When To

Mock

@stuherbert

Introductions

The Holy Grail of Unit Testing

@stuherbert

```
$ git clone git@github.com:foo/bar.git
$ cd bar
$ composer install
$ phpunit
```

```
$ git clone git@github.com:foo/bar.git
$ cd bar
$ composer install
$ phpunit
```

```
$ git clone git@github.com:foo/bar.git
$ cd bar
$ composer install
$ phpunit
```

```
$ git clone git@github.com:foo/bar.git
$ cd bar
$ composer install
$ phpunit
```

Unit Tests

Should Execute Straight Out Of The Box

This Was Easy

In The Pre-Web World

- Our code wasn’t networked
- Our code used embedded data storage engines
- Our code wasn’t multi-user

- Our code wasn't networked
- Our code used embedded data storage engines
- Our code wasn't multi-user

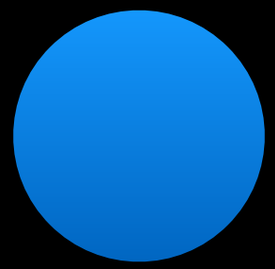
- Our code wasn't networked
- Our code used embedded data storage engines
- Our code wasn't multi-user

Our Code Was

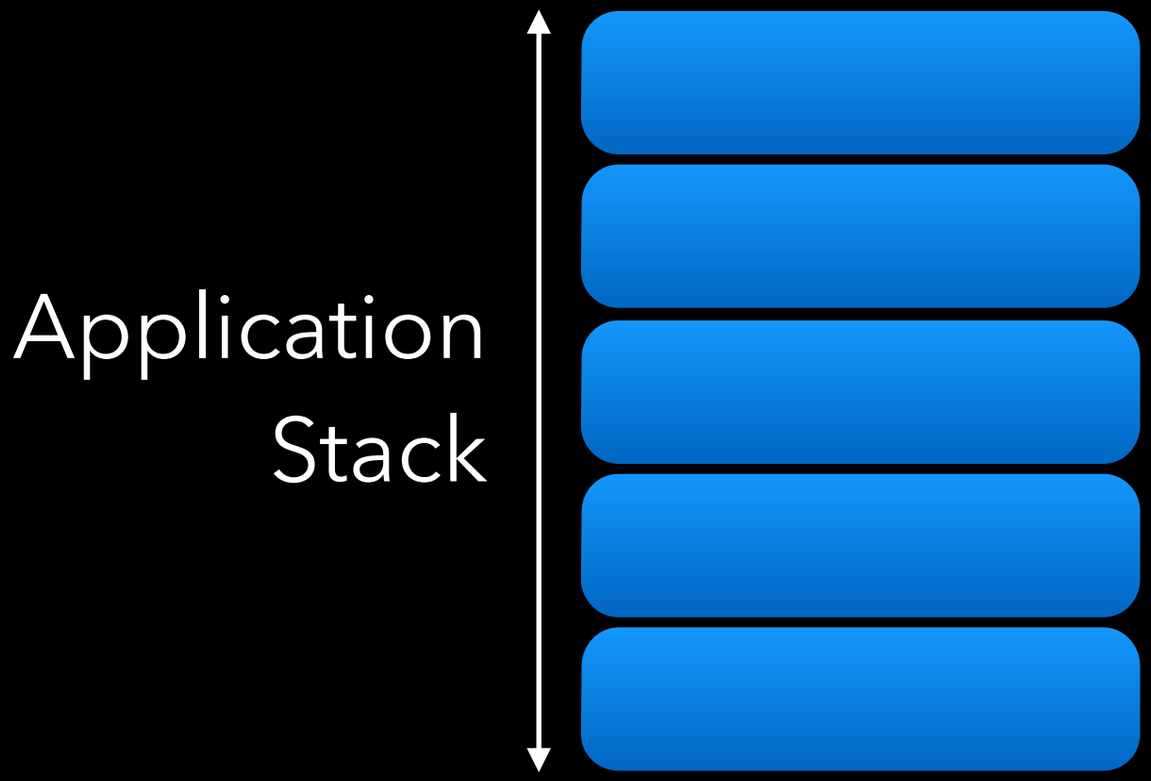
Self-Contained

Pre-Web Architecture Diagram

Pre-Web Architecture Diagram



Pre-Web Architecture Diagram



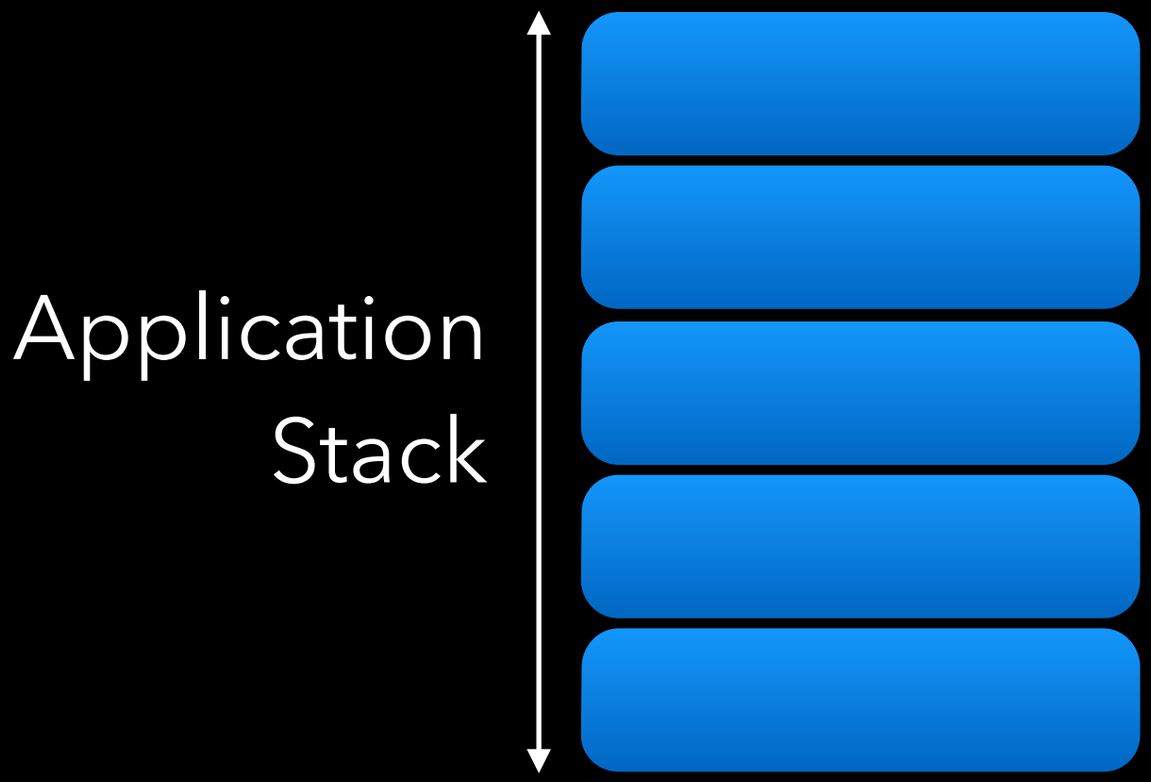
Today's Code

Talks To Things

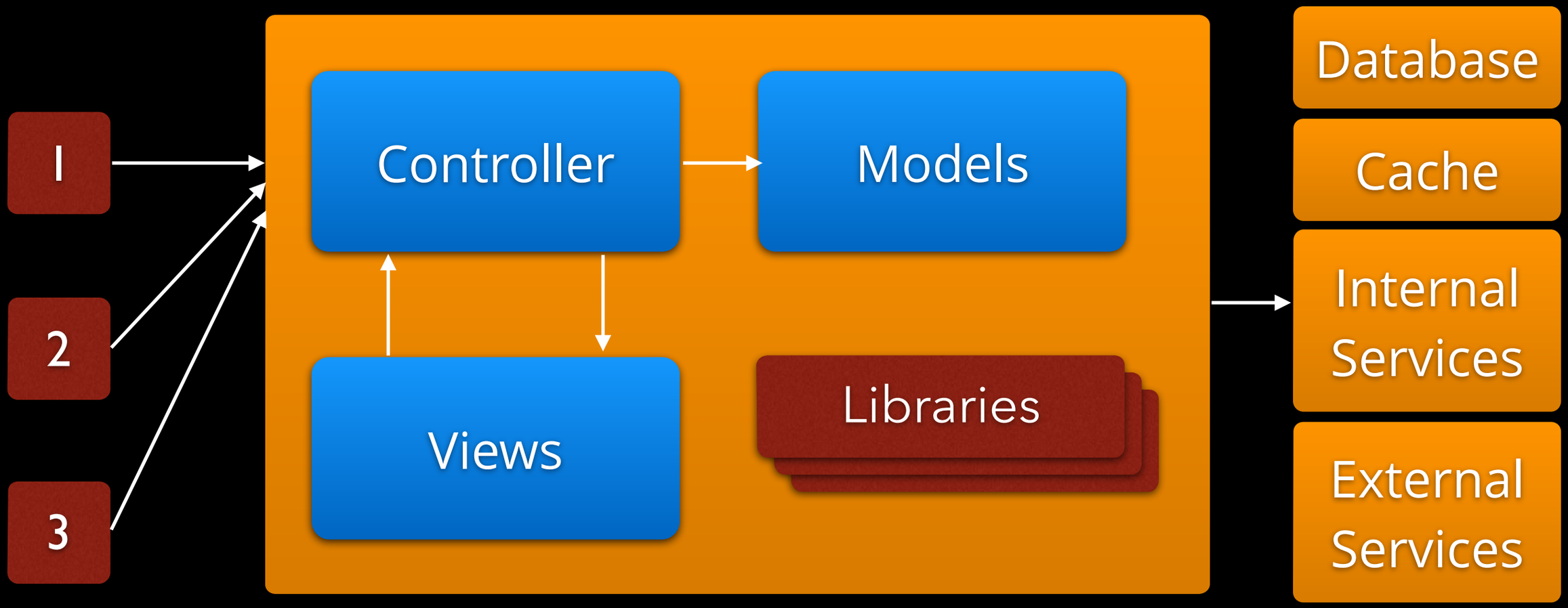
Today's Code Is Anything But

Self Contained

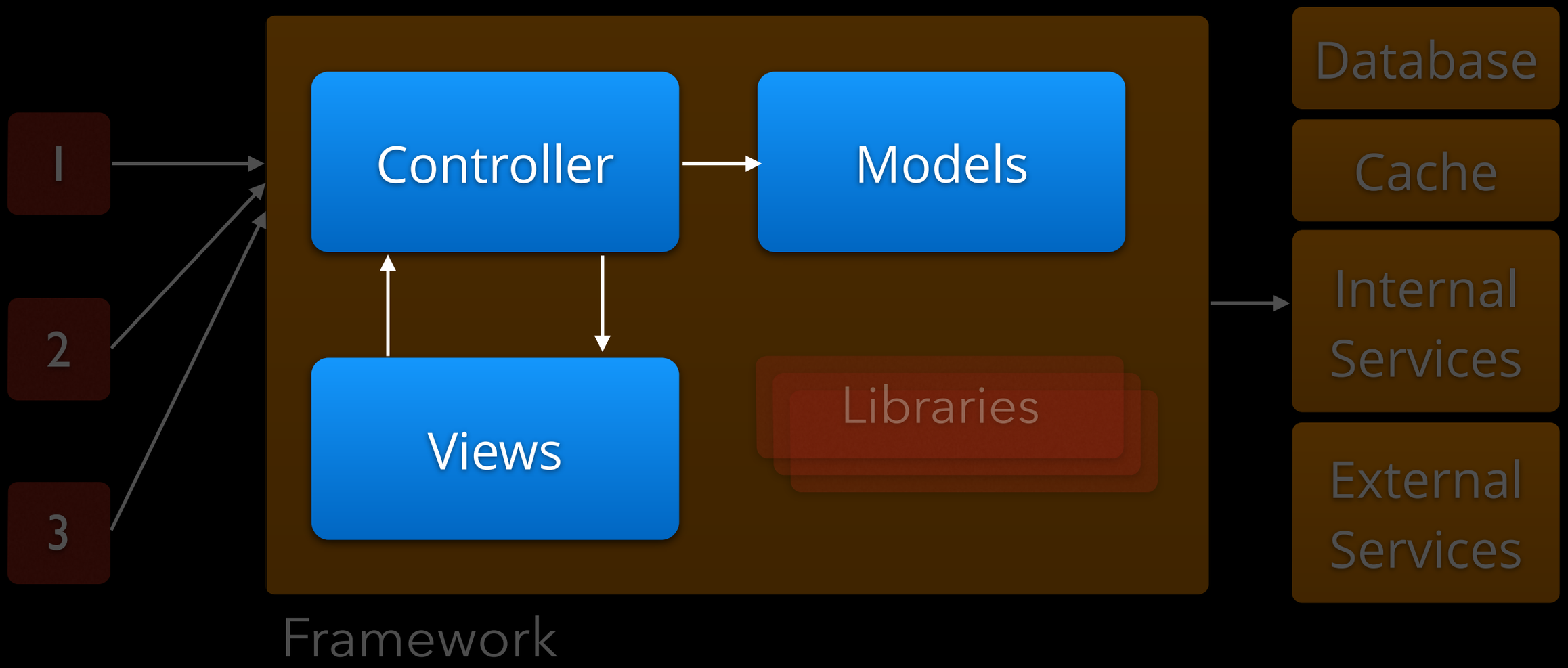
Our Code Has Evolved From This



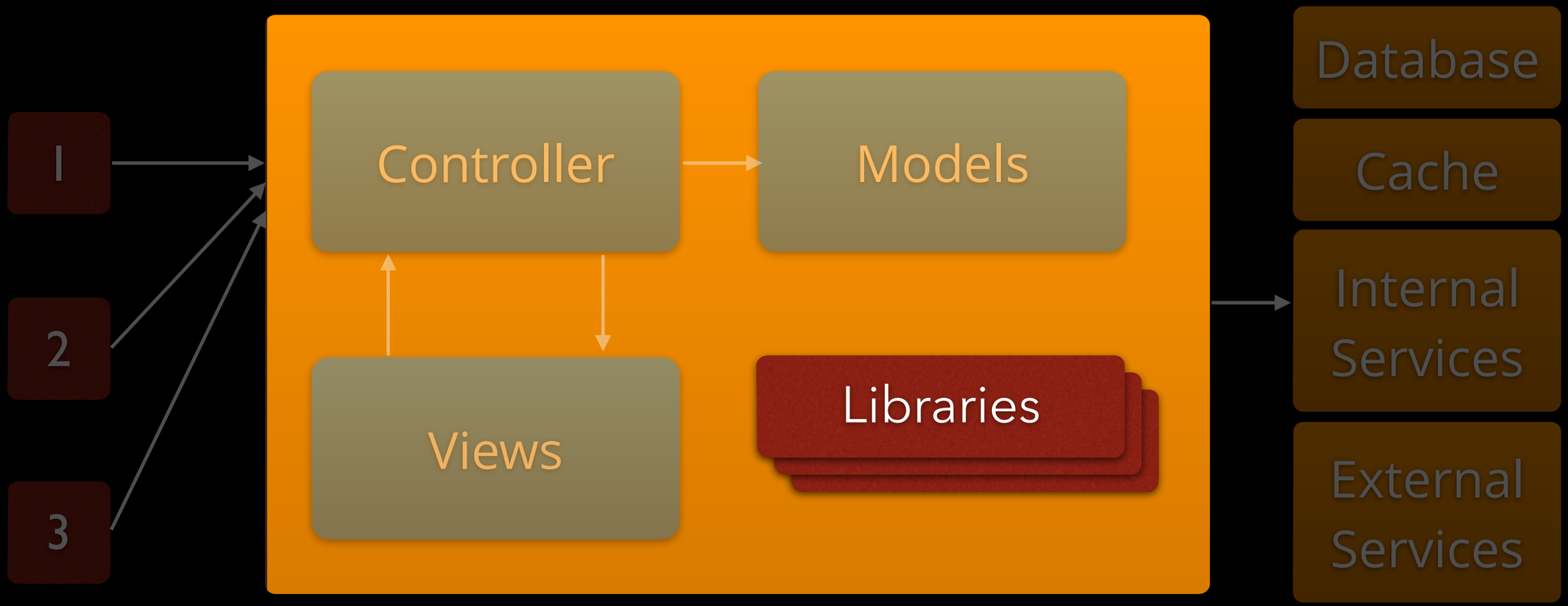
Our Code Now Looks Like This



Internally, More Complexity

