Lecture 8

Quality

Frontend Web Development

Patterns and Architecture

Stateful and Stateless Components

Stateful

- Maintain their own state
- May contain business logic
- Harder to debug
- Harder to reuse
- Usually self-controlled

Stateless

- Have no state
- Simple render logic
- Easier to debug
- Easier to reuse
- Fully controlled by parent

Components: stateful, stateless, dumb and smart

Stateless Component Example

```
const List = (props) => (
    <div>
        <div className="coolHeader">{props.title}</div>
        <ul>
            {props.list.map(listItem => {
                <li
                  className="coolListItem"
                >
                  {listItem}
                </li>
            })}
        </ul>
    </div>
)
```

Stateful Component Example

```
const TypeStuffIn = () => {
   const [input, setInput] = useState('');
   const handleChange = e => setInput(e.target.value);

   return (
       <input
           type="text"
           value={this.state.input}
           onChange={this.handleChange}
       />
   )
}
```

Dumb and Smart Components

Smart (Container)

- Are concerned with how things work
- Aware of its environment
- Can produce side effects
- Can use other smart and dumb components

Dumb (Presentational)

- Are concerned with how things look
- Don’t depend on its environment
- Don’t produce side effects
- Can use dumb components and html

Presentational and Container Components

Dumb Component Example

```
const List = props => (
    <div>
        <div className="coolHeader">{props.title}</div>
        <ul>
            {props.list.map(listItem => {
                <li
                  className="coolListItem"
                >
                  {listItem}
                </li>
            })}
        </ul>
    </div>
)
```

Smart Component Example

```
import { createStore, createApi } from 'effector'
import { useStore } from 'effector-react'
import { $counter, increment, decrement } from './counter-store'

const App = () => {
   const counter = useStore($counter)
   return (
       <div>
           {counter}
           <button onClick={increment}>Increment</button>
           <button onClick={decrement}>Decrement</button>
       </div>
   )
}
```

Classic Project Structure

```
React-app
├── node_modules
├── build
├── src
│   ├── components
│   │   ├── Component
│   │   │   ├── Component.jsx
│   │   │   ├── Component.styles.js
│   │   │   └── ... (component assets, tests or nested components)
│   │   └── ... (components group, for example, ui-components)
│   ├── pages
│   │   └── ... (page)
│   ├── services
│   ├── store
│   ├── utils
│   ├── ... (other groups, for example, hooks, assets)
│   ├── index.js
├── webpack.config.js
├── ... (other project configs)
```

General Recommendations (React)

- Put only one component in one file
- Place related stuff (tests, styles, assets, etc.) near to the component
- Place reusable components into `components` folder
- You can add nesting folders if components are used only in context of other components
- Place any utility files in a special folder (`utils`, `hooks`, `validators`, etc.) only in case they're reusable
- Place any configs only on top level or in `config` folder
- Place top-level components to a specific folder (`pages`, `containers`, etc.)
- Respect naming conventions of your team and framework

