ThoughtWorks®

QUALITY: EVERYONE'S RESPONSIBILITY

Tim Cochran and Dan Lockman

bit.ly/tw-philly

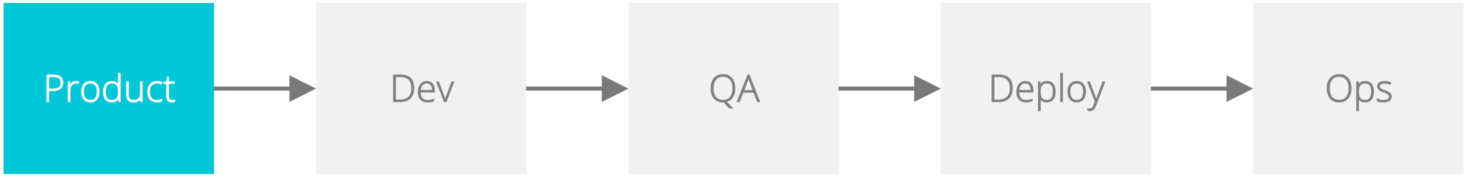
AGENDA

Quality: Everyone’s Responsibility

Common problems with Acceptance Testing

Designing for Testability

WHAT GOES WRONG



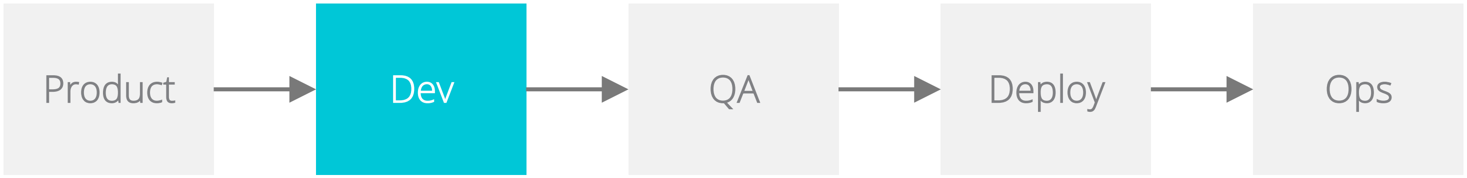
Product requirements are written in unclear or subjective language

Feature decisions are done in isolation

Designs can be over specified

Not enough user research

WHAT GOES WRONG



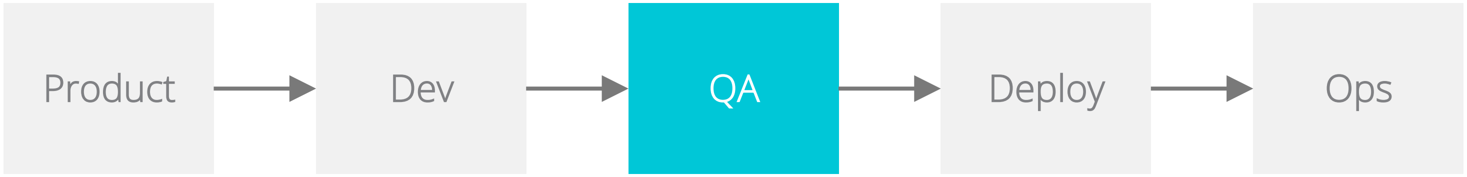
Developers don’t think about E2E regression tests

Testing coverage is not exposed to QA

There is no conversation about the right amount of testing for the product as a whole

QA don’t have access or the skills to contribute to the dev test suite

WHAT GOES WRONG



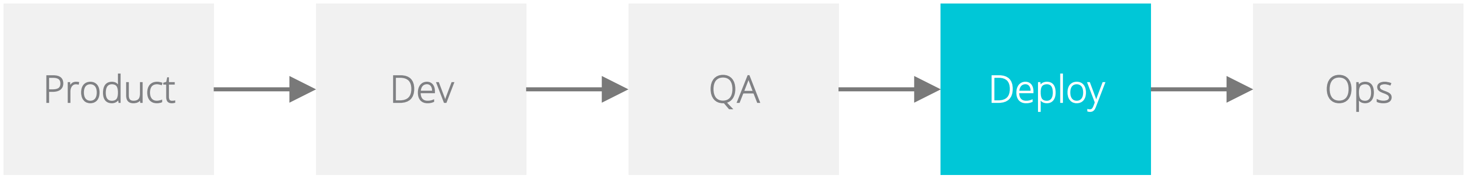
Too many end to end tests, too slow, too brittle

Done in cycles rather than continuously

Little communication, done in isolation relying on documentation

All tests are black box

WHAT GOES WRONG

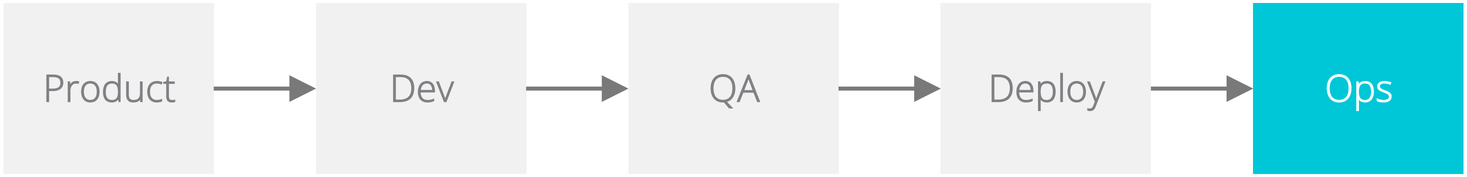


Differences in environments

Automated testing is not performed in integrated environments

Not enough automation means feedback cycle can be slow

WHAT GOES WRONG



Bugs are handled without feedback to QA or Dev

Monitoring the system but not the users' experience

Operations staff are not informed to the goals of the application

Helpdesk staff do not have enough information to help or train users

HOW DO WE FIX IT?