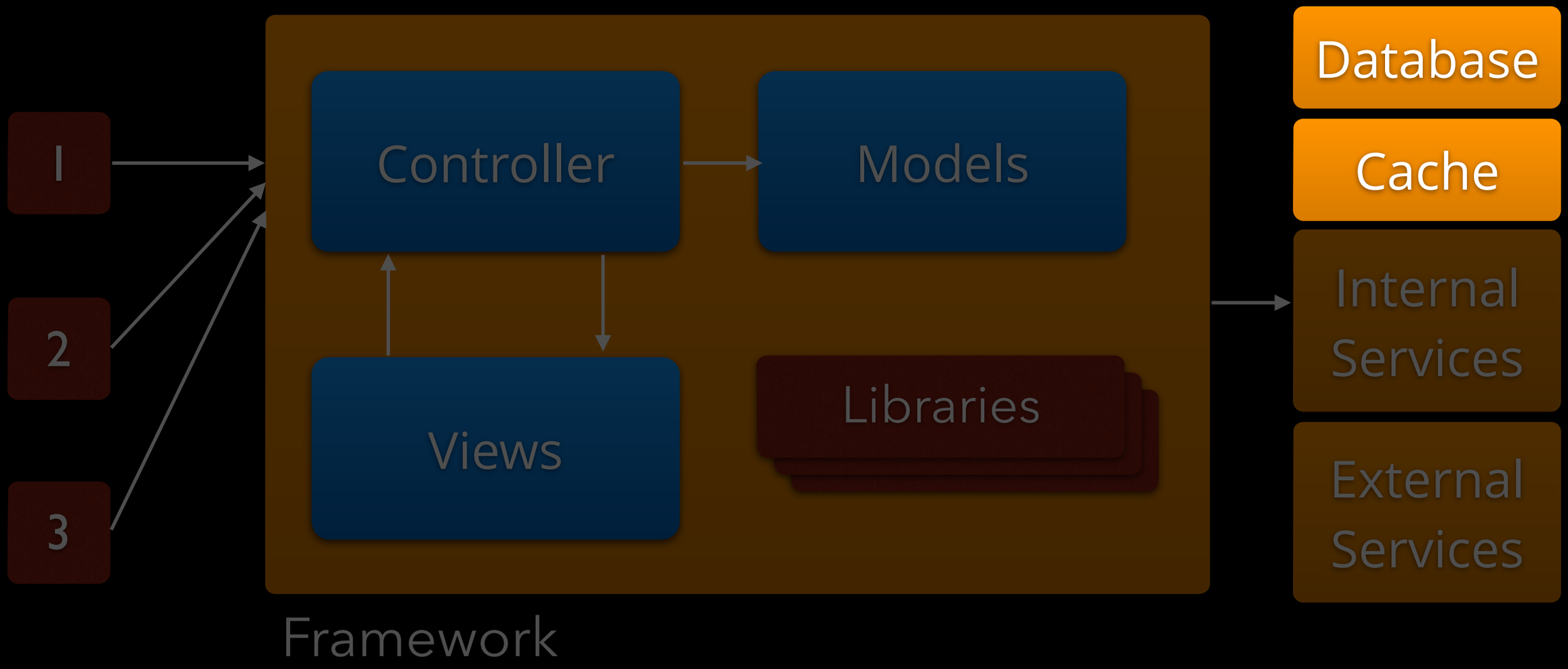


Internally, More Dependencies

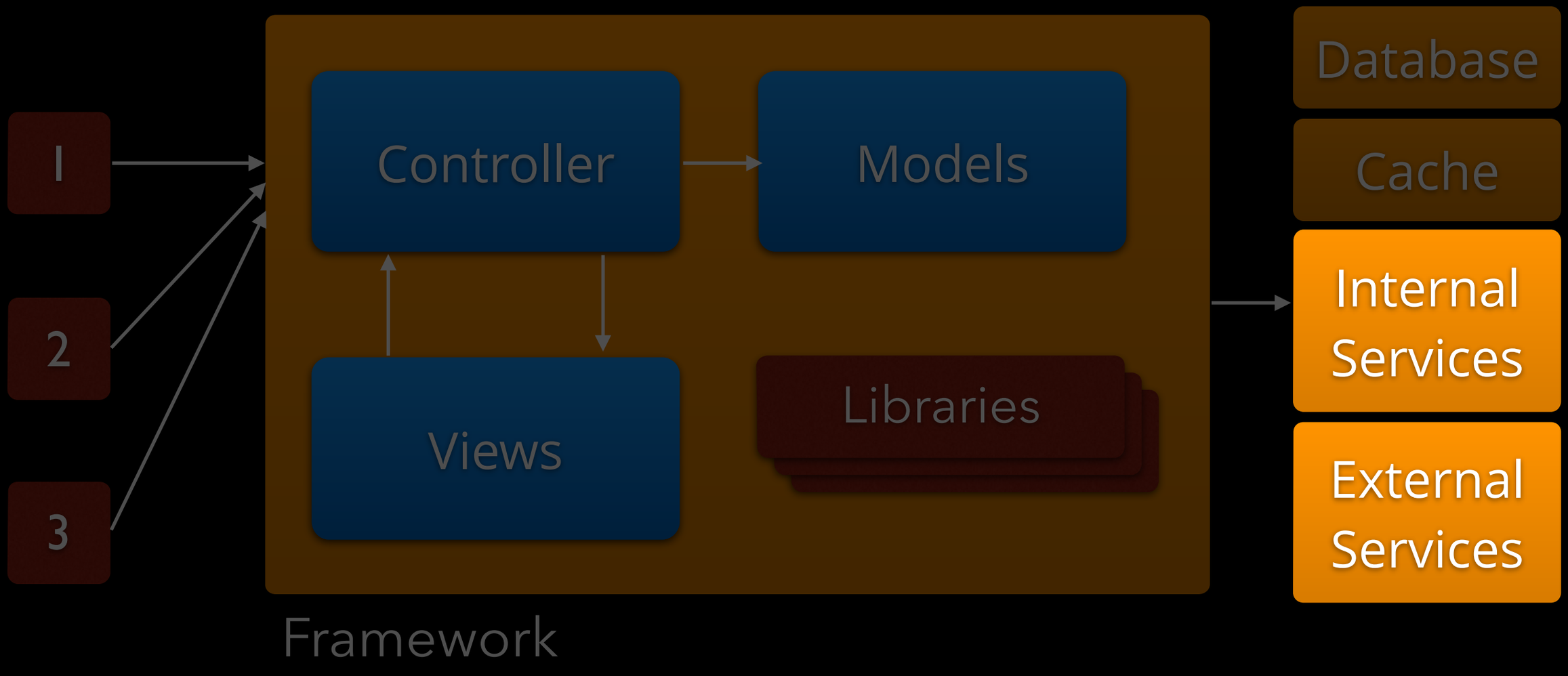


Framework

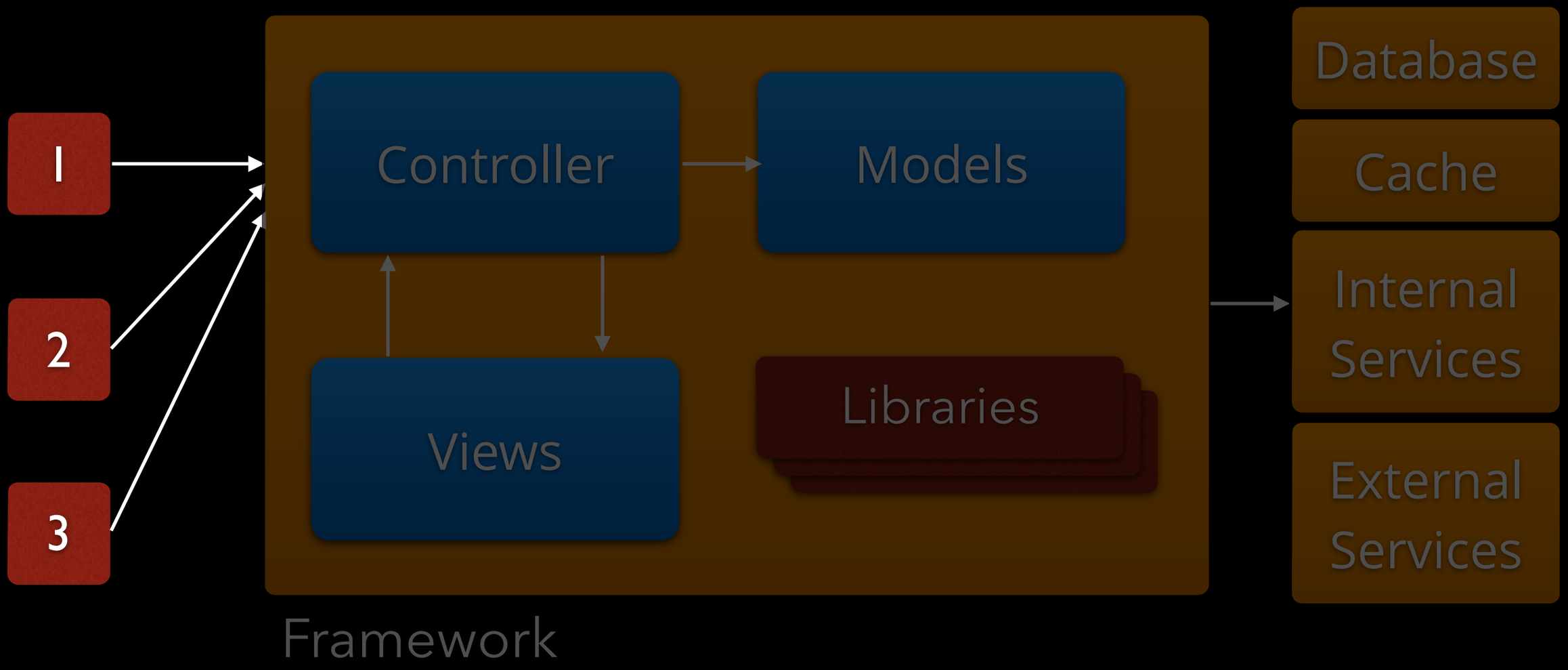
Storage Is Now A Service



Functionality Is Now A Service



Apps Are Now A Service To Users



Unit Tests

Should Execute Straight Out Of The Box

Is This Still

Good Advice?

Still Good Advice?

- Makes testing reproducible
- Ease of use by new project members
- Supports continuous integration

Still Good Advice?

- Makes testing reproducible
- Ease of use by new project members
- Supports continuous integration

Still Good Advice?

- Makes testing reproducible
- Ease of use by new project members
- Supports continuous integration

