

Cypress

[What is end to end testing | Cypress introduction \(Youtube\)](#)

It provides a browser with head not a headless browser.

We can use Cypress to test any web application built on any type of technology.

There is not feature to test things on separate tab like testing third party login.

[Cypress End-to-End Testing \(Youtube\)](#)

It has four key folders – fixtures, integrations, support and plugins.

Best practices – we should avoid using id's or css classes to select element from the DOM, it will make the test brittle, because those things are likely to change. We should either use data attribute or actual component name itself.

```
// Fill out the form
cy.get('input[name=email]').type(email);
cy.get('input[name=password]').type(pass);
cy.get('button[type=submit]').click();
```

[Introduction to automation testing with Cypress.io \(Non-selenium framework\) \(Youtube\)](#)

Whereas selenium executes remote commands through the network, Cypress runs in the same run-loop as our application. Other tools like protector uses selenium under the hood unlike Cypress.

With Cypress, all related things are available out of the box –

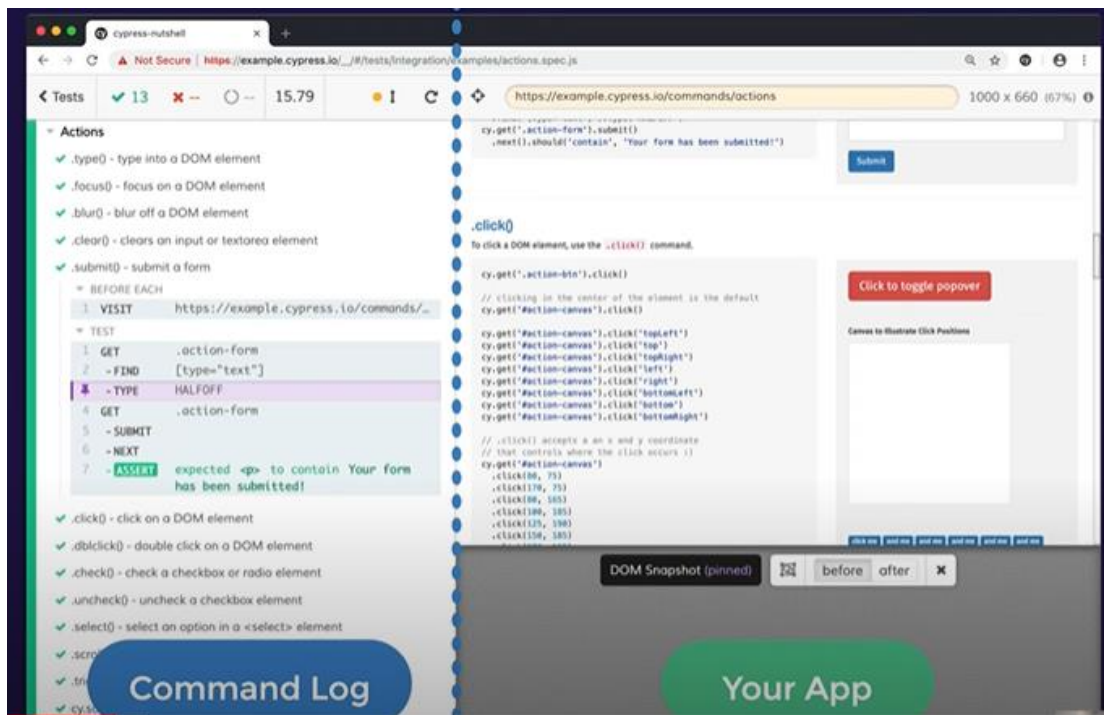


It provides features like real time loading, time travel, consistent results.

[Cypress in a Nutshell \(Youtube\)](#)

It is a tool for reliably testing anything that runs in web browser.

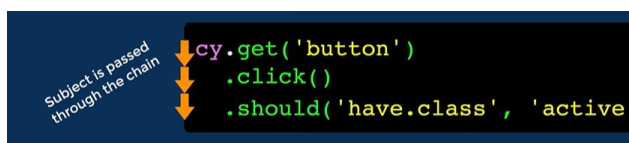
Our application will be pulled-in using an iframe –



Cypress command API – it is a chained API where subject is passed through the chain.

```
cy.get('button')
  .click()
  .should('have.class', 'active')
```

```
cy.request('/users/1')
  .its('body')
  .should('deep.eq', { name: 'Amir' })
```



Test commands are executed in a deterministic manner, resulting in flake-free testing. Cypress will automatically wait for this assertion `“.should”` (4 seconds by default). So we don't need to write code for wait and sleep until element is ready –

```
cy.get('button')
  .click()
  .should('have.class', 'active')
```

Cypress will automatically wait for this assertion (4 seconds by default)

By using the task command we can execute the javascript at the system level.

By using below approach, using store, we can directly dispatch login functionality through programmatically through our actual application code –

```
8
9 Cypress.Commands.add('login', (email, password) => {
10   return cy.window().then(win => {
11     return win.app.$store.dispatch('login', {
12       email: 'amir@cypress.io',
13       password: '1234'
14     })
15   })
16 })
```

In cypress we can also stub network response with fixtures by using `cy.server()` command.

To run the Cypress in the headless mode use “`cypress run`” command.

We can record results to Cypress dashboard by using “`npx cypress run --record`” command.

