Back to the blog

Tutorials

Stop using Page Objects and Start using App Actions

Gleb Bahmutov / January 3 2019

Writing maintainable end-to-end tests is challenging. Often testers create another layer of indirection on top of the web page called **page objects** to execute common actions. In this post I argue that page objects are a bad practice, and suggest dispatching actions directly to the application's internal logic. This works great with a modern Cypress.io test runner that runs the test code directly alongside the application's code.

Page objects

Page objects 1, 2 are intended to make end-to-end tests readable and easy to maintain. Instead of ad-hoc interactions with a page, a test controls the page using an instance that represents the page user interface. For example here is a login page abstraction taken directly from the Selenium Wiki page.

```
public class LoginPage {
        private final WebDriver driver;

        public LoginPage(WebDriver driver) {
                this.driver = driver;
        // Check that we're on the right page.
                if (!"Login".equals(driver.getTitle())) {
                        // Alternatively, we could navigate to the
                        // login page, perhaps logging out first
                        throw new IllegalStateException("This is not
                }
        }
        // The login page contains several HTML elements
        // that will be represented as WebElements.
        // The locators for these elements should only be defined onc
        By usernameLocator = By.id("username");
        By passwordLocator = By.id("password");
        By loginButtonLocator = By.id("login");

        // The login page allows the user to type their
        // username into the username field
        public LoginPage typeUsername(String username) {
                // This is the only place that "knows" how to enter a
                driver.findElement(usernameLocator).sendKeys(username

                // Return the current page object as this action does
                // navigate to a page represented by another PageObje
                return this;
        }

        // other methods
        //  - typePassword
        //  - submitLogin
        //  - submitLoginExpectingFailure
        //  - loginAs
}
```

Page objects have two main benefits:

1. They keep all page element selectors in one place
2. They standardize how tests interact with the page

A typical test would use page objects like this for example:

```
public void testLogin() {
        LoginPage login = new LoginPage(driver);
        login.typeUsername('username')
        login.typePassword('username')
        login.submitLogin()
}
```

Martin Fowler in his PageObject post describes page objects as another API on top of HTML. Conceptually, they are sitting on top of HTML.

```
     Tests
-----------------
   Page Objects
~ ~ ~ ~ ~ ~ ~ ~ ~
        HTML UI
-----------------
Application code
```

The 4 levels in the above diagram have 3 interfaces of different tightness.

1. App code to HTML is tight
2. HTML to Page Objects is very loose
3. Tests to Page Objects is tight

The coupling of application code to the HTML UI is very tight, because the code renders the HTML elements into the DOM - there is a one to one relationship between the `render` function in the code and the output DOM elements. Static types and linters help ensure that the application's code is consistent and outputs meaningful HTML.

Page objects to HTML are very loosely coupled, that's why I have drawn the boundary using `~ ~` characters. They use selectors to find elements, which is NOT checked by any linter or code compiler! Application code can change at any moment, output a different DOM structure or different element classes, and the tests will break *at runtime* without warning.

💡This is a great untapped opportunity for someone to write a linter that would tie output HTML to selectors used in tests.

Finally, the page objects to tests boundary is tight - because both levels are in the same code and can be checked by the compiler against a programmer's mistakes.

Page objects in Cypress

You can easily use page objects in Cypress tests. Here is a typical example from the blog post Deep diving PageObject pattern and using it with Cypress. A typical PageObject class `SignInPage` is similar to the `LoginPage` from Selenium shown above.

```
class SignInPage {
        visit() {
                cy.visit('/signin');
        }

        getEmailError() {
                return cy.get(`[data-testid=SignInEmailError]`);
        }

        getPasswordError() {
                return cy.get(`[data-testid=SignInPasswordError]`);
        }

        fillEmail(value) {
                const field = cy.get(`[data-testid=SignInEmailField]`
                field.clear();
                field.type(value);

                return this;
        }

        fillPassword(value) {
                const field = cy.get(`[data-testid=SignInPasswordFiel
                field.clear();
                field.type(value);

                return this;
        }

        submit() {
                const button = cy.get(`[data-testid=SignInSubmitButto
                button.click();
        }
}

export default SignInPage;
```

When writing a test for “Home page” we can reuse the `SignInPage` from another Page Object

```
import Header from './Headers';
import SignInPage from './SignIn';

class HomePage {
        constructor() {
                this.header = new Header();
        }

        visit() {
                cy.visit('/');
        }

        getUserAvatar() {
                return cy.get(`[data-testid=UserAvatar]`);
        }

        goToSignIn() {
                const link = this.header.getSignInLink();
                link.click();
                const signIn = new SignInPage();
                return signIn;
        }
}

export default HomePage;
```

This is a typical scenario - you have to write an entire PageObject class hierarchy, where parts of the page are using different page objects, composing them using object-oriented design. A typical test then looks like this.

```
import HomePage from '../elements/pages/HomePage';

describe('Sign In', () => {

        it('should show an error message on empty input', () => {
                const home = new HomePage();
                home.visit();
                const signIn = home.goToSignIn();

                signIn.submit();

                signIn.getEmailError()
                        .should('exist')
                        .contains('Email is required');

                signIn
                        .getPasswordError()
                        .should('exist')
                        .contains('Password is required');
                });

        // more tests
});
```

Cypress comes with a JavaScript bundler included, so the above code just works.

You do not have to use the object-oriented PageObject implementation. You can also move typical logic into reusable Cypress Custom Commands that do not have any internal state and just allow you to reuse code. For example, you could implement a "login" command.

```
// in cypress/support/commands.js
Cypress.Commands.add('login', (username, password) => {
        cy.get('#login-username').type(username)
        cy.get('#login-password').type(password)
        cy.get('#login').submit()
})
```

After adding a custom command, the tests can use it just like any built-in command.

```
// cypress/integration/spec.js
it('logs in', () => {
        cy.visit('/login')
        cy.login('username', 'password')
})
```

Note that you do not have to always create custom commands and simple JavaScript functions work just as well (if not better because the type check step can understand individual function signatures).

```
// cypress/integration/util.js
export const login = (username, password) => {
        cy.get('#login-username').type(username)
        cy.get('#login-password').type(password)
        cy.get('#login').submit()
}
```

```
// cypress/integration/spec.js
import { login } from './util'

it('logs in', () => {
        cy.visit('/login')
        login('username', 'password')
})
```

Page objects problems

In the next sections I will look at concrete examples where the PageObject pattern falls short from what we need to write good end-to-end tests.

