← Advanced Cypress & API Testing

BrowserStack Integration →

Mobile Testing with WebDriverIO

Table of Contents

Mobile Testing Concepts and Challenges

Types of Mobile Applications

1. Native Applications

- Built specifically for a platform (iOS/Android)
- Written in platform-specific languages (Swift/Kotlin)
- Direct access to device features

Examples: WhatsApp, Instagram, Uber

2. Hybrid Applications

- Web content wrapped in native container
- Built with HTML, CSS, JavaScript
- Uses frameworks like Ionic, React Native, Flutter
- Single codebase for multiple platforms
- Access to device features via plugins

Examples: Gmail, Twitter

3. Web Applications

- Mobile-optimized websites
- Run in mobile browsers
- No installation required
- Platform-independent
- Limited device access

Examples: Mobile versions of websites

Mobile Testing Challenges

Device Fragmentation:

```
 Android: 1000+ device models
iOS: 20+ device models
Screen sizes: 4" to 7"
OS versions: Multiple versions in use
Manufacturers: Samsung, Google, Huawei, etc.
```

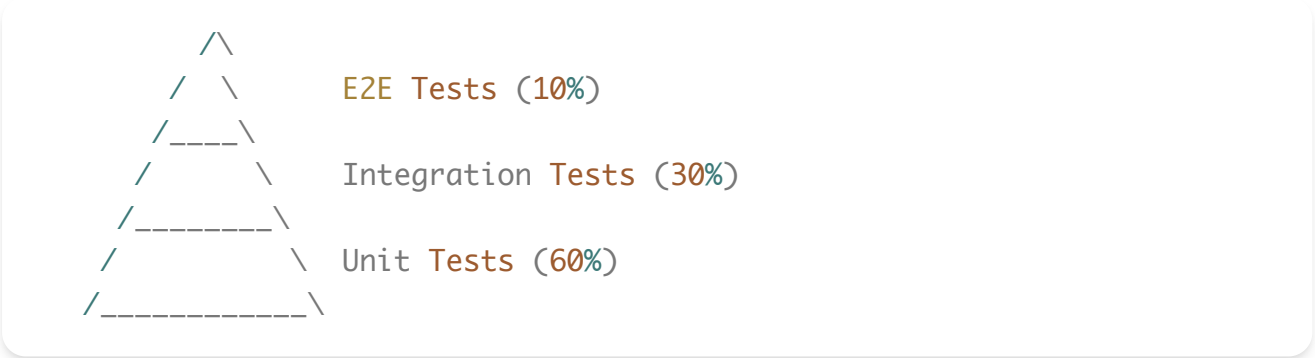
Common Challenges:

- Screen size variations

- Battery constraints
- Touch gestures complexity
- Device permissions
- App background/foreground states
- Orientation changes
- Interruptions (calls, notifications)

Mobile Testing Strategy

Testing Pyramid for Mobile:



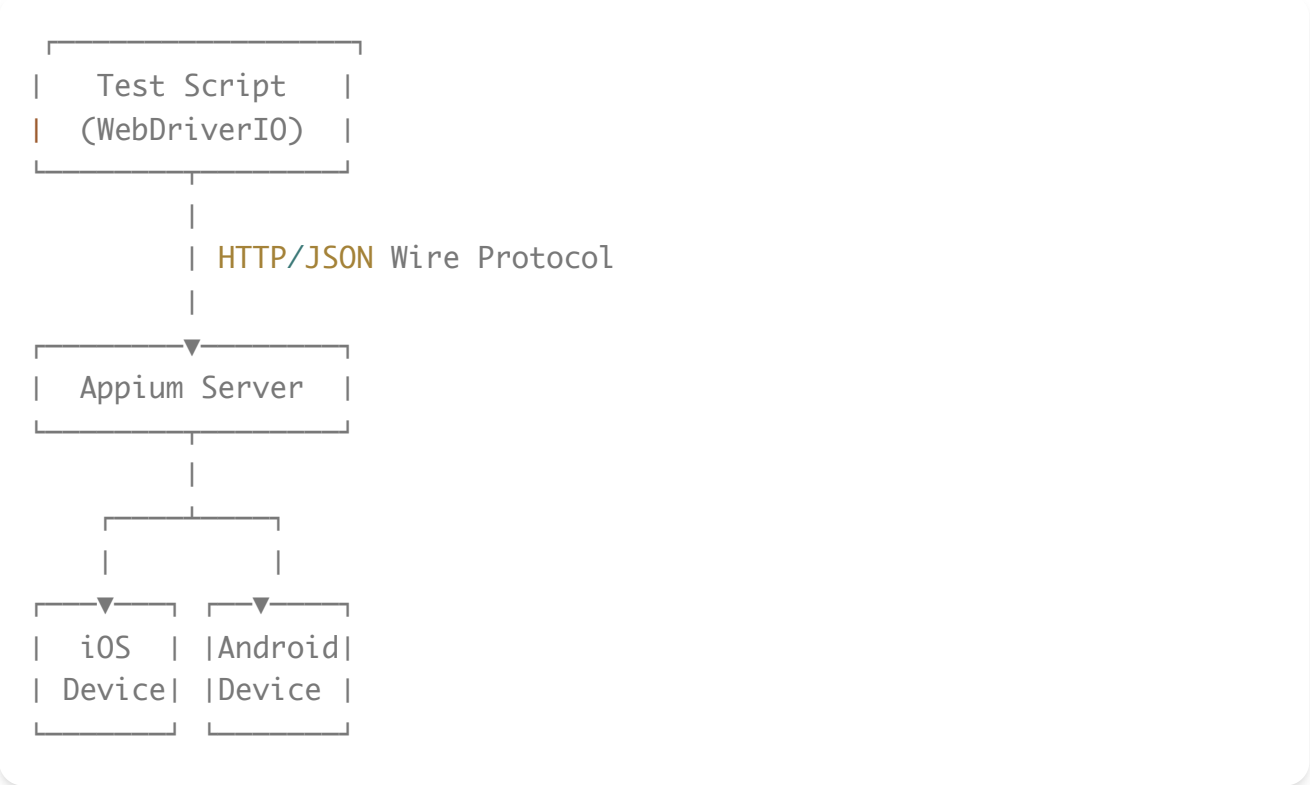
What to Test:

- Functionality across devices
- UI responsiveness
- Performance
- Network handling
- Battery usage
- Security
- Gestures and touch
- Permissions
- Background/foreground
- Interruptions

Appium Overview

- Cross-platform (iOS, Android)
- Multiple programming languages
- WebDriver protocol
- No app modification needed
- Open source

Architecture:



Introduction to WebDriverIO

What is WebDriverIO?

WebDriverIO is a modern automation framework for web and mobile applications.

Key Features:

- Built on WebDriver protocol

- Excellent documentation
- Active community
- Cloud service integration

Installation

Install WebDriverIO:

```
 # Create project
mkdir mobile-testing
cd mobile-testing

# Initialize npm
npm init -y

# Install WebDriverIO CLI
npm install --save-dev @wdio/cli

# Generate configuration
npx wdio config
```

Configuration wizard answers:

```
 ? Where is your automation backend located? On my local machine
? Which framework do you want to use? Mocha
? Do you want to use a compiler? TypeScript
? Where are your test specs located? ./test/specs/**/*.ts
? Which reporter do you want to use? spec
? Do you want to add a plugin to your test setup?
? Do you want to add a service to your test setup? appium
```

Project Structure

Generated structure:

```
 mobile-testing/
├── test/
|   ├── specs/
```

```
├── wdio.conf.ts
├── package.json
└── tsconfig.json
```