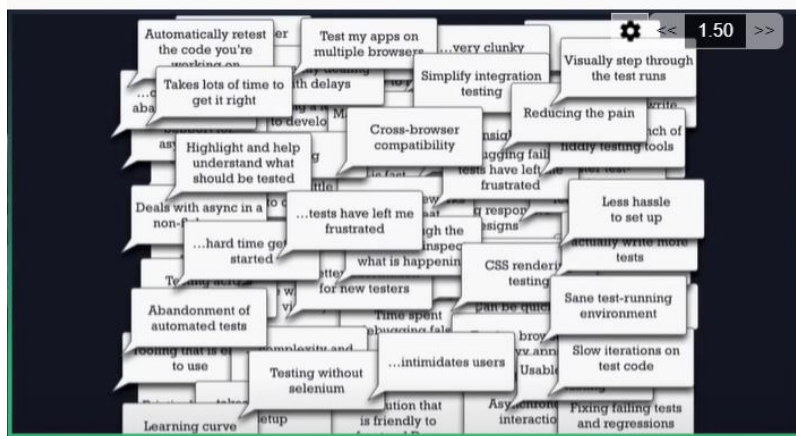
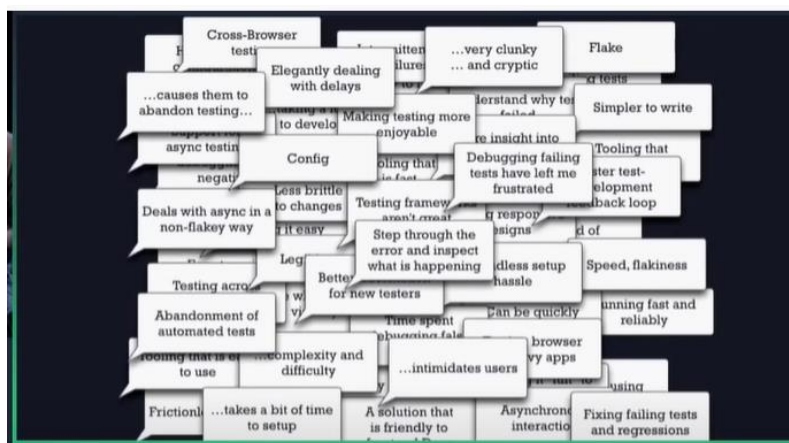
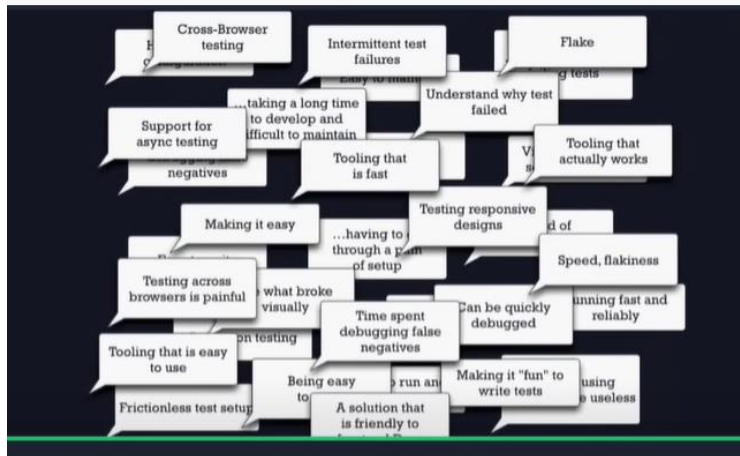
We can use `--parallel` flag to optimize CI usage with Parallelization, Cypress will automatically load balance the test files by using command “`npx cypress run --record --parallel`”.

Refer cypress example recipes - <https://github.com/cypress-io/cypress-example-recipes>

Testing The Way It Should Be (aka Intro Into Cypress) (Youtube)

People suggestion for creating a new tool -





[Other Resources]

Selenium and similar tools were designed to test applications that require a full-page refresh. Supporting SPAs with Ajax data fetching was an afterthought. This led to many issues with timing and flakey tests. Tests would sometimes fail due to slow API requests or network latency. Fixing these flakey tests typically required adding sleep statements and increasing timeouts. This made the test code more brittle. Not to mention extremely slow.

It's worth mentioning Google's Puppeteer has inner access to web browser events, allowing us to wait on things like Ajax calls. However, writing tests with Puppeteer requires more initial setup work and more effort to write each test than it should.

Cypress.io is a relatively new framework. It overcomes many shortcomings found in Selenium, Phantom.js, and others before them. It uses an event-based architecture that hooks into Google Chrome's lifecycle events. This enables it to wait for things like Ajax requests to complete without using a polling/timeout mechanism. This leads to reliable and fast tests. In short, it is truly the future of E2E testing and how it should have been in the first place.

You can run Cypress in two modes: full-mode and headless-mode. The former lets you see your app's UI and tests performed one step at a time. This mode is excellent for building up your test suite and debugging. The latter is great for a Continuous Integration (CI) environment. Another use case for headless-mode: you just want to make sure you haven't broken anything with new changes but don't care about the detailed steps.

Headless-mode is useful for running on a Continuous Integration (CI) server like CircleCI. Once you start writing tests more regularly as part of your development, you should invest time in getting a CI server configured so that every git commit runs the entire test suite.

