Transform your monolith into a nice SOA application

About – me

- @matgillot

  - Backend developer
  - Open-source enthusiast
  - Enjoy
    - Working
    - Traveling
    - Tractors

About – this presentation

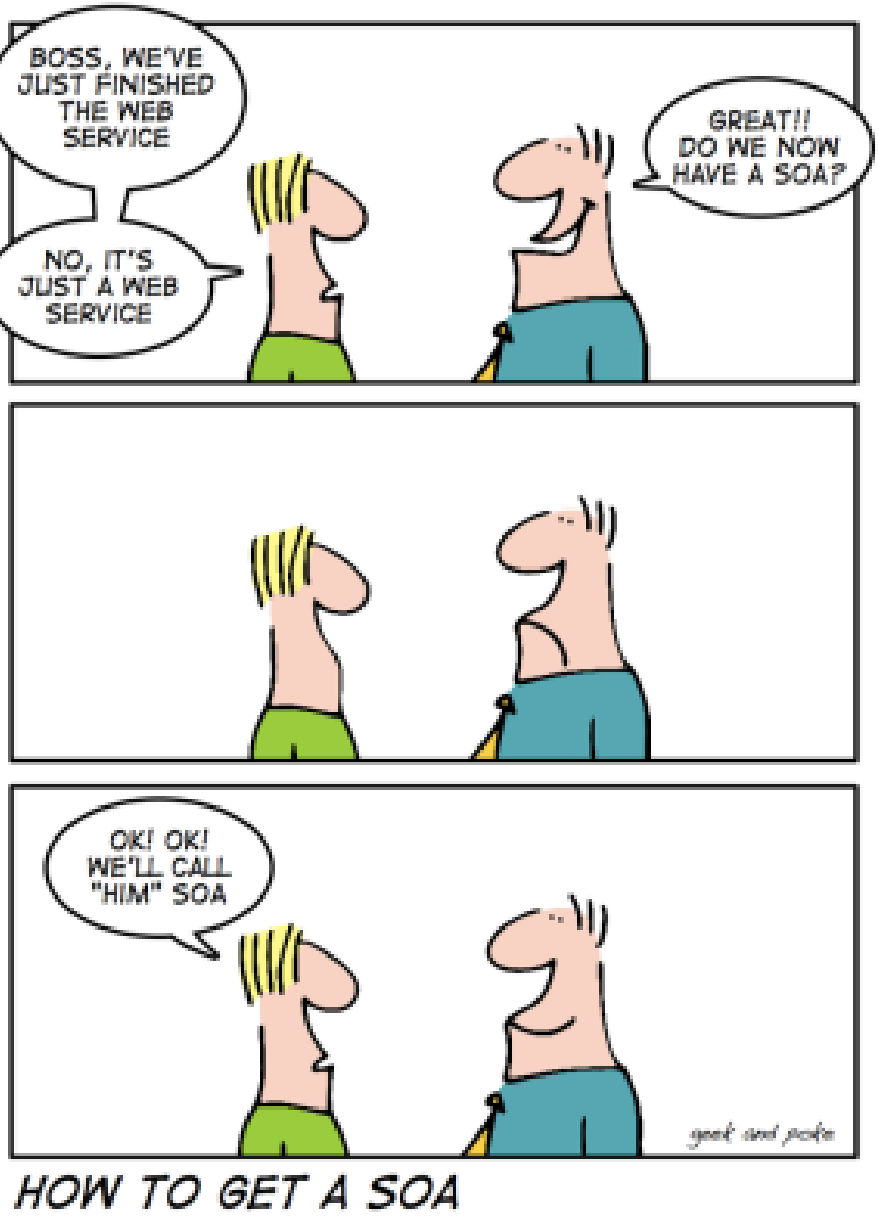
- Focused on the migration of a live application
- Make your app easy to
  - Work with
  - Understand
  - Scale

Plan

- SOA architecture
- Refactoring
- Best Practices
- Migration

SOA Architecture

- Concepts
- Principles
- SOA in your application



SOA - Concepts

Business value	Technical strategy
Strategic goals	Project-specific benefits
Inter-operability	Custom integration
Shared services	Specific-purpose implementations
Flexibility	Optimization
Evolutionary refinement	Initial perfection

