Estrategias de Testing para Frontend

Juan Felipe Avalo

Como asegurar que un software no tenga defectos

¿Es imposible?

- Complejidad creciente
- *Unknown unknowns*
- Sistemas no funcionan en un vacío
  - left-pad
  - DIG
  - Equifax

Formas de atrapar defectos en el desarrollo de software

- Planificación
  - Requisitos claros
- Buenas prácticas
- *Pull Request*
- *Testing*

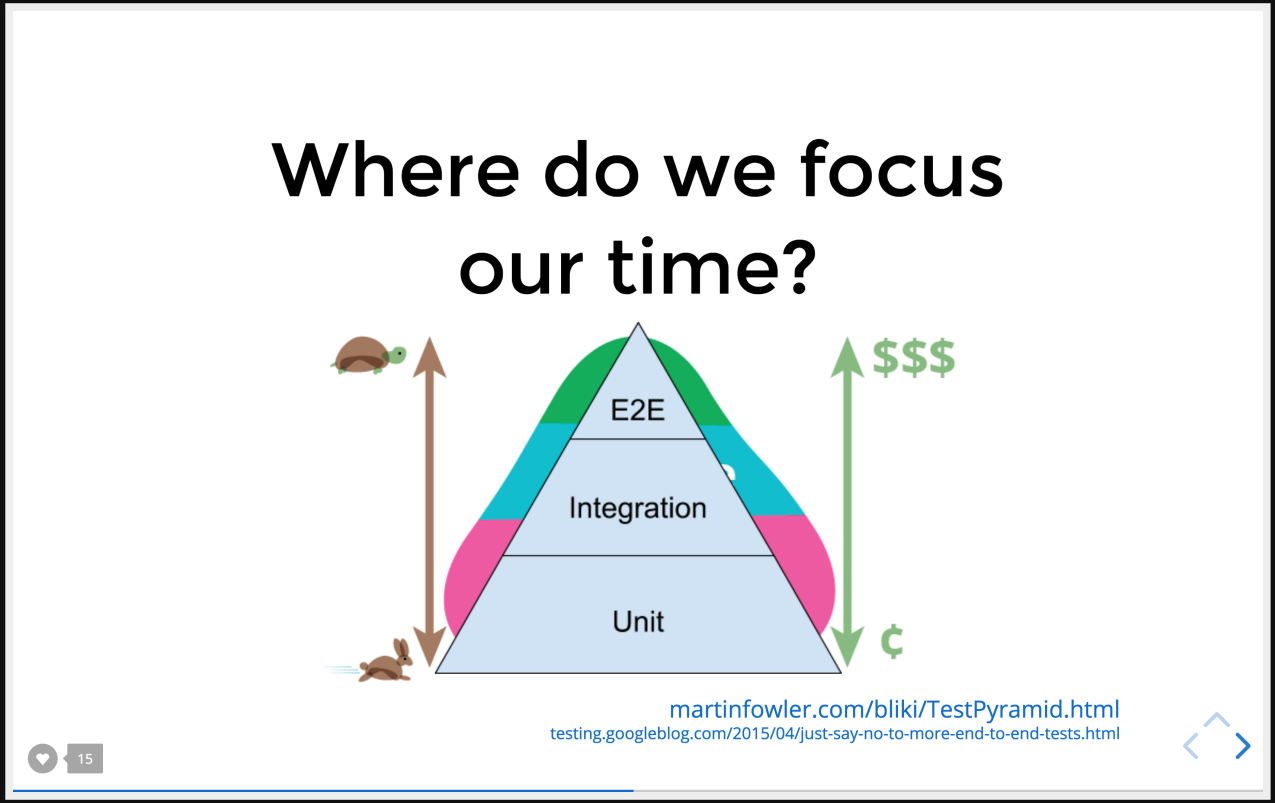
¿Por qué hacer testing?