Cypress is an end-to-end framework that was created by Brian Mann, who wanted to solve some pain points that a lot of developers face when writing integration tests: hard to write, unreliable and too slow. Similar to TestCafe, it was built on top of Node, with no dependencies on Selenium, and is a standalone testing framework that supports Javascript.

You can have a 100% code coverage with unit-tests that test all your components in isolation, but your application might still fail when components start to communicate with each other.

The real important tests are the ones that test functionalities that your users use every day. These are things like: "Can a user buy a product?" and "Will my order be shipped to the right address if I change the address later?" These kinds of things are impossible to test with unit tests, as they use all components of your application.





Cypress [End-to-end Vue Testing with Cypress]

End-to-end testing is a type of testing in which business processes are tested from start to finish under production-like circumstances.






It has a completely different architecture and run in the same run-loop as the application that is being tested.

In cypress, there is no multi-tabs support and does not support multiple browsers open at the same time. Also, there is not any native or mobile events support. Workaround is required for `cy.hover()` command.

Folder structure –

 Fixtures	Static files that are used in tests such as JSON files and images
 Integration	Tests
 Plugins	Modifications and extension of Cypress's internal behavior
 Support	Reusable pieces of code such as custom commands

Preferred method of choosing an element – 3rd, 4th and 5th are the preferred ones –

<code>cy.get('input')</code>	
<code>cy.get('.form-input')</code>	
<code>cy.get('#event-name')</code>	
<code>cy.get('input[name="event-name"]')</code>	
<code>cy.get('input[data-cy="event-name"]')</code>	

For the events, cypress provides additional parameters like passing the pixel location or name location –

<code>.click()</code>	<code>.select('value')</code>
<code>.click({ force: true })</code>	<code>.select('val', { log: false })</code>
<code>.click(5, 10)</code>	<code>.select(['val1', 'val2'])</code>
<code>.click('topLeft')</code>	<code>.select(['v1', 'v2'], { log: false })</code>
<code>.type('text')</code>	
<code>.type('text', { delay: 10 })</code>	
<code>.type('{backspace}')</code>	

We can also pass the value to be checked in the check group element –

```

17   cy.get('input[name="event-image"]').check('./
18   // submit the form
19   cy.get('button').contains('Create').click();
20   // assert that a new event has been created
21
22   });

```

Cypress commands do not return their subjects. They yield them. Cypress commands are asynchronous.

We should not do conditional testing as it make the test flaky. Conditional testing a test design consideration.

Cypress uses retry-ability for the commands automatically to avoid hard code waits. Below if the assertion is failed then it will go back to the command and wait for a certain period of time –

```
cy.get('.main-list li') // command
  .should('have.length', 3) // assertion
```

It only retries commands that query the DOM like get(), find() or contains(). Commands that are not retried are the ones that could potentially change the state of the application.

We should not set the timeout on a global level, but ideally we should do it in individual test cases and to make it disable we need to set it as 0 –

```
{
  "defaultCommandTimeout": 0
}

cy.get('button.primary')
  .click({ timeout: 0 });
```




Also, it will only retry the last command before the expression like in below it will do the several retry for the .find() but no retry for the get().

```
cy.get('.main-list li') // yields only one <li>
  .find('label') // retries multiple times on a single <li>
  .should('contain', 'My Item') // never succeeds
```

Making HTTP request from Cypress – we can use this in a scenarios like where we need to perform login without using UI or to get and create data.

```
cy.request('POST', 'http://localhost:8080/events', { name: 'New Event' })
  .then((response) => {
    expect(response.status).to.eq(201);
    expect(response.body).to.have.property('name', 'New Event');
  })
);
```

Types of commands – parent command start a new change and ignore previously yielded subject, child command cannot be directly used and need to apply on parent command or another child command. The dual commands can do both.

	Parent .visit(), .get()
	Child .click(), .type(), .should()
	Dual .contains()

Defining custom commands –

```
// parent command
Cypress.Commands.add('login', (email, password) => {
  cy.get('#email').type(email);
  cy.get('#password').type(password);
});

// child command
Cypress.Commands.add('login', { prevSubject: true }, (email, password) => {
  }); cy.get('#email').type(email);
  cy.get('#password').type(password);
});

// dual command
Cypress.Commands.add('login', { prevSubject: "optional" }, (email, password) => {
  }); cy.get('#email').type(email);
  cy.get('#password').type(password);
});
```

We can also overwrite an existing command –

```
Cypress.Commands.overwrite('type', (originalFn, element, text, options) => {
  if (options && options.sensitive) {
    options.log = false;
  }

  return originalFn(element, text, options);
});
```

We can create custom commands to avoid repeating the same code multiple time. Creating custom plugins by using task –

plugins/index.js	integration/homepage.spec.js
<pre>module.exports = (on, config) => { on('task', { log(message) { console.log(message); return null; } }) }</pre>	<pre>cy.task('log', 'This will be output.');</pre>

Hooks are the methods which run after or before a set/each of the test.


```

describe('Hooks', () => {
  before(() => {
    // runs once before all tests in the block
  });

  beforeEach(() => {
    // runs before each test in the block
  });

  afterEach(() => {
    // runs after each test in the block
  });

  after(() => {
    // runs once after all tests in the block
  });
})

```

It is suggested to do the clean-up before and not after tests.