SOA - Principles

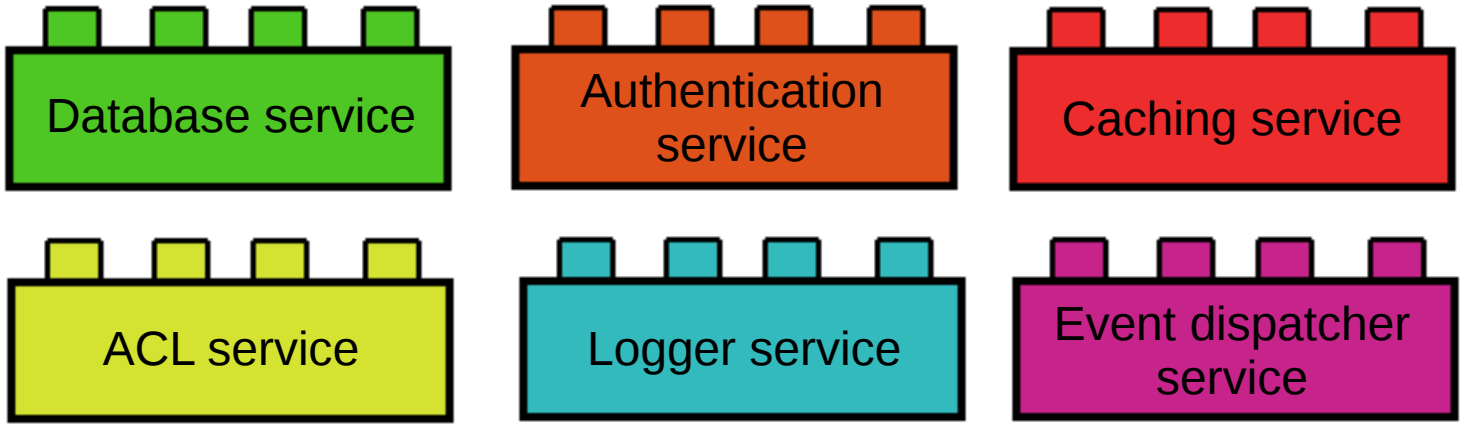
- Agnostic services
  - **Abstraction:** services act as black boxes
  - **Statelessness:** return the requested value or give an exception
  - **Composability:** services can be used to compose other services
  - **Reusability:** logic is divided into various services, to promote reuse of code
  - **Encapsulation:** services which were not initially planned under SOA, may get encapsulated or become a part of SOA

Encapsulation

```php
$container['projectManager'] = $container->share(function($container){
    require_once(LIBRARY . '/core/project/projectManager.php');
    return \ProjectManager::getInstance();
});
```

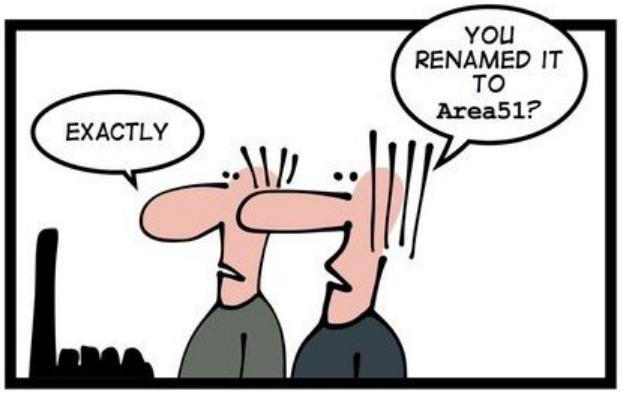
SOA in your application - kill the monolith !