Cross functional team collaboration

Talk to each other

Work together

Inform each other

BARRIERS

Geographical

Language

Systems

Culture

Skills

Organisational

BREAKING THE BARRIERS

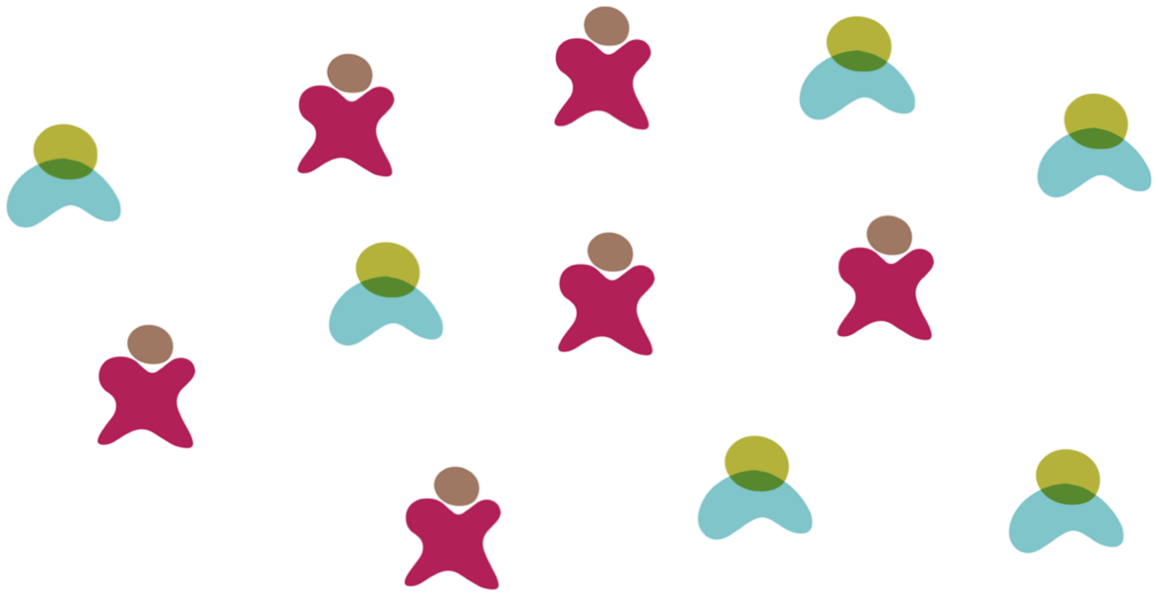
Devs

will be better at developing if they understand the QA goal

QAs

will be better at testing if they understand the application design and code

BREAKING THE BARRIERS

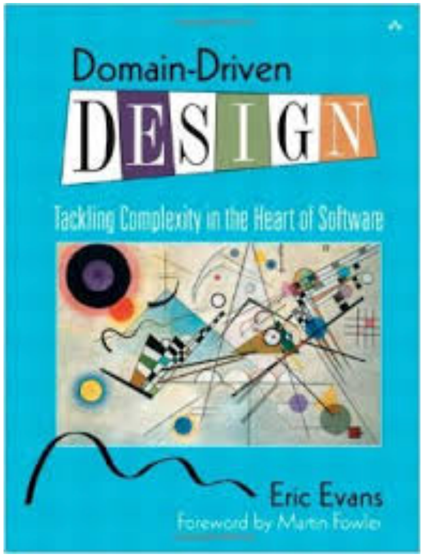


REMOVING AMBIGUITY

Domain Driven Design.

Define a glossary of Domain terms

Requirements, Tests and Objects should be using the domain terminology


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

REMOVING AMBIGUITY

Back up requirements with a concrete examples

Form must validate the email

Given a user is creating a helpdesk ticket

When the user enters the email “tim@tw”

Then the error “Please enter a valid email is displayed”

Specification by example and BDD frameworks can help with this (but be careful).

QA - Test Suite

A test suite should be -

- Easy to maintain
- Consistently Repeatable
- Valuable
- Quick

Antipattern - the QA team decides to adopt automated testing in isolation, they test every requirement via an automated black box test, this results in a long running, flaky test, that is ultimately not valuable.

AN APPROACH

1. QA is role that can be rotated, a developer can play it.
2. The Developer is expected to add regression tests for their feature/story
3. As part of story signoff, a QA will review the tests, can reject if there is not enough testing
4. Testing infrastructure and tooling is built by QAs and Developers pairing
5. Exploratory testing still done by QA, add edge cases into suite

WHAT LEVEL OF QUALITY?

Doing QA and automated testing is hard work!

Building the product right

VS

Building the right product

Antipattern - having a very rigorous QA cycle, amazing test suite, but having no fallback if there is a bug in production.

HOW TO AVOID QA?

Simple design

The right amount of QA for the domain - *building a helpdesk is a lot simpler than a trading platform*

The business can test if you give them the tools

Avoid differences in environments

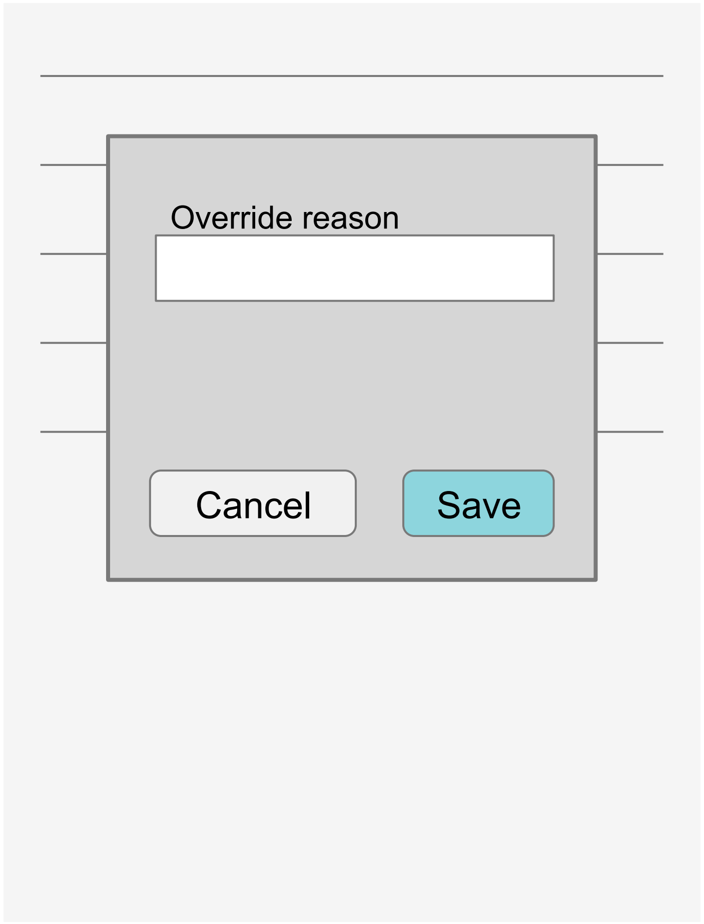
Build a test suite that runs in all envs

Retrospect

ThoughtWorks®

COMMON PROBLEMS WITH ACCEPTANCE TESTING

AN EXAMPLE



Behaviour:

Single Page application

- User adds new override
- Modal display, enters an override reason
- Clicks Save
- Person screen reloads listing the new override

PAGE OBJECTS

Models the page behaviour, rather exposing HTML elements.

Reduces the amount of duplicated code, if the UI changes, the fix need only be applied in one place.

```
class ModalPO():

        def save_override():
                driver.find_element_by_css_selector("input[type=submit]").click()

        def enter_reason(reason):
                element = driver.find_element_by_id("reason")
                element.send_keys(reason)

class PersonPO():

        def override_reasons():
                return [el.text for el in driver.find_element_by_css("li.reason")]
```

AN EXAMPLE

Test Code:

```
modalPO.enter_reason("a reason")
modalPO.save_override()

assertEqual("a reason", personPO.reasons())
```

Test Result:
Fails

AN EXAMPLE

Test Code:

```
modalPO.enter_reason("a reason")
modalPO.save_override()

time.sleep(0.2)

assertEqual("a reason", personPO.reasons())
```

Test Result:
Passes locally
Fails on CI

AN EXAMPLE

Test Code:

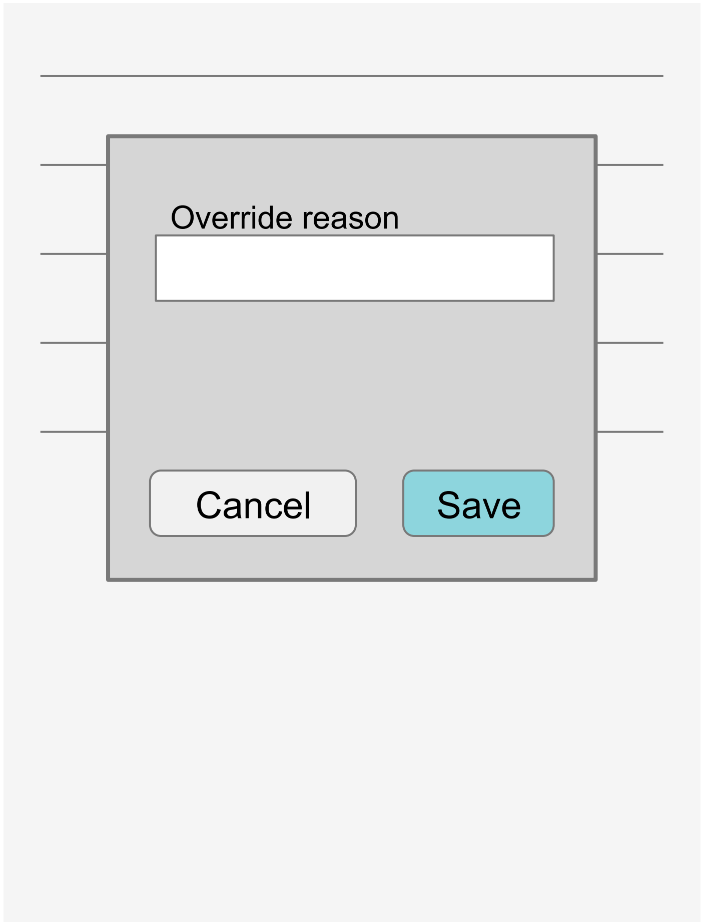
```
modalPO.enter_reason("a reason")
modalPO.save_override()

time.sleep(5)

assertEqual("a reason", personPO.reasons())
```

Test Result:
Passes locally
Passes on CI

AN EXAMPLE



Behaviour:

Single Page application

- User adds new override
- Modal display, enters an override reason
- Clicks Save
- Person screen reloads listing the new override

AN EXAMPLE

Test Code:

```
modalPO.enter_reason("a reason")
modalPO.save_override()

wait = WebDriverWait(driver, 10)
element = wait.until(EC.presence_of_all_elements_located((By.CSS,'li.reason')))

assertEqual("a reason", personPO.reasons())
```

Test Result:
Passes locally
Passes on CI

Page Object

```
class ModalPO():

            def save_override():
                        driver.find_element_by_css_selector("input[type=submit]").click()

            def enter_reason(reason):
                        element = driver.find_element_by_id("reason")
                        element.send_keys(reason)

class PersonPO():

            def override_reasons():
                        wait = WebDriverWait(driver, 10)
            wait.until(EC.presence_of_all_elements_located((By.CSS,'li.reason')))

            return [el.text for el in driver.find_elements_by_css("li.reason")]
```

AN EXAMPLE

Test Code:

```
modalPO.enter_reason("a reason")
modalPO.save_override()

assertEqual("a reason", personPO.reasons())
```

Test Result:

Passes locally

Passes on CI

AN EXAMPLE

OTHER WAYS

Implicitly wait for elements

```
driver = webdriver.Firefox()
driver.implicitly_wait(10) # seconds
driver.find_element_by_css("li.reason")
```

Built into the framework

Protractor
end to end testing for AngularJS

AN EXAMPLE

Test Code:

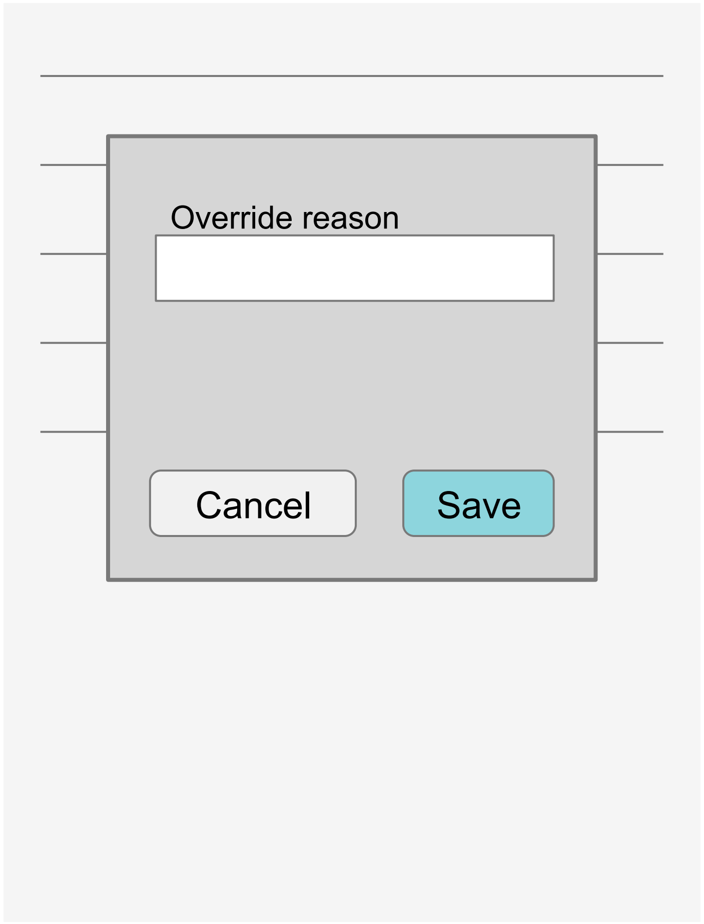
```
modalPO.enter_reason("a reason")
modalPO.save_override()

assertEqual("a reason", personPO.reasons())
```

Test Result:
Reasons don’t reload on QA

FAILS 1 in 5 times locally
FAILS 1 in 10 times on CI

AN EXAMPLE



Behaviour:

Single Page application

- User adds new override
- Modal displays, enters an override reason
- Clicks Save
- Person screen reloads listing the new override

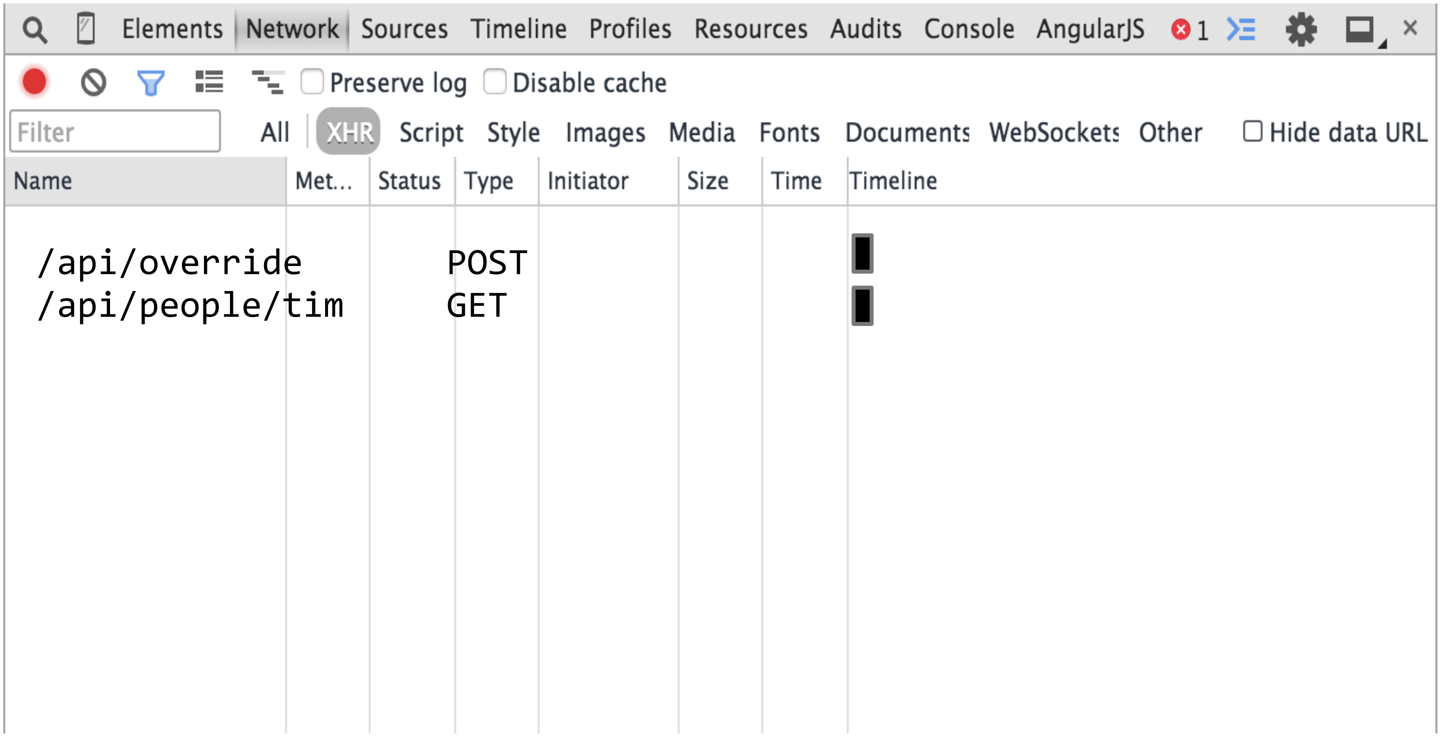
DEBUGGING

Be aware of common gotchas

Understand the application design

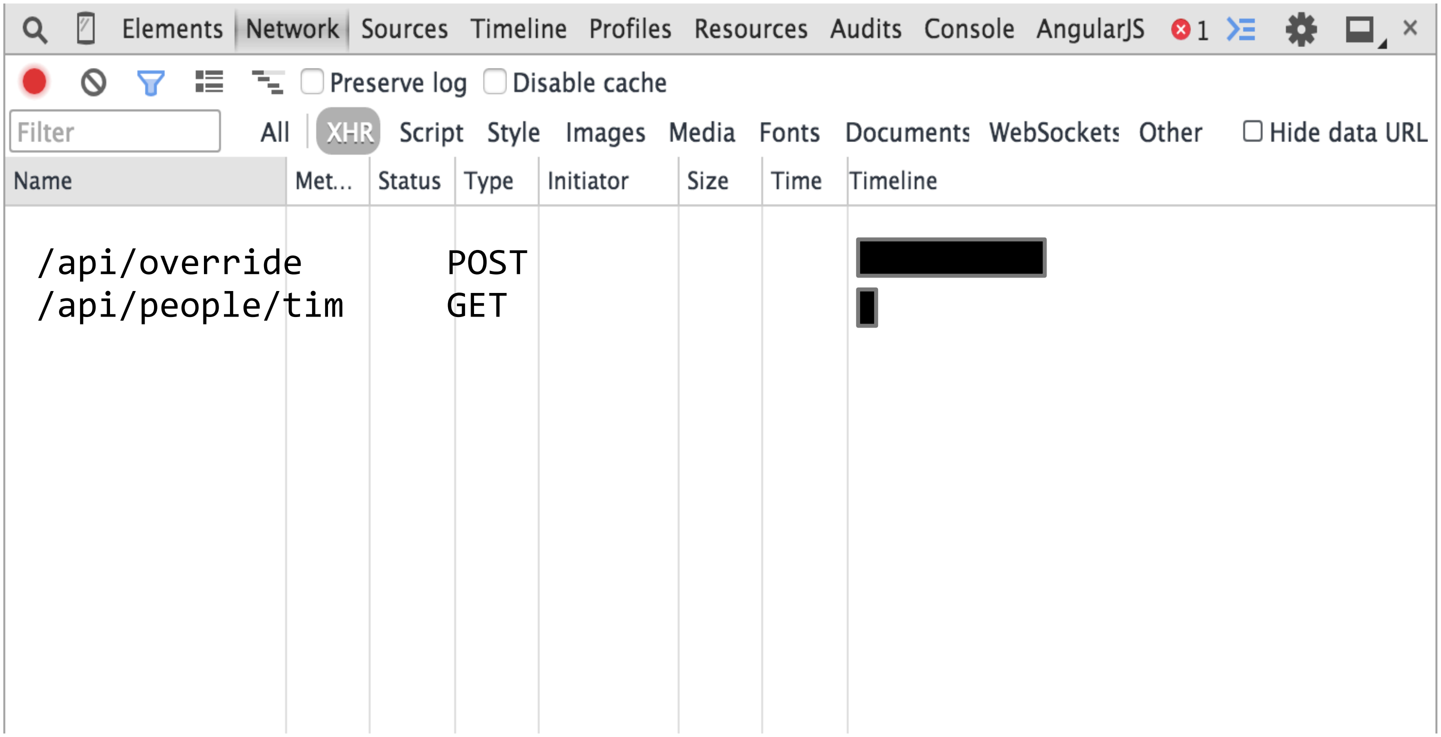
Look at XHR calls

DEBUGGING



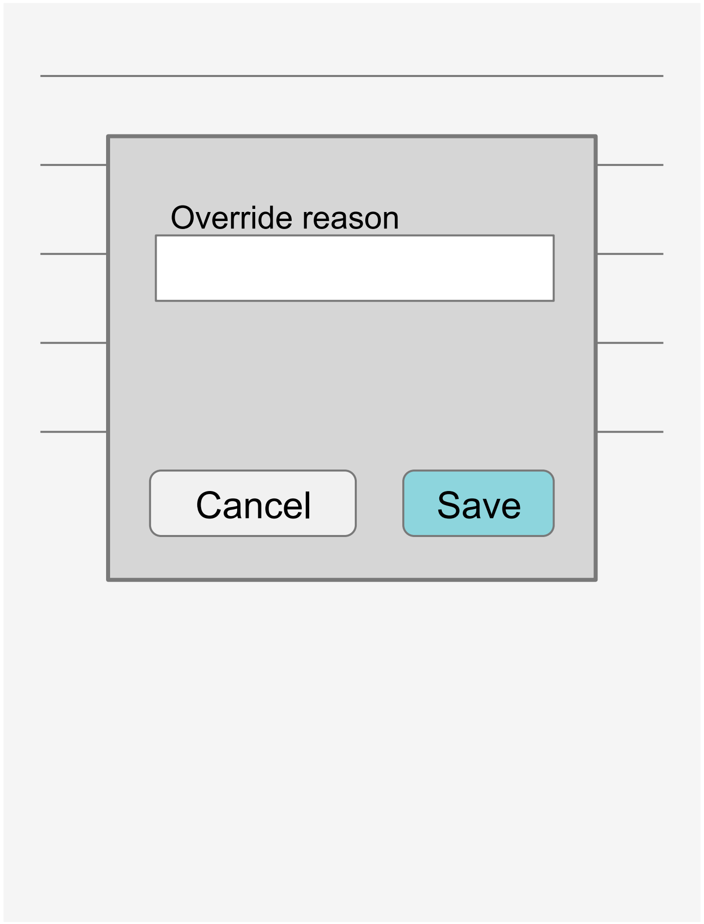
Name	Met...	Status	Type	Initiator	Size	Time	Timeline
/api/override	POST						▮
/api/people/tim	GET						▮

DEBUGGING



```
Name               Met...  Status  Type  Initiator  Size  Time  Timeline
/api/override      POST
/api/people/tim    GET
```

AN EXAMPLE



Problems

- Save button fires ajax override POST
- Reload is fired on modal close

Further Problems

- Development and CI server is single threaded
- QA and Prod is multi threaded

WHAT HAVE WE LEARNT?

Acceptance testing is hard!

PROBLEM TYPES

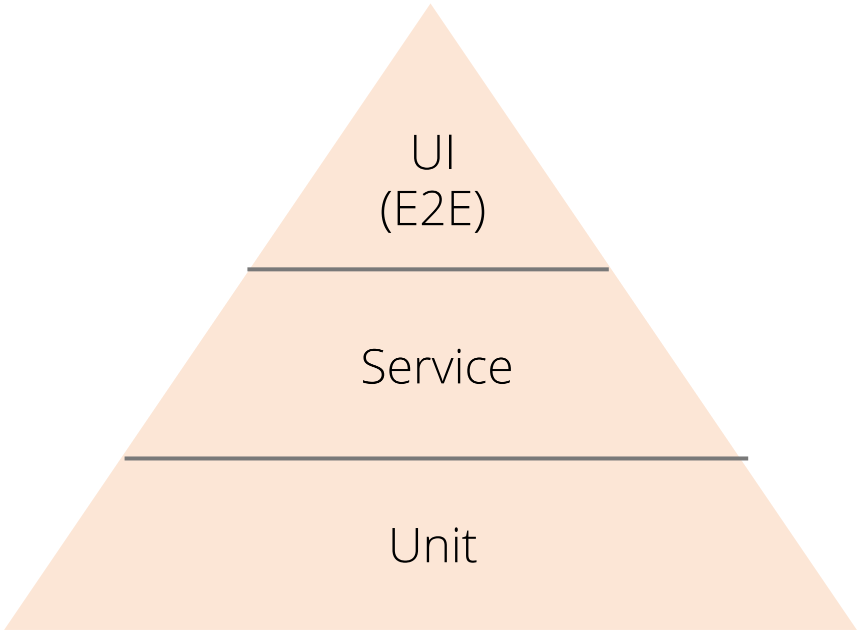
Slowness

Flakiness

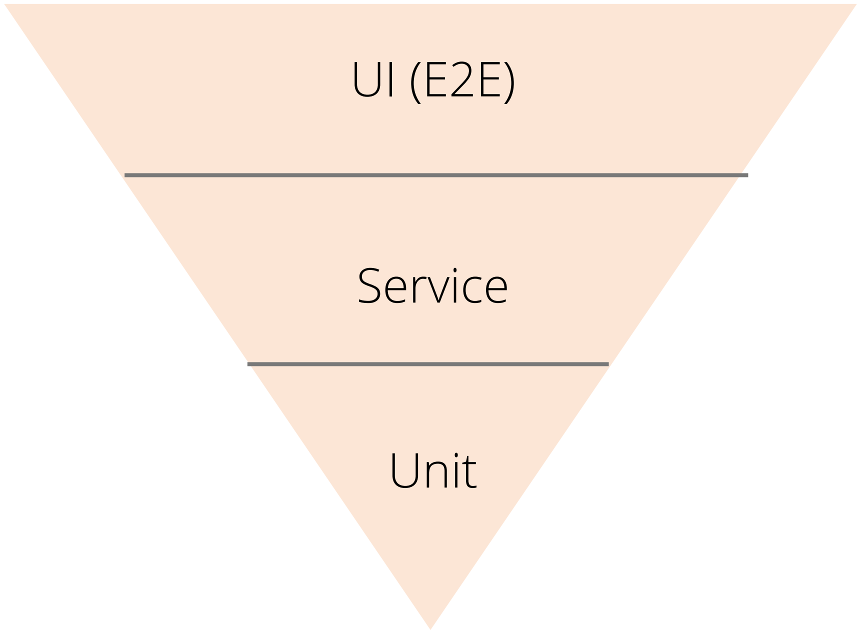
Difficulty to develop and maintain

Not providing value

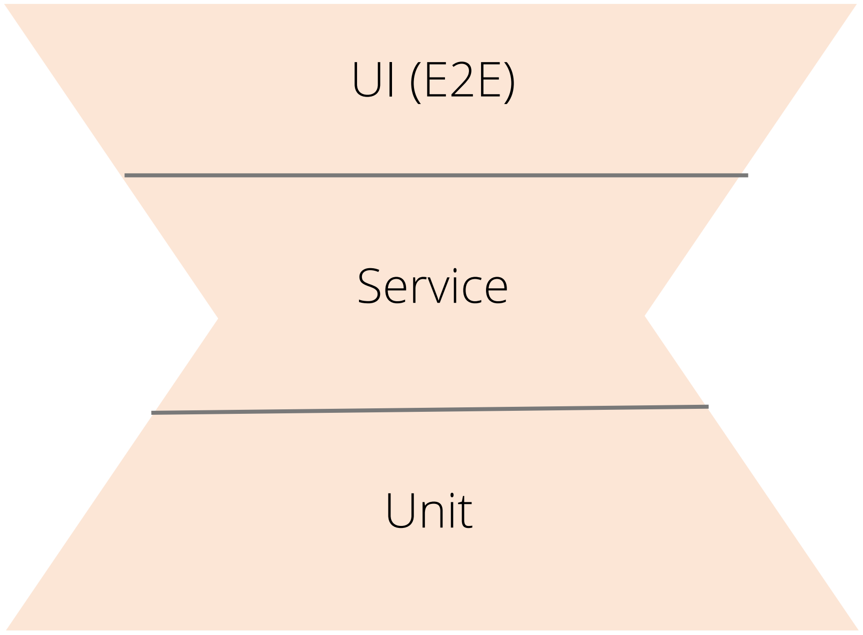
TESTING PYRAMID



TESTING PYRAMID

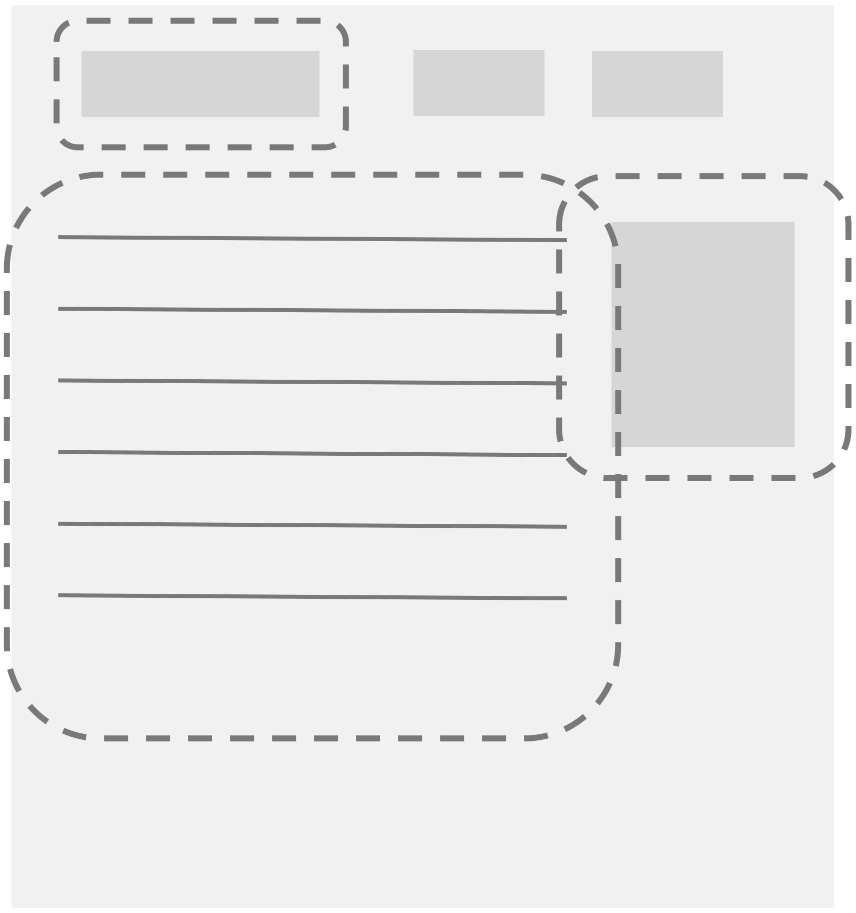


TESTING PYRAMID



COMPONENT TESTING

COMPONENT TESTING



JAVASCRIPT FUNCTIONAL TESTING

Front end frameworks have MVC

You can test services in isolation

e.g. `SearchService`

Object Factory

Expose factories for objects you need, saves you from going through the UI every time.

```
class RequestBuilder()

        def __init__():
                self.data = {
                        'created_by': 'tim',
                        'created_at': datetime.now,
                        'status': 'pending',
                        'history': []
                }

        def build():
                return self.data

        def with_status(status):
                self.data['status'] = status
                self.data['history'].append(status)
```

Object Factory

Use:

```
def test_submitted_request_appears_correctly_on_request_page(self):

            request = RequestBuilder()
                        .with_status('created')
                        .build()