Intercepting a http server side request, it can be useful in a scenario where we only want to test UI side of the application and make it independent from backend server –

```

describe('Event', () => {
  beforeEach(() => {
    cy.intercept('GET', 'http://localhost:8081/event', { fixture: 'list-events.json' }).as(
      'list-events'
    );
  });

  it('should list all events', () => {
    cy.visit('/');
    cy.wait('@list-events', { timeout: 10000 });
  })
})

```

Accessing Vue application instances –

```

events.spec.js U  main.js M x  list-events.json U  cypress.json M
client > src > main.js > ...
1  import { createApp } from 'vue'
2  import App from './App.vue'
3  import store from './store'
4  import router from './router'
5  import 'normalize.css'
6
7  const app = createApp(App).use(router).use(store).mount('#app');
8
9  if (window.Cypress) {
10   window.__app__ = app;
11 }

```

```

it('should create event', () => {
  cy.visit('/');
  cy.window().then(win => {
    cy.fixture('create-event.json').then(fixt => {
      win.__app__.$store.dispatch('createEvent', fixt);
    })
  });
  cy.wait('@create-event');
});

```

We can also test component in isolation. It need some other packages to be installed to get it working. Component testing with Cypress differs with traditional component testing because components are rendered directly in the browser with Cypress.

Cypress UI should only be used for developing and debugging tests. We should use command line interface if we just want to run tests or on CI server. We can use “cypress run”. To only run a single spec file then we need write “cypress run --spec “cypress/integration/my-spec.js”.

```
> _ npx cypress run
```

```
> _ npx cypress run --spec "cypress/integration/my-spec.js"
```

```
> _ npx cypress run --browser chrome
```

```
> _ npx cypress run --browser chrome --headless
```

For CI, we need to set the reporter options and need to disable the video to increase performance –

```

{
  "baseUrl": "http://localhost:8080/",
  "component": {
    "componentFolder": "src/",
    "testFiles": "**/*.spec.js"
  },
  "reporter": "junit",
  "reporterOptions": {
    "mochaFile": "tests/TEST-output-[hash].xml",
    "toConsole": true,
    "attachments": true
  },
  "video": false
}

```

Use start-server-and-test package to run both (application and test) –

```

6   "scripts": {
7     "serve": "vue-cli-service serve",
8     "build": "vue-cli-service build",
9     "lint": "vue-cli-service lint",
10    "cy:run-ct": "cypress run-ct",
11    "test": "cypress run",
12    "cy:run-e2e": "start-server-and-test serve http://localhost:8080 test"
13  }

```

End-to-end Angular Testing with Cypress

A good selector should be unambiguous and future proof. Like not using nth-child but id selector –

```

<div>
  <button>Button 1</button>
  <button>Button 1.1</button>
  <button id="read-more">Button 2</button>
  <button>Button 3</button>
</div>

```

button

~~div~~ → ~~button:nth-child(2)~~

button#read-more

The data-* attributes allow us to store extra information on standard, semantic HTML elements.

Specify base path in cypress.json like below –

```

1  {
2    "integrationFolder": "cypress/integration",
3    "supportFile": "cypress/support/index.ts",
4    "videosFolder": "cypress/videos",
5    "screenshotsFolder": "cypress/screenshots",
6    "pluginsFile": "cypress/plugins/index.ts",
7    "fixturesFolder": "cypress/fixtures",
8    "baseUrl": "http://localhost:4200"
9  }

```

To get the link element, we should use the href attribute selector –

```

it('should open the Breithorn adventure', () => {
  cy.get('a[href="/adventure/1"]').click();
});

```

By using this format with “then”, the Cypress won’t look the element in entire page, but only in the parent element –

```
cy.get('div[data-test-automation="user-comments"] blockquote:last-child').then($el => {
  cy.wrap($el).find('p').should('have.text', 'What a great adventure!');
  cy.wrap($el).find('footer').should('have.text', 'Josh');
})
```

Assertions types – length, class, value, visibility, exist, css

length	cy.get('li.selected').should('have.length', 1)
class	cy.get('input[name]').should('not.have.class', 'disabled')
value	cy.get('textarea#comment').should('have.value', 'Great!')
visibility	cy.get('button#add-comment').should('be.visible')

We can use multiple assertions into the chain –

```
cy.get('a.more-details')
  .should('have.class', 'btn')
  .and('have.attr', 'href')
  .and('include', 'adventures')

cy.get('span[name="loading"]')
  .should('exist')
  .and('not.exist')
```



```
cy.get('span[name="loading"]').should('exist')
cy.get('span[name="loading"]').should('not.exist')
```

Conditional testing can only be used when the state of the application has stabilized means there is no pending network requests, no timeouts and intervals and no async/await code.

There are two design pattern while writing the E2E to make it more maintainable and free from future breakage – Page Object Model and App Actions.

App Actions is an approach where tests directly access the internal implementation of the application under test. It enable changing application’s state without interacting with the application through the UI.