- Page objects are hard to maintain and take away time from actual application development. I have never seen PageObjects documented well enough to actually help one write tests.
- Page objects introduce additional state into the tests, which is separate from the application's internal state. This makes understanding the tests and failures harder.
- Page objects try to fit multiple cases into a uniform interface, falling back to conditional logic - a huge anti-pattern in our opinion.

- Page objects make tests slow because they force the tests to always go through the application user interface.

Do not despair! I will also show an alternative to page objects that I call **"Application Actions"** that our end-to-end tests can use. I believe application actions solve the above problems very well, making end-to-end tests fast and productive.

Adding items example

Let us take the TodoMVC tests as an example. First we will test if the user can enter todos. We will use Cypress to do this via the UI - just like a real user would enter items.

```
describe('TodoMVC', function () {
        // set up these constants to match what TodoMVC does
        let TODO_ITEM_ONE = 'buy some cheese'
        let TODO_ITEM_TWO = 'feed the cat'
        let TODO_ITEM_THREE = 'book a doctors appointment'

        beforeEach(function () {
                cy.visit('/')
        })
        context('New Todo', function () {
                it('should allow me to add todo items', function () {
                        cy.get('.new-todo').type(TODO_ITEM_ONE).type(
                        cy.get('.todo-list li').eq(0).find('label').s
                        cy.get('.new-todo').type(TODO_ITEM_TWO).type(
                        cy.get('.todo-list li').eq(1).find('label').s

                // more tests for adding items
                // - adds items
                // - should clear text input field when an item is ad
                // - should append new items to the bottom of the lis
                // - should trim text input
                // - should show #main and #footer when items added
        })
})
```

All these tests inside the “New Todo” block enter items using the `<input class="new-todo" />` without taking any shortcuts. Here are these tests running by themselves.

Every test starts by typing “buy some cheese”, and a few other items, just like a real cheese-loving user would.

Completing items example

Now let us test the “marking all items as completed” application feature. The user can click on an element in our application with markup to mark all current items completed.

```
<input
       className='toggle-all'
       type='checkbox'
       onChange={this.toggleAll}
       checked={activeTodoCount === 0} />
```

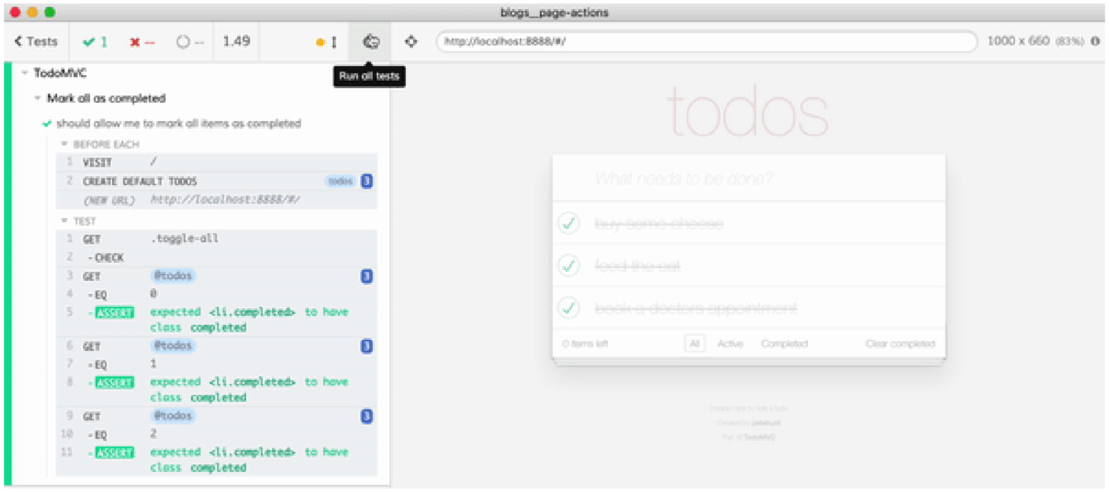
Here is the million dollar question - how do we enter todo items *before* clicking on `.toggle-all` ? We could write and use a custom command like `cy.createDefaultTodos().as('todos')` to go through the page UI interface, in essence manipulating the page to create items.

```
// cypress/support/commands.js
const TODO_ITEM_ONE = 'buy some cheese'
const TODO_ITEM_TWO = 'feed the cat'
const TODO_ITEM_THREE = 'book a doctors appointment'

Cypress.Commands.add('createDefaultTodos', function () {
        cy.get('.new-todo')
                .type(`${TODO_ITEM_ONE}{enter}`)
                .type(`${TODO_ITEM_TWO}{enter}`)
                .type(`${TODO_ITEM_THREE}{enter}`)
                .get('.todo-list li')
})
```

We will create this new custom command `createDefaultTodos` before each test in the block.

```
// cypress/integration/spec.js
context('Mark all as completed', function () {
        beforeEach(function () {
                cy.createDefaultTodos().as('todos')
        })

        it('should allow me to mark all items as completed', function
                // complete all todos
                // we use 'check' instead of 'click'
                // because that indicates our intention much clearer
                cy.get('.toggle-all').check()
                // get each todo li and ensure its class is 'complete
                cy.get('@todos').eq(0).should('have.class', 'complete
                cy.get('@todos').eq(1).should('have.class', 'complete
                cy.get('@todos').eq(2).should('have.class', 'complete
        })
        // more tests
        // - should allow me to clear the complete state of all items
        // - complete all checkbox should update state when items are
})
```

Here is the first test by itself



```
TodoMVC
  Mark all as completed
    ✓ should allow me to mark all items as completed
      BEFORE EACH
      1  VISIT                 /
      2  CREATE DEFAULT TODOS                      todos
         (NEW URL)             http://localhost:8888/#/
      TEST
      1  GET                   .toggle-all
      2   -CHECK
      3  GET                   @todos
      4   -EQ                  0
      5   -ASSERT              expected <li.completed> to have
                               class completed
      6  GET                   @todos
      7   -EQ                  1
      8   -ASSERT              expected <li.completed> to have
                               class completed
      9  GET                   @todos
      10  -EQ                  2
      11  -ASSERT              expected <li.completed> to have
                               class completed
```

But consider two things:

1. We are always entering the items via the UI - repeating what every test in context "New Todo" has done.
2. The majority of the test's running time was taken up by entering the items, and not by checking the item.

The last point is important - our tests are slow because of entering 3 items via the UI before each test. The 3 tests in the above context "Mark all as completed" take usually somewhere between 4 and 5 seconds.

Application actions

> **Source Code:** You can find the final source code for this blog post in the Application Actions recipe. You can also see the same tests implemented in different styles including Page Object and App Actions in repo bahmutov/test-todomvc-using-app-actions.