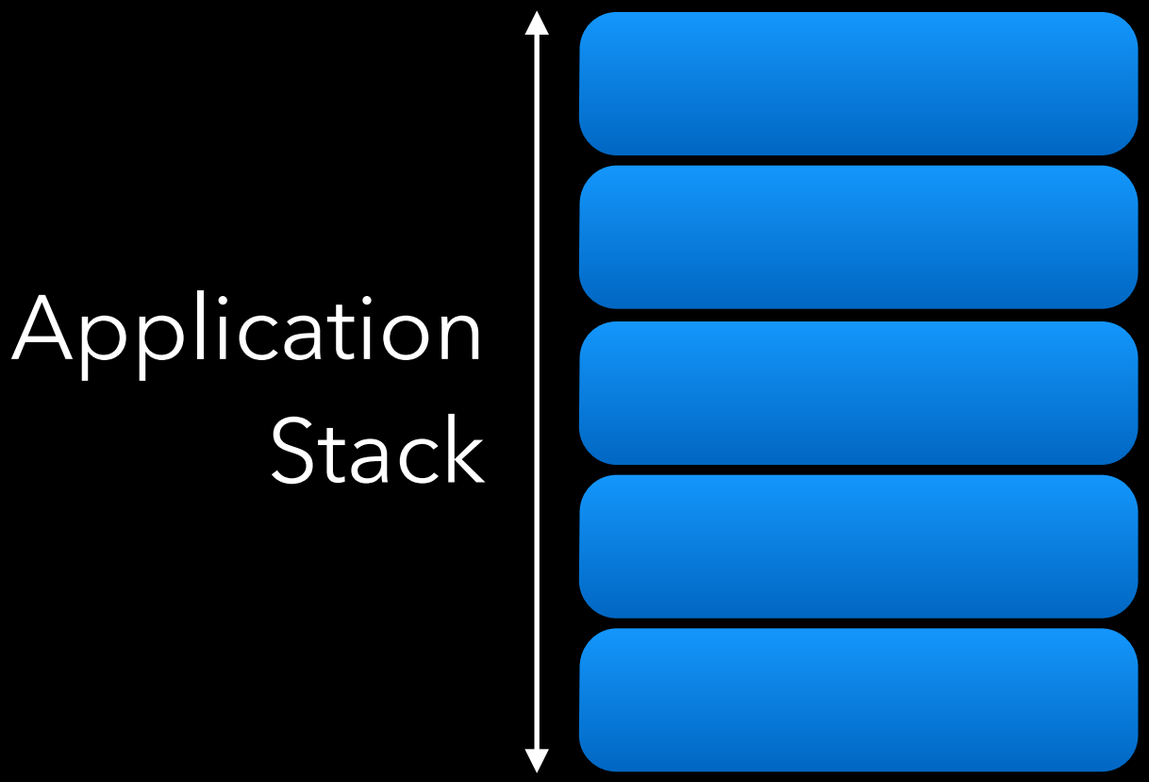
Unit Tests

Should Execute Straight Out Of The Box

Making Unit Tests Execute Out Of The Box

Now Requires Help

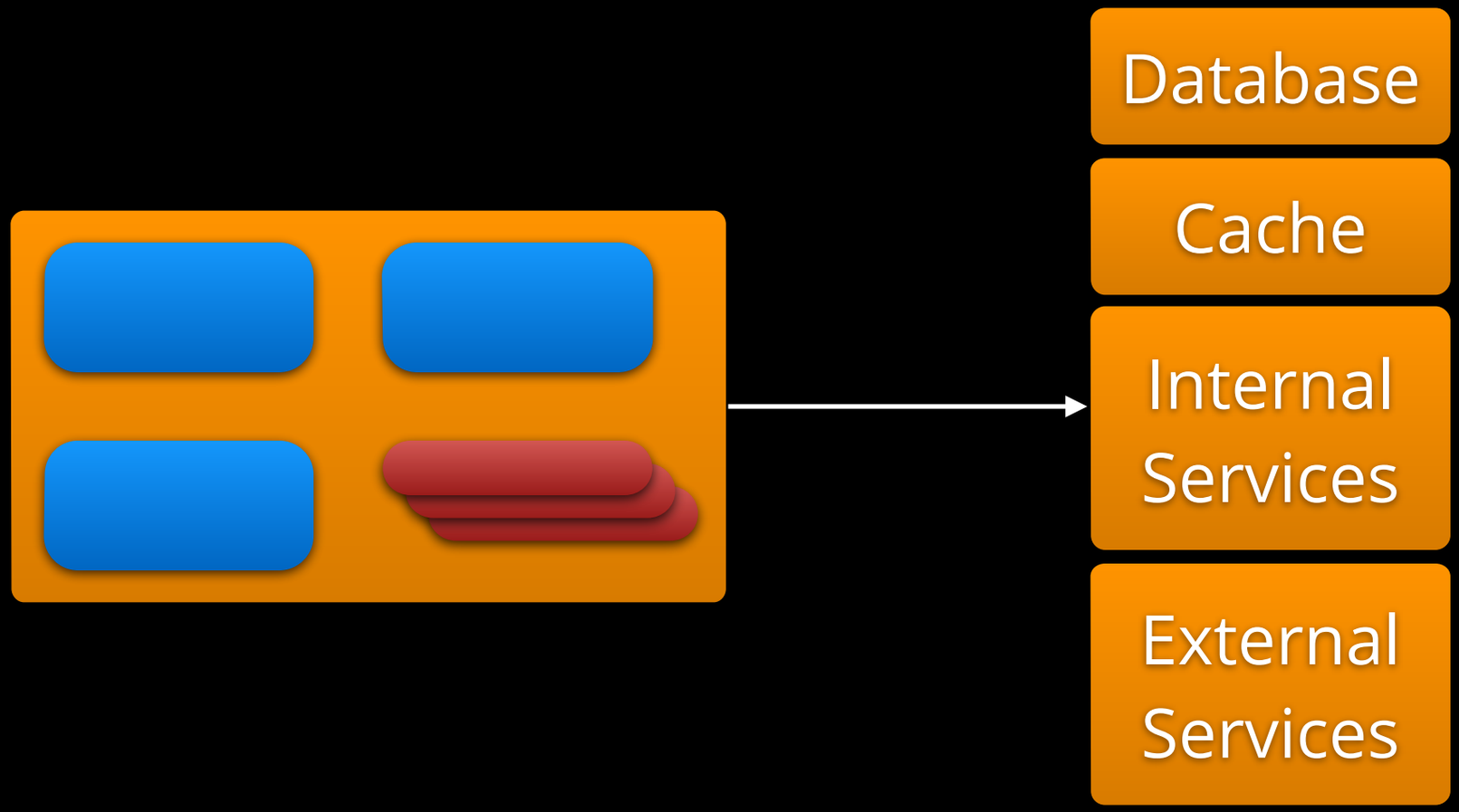
We Want To Unit Test Like This



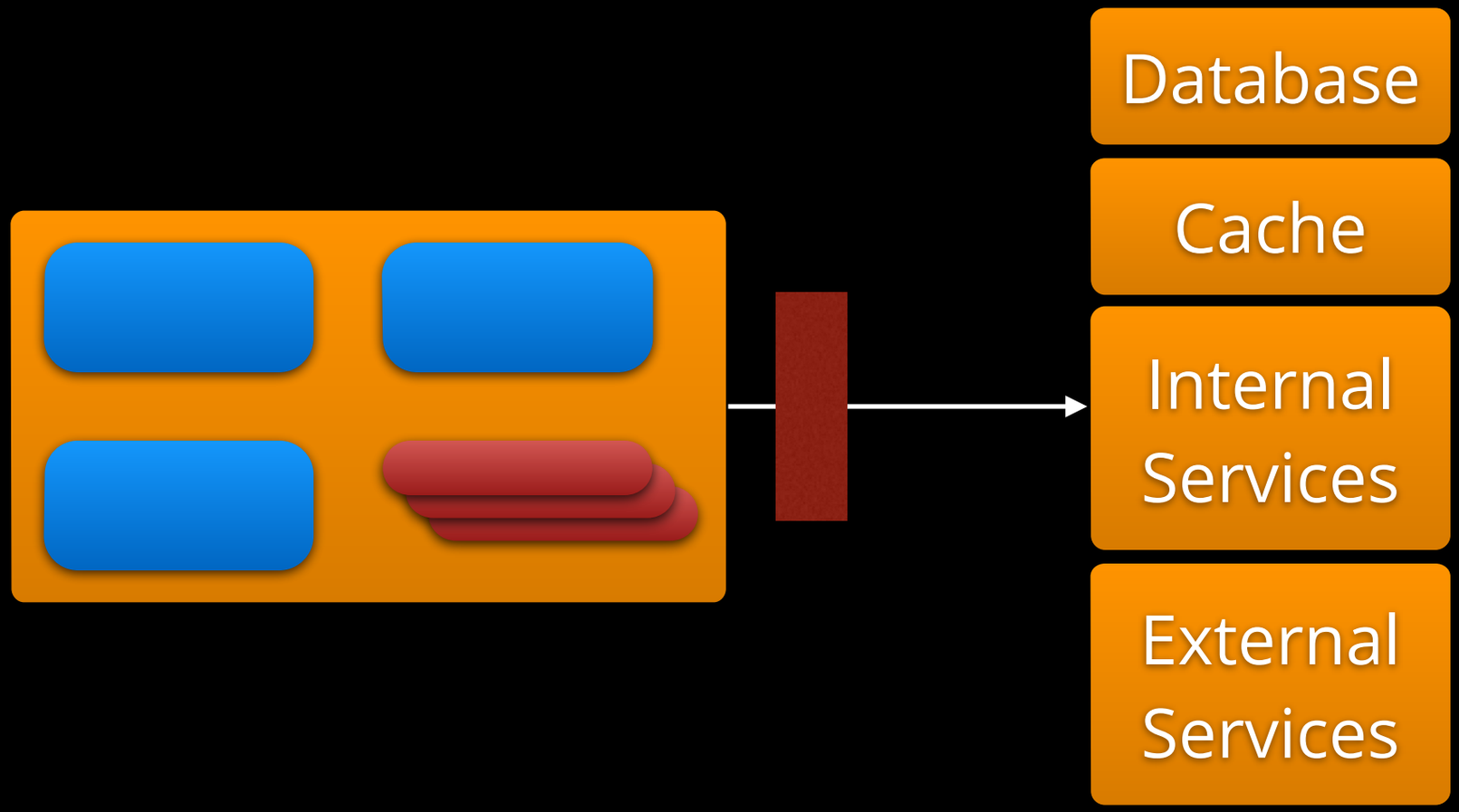
But Those Days

Are Gone

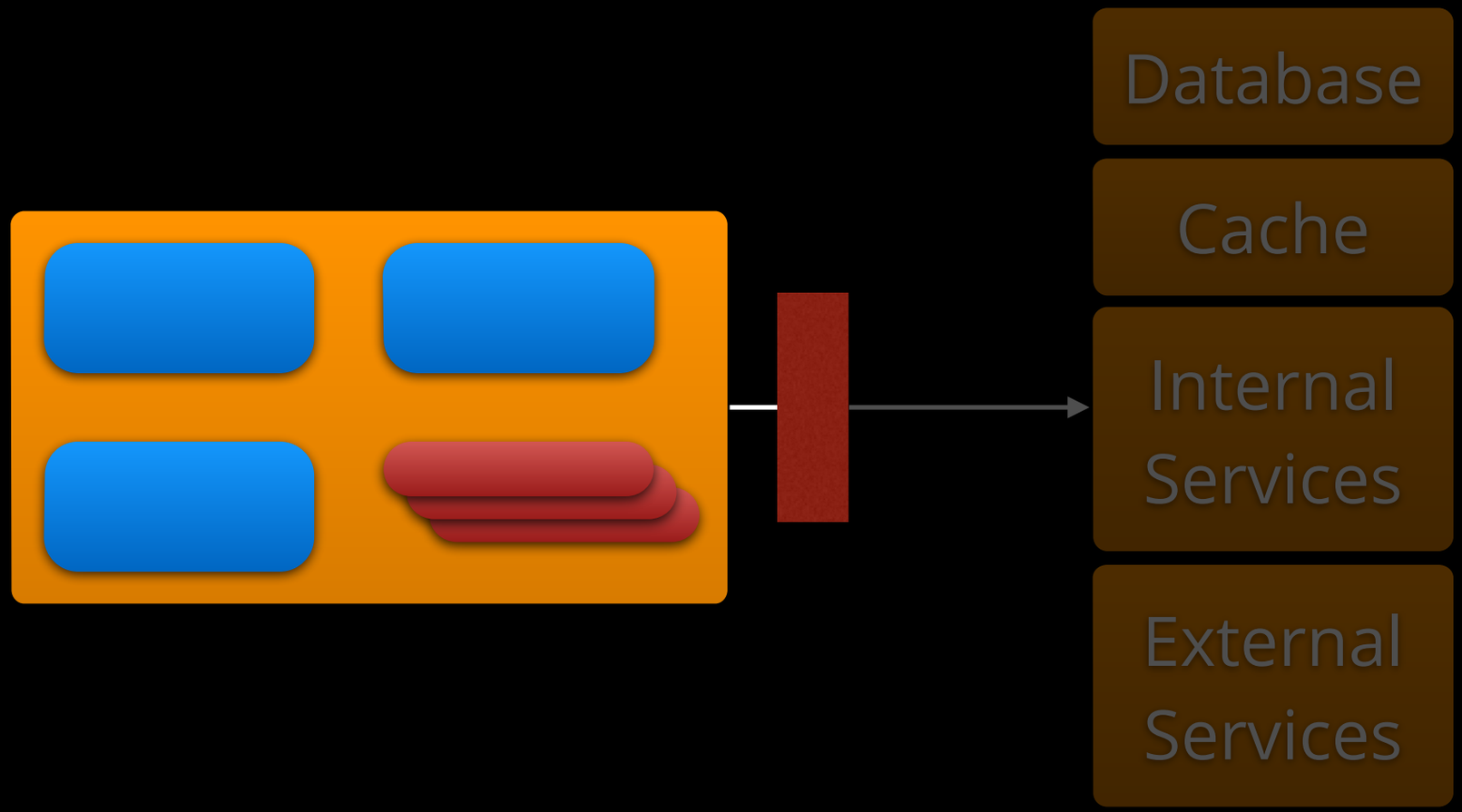
Our App Now Needs These



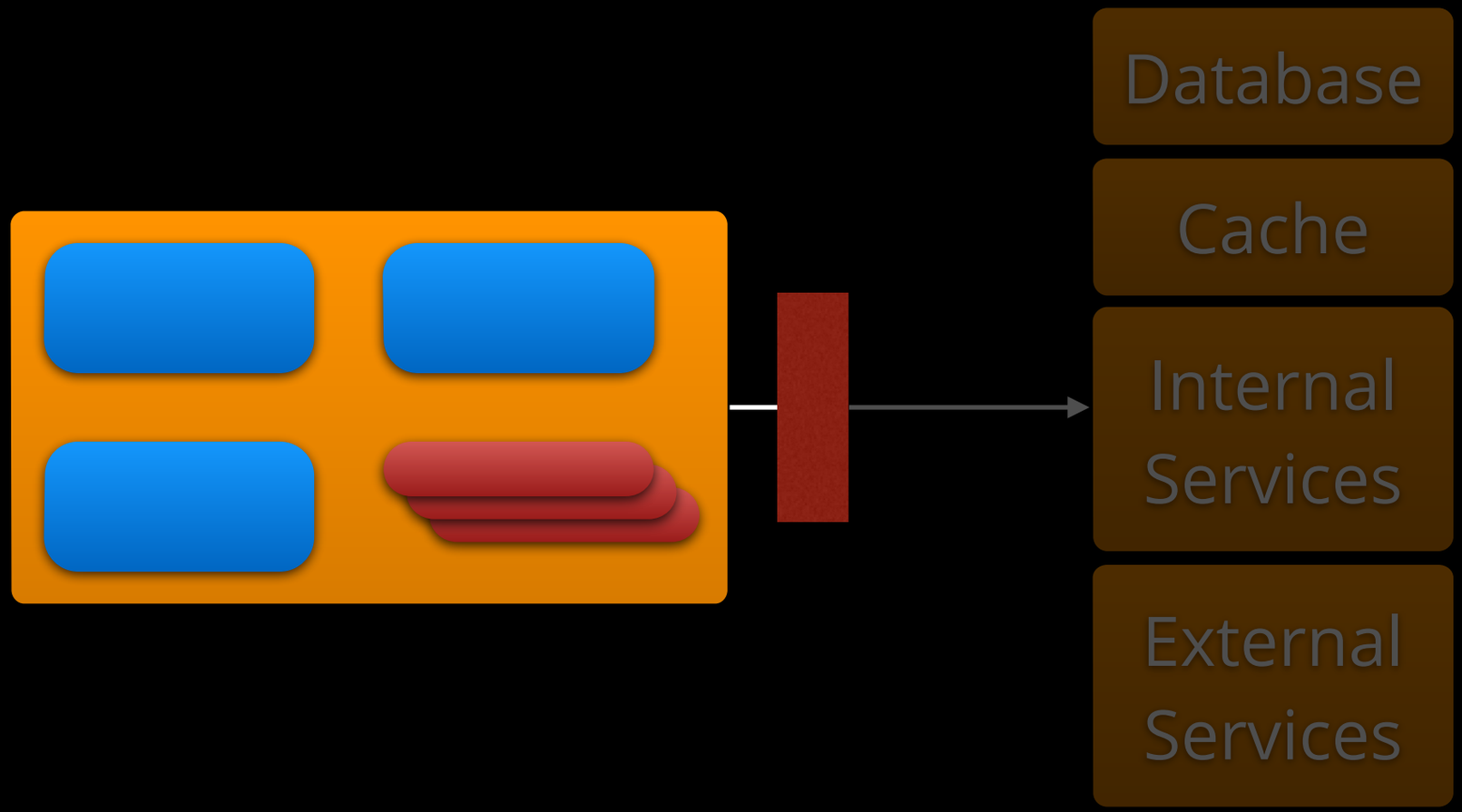
If We Mock These



We Can Unit Test Without Them



& We Can Simulate Failure Too



Three Key Questions

- Can we mock all the things?
- Should we mock all the things?
- Should we mock inside our app?