For App actions, we can add a angular component on the window object like below which can be accessed in cypress tests –

```
export class FilterComponent {
  ...
  constructor(private filterService: FilterService) {
    this.filterService.filterValueChange.subscribe(value => this.filterValue = value);

    if (window.Cypress) {
      ➡ window.FilterComponent = this;
    }
  }

  onChange(value: string): void {
    this.filterService.filterBy(value);
  }
}
```

```
it('should display filter criteria', () => {
  cy.visit('/');
  cy.window().then((window: Window) => {
    ➡ cy.wrap(window).its('FilterComponent').invoke('onChange', 'Tara');
  });
  ...
});
```

Benefits	Drawbacks
Tests become much faster	Fail to catch certain bugs
Tests become more focused	Easy to create unrealistic test scenarios
Tests are cleaner and easier to maintain	Knowledge of the front-end framework of the application is required (Angular)
Test stability is increased	

Page Object model is a wrapper over a web page and a design pattern where web pages are represented as classes. Encapsulates the mechanics required to interact with the user interface. Second one, is the example for page object –

```
it('should open the Breithorn adventure', () => {
  cy.get('a[href="/adventure/1"]').click();
  cy.get('#title').should('have.text', 'Breithorn, Pennine Alps');
});
```

```
it('should open the Breithorn adventure', () => {
  homePage.clickMoreDetailsBtn(1)
    .getAdventureTitle().should('have.text', 'Breithorn, Pennine Alps');
});
```

```
import { AdventureDetailsPage } from "../adventure-details.page";

export class HomePage {

→ visit(): HomePage {
    cy.visit('/');
    return this;
}

→ clickMoreDetailsBtn(adventureId: Number): AdventureDetailsPage {
    cy.get(`a[href="/adventure/${adventureId}"]`).click();
    return new AdventureDetailsPage;
}
}
```

Benefits

- Better maintainability when UI changes occur
- Re-using the same code across multiple tests
- Tests are easier to read and follow

Drawbacks

- Easy to violate a single responsibility principle
- Additional time and effort are required for maintenance

Adding application reference to the window object –

```
19
20 declare global {
21   interface Window { appRef: ApplicationRef, Cypress: any }
22 }
23
```

```
47 export class AppModule implements DoBootstrap {
48   ngDoBootstrap(appRef: ApplicationRef): void {
49     appRef.bootstrap(AppComponent);
50
51     if(window.Cypress) {
52       window.appRef = appRef;
53     }
54   }
55 }
56
```



```

3
4 declare global {
5   interface Window { FilterComponent: FilterComponent }
6 }
7
8 @Component({
9   selector: 'app-filter',
10  templateUrl: './filter.component.html',
11  styleUrls: ['./filter.component.css']
12 })
13 export class FilterComponent {
14
15   filterValue: string = '';
16   @Input() placeholder: string = '';
17
18   constructor(private filterService: FilterService) {
19     this.filterService.filterValueChange.subscribe(value => this.filterValue = value);
20
21     if(window.Cypress) {
22       window.FilterComponent = this;
23     }
24   }
25

```

```

app.module.ts M  filter.component.ts M  filter.spec.ts U x
cypress > integration > filter.spec.ts > describe('Filter') callback > it('should display filter criteria') callback
1 describe('Filter', () => {
2   it('should display filter criteria', () => {
3     cy.visit('/');
4     cy.window().then((window: Window) => {
5       cy.wrap(window).its('FilterComponent').invoke('onChange', 'Tara');
6       cy.wait(1000);
7       cy.wrap(window).its('appRef').invoke('tick');
8     });
9
10    cy.get('p[data-test-automation="filtered-by"]')
11      .should('have.text', 'Filtered by: Tara');
12  });
13 });

```

By creating page object classes, it will increase the maintainability, code reusability and readability of our code –

```
adventure.spec.ts • home.page.ts U • adventure-details.page.ts U •
cypress > pages > adventure-details.page.ts > AdventureDetailsPage > addComment
1  export class AdventureDetailsPage {
2
3      getAdventureTitle(): Cypress.Chainable {
4          return cy.get('#title');
5      }
6
7      resetComments(): AdventureDetailsPage {
8          cy.contains('Reset Comments').click();
9          return this;
10     }
11
12     addComment(name: string, comment: string): AdventureDetailsPage {
13         cy.contains('Add Comment').click();
14
15         if (name) {
16             cy.get('#name').type(name);
17         }
18
19         if (comment) {
20             cy.get('#comment-text').type(comment);
21         }
22         cy.get('#add-comment-button').click();
23
24         return this;
25     }
26 }
```

App Actions and page objects are not mutually exclusive. We can combined them both.

Commands are functions that perform specific tasks. Plugins enable you to tap into, modify, or extend Cypress's internal behavior.

While writing the plugins, they run in Cypress background tasks in Node, since they executed in Node not in the browser, we can't use Cypress syntax while writing plugins code.