Basic WebDriverIO Test

Simple test:

```
// test/specs/basic.e2e.ts
describe('My App', () => {
  it('should have correct title', async () => {
    await browser.url('https://example.com')
    const title = await browser.getTitle()
    expect(title).toBe('Example Domain')
  })

  it('should find element', async () => {
    await browser.url('https://example.com')
    const heading = await $('h1')
    const text = await heading.getText()
    expect(text).toContain('Example')
  })
})
```

Run tests:

```
npx wdio run wdio.conf.ts
```

WebDriverIO Syntax

Browser object:

```
// Navigation
await browser.url('https://example.com')
await browser.back()
await browser.forward()
await browser.refresh()
```

```
const size = await browser.getWindowSize()

// Wait
await browser.pause(1000)
await browser.waitUntil(async () => {
  return await $('#element').isDisplayed()
}, { timeout: 5000 })
```

Element selectors:

```
// Single element
const element = await $('selector')
const button = await $('button')
const input = await $('#username')

// Multiple elements
const elements = await $$('selector')
const buttons = await $$('button')

// By text
const link = await $('=Link Text')
const partialLink = await $('*=Partial')

// XPath
const element = await $('//button[@type="submit"]')

// Accessibility ID (mobile)
const element = await $('~accessibilityId')

// Chain selectors
const element = await $('.container').$('button')
```

Element actions:

```
// Click
await button.click()

// Type
await input.setValue('text')
await input.addValue('more text')
await input.clearValue()
```

```
const displayed = await element.isDisplayed()
const enabled = await element.isEnabled()

// Wait for element
await element.waitForDisplayed({ timeout: 5000 })
await element.waitForEnabled()
await element.waitForClickable()
```

Assertions:

```
 // Expect (built-in)
await expect(browser).toHaveTitle('Example')
await expect(element).toBeDisplayed()
await expect(element).toHaveText('Expected Text')
await expect(element).toHaveValue('value')

// Custom assertions
const text = await element.getText()
expect(text).toBe('Expected')
expect(text).toContain('Exp')
```

BrowserStack App Automate Setup

BrowserStack Overview

BrowserStack App Automate provides real devices in the cloud for mobile testing.

Benefits:

- Real devices (not emulators)
- Multiple OS versions
- Various manufacturers
- Parallel execution
- Video recordings
- Debug tools

```
npm install --save-dev @wdio/browserstack-service
```

Configuration:

```
// wdio.conf.ts
export const config = {
  user: process.env.BROWSERSTACK_USERNAME,
  key: process.env.BROWSERSTACK_ACCESS_KEY,

  services: [
    ['browserstack', {
      app: process.env.BROWSERSTACK_APP_ID,
      buildIdentifier: '${BUILD_NUMBER}',
      browserstackLocal: true,
    }]
  ],

  capabilities: [{
    'bstack:options': {
      deviceName: 'Samsung Galaxy S21',
      platformVersion: '11.0',
      platformName: 'Android',
      app: 'bs://app-id-here',
      projectName: 'Mobile Testing Project',
      buildName: 'Build 1',
      sessionName: 'Test Session',
      debug: true,
      networkLogs: true,
      video: true,
    }
  }],

  maxInstances: 5,
  logLevel: 'info',
  bail: 0,
  waitforTimeout: 10000,
  connectionRetryTimeout: 120000,
  connectionRetryCount: 3,

  framework: 'mocha',
  reporters: ['spec'],
```

```
    timeout: 60000
  }
}
```

Upload App to BrowserStack

Upload via API:

```
curl -u "USERNAME:ACCESS_KEY" \
  -X POST "https://api-cloud.browserstack.com/app-automate/upload" \
  -F "file=@/path/to/app.apk"
```

Response:

```
{
  "app_url": "bs://c700ce60cf13ae8ed97705a55b8e022f13c5827c",
  "custom_id": "MyApp",
  "shareable_id": "username/MyApp"
}
```

Use in config:

```
capabilities: [{
  'bstack:options': {
    app: 'bs://c700ce60cf13ae8ed97705a55b8e022f13c5827c'
  }
}]
```

