# d End to end testing



So far we have tested the backend as a whole on an API level using integration tests and tested some frontend components using unit tests.

Next, we will look into one way to test the system as a whole using *End to End* (E2E) tests.

We can do E2E testing of a web application using a browser and a testing library. There are multiple libraries available. One example is Selenium, which can be used with almost any browser. Another browser option is so-called headless browsers, which are browsers with no graphical user interface. For example, Chrome can be used in headless mode.

E2E tests are potentially the most useful category of tests because they test the system through the same interface as real users use.

They do have some drawbacks too. Configuring E2E tests is more challenging than unit or integration tests. They also tend to be quite slow, and with a large system, their execution time can be minutes or even hours. This is bad for development because during coding it is beneficial to be able to run tests as often as possible in case of code regressions.

E2E tests can also be flaky. Some tests might pass one time and fail another, even if the code does not change at all.

## Cypress

E2E library Cypress has become popular within the last year. Cypress is exceptionally easy to use, and when compared to Selenium, for example, it requires a lot less hassle and headache. Its operating principle is radically different than most E2E testing libraries because Cypress tests are run completely within the browser. Other libraries run the tests in a Node process, which is connected to the browser through an API.

Let's make some end-to-end tests for our note application.

We begin by installing Cypress to *the frontend* as a development dependency

```
npm install --save-dev cypress
```

copy

and by adding an npm-script to run it:

```
{
  // ...
  "scripts": {
    "dev": "vite --host",
    "build": "vite build",
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-
warnings 0",
    "preview": "vite preview",
    "server": "json-server -p3001 --watch db.json",
    "test": "jest",
    "cypress:open": "cypress open"
  },
  // ...
}
```

copy

We also made a small change to the script that starts the application, without the change Cypress can not access the app.

Unlike the frontend's unit tests, Cypress tests can be in the frontend or the backend repository, or even in their separate repository.

The tests require the tested system to be running. Unlike our backend integration tests, Cypress tests *do not start* the system when they are run.

Let's add an npm script to *the backend* which starts it in test mode, or so that *NODE_ENV* is *test*.

```
{
  // ...
  "scripts": {
    "start": "NODE_ENV=production node index.js",
    "dev": "NODE_ENV=development nodemon index.js",
    "build:ui": "rm -rf build && cd ../frontend/ && npm run build && cp -r build
../backend",
    "deploy": "fly deploy",
    "deploy:full": "npm run build:ui && npm run deploy",
    "logs:prod": "fly logs",
    "lint": "eslint .",
    "test": "jest --verbose --runInBand",
    "start:test": "NODE_ENV=test node index.js"
  },
  // ...
}
```

copy

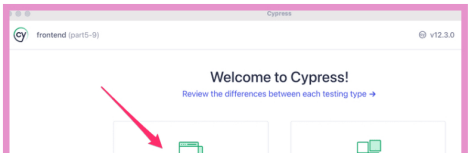
**NB** To get Cypress working with WSL2 one might need to do some additional configuring first. These two links are great places to start.

When both the backend and frontend are running, we can start Cypress with the command

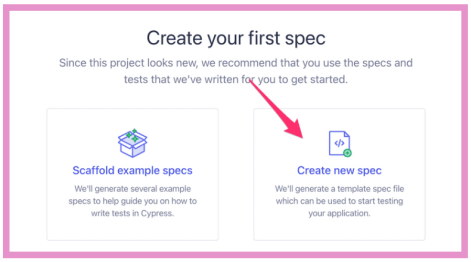
```
npm run cypress:open
```

copy

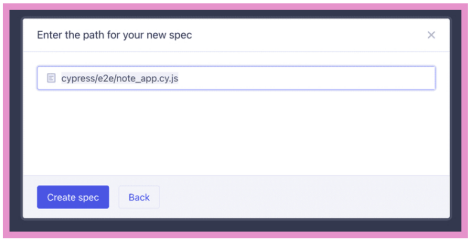
Cypress asks what type of tests we are doing. Let us answer "E2E Testing":



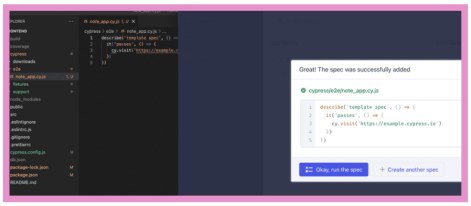

Next a browser is selected (e.g. Chrome) and then we click "Create new spec":




Let us create the test file *cypress/e2e/note_app.cy.js*:




We could edit the tests in Cypress but let us rather use VS Code:



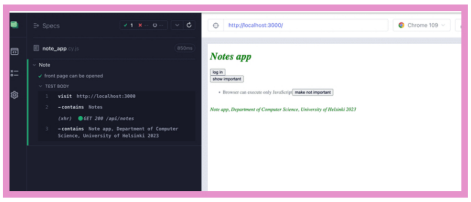

We can now close the edit view of Cypress.