Writing Custom command – they increases the code reusability and readability –

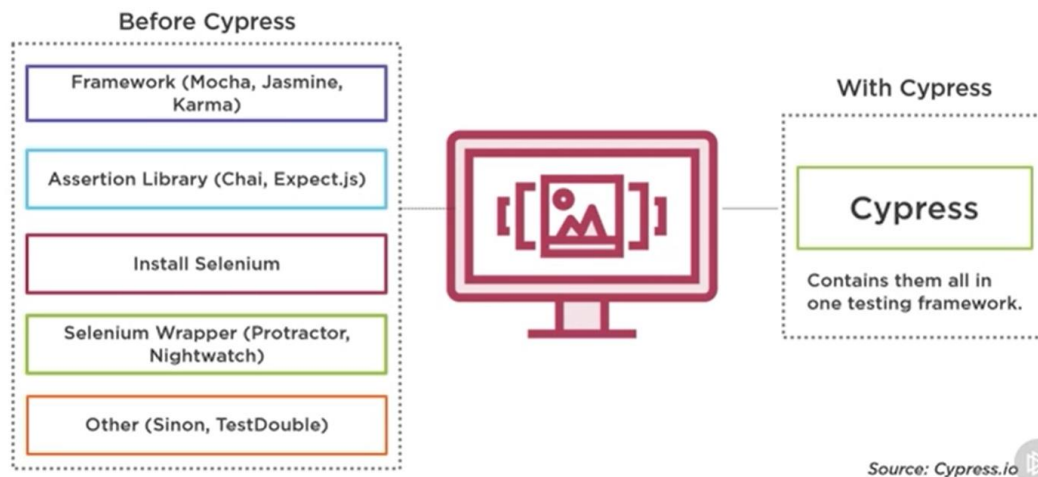
```
44
45  declare namespace Cypress {
46      interface Chainable {
47          filterAdventures(text: string): void;
48      }
49  }
50
51  Cypress.Commands.add('filterAdventures', (text: string): void => {
52      cy.window().then((window: Window) => {
53          cy.wrap(window).its('FilterComponent').invoke('onChange', text);
54          cy.wait(1000);
55          cy.wrap(window).its('appRef').invoke('tick');
56      });
57  });
```

Cypress run will run all of the tests in headless mode of electron browser by default.

We should identify steps to automate through the UI and then use the App Actions for supporting steps. Like for login workflow use it once by browser interaction and then for other dependent tests use via App Actions. Also, we should use precise and future proof selectors.

Cypress: End-to-end JavaScript Testing

Before and After Cypress –



To select an element, we should use the 'class' as style can be changed anytime, nor we should use tags as there can be many tags or new element can be added. We should use name attribute or use contains command with text. Ideally we should use data-cy attributes.

```
cy.get('form-control- form-control-lg').click()
cy.get('input').click()
cy.get('[name=username]').click()
cy.contains('Username').click()
cy.get('[data-cy=username]').click()
```

Assertion types – be, equal, have.length, not.exist

Cypress only retries commands that query the DOM like get(), find(), contains(), etc.

To increase the readability we can use the aliases in before each like below –

```
// adding a hook that runs before the test
beforeEach(() => {
  cy.task('cleanDatabase')
  cy.registerUserIfNeeded()
  cy.login()

  // define aliases - Remember not to create aliases when you are deep in the chain
  cy.get('[data-cy=new-post]').click().as('ClickOnNewPost')
})

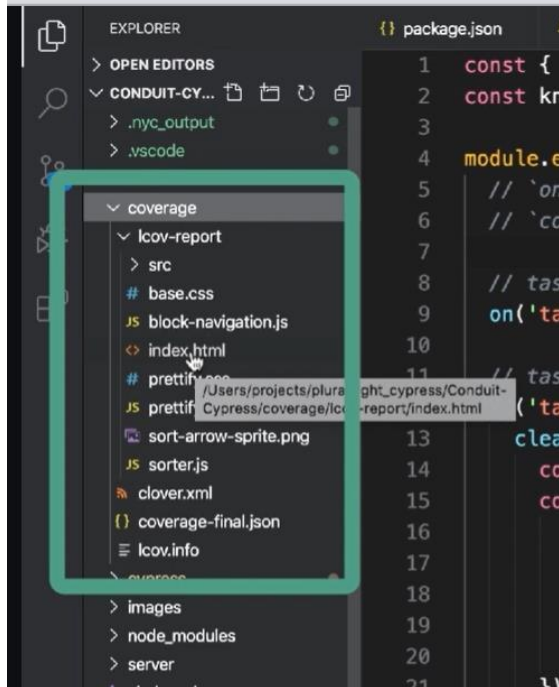
it('write a new post', () => {
  // Fill details to write a new post
  cy.get('@ClickOnNewPost')
})
```

We can use `cy.screenshot()` to take the screenshot of the application while testing. By default, Cypress takes screenshots and videos for the tests which has failed when we are running it with command line.

```
{ } cypress.json  JS index.js x
1  /// <reference types="cypress" />
2  import '@cypress/code-coverage/support'
3  import '@bahmutov/cy-api/support'
4
5  const apiUrl = Cypress.env('apiUrl')
6
7  Cypress.Screenshot.defaults({
8    screenshotOnRunFailure: false
9  })
10
11 // a custom Cypress command to login using XHR call
12 // and then set the received token in the local storage
13 // can log in with default user or with a given one
14 Cypress.Commands.add('login', (user = Cypress.env('user')) => {
15   cy.getLoginToken(user).then(token => {
16     // ...
17   })
18 })
```

To show the code-coverage feature, we need to install the `@cypress/code-coverage` library and Istanbul. It will generate the `lcov` folder for the code coverage.