- Break coupling!
- Use dependency injection
- Composability: Play Lego (TM)



Refactoring

- Concept
- Daily work
  - TDD refactoring
- Optimistic refactoring
  - Litter-Pickup Refactoring
  - Comprehension Refactoring
- Before starting a development
  - Preparatory refactoring
- Large-scale restructuring tasks
  - Planned Refactoring
  - Long-Term Refactoring

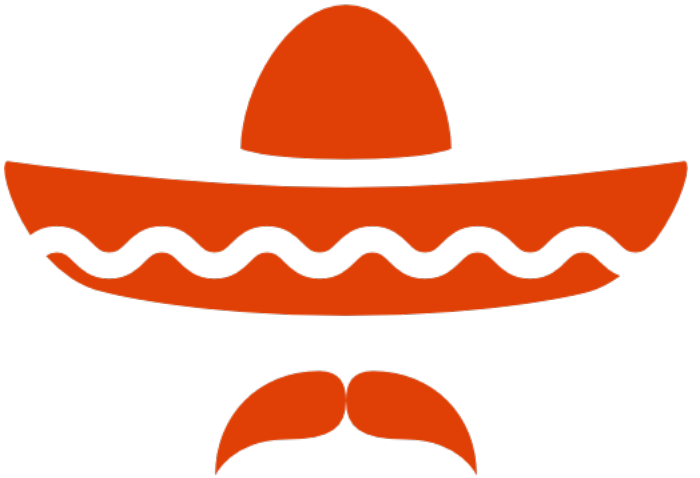


Refactoring : what is it ?

“A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.”

Martin Fowler

Refactoring – two hats



Daily Work - TDD refactoring

- Write failing test

  describe the expected behaviors of your functionality using assertions

- Make test pass 

  focus on adding the new functionality, without thinking about how this functionality should be best structured.

- Refactor 

  Concentrate on good design, while working in the safer refactoring mode of small steps on a green test base

Optimistic refactoring - Litter-Pickup Refactoring

- Boy scout rule

  always leave the code better
  than when you found it.

- Cleaning up code as we work in it

  make things quicker for us the
  next time we need to work with it

Optimistic refactoring - Comprehension refactoring

- Implement clear code is hard

  often you can only tell how to make it clear when someone else looks at it, or you come back to it at a later date.

- Build your understanding of the problem

  whenever you have to figure out what code is doing, you are building some understanding in your head.

- Move it into the code 

  so nobody has to build it from scratch in their head again

Optimistic Refactoring

- Good move if
  - simple fix
  - will make it easier to add the feature you're working on
- Requirements
  - tested and stable codebase
  - require less than [SUBJECTIVE_VALUE] % of the time to develop the feature

Preparatory refactoring

- Refactor codebase before adding a new functionality

  Good move if

  - Overall change is faster than implementation on entire codebase
  - Codebase will be used in many places
  - Codebase is fully tested (if not, split in two tasks)

Large-scale - Planned Refactoring

- Fix larger areas on problematic code
- The more you'll work with quality approach, the less you'll have to do it
- If it happens often, incorporate optimistic & planned refactoring processes in your daily work

Large-scale - Long-Term Refactoring

- Clearly define your needs
- Use branch by abstraction to reduce risk
- Code has to be stable at the end of every small step

Best practices

- Decoupling
- SOLID Pattern
- Test Automation
  - Unit Testing
  - Functional/Integration Testing
  - End-User Testing
  - Code Coverage
- Monitoring

Decoupling

- Law of Demeter
- SOLID principles
- Dependency Injection
- Events
- Event bus

Law of demeter – counter-example

```php
public function changeTire(Car $car, Tire $tire)
{
    // remove old tire
    $car->getWheel()->deleteTire();

    // add a new tire
    $car->getWheel()->addTire($tire);
}
```

Events – Be careful !

Automated testing

- Unit testing
- Integration & functional testing
- End-user testing
- Code coverage

Unit Testing

- Test algorithms/methods individually
- Mock dependencies
- Cover all scenarios
- Don’t interact with environment

Integration & Functional Testing

- Combine units of code and test combination functions correctly
- Test the result of an entire workflow by providing inputs and testing outputs

- Unit test OK
- Integration tests missing

End-User Testing

- Access the application
- Test what is displayed to end-user
- Based on scenarios
- Use test description specification language

