CS 4530: Fundamentals of Software Engineering

Module 12: Designing Tests for Large Systems

Jonathan Bell, Adeel Bhutta, Mitch Wand

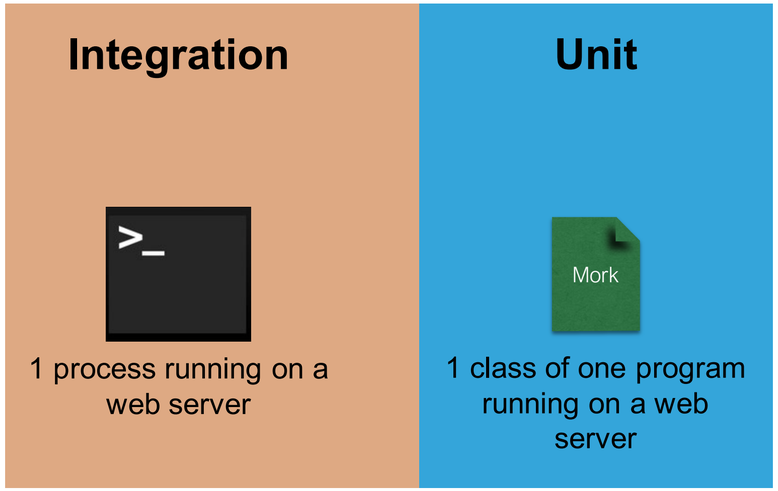
Khoury College of Computer Sciences

© 2022 Released under the CC BY-SA license

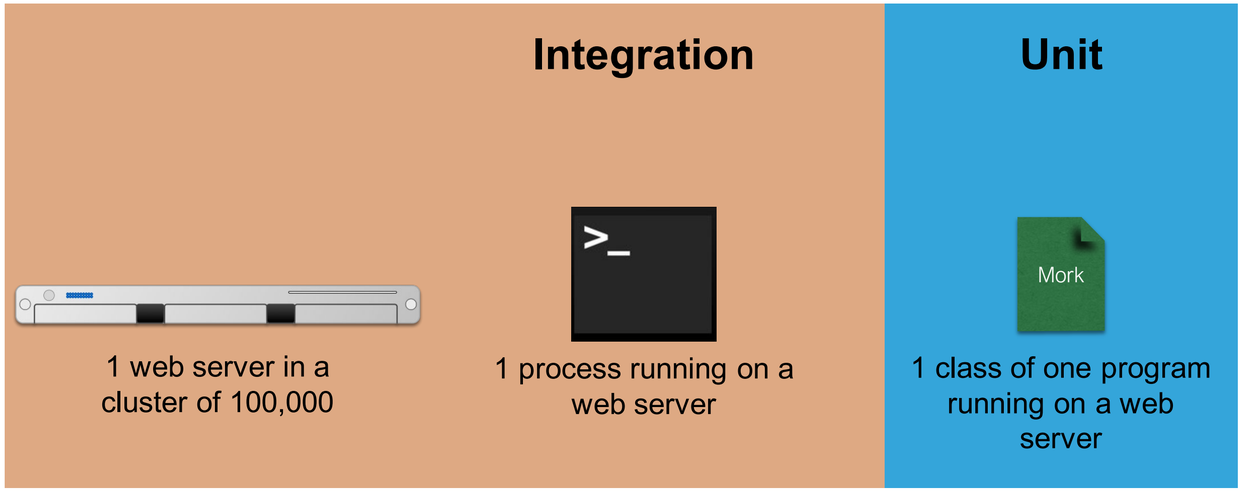
Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
  - Explain why you might need a "test double“ in your testing
  - Understand how and when to apply different kinds of test “doubles” such as “mocks and spies”