<pre>cypress/support/index.js import '@cypress/code-coverage/support'</pre>	<pre>cypress/plugins/index.js module.exports = (on, config) => { // tasks for code coverage on('task', require('@cypress/code-coverage/task')) }</pre>
--	---



By using plugins, we can modify or extend the internal behavior of Cypress. We can write our own custom code that executes during different Cypress stages. By this we can also tap into node process running outside the browser. We can also alter the configuration and environment variables or customize how test code is transpiled and sent to browser or to manipulate the database.

```
// export a function
module.exports = (on, config) => {

  on('<event>', (arg1, arg2) => {
    // plugin stuff here
  })
}
```

```
3
4 module.exports = (on, config) => {
5   // 'on' is used to hook into various events Cypress emits
6   // 'config' is the resolved Cypress config
7
8   // tasks for code coverage
9   on('task', require('@cypress/code-coverage/task'))
10
11   // tasks for resetting database during tests
12   on('task', {
13     cleanDatabase () {
14       const filename = join(__dirname, '..', '..', 'server', '.tmp.db')
15       const knex = knexFactory({
16         client: 'sqlite3',
17         connection: {

```

Automated Testing All the Things with Cypress

Testing provides a great documentation and increase confidence, safe refactoring.

We can use Cypress for unit testing, integration testing and e2e testing.

Testing Complex Interfaces and Visualizations

Testing tooltips and cursor is the hardest part to test for interaction.

Data visualization interactions –

Hover

- Tooltips and cursor
- Update adjacent content: highlight rows, show data in a table, ...

Click

- Simple click: toggle data, open popup, act as a link, start selection
- Click+drag: zoom! pan! filter! rearrange! 🤖

Keyboard: not used often

Mouseout, blur: make it all go away. Leave me alone. 🚀

Testing line chart – it is visible, it loads (no error or timeout), it renders data.

```
it('renders charts with data (latency, errors, rate)', () => {
  const options = { timeout: CHART_LOADING_TIMEOUT };
  cy.getByTestId('latency-chart').assertChartHasLineData({ options });
  cy.getByTestId('error-chart').assertChartHasLineData({ options });
  cy.getByTestId('rate-chart').assertChartHasLineData({ options });
});
```

```
Cypress.Commands.add(
  'assertChartHasLineData',
  { prevSubject: 'element' },
  (subject, { ...options }) => {
    return cy
      .get('.recharts-line path', { withinSubject: subject, ...options })
      .first()
      .invoke('attr', 'd')
      .should('have.length.of.at.least', 2 * 1000);
  }
);
```

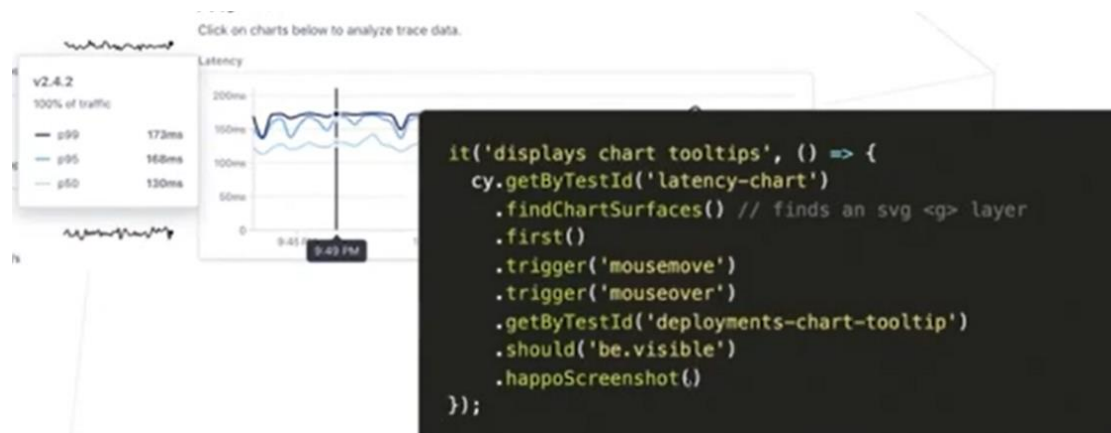
For screenshot testing, cypress can take Haplo screenshots and trigger the comparison –

- Use <https://github.com/haplo/haplo-cypress>
- Use fixtures for all data
- Make everything completely repeatable (“spurious diffs”)
- Wait for animations to end (usually `cy.wait()`, sadly)
- `cy.get('mything').haploScreenshot()`
- `cy.get('mything').haploScreenshot({ variant: 'after cart cleared' })`

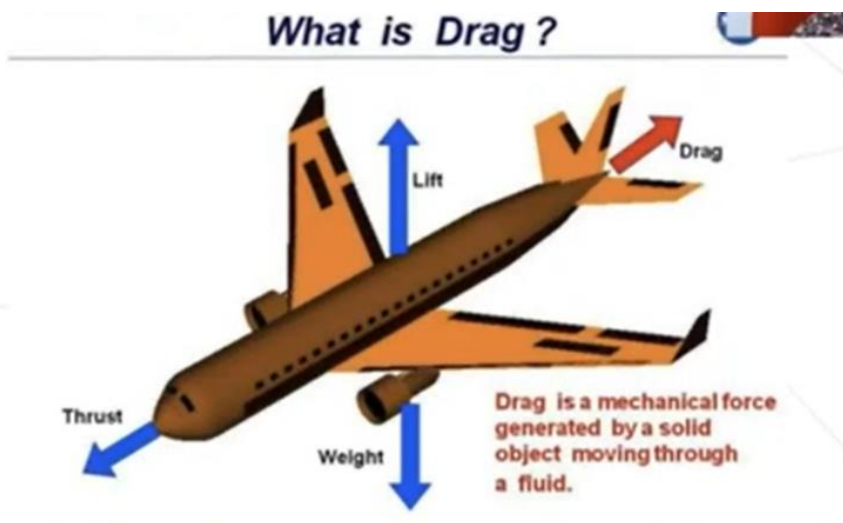
- **Data loads:** Cypress tests
 - Using real app backends or fixtures
- **Data is formatted properly:** Haplo + Storybook
- **Data is accurate:**
 - Haplo + Storybook -and/or-
 - Haplo + Cypress