Environment Variables

Create .env file:

```
BROWSERSTACK_USERNAME=your_username
BROWSERSTACK_ACCESS_KEY=your_access_key
BROWSERSTACK_APP_ID=bs://app-id
```

```
import * as dotenv from 'dotenv'
dotenv.config()

export const config = {
  user: process.env.BROWSERSTACK_USERNAME,
  key: process.env.BROWSERSTACK_ACCESS_KEY,
}
```

Multiple Device Configuration

Parallel testing:

```
capabilities: [
  {
    'bstack:options': {
      deviceName: 'Samsung Galaxy S21',
      platformVersion: '11.0',
      platformName: 'Android',
      app: process.env.BROWSERSTACK_APP_ID,
    }
  },
  {
    'bstack:options': {
      deviceName: 'iPhone 13',
      platformVersion: '15',
      platformName: 'iOS',
      app: process.env.BROWSERSTACK_APP_ID,
    }
  },
  {
    'bstack:options': {
      deviceName: 'Google Pixel 6',
      platformVersion: '12',
      platformName: 'Android',
      app: process.env.BROWSERSTACK_APP_ID,
    }
  }
]
```

BrowserStack Local Testing

```
  services: [
    ['browserstack', {
      browserstackLocal: true,
      opts: {
        forceLocal: true,
      }
    }]
]
```

Mobile Locator Strategies

Appium Inspector

Install Appium Inspector:

- Download from: https://github.com/appium/appium-inspector
- Connect to Appium server
- Inspect element properties
- Get locator recommendations

Locator Types

1. Accessibility ID

```
 // Best practice for mobile
// iOS: accessibility identifier
// Android: content-desc

const element = await $('~login-button')
const username = await $('~username-field')
```

2. ID

```
const element = await $('#com.example.app:id/login_button')
const input = await $('#username')
```

3. Class Name

```
 const button = await $('android.widget.Button')
const textView = await $('android.widget.TextView')

// iOS
const button = await $('XCUIElementTypeButton')
const textField = await $('XCUIElementTypeTextField')
```

4. XPath

```
 // By text
const element = await $('//android.widget.Button[@text="Login"]')

// By resource-id
const input = await $('//*[@resource-id="username"]')

// By content-desc
const button = await $('//*[@content-desc="Submit"]')

// Complex
const element = await $('//android.widget.LinearLayout/android.widget.Butt
```

5. Text (exact)

```
 const button = await $('=Login')
const link = await $('=Sign Up')
```

6. Partial Text

```
 const button = await $('*=Log')  // Matches "Login", "Logout"
const element = await $('*=Submit')
```

```
// By text
const element = await $('android=new UiSelector().text("Login")')

// By description
const button = await $('android=new UiSelector().description("Submit")')

// By resource ID
const input = await $('android=new UiSelector().resourceId("username")')

// By class
const button = await $('android=new UiSelector().className("android.widget

// Complex
const element = await $('android=new UiSelector().text("Login").enabled(tr
```

8. iOS Predicate String

```
// By label
const element = await $('ios=label == "Login"')

// By name
const button = await $('ios=name == "Submit"')

// By value
const element = await $('ios=value == "Test"')

// Complex
const element = await $('ios=label CONTAINS "Log" AND visible == 1')
```

9. iOS Class Chain

```
const element = await $('-ios class chain:**/XCUIElementTypeButton[`label
```

Locator Best Practices

Priority order:

1. Accessibility ID (best for cross-platform)
2. ID/Resource ID

Examples:

```
// ✅ Good - Accessibility ID
const loginBtn = await $('~login-button')

// ✅ Good - Resource ID
const username = await $('#com.app:id/username')

// ⚠️ Okay - Text (may change with localization)
const button = await $('=Login')

// ❌ Avoid - Complex XPath (brittle)
const element = await $('//android.widget.LinearLayout[1]/android.widget.B
```

