

5 levels of API test automation



ALLAN GRAY
LONG-TERM INVESTING

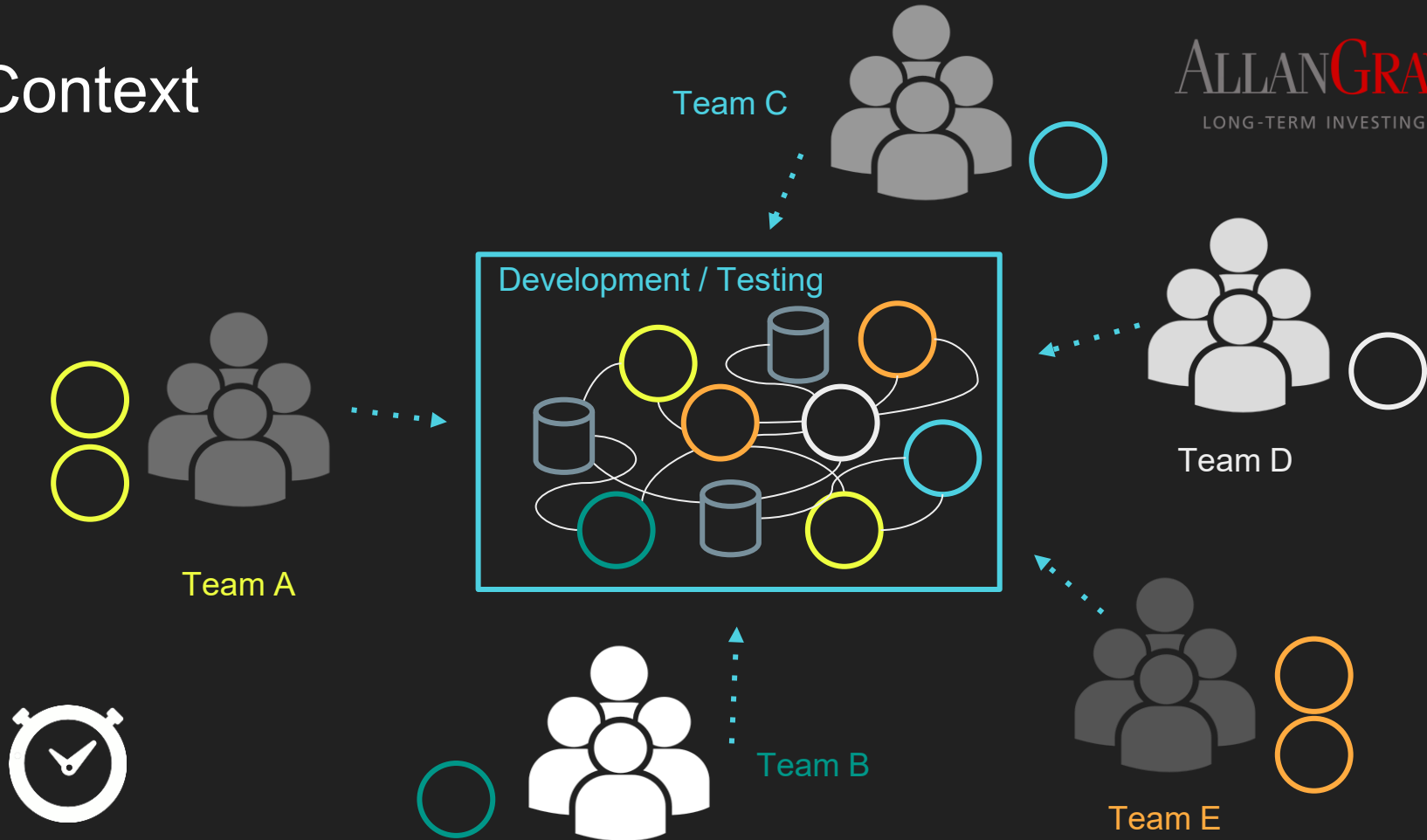
About me



13 years
testing
experience

Full stack
tester

Context



Some key AllnGray automation testing principles

Programmatic approach.

- Anyone can contribute as they build features.
- Test code lives with the feature code.

Design the framework to be easily maintainable.

- Testers are at different levels.
- Consistency is king.

Automation is a tool

- Automate to support testing not replace it.

API test example

API test, so why 5 levels?

```
test('Search people named shakes', async () => {  
  const url = 'http://api.dev.com/people?search=shakes';  
  const response = await request.GET(url);  
  expect(response.code).toBe(200);  
});
```

Unpack a test

```
test('Search people named shakes', async () => {  
  const url = 'http://api.dev.com/people?search=shakes';  
  const response = await request.GET(url);  
  expect(response.code).toBe(200);  
})
```

Unpack a test

I need legit data



```
test('Search people named shakes', async () => {  
  const url = 'http://api.dev.com/people?search=shakes';  
  const response = await request.GET(url);  
  expect(response.code).toBe(200);  
})
```



Unpack a test

```
test('Search people named shakes', async () => {  
  const url = 'http://api.dev.com/people?search=shakes';  
  const response = await request.GET(url);  
  expect(response.code).toBe(200);  
})
```

I need legit data



Should probably test
different response
codes... but how?



Unpack a test

API has been deployed to an env.

I need legit data

```
test('Search people named shakes', async () => {  
  const url = 'http://api.dev.com/people?search=shakes';  
  const response = await request.GET(url);  
  expect(response.code).toBe(200);  
});
```

Should probably test different response codes... but how?

Unpack a test

API has been deployed to an env.

I need legit data

```
test('Search people named shakes', async () => {  
  const url = 'http://api.dev.com/people?search=shakes';  
  const response = await request.GET(url);  
  expect(response.code).toBe(200);  
  expect(response.body.length).toBeGreaterThan(0);  
  expect(response.body[0].name).toContain('shakes');  
});
```

Should probably test different response codes... but how?

Good idea to do more checks.

Unpack a test

API has been deployed to an env.

I need legit data

```
test('Search people named shakes', async () => {  
  const url = 'http://api.dev.com/people?search=shakes';  
  const response = await request.GET(url);  
  expect(response.code).toBe(200);  
  expect(response.body.length).toBeGreaterThan(0);  
  expect(response.body[0].name).toContain('shakes');  
});
```

Should probably test different response codes... but how?

Good idea to do more checks.

Are there restrictions?
Numbers? special chars?

Unpack a test

API has been deployed to an env.

I need legit data

```
test('Search people named shakes', async () => {  
  const url = 'http://api.dev.com/people?search=shakes';  
  const response = await request.GET(url);  
  expect(response.code).toBe(200);  
  expect(response.body.length).toBeGreaterThan(0);  
  expect(response.body[0].name).toContain('shakes');  
});
```

Should probably test different response codes... but how?

Good idea to do more checks.

Test ran fine 15 mins ago ???

Are there restrictions?
Numbers? special chars?

Levels

What type of tests per level?

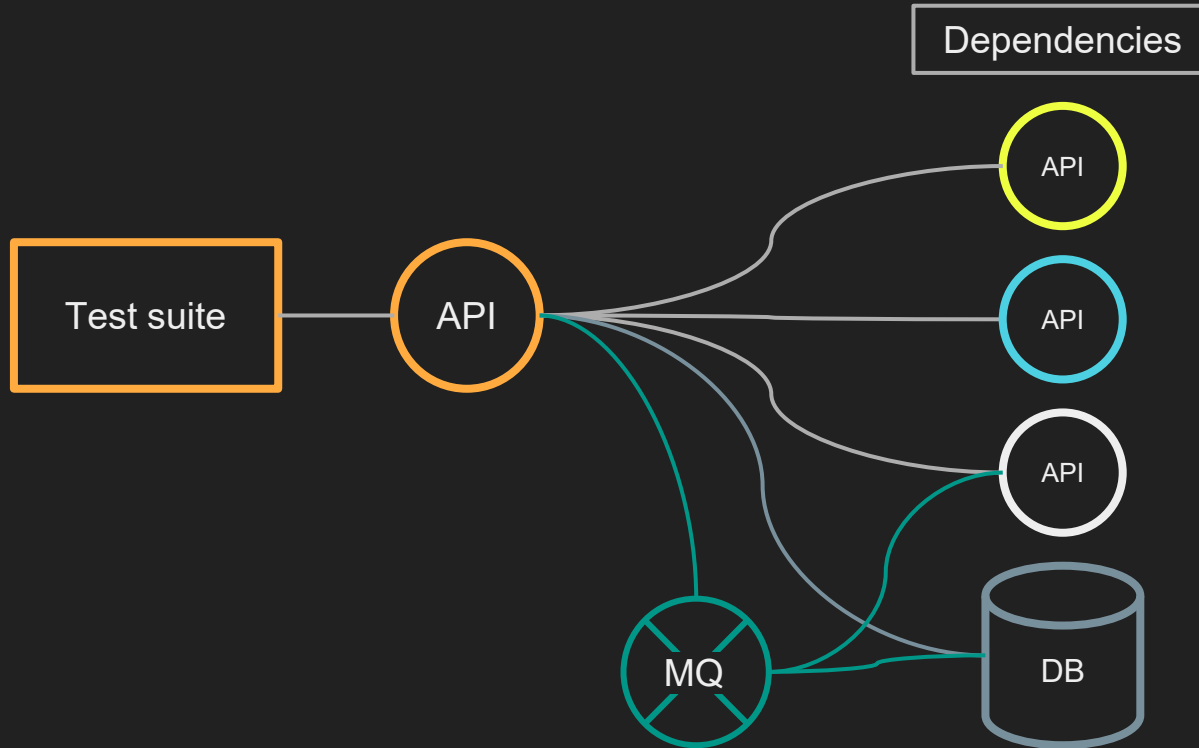
What data to use?

When do they run?

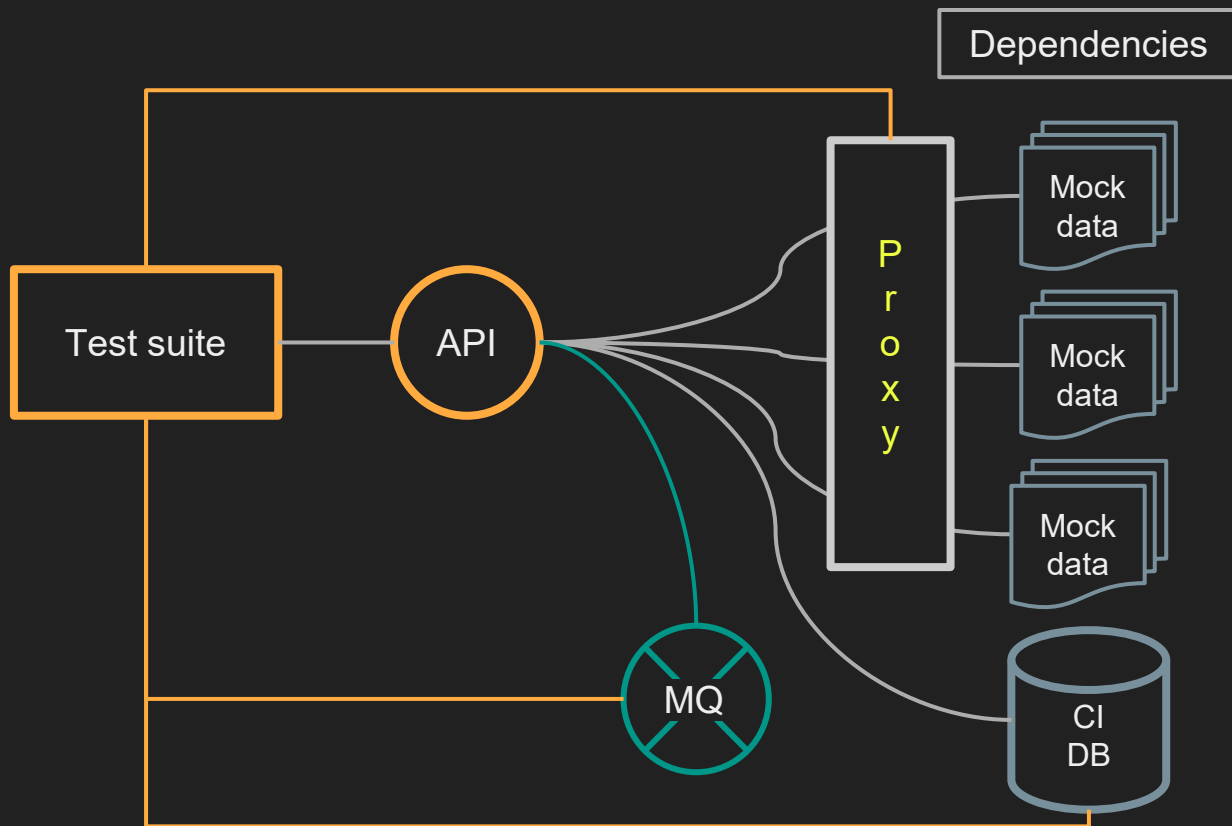
- Acceptance / Component tests
- Verification / Integration tests
- Smoke tests
- Synthetic tests
- Data chaos monkey tests

Assumption that unit / unit integration tests have been done.

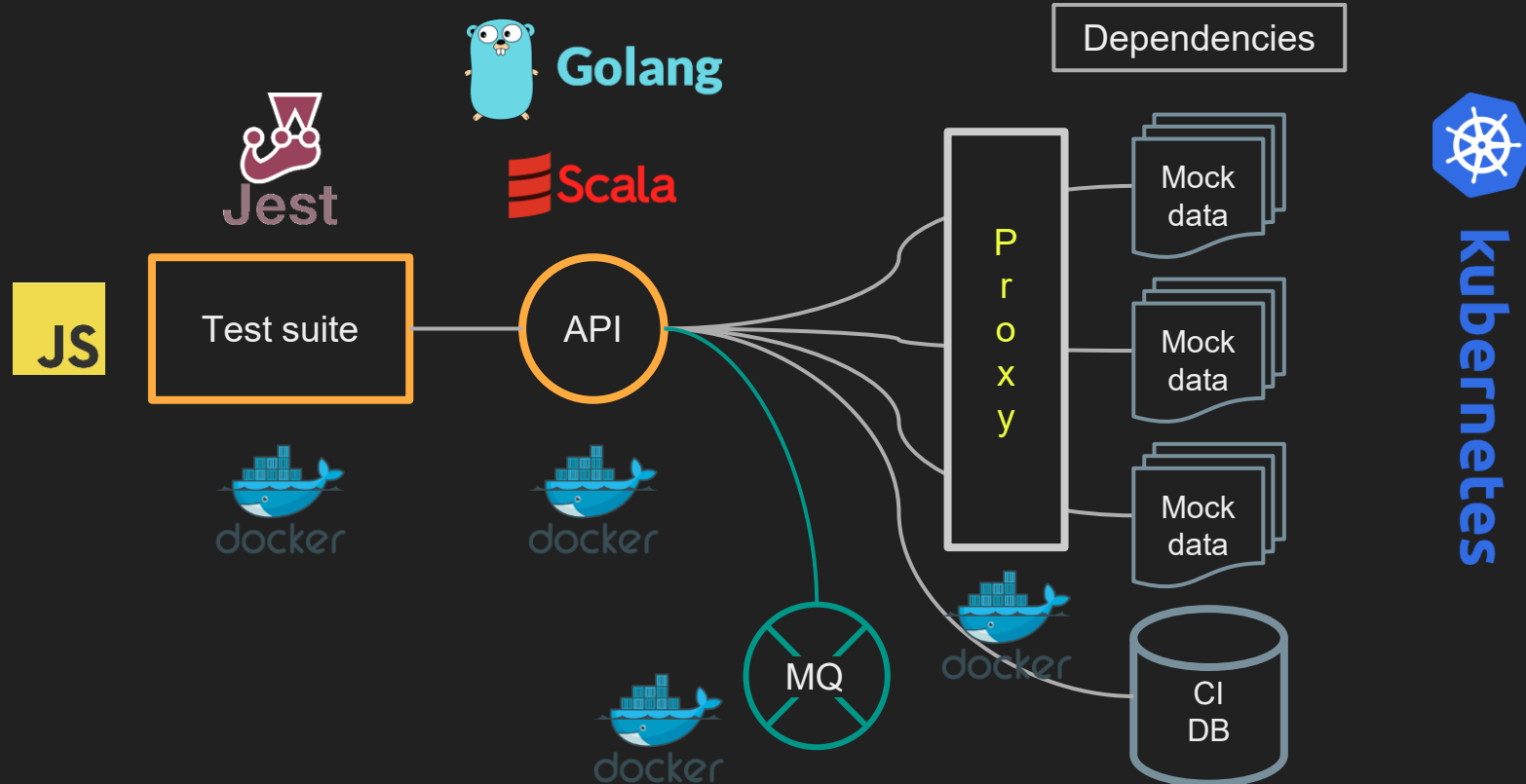
Acceptance / component tests



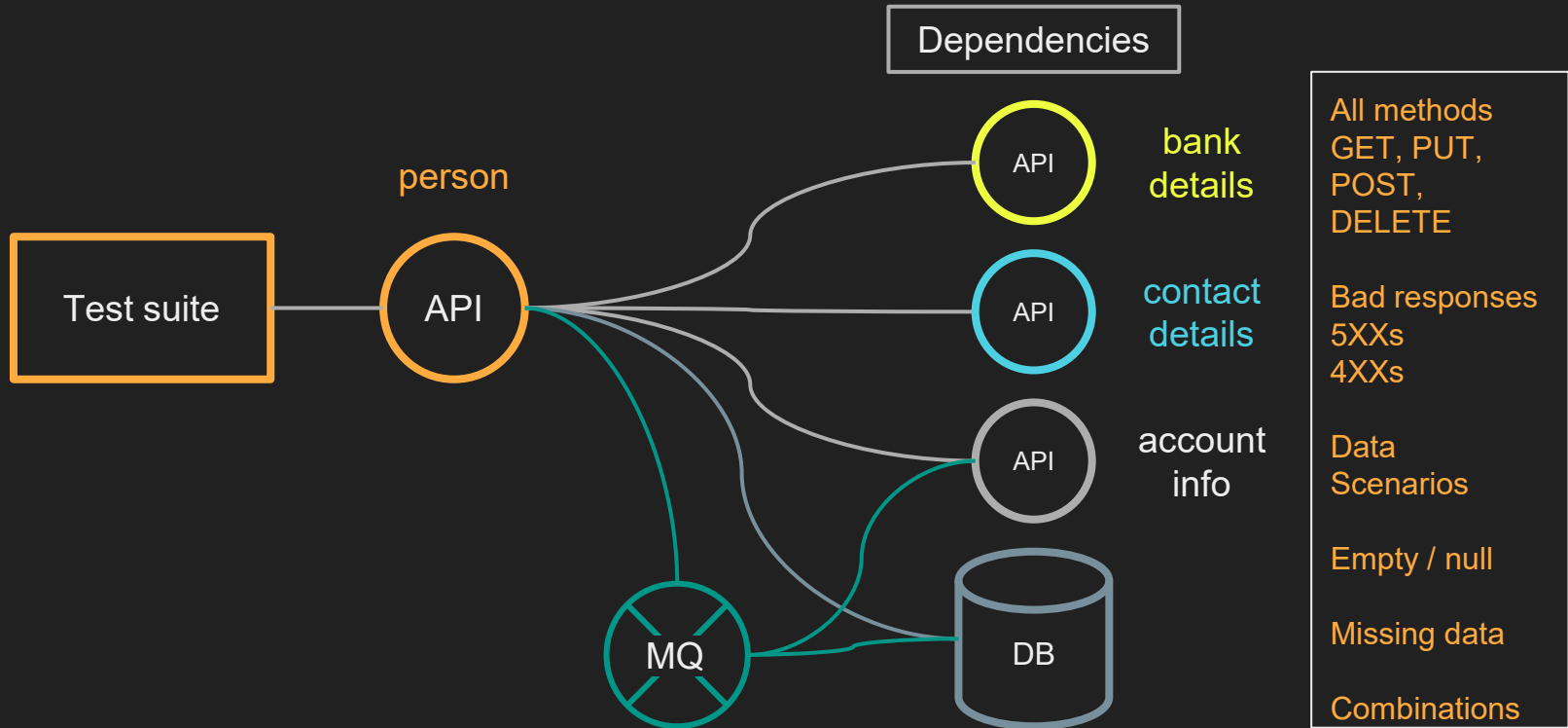
Acceptance / component tests



Acceptance / component tests



Acceptance / component tests



Mocking is great...

Amazing test scenario coverage.

Tests pass consistently in this mocked world.



```
test('Search people named bob with mocked dependencies' function () {  
  mockServer.loadFixtures('dependencies', 'dependencyFixture2',  
    dependencyFixture);  
  
  const url = 'http://api.local.com/people?search=bob';  
  const response = await request.GET(url);  
  expect(response).toMatchSnapshot();  
});
```

Verification / Integration tests

Understand the implementation.

Tests focus on integration points.

Core functionality 80:20

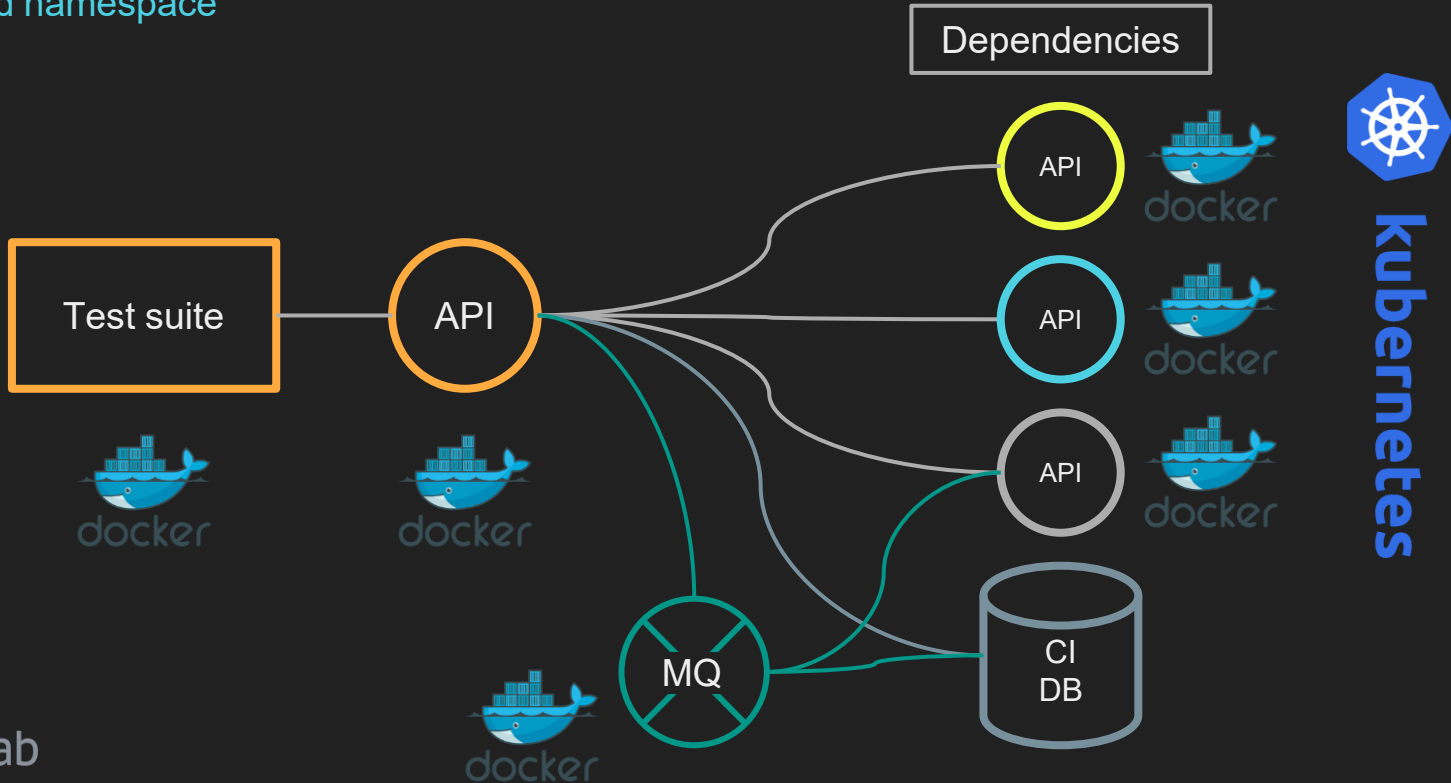
Isolate the integration environment.