Code coverage

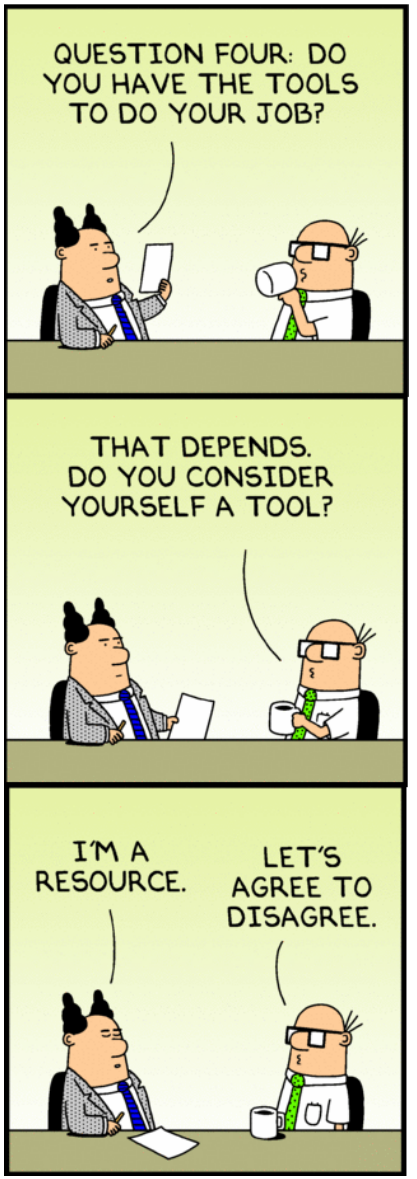
- DON’T make it a target
  - no correlation with code quality
  - focus on risky code
    - cause critical bugs
    - used in many places
    - tricky algorithm
  - 10% coverage for 100% of scenarios is far better than 100% coverage for 10% of scenarios

Monitoring

- Profile and monitor to identify
  - Bottlenecks
  - Heaviest parts of your application
  - Make a distinction between I/O and processing
- Some tools
  - Xhprof
  - Valgrind
  - Pinba
  - ...

Tools

- Compatibility tests
  - Concept
  - Example
- Indicators
  - CRAP Index
  - Progression Metrics
- Monitoring



Compatibility tests - concept

- Ensure you don’t break compatibility
- Help to make the migration safe
- Short-lived tests

Compatibility test - example

```php
/**
* @group compatibility
*/
public function testProjectRepositoryLoad()
{
    global $container;

    // assert we're compatible on load without sponsorUser acl check
    $projectOld = $container['projectManager']->loadProject('project1', 1);
    $projectNew = $container['projectSecuredRepository']->load('project1', $this->sf);
    $this->assertEquals($projectOld, $projectNew);

    // assert we're compatible on with sponsorUser acl check OK
    $projectOld = $container['projectManager']->loadProject('project1', 1, 1);
    $projectNew = $container['projectSecuredRepository']->load('project1', $this->sf, $this->su);
    $this->assertEquals($projectOld, $projectNew);

    // test without sponsorUser acl check but with incorrect sponsorFrontend
    $projectOld = $container['projectManager']->loadProject('project2', 1);
    $projectNew = $container['projectSecuredRepository']->load('project2', $this->sf);
    $this->assertNull($projectOld);
    $this->assertNull($projectNew);
}
```

Indicators – CRAP index

- Change Risk Analysis and Predictions
- based on coverage & complexity
- identify parts of your code with higher risk

Progression metrics

- Keep it (very) simple
- Make product managers happy

Monitoring

- Keep an eye on performance when you replace a module
- Use anomaly detection and alerting to spot regressions

Put it all together – kill the monolith !