Finding Multiple Elements

```
// Get all elements
const buttons = await $$('android.widget.Button')

// Iterate
for (const button of buttons) {
  const text = await button.getText()
  console.log(text)
}

// Filter
const enabledButtons = []
for (const button of buttons) {
  if (await button.isEnabled()) {
    enabledButtons.push(button)
  }
}

// Count
const count = buttons.length
expect(count).toBeGreaterThan(0)
```

Element Hierarchy Navigation

```
// Get next sibling
const next = await element.nextElement()

// Get previous sibling
const prev = await element.previousElement()

// Find child
const child = await parent.$('.child-selector')
const children = await parent.$$('.child-selector')
```

Touch Gestures and Mobile Interactions

Basic Touch Actions

Tap:

```
 const button = await $('~login-button')
await button.click()

// With coordinates
await browser.touchAction({
  action: 'tap',
  x: 100,
  y: 200
})
```

Long Press:

```
 await browser.touchAction({
  action: 'longPress',
  x: 100,
  y: 200,
  duration: 2000  // milliseconds
})
```

```
await browser.touchAction([
  { action: 'press', x: 100, y: 200 },
  { action: 'wait', ms: 1000 },
  { action: 'release' }
])
```

Swipe Gestures

Horizontal swipe:

```
// Swipe right to left
await browser.touchAction([
  { action: 'press', x: 300, y: 500 },
  { action: 'wait', ms: 500 },
  { action: 'moveTo', x: 100, y: 500 },
  { action: 'release' }
])

// Swipe left to right
await browser.touchAction([
  { action: 'press', x: 100, y: 500 },
  { action: 'wait', ms: 500 },
  { action: 'moveTo', x: 300, y: 500 },
  { action: 'release' }
])
```

Vertical swipe:

```
// Swipe up
await browser.touchAction([
  { action: 'press', x: 200, y: 800 },
  { action: 'wait', ms: 500 },
  { action: 'moveTo', x: 200, y: 200 },
  { action: 'release' }
])

// Swipe down
await browser.touchAction([
  { action: 'press', x: 200, y: 200 },
  { action: 'wait', ms: 500 },
  { action: 'moveTo', x: 200, y: 800 },
```

Swipe helper function:

```
async function swipe(direction: 'up' | 'down' | 'left' | 'right') {
  const { width, height } = await browser.getWindowSize()
  const centerX = width / 2
  const centerY = height / 2

  let startX, startY, endX, endY

  switch (direction) {
    case 'up':
      startX = centerX
      startY = height * 0.8
      endX = centerX
      endY = height * 0.2
      break
    case 'down':
      startX = centerX
      startY = height * 0.2
      endX = centerX
      endY = height * 0.8
      break
    case 'left':
      startX = width * 0.8
      startY = centerY
      endX = width * 0.2
      endY = centerY
      break
    case 'right':
      startX = width * 0.2
      startY = centerY
      endX = width * 0.8
      endY = centerY
      break
  }

  await browser.touchAction([
    { action: 'press', x: startX, y: startY },
    { action: 'wait', ms: 500 },
    { action: 'moveTo', x: endX, y: endY },
    { action: 'release' }
  ])
}
```

```
await swipe('left')
```

Scroll Actions

Scroll to element:

```
const element = await $('~target-element')
await element.scrollIntoView()
```

Scroll in container:

```
const scrollView = await $('~scroll-view')
await scrollView.scroll({ direction: 'down' })
```

Scroll until element found:

```
async function scrollToElement(selector: string, maxScrolls = 10) {
  for (let i = 0; i < maxScrolls; i++) {
    try {
      const element = await $(selector)
      if (await element.isDisplayed()) {
        return element
      }
    } catch (e) {
      // Element not found, continue scrolling
    }

    await swipe('up')
    await browser.pause(500)
  }

  throw new Error(`Element ${selector} not found after ${maxScrolls} scrol
}

// Usage
const element = await scrollToElement('~target-element')
await element.click()
```

```
await browser.execute('mobile: pinchClose', {
  element: await element.elementId,
  percent: 50,
  steps: 50
})
```

