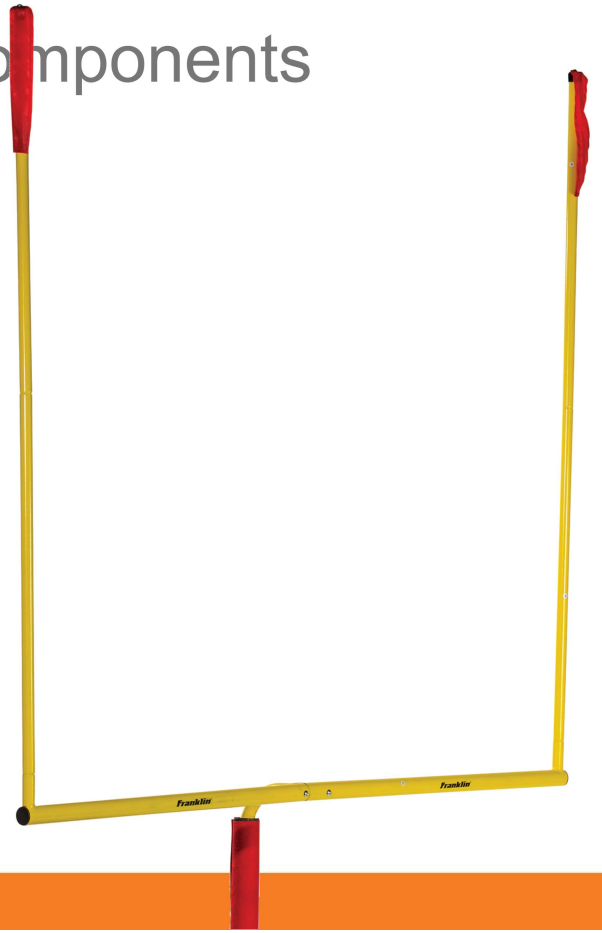
Quality

Develop Intelligence

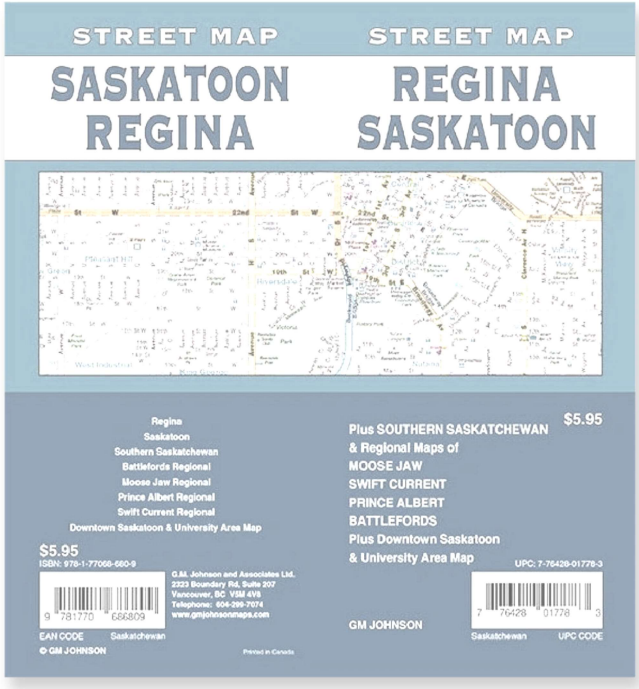
Goals

1. Explain the tradeoffs involved with testing components
2. List 3 best practices



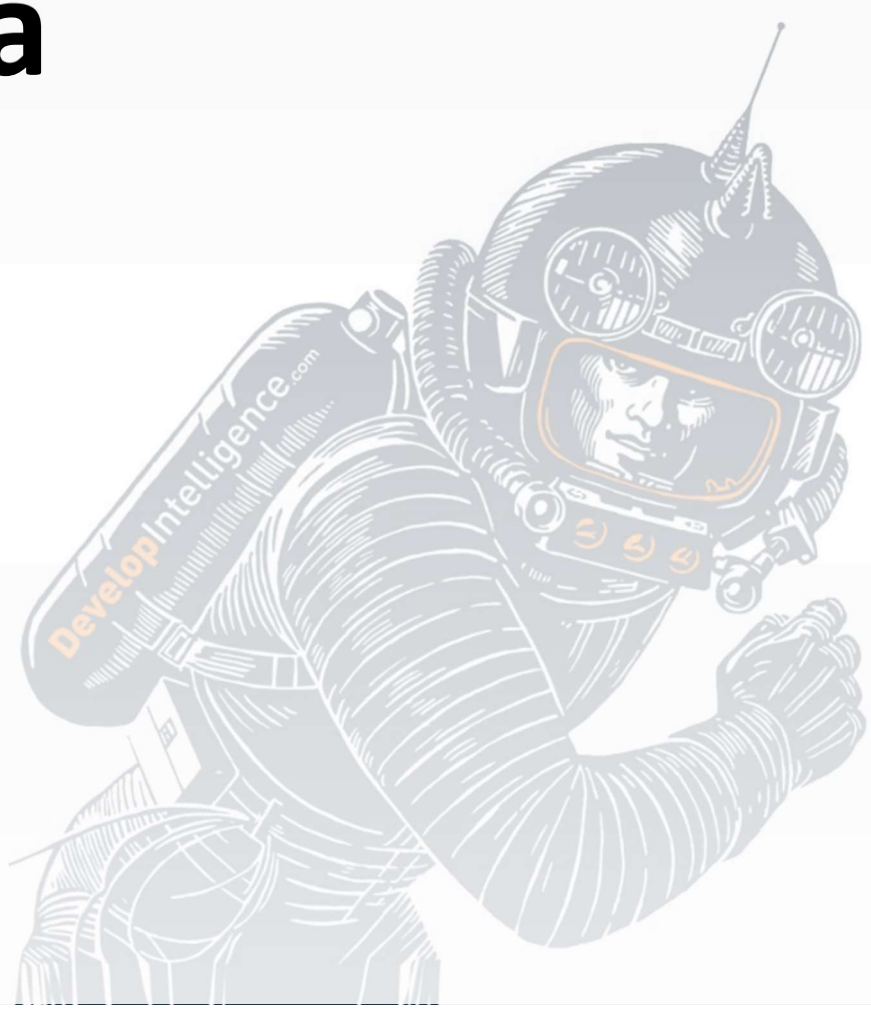
Roadmap

1. Lint Extravaganza
2. Testing
3. Best Practices



Develop
Intelligence

Lint Extravaganza



Linting Defined

Develop Intelligence

- Linting is
  - Static analysis
  - Finds style and quality issues
- In JavaScript
  - JSLint created in 2002 by Doug Crockford
  - ESLint is the current standard

Pros and Cons

- **Advantages**
  - Catch possible errors
  - Enforce consistent style
  - Automatically fix problems
- **Disadvantages**
  - Irritating and overbearing out of the box

Category: Likely Errors

Develop Intelligence

- Obvious problems that a compiler would catch
- Best practice: Treat these as compilation errors

Duplicates

- Weirdly, this is legit JavaScript

```
function xyz(a, b, a) {
  //Do stuff
}
```

- Same here

```
var cone = {
  flavor: "vanilla",
  flavor: "chocolate"
};
```

Reassign Function

Develop Intelligence

```
1 function format(){/*...*/}
2 //Other stuff
3 function format(){/*...*/}
```

Unreachable Code

Develop Intelligence

```
1 function xyz() {
2   //Do stuff
3   return something;
4   console.log("Warning!!");
5 }
```

typeof failures

Develop Intelligence

- Operator `typeof` returns well known strings
- Linters can catch bad spelling

```
1 typeof label === "strnig"
2 typeof value == "undefimed"
3 typeof result != "nunber"
```