How to spot bad code that is easy to migrate

- Not used in too many places (easy to deploy, reduce conflicts)
- Logic will be easy to split
- Compatibility test will be fast to implement

Steps to kill old code – easy task

- Write agnostic services, with all dependencies injected
- Write compatibility tests
- Replace old code usage by your brand new service
- Remove old code and compatibility test
- For each method migrated :
  - Global complexity will decrease
  - Coverage will increase

    => CRAP (risk) index naturally goes down

Steps to kill old code – complicated task (plan A)

- Implement new agnostic services
- Write compatibility test
- Inject new services in old manager (dependencies of the method)
- Replace smoothly in sequential small tasks

Plan A - example

```php
/**
* this is just a proxy to the new system
*/
public function loadProject($id, $userId)
{
    global $container;

    $from = debug_backtrace(DEBUG_BACKTRACE_IGNORE_ARGS, 2)[0];

    $container['loggerFactory']('legacy_migration')->info(
        sprintf('legacy projectManager::loadProject called from %s on line %s', $from['file'], $from['line'])
    );

    return $container['projectSecuredRepository']->load(
        $id,
        $this->userRepository->loadUser($userId)
    );
}
```

Steps to kill old code – complicated task (plan B)

- Inject old manager in your new services
- Mock methods which are dependencies of the method to kill
- Call dependencies methods using the manager injected
- These methods will be your next targets

Plan B - example

```php
class FormRepository implements FormRepositoryInterface {

    /**
    * @var \FormManager form manager legacy service
    * to use legacy methods dependencies without breaking
    * law of demeter
    */
    protected $formManager;

    /**
    * @var Doctrine\DBAL\Connection database connection
    */
    protected $dbal;

    public function __construct(FormManager $formManager, Connection $dbal)
    {
        $this->formManager = $formManager;
        $this->dbal = $dbal;
    }
```

Keep in mind

- Small steps
  - Easier to release
  - Avoid regressions
- Every step should end in a stable state
  - All new services have to be tested
  - Your code coverage will increase naturally
- Start where it hurts !

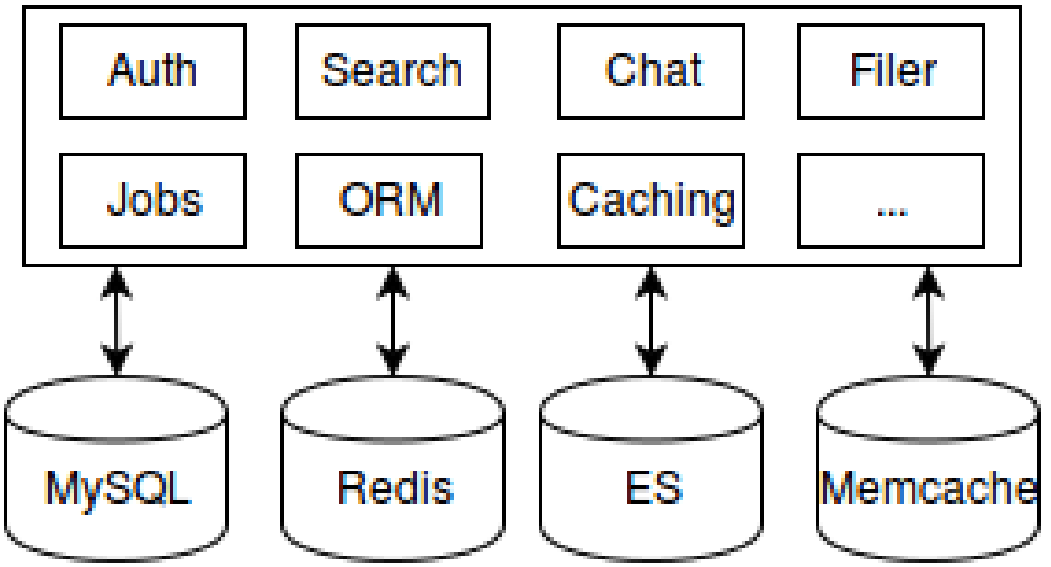
To infinity and beyond : moving to a microservice environment

- Initial purpose of SOA
- Small webservices, single responsibility
- Try to keep a consistent communication protocol in your ecosystem
- API-first architecture

