Axios

```
axios.get('http://...').then(...).catch((err)=>...)

axios.post('/student/', `name=${std.name}`);
```

JEST

function containing multiple tests

```
test(name, fn, timeout), describe(name, fn)

expect(value).<condition>
```

Primitive Values

```
expect(sum(4,2)).toBe(6) → Pass
```

Exceptions

```
expect(()=>sum(4,2)).toThrow();

test('exception attributes', ()=>{
  let err;
  expect(err).toBe(undefined);
  try{
    calc.div(6,0);
  }catch(e){
    err=e;
  }
  expect(err).not.toBe(undefined);
  expect(err.name).toBe('Error');
});
```

Objects

Partial Match

```
expect(o).toMatchObject({major:'History'})
expect(o).toStrict Equal({name:'J',...})
```

UI Testing

Testing-library: tests webpages by interacting w/ DOM nodes.

Component Rendering

```
import {render, screen} from '@testing-library/react';

test('renders Boston link', ()=>{
  render(<WeatherApp />);
  const linkElement = screen.getByText(/Boston/i);
  expect(linkElement).toBeInTheDocument();
});
```

Snapshot Testing

```
import renderer from 'react-test-renderer';

test('Weather app matches snapshot', ()=>{
  const component = renderer.create(<App/>);
  const tree = component.toJSON();
  expect(tree).toMatchSnapshot();
});
```

Events

```
test('typing into textbox', async()=>{
  render(<WeatherApp />);
  const element = screen.getByRole('textbox');
  await userEvent.type(element,'text');
  expect(element).toHaveValue('text');
});
```

Promise Testing

```
test('the city is Philly', async()=>{
  const city = 'Philadelphia';
  try{
    const data = await getWeather(city);
    expect(data.name).toBe('Philadelphia');
  }catch(err){
```

Testing

A test case: input, expected output, actual output.

False negative: test passes when it should fail.
False positive: test passes when it should pass.

Block Box testing

cover as many of the equivalence classes as possible.
Cover equivalence classes combinations as well. Single Fault Assumption
select test inputs on the boundary between equivalence classes.
Robustness testing
select test inputs that are legal but absurd. ie negative height.

White Box Testing

cover as much of your code as possible.
Graph: nodes: statements, edges represent transitions.
Statement Coverage: how many statements are covered?
Branch Coverage and Path Coverage defined similarly.

JEST Mocking

Manual Mock

```
jest.mock('<name_of_module>');
```

declare module to be mocked

```
<name_module>.<func_to_mock>.mockResolvedValue(...)
<name_module>.<func_to_mock>.mockRejectedValue(...)
```

Example

```
import GetWeather from '../api/getData';
jest.mock('../api/getData.js');
getWeather.mockResolvedValue({name:'Philly'})
test('temperature is correct', async()=>{
  const data = await getWeather('Philadelphia');
  expect(data.main.temp).toBe(70);
});
```

React Design

Each component should perform only one task. same schema
A component can display a collection of resources from schema
Components hierarchy: App at the top, parent-child.
Event Handling: `<textarea onClick={updateEvent} />`
Conditional Rendering: `if(login){return(...)}else{return()}`

Events

DOM and JS communicate through events → dispatched and handled.
Event Dispatcher: creates an event and propagates through DOM.
Event Listener: provides a callback method to handle event.
Event Object: type, timeStamp, target, eventPhase.
Event Types: keyboard, mouse, drag&drop, form events.

```
document.getElementById("btn").onclick = function(){...};
btn.addEventListener("mouseover", function(){...});
```

Event Propogation: Capture phase, Target phase, Bubbling phase.

```
element.addEventListener(event, handler, true) → Capture phase
```

JSON: lightweight format for storing and transporting data.
Data is in name/value pairs, seperated by commas.
JSON.Parse() takes JSON-formatted string ⇒ JSON object.
JSON.stringify() takes JSON object ⇒ returns string representation.
REST API: response and request data sent through JSON.

States and Hooks

State: mutating a state triggers the rerendering of the component.

```
const [city, setCity] = useState('');
```

Effect Hook: runs after DOM updates, performs side effects.
returns an optional function that performs cleanup operations.

```
useEffect(()=>{code}, [dependencyList]);
```

Lifting state up: used when child component triggers the rendering of a parent or sibling component.
Cons: requires nested components, many components rerendered.
useRef: an object that persists for the lifetime of the component.

```
const myCounter = useRef(0);  myCounter.current=...;
```

Async Programming

improves performance and responsiveness.

Program subroutines executed separately from main thread.
Thread: smallest sequence managed by a scheduler.
All threads share a common heap.
JS is single threaded.
Event loop: coordinates deferred operations.
The event loop has one or more task queues and processes messages as they arrive.
Callback function: used to continue code execution after an async operation has completed.
Promise: object returned by async operation to represent its state. avoids Pyramid of Doom.
Promise states: pending, fulfilled, rejected.

```
const p = new Promise((resolve,reject)=>{
  const c = parseInt('10');
  if(c==10){resolve("fulfilled");}
  else{reject(new Error("c is not 10"));}
```

Then / catch

```
Promise.then((value)=>code)
       .catch((reason)=>code);
```

Async / await

```
const f = async()=>{
  try{
    const val = await Promise;
  }catch(e){
    code;
  }
```

Benefits of Cross-Platform Mobile Development

Reduction of Required Skills: same programming lang.
Reduction of coding
Reduction of Development time and maintenance.
Decrement of API knowledge.
Greater ease of Development.

Single Page Applications

decoupling of frontend and backend.
easier to develop, test and deploy.
JS environment preserved during execution.
Client side routing.

Deep Linking

The URL must link to a specific resource not just the homepage.
The URL must be shareable and bookmarkable.

SPA provides URLs that capture state of app.

Deep Linking in SPA Two Approaches:

1) Update URL to capture state of App.
2) Use a share button to generate URLs.

SPA uses URL to restore the application state.

```
<Router>
  <Link to='http://...' onClick={}></Link>
  <Route path='http://...' render={()=><App/>}/>
</Router>
```

Agile Software Development

Focus on	Instead of
Individuals & Interactions	Processes & tools
Working Software	Comprehensive Documentation
Customer Collaboration	Contract Negotiation
Responding to Change	Following a Plan.

Agile Technical Principles

Automated
Refactor Mercilessly Pair Programming Tests.
User Story: each feature is described in few words.
Story Points: effort required to implement user story.
Project Velocity: story points per iteration.