Optimize Your Testing Performance by Using Cypress.io

Ryan Song

ryan.song@iterable.com

Abstract

Automation testing is one of the essential components in the software development life cycle, yet it can be very time consuming and resource demanding to maintain. Quality Engineers constantly face the need to trade off or prioritize among test speed, test coverage, and test stability, as achieving all three simultaneously is challenging. This article will discuss the approach using Cypress.io to improve test speed, test coverage and test stability at the same time. Additionally, it includes some techniques and processes that leverage Cypress to optimize end-of-end (E2E) tests.

Biography

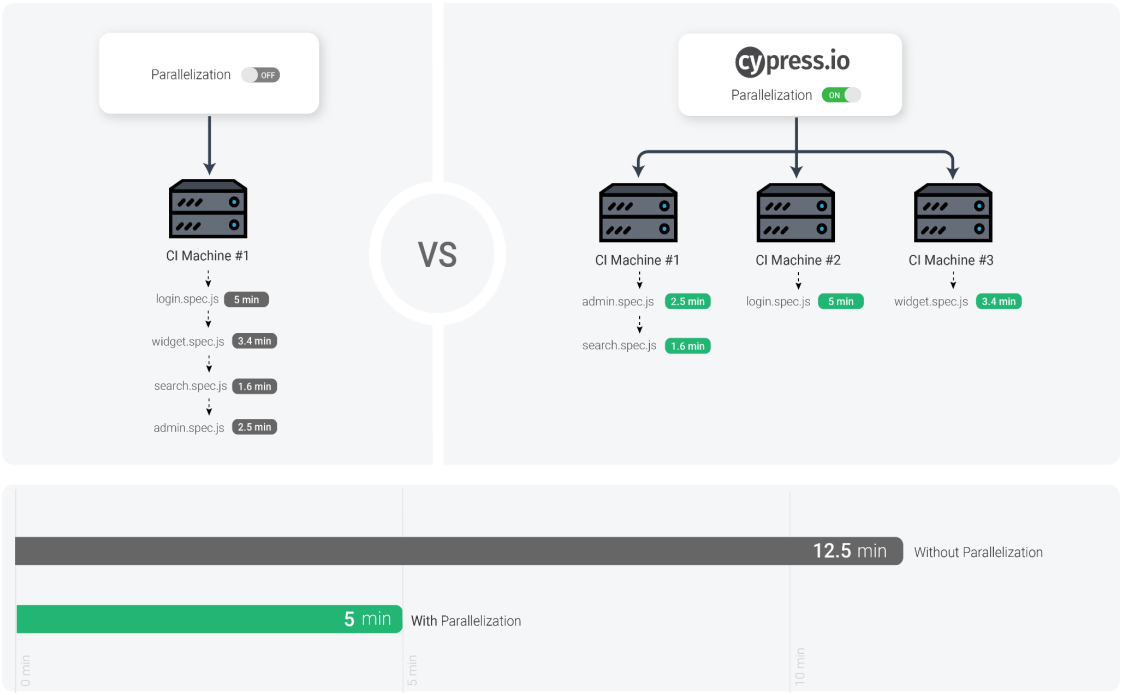
Ryan is a Staff Test Engineer at Iterable who has been working in the quality industry for 8 years. He has experience working on projects of varying sizes, such as startups, federal/defense and Fortune 50-level companies. He is passionate about Automation Testing and CI/CD process and specializes in system optimization and operation research. Ryan is located in Los Angeles and graduated from Texas A&M University with a degree in Industrial and System engineering.

Introduction

In automation testing, Quality Engineers often find it challenging to balance test speed, test coverage, and test stability with competing priorities. For example, an increase in test speed might lead to a decrease in test stability, and expanding test coverage could result in a corresponding decrease in test speed. Balancing these three aspects requires a significant amount of work experience and technical knowledge. This article will discuss the value of upholding these three pillars of automation testing and share techniques to achieve this.