Let us change the test content as follows:

```
describe('Note app', function() {
  it('front page can be opened', function() {
    cy.visit('http://localhost:5173')
    cy.contains('Notes')
    cy.contains('Note app, Department of Computer Science, University of Helsinki
2023')
  })
})
```

The test is run by clicking the test in the Cypress:

Running the test shows how the application behaves as the test is run:




The structure of the test should look familiar. They use *describe* blocks to group different test cases, just like Jest. The test cases have been defined with the *it* method. Cypress borrowed these parts from the Mocha testing library it uses under the hood.

cy.visit and cy.contains are Cypress commands, and their purpose is quite obvious. cy.visit opens the web address given to it as a parameter in the browser used by the test. cy.contains searches for the string it received as a parameter from the page.

We could have declared the test using an arrow function

```
describe('Note app', () => {
  it('front page can be opened', () => {
    cy.visit('http://localhost:5173')
    cy.contains('Notes')
    cy.contains('Note app, Department of Computer Science, University of Helsinki
2023')
  })
})
```

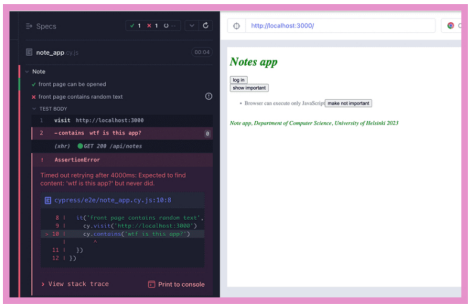
However, Mocha recommends that arrow functions are not used, because they might cause some issues in certain situations.

If *cy.contains* does not find the text it is searching for, the test does not pass. So if we extend our test like so

```
describe('Note app', function() {
  it('front page can be opened',  function() {
    cy.visit('http://localhost:5173')
    cy.contains('Notes')
    cy.contains('Note app, Department of Computer Science, University of Helsinki
2023')
  })

  it('front page contains random text', function() {
```

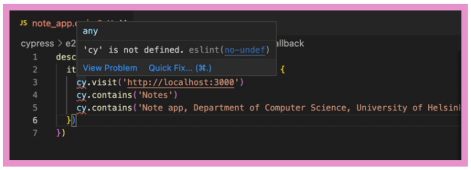
```
    cy.visit('http://localhost:5173')
    cy.contains('wtf is this app?')
  })
})
```

the test fails




Let's remove the failing code from the test.

The variable `cy` our tests use gives us a nasty Eslint error




We can get rid of it by installing eslint-plugin-cypress as a development dependency

```
npm install eslint-plugin-cypress --save-dev
```

and changing the configuration in *.eslintrc.cjs* like so:

```
module.exports = {
  "env": {
    browser: true,
    es2020: true,
    "jest/globals": true,
    "cypress/globals": true
  },
  "extends": [
    // ...
  ],
  "parserOptions": {
    // ...
  },
  "plugins": [
      "react", "jest", "cypress"
  ],
  "rules": {
    // ...
  }
}
```

## Writing to a form

Let's extend our tests so that the test tries to log in to our application. We assume our backend contains a user with the username *mluukkai* and password *salainen*.

The test begins by opening the login form.

```
describe('Note app',  function() {
  // ...

  it('login form can be opened', function() {
    cy.visit('http://localhost:5173')
    cy.contains('log in').click()
  })
})
```

The test first searches for the login button by its text and clicks the button with the command cy.click.

Both of our tests begin the same way, by opening the page *http://localhost:5173*, so we should separate the shared part into a *beforeEach* block run before each test:

```
describe('Note app', function() {
  beforeEach(function() {
    cy.visit('http://localhost:5173')
  })

  it('front page can be opened', function() {
    cy.contains('Notes')
    cy.contains('Note app, Department of Computer Science, University of Helsinki
2023')
  })

  it('login form can be opened', function() {
    cy.contains('log in').click()
  })
})
```

The login field contains two *input* fields, which the test should write into.

The cy.get command allows for searching elements by CSS selectors.

