Testing Web Apps

SENG 4640
Software Engineering for Web Apps
Winter 2023

Sina Keshvadi
Thompson Rivers University

Testing WebApps

- Testing is the process of exercising a WebApp with the intent of finding (and ultimately correcting) errors.
- Tests must be design to uncover errors in WebApps that are implemented in:
  - different operating systems
  - browsers [or other interface devices such as set-top boxes, personal digital assistants (PDAs), and mobile phones]
  - hardware platforms
  - communications protocols

The "Dimensions" of Quality - I

- Reviews and testing examine one or more of the following quality dimensions:
  - *Content* is evaluated at both a syntactic and semantic level.
    - At the syntactic level, spelling, punctuation, and grammar are assessed for text-based documents.
    - At a semantic level, correctness (of information presented), consistency (across the entire content object and related objects), and lack of ambiguity are all assessed.
  - *Function* is tested to uncover errors that indicate lack of conformance to stakeholder requirements. Each WebApp function is assessed for correctness, instability, and general conformance to appropriate implementation standards (e.g., Java or XML language standards).
  - *Structure* is assessed to ensure that it properly delivers WebApp content and function, is extensible, and can be supported as new content or functionality is added.

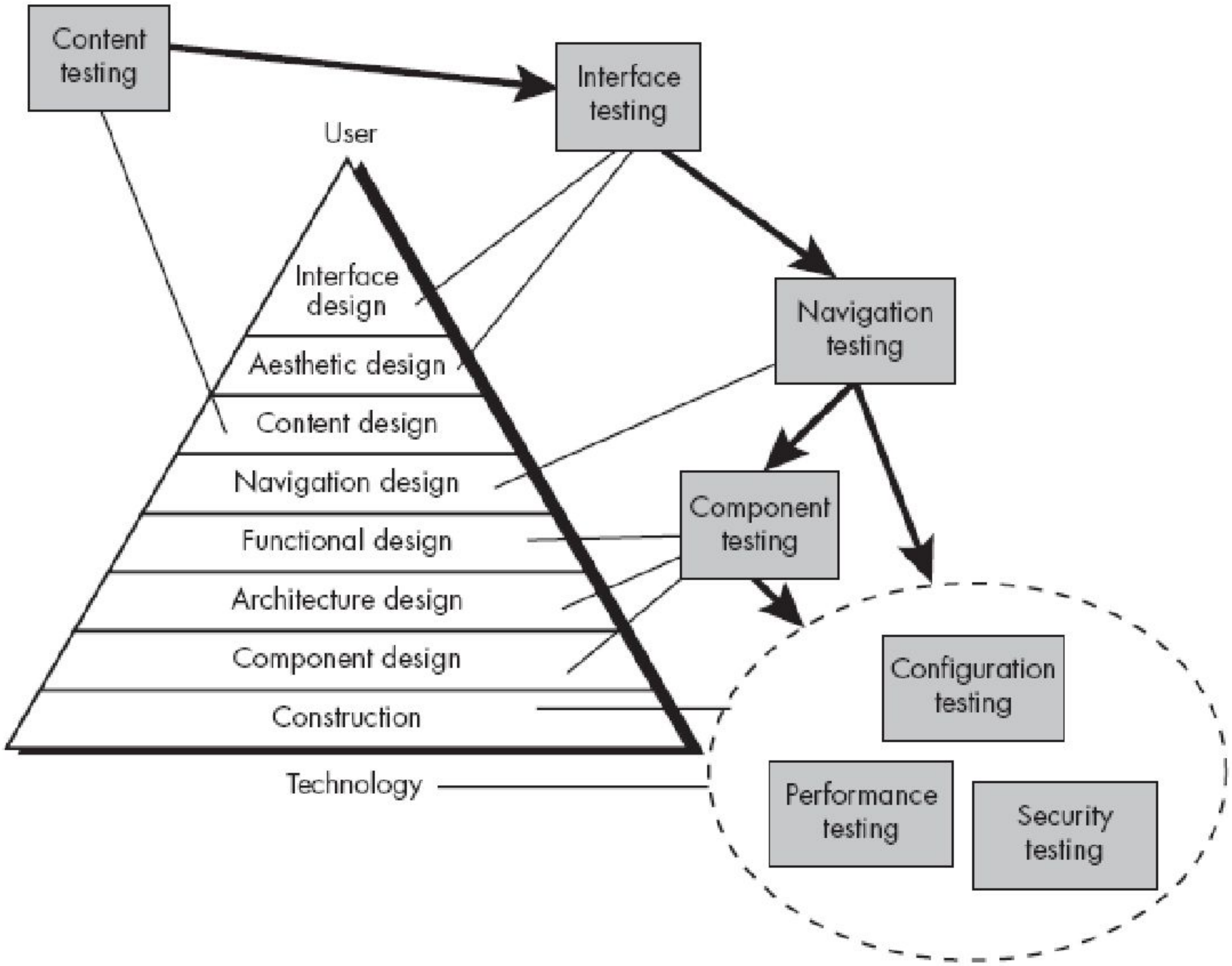
The “Dimensions” of Quality - II

- *Usability* is tested to ensure that each category of user is supported by the interface and can learn and apply all required navigation syntax and semantics.
- *Navigability* is tested to ensure that all navigation syntax and semantics are exercised to uncover any navigation errors (e.g., dead links, improper links, erroneous links).
- *Performance* is tested under a variety of operating conditions, configurations, and loading to ensure that the system is responsive to user interaction and handles extreme loading without unacceptable operational degradation.
- *Compatibility* is tested by executing the WebApp in a variety of different host configurations on both the client and server sides. The intent is to find errors that are specific to a unique host configuration.
- *Interoperability* is tested to ensure that the WebApp properly interfaces with other applications and/or databases.
- *Security* is tested by assessing potential vulnerabilities and attempting to exploit each. Any successful penetration attempt is deemed a security failure.