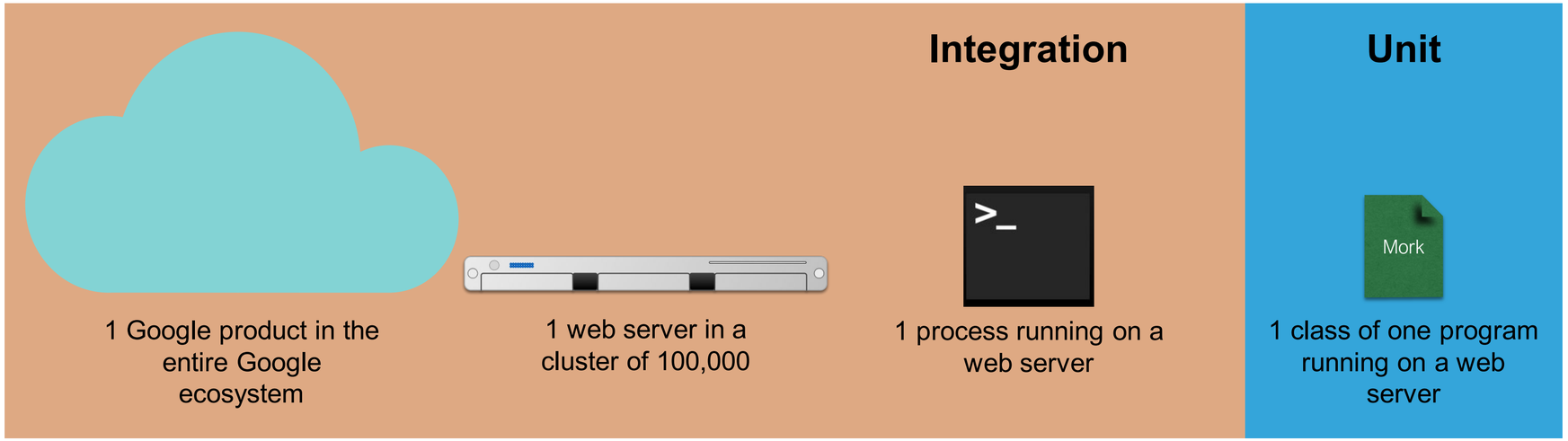
Testing Scopes Larger than Units



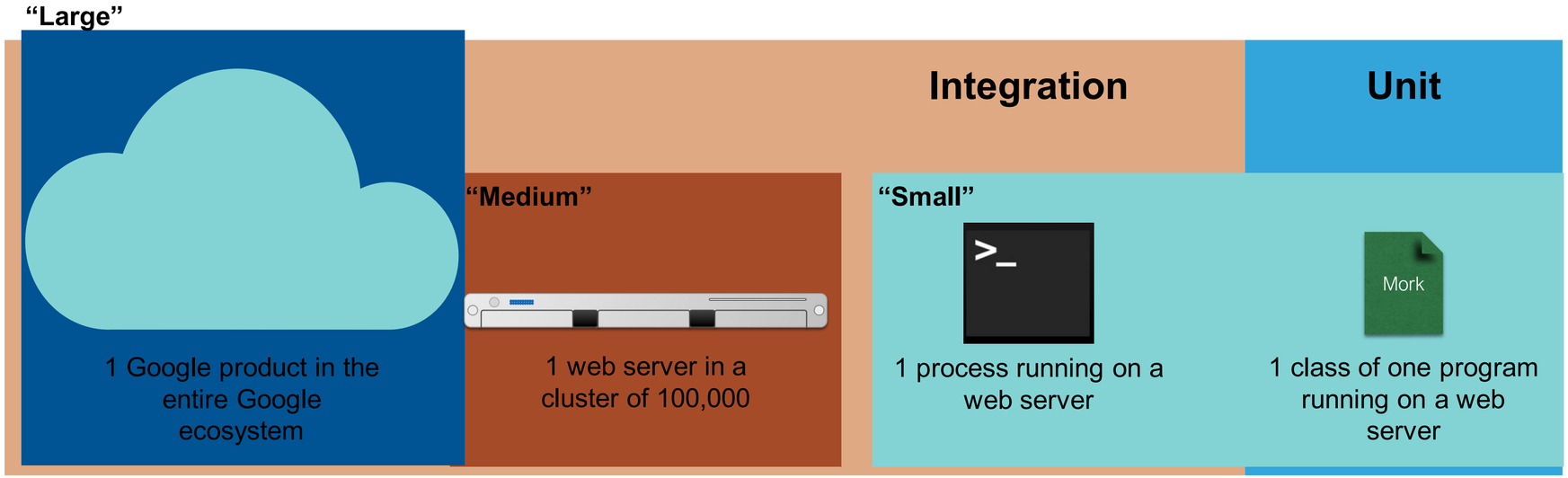
“Integration” Tests Might be Larger



Some Tests are Enormous



Classify Tests by Size and Scope



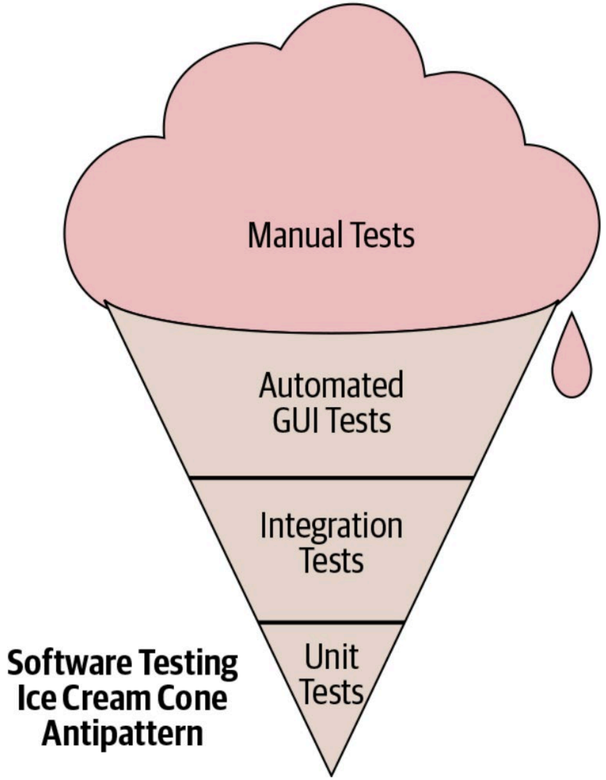
“Software Engineering at Google: Lessons Learned from Programming Over Time,” Wright, Winters and Manshreck, 2020 (O’Reilly)

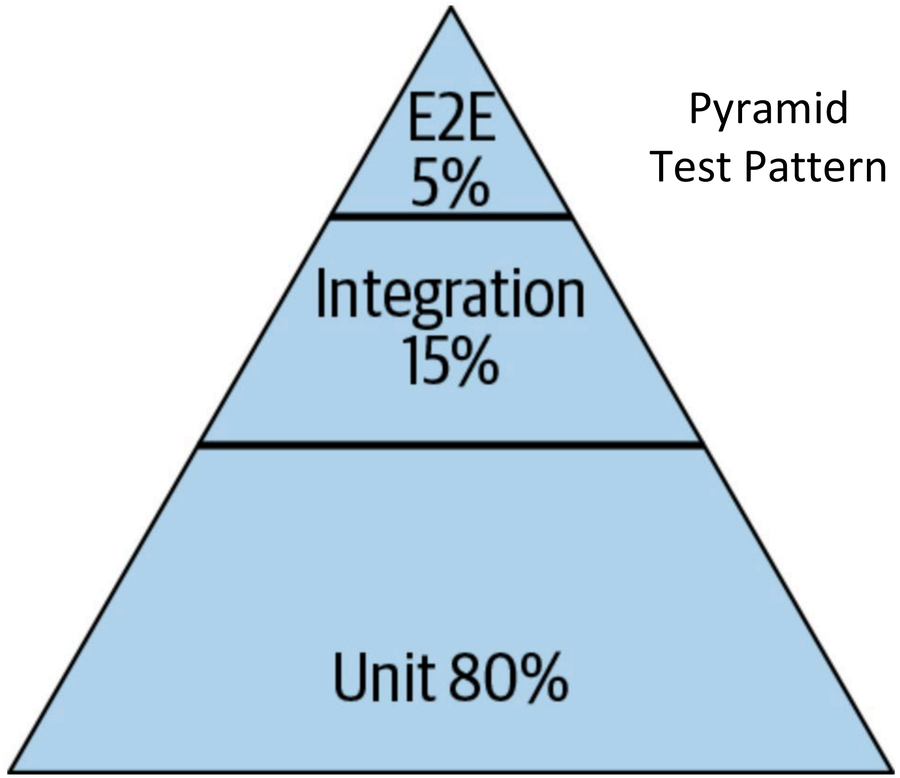
How big is my test?

- Small: run in a single thread, can't sleep, perform I/O or making blocking calls
- Medium: run on single computer, can use processes/threads, perform I/O, but only contact localhost
- Large: Everything else

"Software Engineering at Google: Lessons Learned from Programming Over Time," Wright, Winters and Manshreck, 2020 (O'Reilly)

Testing Distribution (How much of each kind of testing we should do?)





