1. Problem Statement

Case Study: NewsFleet – Real-Time Newsroom Dashboard

NewsFleet is a real-time newsroom dashboard:

- Editors can add, edit, and approve articles, each with live comment feeds and analytics.
- The app must be reliable: a bug could publish the wrong article or lose comments.
- Features are built by multiple teams—regressions and accidental breakage are a real risk.
- Fast iteration is key, but every deploy must be safe and bug-free.

CASE STUDY:
NEWSFLEET
REAL-TIME NEWSROOM DASHBOARD
CHALLENGE:
HOW DO YOU ENSURE EVERY FEATURE IS TESTED, EVERY BUG IS CAUGHT EARLY, AND THE CODEBASE REMAINS MAINTAINABLE AS NEWSFLEET GROWS?

The challenge:
How do you ensure every feature is tested, every bug is caught early, and the codebase remains maintainable as NewsFleet grows?

2. Learning Objectives

By the end of this tutorial, you will:

- Set up **Jest** for type-safe unit and integration testing in React/TypeScript.
- Write and organize tests for components, hooks, and business logic.
- Use **linting** (ESLint, Prettier, Biome) for code quality and consistency.
- Debug React apps efficiently with modern tools.

- Integrate testing and linting into your CI/deployment workflow.

---

3. Concept Introduction with Analogy

Analogy: The Newsroom’s Editorial Workflow

- **Jest tests** are like fact-checkers: They catch errors before articles (features) go live.
- **Linters** are like copy editors: They enforce style, consistency, and best practices in every story (file).
- **Debugging tools** are the newsroom’s review meetings: They help editors trace issues and fix them before publication.

---

4. Technical Deep Dive

A. Testing with Jest in TypeScript React Apps

1. Why Jest?

- Fast, zero-config, works with both frontend and backend TypeScript.
- Supports unit, integration, and snapshot testing.
- Huge ecosystem (React Testing Library, mocking, coverage).

2. Setting Up Jest

```
npm install --save-dev jest @types/jest ts-jest @testing-library/react @testing-library/jest-dom
```

- Add to `package.json`:

```
"scripts": {
  "test": "jest"
}
```

- Create `jest.config.js`:

```
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'jsdom',
  setupFilesAfterEnv: ['@testing-library/jest-dom/extend-expect']
};
```

3. Writing a Component Test

```
// components/ArticleCard.tsx
import React from 'react';

interface ArticleCardProps {
  title: string;
  author: string;
  onApprove: () => void;
}

export const ArticleCard: React.FC<ArticleCardProps> = ({ title, author, onApprove }) => (
  <div>
    <h2>{title}</h2>
    <p>By {author}</p>
    <button onClick={onApprove}>Approve</button>
  </div>
);
```

```
// __tests__/ArticleCard.test.tsx
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import { ArticleCard } from '../components/ArticleCard';

test('renders article title and author', () => {
  render(<ArticleCard title="Breaking News" author="Jane Doe" onApprove={() => {}} />);
  expect(screen.getByText('Breaking News')).toBeInTheDocument();
  expect(screen.getByText('By Jane Doe')).toBeInTheDocument();
});

test('calls onApprove when button is clicked', () => {
  const mockApprove = jest.fn();
  render(<ArticleCard title="Test" author="John" onApprove={mockApprove} />);
  fireEvent.click(screen.getByText('Approve'));
  expect(mockApprove).toHaveBeenCalled();
});
```

4. Testing Hooks and Business Logic

```
// hooks/useApproval.ts
import { useState } from 'react';
export function useApproval() {
  const [approved, setApproved] = useState(false);
  const approve = () => setApproved(true);
  return { approved, approve };
}

// __tests__/useApproval.test.ts
import { renderHook, act } from '@testing-library/react';

import { useApproval } from '../hooks/useApproval';

test('approves correctly', () => {
  const { result } = renderHook(() => useApproval());
  expect(result.current.approved).toBe(false);
  act(() => result.current.approve());
  expect(result.current.approved).toBe(true);
});
```

B. Linting: ESLint, Prettier, and Biome

1. Why Lint?

- Prevents bugs, enforces style, and ensures code consistency.
- Catches unused variables, type errors, and anti-patterns before testing or deployment.