Imagine that instead of always entering new items via the UI we could set the state of the application directly from our test. Because Cypress architecture allows interacting with the application under test, this is simple. All we need to do is to expose a reference to the application's model object by attaching it to the `window` object for example.

```
// app.jsx code
var model = new app.TodoModel('react-todos');
```

```
if (window.Cypress) {
        window.model = model
}
```

Setting a `model` reference as a property of the application's `window` object gives our tests an easy way to call a method `model.addTodo` that already exists in `js/todoModel.js`

```
// js/todoModel.js
// Model: keeps all todos and has methods to act on them
app.TodoModel = function (key) {
        this.key = key  this.todos = Utils.store(key)
        this.onChanges = []
}
app.TodoModel.prototype.addTodo = function (title) {
        this.todos = this.todos.concat({
                id: Utils.uuid(),
                title: title,
                completed: false
        });

        this.inform();
};

app.TodoModel.prototype.inform = ...
app.TodoModel.prototype.toggleAll = ...
// other methods
```

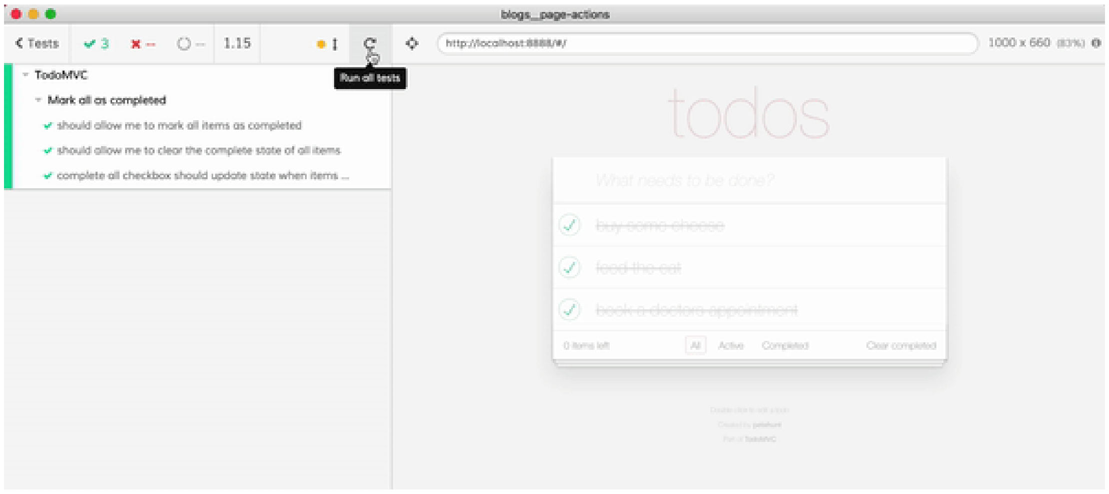
Instead of using a page object custom command to create todos like `cy.createDefaultTodos().as('todos')` we can use `model.addTodo` to add items directly using the application's internal "api". In the code below I am using `cy.window()` to grab the application's window, then its property `model` and then `.invoke()` to call the method `addTodo` on the model instance.

```
beforeEach(function () {
        cy.window().its('model').invoke('addTodo', TODO_ITEM_ONE)
        cy.window().its('model').invoke('addTodo', TODO_ITEM_TWO)
        cy.window().its('model').invoke('addTodo', TODO_ITEM_THREE)
        cy.get('.todo-list li').as('todos')
})
```

With the above setup, our tests are running *much* faster - all 3 finish in slightly more than 1 second, already 3 times faster than before. But even the above code is slower than necessary - because we are using several Cypress commands to add each item, which brings overhead. Instead we can change the `TodoModel.prototype.addTodo` to accept multiple items at once.

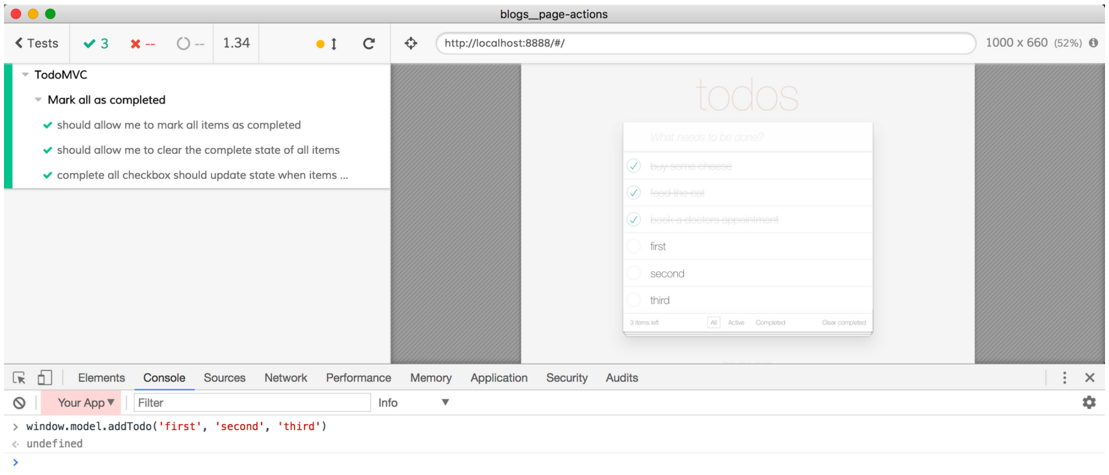
```
// js/todoModel.js
app.TodoModel.prototype.addTodo = function (...titles) {
        titles.forEach(title => {
                this.todos = this.todos.concat({
                        id: Utils.uuid(),
                        title: title,
                        completed: false
                });
        })

        this.inform();
};
```

```
// cypress/integration/spec.js
beforeEach(function () {
                cy.window().its('model').invoke('addTodo',
                        TODO_ITEM_ONE, TODO_ITEM_TWO, TODO_ITEM_THREE
                cy.get('.todo-list li').as('todos')
})
```



Have you noticed what we did to make our test better? We have NOT changed the test code, instead we improved the *application code*. By using the application's internal actions from our end-to-end tests we are bound to make the application better while writing our tests! Our effort directly leads to clarifying the model's interface, making it more testable, better documented, and making it easier to use *from other application code*.

You can also invoke application actions from the DevTools console directly by switching the context to "Your app", see the screenshot below.



```
> window.model.addTodo('first', 'second', 'third')
< undefined
>
```

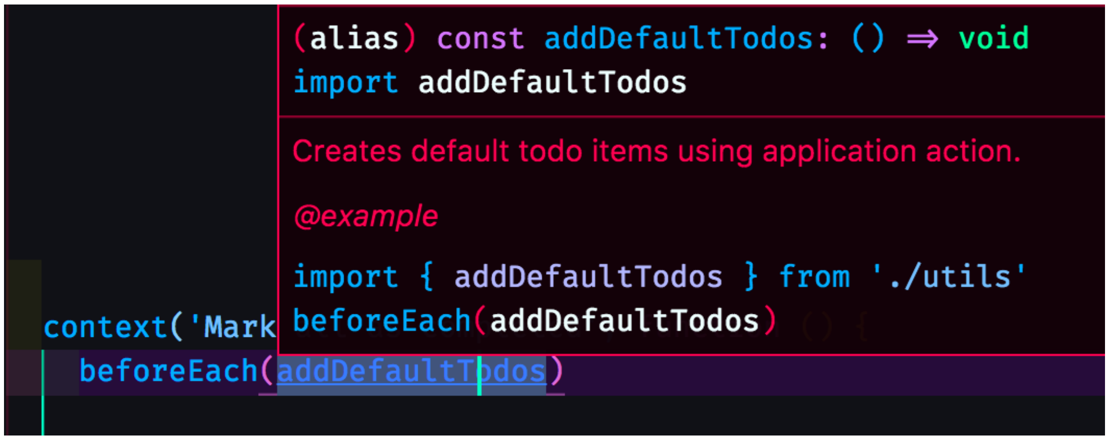
Just functions

We can move the application actions logic to custom commands, replacing using the UI to manipulate state with the invoking application internal model interface. But I prefer creating small reusable functions, rather than attaching additional methods to the `cy` object.

```
const addDefaultTodos = () => {
                cy.window().its('model').invoke('addTodo',
                        TODO_ITEM_ONE, TODO_ITEM_TWO, TODO_ITEM_THREE
                cy.get('.todo-list li').as('todos')
}

beforeEach(addDefaultTodos)
```

Because Cypress comes with a bundler included, we can move `addDefaultTodos` to a separate file with utilities and use `require` or `import` directives to use it from the spec file. And we can document `addDefaultTodos` using JSDoc convention to get beautiful intelligent code completion in our test files.

```
// utils.js
const TODO_ITEM_ONE = 'buy some cheese'const
TODO_ITEM_TWO = 'feed the cat'const
TODO_ITEM_THREE = 'book a doctors appointment'

/**
        * Creates default todo items using application action.
        * @example
        *  import { addDefaultTodos } from './utils'
        *  beforeEach(addDefaultTodos)
*/

export const addDefaultTodos = () => {
        cy.window().its('model').invoke('addTodo',
                TODO_ITEM_ONE, TODO_ITEM_TWO, TODO_ITEM_THREE)
        cy.get('.todo-list li').as('todos')
}
```



```
(alias) const addDefaultTodos: () => void
import addDefaultTodos

Creates default todo items using application action.

@example

import { addDefaultTodos } from './utils'
beforeEach(addDefaultTodos)
```

```
context('Mark
  beforeEach(addDefaultTodos)
```

Using application actions is just using JavaScript functions, and using functions is simple.

Persistance example

There is another example in the TodoMVC tests that shows the power of setting initial state and actions. The persistance test adds two items, clicks on one of them, then reloads the page. The two items should be there and the completed state should be preserved. The original test in Cypress does everything through the UI.

```
context('Persistence', function () {
        it('should persist its data', function () {
                // mimicking TodoMVC tests
                // by writing out this function
                function testState () {
                        cy.get('@firstTodo').should('contain', TODO_I
                                .and('have.class', 'completed')
                        cy.get('@secondTodo').should('contain', TODO_
                                .and('not.have.class', 'completed')
        }
        cy.createTodo(TODO_ITEM_ONE).as('firstTodo')
        cy.createTodo(TODO_ITEM_TWO).as('secondTodo')
        cy.get('@firstTodo').find('.toggle').check()
                .then(testState)
                .reload()
                .then(testState)
        })
})
```

The helper function `testState` checks both items - first should be completed, and the second should not be. We check before reloading the page and after.

But why are we even creating items, and why are we clicking on the first items to mark it complete? We know it works! We have another test above that *has already tested the UI* for completing an item. That test was called `Item - should allow me to mark items as complete` and it looks almost exactly the same:

```
context('Item', function () {
        it('should allow me to mark items as complete', function () {
                cy.createTodo(TODO_ITEM_ONE).as('firstTodo')
                cy.createTodo(TODO_ITEM_TWO).as('secondTodo')

                cy.get('@firstTodo').find('.toggle').check()
                cy.get('@firstTodo').should('have.class', 'completed'

                cy.get('@secondTodo').should('not.have.class', 'compl
                cy.get('@secondTodo').find('.toggle').check()

                cy.get('@firstTodo').should('have.class', 'completed'
                cy.get('@secondTodo').should('have.class', 'completed
        })
})
```

We should NOT repeat the tests for the same user interface actions. We should NOT repeat UI interactions even if we follow the best practices and test the features and not the implementation by using test ids and a good helper library like cypress-testing-library - it is still tying our tests to the page structure, and that can change.

Here is our original application manipulation using the UI.

```
cy.createTodo(TODO_ITEM_ONE).as('firstTodo')
cy.createTodo(TODO_ITEM_TWO).as('secondTodo')
cy.get('@firstTodo').find('.toggle').check()
```

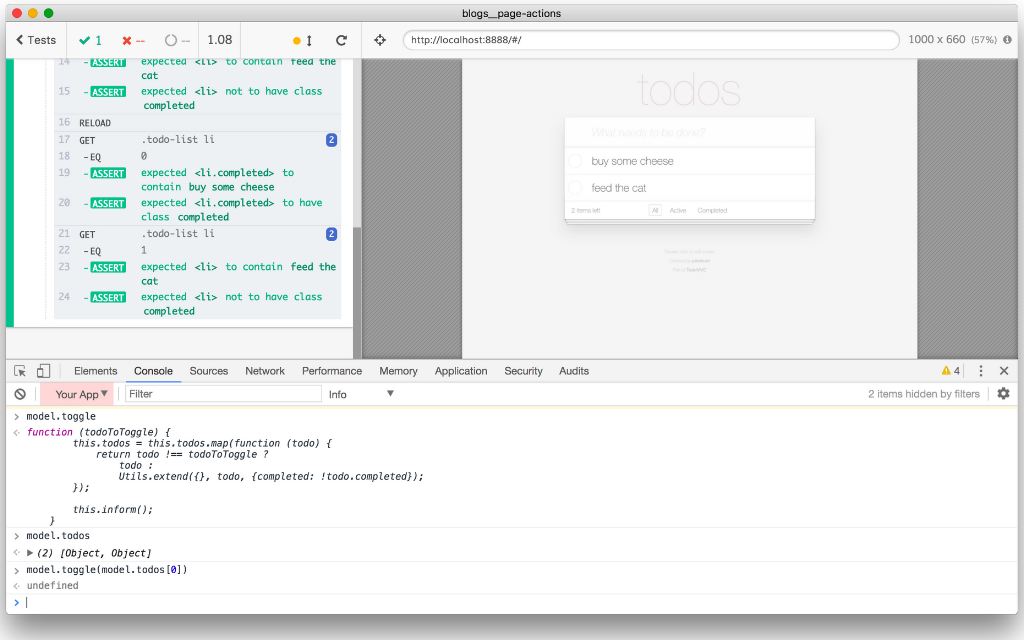
And here is how we can redo this to use application actions. First we can use our utility function `addTodo` to control the application, and still use the checkbox `class ="toggle"` to toggle the first item as completed.

```
// spec.js
import { addTodos } from './utils';
addTodos(TODO_ITEM_ONE, TODO_ITEM_TWO)c
y.get('.todo-list li').eq(0).find('.toggle').check()
```

Next we can look at the model methods in the `todoModel.js` to see how we can toggle a todo item directly.

```
app.TodoModel.prototype.toggle = function (todoToToggle) {
        this.todos = this.todos.map(function (todo) {
                return todo !== todoToToggle ?
                        todo :
                        Utils.extend({}, todo, {completed: !todo.comp
        });

        this.inform();
};
```

Can we use the `model.toggle` method to toggle the `completed` flag? Cypress can do anything you can do from the DevTools. So again we can open DevTools from the test runner, switch to “Your App” context and try. Notice how after the test has finished I called `model.toggle(model.todos[0])` and the first item in the app has switched back to being incomplete.



```
14  - ASSERT   expected <li> to contain feed the
               cat
15  - ASSERT   expected <li> not to have class
               completed
16  RELOAD
17  GET        .todo-list li                    2
18   - EQ      0
19  - ASSERT   expected <li.completed> to
               contain buy some cheese
20  - ASSERT   expected <li.completed> to have
               class completed
21  GET        .todo-list li                    2
22   - EQ      1
23  - ASSERT   expected <li> to contain feed the
               cat
24  - ASSERT   expected <li> not to have class
               completed
```

```
> model.toggle
< function (todoToToggle) {
          this.todos = this.todos.map(function (todo) {
              return todo !== todoToToggle ?
                  todo :
                  Utils.extend({}, todo, {completed: !todo.completed});
          });

          this.inform();
      }
> model.todos
< ▶ (2) [Object, Object]
> model.toggle(model.todos[0])
< undefined
> 
```

Let us write a utility function to invoke the application action `toggle`. For our tests we probably want to toggle an item not by reference but by index.

```
/**
* Toggle given todo item. Returns chain so you can attach more Cypres
* @param {number} k index of the todo item to toggle, 0 - first item
* @example import { addTodos, toggle } from './utils' it('completes a
*/
export const toggle = (k = 0) =>
        cy.window().its('model')
                .then(model => {
                        expect(k, 'check item index').to.be.lessThan(
                        model.toggle(model.todos[k])
                })
```

Alternatively, we could have changed the application's model `toggle` function to take an index as an argument. See how the testing code is now a "client" of the application's code and can influence the application's architecture and design?

Our changed test creates the items and toggles the first one, and it runs quickly.

```
context('Persistence', function () {
        // mimicking TodoMVC tests
        // by writing out this function
        function testState () {
                cy.get('.todo-list li').eq(0)
                        .should('contain', TODO_ITEM_ONE).and('have.c
                cy.get('.todo-list li').eq(1)
                        .should('contain', TODO_ITEM_TWO).and('not.ha
        }

        it('should persist its data', function () {
                addTodos(TODO_ITEM_ONE, TODO_ITEM_TWO)
                toggle(0)
                        .then(testState)

                        .reload()
                        .then(testState)
        })
})
```

Now I can go through other tests and replace every `cy.get('.todo-list li').eq(k).find('.toggle').check()` with `toggle(k)`. Faster and future-proof.

Similarly, we can update routing end-to-end tests to NOT go through the UI elements when setting up the page, instead using application actions. At the same time we leave the clicking on the actual links we are testing as is - the test is asserting that the user interface link with text “Active” works!

```
context('Routing', function () {
        beforeEach(addDefaultTodos) // app action

        it('should allow me to display active items', function () {
                toggle(1) // app action
                // the UI feature we are actually testing - the "Acti
                cy.get('.filters').contains('Active').click()
                cy.get('@todos').eq(0).should('contain', TODO_ITEM_ON
                cy.get('@todos').eq(1).should('contain', TODO_ITEM_TH
        })
        // more tests
})
```

Notice that whenever you end up writing a slightly longer utility test function like `toggle`, it is a good indicator that maybe the application's internal interface might need changing instead of writing more test code!

```
// hmm, maybe we need to add a `model.toggleIndex()` method?
export const toggle = (k = 0) =>
        cy.window().its('model')
                .then(model => {
                        xpect(k, 'check item index').to.be.lessThan(m
                model.toggle(model.todos[k])
        })
```

If we do add a `model.toggleIndex` method to the application, then the app will become more testable, and maybe even easier to develop in the future. The testing code will also be simplified.

DRY test code

Each block of tests are really just a closure. We can use this to our advantage with app actions. The element's selectors passed into a test block will be localized to that block. This naturally keeps the selectors local to each closure. All of the following test blocks can use app actions and do not need to know about the selector. In the example below look at the selectors `NEW_TODO` and `TOGGLE_ALL`.

```
describe('TodoMVC', function () {
        // testing item input  context('New Todo', function () {
        // selector to enter new todo item is private to these tests
        const NEW_TODO = '.new-todo'
        it('should allow me to add todo items', function () {
                cy.get(NEW_TODO)
                        .type(TODO_ITEM_ONE)
                        .type('{enter}')
                // more commands
        })
        // more tests that use NEW_TODO selector
})
```

```
        // testing toggling all items
        context('Mark all as completed', function () {
                // selector to toggle all items is private to these t
                const TOGGLE_ALL = '.toggle-all'
                beforeEach(addDefaultTodos)
                it('should allow me to mark all items as completed',
                        cy.get(TOGGLE_ALL).check()
                        // more commands
                })
                // more tests that use TOGGLE_ALL selector
        })
})
```

The above tests show how each selector is private to the specific block of tests. For example the selector `const NEW_TODO = '.new-todo'` is private to the block of tests `"New Todo"` , and the selector `const TOGGLE_ALL = '.toggle-all'` is private to the block of tests `"Mark all as completed"` . Other tests do not need to know the selectors for the page elements to add items or mark all completed - the tests can use app actions instead.

But in some situations you might want to share a selector. For example many tests from multiple blocks might need to grab all Todo items on the page, and there is no getting away from this. We can still keep the selector in the tests without creating page objects as an `ALL_ITEMS` local variable.

```
describe('TodoMVC', function () {
        // common selector used across many tests
        const ALL_ITEMS = '.todo-list li'

        context('New Todo', function () {
                const NEW_TODO = '.new-todo'

                it('should allow me to add todo items', function () {
                        cy.get(NEW_TODO)
                                .type(TODO_ITEM_ONE)
                                .type('{enter}')
                        cy.get(ALL_ITEMS)
                                .eq(0)
                                .find('label')
                                .should('contain', TODO_ITEM_ONE)
        })
        // more tests
    })

    context('Mark all as completed', function () {
        const TOGGLE_ALL = '.toggle-all'

        beforeEach(addDefaultTodos)

        it('should allow me to mark all items as completed',
            cy.get(TOGGLE_ALL).check()
            cy.get(ALL_ITEMS)
                .eq(0)
                .should('have.class', 'completed')
        })
        // more tests
    })
})
```

In the above example we are using the `const ALL_ITEMS = '.todo-list li'` selector in multiple tests. I even prefer to create a local utility function `allItems` to return all list items rather than share a selector constant.

```
describe('TodoMVC', function () {
    const ALL_ITEMS = '.todo-list li'

    /**
    * Returns all todo items
    */

    const allItems = () => cy.get(ALL_ITEMS)
    context('New Todo', function () {
        const NEW_TODO = '.new-todo'
        it('should allow me to add todo items', function () {
            cy.get(NEW_TODO)
                .type(TODO_ITEM_ONE)
                .type('{enter}')
            allItems()
                .eq(0)
                .find('label')
                .should('contain', TODO_ITEM_ONE)
        })
```

```
                // more tests
        })
        context('Mark all as completed', function () {
                const TOGGLE_ALL = '.toggle-all'

                beforeEach(addDefaultTodos)

                it('should allow me to mark all items as completed',
                        cy.get(TOGGLE_ALL).check()
                        allItems()
                                .eq(0)
                                .should('have.class', 'completed')
                })
                // more tests
        })
})
```

As the number of tests grows, we might naturally split our single spec file into multiple spec files. This would allow our continuous integration server to run all tests in parallel. In that case we might want to move the utility function `allItems` and the selector `ALL_ITEMS` to a common utility file and import `allItems` from all specs that need it.

```
// cypress/integration/utils.js
const ALL_ITEMS = '.todo-list li'

/**
* Returns all todo items
* @example    import {allItems} from './utils'    allItems().should('
*/
export const allItems = () => cy.get(ALL_ITEMS)

// cypress/integration/spec.js
import { allItems } from './utils'

describe('TodoMVC', function () {
        context('New Todo', function () {
                const NEW_TODO = '.new-todo'

                it('should allow me to add todo items', function () {
                        cy.get(NEW_TODO)
                        .type(TODO_ITEM_ONE)
                        .type('{enter}')
                allItems()
                        .eq(0)
                        .find('label')
                        .should('contain', TODO_ITEM_ONE)
        })
        // more tests
    })
    context('Mark all as completed', function () {
        const TOGGLE_ALL = '.toggle-all'
        beforeEach(addDefaultTodos)
        it('should allow me to mark all items as completed',
                cy.get(TOGGLE_ALL).check()
                allItems()
                        .eq(0)
                        .should('have.class', 'completed')
        })
        // more tests
    })
})
```

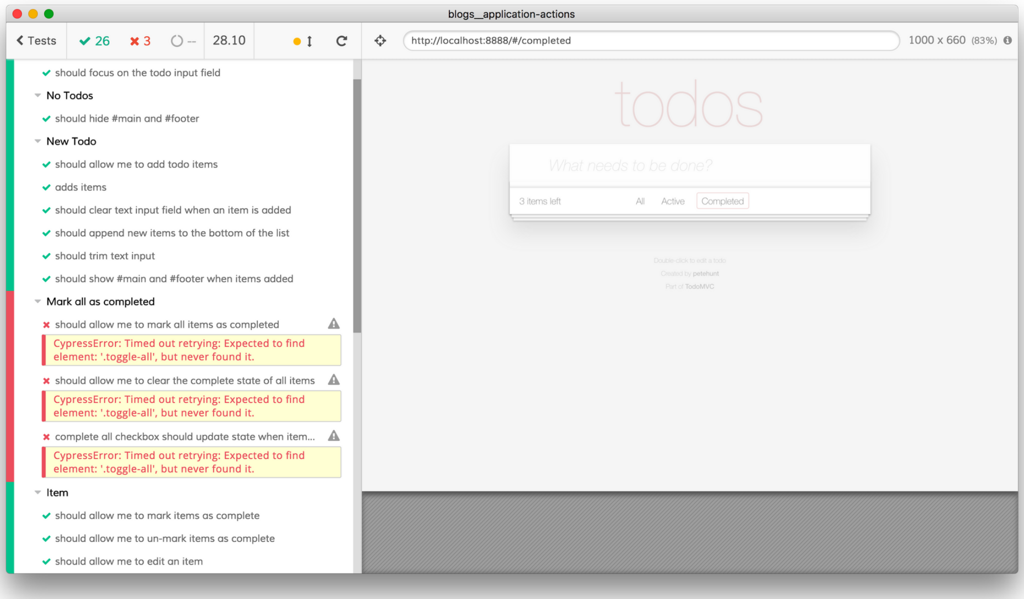
The goal is to make tests easy to read, simple to understand and safe to change when needed.

Focused errors

One beautiful benefit of using application actions to drive tests are very focused errors. For example, the application UI has a checkbox to toggle all items as completed. The code looks like this.

```
if (todos.length) {
        main = (
                <section className='main'>
            <input
                className='toggle-all'
                type='checkbox'
                onChange={this.toggleAll}
                checked={activeTodoCount === 0}
                />
            <ul className='todo-list'>{todoItems}</ul>
            </section>
        )
    }
```

If I remove the `<input className='toggle-all' ... />` element, only the tests inside the "Mark all as completed" block break.



```
✓ should focus on the todo input field
No Todos
✓ should hide #main and #footer
New Todo
✓ should allow me to add todo items
✓ adds items
✓ should clear text input field when an item is added
✓ should append new items to the bottom of the list
✓ should trim text input
✓ should show #main and #footer when items added
Mark all as completed
✗ should allow me to mark all items as completed
  CypressError: Timed out retrying: Expected to find
  element: '.toggle-all', but never found it.
✗ should allow me to clear the complete state of all items
  CypressError: Timed out retrying: Expected to find
  element: '.toggle-all', but never found it.
✗ complete all checkbox should update state when item...
  CypressError: Timed out retrying: Expected to find
  element: '.toggle-all', but never found it.
Item
✓ should allow me to mark items as complete
✓ should allow me to un-mark items as complete
✓ should allow me to edit an item
```

No other test is going through this user interface element, thus no other test breaks.

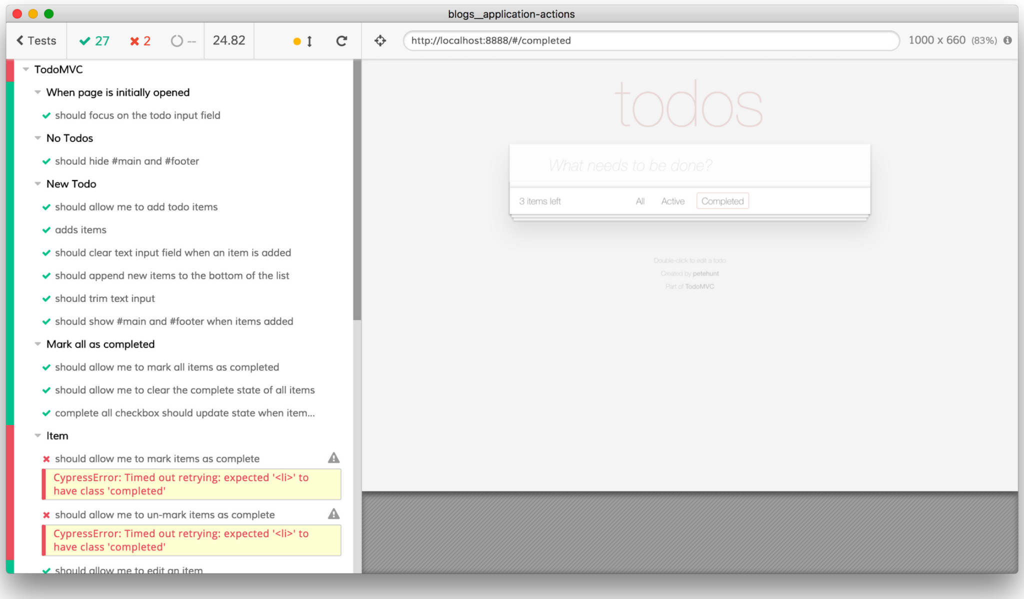
Similarly, each Todo item renders a checkbox to mark its own item completed. The code looks like this.

```
<input
        className="toggle"
        type="checkbox"
        checked={this.props.todo.completed}
        onChange={this.props.onToggle}
/>
```

If I comment out the `onChange={this.props.onToggle}` line like this.

```
<input
        className='toggle'
        type='checkbox'
        checked={this.props.todo.completed}
        // onChange={this.props.onToggle}
/>
```

Then only the tests for completing the individual items break.



```
TodoMVC
  When page is initially opened
    ✓ should focus on the todo input field
  No Todos
    ✓ should hide #main and #footer
  New Todo
    ✓ should allow me to add todo items
    ✓ adds items
    ✓ should clear text input field when an item is added
    ✓ should append new items to the bottom of the list
    ✓ should trim text input
    ✓ should show #main and #footer when items added
  Mark all as completed
    ✓ should allow me to mark all items as completed
    ✓ should allow me to clear the complete state of all items
    ✓ complete all checkbox should update state when item...
  Item
    ✗ should allow me to mark items as complete
      CypressError: Timed out retrying: expected '<li>' to
      have class 'completed'
    ✗ should allow me to un-mark items as complete
      CypressError: Timed out retrying: expected '<li>' to
      have class 'completed'
    ✓ should allow me to edit an item
```

I am really happy about having one page feature only affect a single set of tests. This is a welcome change from the typical “some small UI thing has changed - half of the end-to-end tests are red now”.

If we use TypeScript to write our application, our tests can even use the application model interface definition to correctly invoke the existing methods. This would speed up refactoring, because type definitions would allow immediately refactoring all places where the test code is calling into the application code.

Application actions limitations

Calling too many actions too fast

When using app actions to execute multiple operations, your tests might run ahead of the application. For example if the application saves added todos on the server before storing them locally, you cannot immediately mark them as complete.

```
// model
app.TodoModel.prototype.addTodo = function (...todos) {
        // make XHR to the server to save todos
        ajax({
                method: 'POST',
                url: '/todos',
                data: todos  }).then(() =>
                        then update local state
                        this.saveTodos(todos)
                ).then(() =>
                        // this triggers DOM render
                        this.inform()
        )
}
// spec.js
it('completes all items', () => {
        addDefaultTodos()
        toggle(1)
        // marks item completed
        // click on "Completed" link
        // assert there is 1 completed item
})
```

The above test `completes all items` is likely to sometimes pass and sometimes fail. And it is all because the test runs faster than the application can handle actions.

For example while the application is still adding new todos inside the `addTodo` method, the test is already sending a `toggle` action which will try completing the todo item with index 1. Maybe the application had enough time to send the original list of todos to the server and set them in the local state - in that case the test will pass. But most of the time the application is still waiting for the server to respond - in this case the local list of items is still empty and trying to toggle the item with index 1 will trigger an error.

By using app actions to drive the application, we moved away from the way a user would use our application. Users would not be able to toggle an item before an item is shown to the user on the page. Thus our tests need to *wait for items to appear in the UI* before executing `toggle(1)` . Again a simple reusable function should be enough.

```
it('completes all items', () => {
        addDefaultTodos()
        allItems().should('have.length', 3)
        toggle(1)
        // marks item completed
        // click on "Completed" link
        // assert there is 1 completed item
})
```

I highly recommend the pattern shown above - execute an app action, wait for UI to update to desired state by writing an assertion, then execute another app action, and again wait for the UI to update. This runs as fast as possible because Cypress can directly observe the DOM and continue to the next action as soon as the assertion passes.

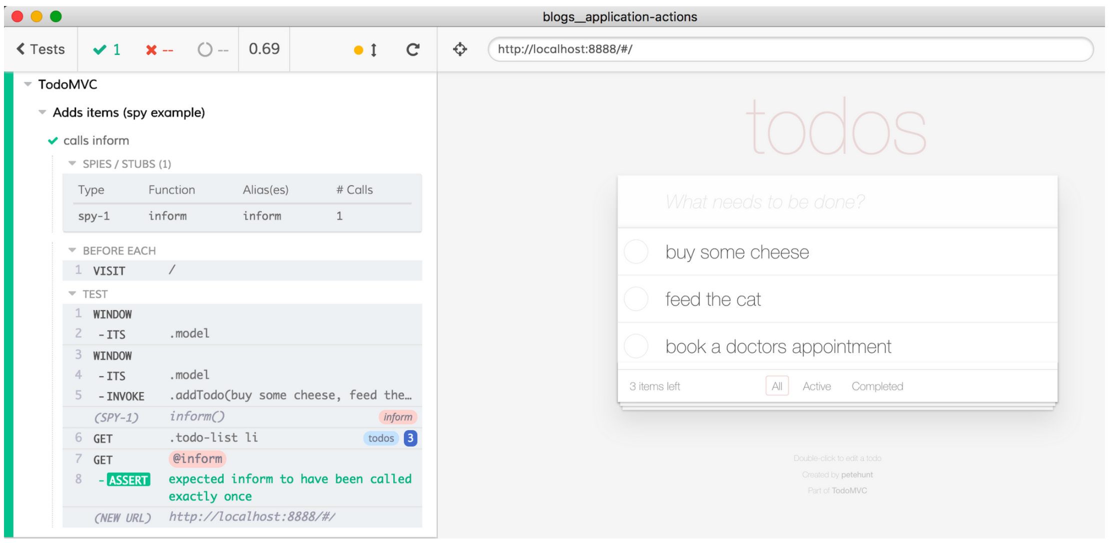
You are not restricted to observing the DOM - you can just as easily spy on the network calls. For example we can spy on the `POST /todos` XHR call from the application to the server and wait for the network call before executing the action `toggle(1)` .

```
it('completes all items', () => {
        cy.server()  cy.route('POST', '/todos').as('save')
        addDefaultTodos()
        cy.wait('@save')
        // waits for XHR POST /todos before test continues
        toggle(1) // marks item completed
        // click on "Completed" link
        // assert there is 1 completed item
})
```

Even better - we can spy on methods directly in our application! Since our application calls `model.inform` when it is done updating the state, that’s a good sign that we can call another app action.

```
it('completes all items', () => {
        cy.window()    .its('model')
                .then(model => {
                        cy.spy(model, 'inform').as('inform')
                })
        addDefaultTodos()
        // wait until the spy is called once
        cy.get('@inform').should('have.been.calledOnce')
        toggle(1) // marks item completed
        // click on "Completed" link
        // assert there is 1 completed item
})
```

The Cypress UI shows information about each method we are spying on in the Command Log.



```
TodoMVC
  Adds items (spy example)
    ✓ calls inform
      SPIES / STUBS (1)
        Type    Function   Alias(es)   # Calls
        spy-1   inform     inform      1
      BEFORE EACH
        1 VISIT      /
      TEST
        1 WINDOW
        2  - ITS     .model
        3 WINDOW
        4  - ITS     .model
        5  - INVOKE  .addTodo(buy some cheese, feed the…
          (SPY-1)    inform()                        inform
        6 GET        .todo-list li                   todos 3
        7 GET        @inform
        8  - ASSERT  expected inform to have been called
                     exactly once
          (NEW URL)  http://localhost:8888/#/
```

To summarize: app actions might be called from the test faster than the application can process them. In this case, you may interpret the tests as flaky due to the race between the test and the application. Luckily, you can synchronize the test and the application in several ways. The test can:

1. Wait for the DOM to be updated as expected.
2. Observe network traffic and wait for an expected XHR call to happen.
3. Spy on a method in the application and continue when it gets called.

Actions are restricted

Sometimes the application code cannot achieve the desired action. For example in the Cypress Best Practices talk, Brian Mann has argued that:

- When testing a login page, the end-to-end tests should use the UI just like the user does
- When testing any other user flow that requires a login, the test should execute the login directly (for example using the `cy.request()` command), and not go through the UI again and again.

In the above implementation, the application code *cannot* do the login using the same method as `cy.request`. Thus the end-to-end tests should call `cy.request()` and not invoke an application action. This still avoids using the page object pattern - a custom command or a simple function is enough to achieve it.

Final thoughts

Switching from page objects that always go through the page's user interface to application actions that control the application via its internal model API brings many benefits.

- The tests become much faster. Even the simple TodoMVC tests running locally against Cypress's Electron browser went from 34 seconds to 17 seconds after switching from going through the user interface to using application actions - a 50% speed up.
- The tests now influence and benefit from refactoring the application's code. The more sensible and better documented the application's internal interface becomes, the easier it will be to write end-to-end tests for them.
- You avoid writing a loosely coupled separate layer of code on top of ephemeral and unstable user interfaces. Instead the tests use and are tied to the longer-lasting internal model interface of the application.

In fact, the utility functions I had to write only map test syntax to application actions, and the majority are just stateless syntax sugar like.

```
export const addTodos = (...todos) => {
        cy.window().its('model').invoke('addTodo', ...todos)
}
```

There is no parallel state (inside page objects), no conditional testing logic - just directly invoking the application code, just like you can do from the DevTools console.

More information

You can find the source code for this post in the Application Actions recipe. You can also see the same tests implemented in different styles including Page Object and App Actions in repo bahmutov/test-todomvc-using-app-actions.

You can use the same application actions idea to control any application from your tests. In other blog posts I have already shown how to:

- Dispatch Redux actions and even share code between tests and application.
- Dispatch Vuex actions and check DOM updates and network requests.

The above examples and blog posts helped me see the shortcomings of page objects, and the benefits of application actions in end-to-end tests.

Gleb Bahmutov

Read more posts by this author.

Read More

0 Comments **cypress-io** **Dragan Nikolic**

Recommend Tweet Share **Sort by Best**

Start the discussion…

Be the first to comment.

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Disqus' Privacy PolicyPrivacy PolicyPrivacy**

Announcements - 2 min read

Happy 2019 from Cypress!

As our team gets a running start to the new year, we think reflecting on how far Cypress.io has come in 2018 serves as the perfect conduit to determining our goals for

The Cypress Team / January 9 2019

Tutorials - 6 min read

Element Coverage

Our users are periodically asking us to instrument application code and save code coverage information after Cypress runs end-to-end tests. In this blog I will argue that code coverage is less than useful

Gleb Bahmutov / December 20 2018

Get Started

Why Cypress

Install Cypress

Write your first test

Test your app

Developers

Status site

Contribute

Changelog

Roadmap

Send updates to my inbox

Your email address Subscribe

Support

Documentation

Chat

FAQ

Examples

Company

About

Testimonials

Jobs

Events

Media