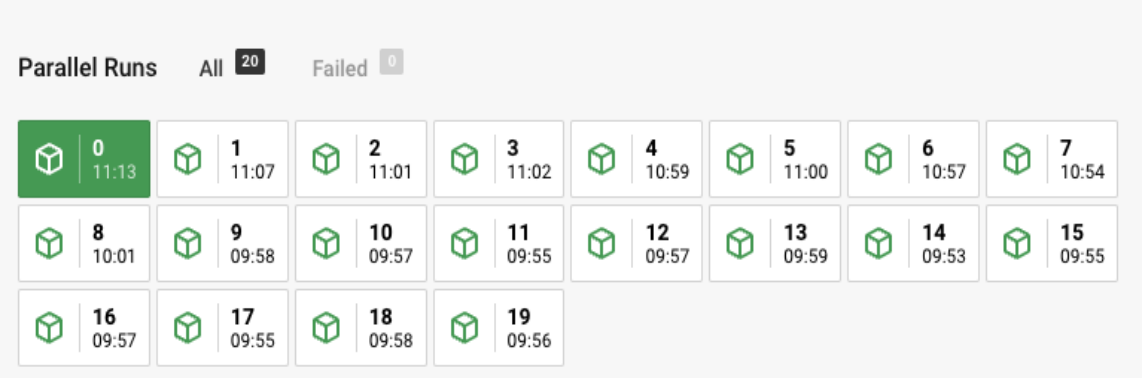
1. The importance of test speed in test automation

The speed of the test automation is one of the most important factors in agile principles, especially when companies implement continuous integration and continuous deployment/delivery (CI/CD). Running automation tests quickly enables developers to iterate faster, therefore they can promptly review test results and make necessary adjustments in a timely manner. One common issue many developers face is when test suites grow too large, resulting in significantly extended completion times for all the tests. Many companies are running all their tests during night time and only able to run one batch of tests per night. This is not ideal for teams that want to be more agile and efficient. Cypress solves this test speed problem by using the built-in feature to automatically split up test suites. This technique will empower developers to run tests in parallel and significantly reduce test run time.



Picture 1: Cypress official site for tests split

As you can see in the picture above, If all the tests are executed on a single machine, it will take about 12.5 mins to get the test results. However, when distributing tests across in three different virtual machines, the total testing times can be reduced to five mins.



Picture 2: CircleCI project page

Currently at Iterable, we run almost 300 Cypress E2E tests in 20 parallel machines within CircleCI (continue integration tool), and each round test run can be finished in less than 15 mins. Many of our developers run many pull requests on a daily basis, frequently validating their alterations by executing E2E tests multiple times throughout the day. This strategy offers much faster feedback to the developers, a crucial aspect of our CI/CD pipeline, given the frequency of multiple deployments we conduct each day.

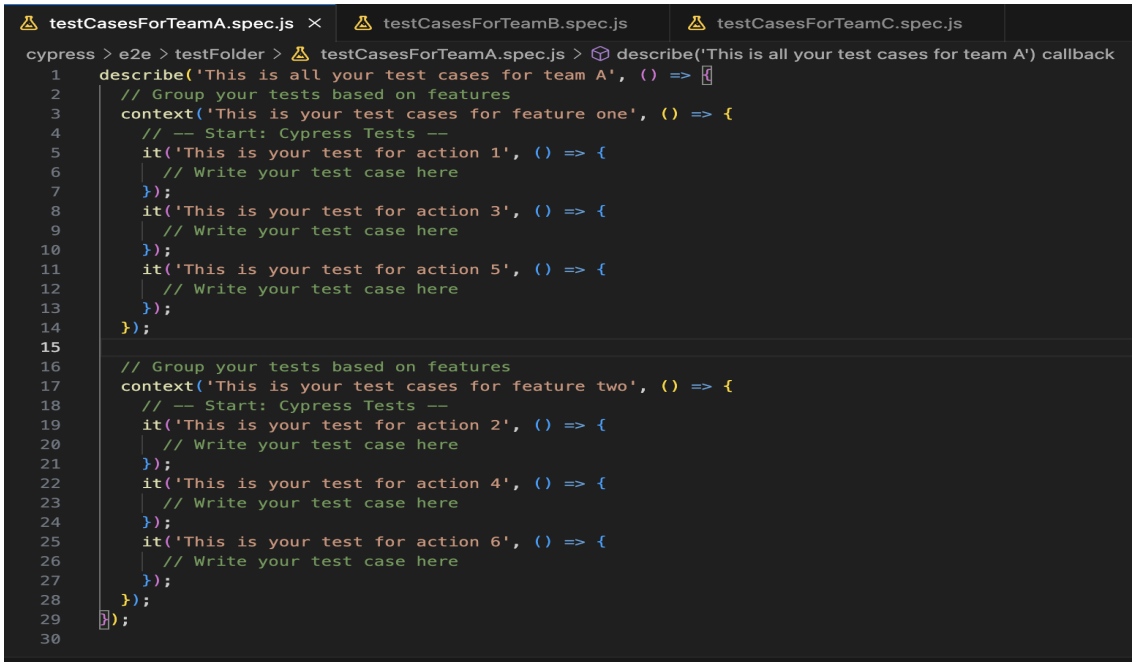
2. The importance of test coverage in test automation

Another important factor for software testing is test coverage. Without appropriate test coverage, many bugs will leak to production. Many companies are using percentage of code coverage as a baseline to track developer's code quality. For example, at Iterable, we use CodeClimate to check the current code test coverage. The problem with this approach is that it is very difficult to have 100% test coverage and challenging to update and maintain all the tests. There are two solutions to solve this issue. The first solution is to build E2E tests by using business scenarios to test the application. Many companies are using tools such as Segment to track customer usage and their QE team can utilize the data from those tools to design the E2E test scenarios. The table below shows the example for one team's feature usage.
The the Feature % Click is calculated based on the (number of clicks/total number of clicks)* 100. The Cumulative % is calculated based on the (current Feature % Click + previous Feature % Click).

Number of Visitors	Name	Number of Clicks	Feature % Clicks	Cumulative %
3302	Action 1	95682	13.27%	13.27%
2768	Action 2	86663	12.02%	25.28%
3415	Action 3	80185	11.12%	36.40%
4753	Action 4	73524	10.19%	46.60%
1503	Action 5	46697	6.47%	53.07%
2782	Action 6	37813	5.24%	58.31%
2515	Action 7	29570	4.10%	62.41%
2606	Action 8	28099	3.90%	66.31%
2532	Action 9	25776	3.57%	69.88%
2199	Action 10	23522	3.26%	73.15%
1209	Action 11	22853	3.17%	76.31%
2199	Action 12	21720	3.01%	79.33%
1853	Action 13	11244	1.56%	80.88%
1839	Action 14	11116	1.54%	82.43%
1257	Action 15	9902	1.37%	83.80%
1116	Action 16	8741	1.21%	85.01%
1900	Action 17	8543	1.18%	86.20%
1110	Action 18	8251	1.14%	87.34%
1490	Action 19	8126	1.13%	88.47%
757	Action 20	6896	0.96%	89.42%
1415	Action 21	6579	0.91%	90.33%
1570	Action 22	5622	0.78%	91.11%
875	Action 23	5430	0.75%	91.87%
632	Action 24	4873	0.68%	92.54%
804	Action 25	4113	0.57%	93.11%
692	Action 26	3779	0.52%	93.64%
1114	Action 27	3657	0.51%	94.14%
825	Action 28	3215	0.45%	94.59%
899	Action 29	3104	0.43%	95.02%
1307	Action 30	2965	0.41%	95.43%
929	Action 31	2873	0.40%	95.83%
367	Action 32	2795	0.39%	96.22%
538	Action 33	2685	0.37%	96.59%
641	Action 34	2476	0.34%	96.93%
463	Action 35	2281	0.32%	97.25%
890	Action 36	2216	0.31%	97.56%
752	Action 37	2091	0.29%	97.85%
762	Action 38	2064	0.29%	98.13%

Picture 3: Mocked test data

As you can see in the chart above, if all actions in this chart are included in the Cypress E2E, then we can assume that **98.13%** of the business scenarios are covered. This strategy will optimize the Cypress test suite and reduce the redundant tests while maintaining high levels of test converge.