We can access the first and the last input field on the page, and write to them with the command cy.type like so:

```
it('user can login', function () {
  cy.contains('log in').click()
  cy.get('input:first').type('mluukkai')
  cy.get('input:last').type('salainen')
})
```

The test works. The problem is if we later add more input fields, the test will break because it expects the fields it needs to be the first and the last on the page.

It would be better to give our inputs unique *ids* and use those to find them. We change our

login form like so:

```
const LoginForm = ({ ... }) => {
  return (
    <div>
      <h2>Login</h2>
      <form onSubmit={handleSubmit}>
        <div>
          username
          <input
            id='username'
            value={username}
            onChange={handleUsernameChange}
          />
        </div>
        <div>
          password
          <input
            id='password'
            type="password"
            value={password}
            onChange={handlePasswordChange}
          />
        </div>
        <button id="login-button" type="submit">
          login
        </button>
      </form>
    </div>
  )
}
```

We also added an id to our submit button so we can access it in our tests.

The test becomes:

```
describe('Note app',  function() {
  // ..
  it('user can log in', function() {
    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')
    cy.get('#password').type('salainen')
    cy.get('#login-button').click()

    cy.contains('Matti Luukkainen logged in')
  })
})
```

The last row ensures that the login was successful.

Note that the CSS id-selector is #, so if we want to search for an element with the id *username* the CSS selector is *#username*.

Please note that passing the test at this stage requires that there is a user in the test database of the backend environment whose username is *mluukkai* and the password is *salainen*. Create a user if needed!

## Testing new note form

Let's next add test methods to test the "new note" functionality:

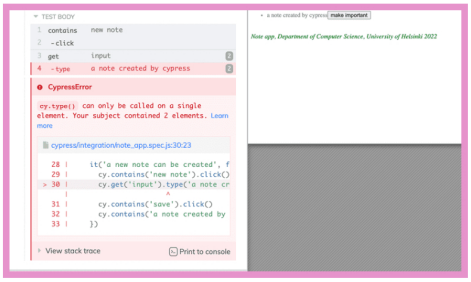
```
describe('Note app', function() {
  // ..
  describe('when logged in', function() {
    beforeEach(function() {
      cy.contains('log in').click()
      cy.get('input:first').type('mluukkai')
      cy.get('input:last').type('salainen')
      cy.get('#login-button').click()
    })

    it('a new note can be created', function() {
      cy.contains('new note').click()
      cy.get('input').type('a note created by cypress')
      cy.contains('save').click()
      cy.contains('a note created by cypress')
    })
  })
})
```

The test has been defined in its own *describe* block. Only logged-in users can create new notes, so we added logging in to the application to a *beforeEach* block.

The test trusts that when creating a new note the page contains only one input, so it searches for it like so:

```
cy.get('input')
```

If the page contained more inputs, the test would break




Due to this problem, it would again be better to give the input an *id* and search for the element by its id.

The structure of the tests looks like so:

```
describe('Note app', function() {
  // ...

  it('user can log in', function() {
    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')
    cy.get('#password').type('salainen')
    cy.get('#login-button').click()

    cy.contains('Matti Luukkainen logged in')
  })

  describe('when logged in', function() {
    beforeEach(function() {
      cy.contains('log in').click()
```

```
      cy.get('input:first').type('mluukkai')
      cy.get('input:last').type('salainen')
      cy.get('#login-button').click()
    })

    it('a new note can be created', function() {
      // ...
    })
  })
})
```

Cypress runs the tests in the order they are in the code. So first it runs *user can log in*, where the user logs in. Then cypress will run *a new note can be created* for which a *beforeEach* block logs in as well. Why do this? Isn't the user logged in after the first test? No, because *each* test starts from zero as far as the browser is concerned. All changes to the browser's state are reversed after each test.

## Controlling the state of the database

If the tests need to be able to modify the server's database, the situation immediately becomes more complicated. Ideally, the server's database should be the same each time we run the tests, so our tests can be reliably and easily repeatable.

As with unit and integration tests, with E2E tests it is best to empty the database and possibly format it before the tests are run. The challenge with E2E tests is that they do not have access to the database.

