O'REILLY®

Testowanie full stack

Praktyczny przewodnik dostarczania oprogramowania wysokiej jakości

Helion

Gayathri Mohan

Tytuł oryginału: Full Stack Testing: A Practical Guide for Delivering High Quality Software

Tłumaczenie: Radosław Meryk

ISBN: 978-83-8322-015-4

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/tefust
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/tefust.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 2.

Ręczne testowanie eksploracyjne

> Nie wszyscy, którzy wędrują, są zgubieni.
>
> — *J.R.R. Tolkien*

Ręczne testowanie eksploracyjne to intensywne działania polegające na próbowaniu testowanej aplikacji. Mają one na celu zbadanie i zrozumienie jej zachowania w różnych sytuacjach, które nigdzie nie są wyraźnie wyartykułowane — ani w dokumencie wymagań, ani w historyjkach użytkownika. W wyniku eksploracji często wykrywane są nowe przepływy pracy użytkowników, takie, które nie były przewidziane podczas fazy analizy lub programowania. Wykrywane są również błędy w istniejących przepływach użytkowników. Takie odkrycia przynoszą testerowi wiele satysfakcji, ponieważ udowadniają jego wysokie umiejętności analityczne i zdolność do wnikliwej obserwacji!

Zazwyczaj ręczne testy eksploracyjne są przeprowadzane w środowisku testowym, w którym jest wdrażana cała aplikacja. Testerzy, w celu symulowania różnych scenariuszy i obserwowania zachowania aplikacji, mogą swobodnie ingerować w czasie rzeczywistym w różne komponenty aplikacji, takie jak baza danych czy usługi lub procesy działające w tle. Ten eksploracyjny styl testowania różni się od tradycyjnego testowania ręcznego, które odnosi się do ręcznego wykonywania określonego zbioru działań opisanych w historyjkach użytkownika lub w dokumencie wymagań jako kryteria akceptacji, aby zweryfikować, czy podane oczekiwania zostały pomyślnie spełnione. Mówiąc inaczej, w testach ręcznych niekoniecznie są wymagane jakiekolwiek umiejętności analityczne, podczas gdy w testach eksploracyjnych testerzy otrzymują zielone światło do wykraczania poza to, co zostało udokumentowane, a nawet poza to, co do tej pory o aplikacji *wiadomo*!

Ze względu na nakładanie się na siebie testów ręcznych z eksploracyjnymi niektóre zespoły, nawet dzisiaj, nie doceniają wartości tych ostatnich. Często istnieje również przekonanie, że nakład pracy związanej z analizą przeprowadzaną w ramach przygotowania i programowania historyjki użytkownika jest wystarczający, zwłaszcza gdy analiza ta jest uzupełniona o testy automatyczne (testy automatyczne szczegółowo omówiono w rozdziale 3.). Przekonanie to pomija jednak fakt, że analiza przeprowadzana podczas tworzenia historyjki użytkownika jest zwykle wykonywana przede wszystkim z punktu widzenia biznesowego, a podczas programowania członkowie zespołu skupiają się na aktualnym zakresie funkcjonalności i ograniczają swoje myślenie do tego wąskiego zakresu.

Takie podejście pozostawia oczywistą lukę, ponieważ aplikacja nie jest testowana z perspektywy docelowego użytkownika oraz działania w środowisku produkcyjnym. Ta luka może spowodować problemy z integracją oraz pominięcie niektórych przepływów pracy docelowych użytkowników — dlatego po zaprogramowaniu aplikacji zespoły potrzebują ręcznej fazy testów eksploracyjnych.

Testy eksploracyjne łączą wszystkie trzy aspekty — wymagania biznesowe, techniczne szczegóły implementacji i potrzeby docelowych użytkowników — i rzucają wyzwanie wszystkiemu, co jest uważane za prawdę z tych wszystkich punktów widzenia. Dobrą praktyką jest wywoływanie kompletnych funkcjonalności dopiero po zautomatyzowaniu nowych przepływów użytkowników oraz przypadków testowych wykrytych podczas testów eksploracyjnych.

Przydzielanie oddzielnej osoby do przeprowadzenia takiego testowania eksploracyjnego po zakończeniu programowania może nie być konieczne, chociaż dzięki wyznaczeniu takiej osoby możliwe jest osiągnięcie lepszych wyników z perspektywy gromadzenia wiedzy o aplikacjach, ponieważ zadanie to wymaga osoby z umiejętnościami wnikliwej obserwacji i analizy. Jeśli wyznaczenie oddzielnej osoby odpowiedzialnej za testowanie eksploracyjne uniemożliwiają problemy z kosztami lub dostępnością, odpowiedzialność za przeprowadzanie testów eksploracyjnych podczas każdej iteracji powinni brać na siebie, po kolei, istniejący członkowie zespołu. Rozwijanie umiejętności testowania eksploracyjnego może pomóc wszystkim członkom zespołu, niezależnie od spełnianej roli, w skuteczniejszym działaniu.

