Chciałym poruszyć temat testów end to end z wykorzystaniem Cypressa. Zacznijmy od tego po co piszemy testy dla naszych aplikacji? Bo chcemy by nasza aplikacja:

- Była łatwiejsza do rozbudowy przy dalszym procesie developmentu

- Miała możliwość monitorowania wszystkich funkcjonalności na bieżąco

- Spełniała oczekiwania użytkownika

Gdy w procesie developmentu na skutek nowych funkcjonalności coś innego nie będzie spełniało swoich funkcjonalności, my zostaniemy o tym poimformowani. Niestety nie dostaniemy konkretnej linijki błędu, ale sama informacja że coś nie działa poprawnie jest już sporym plusem. Testy end to end pomijają zupełnie jakość kodu, skupiają się wyłącznie na poprawnym działaniu aplikacji. Przechodząc już do samego narzędzia z którym będziemy pracować przy pisaniu testów end to end... jest nim Cypress.(ZDJĘCIE 2.png) By szybko dodać go do naszego projektu za pomocą "yarn" wystarczy najpierw go dodać (najlepiej jako --dev) "yarn add cypress --dev", a następnie odpalić już samego cypress'a "yarn run cypress open". (ZDJĘCIE 3.png) Po tych 2 komendach otworzy nam się okno Cypress'a w którym mamy możliwość podejrzenia naszych plików testowych, logowania oraz najważniejsze czyli ustawień przeglądarki na której będziemy odpalać nasze testy. Wrócimy do samego tematu obsługi okna później,(ZDJĘCIE 4.png) bo teraz potrzebujemy omówić strukture folderów która nam się automatycznie wygenerowała przy pierwszym odpaleniu Cypress'a. W folderze "cypress" znajdziemy folder o nazwie integration i tam będziemy dodawać większośćnaszych testów. Na start dostajemy sporo plików przykładowych z których mocno polecam korzystać. Wszystkie pliki testowe powinniśmy kończyć z "spec.js", by Cypress je poprawnie rozpoznał. Zacznijmy nasz pierwszy test. Cypress korzysta z syntaxu "Mokka and Chai". Jeśli ktoś miał dawniej styczność z testami jednostkowymi albo integracyjnymi to powinien się szybko odnaleźć. W "describe" opisujemy ogół funkcjonalności albo modułu który chcemy testować, a w samym "it" piszemy już pełną nazwe funkcjonalności jaką chcemy, by się dokonała w tym teście. "cy" na początku zobaczycie jeszcze wiele razy, jest to odniesienie do Cypress'a i dzięki niemu mamy dostęp do przydatnych rzeczy. Jak np. "cy.visit()" które daje nam redirect na konkretny url naszej aplikacji. Tutaj pojawiają się spore możliwości testów end to end, które mogą zawierać praktycznie całe potężne ścieżki które najczęściej wykonuje typowy użytkownik. Zaczynając od Rejestracji aż do samej ścieżki zakupowej. Wracając jeszcze do "cy.visit()" jest ono obowiązkowe na każdym "it" bo przecież musimy mieć jakiś początkowy route. Następnie "cy.contains" sprawdza czy na stronie znajduje się dalej podany string.(ZDJĘCIE 5.png) Wracając do naszego okna Cypress'a i klikając na nasz plik otworzy nam się dodatkowe okno przeglądarki w którym zobaczymy jak Cypress radzi sobie z naszym testem. Po lewej stronie będziemy widzieć konkretne testy i ich szczegóły typu: co się wykonuje aktualnie czy nazwe testu. Powyżej mamy licznik czasu który zlicza łączną ilość. Niestety testy end to end nie są tak szybkie jak jednostkowe czy integracyjne, ale możemy polepszyć ich wydajność o czym powiem później. Po prawej stronie widzimy naszą aplikację która na bieżąco jest sprawdzana przez Cypress'a. Przejdzmy do czegoś poważniejszego.(ZDJĘCIE 6.png) Stworzyłem prosty komponent który posiada formularz logowania oraz przepuszcza tylko z odpowiednim loginem i passwordem, a przy nieudanej próbie wyświetla nam komunikat o błędzie, a jeśli wszystko zostało poprawnie wpisane to dostajemy redirect na "/myProfile".(ZDJĘCIE 7.png) Na starcie możecie zauważyć, że przeniosłem "cy.visit()" do beforeEach, dzięki temu nie musimy pisać wiele razy tego samego kodu na starcie... DRY trzeba szanować :) Pierwszy test sprawdza czy na naszej stronie znajduje się komponent logowania. Drugi test sprawdza czy użytkownik po wprowadzeniu złych danych dostanie informacje o błędzie. Tutaj korzystamy z "cy.get()" które służy do łapania za konkretne elementy DOM, a następnie ".type()" które pozwala pisać w inputach, a na samym końcu robimy ".click()", by zatwierdzić formularz i sprawdzić co się stanie. Ostatnią linijką jest "cy.contains()" bo chcemy sprawdzić czy po wpisaniu błędnych danych dostaniemy komunikat o tym. W trzecim teście robimy bardzo podobną rzecz tyle, że tym razem podajemy poprawne login i password, a na końcu sprawdzamy czy page zawiera "Welcome" które nie znajduje się na stronie logowania, a dopiero po poprawnym zalogowaniu i redirect.(ZDJĘCIE 8.png) Przejdzmy do czegoś trudniejszego czyli testowania HTTP requests. Na podstronie /myProfile na "created()" life cycle hooku komponentu robimy zapytanie asynchroniczne, gdzie pobieram z jsonplaceholder przykładowe "toDo", a następnie wyświetlam je na ekranie. W pierwszym teście sprawdzamy standardowo czy strona zawiera informacje po zapytaniu asynchronicznym. Takowy test jest poprawny, bo Cypress radzi sobie dobrze z takimi zapytaniami asynchronicznymi, ale nie jest ono bardzo wydajne ponieważ musimy czekać na zwrotke, a jeśli przemnożmy ilości takich requestów w naszych testach to niestety ale będą się one wykonwać kilka dobrych minut. Dlatego stosujemy coś takiego jak mamy w drugim teście, czyli "cy.server()" oraz "cy.route()" które mockują nam konkretnego request'a i nie musimy czekać na zwrotke tylko mamy ją natychmiast. Wykorzystałem także "cy.fixute()" które odnosi się do folderu "fixturex" który wygenerował nam Cypress na starcie. Tam możmy dodać pliki w formacie .json, a następnie odnieść się do nich jako do zwrotki z HTTP request'a którego mockujemy. Dzięki temu wszystkiemu mamy bardzo podobny test jak ten powyżej, ale ten jest znacznie wydajniejszy i mocno podnosi jakość naszego kodu. Zachęcam mocno do sprawdzenia oficjalnej strony Cypress'a: <https://www.cypress.io/> , oraz jeśli chcesz sprawdzić całość kodu jaki omowaliśmy powyżej to zapraszam na: