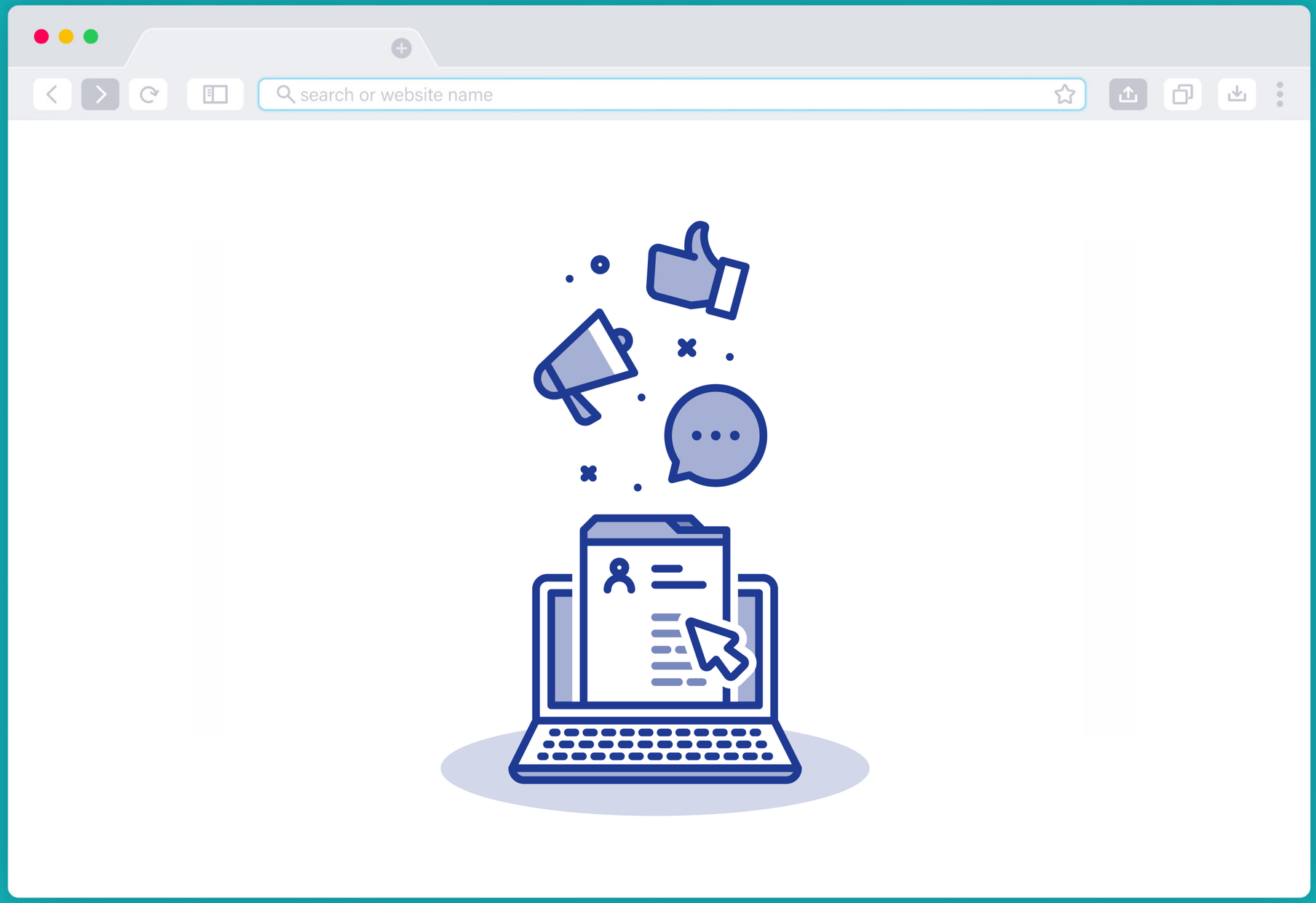
Exploring Features and Core Concepts

Marko Vajs
Software Development Engineer in Test

Module Overview

- Explore the asynchronous nature of Cypress
- Learn how to approach conditional testing
- Explore Cypress's built-in retry-ability
- Learn how to make HTTP requests
- Explain in detail commands and plugins
- Explain lifecycle hooks

Mixing Synchronous and Asynchronous Code


search or website name

```
cy.get('button.primary').click();
```

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

Cypress commands are asynchronous.

```
it('test', () => {
     let username = undefined;

     cy.visit('http://localhost:8080')
     cy.get('#username')
       .then(($el) => {
           username = $el.text()
       });

     if (username) {
          cy.contains(username).click();
     } else {
          cy.contains('MyProfile').click();
     }
});
```

```
it('test', () => {
     let username = undefined;

     cy.visit('http://localhost:8080');
     cy.get('#username')
       .then(($el) => {
           username = $el.text();

           if (username) {
             cy.contains(username).click();
           } else {
             cy.contains('My Profile').click();
           }
       });
});
```

Doing Conditional Testing

Conditional Testing

```
IF (<Condition>)
    THEN <Statements>;
    ELSE <Statements>;
ENDIF;
```

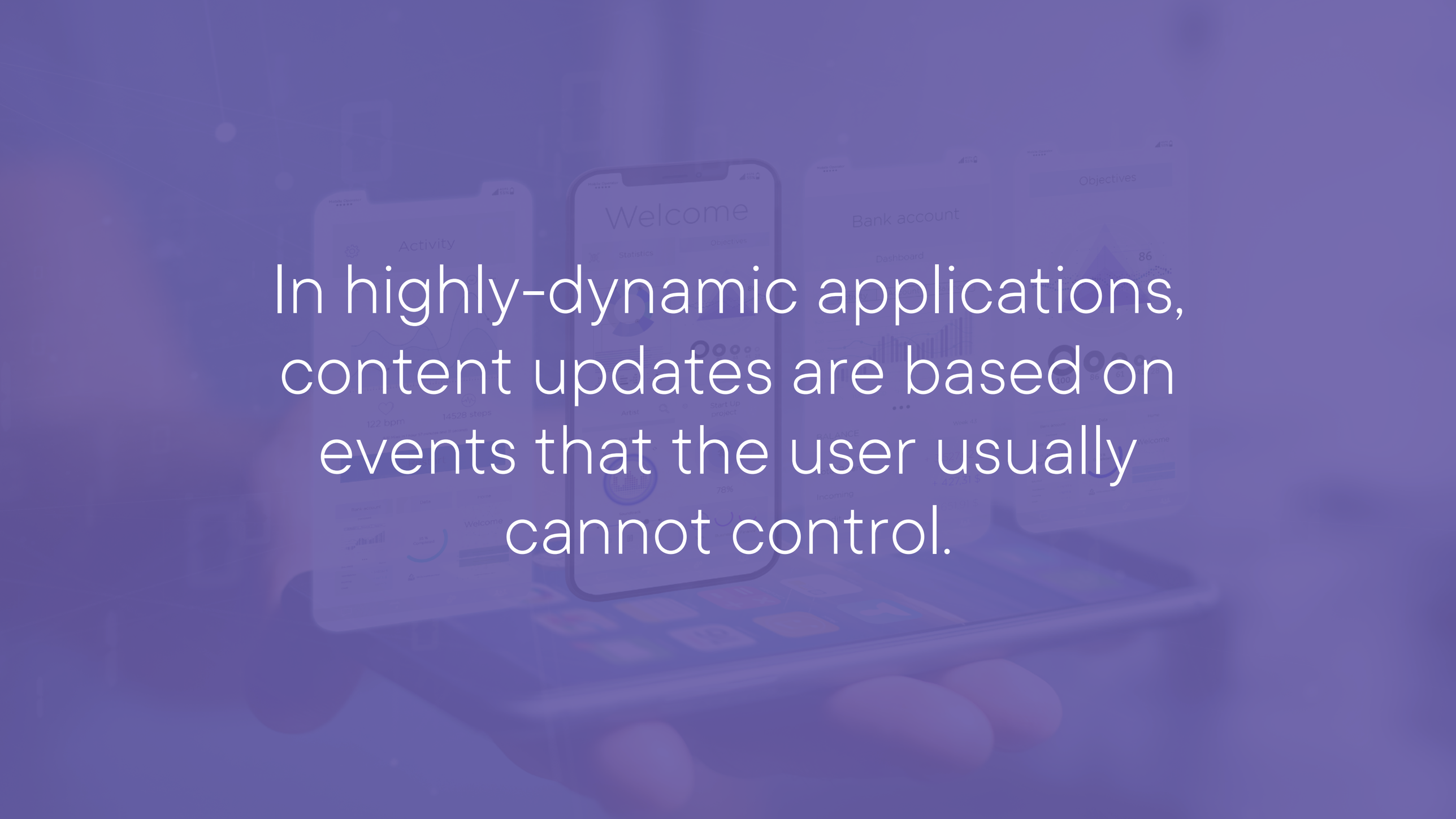
Is It Possible to Make Conditional Tests Consistently?

Conditional testing can only be used when the state of the application has stabilized.

No pending network requests

No timeouts and intervals

No async/await code


In highly-dynamic applications, content updates are based on events that the user usually cannot control.

Strategies to Overcome Non-deterministic Behavior

Remove the need to do conditional testing

Make application behave deterministically

Check other sources of truth (like server or database)

Embed data into other places you could read off

Add data to the DOM that you can read off to know how to proceed


Conditional testing is a test design consideration.

Exploring Retry-ability

Two Types of Methods in Cypress

Commands

Assertions

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

The retry-ability allows the tests to complete each command without hard-coding waits.

Cypress only retries commands that query the DOM.

`.get()` `.find()` `.contains()`

Commands that are not retried are the ones that could potentially change the state of the application.

Changing the Timeout

The default timeout can be changed on a command level or globally.

cypress.json

```
{
    "defaultCommandTimeout": 0
}
```

homepage.spec.js

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

homepage.spec.js

```
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 Requests

With Cypress, the CORS check
is bypassed.

```
cy.request('http://localhost:8080');


cy.visit('http://localhost:8080/list-events');
cy.request('events/1.json'); // url is http://localhost:8080/events/1.json


cy.request('DELETE', 'http://localhost:8080/events/1');


cy.request('POST', 'http://localhost:8888/events', { name: 'New Event' });

cy.request({
    method: 'POST',
    url: 'http://localhost:8888/events',
    body: { name: 'New Event' },
    headers: { 'Content-Type': 'application/json' }
});
```

```
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');
   }
);
```

When to Use Requests?

To log in

To get or create data

Commands and Plugins

Commands and Plugins

Commands

Commands are instructions in the form of methods to perform specific tasks. Cypress has many built-in commands, but there is also an option to define your own custom commands.

Plugins

Plugins are pieces of code that enable you to tap into, modify, or extend Cypress's internal behavior, and they can be executed during different stages of the Cypress life cycle.

Commands

📁 support

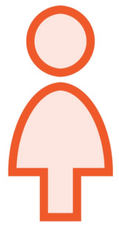
commands.js

```
// ***********************************************
// This example commands.js shows you how to
// create various custom commands and overwrite
// existing commands.
// ***********************************************
```

index.js

```
import './commands'
```

Types of Commands



Parent

```
.visit(), .get()
```

Child

```
.click(), .type(), .should()
```

Dual

```
.contains()
```

Defining a Custom Command

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

// child command
CypressaCommands.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);
});
```

Commands

Defining and using a custom command

support/commands.js

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

integration/homepage.spec.js

```
cy.login('mail@example.com’,
 'Password.1');
```

Overwriting a Command

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

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

Plugins

📁 plugins

index.js

```
/// <reference types="cypress" />

/**
* @type {Cypress.PluginConfig}
*/
module.exports = (on, config) => {
// `on` is used to hook into various events
Cypress emits
// `config` is the resolved Cypress config
}
```

Plugins

Defining and using your own plugin

plugins/index.js

```
module.exports = (on, config) => {
    on('task', {
        log(message) {
            console.log(message);

            return null;
        },
    });
}
```

integration/homepage.spec.js

```
cy.task('log', 'This will be output.');
```

Understanding Hooks

```
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
    });
})
```

```
before(() => {

});


describe('Hooks', () => {
      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.

Summary

Module Summary

- Put asynchronous code inside the `.then()` method
- Conditional testing requires a stable source of truth
- Commands such as `.get()`, `.contains()`, and `.find()` are retried multiple times while others, such as `.click()` are not
- HTTP requests bypass CORS because they are not made from the browser
- Use plugins to alter Cypress's default behavior or do things outside of the browser
- Create custom commands to avoid repeating the same code multiple times
- Use hooks to execute code before or after tests