Verification / Integration tests

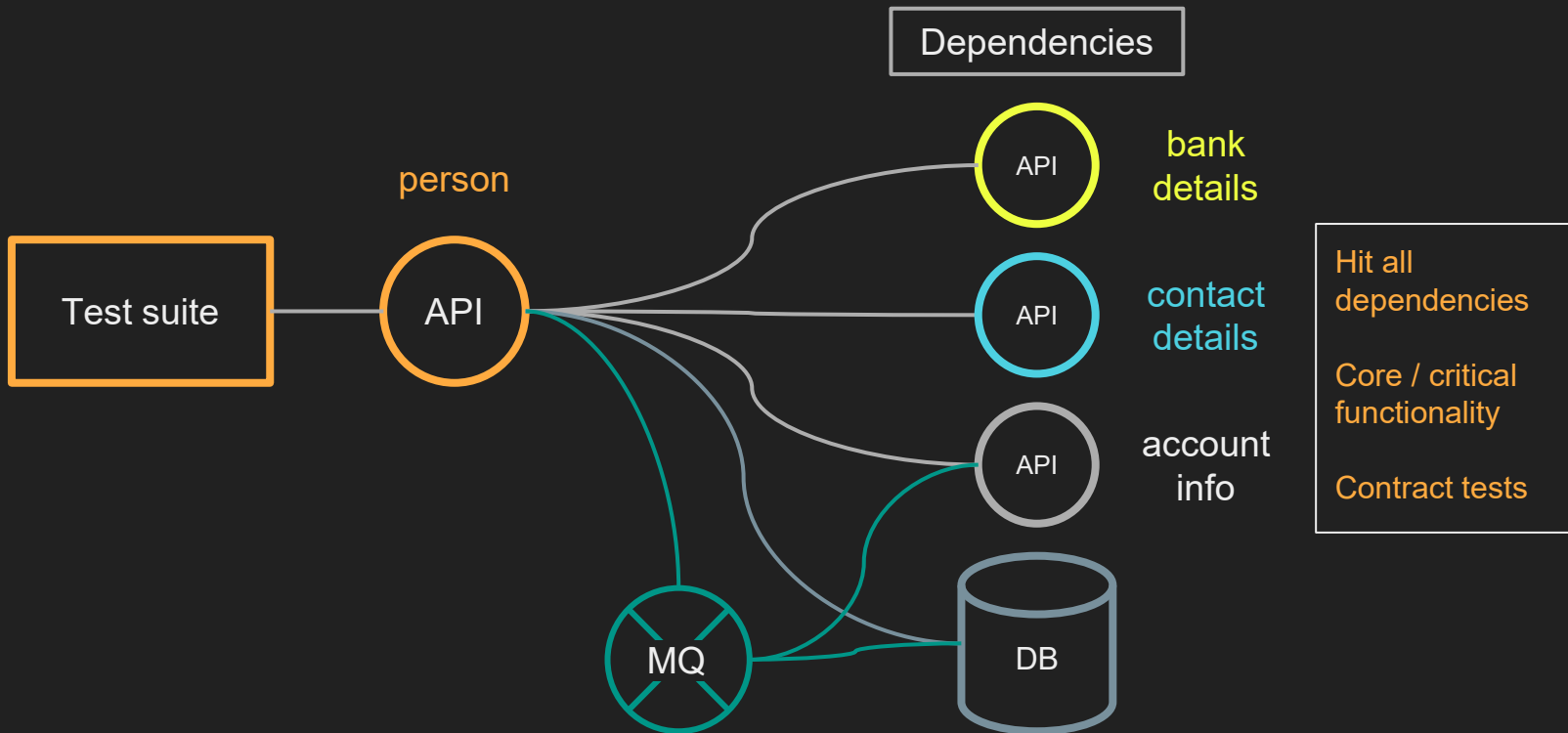
Ringfenced namespace

Test with actual dependencies

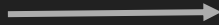
Ringfenced namespace



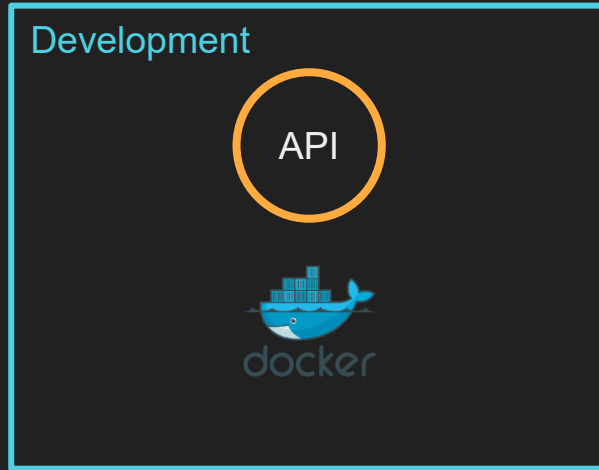
Verification / Integration tests



All good to go... lets deploy



All good to go... lets deploy



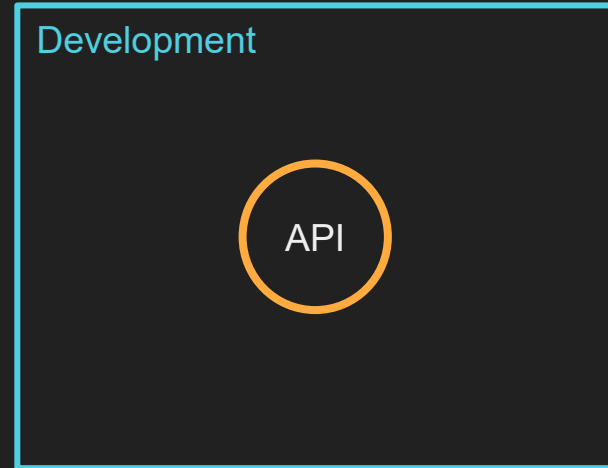
Post deployment Smoke tests

Super happy day cases

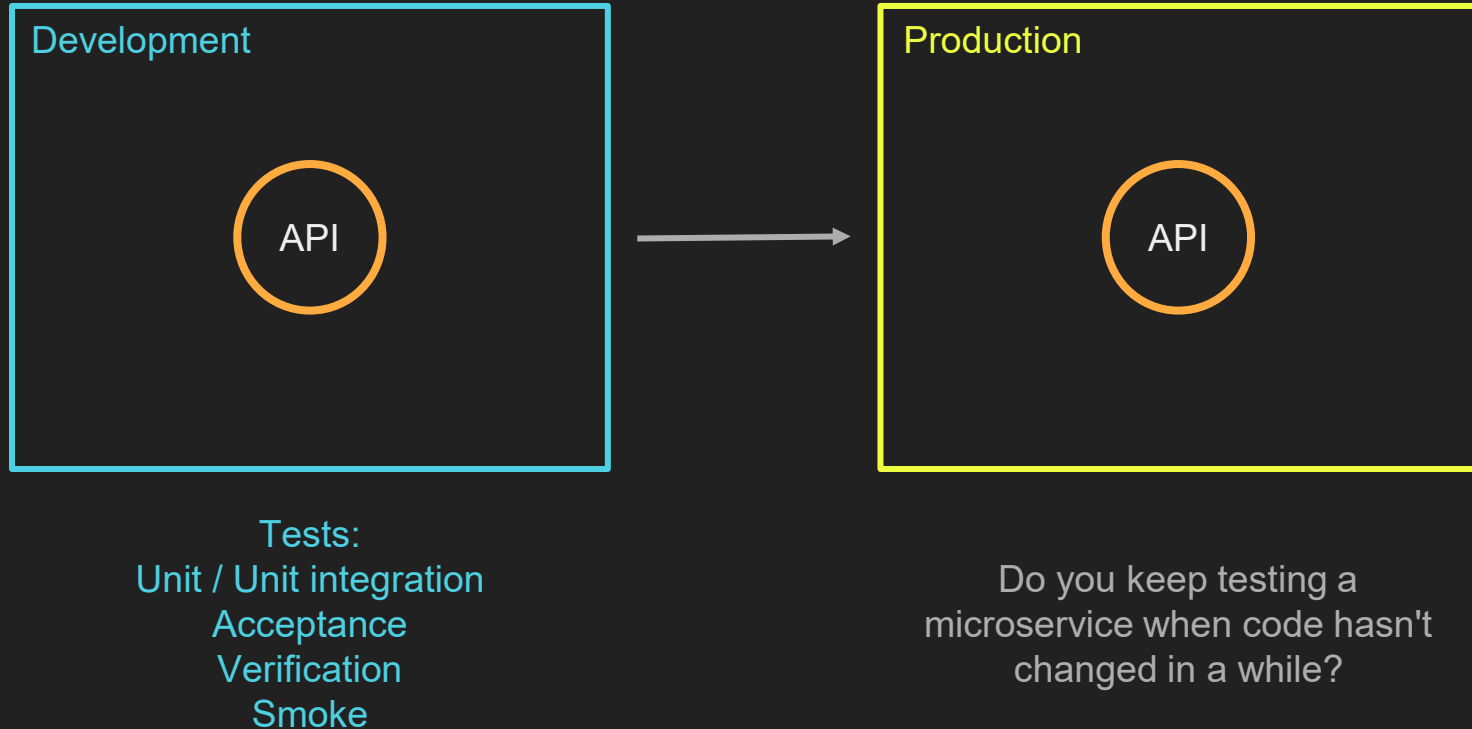
Focus on GET endpoints to avoid manipulating data.

Check any 3rd party integration that couldn't be simulated.

Now we can start testing...



API is stable in PROD



Synthetics tests

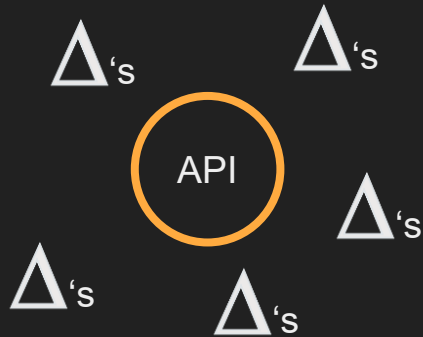
Similar suite of Smoke tests.

Runs on a schedule.

Optimize for best execution time.

```
test('Search people named Shekhar with few results', async () => {  
  const url = 'http://api.dev.com/people?search=Shekhar Ramphal';  
  
  const response = await request.GET(url);  
  
  expect(response.code).toBe(200);  
  expect(response.code).not.toBeUndefined();  
});
```

Checks run all the time



Data chaos monkey tests

Some things we just can't predict.

And our users are crazy.

Set random payloads or params.

Incorrect API usage.

Expect longer running tests.

Fetch data dynamically from DBs.



Best data is PROD data

```
test('Update a random persons name', async () => {  
  const searchTerm = await getRandomNameFromDB();  
  const url = `http://api.dev.com/people?search=${searchTerm}`;  
  
  const response = await request.GET(url);  
  
  expect(response.body[0].name).toBe(searchTerm);  
  
  const updateUrl = `http://api.dev.com/people/${response.id}`;  
  const newName = await getRandomNameFromDB();  
  
  await request.PUT(updateUrl, {name: newName});  
  const updatedResponse = await request.GET(updateUrl);  
  
  expect(updatedResponse.body[0].name).toBe(newName);  
});
```


In sprint?

- Should be able to ask questions without knowing the details.
 - What are the sources for this endpoint ?
 - Does it handle errors ?
 - Which databases does it talk to ?
 - Are there underlying SPs / Views, can I test those directly ?
 - Async ? What does the sequence of events look like ?
- Understand the implementation at a high level.
- Put them down as tasks

In sprint?

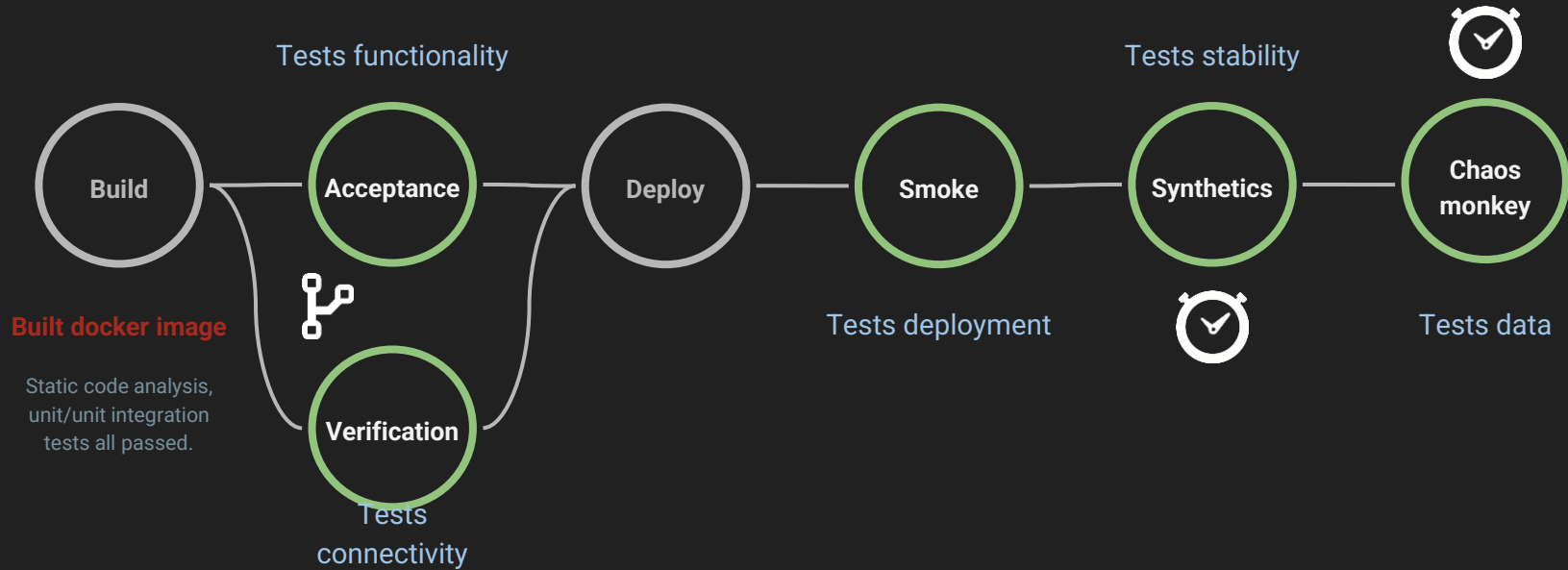
- Write the tests, even if they don't pass yet.

```
test.todo('Search people named bob');  
test.todo('Search people named bob with failed bank call');  
test.todo('Search people named bob with failed contact call');  
test.todo('Search people named bob with failed account call');  
test.todo('Search people with multiple results');  
test.todo('Search people with no results');  
test.todo('Search people with no bank details');  
test.todo('Search people with missing mandatory fields from account');
```

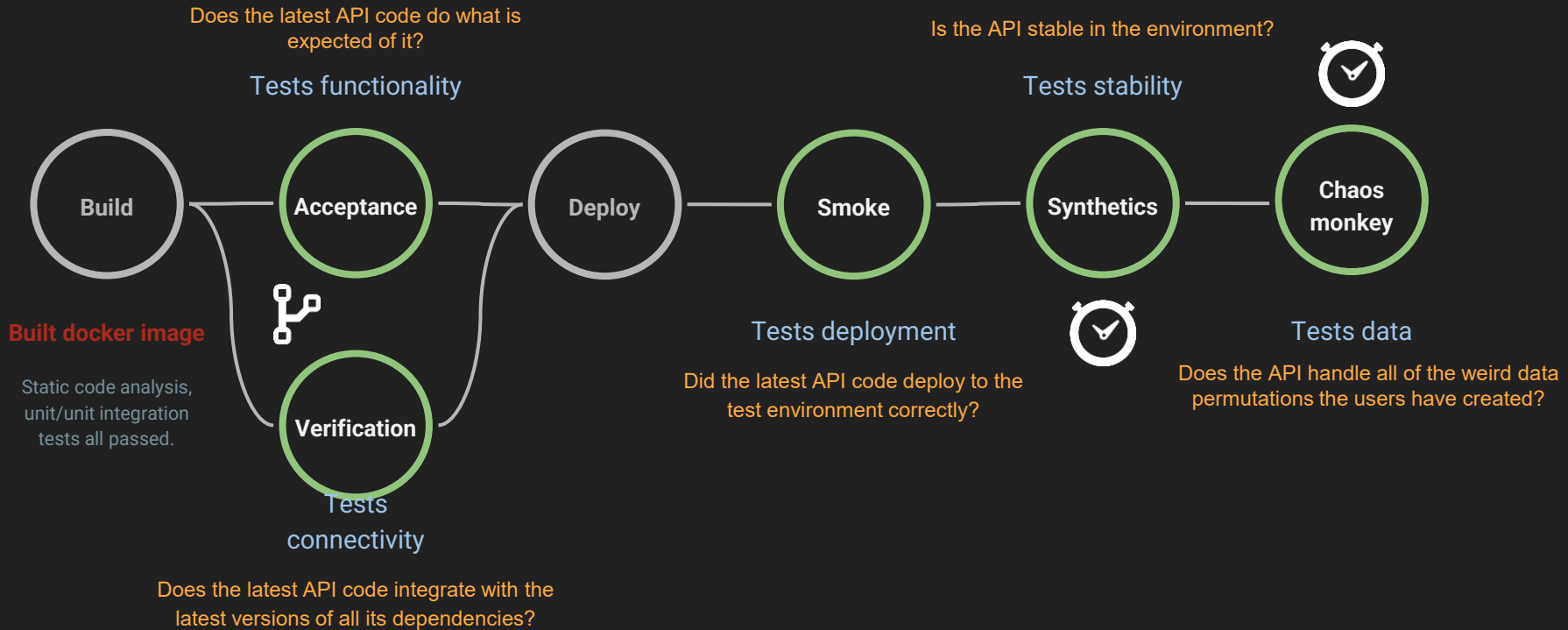
Pipeline



Recap



Recap



?