- Can we mock all the things?
- Should we mock all the things?
- Should we mock inside our app?

- Can we mock all the things?
- Should we mock all the things?
- Should we mock inside our app?

Can We Mock

All The Things?

- Given enough time and effort, yes we can

Should We Mock

All The Things?

- Who is going to build the mocks?
- How do you prove your mock behaves accurately today?
- How do you prove your mock still behaves accurately tomorrow?

- Who is going to build the mocks?
- How do you prove your mock behaves accurately today?
- How do you prove your mock still behaves accurately tomorrow?

- Who is going to build the mocks?
- How do you prove your mock behaves accurately today?
- How do you prove your mock still behaves accurately tomorrow?

- A mock can only be as good as the author's understanding of whatever is being mocked
  - Tests using rotted mocks will continue to pass, but the code will fail when shipped

- A mock can only be as good as the author's understanding of whatever is being mocked
- Tests using rotted mocks will continue to pass, but the code will fail when shipped

Test what we can,

mock what we have to

Should We Mock

Inside Our App?

What Happens Inside

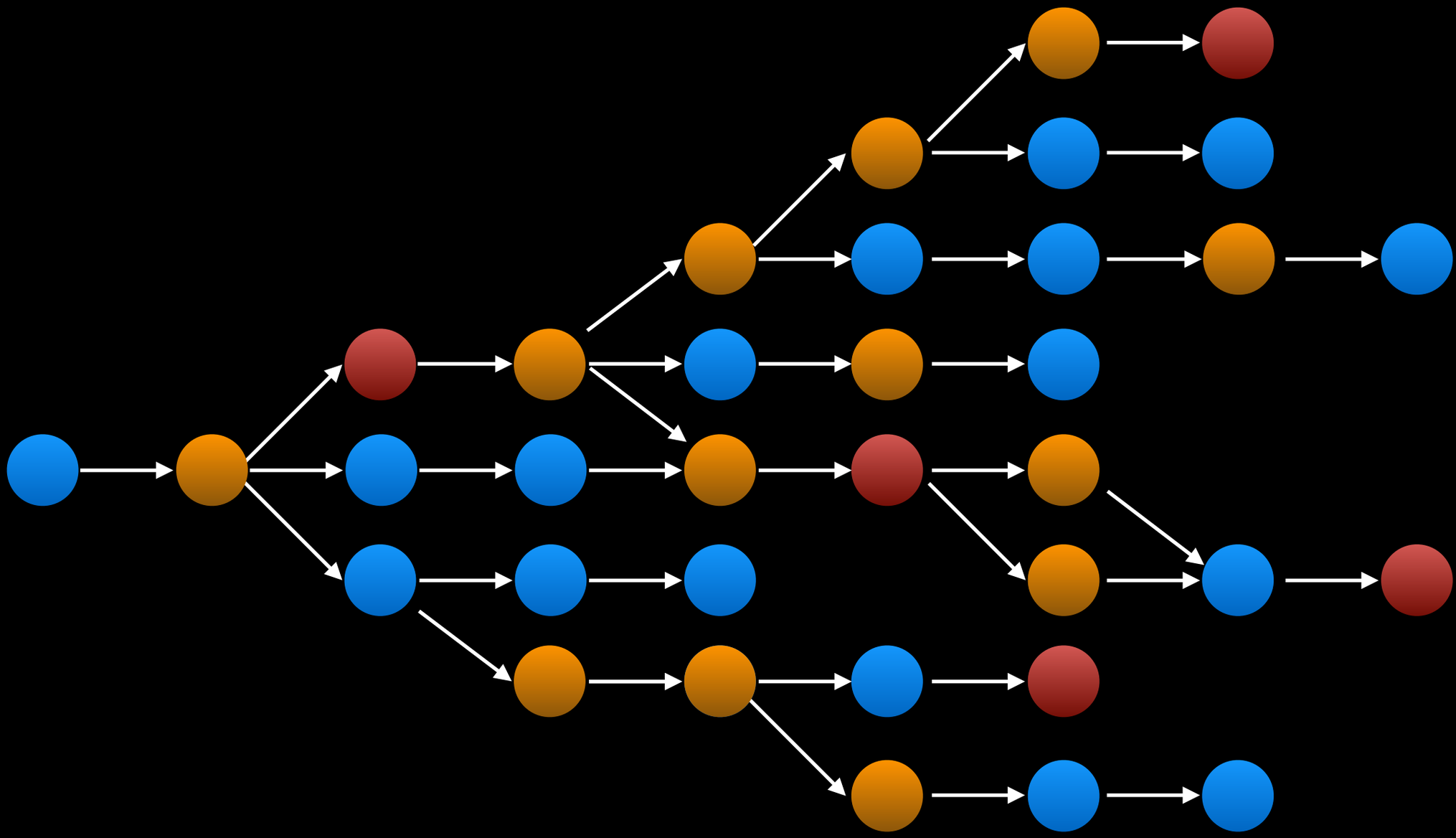
Our Code?