Zoom (pinch open):

```
await browser.execute('mobile: pinchOpen', {
  element: await element.elementId,
  percent: 200,
  steps: 50
})
```

Drag and Drop

```
const source = await $('~drag-source')
const target = await $('~drop-target')

const sourceLocation = await source.getLocation()
const targetLocation = await target.getLocation()

await browser.touchAction([
  { action: 'press', x: sourceLocation.x, y: sourceLocation.y },
  { action: 'wait', ms: 500 },
  { action: 'moveTo', x: targetLocation.x, y: targetLocation.y },
  { action: 'release' }
])
```

Multi-Touch Gestures

```
// Two-finger tap
await browser.multiTouchAction([
  [{ action: 'tap', x: 100, y: 200 }],
  [{ action: 'tap', x: 200, y: 200 }]
])
```

```
  [
    { action: 'press', x: 100, y: 300 },
    { action: 'moveTo', x: 50, y: 300 },
    { action: 'release' }
  ],
  [
    { action: 'press', x: 200, y: 300 },
    { action: 'moveTo', x: 250, y: 300 },
    { action: 'release' }
  ]
])
```

Mobile-Specific Test Scenarios

App Lifecycle

Launch app:

```
await browser.launchApp()
```

Close app:

```
await browser.closeApp()
```

Background app:

```
// Put app in background for 5 seconds
await browser.background(5)
```

Reset app:

```
await browser.reset()
```

```
 await browser.terminateApp('com.example.app')
await browser.activateApp('com.example.app')
```

Device Orientation

Get orientation:

```
 const orientation = await browser.getOrientation()
console.log(orientation)  // 'PORTRAIT' or 'LANDSCAPE'
```

Set orientation:

```
 await browser.setOrientation('LANDSCAPE')
await browser.pause(1000)  // Wait for UI to adjust

// Back to portrait
await browser.setOrientation('PORTRAIT')
```

Test both orientations:

```
describe('Orientation Tests', () => {
  const orientations = ['PORTRAIT', 'LANDSCAPE'] as const

  orientations.forEach((orientation) => {
    it(`should work in ${orientation}`, async () => {
      await browser.setOrientation(orientation)
      await browser.pause(1000)

      const element = await $('~main-button')
      await expect(element).toBeDisplayed()
    })
  })
})
```

Keyboard Interactions

Show/Hide keyboard:

```
// Hide keyboard
await browser.hideKeyboard()

// Or press Done/Return
await browser.pressKeyCode(66)  // Android Enter key
```

Type text:

```
 const input = await $('~username')
await input.setValue('testuser')
```

Clear text:

```
 await input.clearValue()
```

Network Conditions

Airplane mode:

```
 // Enable airplane mode
await browser.execute('mobile: setConnectivity', {
  wifi: false,
  data: false
})

// Disable airplane mode
await browser.execute('mobile: setConnectivity', {
  wifi: true,
  data: true
})
```

Network speed (Android):

```
 await browser.execute('mobile: networkSpeed', {
  netspeed: 'gprs'  // 'gsm', 'gprs', 'edge', '3g', 'lte'
})
```

```
// Android
await browser.execute('mobile: shell', {
  command: 'pm',
  args: ['grant', 'com.example.app', 'android.permission.CAMERA']
})

// iOS
await browser.execute('mobile: setPermission', {
  bundleId: 'com.example.app',
  permission: 'camera',
  value: 'yes'
})
```

Notifications

Handle notification:

```
// Get notifications (Android)
const notifications = await browser.execute('mobile: getNotifications')

// Open notification bar
await browser.openNotifications()

// Close notification bar
await browser.execute('mobile: closeNotifications')
```

Deep Links

Open deep link:

```
await browser.execute('mobile: deepLink', {
  url: 'myapp://screen/login',
  package: 'com.example.app'
})
```

Camera and Gallery

```
await browser.execute('mobile: takeScreenshot')
```

Select from gallery:

```
// This requires interaction with system UI
const galleryButton = await $('~gallery-button')
await galleryButton.click()

// Navigate system gallery
await browser.pause(2000)
// Select image...
```