Testing Strategy

1. The content model for the WebApp is reviewed to uncover errors.
2. The interface model is reviewed to ensure that all use cases have been accommodated.
3. The design model for the WebApp is reviewed to uncover navigation errors.
4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Selected functional components are unit tested.
6. Navigation throughout the architecture is tested.
7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
9. Performance tests are conducted.
10. The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

The Testing Process



Content Testing

- Content testing combines both reviews and the generation of executable test cases.
  - Reviews are applied to uncover semantic errors in content.
  - Executable testing is used to uncover content errors that can be traced to dynamically derived content that is driven by data acquired from one or more databases.
- Content testing has three important objectives:
  - to uncover syntactic errors (e.g., typos, grammar mistakes) in text-based documents, graphical representations, and other media,
  - to uncover semantic errors (i.e., errors in the accuracy or completeness of information) in any content object presented as navigation occurs, and
  - to find errors in the organization or structure of content that is presented to the end user.

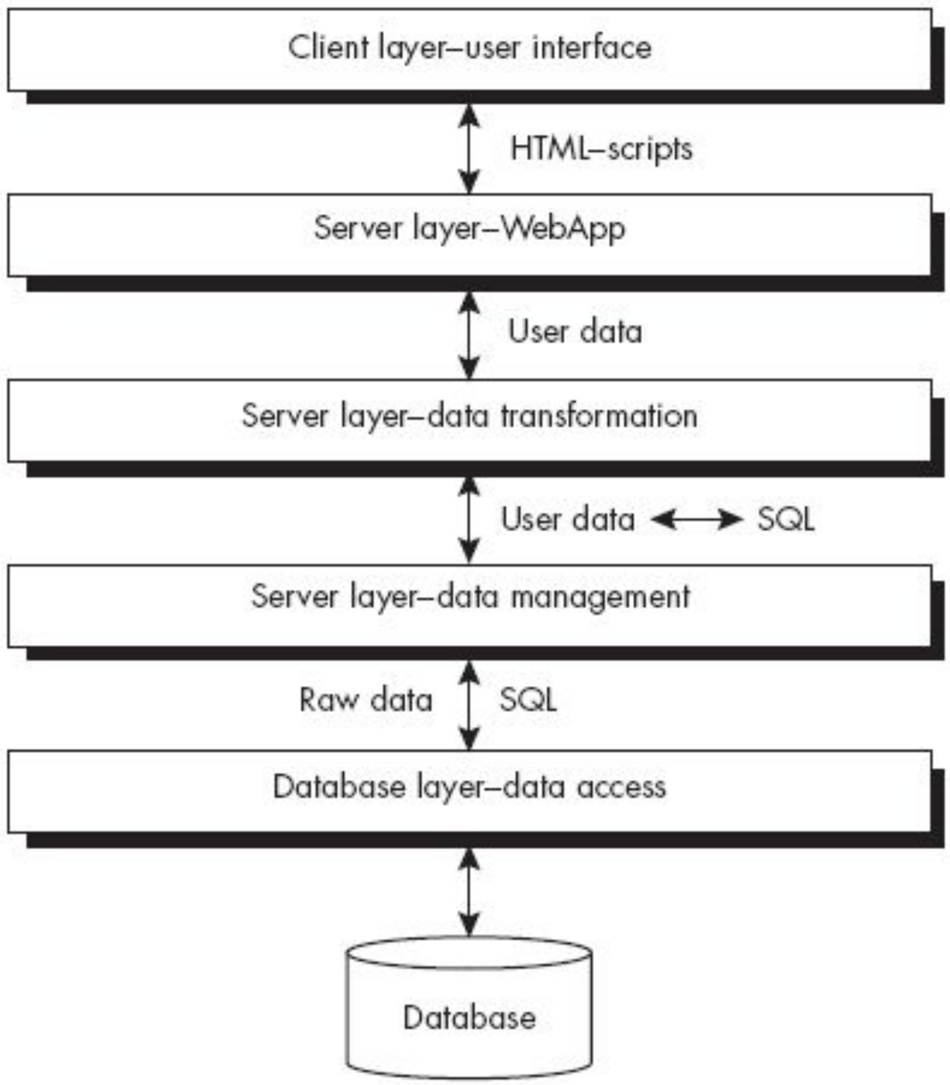
Content Testing - Checklist

- Is the information up to date and factually accurate?
- Is the information concise and to the point?
- Is the layout of the content object easy for the user to understand?
- Can information embedded within a content object be found easily?
- Have proper references been provided for all information derived from other sources?
- Is the information presented consistent internally and consistent with information presented in other content objects?
- Can the content be interpreted as being offensive or misleading, or does it open the door to litigation?
- Does the content infringe on existing copyrights or trademarks?
- Does the content contain internal links that supplement existing content? Are the links correct?
- Does the aesthetic style of the content conflict with the aesthetic style of the interface?