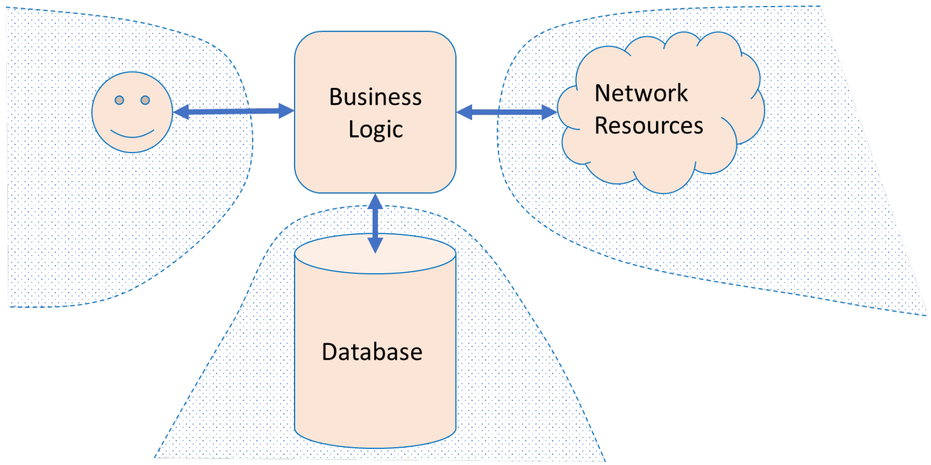
Pyramid Test Pattern

From SoftEng @ Google Chapter 11

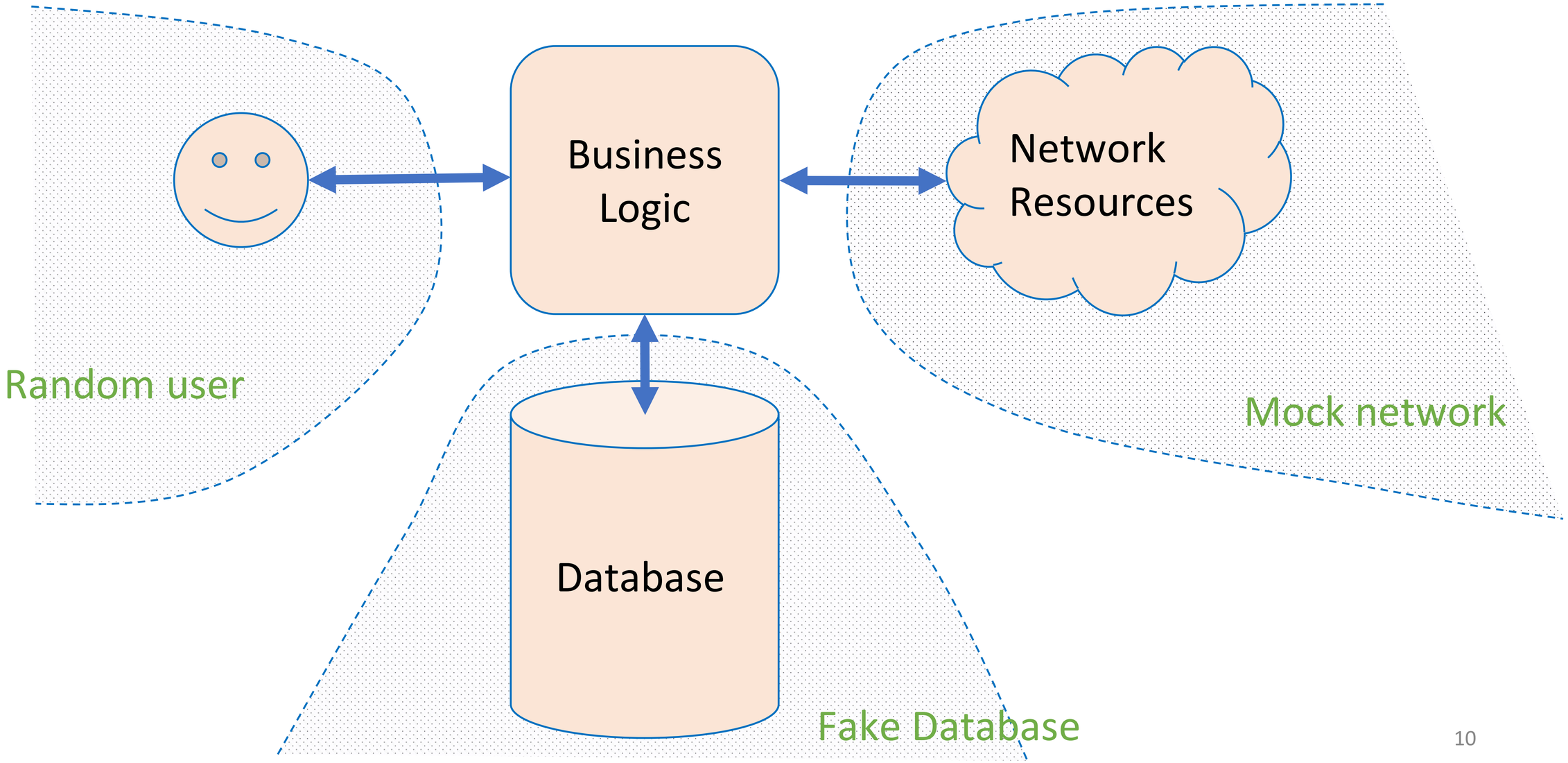
- https://learning.oreilly.com/library/view/software-engineering-