Testing user interactions – hovering on a chart shows a tooltip, click or click + drag on a chart.

Javascript mouse events are easy but CSS mouse events are hard. Cypress can't easily trigger pseudo-classes like :hover. There are docs and options for this, but it is difficult.



Drag is a mechanical force generated by a solid object moving through a fluid.



```

it('allows click-and-drag to make a time selection', () => {
  cy.getByTestId('explorer-histogram-chart', { timeout })
    .then(($el) => {
      const rect = $el[0].getBoundingClientRect(); // an actual DOM element
      const dragY = rect.y + rect.height / 2;
      // select middle 50% of histogram
      const dragStartX = rect.x + rect.width * 0.25;
      const dragEndX = rect.x + rect.width * 0.75;
      cy.get('.LatencyHistogram_Bar', { timeout })
        .trigger('mousedown', { pageX: dragStartX, pageY: dragY }) // ← this
        .trigger('mousemove', { pageX: dragEndX, pageY: dragY }) // ← this
        .trigger('mouseup');
    })
  cy.url()
    .should('include', 'min_latency=')
    .should('include', 'max_latency=')
});

```

Cypress Other Resources

For API testing, we should use SuperTest.

With selenium we have slow, brittle tests and lots of waits as it is just moving the mouse and clicking buttons. it doesn't have much visibility into the DOM.

We should not select element by using text. We should use consistent selectors like id, class. Even better, use test-specific attributes like `<div data-cy="myElement"/>`

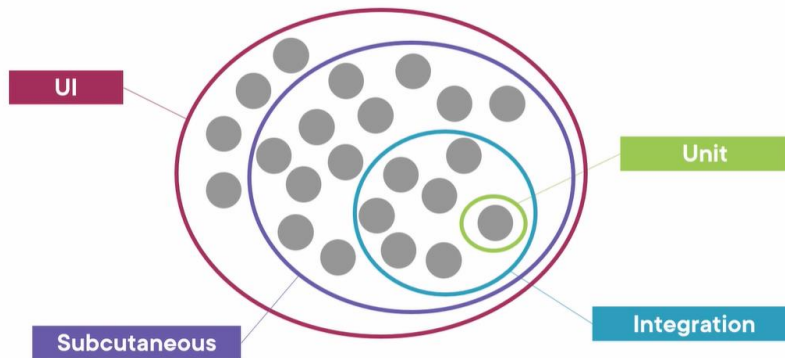
Automated Testing: The Big Picture

Benefits of automated tests vs. manual human –

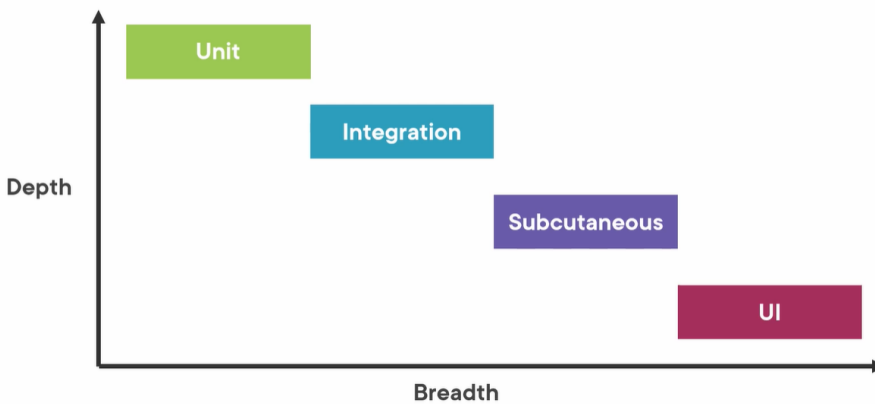
Automated Tests	Manual Human Tests
Free to run as often as required	Cost every test run (staff)
Run at any time	Limited to staffed hours
Quicker	Slower
Generally less error prone	Potentially more error prone
Automated test code in source control	Manual test scripts in external system
Creation and maintenance costs	Creation and maintenance costs

Subcutaneous tests – a level above than the unit and integration tests and just below the surface of the UI. In this we can test all the non-UI components working together.

Functional UI Tests



Test breadth versus depth –



We should write the smallest number of tests possible to reach the required level of quality or confidence in the system being developed.

Characteristics of good automated tests – isolated (no side effects on other tests), independent (can be run in any order), repeatable (always pass or fail), maintainable, valuable.

When a bug is found, a failing automated test can be written to reproduce it. When the bug is fixed the test will pass. In some future change, if the bug reoccurs it will be caught by the automated test.