Template string confusion

Develop Intelligence

```
1 "Hello ${name}!";
2 'Hello ${name}!';
3 "Time: ${12*60*60*1000}";
```

Ruleset Menu

Develop Intelligence

- Airbnb
- Google
- ESLint
- ie11

Customization

- From **.eslintrc.json**

```
{
  "extends": "google",
  "parserOptions": {
      "ecmaVersion": 6,
      "sourceType": "module",
  },
  "rules": {
      "semi": "error",
      "prefer-regex-literals": 0,
  }
}
```

Integration: IDEs

Develop Intelligence

- Linters have IDE Plugins
- Autofix problems where possible

```
{
    "files.autoSave": "onFocusChange",
    "xo.enable": true,
    "xo.format.enable": true,
    "editor.formatOnSave": true,
}
```

Build Integration

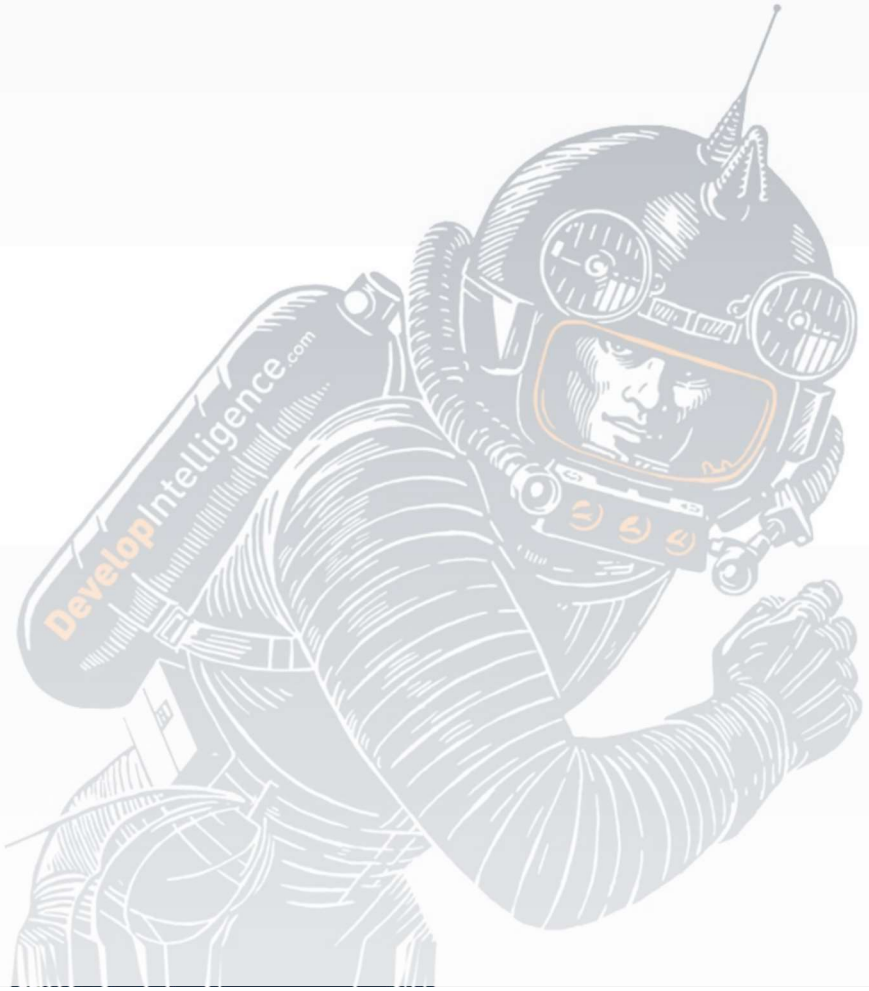
Develop Intelligence

- Treat linting like compiling
- Lint failure means a broken build

```
1 > eslint src/**
```

Develop
Intelligence

Testing



Options (From reactjs.org)

There are a few ways to test React components. Broadly, they divide into two categories:

- *Rendering component trees in a simplified test environment and asserting on their output.*
- *Running a complete app in a realistic browser environment (also known as “end-to-end” tests).*

Example Component

Develop Intelligence

```
export default function Hello(props) {
  if (props.name) {
    return <h1>Hello, {props.name}!</h1>;
  } else {
    return <span>Hey, stranger</span>;
  }
}
```

Example Tests

Develop Intelligence

```
it("renders with or without a name", () => {
  act(() => render(<Hello />, container));
  expect(container.textContent).toBe("Hey, stranger");

  act(() => render(<Hello name="Jenny" />, container));
  expect(container.textContent).toBe("Hello, Jenny!");

  act(() => render(<Hello name="Margaret" />, container));
  expect(container.textContent).toBe("Hello, Margaret!");
});
```

-- From the old reactjs.org

Infrastructure

Develop Intelligence

```
1  import { unmountComponentAtNode } from "react-dom";
2
3  let container = null;
4  beforeEach(() => {
5    // setup a DOM element as a render target
6    container = document.createElement("div");
7    document.body.appendChild(container);
8  });
9
10 afterEach(() => {
11   // cleanup on exiting
12   unmountComponentAtNode(container);
13   container.remove();
14   container = null;
15 });
```

Complaints

- Bad design: Mixes burdens
- Uncaptured abstractions
- Lots of setup and things to mock

Alternative Perspective (From me)

Develop Intelligence

- Don't test rendering
- ***As much as possible***, tease application logic apart from rendering
- That means
  1. Keep views very thin
  2. Put the actual logic in services and pure functions
  3. Unit test the services and pure functions

Example Refactored (I)

Develop Intelligence

```
interface Greeting{
    content:string;
    isLaidBack:boolean;
}

export const makeGreeting = (name?:string):Greeting=>
    !name ? {content:'Hey, stranger', isLaidBack:true};
          : {content:`Hello, ${name}!`, isLaidBack:false};

export default function Hello(props) {
  const greeting = makeGreeting(props.name);
  if (props.isLaidBack) {
    return <span>{greeting.content}</span>;
  }
  return <h1>{greeting.content}</h1>;
}
```

Refactored Test

Develop Intelligence

```
1  import { expect, test } from 'vitest';
2  import { makeGreeting } from './hello';
3
4  test('makeGreeting should be laidback for empty', () => {
5      const actual = makeGreeting('');
6      expect(actual.isLaidBack).toBe(true);
7  });
8
9  test('makeGreeting should be laidback for spaces', () => {
10     const actual = makeGreeting('    ');
11     expect(actual.isLaidBack).toBe(true);
12 });
```

Example Refactored (II)

Develop Intelligence

```
interface Greeting{
    content:string;
    className:string;
}

export const makeGreeting = (name?:string):Greeting=>
    !name ? {content:'Hey, stranger', className: 'g-laid-back'};
          : {content:`Hello, ${name}!`, className:'g-enthusiastic'};

export default function Hello(props) {
  const greeting = makeGreeting(props.name);
  return <span className={greeting.className}>
            {greeting.content}
         </span>;
}
```

Tools

- React Testing Library
- Jest
- Vitest

Useful `renderHook`

Word of Warning

This is a convenience wrapper around render with a custom test component. The API emerged from a popular testing pattern and is mostly interesting for libraries publishing hooks. You should prefer render since a custom test component results in more readable and robust tests since the thing you want to test is not hidden behind an abstraction.