Large Systems are Hard to Test

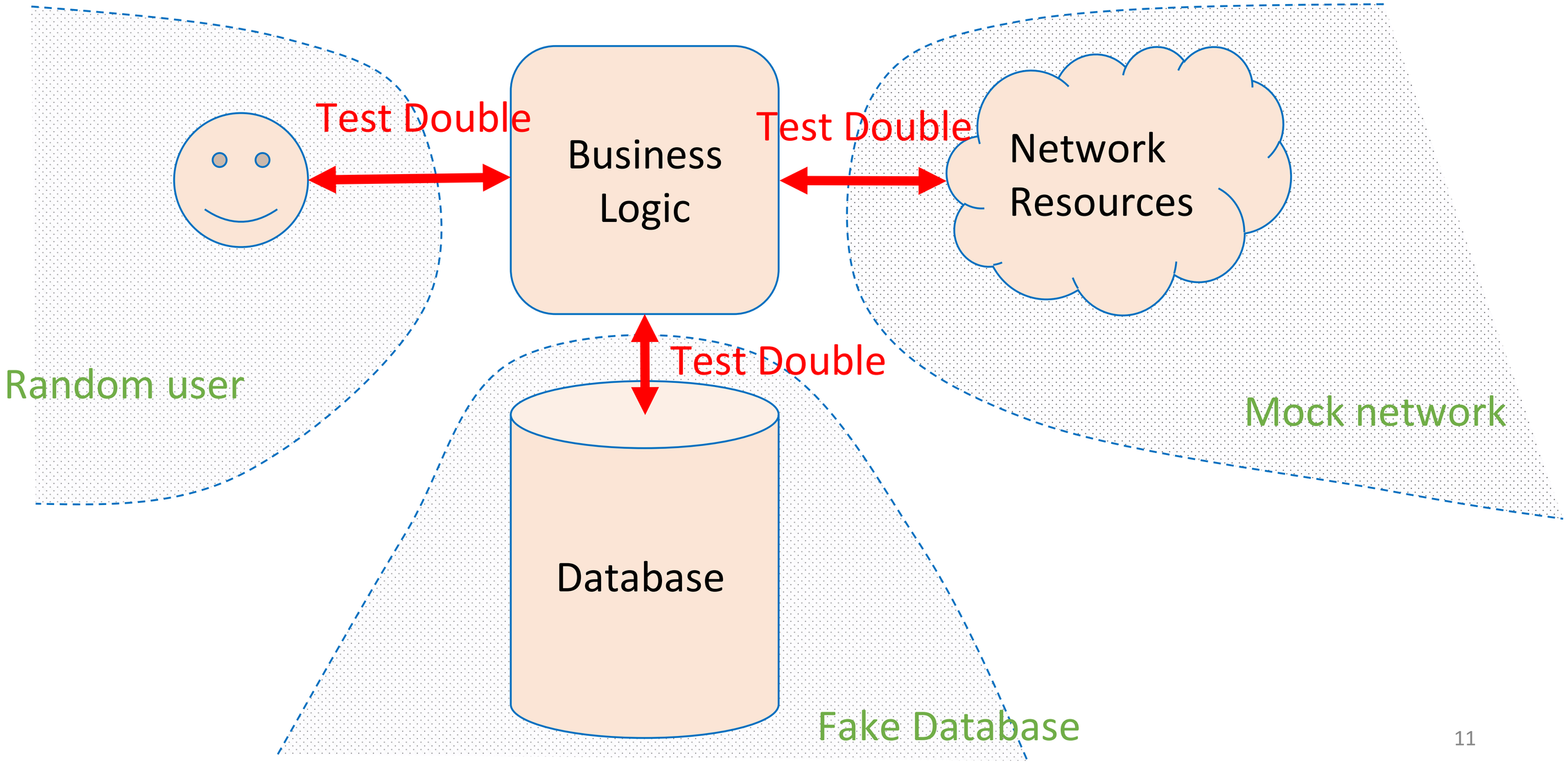
- Database component
  - Contents may need to reflect/simulate real-world;
  - Data may be expensive/proprietary/confidential.
- Network connections
  - ”Real” connections may be slow/flaky/disrupted;
  - Resources may have changed since test was written.
- Environment
  - Interactions with OS, locale or other software.
- Human actors
  - Ultimately unpredictable.
- Specification ambiguity
  - Large systems -> many behaviors/interactions to consider