Content Testing – Dynamic Content

- When content is created dynamically using information maintained within a database, the following issues are considered:
  - The original client-side request for information is rarely presented in the form [e.g., structured query language (SQL)] that can be input to a database management system (DBMS).
  - The database may be remote to the server that houses the WebApp.
    - *What happens if the WebApp is accessible but the database is not?*
  - Raw data acquired from the database must be transmitted to the WebApp server and properly formatted for subsequent transmittal to the client.
  - The dynamic content object(s) must be transmitted to the client in a form that can be displayed to the end user.

Content Testing - Database


Client layer–user interface
HTML–scripts
Server layer–WebApp
User data
Server layer–data transformation
User data
SQL
Server layer–data management
Raw data
SQL
Database layer–data access
Database

User Interface Testing

- Verification and validation of a WebApp user interface occurs at three distinct points in the WebE process.
  - During communication and modeling , the interface model is reviewed to ensure that it conforms to customer requirements and to other elements of the analysis model.
  - During design, the interface design model is reviewed to ensure that generic quality criteria established for all user interfaces have been achieved and that application-specific interface design issues have been properly addressed.
  - During testing, the focus shifts to the execution of application-specific aspects of user interaction as they are manifested by interface syntax and semantics. In addition, testing provides a final assessment of usability.

UI Testing Strategy

- Interface features are tested to ensure that design rules, aesthetics, and related visual content are available to the user without error.
- Individual interface mechanisms are tested in a manner that is analogous to unit testing.
- Each interface mechanism is tested within the context of a use case or navigation pathway for a specific user category.
- The complete interface is tested against selected use cases and navigation pathways to uncover errors in the semantics of the interface.
- The interface is tested within a variety of environments (e.g., operating systems, browsers) to ensure that it will be compatible.

Usability Testing

- Similar to interface semantics testing in the sense that it evaluates:
  - the degree to which users can interact effectively with the WebApp
  - the degree to which the WebApp guides users' actions, provides meaningful feedback and enforces a consistent interaction approach.
- Determines the degree to which the WebApp interface makes the user's life easy

Usability Test Categories

- **Interactivity.** Are interaction mechanisms (e.g., pull-down menus, buttons, pointers) easy to understand and use?
- **Layout.** Are navigation mechanisms, content, and functions placed in a manner that allows the user to find them quickly?
- **Readability.** Is text well written and understandable? Are graphic representations intuitive and easy to understand?
- **Aesthetics.** Do the layout, color, typeface, and related characteristics lead to ease of use? Do users "feel comfortable" with the look and feel of the WebApp?
- **Display characteristics.** Does the WebApp make optimal use of screen size and resolution?
- **Time sensitivity.** Can important features, functions, and content be used or acquired in a timely manner?
- **Personalization.** Does the WebApp appropriately tailor itself to the specific needs of different user categories or individual users?

Usability Evaluation: Checklist

- ❑ Is the system usable without continual help or instruction?
- ❑ Do the rules of interaction help a knowledgeable user to work efficiently?
- ❑ Do interaction mechanisms become more flexible as users become more knowledgeable?
- ❑ Has the system been tuned to the physical and social environment in which it will be used?
- ❑ Are users aware of the state of the system? Do users know where they are at all times?
- ❑ Is the interface structured in a logical and consistent manner?
- ❑ Are interaction mechanisms, icons, and procedures consistent across the interface?
- ❑ Does the interaction anticipate errors and help the user correct them?
- ❑ Is the interface tolerant of errors that are made?
- ❑ Is the interaction simple?

Compatibility Testing

- WebApps operate in complex (and often unpredictable) environments
  - Different browsers, screen resolutions, operating systems, plug-ins, access bandwidths, etc.
- Serious errors can be caused by obscure combinations
- Most common problem is deterioration in usability:
  - Download speeds may become unacceptable
  - Missing plug-ins may make content unavailable
  - Browser differences can change page layout or legibility
  - Forms may be improperly organized.
- Compatibility testing strives to uncover these problems before the WebApp goes online.
  - First step is to define a set of "commonly encountered" client-side configurations and their variants.
  - Next, derive a series of compatibility validation tests (from existing interface tests, navigation tests, performance tests, and security tests).

Configuration Testing

- Configuration variability and instability are important factors that make Web engineering a challenge.
    - Hardware, operating system(s), browsers, storage capacity, network communication speeds, and a variety of other client-side factors are difficult to predict for each user.
- The job of configuration testing is to test a set of probable client-side and server-side configurations to ensure that the user experience will be the same on all of them and to isolate errors that may be specific to a particular configuration.