Biometric Authentication

Simulate fingerprint (Android):

```
await browser.fingerPrint(1)  // Successful auth
```

Simulate Face ID (iOS):

```
await browser.execute('mobile: enrollBiometric', { isEnabled: true })
await browser.execute('mobile: sendBiometricMatch', { type: 'faceId', matc
```

Debugging Mobile Tests

Appium Logs

Enable verbose logging:

```
// wdio.conf.ts
export const config = {
  logLevel: 'debug',
```

```
      args: {
        log: './appium.log',
        logLevel: 'debug'
      }
    }]
  ]
}
```

View logs:

```
tail -f ./logs/appium.log
```

Screenshots

Take screenshot:

```
await browser.saveScreenshot('./screenshots/error.png')
```

On failure:

```
// wdio.conf.ts
afterTest: async function(test, context, { error, result, duration, passed
  if (!passed) {
    await browser.saveScreenshot(`./screenshots/FAILED_${test.title}.png`)
  }
}
```

Element Inspection

Get element info:

```
const element = await $('~button')
const displayed = await element.isDisplayed()
const enabled = await element.isEnabled()
const text = await element.getText()
const location = await element.getLocation()
const size = await element.getSize()
```

```
  enabled,
  text,
  location,
  size
})
```

Page source:

```
 const source = await browser.getPageSource()
console.log(source)
```

Wait Strategies

Wait for element:

```
 const element = await $('~button')

// Wait for displayed
await element.waitForDisplayed({ timeout: 10000 })

// Wait for enabled
await element.waitForEnabled()

// Wait for exist
await element.waitForExist()

// Wait until condition
await browser.waitUntil(async () => {
  const text = await element.getText()
  return text.includes('Expected')
}, {
  timeout: 10000,
  timeoutMsg: 'Element did not contain expected text'
})
```

Debug Mode

Pause execution:

In test:

```
it('debug test', async () => {
  await browser.url('/')
  await browser.debug()  // REPL opens
  // Type commands to debug
  const element = await $('~button')
  await element.click()
})
```

Video Recording (BrowserStack)

Enable in config:

```
capabilities: [{
  'bstack:options': {
    video: true,
    debug: true,
    networkLogs: true
  }
}]
```

View in dashboard:

- Login to BrowserStack
- Go to App Automate dashboard
- View test session
- Play video recording

Practice Exercises

Exercise 1: Basic Mobile Navigation

```
describe('App Navigation', () => {
  it('should navigate between screens', async () => {
    // 1. Launch app
    // 2. Verify home screen
    // 3. Click menu button
    // 4. Navigate to settings
    // 5. Verify settings screen
    // 6. Go back to home

    // Your implementation
  })

  it('should test tab navigation', async () => {
    // 1. Click each tab
    // 2. Verify correct screen loads
    // 3. Verify tab is selected

    // Your implementation
  })
})
```

Exercise 2: Form Input

Task: Test form interactions

```
describe('Login Form', () => {
  it('should login successfully', async () => {
    // 1. Find username field
    // 2. Type username
    // 3. Find password field
    // 4. Type password
    // 5. Hide keyboard
    // 6. Click login button
    // 7. Verify logged in

    // Your implementation
  })

  it('should show validation errors', async () => {
    // 1. Click login without filling
    // 2. Verify error messages
    // 3. Fill fields
    // 4. Verify errors disappear
```

```
})
```

Exercise 3: Scroll and Swipe

Task: Test scrolling and gestures

```
describe('Scroll and Swipe', () => {
  it('should scroll to element', async () => {
    // 1. Navigate to list screen
    // 2. Scroll until element visible
    // 3. Click element
    // 4. Verify action

    // Your implementation
  })

  it('should swipe between pages', async () => {
    // 1. Navigate to carousel
    // 2. Swipe left multiple times
    // 3. Verify page changes
    // 4. Swipe right
    // 5. Verify going back

    // Your implementation
  })
})
```