The solution is to create API endpoints for the backend tests. We can empty the database using these endpoints. Let's create a new router for the tests inside the *controllers* folder, in the *testing.js* file

```
const testingRouter = require('express').Router()
const Note = require('../models/note')
const User = require('../models/user')

testingRouter.post('/reset', async (request, response) => {
  await Note.deleteMany({})
  await User.deleteMany({})

  response.status(204).end()
})

module.exports = testingRouter
```

and add it to the backend only *if the application is run in test-mode*:

```
// ...

app.use('/api/login', loginRouter)
app.use('/api/users', usersRouter)
app.use('/api/notes', notesRouter)

if (process.env.NODE_ENV === 'test') {
  const testingRouter = require('./controllers/testing')
  app.use('/api/testing', testingRouter)
}

app.use(middleware.unknownEndpoint)
app.use(middleware.errorHandler)

module.exports = app
```

After the changes, an HTTP POST request to the */api/testing/reset* endpoint empties the database. Make sure your backend is running in test mode by starting it with this command (previously configured in the package.json file):

```
npm run start:test
```

The modified backend code can be found on the GitHub branch *part5-1*.

Next, we will change the *beforeEach* block so that it empties the server's database before tests are run.

Currently, it is not possible to add new users through the frontend's UI, so we add a new user to the backend from the beforeEach block.

```
describe('Note app', function() {
  beforeEach(function() {
    cy.request('POST', 'http://localhost:3001/api/testing/reset')
    const user = {
      name: 'Matti Luukkainen',
      username: 'mluukkai',
      password: 'salainen'
    }
    cy.request('POST', 'http://localhost:3001/api/users/', user)
    cy.visit('http://localhost:5173')
  })

  it('front page can be opened', function() {
    // ...
  })

  it('user can login', function() {
    // ...
  })

  describe('when logged in', function() {
    // ...
  })
})
```

During the formatting, the test does HTTP requests to the backend with cy.request.

Unlike earlier, now the testing starts with the backend in the same state every time. The backend will contain one user and no notes.

Let's add one more test for checking that we can change the importance of notes.

A while ago we changed the frontend so that a new note is important by default, or the *important* field is *true*:

```
const NoteForm = ({ createNote }) => {
  // ...

  const addNote = (event) => {
    event.preventDefault()
    createNote({
      content: newNote,
      important: true
    })

    setNewNote('')
  }
  // ...
}
```

There are multiple ways to test this. In the following example, we first search for a note and click its make not important button. Then we check that the note now contains a make

click its *make not important* button. Then we check that the note now contains a *make important* button.

```
describe('Note app', function() {                                    copy
  // ...

  describe('when logged in', function() {
    // ...

    describe('and a note exists', function () {
      beforeEach(function () {
        cy.contains('new note').click()
        cy.get('input').type('another note cypress')
        cy.contains('save').click()
      })

      it('it can be made not important', function () {
        cy.contains('another note cypress')
          .contains('make not important')
          .click()

        cy.contains('another note cypress')
          .contains('make important')
      })
    })
  })
})
```

The first command searches for a component containing the text *another note cypress*, and then for a *make not important* button within it. It then clicks the button.

The second command checks that the text on the button has changed to *make important*.

The tests and the current frontend code can be found on the GitHub branch *part5-9*.

## Failed login test

Let's make a test to ensure that a login attempt fails if the password is wrong.

Cypress will run all tests each time by default, and as the number of tests increases, it starts to become quite time-consuming. When developing a new test or when debugging a broken test, we can define the test with *it.only* instead of *it*, so that Cypress will only run the required test. When the test is working, we can remove *.only*.

First version of our tests is as follows:

```
describe('Note app', function() {                                    copy
  // ...

  it.only('login fails with wrong password', function() {
    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')
    cy.get('#password').type('wrong')
    cy.get('#login-button').click()

    cy.contains('wrong credentials')
  })

  // ...
})
```

The test uses cy.contains to ensure that the application prints an error message.

The application renders the error message to a component with the CSS class *error*:

```
const Notification = ({ message }) => {                              copy
  if (message === null) {
    return null
  }

  return (
    <div className="error">
      {message}
    </div>
  )
}
```

We could make the test ensure that the error message is rendered to the correct component, that is, the component with the CSS class *error*:

```
it('login fails with wrong password', function() {                   copy
  // ...

  cy.get('.error').contains('wrong credentials')
})
```

First, we use cy.get to search for a component with the CSS class *error*. Then we check that the error message can be found from this component. Note that the CSS class selector starts with a full stop, so the selector for the class *error* is *.error*.

We could do the same using the should syntax:

```
it('login fails with wrong password', function() {                   copy
  // ...

  cy.get('.error').should('contain', 'wrong credentials')
})
```

Using should is a bit trickier than using *contains*, but it allows for more diverse tests than *contains* which works based on text content only.

A list of the most common assertions which can be used with `should` can be found here.

We can, for example, make sure that the error message is red and it has a border:

```
it('login fails with wrong password', function() {                   copy
  // ...

  cy.get('.error').should('contain', 'wrong credentials')
  cy.get('.error').should('have.css', 'color', 'rgb(255, 0, 0)')
  cy.get('.error').should('have.css', 'border-style', 'solid')
})
```

Cypress requires the colors to be given as rgb.

Because all tests are for the same component we accessed using cy.get, we can chain them using and.

```
it('login fails with wrong password', function() {                   copy
  // ...

  cy.get('.error')
    .should('contain', 'wrong credentials')
    .and('have.css', 'color', 'rgb(255, 0, 0)')
    .and('have.css', 'border-style', 'solid')
})
```

Let's finish the test so that it also checks that the application does not render the success message *'Matti Luukkainen logged in'*:

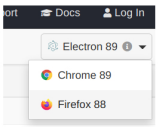
```
it('login fails with wrong password', function() {
  cy.contains('log in').click()
  cy.get('#username').type('mluukkai')
  cy.get('#password').type('wrong')
  cy.get('#login-button').click()

  cy.get('.error')
    .should('contain', 'wrong credentials')
    .and('have.css', 'color', 'rgb(255, 0, 0)')
    .and('have.css', 'border-style', 'solid')

  cy.get('html').should('not.contain', 'Matti Luukkainen logged in')
})
```

The command *should* is most often used by chaining it after the command *get* (or another similar command that can be chained). The *cy.get('html')* used in the test practically means the visible content of the entire application.

We would also check the same by chaining the command *contains* with the command *should* with a slightly different parameter:

```
cy.contains('Matti Luukkainen logged in').should('not.exist')
```

**NOTE:** Some CSS properties behave differently on Firefox. If you run the tests with Firefox:




then tests that involve, for example, `border-style`, `border-radius` and `padding`, will pass in Chrome or Electron, but fail in Firefox:


## Bypassing the UI

Currently, we have the following tests:

```
describe('Note app', function() {
  it('user can login', function() {
    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')
    cy.get('#password').type('salainen')
    cy.get('#login-button').click()

    cy.contains('Matti Luukkainen logged in')
  })

  it('login fails with wrong password', function() {
    // ...
  })

  describe('when logged in', function() {
    beforeEach(function() {
      cy.contains('log in').click()
      cy.get('input:first').type('mluukkai')
      cy.get('input:last').type('salainen')
      cy.get('#login-button').click()
    })

    it('a new note can be created', function() {
      // ...
    })

  })
})
```

First, we test logging in. Then, in their own describe block, we have a bunch of tests, which expect the user to be logged in. User is logged in in the *beforeEach* block.

As we said above, each test starts from zero! Tests do not start from the state where the previous tests ended.

The Cypress documentation gives us the following advice: Fully test the login flow – but only once. So instead of logging in a user using the form in the *beforeEach* block, Cypress recommends that we bypass the UI and do an HTTP request to the backend to log in. The reason for this is that logging in with an HTTP request is much faster than filling out a form.

Our situation is a bit more complicated than in the example in the Cypress documentation because when a user logs in, our application saves their details to the localStorage. However, Cypress can handle that as well. The code is the following

```
describe('when logged in', function() {
  beforeEach(function() {
    cy.request('POST', 'http://localhost:3001/api/login', {
      username: 'mluukkai', password: 'salainen'
    }).then(response => {
      localStorage.setItem('loggedNoteappUser', JSON.stringify(response.body))
      cy.visit('http://localhost:5173')
    })
  })

  it('a new note can be created', function() {
    // ...
  })

  // ...
})
```

We can access the response to a cy.request with the `then` method. Under the hood *cy.request*, like all Cypress commands, are promises. The callback function saves the details of a logged-in user to localStorage, and reloads the page. Now there is no difference to a user logging in with the login form.

If and when we write new tests to our application, we have to use the login code in multiple places. We should make it a custom command.