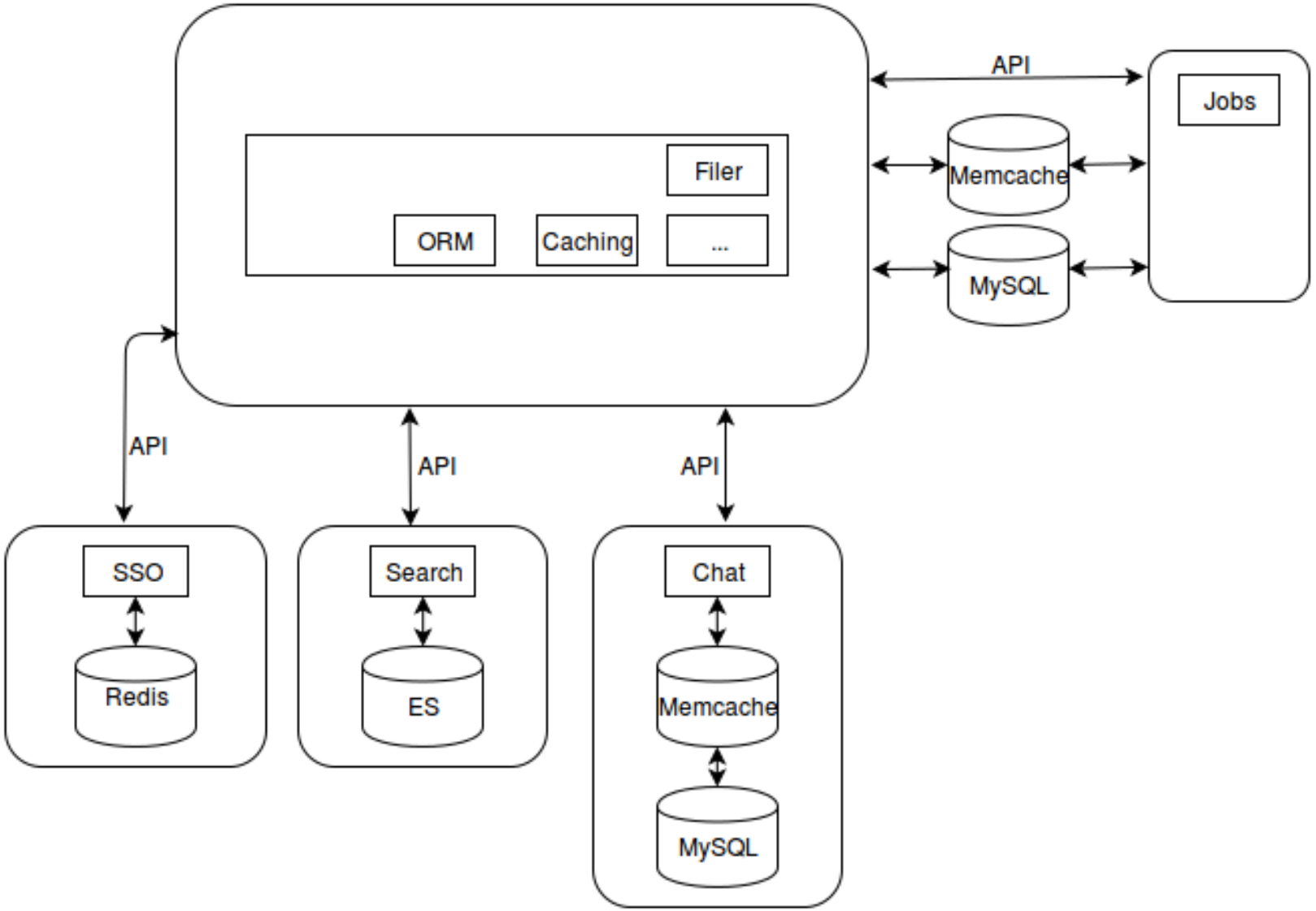
You can start within your framework !