            self.persistence_helper.create_request(request)

            expected_data = {
                        'request_status': 'Open',
                        'created_by': 'Tim Cochran'
            }

            self.request_page.navigate_to()
            assert_equal(expected_data, self.request_page.get_request_data())
```

COMMANDMENTS OF TESTING

(In our *humble* opinion)

1. Test Suite should be under 15 mins
2. Tests should be run before every check in
3. Build should Always be green, no broken windows
4. Don’t just rerun flaky tests and check in
5. Feature test coverage should be reviewed
6. Don’t check in without testing

DESIGNING FOR TESTABILITY

DESIGNING FOR TESTABILITY

Baking testing into the application,

so that functionality can be easily verified as working in

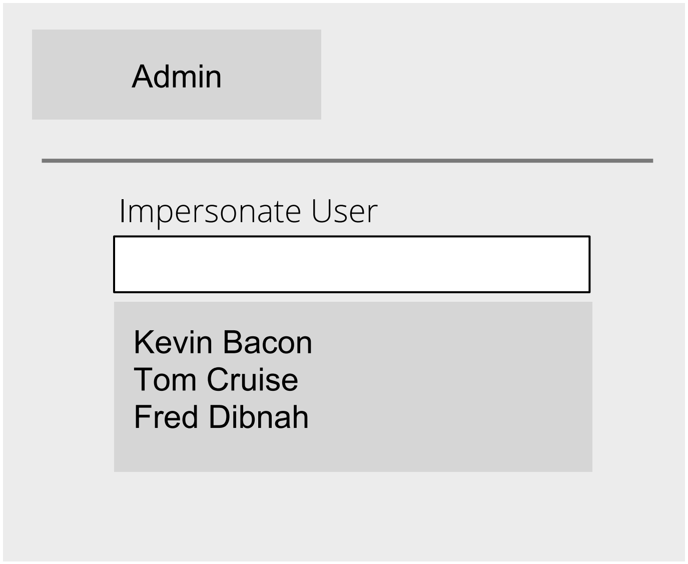
any environment, at any given time, by anyone

DESIGNING FOR TESTABILITY

Problem:

How can we test that users have the right abilities for their roles

Solution:



WHAT ABOUT SECURITY?

You are moving test code into production

It should follow the same quality and security standards as production code.

CAPTURING SCENARIOS

Problem:

Capturing scenarios from stakeholders

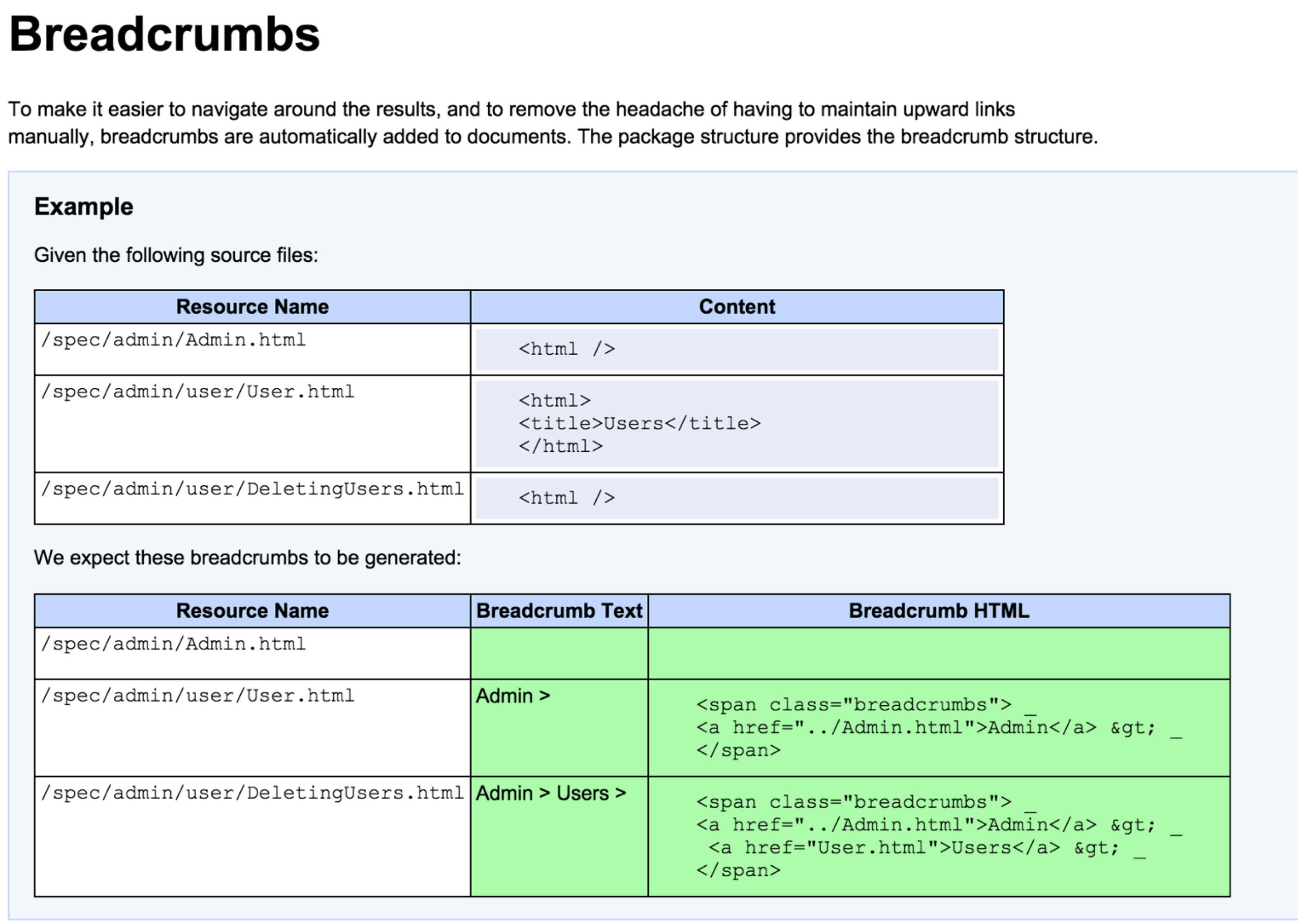
Solutions:

Recording users input into a dsl or testing language

Putting a test framework into the application

- Allows you to test the application quickly in Sub-Prod environment
- Can also become power user functionality

BDD EXAMPLE



Breadcrumbs

To make it easier to navigate around the results, and to remove the headache of having to maintain upward links manually, breadcrumbs are automatically added to documents. The package structure provides the breadcrumb structure.

Example

Given the following source files:

Resource Name	Content
`/spec/admin/Admin.html`	`<html />`

`/spec/admin/user/User.html`	`<html>`
	`<title>Users</title>`
	`</html>`

`/spec/admin/user/DeletingUsers.html`	`<html />`

We expect these breadcrumbs to be generated:

Resource Name	Breadcrumb Text	Breadcrumb HTML
`/spec/admin/Admin.html`		

`/spec/admin/user/User.html`	Admin >	`<span class="breadcrumbs"> _`
		`<a href="../Admin.html">Admin</a> &gt; _`
		`</span>`

`/spec/admin/user/DeletingUsers.html`	Admin > Users >	`<span class="breadcrumbs"> _`
		`<a href="../Admin.html">Admin</a> &gt; _`
		` <a href="User.html">Users</a> &gt; _`
		`</span>`

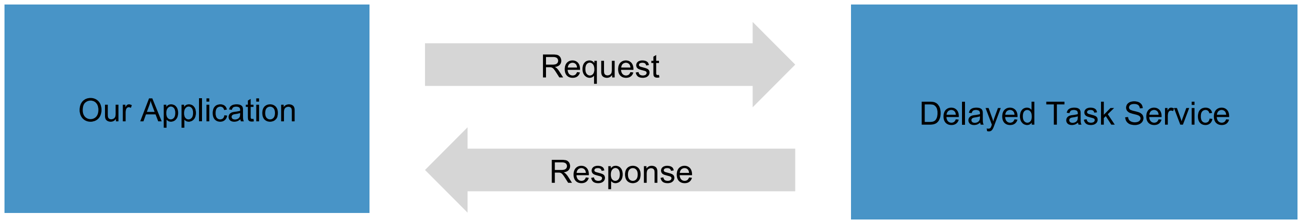
INTEGRATION ERRORS

Problem:

Lots of Interaction with external service (HTTP)

Drives rich UI and logic we want to test (Automation and Exploratory)

Hard to reproduce infrequent error scenarios



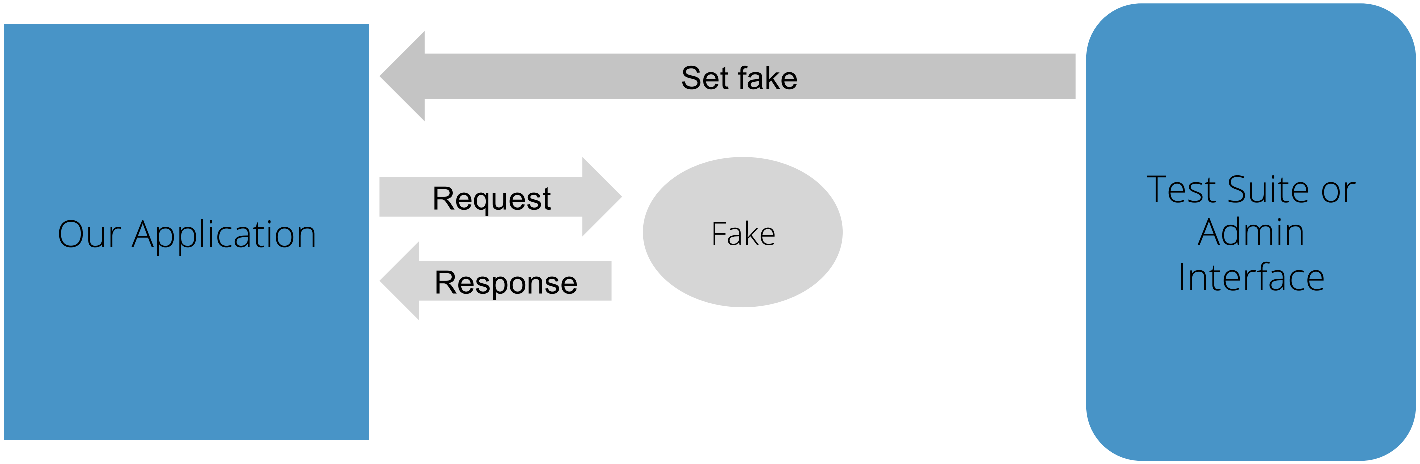
INTEGRATION ERRORS

Solution:

Endpoint for setting fake responses with error codes

Can be developed against, tested manually against, called by automated tests

Postback can be driven from test suite or from additional infrastructure



THANK YOU

For questions or suggestions:

Dan Lockman

dlockman@thoughtworks.com

Tim Cochran

tcochran@thoughtworks.com

bit.ly/tw-philly

ThoughtWorks®