

테스트 주도 개발과 프런트엔드 테스트

실무에 바로 적용하는 프런트엔드 테스트

TDD와 프론트엔드 단위 테스트

0. 컴포넌트 생성



```
export default function TextField() {  
  return <input type="text" />  
}
```

TDD와 프론트엔드 단위 테스트


1. 테스트 실패




```
it('기본 placeholder "텍스트를 입력해 주세요."가 노출된다.', async () => {  
  await render(<TextField />);  
  
  const textInput = screen.getByPlaceholderText('텍스트를 입력해 주세요. ');  
  
  expect(textInput).toBeInTheDocument();  
});
```

TDD와 프론트엔드 단위 테스트

2. 테스트 성공



```
export default function TextField() {  
  return (  
    <input  
      type="text"  
      placeholder="텍스트를 입력해 주세요."   
    />  
  );  
}
```

TDD와 프론트엔드 통합 테스트

1. 테스트 실패




```
it('텍스트를 입력하면 onChange prop으로 등록한 함수가 호출된다.', async () => {  
  const spy = vi.fn();  
  
  const { user } = await render(<TextField onChange={spy} />);  
  
  const textInput = screen.getByPlaceholderText('텍스트를 입력해 주세요. ');  
  
  await user.type(textInput, 'test');  
  
  expect(spy).toHaveBeenCalled('test');  
});
```

TDD와 프론트엔드 통합 테스트

2. 테스트 성공


```
export default function TextField({ onChange }) {  
  const [value, setValue] = useState('');  
  
  return (  
    <input  
      type="text"  
      placeholder="텍스트를 입력해 주세요."  
      onChange={ev => {  
        setValue(ev.target.value);  
        onChange?.(ev.target.value);  
      }}  
      value={value}  
    />  
  );  
}
```



TDD와 프론트엔드 통합 테스트

3. 리팩터링

```
export default function TextField({ onChange }) {  
  const [value, setValue] = useState('');  
  
  const changeValue = ev => {  
    setValue(ev.target.value);  
    onChange?.(ev.target.value);  
  };  
  
  return (  
    <input  
      type="text"  
      placeholder="텍스트를 입력해 주세요."  
      onChange={changeValue}  
      value={value}  
    />  
  );  
}
```



TDD와 프론트엔드 통합 테스트

0. 컴포넌트 생성 및 테스트 환경 설정 (상태 관리 모킹, msw 설정)



```
const PRODUCT_PAGE_LIMIT = 20;

const ProductList = ({ limit = PRODUCT_PAGE_LIMIT }) => {
  return (
    <div>
      ProductList
    </div>
  );
}
```


TDD와 프론트엔드 통합 테스트

1. 테스트 실패

```
it('로딩이 완료된 경우 상품 리스트가 제대로 모두 노출된다.', async () => {
  await render(<ProductList limit={PRODUCT_PAGE_LIMIT} />);

  const productCards = await screen.findAllByTestId('product-card');

  expect(productCards).toHaveLength(PRODUCT_PAGE_LIMIT);

  productCards.forEach((el, index) => {
    const productCard = within(el);
    const product = data.products[index];

    expect(productCard.getByText(product.title)).toBeInTheDocument();
    expect(productCard.getByText(product.category.name)).toBeInTheDocument();
    expect(
      productCard.getByText(formatPrice(product.price)),
    ).toBeInTheDocument();
    expect(
      productCard.getByRole('button', { name: '장바구니' }),
    ).toBeInTheDocument();
    expect(
      productCard.getByRole('button', { name: '구매' }),
    ).toBeInTheDocument();
  });
});
```

TDD와 프론트엔드 통합 테스트

2. 테스트 성공

```
const PRODUCT_PAGE_LIMIT = 20;

const ProductList = ({ limit = PRODUCT_PAGE_LIMIT }) => {
  const filter = useFilterStore(state =>
    pick(state, 'categoryId', 'title', 'minPrice', 'maxPrice'),
  );

  const { data, ...productsMethods } = useProducts({
    limit,
    params: filter,
  });

  const products =
    data?.pages.reduce((acc, cur) => [...acc, ...cur.products], []) ?? [];

  return (
    <div>
      {products.map((product, index) => (
        <div key={`_${product.id}_${index}`}>
          <div>
            <p>
              {product.category}
            </p>
            ...
          </div>
        </div>
      ))}
    </div>
  );
}
```