Example: Fail

Develop Intelligence

```
1  test('useFrozenState should freeze at runtime', () => {
2    const [ n, ] = useFrozenState(10);
3    try {
4      const nUnknown:any = n;
5      nUnknown.content= 50;
6    } catch {
7      return;
8    }
9    assert(false);
10 });
```

Example: Fixed

Develop Intelligence

```
1  test('useFrozenState should freeze at runtime', () => {
2    const {result} = renderHook(()=>useFrozenState(10));
3    try {
4      const nUnknown:any = result.current;
5      nUnknown.content= 50;
6    } catch (e:unknown) {
7      console.log(`\n\n------\nError:${JSON.stringify(e)}\n\n--------\n`)
8      return;
9    }
10   assert.fail("No error thrown.");
11 });
```

Example (I)

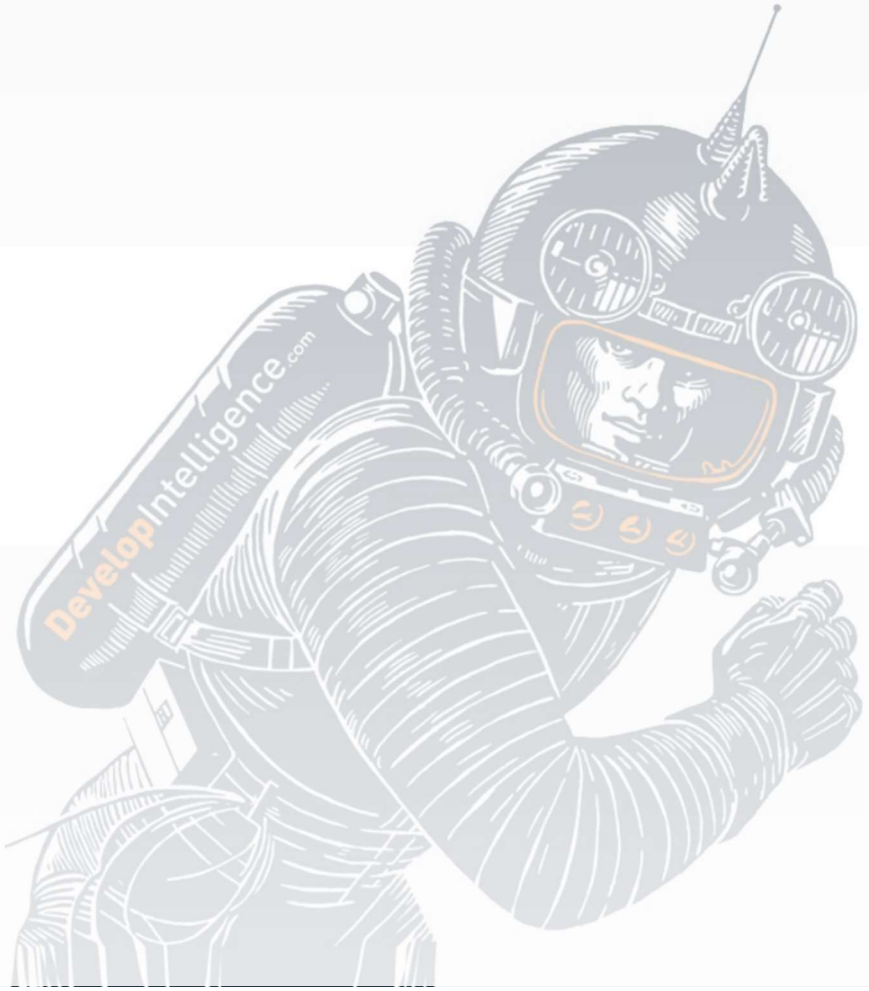
Develop Intelligence

```
import {renderHook} from '@testing-library/react'

test('returns logged in user', () => {
  const {result} = renderHook(() => useLoggedInUser())
  expect(result.current).toEqual({name: 'Alice'})
});
```

Develop
Intelligence

Best Practices



Incomplete and Mostly Wrong

1. Keep views thin
2. Test flow control instead of rendering
3. Organize files by problem domain, *not* react types
4. If you have multiple routes, use a framework
5. Use pre-commit hooks to lint, test, format, and typecheck
6. All the usual advice: DRY, low coupling, high cohesion, etc.

Develop
Intelligence

Review

1. Explain the tradeoffs involved with testing components
2. List 3 best practices