Testing Strategy

- **Server-side.** configuration test cases are designed to verify that the projected server configuration [i.e., WebApp server, database server, operating system(s), firewall software, concurrent applications] can support the WebApp without error.
- **Client-side.** On the client side, configuration tests focus more heavily on WebApp compatibility with configurations that contain one or more permutations of the following components:
  - **Hardware.** CPU, memory, storage, and printing devices
  - **Operating systems.** Linux, Macintosh OS, Microsoft Windows, a mobile-based OS
  - **Browser software.** FireFox, Internet Explorer, Safari, Mozilla/Netscape, Opera, and others
  - **User interface components.** Active X, Java applets, and others
  - **Plug-ins.** QuickTime, RealPlayer, and many others
  - **Connectivity.** Cable, DSL, regular modem, industry-grade connectivity

Security and Performance Testing

- Security and performance testing address the three distinct elements of the WebApp infrastructure
  - the server-side environment that provides the gateway to Internet users
  - the network communication pathway between the server and the client machine
  - the client-side environment that provides the end user with a direct interface to the WebApp.
- **Security testing** focuses on unauthorized access to WebApp content and functionality along with other systems that cooperate with the WebApp on the server side.
- **Performance testing** focuses on the operating characteristics of the WebApp and on whether those operating characteristics meet the needs of end users.

Security Testing

- One or more of the following security elements is implemented [Ngu01]:
    - **Firewalls.** A filtering mechanism that is a combination of hardware and software that examines each incoming packet of information to ensure that it is coming from a legitimate source, blocking any data that are suspect.
    - **Authentication.** A verification mechanism that validates the identity of all clients and servers, allowing communication to occur only when both sides are verified.
    - **Encryption.** An encoding mechanism that protects sensitive data by modifying it in a way that makes it impossible to read by those with malicious intent. Encryption is strengthened by using *digital certificates* that allow the client to verify the destination to which the data are transmitted.
    - **Authorization.** A filtering mechanism that allows access to the client or server environment only by those individuals with appropriate authorization codes (e.g., user ID and password).
- Security tests should be designed to probe each of these security technologies in an effort to uncover security holes that can be exploited by those with malicious intent.

Performance Testing

- Objectives:
  - Does the server response time degrade to a point where it is noticeable and unacceptable?
  - At what point (in terms of users, transactions, or data loading) does performance become unacceptable?
  - What system components are responsible for performance degradation?
  - What is the average response time for users under a variety of loading conditions?
  - Does performance degradation have an impact on system security?
  - Is WebApp reliability or accuracy affected as the load on the system grows?
  - What happens when loads that are greater than maximum server capacity are applied?
  - What is the impact of poor performance on company revenues?
- *Load testing* determines how the WebApp and its server-side environment will respond to various loading conditions.
- *Stress testing* is a continuation of load testing, but in this instance the variables, *N*, *T*, and *D* are forced to meet and then exceed operational limits.

Testing React Apps

- **Mocha** – widely used test runner (testing framework) used to run JavaScript tests
- **Chai** – assertion library for Behavior Driven Testing
- **Enzyme** – testing utility for React for manipulating and inspecting React Component state and output

Getting Started - Installation

- To include Enzyme and Chai as dependencies, run the following command:

```
npm install --save-dev enzyme react-test-renderer chai
```

```
npm install --force --save-dev @wojtekmaj/enzyme-adapter-react-17
```

```
Use the --force command since React 18 is a new version, and the Enzyme
library has not yet been published for this version.
```

Getting Started - Installation

- To include Enzyme and Chai as dependencies, run the following command:

```
npm install --save-dev enzyme react-test-renderer chai
```

```
npm install --force --save-dev @wojtekmaj/enzyme-adapter-react-17
```

- Note that the default file structure places all JavaScript and CSS code in the ‘`src`’ folder.
- We will create an additional folder within ‘`src`’ named ‘`tests`’ in which we include all testing scripts
- All test files must be in the form of `*`**`.test.js`**, e.g. **`Dogs.test.js`**

Getting Started

- Node.js should create a default `App.test.js`, or you can write your own

Getting Started

- Node.js should create a default **`App.test.js`**, or you can write your own
- Include libraries necessary for testing
  - Import React and ReactDOM for component manipulation

```
import React from 'react';
import ReactDOM from 'react-dom';
```

  - Import keywords from Enzyme

```
import { mount, shallow } from 'enzyme';
```

  - Import keywords from Chai

```
import {expect} from 'chai';
```

Import this headers in file.test.js

```
import React from "react";
import ReacDOM from "react-dom";

import { expect } from "chai";
import {mount, shallow } from "enzyme";
import Enzyme from "enzyme";
import Adapter from
   "@wojtekmaj/enzyme-adapter-react-17";
Enzyme.configure({ adapter: new Adapter() });
import App from "../App";
```

Important: This is a new change, and I will not modify the upcoming slides, but instead use this headers consistently.

Getting Started

- Node.js should create a default `App.test.js`, or you can write your own
- Include libraries necessary for testing
  - Import React and ReactDOM for component manipulation

```
import React from 'react';
import ReactDOM from 'react-dom';
```

  - Import keywords from Enzyme

```
import { mount, shallow } from 'enzyme';
```

  - Import keywords from Chai

```
import {expect} from 'chai';
```

Getting Started

- Node.js should create a default **`App.test.js`**, or you can write your own
- Include libraries necessary for testing
  - Import React and ReactDOM for component manipulation

```
import React from 'react';
import ReactDOM from 'react-dom';
```

  - Import keywords from Enzyme

```
import { mount, shallow } from 'enzyme';
```

  - Import keywords from Chai

```
import {expect} from 'chai';
```

The curly braces indicate we only want to import the 'mount' and 'shallow' functions from the 'enzyme' module, rather than the entire module.