Exercise 4: Orientation Testing

Task: Test landscape and portrait

```
describe('Orientation', () => {
  const orientations = ['PORTRAIT', 'LANDSCAPE'] as const

  orientations.forEach((orientation) => {
    it(`should display correctly in ${orientation}`, async () => {
      // 1. Set orientation
      // 2. Wait for UI adjustment
      // 3. Verify layout
      // 4. Test interactions
```

```
  })
})
```

Exercise 5: Multiple Devices

Task: Test on different devices

```
 // Configure multiple devices in wdio.conf.ts
// Run same tests on all devices

describe('Cross-Device Tests', () => {
  it('should work on all devices', async () => {
    // 1. Get device capabilities
    // 2. Perform actions
    // 3. Verify results
    // 4. Adjust for device differences if needed

    // Your implementation
  })
})
```

Exercise 6: App Lifecycle

Task: Test app background/foreground

```
 describe('App Lifecycle', () => {
  it('should handle background', async () => {
    // 1. Perform action
    // 2. Put app in background (5 seconds)
    // 3. Verify state preserved
    // 4. Continue workflow

    // Your implementation
  })

  it('should handle app restart', async () => {
    // 1. Perform action
    // 2. Close app
    // 3. Launch app
    // 4. Verify state (logged out or persisted)
```

```
})
```

Exercise 7: Hybrid Testing

Task: Test both native and web views

```
describe('Hybrid App', () => {
  it('should switch between contexts', async () => {
    // 1. Get available contexts
    // 2. Switch to web view
    // 3. Interact with web elements
    // 4. Switch back to native
    // 5. Interact with native elements

    // Your implementation
  })
})
```

Summary

In Day 10, you mastered:

✅ Mobile Testing Concepts

- Native, Hybrid, Web apps
- Device fragmentation
- Mobile testing challenges
- Testing strategies

✅ WebDriverIO

- Installation and setup
- Configuration
- Basic syntax

- App upload
- Device configuration
- Parallel testing
- Local testing

✅ Mobile Locators

- Accessibility ID
- ID/Resource ID
- XPath for mobile
- UIAutomator/Predicate
- Best practices

✅ Touch Gestures

- Tap, long press
- Swipe (all directions)
- Scroll patterns
- Pinch/zoom
- Drag and drop
- Multi-touch

✅ Mobile Scenarios

- App lifecycle
- Orientation changes
- Keyboard handling
- Network conditions
- Permissions
- Deep links
- Biometric auth

- Screenshots
- Element inspection
- Wait strategies
- Video recording

Key Takeaways

- **Mobile testing** requires different strategies
- **WebDriverIO** provides powerful automation
- **BrowserStack** enables real device testing
- **Accessibility ID** is best locator strategy
- **Touch gestures** are mobile-specific
- **App lifecycle** must be tested
- **Multiple devices** ensure compatibility

Best Practices

1. Use Accessibility IDs
2. Test on real devices
3. Handle orientation changes
4. Test app lifecycle states
5. Implement wait strategies
6. Take screenshots on failure
7. Test multiple devices/OS versions
8. Handle interruptions
9. Test network conditions
10. Verify permissions

Mobile Testing Checklist

- ☐ Functionality on multiple devices

- [ ] Various OS versions
- [ ] Network conditions (WiFi, 3G, offline)
- [ ] Interruptions (calls, notifications)
- [ ] Background/foreground
- [ ] App permissions
- [ ] Touch gestures
- [ ] Keyboard interactions

Next Steps

Week 3 begins! Topics:

- Day 11: BrowserStack & Cloud Testing
- Day 12: AWS for Test Automation
- Day 13: CI/CD & Test Reporting
- Day 14: AI in Test Automation
- Day 15: Framework Design & Capstone

End of Day 10 Documentation End of Week 2! 🎉

← Advanced Cypress & API Testing

BrowserStack Integration →