- computation
- branching
- input / output

- computation
- branching
- input / output

- computation
- branching
- input / output

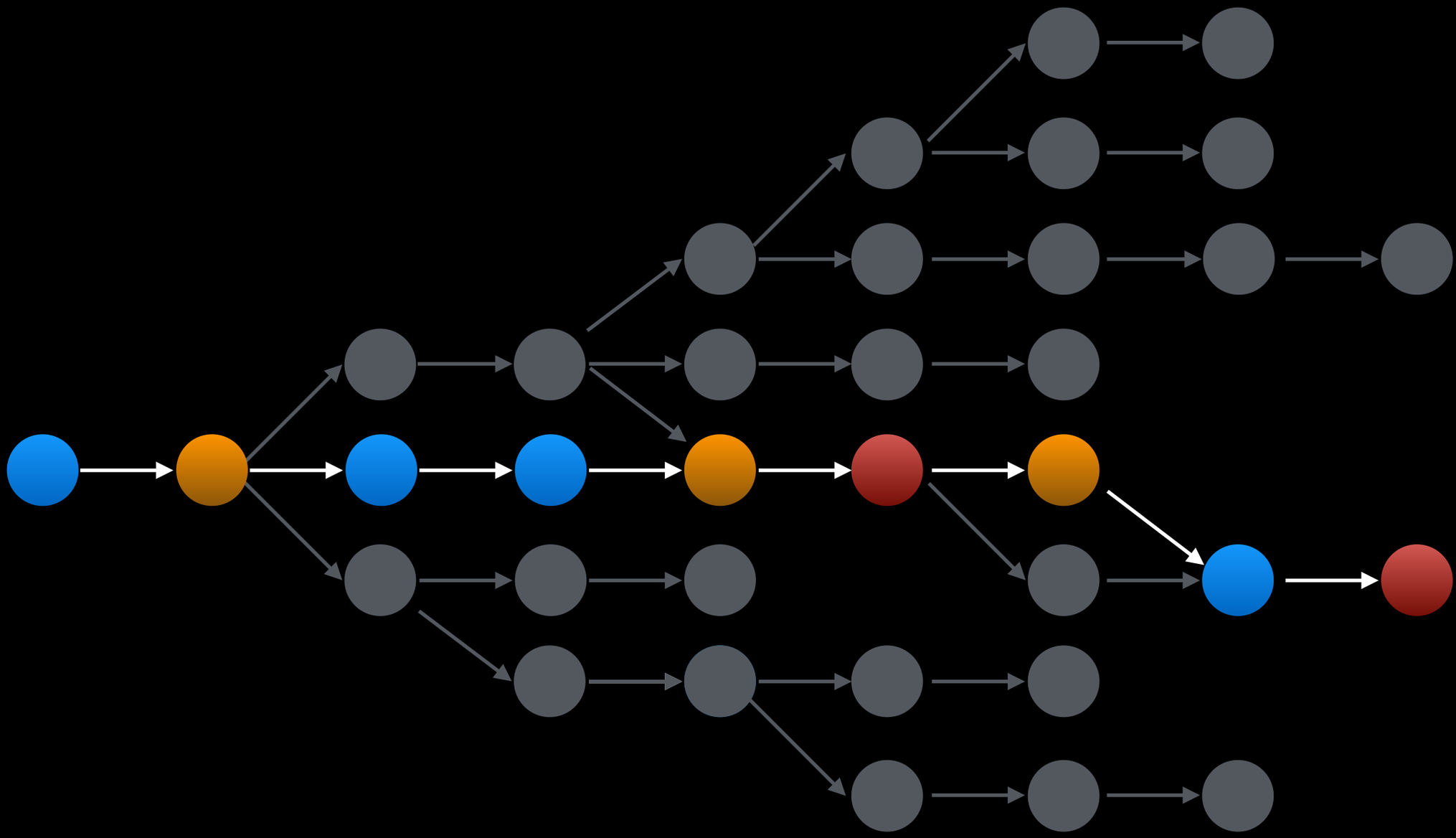
Forming Execution Paths



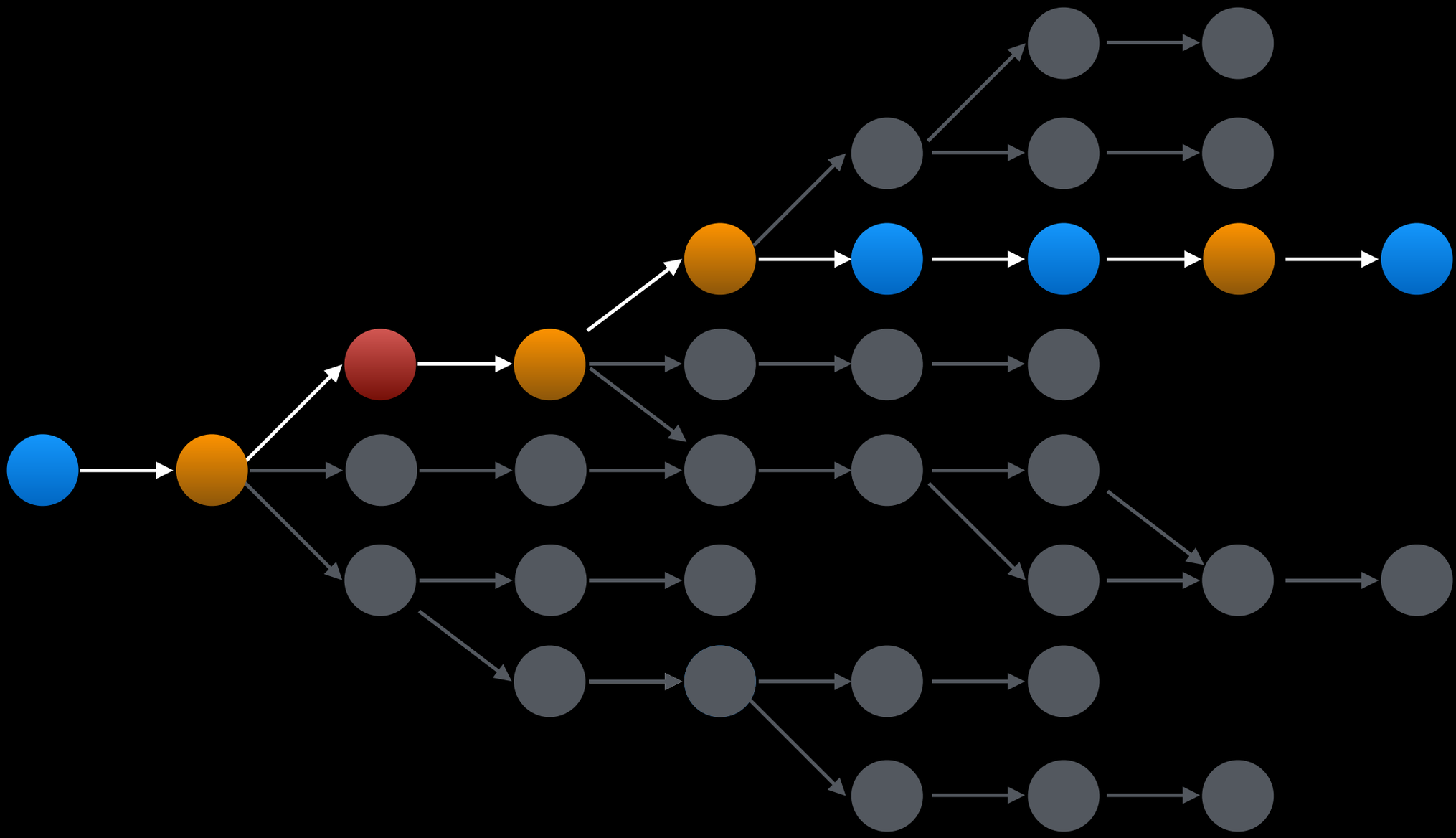
Each Unit Test Exercises

One Execution Path

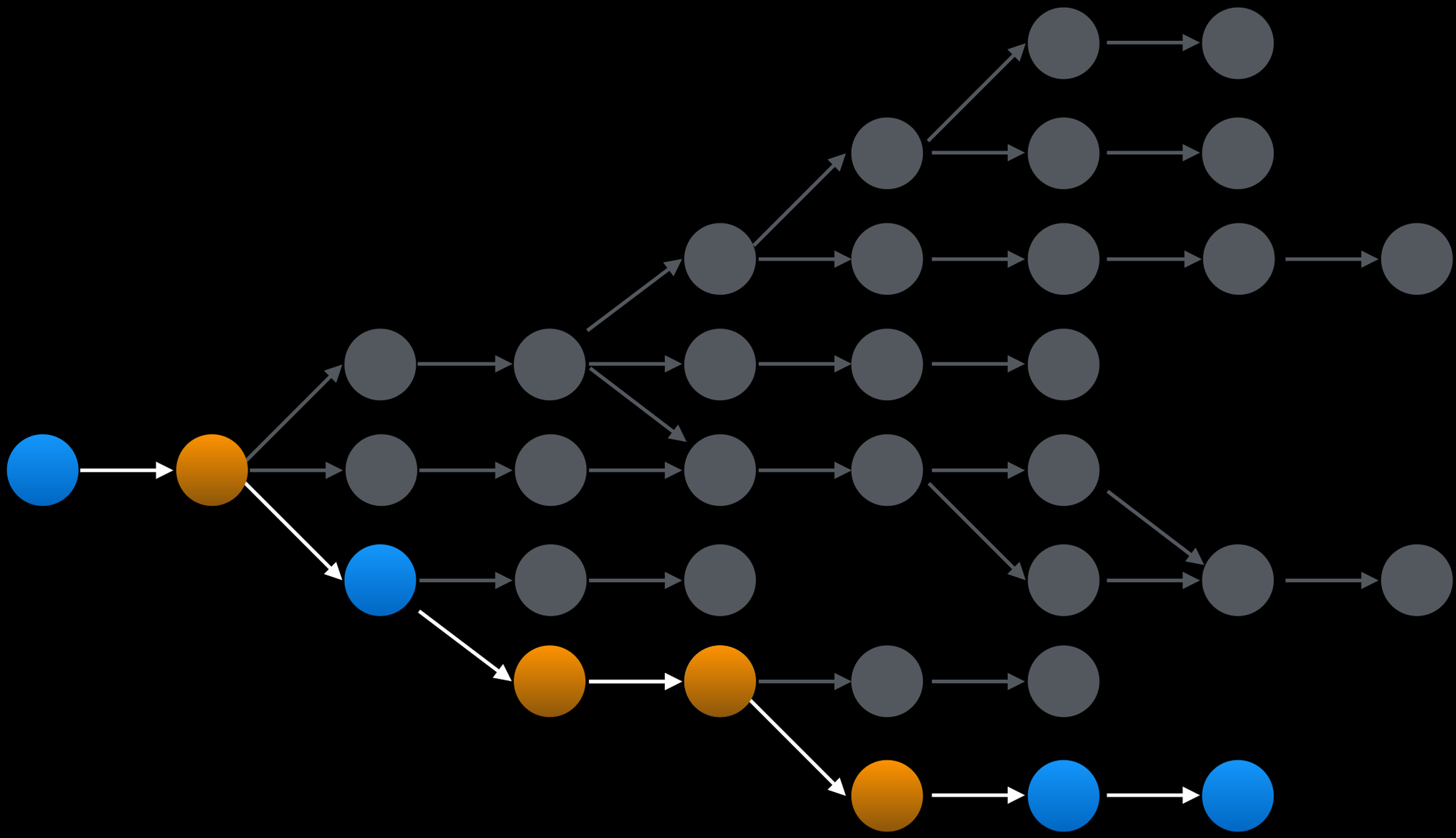
Such As This



Or This



Or This

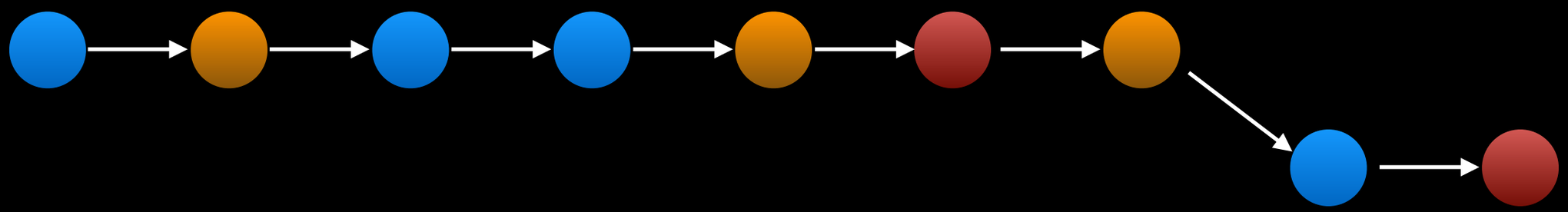


What Happens When You Mock The Code You Call?

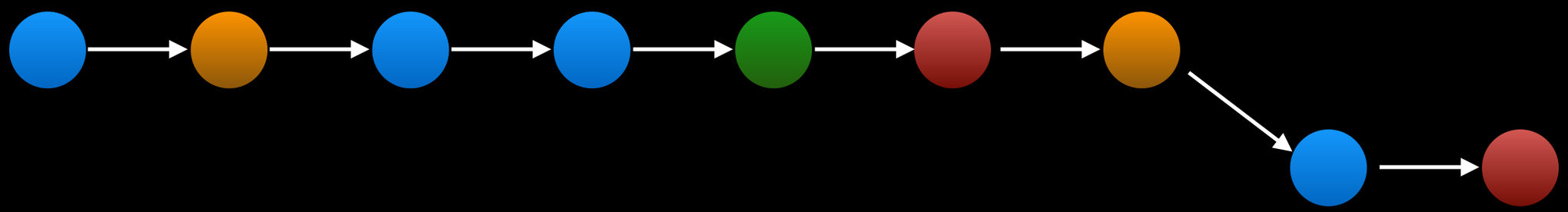
Adding Mocks

Shortens The Paths We Can Test

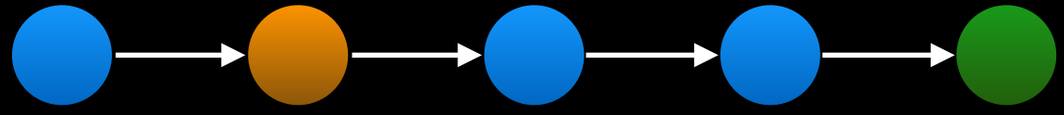
Take A Code Path



Inject A Mock



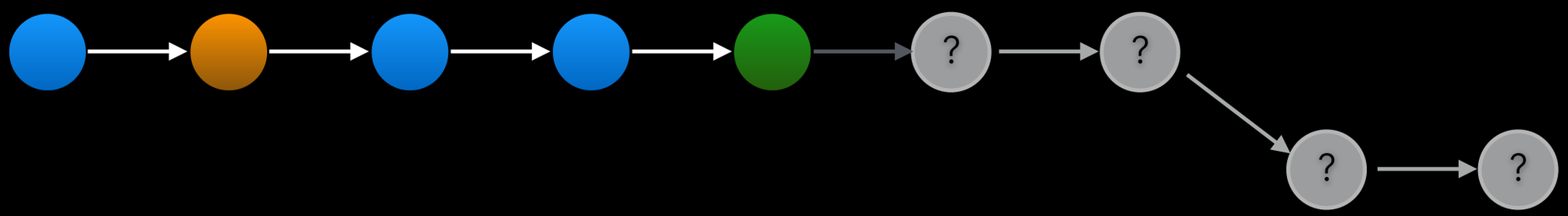
Shorter Code Path



But What About The Code

Behind The Mock?

Unreachable Code



Mocks Inside Your App Create

Unreachable Code

Unreachable Code

Is Untestable Code

What Is The Risk

From The Code That You Can't Test?

Key Questions

- How do you prove your mock behaves accurately today?
- How do you prove your mock still behaves accurately tomorrow?

- How do you prove your mock behaves accurately today?
- How do you prove your mock still behaves accurately tomorrow?

- A mock can only be as good as the author's understanding of whatever is being mocked
- Tests using rotted mocks will continue to pass, but the code will fail when shipped

Mocks Are Sometimes

The Best Approach

Why?

Not Every Code Path

Is Reachable

Testing For Failure

- How do you get bad responses from the code you call?
- How do you trigger your error handling?

Mocks Are A Great Way

To Test For Failure

In Summary

Test what we can,

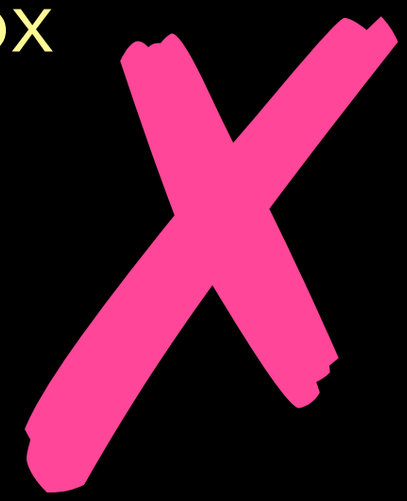
mock what we have to

Unit Tests

Should Execute Straight Out Of The Box

Unit Tests

Should Execute Straight Out Of The Box



Unit Tests

Should Execute Straight Out Of The Box
As Long As Test Accuracy Is Not Compromised

Unit Tests

Should Be One Of Several Layers
In Your Test Strategy

Use Other Layers

To Test For Things You Can't Prove

Because Of Your Mocks

- Storyplayer for factory acceptance testing
- Behat / BDD for product acceptance testing

Thank You

Any Questions?