Code organization

Feature-Sliced Design

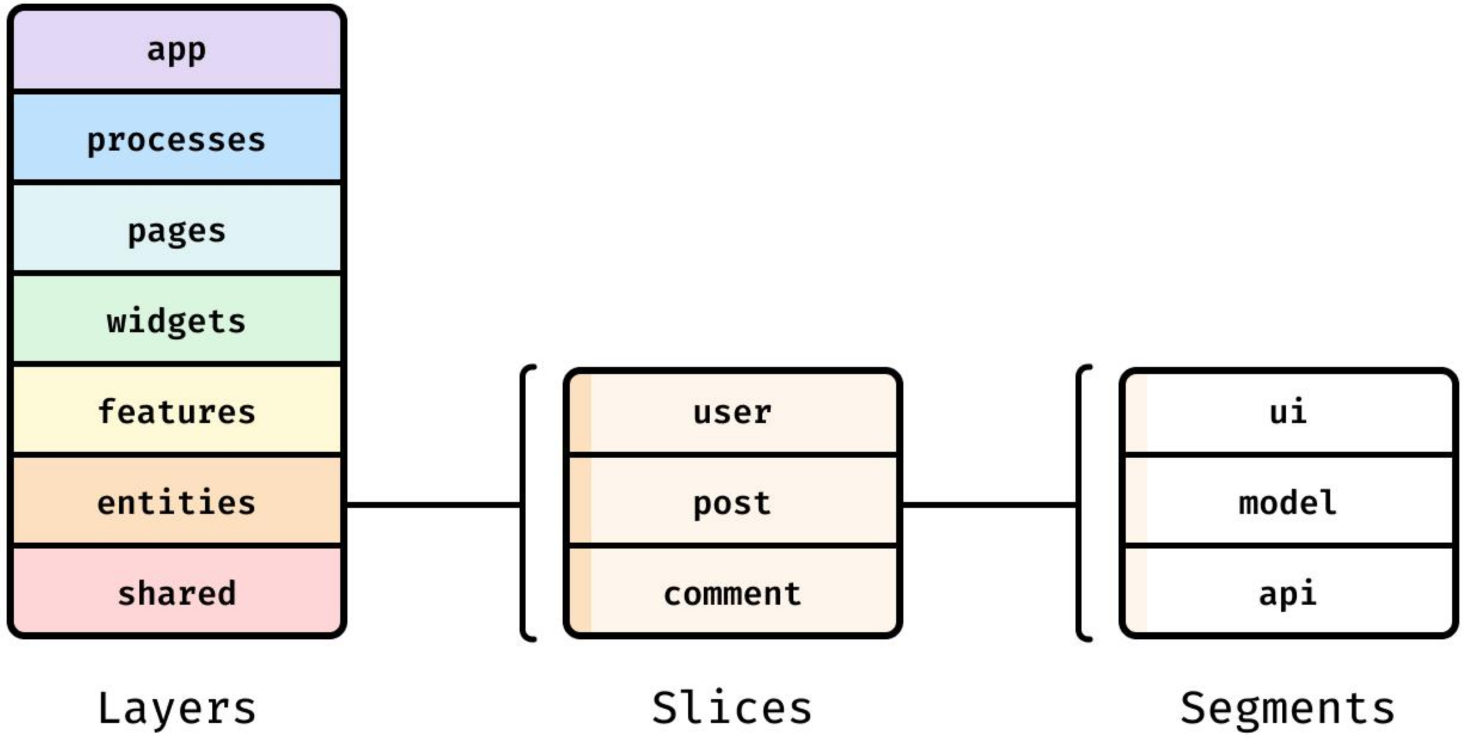
What developers can get?

- Focus on business features, not on architecture problems
- Solution that is proven by experience of others
- Track and solve problems of tech debt earlier

What business can get?

- Better and faster onboarding
- Solution that is proven by experience of others
- Applicability for different stages of the project

Motivation


app
processes
pages
widgets
features
entities
shared
user
post
comment
ui
model
api
Layers
Slices
Segments

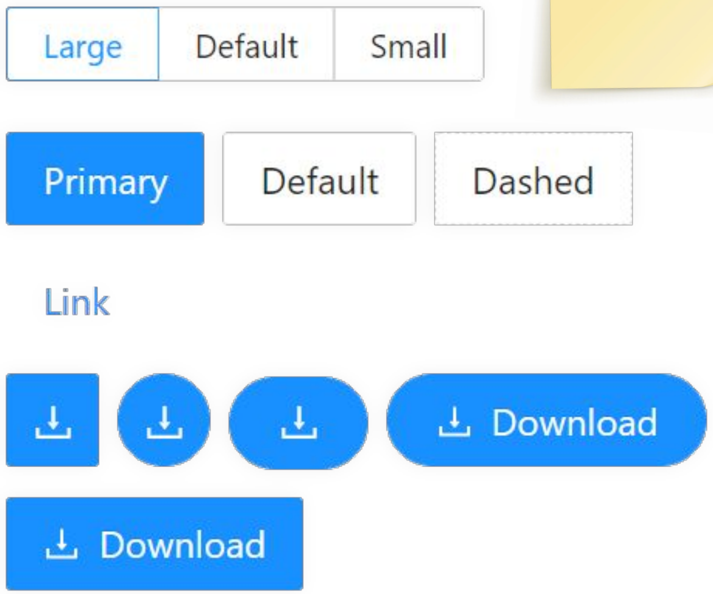
Layers

FSD - Layers

```
└── src/
    ├── app/                 # Initializing application logic
    ├── processes/           # (*) Application processes running over pages
    ├── pages/               # Application pages
    ├── widgets/             # Independent and self-contained blocks for pages
    ├── features/            # (*) Processing of user scenarios
    ├── entities/            # (*) Business entities the domain logic operates with
    └── shared/              # Reused modules, non business specific
```

SHARED

**Reusable modules,
without binding to business logic**

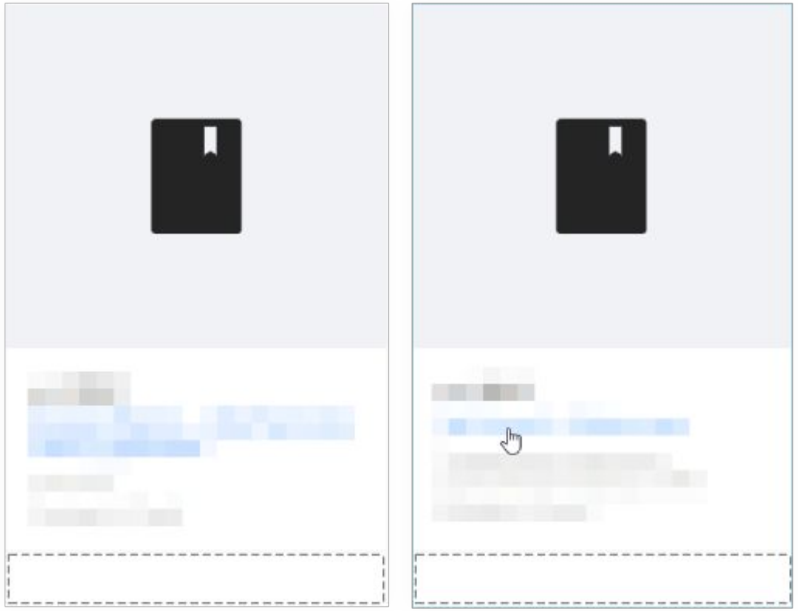


Layers - shared

Layers - entities

ENTITIES

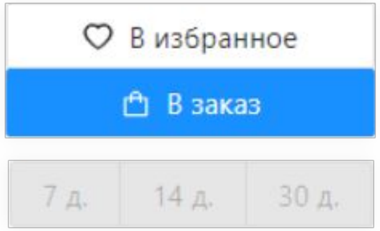
Business entities

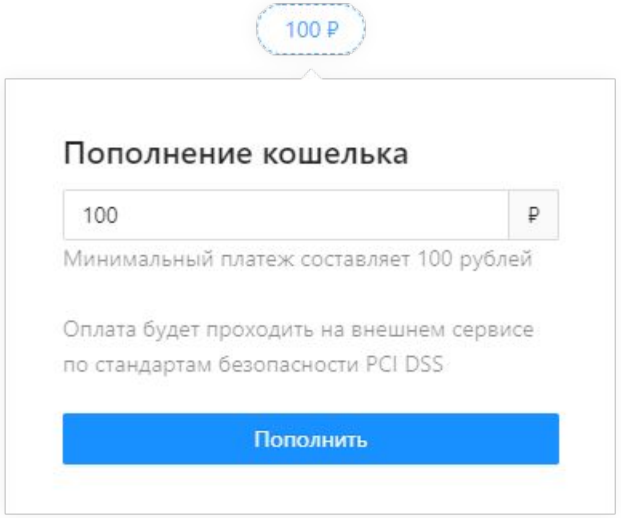


Layers - features

FEATURES

Parts of functionality that carry business value





Пополнение кошелька

100 ₽

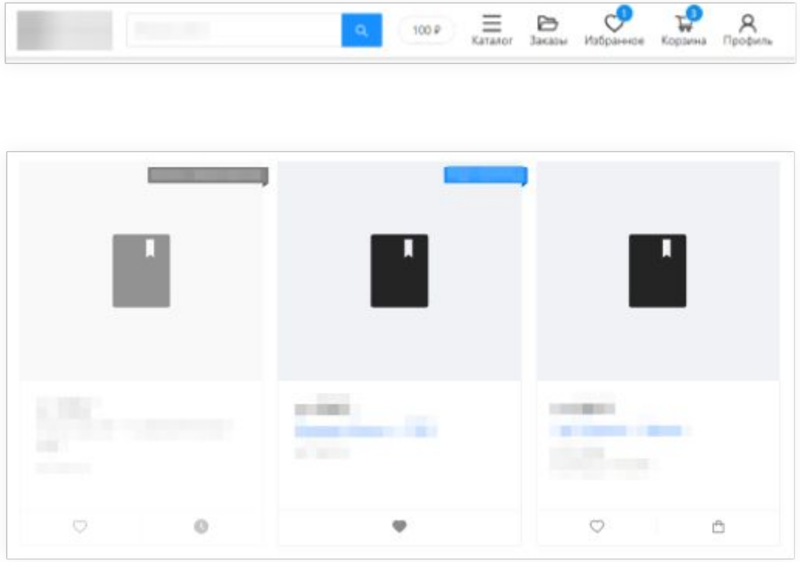
Минимальный платеж составляет 100 рублей

Оплата будет проходить на внешнем сервисе по стандартам безопасности PCI DSS

Layers - widgets

WIDGETS

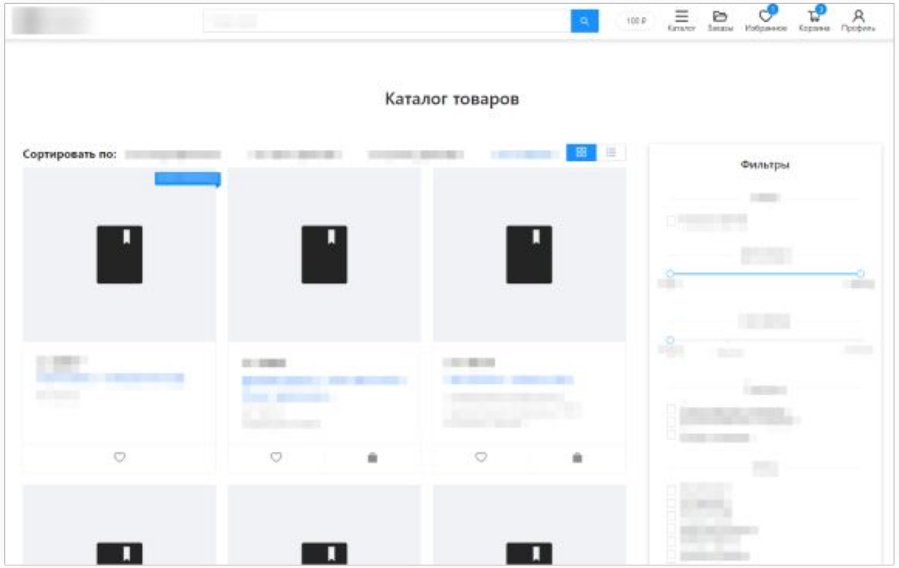
Independent blocks combining the lower layers



Layers - pages

PAGES

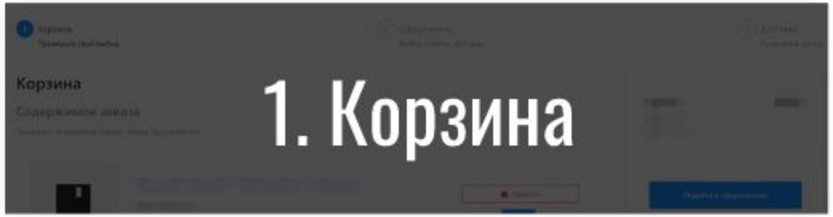
Pages/screens of the app

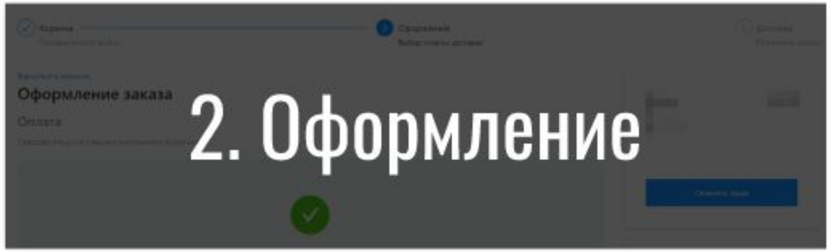


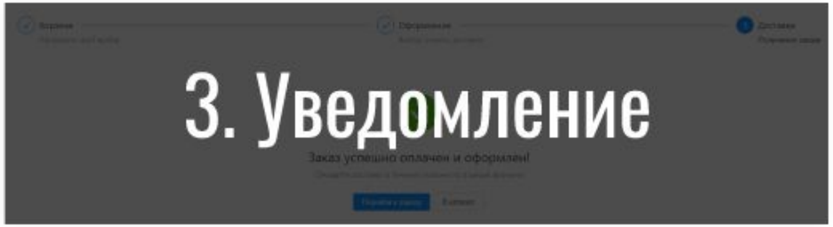
Layers - processes

PROCESSES

Processes that run through multiple page







Layers - app

APP

Application logic initialization

Product

```
+ withRouter
+ withStore
+ withGlobalStyles
+ withUIKitConfig
+ ...
```

Layer	Can use	Can be used by
app	`shared`, `entities`, `features`, `widgets`, `pages`, `processes`	-
processes	`shared`, `entities`, `features`, `widgets`, `pages`	`app`
pages	`shared`, `entities`, `features`, `widgets`	`processes`, `app`
widgets	`shared`, `entities`, `features`	`pages`, `processes`, `app`
features	`shared`, `entities`	`widgets`, `pages`, `processes`, `app`
entities	`shared`	`features`, `widgets`, `pages`, `processes`, `app`
shared	-	`entities`, `features`, `widgets`, `pages`, `processes`, `app`

Slices

```
├── app/
|   # Does not have specific slices, contains meta-logic on the project
├── processes/
|   # Slices implementing processes on pages
|    ├── payment
|    ├── auth
|    ├── quick-tour
|    └── ...
├── pages/
|   # Slices implementing application pages
|    ├── profile
|    ├── sign-up
|    ├── feed
|    └── ...
```

```
├── widgets/
|   # Slices implementing independent page blocks
|   ├── header
|   ├── feed
|   └── ...
├── features/
|   # Slices implementing user scenarios on pages
|   ├── auth-by-phone
|   ├── inline-post
|   └── ...
├── entities/
|   # Slices of business entities for implementing a more complex BL
|   ├── viewer
|   ├── posts
|   ├── i18n
|   └── ...
├── shared/
|    # Does not have specific slices
|    # is rather a set of commonly used segments, without binding to the BL
```

Segments

```
{layer}/
    ├── {slice}/
    |    ├── ui/          # UI-logic (components, ui-widgets,...)
    |    ├── model/       # Business logic (store, actions, effects, …)
    |    ├── lib/         # Infrastructure logic (utils/helpers)
    |    ├── config*/     # Configuration (of the project / slice)
    |    └── api*/        # Logic of API requests (api instances, …)
```

When is the methodology not needed?

- If the project will live for a short time
- If the project does not need a supported architecture
- If the business does not perceive the connection between the code base and the speed of feature delivery 🚩
- If it is more important for the business to close orders as soon as possible, without further support

Some drawbacks

- Not so easy to enter the methodology
- Requires more awareness and culture of development
- It’s quite difficult at the moment, but not after some time

Linting and Formatting

Formatting and Prettier

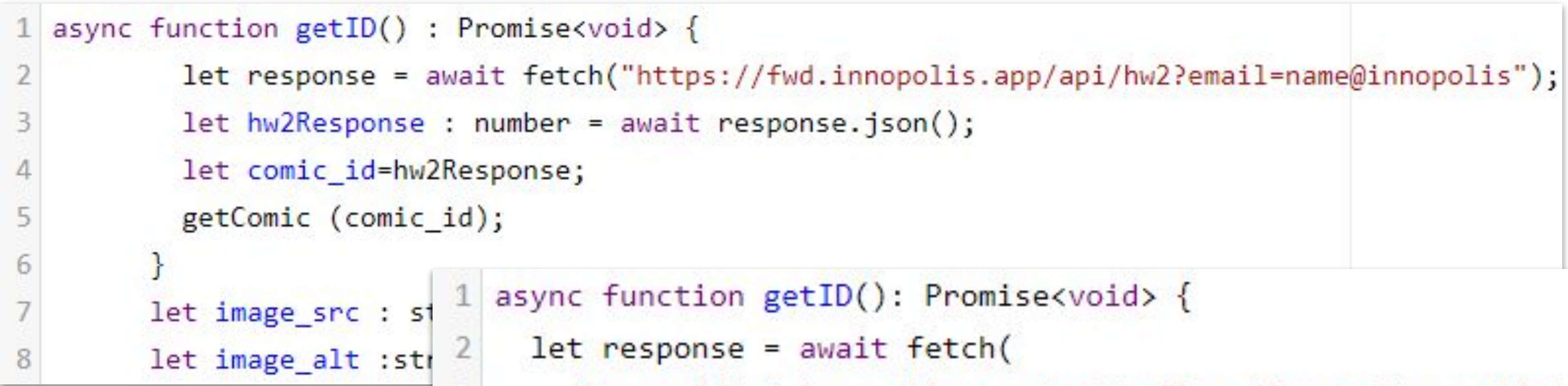
Formatting helps to keep your code base consistent across the project, especially when it is developed by several developers.

It can typically be done automatically since we are only focused on the style of the code, thus developers can think about functionality, but not the style.

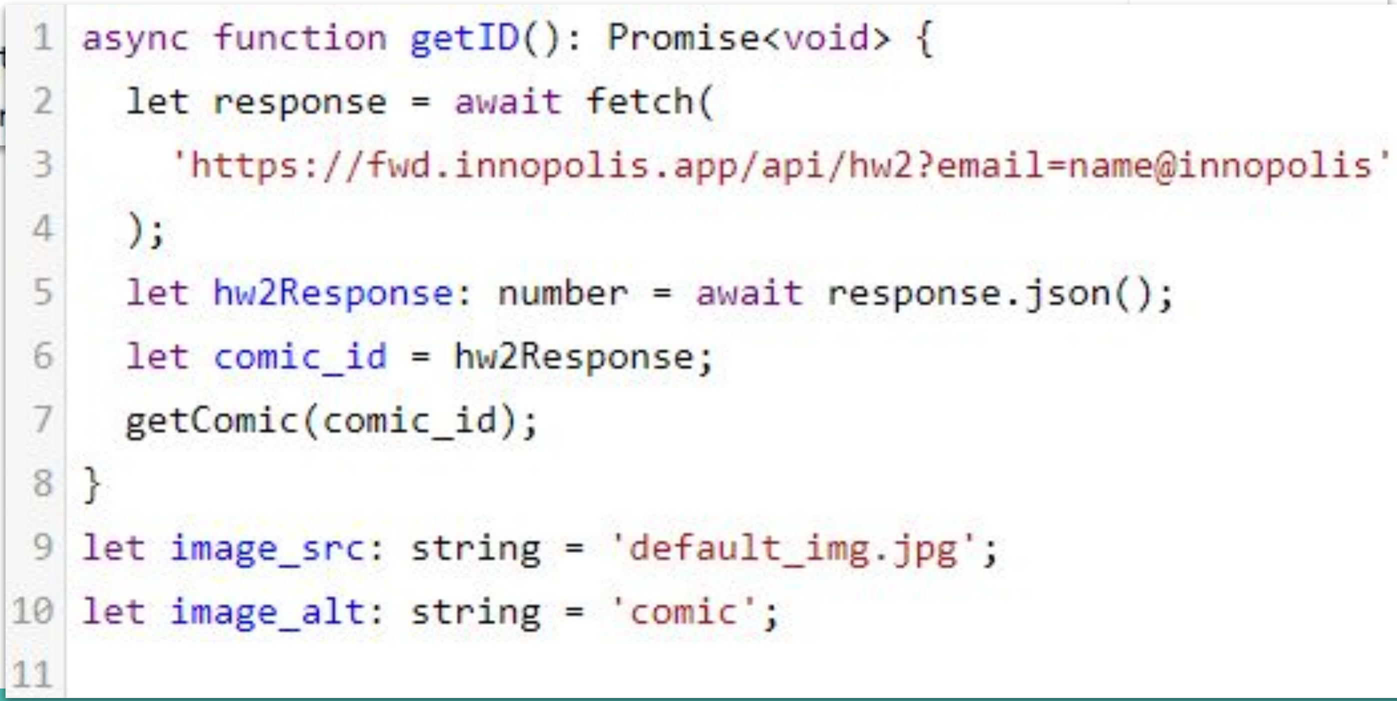
Prettier

- An opinionated code formatter (provides few options)
- Supports many languages (and has a plugin system)
- Integrates with most editors
- Ensures consistency in code style
- Saves you time and energy
  - Format on save
  - No need to discuss code style in code review

Why Prettier



```
async function getID() : Promise<void> {
        let response = await fetch("https://fwd.innopolis.app/api/hw2?email=name@innopolis");
        let hw2Response : number = await response.json();
        let comic_id=hw2Response;
        getComic (comic_id);
      }
      let image_src : st
      let image_alt :str
```



```
async function getID(): Promise<void> {
  let response = await fetch(
    'https://fwd.innopolis.app/api/hw2?email=name@innopolis'
  );
  let hw2Response: number = await response.json();
  let comic_id = hw2Response;
  getComic(comic_id);
}
let image_src: string = 'default_img.jpg';
let image_alt: string = 'comic';

```

Linting & ESLint

Linting is a type of static analysis that finds problematic patterns and code that doesn't adhere to certain style guidelines.

ESLint is a linter that is supported by many editors and CI/CD automation tools, and has a lot of rules that can help to analyze your code.

Supports auto-fixes for many rules

ESLint

no-fallthrough: incorrect

```
switch(foo) {
   case 1:
       doSomething();

   case 2:
       doSomethingElse();
}
```

no-fallthrough: correct

```
switch(foo) {
    case 1:
        doSomething();
        break;

    case 2:
        doSomething();
}

switch(foo) {

   case 1:

   case 2:

       doSomething();

}
```

```
function bar(foo) {
    switch(foo) {
        case 1:
            doSomething();
            return;

        case 2:
            doSomething();
    }
}
```

no-dupe-else-if: incorrect

```
if (n === 1) {
    foo();
} else if (n === 2) {
    bar();
} else if (n === 3) {
    baz();
} else if (n === 2) {
    quux();
} else if (n === 5) {
    quuux();
}
```

```
if (a) {
    foo();
} else if (b) {
    bar();
} else if (a || b) {
    baz();
}
```

no-dupe-else-if: correct

```
if (n === 1) {
    foo();
} else if (n === 2) {
    bar();
} else if (n === 3) {
    baz();
} else if (n === 4) {
    quux();
} else if (n === 5) {
    quuux();
}
```

```
if (a) {
    foo();
} else if (b) {
    bar();
}
if (a || b) {
    baz();
}
```

Presets

Some companies and development teams offer their own presets that anyone can use:

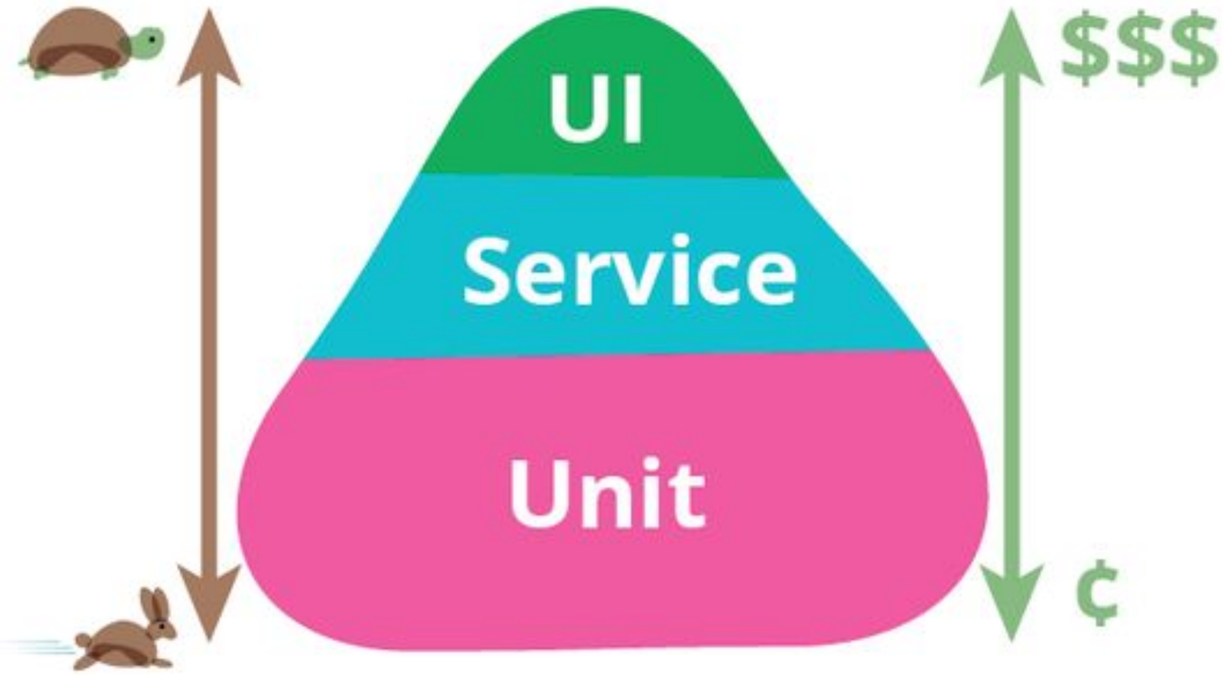
- Airbnb - Shareable config for Airbnb's style guide.
- Airbnb-typescript - Airbnb's ESLint config with TypeScript support.
- ESLint - Contains the ESLint configuration used for projects maintained by the ESLint team.
- Facebook - Shareable config for Facebook's style guide.
- Google - Shareable config for the Google style.
- React App - Shareable config for React projects.

Awesome ESLint

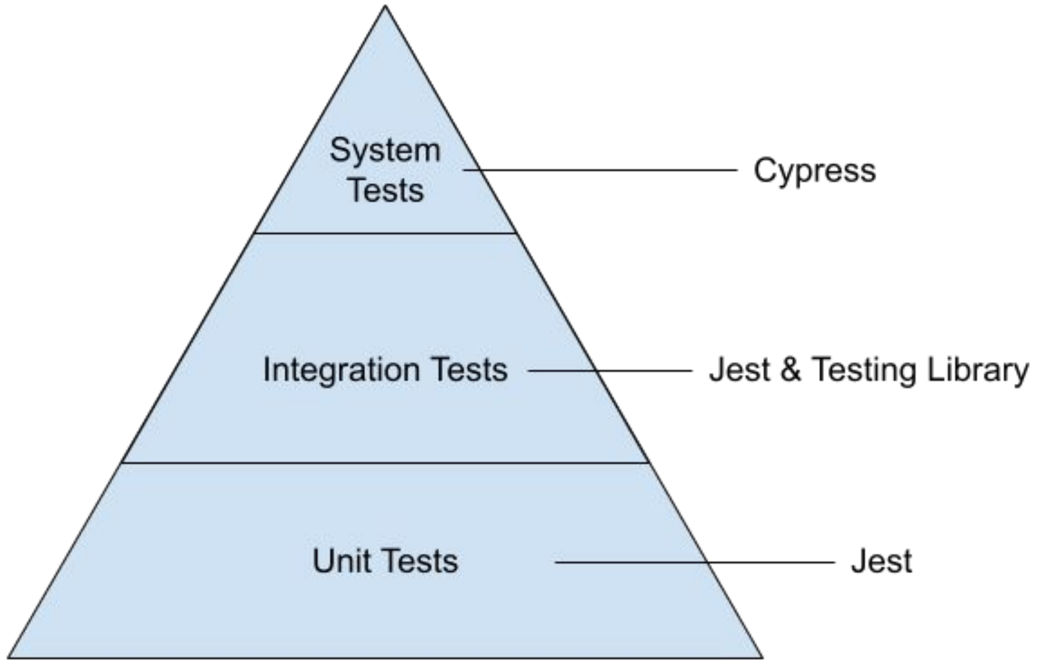
Testing

100 SECONDS OF
software
TESTING
FAIL
PASS

Testing Pyramid



Testing Pyramid: Technologies



Jest

Jest is a JS testing framework with following features:

- test runner
- code coverage generator
- mock functions
- matchers

```
const myMock = jest.fn();

beforeAll(() => {
   myMock
       .mockReturnValueOnce(10)
       .mockReturnValueOnce('x')
       .mockReturnValue(true);
});

beforeEach(() => {
   myMock.mockRestore();
});

test('check mock', () => {
   expect(myMock()).toBe(10);
   expect(myMock()).not.toBeUndefined();
   expect(myMock()).toBeTruthy();
});
```

Testing Library

The `@testing-library` family of packages helps you test UI components in a user-centric way.

React Testing Library builds on top of DOM Testing Library by adding APIs for working with React components.

Testing Library

React Testing Library

```
import {render, fireEvent, waitFor, screen} from '@testing-library/react'
import '@testing-library/jest-dom'

test('loads and displays greeting', async () => {
   render(<Fetch url="/greeting" />)

   fireEvent.click(screen.getByText('Load Greeting'))

   await waitFor(() => screen.getByRole('heading'))

   expect(screen.getByRole('heading')).toHaveTextContent('hello there')
   expect(screen.getByRole('button')).toBeDisabled()
})
```

Cypress

Cypress

Cypress was originally designed to run end-to-end (E2E) tests on anything that runs in a browser.

A typical E2E test visits the application in a browser and performs actions via the UI just like a real user would.

```
it('adds todos', () => {
   cy.visit('https://todo.app.com')
   cy.get('[data-testid="new-todo"]')
       .type('write code{enter}')
       .type('write tests{enter}')

   // confirm the application
   // is showing two items
   cy.get('[data-testid="todos"]')
       .should('have.length', 2)
})
```

Further readings

- https://martinfowler.com/articles/practical-test-pyramid.html
- https://feature-sliced.design/
- Test-Driven Development (Fireship video)
- https://github.com/aabounegm/cast