Services communicate using APIs
- Each service is small, encapsulated and focused
- Services are easy to replace

```php
namespace Acme\Payment\Service\Payment;

interface PaymentGatewayInterface {

    private $apiKey;
    private $options;

    public function __construct($apiKey, $options = null);

    public function charge($params = null, $options = null);

}
```

acme_payment/Service/Payment/PaymentGatewayInterface.php

```php
use Acme\Payment\Service\Payment;

return array(
    'author'              => 'Acme Corp',
    'author_url'          => 'https://acmecorp.com',
    'description'         => 'Acme payment gateway',
    'name'                => 'Acme Payment',
    'namespace'           => 'Acme\Payment',
    'settings_exist'      => false,
    'version'             => '1.0',
    'services'    => [
        'payment'    => function($addon){
            $apiKey = ee('Config')->get('paypal_api_key');
            $options = ee('Config')->get('paypal_options');

            return new Payment\PaypalPaymentGateway($apiKey, $options);
        }
    ],
);
```

acme_payment/addon.setup.php

```php
use Acme\Payment\Service\Payment;

return array(
    'author'              => 'Acme Corp',
    'author_url'          => 'https://acmecorp.com',
    'description'         => 'Acme payment gateway',
    'name'                => 'Acme Payment',
    'namespace'           => 'Acme\Payment',
    'settings_exist'      => false,
    'version'             => '1.0',
    'services'    => [
        'payment'   => function($addon){
            $gateway = ee('Config')->get('acme_payment_gateway', 'Stripe');
            $classname = 'Acme\\Payment\\Service\\Payment\\'.$gateway.'PaymentGateway';

            return new $classname(
                ee('Config')->get('acme_payment_api_key'),
                ee('Config')->get('acme_payment_options')
            );
        }
    ],
);
```

acme_payment/addon.setup.php

Prepare for scale now

Add flexibility by coding to interfaces

Make few assumptions about the future

Ask, “Can this functionality easily be replaced by an http request to another server?”

Ask, “Do I have to store stateful information?”

Commercial Addons

Expose services to allow developers to use
your addon in ways that you have not yet conceived

Allow for flexibility to support multi-node
cloud infrastructure for larger, high traffic sites

Consider bundling your paid
addons with free support addons

Summary

Microservice Architecture has many
principles that can be applied to your projects now

Break up the giant site addon into capability focused addons

Start using the EE3 DI Container to expose addon services

Start thinking about scale from the beginning

Further Reading/Viewing

Netflix Tech Blog

techblog.netflix.com

AWS re:Invent

reinvent.awsevents.com

Martin Fowler

martinfowler.com
GOTO 2014 Talk

Questions?

KEEP IN TOUCH

julian@vectormediagroup.com
github.com/julianfann
@booleanfann

vectormediagroup.com