“También lo subí directamente a producción”

- Informar expectativas del código nuevo
- Asegurar que la funcionalidad nueva cumpla con sus requisitos
- Asegurar que la funcionalidad nueva se integre con el sistema actual
  - Que no rompa la aplicación

> *La función para verificar rut solo recibe 9 dígitos sin guión*

Tipos de testing



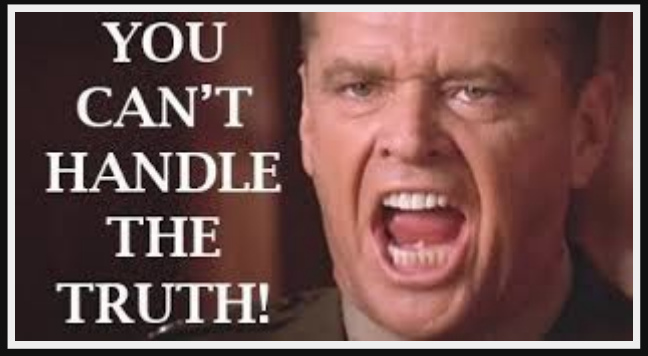
En desarrollo usualmente estamos en los dos primeros niveles

Unit Test vs Integration Test

Testing en Frontend

Particularidades

- Difícil tener un *oráculo de la verdad*
  - Backend: Definiciones precisas
  - Frontend: “Lo quiero un poco mas a la derecha”



- Difícil automatizar test visuales

¿Qué se puede hacer durante el desarrollo en frontend?

- Probar manualmente cada cambio.
- Usar herramientas para hacer *visual diffs*.
- Usar herramientas para probar la estructura de la página.
- Ver que al menos la página es compilada (*smoke test*).
- Verificar solo casos con una condición *fail/pass* concreta.
  - Lógica vs presentación

Como probar en React

En general

- Existen herramientas para:
  - ejecutar test unitarios en javascript (*runners*)
  - ayudar en el desarrollo de pruebas (*helpers*)

Jest

- Test runner
- Ofrece utilidades para hacer asserts

```
expect(fecha).toBe("06/06/2666");
expect(fecha).not.toBe("05-04-33");
```

- Ofrece utilidades para hacer *mocks*

```
jest.mock('../utils/tealium');
...

expect(tealium.sendView.mock.calls.length).toBe(1);
```

- Puede realizar comparaciones por *snapshots*

```
  expect(component).toMatchSnapshot()
```

```
exports[`BackButton is rendered correctly when used with a string 1`] = `
<jss(basebutton) as="{[Function]}" classname="primary rounded undefined"
id="backButton" to="link" uppercase="{false}">
  <component>
  <formattedbackmessage defaultmessage="Default message" id="INTL_ID">
</formattedbackmessage></component></jss(basebutton)>
`;
```

- El uso de snapshots es bueno para asegurar que la estructura de un componente sea consistente.
- No asegura problemas visuales por interacciones.
- Acoplado a la implementación.

Enzyme

- Creado por *AirBnB*
- Herramienta con helpers para React.
- Ej: Acceder a *states*, *props*, hijos
- *shallow rendering*
  - Solo se renderea componente.
  - Permite sumergirse en componentes hijos.
  - Evita testear dependencias.

React Testing Library

The more your tests resemble the way your software is used, the more confidence they can give you. –Kent C. Dobbs

- Utilidades para montar componentes, simular eventos, etc.
- No usa *shallow rendering* por diseño.
  - Monta el componente con todas sus dependencias.
  - Solo mocks a servicios muy lentos/críticos

```
fireEvent.click(getByText('Test'));
```

Recomendaciones

Enzyme vs React Testing Library

- Ambos hacen tareas similares.
- Distintos enfoques.

- Enzyme tiende a introducir más coupling con la implementación
  - Permite entrar con mayor detalle al estado del componente
  - Tendencia a que se rompan tests por cambios pequeños
  - Mas fácil para probar componentes que estén dentro de una estructura muy profunda

- React Testing Library tiene el potencial de ser más robusto
  - Probar simulando a un usuario es bastante intuitivo
  - No acceder a estado interno no ha sido problema (hasta ahora)
  - Si se rompe test es más significativo
  - Componentes probablemente necesiten hacer un setup y/o mock de sus prerequisitos

Sobre los unit tests

- Importante pensar en pruebas que sean significativas.
  - Deben dar información si se rompen

- Testear solo lo no trivial

“Voy a testear que 5 == 5”

- Tests son código, y el código se mantiene
- Evitar escribir componentes gigantes.
  - Tienen más dependencias
  - Más fácil que se rompan.

Sobre snapshots

- Snapshots son útiles en ciertos casos:
  - Componentes de presentación.
  - Comportamiento se refleja en el árbol de componente.
  - Como *smoke test*.

- Pueden traer dificultades asociadas:
  - Falsos positivos: Componente cambia sin que afecte su funcionalidad.
  - Componentes muy grandes:
    - No se entienden bien.
    - Son ignoradas.