- A service in your dependency injection container could become a simple wrapper to an external micro-service
- Good way to kill your old framework by moving parts of code to stand-alone agnostic micro-applications

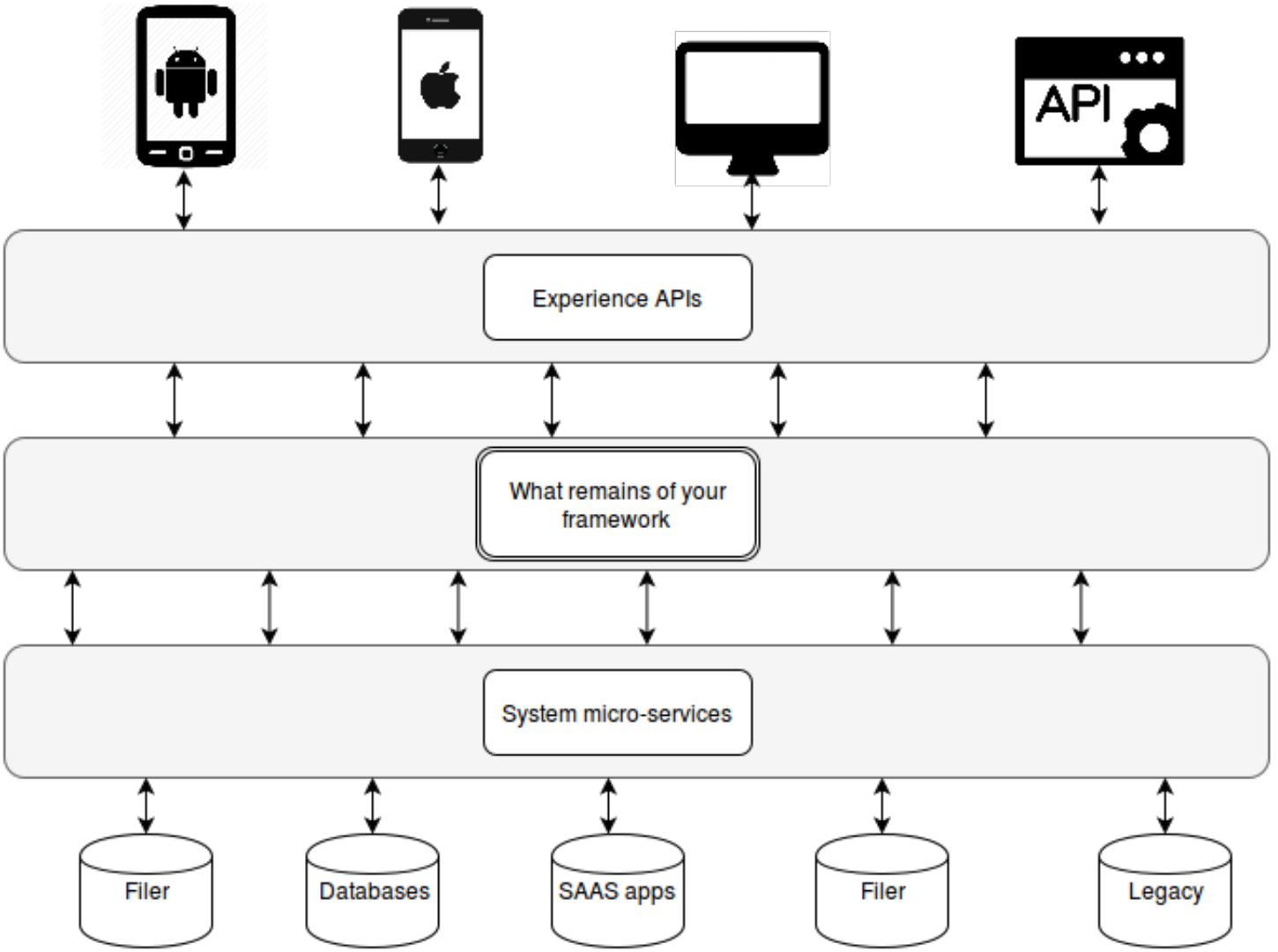
Initial State



Intermediate state



“Close to the end” state



Inter-services communication : be careful !

Take away

There is no silver bullet !

Thank you !

Are you interested in solving similar problems?

Join our team, we’re hiring!

Resources

- https://martinfowler.com/
- http://blogs.mulesoft.com/
- http://www.artima.com/weblogs/viewpost.jsp?thread=210575
- http://www.exampler.com/testing-com/writings/coverage.pdf
- http://engineering.dailymotion.com/monitor-your-application-using-pinba/

Questions ?