Picture 4: Example of Cypress test suite setup

The first solution is to group the test suites based on the business usage. Each team can have its own test folder and test files. The test scenarios can be broken down to feature leave and all the actions related to that feature can be grouped together based on business logic. This will provide a clear structure in the Cypress tests and easy to manage test coverages when a new feature is added to the system or existing feature is updated.

3. The importance of test stability in test automation

The third important factor for automation testing is the test stability. Many developers are facing issues of flaky tests failing on the pull request. Those flaky tests significantly impact the developer productivity and cost company time and money. There are many reasons that cause flaky tests, such as web page rendering too fast, race conditions when spins up servers, test setup depended on timezones, and etc. Cypress has many built in features that address some of those issues. There are few features or techniques that helped my team to make the test more stable. The first one is the *Test Retries* feature, which is able to retry the failed tests a few times before failing the test. The extra retry will improve the passing rates and save developer time and resources. This is a built-in feature and can be configured in the cypress.config.js file as shown below.

```
{
  "retries": {
    // Configure retry attempts for `cypress run`
    // Default is 0
    "runMode": 2,
    // Configure retry attempts for `cypress open`
    // Default is 0
    "openMode": 0
  }
}
```

Picture 6: Cypress retries feature

The second feature is the wait mechanism, Cypress has a built-in feature to wait for network calls to complete before continuing the next step. When running the tests, developers can interpret the network calls and let Cypress wait for a specific request to respond before executing the next test step. This feature significantly reduces the race condition that happens within test runs.

End-to-End Test | Component Test

```
// Wait for the alias 'getAccount' to respond
// without changing or stubbing its response
cy.intercept('/accounts/*').as('getAccount')
cy.visit('/accounts/123')
cy.wait('@getAccount').then((interception) => {
  // we can now access the low level interception
  // that contains the request body,
  // response body, status, etc
})
```

Picture 7: Cypress intercept requests and wait

The third technique is to utilize Cypress API commands to do data preparation test steps. Setting up test data in a browser and running through all the preparation steps will increase the risk of flaky tests. Moreover, test run time will increase significantly when there is a substantial amount of test data that needs to be set up. For example, the picture below shows the significant time difference between running a test through a web browser compared to using API calls to configure the test data.Therefore, directly making API calls to set up test data is a better approach. API calls are usually more stable and fast to complete.

Test steps	API	Browser
Login	3s	10s
Turn on the feature flags	5s	20s

Picture 8: Test preparation steps between API call vs browser

```
cy.request({
  method: 'POST',
  url: '/login_with_form', // baseUrl is prepend to URL
  form: true, // indicates the body should be form urlencoded
  body: {
    username: 'jane.lane',
    password: 'password123',
  },
})

// to prove we have a session
cy.getCookie('cypress-session-cookie').should('exist')
```

Picture 9: Login via API call

```
cy.request({
  method: 'PUT',
  url: `/featureFlags/${featureFlag}`,
  form: true,
  headers: {
    'X-XSRF-TOKEN': XXSRFTOKEN,
  },
  body: {
    ...param,
  },
});
```

Picture 9: Turning on/off feature flags via API call

Conclusion

This paper highlighted the importance of test speed, test coverage and test stability. There is no question that using Cypress.io with the correct strategy will help your organization to improve all three areas simultaneously. The results shown in this paper should encourage you and your team to explore and adopt Cypress.io and improve your current test framework and strategies. Utility leaving to a more robust efficient testing program. As we continue to enhance our understanding of proactive automation testing, we can drive innovation for new techniques and cultivate a culture of continuous improvement.

References

Cypress official documentation:
https://docs.cypress.io/guides/cloud/smart-orchestration/parallelization
https://docs.cypress.io/api/commands/wait
https://docs.cypress.io/api/commands/request