TDD와 프론트엔드 통합 테스트

2. 테스트 성공

```
const PRODUCT_PAGE_LIMIT = 20;

const ProductList = ({ limit = PRODUCT_PAGE_LIMIT }) => {
  const filter = useFilterStore(state =>
    pick(state, 'categoryId', 'title', 'minPrice', 'maxPrice'),
  );

  const { data, ...productsMethods } = useProducts({
    limit,
    params: filter,
  });

  const products =
    data?.pages.reduce((acc, cur) => [...acc, ...cur.products], []) ?? [];

  return (
    <div>
      {products.map((product, index) => (
        <div key={`_${product.id}_${index}`}>
          <div>
            <p>
              {product.category}
            </p>
            ...
          </div>
        </div>
      ))}
    </div>
  );
}
```

3. 리팩터링

```
const ProductList = ({ limit = PRODUCT_PAGE_LIMIT }) => {
  const filter = useFilterStore(state =>
    pick(state, 'categoryId', 'title', 'minPrice', 'maxPrice'),
  );

  const { data, ...productsMethods } = useProducts({
    limit,
    params: filter,
  });

  const products =
    data?.pages.reduce((acc, cur) => [...acc, ...cur.products], []) ?? [];
  const { fetchNextPage, isFetchingNextPage, hasNextPage } = productsMethods;

  return (
    <Grid container spacing={1} rowSpacing={1} justifyContent="center">
      {products.map((product, index) => (
        <Grid
          key={`_${product.id}_${index}`}
          item
          xs={6}
          sm={6}
          md={3}
          onClick={handleClickItem}
          data-testid="product-card"
        >
          <Card sx={{ maxWidth: 345, cursor: 'pointer' }}>
            <CardMedia component="img" height="140" image={images?.[0]} />
            <CardContent>
              ...
            </CardContent>
          </Card>
        </Grid>
      ))}
    </Grid>
  );
};
```

TDD와 프론트엔드 통합 테스트

3. 추가 리팩터링

```
const ProductList = ({ limit = PRODUCT_PAGE_LIMIT }) => {
  const filter = useFilterStore(state =>
    pick(state, 'categoryId', 'title', 'minPrice', 'maxPrice'),
  );

  const { data, ...productsMethods } = useProducts({
    limit,
    params: filter,
  });

  const products =
    data?.pages.reduce((acc, cur) => [...acc, ...cur.products], []) ?? [];

  return (
    <Grid container spacing={1} rowSpacing={1} justifyContent="center">
      {products.map((product, index) => (
        <ProductCard
          key={`_${product.id}_${index}`}
          product={product}
        />
      ))}
    </Grid>
  );
};
```

TDD는 만병통치약?

TDD는 만병통치약?

테스트를 작성한다는 것이 곧 TDD를 따라야 하는 것은 아니다.

- 프로토타이핑을 간단하게 한 후 테스트를 작성할 수도 있음
- 간단한 로직 추가 후 테스트를 작성할 수도 있음
- 프론트엔드에서는 TDD를 도입하기 어려운 영역도 존재

TDD는 만병통치약?

테스트를 작성한다는 것이 곧 TDD를 따라야 하는 것은 아니다.

결국 중요한 것은 개발 단계에서
테스트 피드백을 통해 기능의 안정성을 높이는 것

정리

TDD와 단위 테스트

- 공통 컴포넌트, 혹은 같은 모듈은 TDD를 적용하기에 적합
 - 검증하고자 하는 기능이 명확하고 범위가 넓지 않다.

TDD와 통합 테스트

- 비즈니스 로직에 대한 테스트는 TDD를 적용하기에 적합
- 상태 관리, API 호출 로직, 컴포넌트 조합 등을 TDD를 통해 안정적으로 리팩토링 할 수 있음

모든 테스트를 작성할 때 TDD를 적용할 필요는 없다.

- 결국 중요한 것은 개발 단계에서 테스트 피드백을 통해 기능의 안정성을 높이는 것
- 방법론 자체에 몰두하기 보다는, 테스트의 목적에 집중하자.
- 꼭, TDD가 아니더라도 테스트를 도입하는 현실적인 방법을 찾아 팀 문화로 만들고 정착시키자.