# YourSpace Testing Suite

Comprehensive testing documentation for the YourSpace Creative Labs application.

## Overview

The YourSpace application uses a multi-layered testing approach to ensure reliability, performance, and user experience quality:

* **Unit Tests**: Component and function level testing
* **Integration Tests**: Testing component interactions and API integrations
* **End-to-End Tests**: Full user workflow testing
* **Visual Regression Tests**: UI consistency verification

## Testing Stack

### Unit & Integration Testing

* **Vitest**: Fast unit test runner
* **React Testing Library**: Component testing utilities
* **Jest DOM**: Additional DOM testing matchers
* **MSW** (Mock Service Worker): API mocking

### End-to-End Testing

* **Playwright**: Cross-browser E2E testing
* **Multiple browsers**: Chrome, Firefox, Safari, Mobile

### Code Coverage

* **V8 Coverage Provider**: Built-in coverage reporting
* **Codecov**: Coverage reporting and tracking

## Running Tests

### Unit Tests

# Run tests in watch mode  
npm run test  
  
# Run tests once  
npm run test:run  
  
# Run with coverage  
npm run test:coverage  
  
# Run with UI  
npm run test:ui

### End-to-End Tests

# Run E2E tests  
npm run e2e  
  
# Run with UI  
npm run e2e:ui  
  
# Run in headed mode  
npm run e2e:headed

### All Tests

# Run complete test suite  
npm run test:all

## Test Structure

### Unit Tests Location

src/  
 \_\_tests\_\_/  
 setup.ts # Test setup and global mocks  
 components/ # Component tests  
 Auth.test.tsx  
 MusicPlayer.test.tsx  
 ProfileBuilder.test.tsx  
 VirtualRoom.test.tsx  
 hooks/ # Custom hooks tests  
 CustomHooks.test.tsx  
 lib/ # Utility and API tests  
 Utils.test.ts  
 integration/ # Integration tests  
 App.test.tsx

### E2E Tests Location

tests/  
 e2e/  
 main.spec.ts # Main application E2E tests

## Test Coverage Requirements

The project maintains high code coverage standards:

* **Branches**: 80% minimum
* **Functions**: 80% minimum
* **Lines**: 80% minimum
* **Statements**: 80% minimum

## Writing Tests

### Unit Test Example

import { describe, it, expect, vi } from 'vitest';  
import { render, screen, fireEvent } from '@testing-library/react';  
import MyComponent from './MyComponent';  
  
describe('MyComponent', () => {  
 it('renders correctly', () => {  
 render(<MyComponent />);  
 expect(screen.getByText('Hello World')).toBeInTheDocument();  
 });  
  
 it('handles click events', () => {  
 const mockClick = vi.fn();  
 render(<MyComponent onClick={mockClick} />);  
   
 fireEvent.click(screen.getByRole('button'));  
 expect(mockClick).toHaveBeenCalled();  
 });  
});

### E2E Test Example

import { test, expect } from '@playwright/test';  
  
test('user can create a profile', async ({ page }) => {  
 await page.goto('/profile');  
   
 await page.click('text=Add Widget');  
 await page.fill('textarea[name="content"]', 'My bio content');  
 await page.click('button[type="submit"]');  
   
 await expect(page.locator('text=Profile saved')).toBeVisible();  
});

## Mocking Guidelines

### Supabase Mocking

vi.mock('../lib/supabase', () => ({  
 supabase: {  
 auth: {  
 getUser: vi.fn(),  
 signIn: vi.fn(),  
 signOut: vi.fn(),  
 },  
 from: vi.fn(() => ({  
 select: vi.fn().mockReturnThis(),  
 insert: vi.fn().mockReturnThis(),  
 // ... other methods  
 })),  
 },  
}));

### Context Providers

const renderWithProviders = (component: React.ReactElement) => {  
 return render(  
 <BrowserRouter>  
 <AuthContext.Provider value={mockAuthContext}>  
 <MusicPlayerContext.Provider value={mockMusicContext}>  
 {component}  
 </MusicPlayerContext.Provider>  
 </AuthContext.Provider>  
 </BrowserRouter>  
 );  
};

## CI/CD Integration

Tests are automatically run on: - Every push to main and develop branches - All pull requests - Multiple Node.js versions (18.x, 20.x) - Multiple browsers for E2E tests

### GitHub Actions Workflow

1. **Lint Check**: Code style verification
2. **Unit Tests**: Component and function testing
3. **Coverage Report**: Upload to Codecov
4. **E2E Tests**: Cross-browser testing
5. **Build Verification**: Ensure app builds successfully
6. **Security Scan**: Vulnerability detection

## Test Data Management

### Mock Data

Mock data is centralized and reusable:

export const mockUser = {  
 id: 'user-1',  
 email: 'test@example.com',  
 profile: {  
 username: 'testuser',  
 avatar\_url: '/avatar.jpg'  
 }  
};  
  
export const mockTrack = {  
 id: '1',  
 title: 'Test Track',  
 artist: 'Test Artist',  
 duration: 180,  
 url: '/test-track.mp3'  
};

### Test Database

For integration tests requiring database state: - Use Supabase test instance - Clean database between test runs - Seed with consistent test data

## Performance Testing

### Load Testing

# Install artillery for load testing  
npm install -g artillery  
  
# Run load tests  
artillery run load-test-config.yml

### Bundle Analysis

# Analyze bundle size  
npm run build  
npx vite-bundle-analyzer dist

## Accessibility Testing

E2E tests include accessibility checks:

test('page is accessible', async ({ page }) => {  
 await page.goto('/profile');  
   
 const accessibilityScanResults = await new AxePuppeteer(page).analyze();  
 expect(accessibilityScanResults.violations).toEqual([]);  
});

## Debugging Tests

### Unit Tests

# Debug with Vitest UI  
npm run test:ui  
  
# Debug specific test  
npm test -- --reporter=verbose MyComponent.test.tsx

### E2E Tests

# Debug with Playwright UI  
npm run e2e:ui  
  
# Debug with headed browser  
npm run e2e:headed  
  
# Debug specific test  
npx playwright test --debug main.spec.ts

## Best Practices

1. **Test Naming**: Use descriptive test names that explain the expected behavior
2. **Arrange-Act-Assert**: Structure tests clearly
3. **Mock External Dependencies**: Isolate units under test
4. **Test User Behavior**: Focus on what users actually do
5. **Maintain Test Data**: Keep test data consistent and realistic
6. **Clean Up**: Ensure tests don’t affect each other
7. **Fast Tests**: Prefer unit tests for speed, E2E for critical paths
8. **Continuous Feedback**: Run tests early and often

## Troubleshooting

### Common Issues

**Flaky Tests** - Use proper wait conditions in E2E tests - Mock time-dependent functionality - Ensure proper cleanup between tests

**Slow Tests** - Optimize database queries in integration tests - Use appropriate test parallelization - Mock heavy external dependencies

**Environment Issues** - Ensure proper environment variables are set - Use consistent Node.js versions - Clear node\_modules and reinstall if needed

## Contributing

When adding new features: 1. Write tests first (TDD approach) 2. Ensure all tests pass 3. Maintain or improve coverage 4. Update test documentation 5. Add E2E tests for critical user journeys

For questions about testing, check the team documentation or reach out to the development team.