2. Setting Up ESLint and Prettier

```
npm install --save-dev eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin prettier eslint-co
```

- `.eslintrc.js` example:

```
module.exports = {
  parser: '@typescript-eslint/parser',
  plugins: ['@typescript-eslint', 'react'],
  extends: [
    'eslint:recommended',
    'plugin:@typescript-eslint/recommended',
    'plugin:react/recommended',
    'prettier'
  ],
  rules: {
    'react/prop-types': 'off'
  }
};
```

- Add a `.prettierrc` for formatting preferences.

3. Biome (Optional Modern Linter)

- Biome is a new, fast alternative to ESLint/Prettier.

```
npm install --save-dev @biomejs/biome
```

- Add a `biome.json` config and run with `npx biome check .`.

C. Debugging React with TypeScript

- Use **React Developer Tools** for inspecting component state, props, and re-renders.
- Use **VSCode debugging**: set breakpoints in `.tsx` files, step through logic, and inspect variables.
- Use **console.log** and **Jest's debug output** for test failures.

D. Integrating Testing and Linting into CI

- Add `npm run test` and `npm run lint` to your CI pipeline (GitHub Actions, GitLab CI, etc.).
- Fail the build if tests or linting fail.
- Use coverage reports (`jest --coverage`) to track test completeness.

---

5. Step-by-Step Data Modeling & Code Walkthrough

A. Article Approval Workflow

```
// components/ArticleApproval.tsx
import React from 'react';
import { ArticleCard } from './ArticleCard';
import { useApproval } from '../hooks/useApproval';

export function ArticleApproval({ article }) {
  const { approved, approve } = useApproval();
  return (
    <div>
      <ArticleCard title={article.title} author={article.author} onApprove={approve} />
      {approved && <span>Approved!</span>}
    </div>
  );
}
```

Test:

```
// __tests__/ArticleApproval.test.tsx
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import { ArticleApproval } from '../components/ArticleApproval';

test('shows Approved! after clicking approve', () => {
  render(<ArticleApproval article={{ title: 'T1', author: 'A1' }} />);
  fireEvent.click(screen.getByText('Approve'));
  expect(screen.getByText('Approved!')).toBeInTheDocument();
});
```

B. Linting and Formatting Example

- Run `npx eslint .` and `npx prettier --check .` before every commit.
- Fix errors and warnings before pushing code.

C. Debugging Example

- Set a breakpoint in `useApproval` or `ArticleApproval` in VSCode.
- Use React DevTools to inspect the `approved` state as you interact with the UI.

---

6. Interactive Challenge / Mini-Project

Your Turn!

1. Write a test for a `CommentBox` component that:
    - Renders an input and a "Post" button.
    - Calls a provided `onPost` callback with the input value when clicked.
    - Clears the input after posting.
2. Add a lint rule that forbids `console.log` statements in production code.
3. Debug a failing test: The test expects "Approved!" to appear, but it doesn't—what could be wrong?

7. Common Pitfalls & Best Practices

Common Pitfalls & Best Practices (Testing & Code Quality)

Pitfall	Best Practice
Not testing edge cases	Write tests for empty, error, and boundary cases
Skipping linting/formatting	Run lint/format on every commit/CI
Ignoring test failures	Never merge code with failing tests
Not using coverage reports	Track and improve test completeness
Debugging only in browser	Use VSCode/IDE debuggers for TypeScript

8. Optional: Programmer’s Workflow Checklist

- Write tests for every new component and hook.
- Run lint and format on every commit.
- Use coverage reports to track test completeness.
- Debug with React DevTools and VSCode breakpoints.
- Integrate tests and linting into CI/CD pipelines.