Getting Started

- Node.js should create a default **`App.test.js`**, or you can write your own
- Include libraries necessary for testing
  - Import React and ReactDOM for component manipulation

```
import React from 'react';
import ReactDOM from 'react-dom';
```

  - Import keywords from Enzyme

```
import { mount, shallow } from 'enzyme';
```

  - Import keywords from Chai

```
import {expect} from 'chai';
```

Anatomy of a React Test

```
 import React from 'react';
 import { expect } from 'chai';
 import { mount, shallow } from 'enzyme';
 import App from '../App';

 describe("Test suite for App component", function(){

   it("only one element in App class", function() {

        const wrapper = shallow(<App />);
        expect(wrapper.find(".App")).length(1);
   });
});
```

Anatomy of a React Test

```
import React from 'react';
import { expect } from 'chai';
import { mount, shallow } from 'enzyme';
import App from '../App';

describe("Test suite for App component", function(){

  it("only one element in App class", function() {

      const wrapper = shallow(<App />);
      expect(wrapper.find(".App")).length(1);
  });
});
```

The component that we are testing

Anatomy of a React Test

```
 import React from 'react';
 import { expect } from 'chai';
 import { mount, shallow } from 'enzyme';
 import App from '../App';

 describe("Test suite for App component", function(){

   it("only one element in App class", function() {

       const wrapper = shallow(<App />);
       expect(wrapper.find(".App")).length(1);
   });
});
```

describe a test suit (a collection of test cases) using Mocha

Anatomy of a React Test

```
 import React from 'react';
 import { expect } from 'chai';
 import { mount, shallow } from 'enzyme';
 import App from '../App';

 describe("Test suite for App component", function(){

   it("only one element in App class", function() {

        const wrapper = shallow(<App />);
        expect(wrapper.find(".App")).length(1);
   });
});
```

a message that describe what test suit does

Anatomy of a React Test

```
 import React from 'react';
 import { expect } from 'chai';
 import { mount, shallow } from 'enzyme';
 import App from '../App';

 describe("Test suite for App component", function(){

   it("only one element in App class", function() {

        const wrapper = shallow(<App />);
        expect(wrapper.find(".App")).length(1);
   });
});
```

to write an individual test case, we use keyword (function) ***it***

Anatomy of a React Test

```
import React from 'react';
import { expect } from 'chai';
import { mount, shallow } from 'enzyme';
import App from '../App';

describe("Test suite for App component", function(){

  it("only one element in App class", function() {

      const wrapper = shallow(<App />);
      expect(wrapper.find(".App")).length(1);
  });
});
```

```
function it the test itself
```

Anatomy of a React Test

```
 import React from 'react';
 import { expect } from 'chai';
 import { mount, shallow } from 'enzyme';
 import App from '../App';

 describe("Test suite for App component", function(){

   it("only one element in App class", function() {

       const wrapper = shallow(<App />);
       expect(wrapper.find(".App")).length(1);
   });
});
```

shallow (imported from enzyme) create a shallow component (without its childs)

Anatomy of a React Test

```
 import React from 'react';
 import { expect } from 'chai';
 import { mount, shallow } from 'enzyme';
 import App from '../App';

 describe("Test suite for App component", function(){

   it("only one element in App class", function() {

       const wrapper = shallow(<App />);
       expect(wrapper.find(".App")).length(1);
   });
});
```

```
expect keyword: what we expect to happen
```

Testing React Component Relationships

```
it('Dog List contains two dogs', function() {
  const wrapper = mount(<App/>);
  expect(wrapper.find('Dogs')

               .find('DogItem')).length(2);
});
```

this could be in the same test file (test suit) or in a different one.

Testing React Component Relationships

```
it('Dog List contains two dogs', function() {
  const wrapper = mount(<App/>);
  expect(wrapper.find('Dogs')

                  .find('DogItem')).length(2);
});
```

unlike shallow, mount create a deep creation (app component and its child(s))

Testing React Component Relationships

```
it('Dog List contains two dogs', function() {
  const wrapper = mount(<App/>);
  expect(wrapper.find('Dogs')
                 .find('DogItem')).length(2);
});
```

find two dog items

Now, let’s see how we can Simulate user interaction with the App

Testing Data Entry and Form Submission

```
it("successfully adds dog to list when form submitted",
   function() {
     const wrapper = mount(<App/>);
     const adddog = wrapper.find('AddDog');

     adddog.find('#dogName').at(0).getDOMNode().value = 'Lola';
     adddog.find('#imageURL').at(0).getDOMNode().value =
              'https:// static.pexels.com/photos/54386/pexels-
              photo-54386.jpeg';
     adddog.find('#dogBreed').at(0).getDOMNode().value =
     'Beagle';

     const form = adddog.find('form');
     form.simulate('submit');
     expect(wrapper.find('Dogs')
                   .find('DogItem')).length(3);
     expect(wrapper.state().dogs[2].name == 'Lola');
});
```

```
it("successfully adds dog to list when form submitted",
   function() {
      const wrapper = mount(<App/>);
      const adddog = wrapper.find('AddDog');

      adddog.find('#dogName').at(0).getDOMNode().value = 'Lola';
      adddog.find('#imageURL').at(0).getDOMNode().value =
               'https:// static.pexels.com/photos/54386/pexels-
               photo-54386.jpeg';
      adddog.find('#dogBreed').at(0).getDOMNode().value =
      'Beagle';

      const form = adddog.find('form');
      form.simulate('submit');
      expect(wrapper.find('Dogs')
                    .find('DogItem')).length(3);
      expect(wrapper.state().dogs[2].name == 'Lola');
});
```