Test Doubles replace uncontrollable pieces of the environment



What are Test Doubles?



When to use Test Doubles?

- To create “small” tests that are faster and less flaky
  - Example: Testing a unit that processes result of an external API call; only interested in testing what happens *after* the external call returns
- When the real thing is unavailable
  - Example: Integrating with external vendors
- When testing for unusual or exceptional cases that are hard to make happen in practice
  - Example: when external service fails in the middle of a transaction

Test Doubles Intercept Calls to Methods

- Testing frameworks provide two common abstractions for doubles
  - Transparently modifies programs while running to intercept calls
- **Spies** invoke the original method, but record the parameters and call information
- **Mocks** do not invoke the original method
  - Default is to provide canned responses (Jest picks: `undefined`)
  - Also can provide a mock implementation to entirely replace the original method
- Other frameworks use terms like “fake” and “stub” for variants of these; we focus on Jest’s features (spies, mocks)

Test Spy is a stub that remembers how the object was called

- Test can check what happened earlier;
  - For example: a particular method should be called
    1. First with parameters “foo” and 42;
    2. Then with parameters “quux” and -88.
- A spy can be useful in conjunction with the “real” environment:
  - What was sent on the network?
  - How many times a problem was logged?
  - What was inserted in the database?

Spy “remembers”

Example: Test Spies in IP2

- useConversationAreaOccupants should call the method *addListener* when rendered, and on cleanup, call *removeListener* with the same exact argument
- Our test for this requirement uses two *spies* to inspect calls to these methods

```
beforeEach(() => {
  conversationAreaController = new ConversationAreaController(nanoid(), nanoid());
  addListenerSpy = jest.spyOn(conversationAreaController, 'addListener');
  removeListenerSpy = jest.spyOn(conversationAreaController, 'removeListener');
});
```

Before each test: create a ConversationAreaController to test with the hook, spy on its addListener and removeListener methods

```
it('Removes its update listener when the component unmounts', () => {
  const listenerAdded = getSingleListenerAdded('occupantsChange');
  cleanup();
  const listenerRemoved = getSingleListenerRemoved('occupantsChange');
  expect(listenerAdded).toBe(listenerRemoved);
});
```

Test that the listener added is the exact same listener removed, getSingleListenerAdded/removed uses spy.mock.calls to find the arguments passed to addListener

Test Mock is a Double that has Scripted results

- A test mock has scripted results:
  - If such-and-such a method is called
    - return some particular value.
- A complex mock can have many scripts:
  - Multiple methods;
  - Different results for subsequent calls.
- Useful mocking assumes we know how mocked object will be used.
- Jest's default behavior is to return "undefined", we can modify this

Mock has "scripted answers" and is used for "behavior verification"

Jest supports Mocks

Jest's Mock API: https://jestjs.io/docs/mock-function-api

- Replacing TwilioVideo with Mock

```
const mockTwilioVideo = mockDeep<TwilioVideo>();
jest.spyOn(TwilioVideo, 'getInstance').mockReturnValue(mockTwilioVideo);
```

You will see more of these in IP2

- Jest Tests can be written

```
it('should use the coveyTownID and player ID properties when requesting a video token',
      async () => {
        const townName = `FriendlyNameTest-${nanoid()}`;
        const townController = new CoveyTownController(townName, false);
        const newPlayerSession = await townController.addPlayer(new Player(nanoid()));
        expect(mockTwilioVideo.getTokenForTown).toBeCalledTimes(1);
        expect(mockTwilioVideo.getTokenForTown).toBeCalledWith(townController.coveyTownID, newPlayerSession.player.id);
      });
```

Here is another Example of Mock /1

```
describe('conversationAreaCreateHandler', () => {

    const mockCoveyTownStore = mock<CoveyTownsStore>();
    const mockCoveyTownController = mock<CoveyTownController>();
    beforeAll(() => {
      // Set up a spy for CoveyTownsStore that will always return our mockCoveyTownsStore as the
singleton instance
      jest.spyOn(CoveyTownsStore, 'getInstance').mockReturnValue(mockCoveyTownStore);
    });
    beforeEach(() => {
      // Reset all mock calls, and ensure that getControllerForTown will always return the same mock
controller
      mockReset(mockCoveyTownController);
      mockReset(mockCoveyTownStore);
      mockCoveyTownStore.getControllerForTown.mockReturnValue(mockCoveyTownController);
    });
. . . .
```

spying on getInstance() method

Here is another Example of Mock /2

```
. . . .
    it('Checks for a valid session token before creating a conversation area', ()=>{
      const coveyTownID = nanoid();
      const conversationArea :ServerConversationArea = { boundingBox: { height: 1, width: 1, x:1, y:1 }, label: nanoid(),
occupantsByID: [], topic: nanoid() };
      const invalidSessionToken = nanoid();
      // Make sure to return 'undefined' regardless of what session token is passed
      mockCoveyTownController.getSessionByToken.mockReturnValueOnce(undefined);
      requestHandlers.conversationAreaCreateHandler({
        conversationArea,
        coveyTownID,
        sessionToken: invalidSessionToken,
      });
      expect(mockCoveyTownController.getSessionByToken).toBeCalledWith(invalidSessionToken);
      expect(mockCoveyTownController.addConversationArea).not.toHaveBeenCalled();
   });
});
```

If SessionToken is invalid, don't call addConversationArea()

Supply Implementation to Mocks to Simulate Behaviors

- Sometimes called a *fake*, these mocks have an implementation of the object being replaced
  - A *low-fidelity* fake implements things partially
    - Enough to work for the test.
  - A *high-fidelity* fake implements most aspects:
    - Usually all functional aspects;
    - Usually not as efficiently or as scalable.
- The purpose of this mock is to avoid processes/network/cost, but still perform some activities
- Create fakes in Jest with *mock.mockImplementation(…)*

Fake has "semi-real implementation"

Testing Large Systems is Hard

- What to do if the specification is incomplete, and likely to change frequently?
  - Writing thorough test suite is even harder, less useful
- Still: vital to detect breaking changes
- Examples:
  - Detailed layout of GUIs
  - Side-effects of APIs, particularly under corner-cases

Snapshot GUI Tests Detect Changes

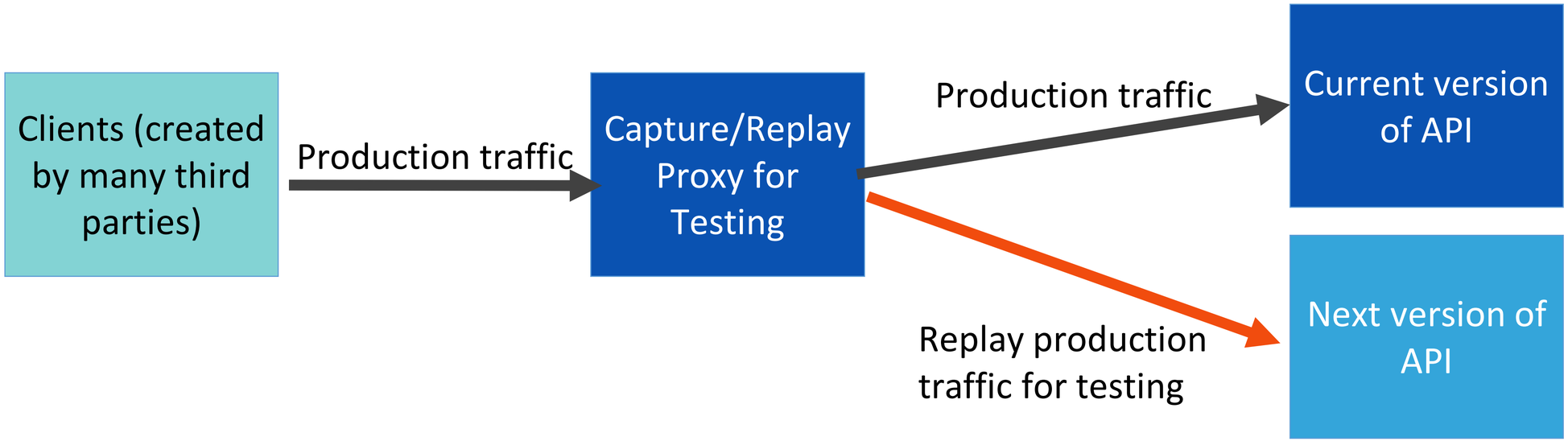
- The first time the test runs, it saves a “snapshot” of the rendered GUI
- Subsequent runs will fail if the snapshot changes

```
import renderer from 'react-test-renderer';
import Link from '../Link';

it('renders correctly', () => {
  const tree = renderer
    .create(<Link
page="http://www.facebook.com">Facebook</Li
nk>)
      .toJSON();
  expect(tree).toMatchSnapshot();
});
```

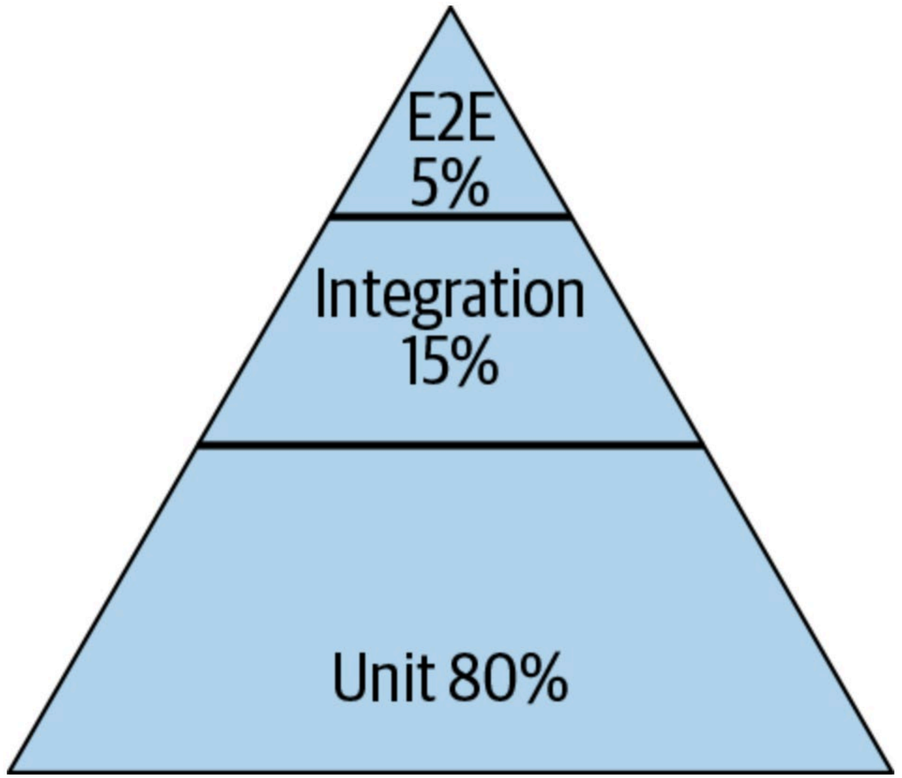
Capture/Replay of API Traffic Detects Breaking Changes

- Record the API requests and responses that clients make
- Test new versions of the API by identifying requests that result in different responses (“breaking changes”)



Test Doubles Have Weaknesses

- Some failures may occur purely at the integration between components:
  - The test may assume wrong behavior (wrongly encoded by mock)
  - Higher fidelity mocks (e.g. capture/replay) can help, but still just a snapshot of the real world
- The SUT may use a different algorithm:
  - The Spies expect a particular usage of double;
  - The test is “brittle” because it depends on internal behavior of SUT;
- Potential maintenance burden: as SUT evolves, mocks must evolve
  - Capture/replay is a bit less, at least…



Review: Learning Objectives for this Lesson

- You should now be able to:
  - Explain why you might need a "test double“ in your testing
  - Understand how and when to apply different kinds of test “doubles” such as “mocks and spies”

- For Further Reading
  - Check out Martin Fowler’s article, “Mocks Aren’t Stubs” https://martinfowler.com/articles/mocksArentStubs.html
  - “xUnit Test Patterns: Refactoring Test Code” by Gerard Meszaros