Custom commands are declared in *cypress/support/commands.js*. The code for logging in is as follows:

```
Cypress.Commands.add('login', ({ username, password }) => {
  cy.request('POST', 'http://localhost:3001/api/login', {
    username, password
  }).then(({ body }) => {
    localStorage.setItem('loggedNoteappUser', JSON.stringify(body))
    cy.visit('http://localhost:5173')
  })
})
```

Using our custom command is easy, and our test becomes cleaner:

```
describe('when logged in', function() {
  beforeEach(function() {
    cy.login({ username: 'mluukkai', password: 'salainen' })
  })

  it('a new note can be created', function() {
    // ...
  })

  // ...
})
```

The same applies to creating a new note now that we think about it. We have a test, which makes a new note using the form. We also make a new note in the *beforeEach* block of the test testing changing the importance of a note:

```
describe('Note app', function() {
  // ...

  describe('when logged in', function() {
    it('a new note can be created', function() {
      cy.contains('new note').click()
      cy.get('input').type('a note created by cypress')
      cy.contains('save').click()

      cy.contains('a note created by cypress')
    })

    describe('and a note exists', function () {
      beforeEach(function () {
        cy.contains('new note').click()
        cy.get('input').type('another note cypress')
        cy.contains('save').click()
      })

      it('it can be made important', function () {
        // ...
      })
    })
  })
})
```

Let's make a new custom command for making a new note. The command will make a new note with an HTTP POST request:

```
Cypress.Commands.add('createNote', ({ content, important }) => {
  cy.request({
    url: 'http://localhost:3001/api/notes',
    method: 'POST',
    body: { content, important },
    headers: {
      'Authorization': `Bearer
${JSON.parse(localStorage.getItem('loggedNoteappUser')).token}`
    }
  })

  cy.visit('http://localhost:5173')
})
```

The command expects the user to be logged in and the user's details to be saved to localStorage.

Now the formatting block becomes:

```
describe('Note app', function() {
  // ...

  describe('when logged in', function() {
    it('a new note can be created', function() {
      // ...
    })

    describe('and a note exists', function () {
      beforeEach(function () {
        cy.createNote({
          content: 'another note cypress',
          important: true
        })
      })

      it('it can be made important', function () {
        // ...
      })
    })
  })
})
```

There is one more annoying feature in our tests. The application address *http:localhost:5173* is hardcoded in many places.

Let's define the *baseUrl* for the application in the Cypress pre-generated configuration file *cypress.config.js*:

```
const { defineConfig } = require("cypress")

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {
    },
    baseUrl: 'http://localhost:5173'
  },
})
```

All the commands in the tests use the address of the application

```
cy.visit('http://localhost:5173' )
```

can be transformed into

```
cy.visit('')
```

The backend's hardcoded address *http://localhost:3001* is still in the tests. Cypress documentation recommends defining other addresses used by the tests as environment variables.

Let's expand the configuration file *cypress.config.js* as follows:

```
const { defineConfig } = require("cypress")

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {
    },
    baseUrl: 'http://localhost:5173',
    env: {
      BACKEND: 'http://localhost:3001/api'
    }
  },
})
```

Let's replace all the backend addresses from the tests in the following way

```
describe('Note ', function() {
  beforeEach(function() {

    cy.request('POST', `${Cypress.env('BACKEND')}/testing/reset`)
    const user = {
      name: 'Matti Luukkainen',
      username: 'mluukkai',
      password: 'secret'
    }
    cy.request('POST', `${Cypress.env('BACKEND')}/users`, user)
    cy.visit('')
  })
  // ...
})
```

The tests and the frontend code can be found on the GitHub branch *part5-10*.

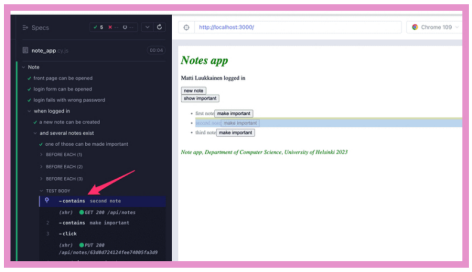
## Changing the importance of a note

Lastly, let's take a look at the test we did for changing the importance of a note. First, we'll change the formatting block so that it creates three notes instead of one:

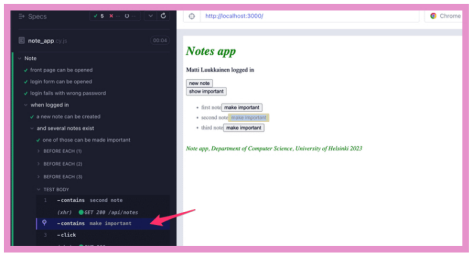
```
describe('when logged in', function() {
  describe('and several notes exist', function () {
    beforeEach(function () {
      cy.login({ username: 'mluukkai', password: 'salainen' })
      cy.createNote({ content: 'first note', important: false })
      cy.createNote({ content: 'second note', important: false })
      cy.createNote({ content: 'third note', important: false })
    })

    it('one of those can be made important', function () {
      cy.contains('second note')
        .contains('make important')
        .click()

      cy.contains('second note')
        .contains('make not important')
    })
  })
})
```

How does the cy.contains command actually work?