Here we test if our app successfully adds a dog to the list when we submit the form.

```
it("successfully adds dog to list when form submitted",
   function() {
      const wrapper = mount(<App/>);
      const adddog = wrapper.find('AddDog');

      adddog.find('#dogName').at(0).getDOMNode().value = 'Lola';
      adddog.find('#imageURL').at(0).getDOMNode().value =
               'https:// static.pexels.com/photos/54386/pexels-
               photo-54386.jpeg';
      adddog.find('#dogBreed').at(0).getDOMNode().value =
      'Beagle';

      const form = adddog.find('form');
      form.simulate('submit');
      expect(wrapper.find('Dogs')
                     .find('DogItem')).length(3);
      expect(wrapper.state().dogs[2].name == 'Lola');
});
```

mount the App component and use that component to find the addDog component. addDog was the component that allows users to add a new dog.

```
it("successfully adds dog to list when form submitted",
    function() {
      const wrapper = mount(<App/>);
      const adddog = wrapper.find('AddDog');

      adddog.find('#dogName').at(0).getDOMNode().value = 'Lola';
      adddog.find('#imageURL').at(0).getDOMNode().value =
               'https:// static.pexels.com/photos/54386/pexels-
               photo-54386.jpeg';
      adddog.find('#dogBreed').at(0).getDOMNode().value =
      'Beagle';

      const form = adddog.find('form');
      form.simulate('submit');
      expect(wrapper.find('Dogs')
                     .find('DogItem')).length(3);
      expect(wrapper.state().dogs[2].name == 'Lola');
});
```

```
set Lola to the HTML element that has id=dogName
get(0) is for the case that we have (shouldn’t) more than one elements with
#dogName id
```

```
it("successfully adds dog to list when form submitted",
   function() {
     const wrapper = mount(<App/>);
     const adddog = wrapper.find('AddDog');

     adddog.find('#dogName').at(0).getDOMNode().value = 'Lola';
     adddog.find('#imageURL').at(0).getDOMNode().value =
              'https:// static.pexels.com/photos/54386/pexels-
              photo-54386.jpeg';
     adddog.find('#dogBreed').at(0).getDOMNode().value =
     'Beagle';

     const form = adddog.find('form');
     form.simulate('submit');
     expect(wrapper.find('Dogs')
                    .find('DogItem')).length(3);
     expect(wrapper.state().dogs[2].name == 'Lola');
});
```

Simulate the rest of the form

```
it("successfully adds dog to list when form submitted",
    function() {
      const wrapper = mount(<App/>);
      const adddog = wrapper.find('AddDog');

      adddog.find('#dogName').at(0).getDOMNode().value = 'Lola';
      adddog.find('#imageURL').at(0).getDOMNode().value =
               'https:// static.pexels.com/photos/54386/pexels-
               photo-54386.jpeg';
      adddog.find('#dogBreed').at(0).getDOMNode().value =
      'Beagle';

      const form = adddog.find('form');
      form.simulate('submit');
      expect(wrapper.find('Dogs')
                     .find('DogItem')).length(3);
      expect(wrapper.state().dogs[2].name == 'Lola');
});
```

simulate the submission the form (no need to simulate the add button)

```
it("successfully adds dog to list when form submitted",
    function() {
      const wrapper = mount(<App/>);
      const adddog = wrapper.find('AddDog');

      adddog.find('#dogName').at(0).getDOMNode().value = 'Lola';
      adddog.find('#imageURL').at(0).getDOMNode().value =
               'https:// static.pexels.com/photos/54386/pexels-
               photo-54386.jpeg';
      adddog.find('#dogBreed').at(0).getDOMNode().value =
      'Beagle';

      const form = adddog.find('form');
      form.simulate('submit');
      expect(wrapper.find('Dogs')
                       .find('DogItem')).length(3);
      expect(wrapper.state().dogs[2].name == 'Lola');
});
```

now we expect to have three dog items (originally was 2)

```
it("successfully adds dog to list when form submitted",
    function() {
      const wrapper = mount(<App/>);
      const adddog = wrapper.find('AddDog');

      adddog.find('#dogName').at(0).getDOMNode().value = 'Lola';
      adddog.find('#imageURL').at(0).getDOMNode().value =
               'https:// static.pexels.com/photos/54386/pexels-
               photo-54386.jpeg';
      adddog.find('#dogBreed').at(0).getDOMNode().value =
      'Beagle';

      const form = adddog.find('form');
      form.simulate('submit');
      expect(wrapper.find('Dogs')
                     .find('DogItem')).length(3);
      expect(wrapper.state().dogs[2].name == 'Lola');
});
```

state() function get access to the app component’s state

Let's see what happens when we try to delete one of the dogs by clicking on the link that was next to the dog's breed.