When we click the `cy.contains('second note')` command in Cypress Test Runner, we see that the command searches for the element containing the text *second note*:




By clicking the next line `.contains('make important')` we see that the test uses the 'make important' button corresponding to the *second note*:

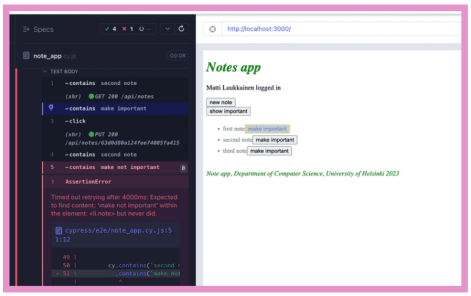



When chained, the second *contains* command *continues* the search from within the component found by the first command.

If we had not chained the commands, and instead write:

```
cy.contains('second note')
cy.contains('make important').click()
```

the result would have been entirely different. The second line of the test would click the button of a wrong note:

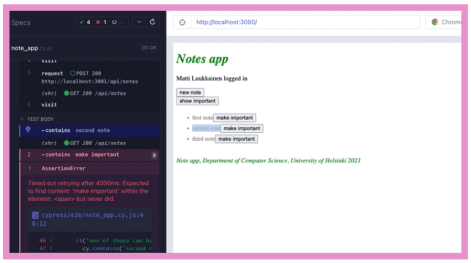



When coding tests, you should check in the test runner that the tests use the right components!

Let's change the `Note` component so that the text of the note is rendered to a *span*.

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important' : 'make important'

  return (
    <li className='note'>
      <span>{note.content}</span>
      <button onClick={toggleImportance}>{label}</button>
    </li>
  )
}
```

Our tests break! As the test runner reveals, `cy.contains('second note')` now returns the component containing the text, and the button is not in it.




One way to fix this is the following:

```
it('one of those can be made important', function () {
  cy.contains('second note').parent().find('button').click()
  cy.contains('second note').parent().find('button')
    .should('contain', 'make not important')
})
```

In the first line, we use the parent command to access the parent element of the element containing *second note* and find the button from within it. Then we click the button and check that the text on it changes.

Note that we use the command find to search for the button. We cannot use cy.get here, because it always searches from the *whole* page and would return all 5 buttons on the page.

Unfortunately, we have some copy-paste in the tests now, because the code for searching for the right button is always the same.

In these kinds of situations, it is possible to use the as command:

```
it('one of those can be made important', function () {
  cy.contains('second note').parent().find('button').as('theButton')
  cy.get('@theButton').click()
  cy.get('@theButton').should('contain', 'make not important')
})
```

Now the first line finds the right button and uses *as* to save it as *theButton*. The following lines can use the named element with *cy.get('@theButton')*.

## Running and debugging the tests

Finally, some notes on how Cypress works and debugging your tests.

The form of the Cypress tests gives the impression that the tests are normal JavaScript code, and we could for example try this:

```
const button = cy.contains('log in')
button.click()
debugger
cy.contains('logout').click()
```

This won't work, however. When Cypress runs a test, it adds each `cy` command to an execution queue. When the code of the test method has been executed, Cypress will execute each command in the queue one by one.

Cypress commands always return `undefined`, so `button.click()` in the above code would cause an error. An attempt to start the debugger would not stop the code between executing the commands, but before any commands have been executed.

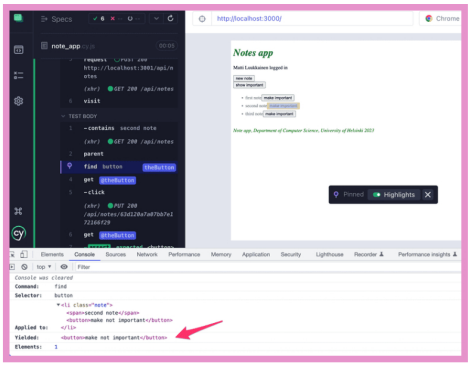
Cypress commands are *like promises*, so if we want to access their return values, we have to do it using the then command. For example, the following test would print the number of buttons in the application, and click the first button:

```
it('then example', function() {
  cy.get('button').then( buttons => {
    console.log('number of buttons', buttons.length)
    cy.wrap(buttons[0]).click()
  })
})
```

Stopping the test execution with the debugger is possible. The debugger starts only if Cypress test runner's developer console is open.

The developer console is all sorts of useful when debugging your tests. You can see the HTTP requests done by the tests on the Network tab, and the console tab will show you information

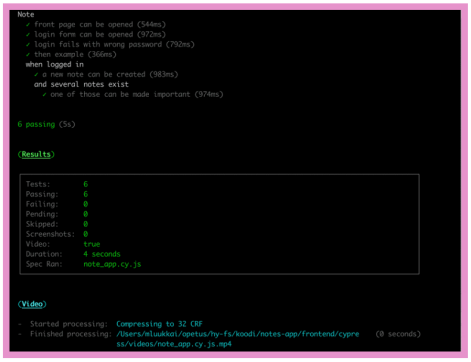
about your tests:




So far we have run our Cypress tests using the graphical test runner. It is also possible to run them from the command line. We just have to add an npm script for it:

```
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
    "server": "json-server -p3001 --watch db.json",
    "cypress:open": "cypress open",
    "test:e2e": "cypress run"
  },
```

Now we can run our tests from the command line with the command *npm run test:e2e*




Note that videos of the test execution will be saved to *cypress/videos/*, so you should probably git ignore this directory. It is also possible to turn off the making of videos.

The frontend and the test code can be found on the GitHub branch *part5-11*.

## Exercises 5.17.-5.23.

In the last exercises of this part, we will do some E2E tests for our blog application. The material of this part should be enough to complete the exercises. You **must check out the Cypress documentation**. It is probably the best documentation I have ever seen for an open-source project.

I especially recommend reading Introduction to Cypress, which states

> *This is the single most important guide for understanding how to test with Cypress. Read it. Understand it.*

### 5.17: bloglist end to end testing, step1

Configure Cypress for your project. Make a test for checking that the application displays the login form by default.

The structure of the test must be as follows:

```
describe('Blog app', function() {
  beforeEach(function() {
    cy.request('POST', 'http://localhost:3003/api/testing/reset')
    cy.visit('http://localhost:5173')
  })

  it('Login form is shown', function() {
    // ...
  })
})
```

The *beforeEach* formatting blog must empty the database using for example the method we used in the material.

### 5.18: bloglist end to end testing, step2

Make tests for logging in. Test both successful and unsuccessful login attempts. Make a new user in the *beforeEach* block for the tests.

The test structure extends like so:

```
describe('Blog app', function() {
  beforeEach(function() {
    cy.request('POST', 'http://localhost:3003/api/testing/reset')
    // create here a user to backend
    cy.visit('http://localhost:5173')
  })

  it('Login form is shown', function() {
    // ...
  })

  describe('Login',function() {
```

```
    it('succeeds with correct credentials', function() {
      // ...
    })

    it('fails with wrong credentials', function() {
      // ...
    })
  })
})
```

*Optional bonus exercise*: Check that the notification shown with unsuccessful login is displayed red.

#### 5.19: bloglist end to end testing, step3

Make a test that verifies a logged-in user can create a new blog. The structure of the test could be as follows:

```
describe('Blog app', function() {                                    copy
  // ...

  describe('When logged in', function() {
    beforeEach(function() {
      // log in user here
    })

    it('A blog can be created', function() {
      // ...
    })
  })

})
```

The test has to ensure that a new blog is added to the list of all blogs.

#### 5.20: bloglist end to end testing, step4

Make a test that confirms users can like a blog.

#### 5.21: bloglist end to end testing, step5

Make a test for ensuring that the user who created a blog can delete it.

#### 5.22: bloglist end to end testing, step6

Make a test for ensuring that only the creator can see the delete button of a blog, not anyone else.

#### 5.23: bloglist end to end testing, step7

Make a test that checks that the blogs are ordered according to likes with the blog with the most likes being first.

*This exercise is quite a bit trickier than the previous ones.* One solution is to add a certain class for the element which wraps the blog's content and use the eq method to get the blog element in a specific index:

```
cy.get('.blog').eq(0).should('contain', 'The title with the most likes')      copy
cy.get('.blog').eq(1).should('contain', 'The title with the second most likes')
```

Note that you might end up having problems if you click a like button many times in a row. It might be that cypress does the clicking so fast that it does not have time to update the app state in between the clicks. One remedy for this is to wait for the number of likes to update in between all clicks.

This was the last exercise of this part, and it's time to push your code to GitHub and mark the exercises you completed in the exercise submission system.