Testing Links

```
it('removes dog from list when deleted', function() {
    const wrapper = mount(<App/>);
    const deleteLink = wrapper.find('a').first();

    deleteLink.simulate('click');

    expect(wrapper.find('Dogs')
                  .find('DogItem')).length(1);
});
```

Testing Links

```
it('removes dog from list when deleted', function() {
    const wrapper = mount(<App/>);
    const deleteLink = wrapper.find('a').first();

    deleteLink.simulate('click');

    expect(wrapper.find('Dogs')
                  .find('DogItem')).length(1);
});
```

```
find a link using a for the anchor
We'll get the first link that we find, this will be the delete link
```

Testing Links

```
it('removes dog from list when deleted', function() {
    const wrapper = mount(<App/>);
    const deleteLink = wrapper.find('a').first();

    deleteLink.simulate('click');

    expect(wrapper.find('Dogs')
                   .find('DogItem')).length(1);
});
```

simulate clicking on that link

Testing Links

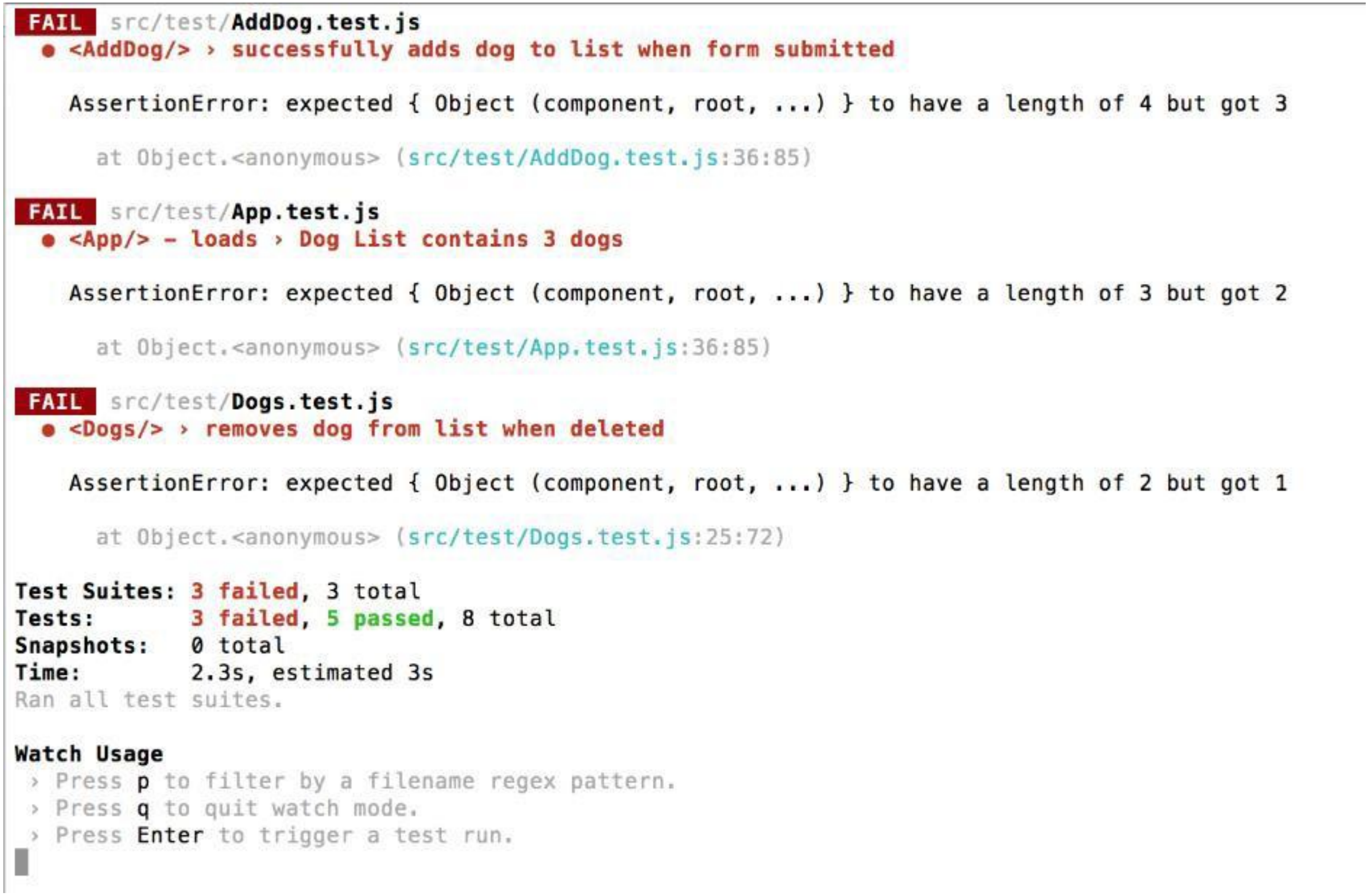
```
it('removes dog from list when deleted', function() {
    const wrapper = mount(<App/>);
    const deleteLink = wrapper.find('a').first();

    deleteLink.simulate('click');

    expect(wrapper.find('Dogs')
                  .find('DogItem')).length(1);
});
```

```
now we expect to have 1 dog
note that each test case starts with the original state
```

Running Tests

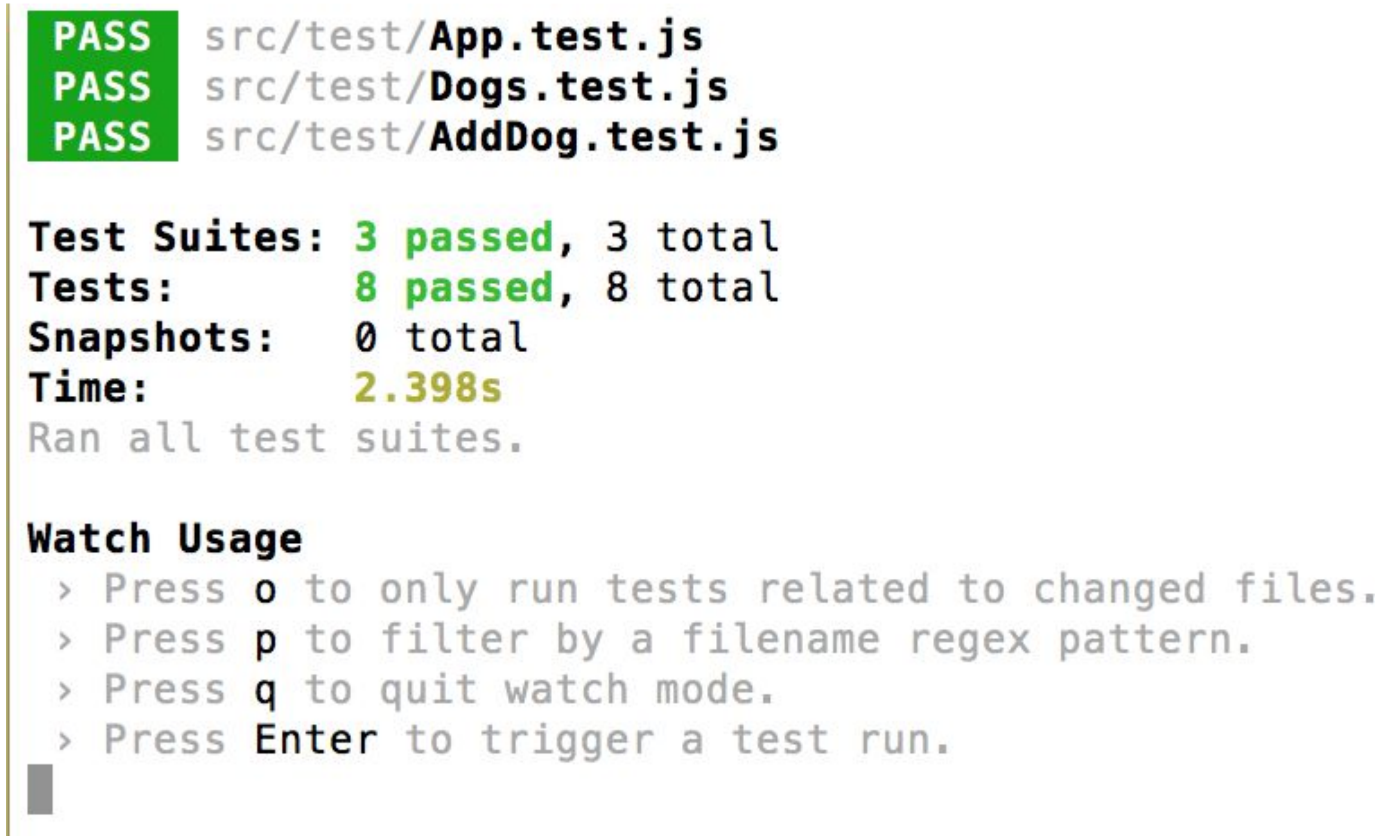
- To run tests, navigate to the project within the terminal and run the following command:

```
npm run test
```



```
FAIL  src/test/AddDog.test.js
  ● <AddDog/> › successfully adds dog to list when form submitted

    AssertionError: expected { Object (component, root, ...) } to have a length of 4 but got 3

      at Object.<anonymous> (src/test/AddDog.test.js:36:85)

FAIL  src/test/App.test.js
  ● <App/> - loads › Dog List contains 3 dogs

    AssertionError: expected { Object (component, root, ...) } to have a length of 3 but got 2

      at Object.<anonymous> (src/test/App.test.js:36:85)

FAIL  src/test/Dogs.test.js
  ● <Dogs/> › removes dog from list when deleted

    AssertionError: expected { Object (component, root, ...) } to have a length of 2 but got 1

      at Object.<anonymous> (src/test/Dogs.test.js:25:72)

Test Suites: 3 failed, 3 total
Tests:       3 failed, 5 passed, 8 total
Snapshots:   0 total
Time:        2.3s, estimated 3s
Ran all test suites.

Watch Usage
 › Press p to filter by a filename regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Running Tests: Success!


```
 PASS  src/test/App.test.js
 PASS  src/test/Dogs.test.js
 PASS  src/test/AddDog.test.js

Test Suites: 3 passed, 3 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        2.398s
Ran all test suites.

Watch Usage
 › Press o to only run tests related to changed files.
 › Press p to filter by a filename regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Summary

- We can use **Node.js** to create React applications
- This allows us to put component code into separate .js files and then include them into our App as necessary
- **Mocha**, **Chai**, and **Enzyme** can be used for testing our React apps