Jeśli jesteś jednym z takich członków zespołu, który chce rozwinąć swoje umiejętności testowania eksploracyjnego, ten rozdział jest dla Ciebie. Omówię w nim istniejące w branży schematy postępowania, które mogą pomóc w testach eksploracyjnych, oraz strategie podejścia do tego zadania. Ćwiczenia zamieszczone w tym rozdziale koncentrują się w szczególności na przeprowadzaniu testów eksploracyjnych webowych interfejsów użytkownika oraz interfejsów API. Zbadany zostanie również zbiór przydatnych praktyk mających na celu utrzymanie higieny środowiska testowego, ponieważ w powodzeniu ręcznych testów eksploracyjnych kluczową rolę odgrywa w pełni wdrożone środowisko testowe.

Najczęściej używane terminy

Poniżej przedstawiono listę niektórych często używanych terminów, które można spotkać w tym rozdziale:

- **Funkcja** (ang. *feature*) lub **funkcjonalność** to sposób, w jaki aplikacja zapewnia wartość swoim docelowym użytkownikom. Na przykład logowanie to funkcja, która zapewnia bezpieczeństwo docelowym użytkownikom.
- **Przepływ użytkownika** (ang. *user flow*) to zestaw działań wykonywanych w aplikacji przez docelowego użytkownika w celu osiągnięcia wartości zapewnianej przez funkcjonalność. Na przykład aby się zalogować, użytkownik docelowy musi wprowadzić swoje poświadczenia i zalogować się — jest to przepływ logowania użytkownika.

- **Przypadek testowy** (ang. *test case*) to zbiór działań sprawdzających, czy funkcja działa zgodnie z oczekiwaniami. Na przykład przypadkiem testowym jest wprowadzenie prawidłowej nazwy użytkownika i hasła oraz sprawdzenie, czy logowanie się powiodło. Podobnie przypadkiem testowym jest również wprowadzenie nieprawidłowej nazwy użytkownika i sprawdzenie, czy aplikacja wyświetli komunikat o błędzie. Pierwszy z nich jest pozytywnym przypadkiem testowym, ponieważ pozwala docelowemu użytkownikowi pomyślnie osiągnąć wartość zapewnianą przez funkcjonalność, podczas gdy drugi jest negatywnym przypadkiem testowym, ponieważ nie pozwala na osiągnięcie tej wartości. Aby w pełni zbadać funkcjonalność, trzeba zasymulować i zaobserwować zarówno pozytywne, jak i negatywne przypadki testowe.
- **Przypadek brzegowy** (ang. *edge case*) to negatywny przypadek testowy, który występuje bardzo rzadko.

Bloki konstrukcyjne

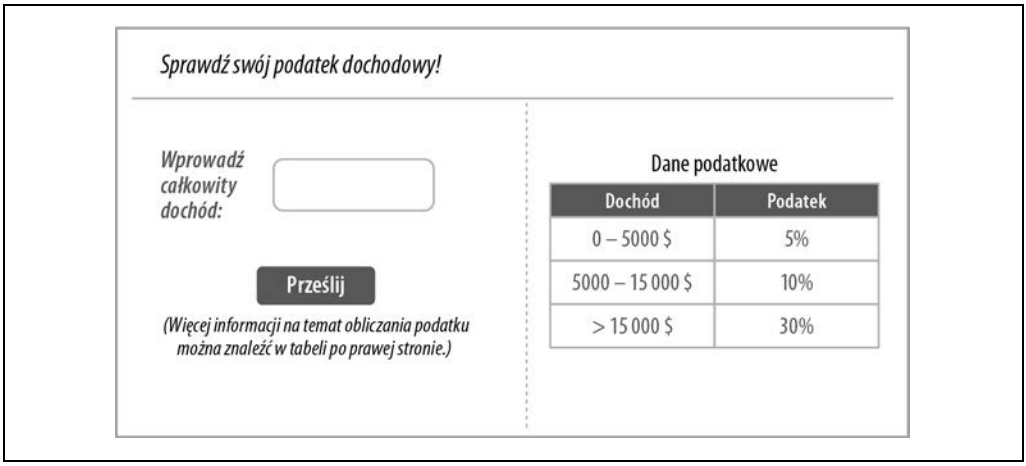
Zacznijmy od przyjrzenia się ośmiu schematom działania w testach eksploracyjnych, wraz z praktycznymi przykładami ich użycia. W dalszej części rozdziału przećwiczymy odkrywanie funkcjonalności.

Schematy stosowane w testach eksploracyjnych

Celem schematów stosowanych w testach eksploracyjnych jest pomoc w tworzeniu modeli myślowych, które można intuicyjnie zastosować do odpowiednich kontekstów w aplikacji. Mają one na celu zawężenie zakresu testów poprzez nadanie określonej funkcjonalności czytelności i struktury. Na przykład w aplikacjach powszechnie występują pola do wprowadzania liczb. Zamiast losowo testować wszystkie możliwe wartości liczbowe w celu sprawdzenia takiego pola, schematy działania udostępniają struktury pozwalające na logiczny podział danych wejściowych na zbiory próbek. Istnieją również schematy działania, które próbują ustrukturyzować reguły biznesowe, a tym samym pomagają zaobserwować różne przepływy pracy użytkowników oraz różne przypadki testowe. Spróbujmy zagłębić się w nie jeden po drugim, wraz z odpowiednimi przykładami.

Jako pierwszy przykład weźmy stronę internetową, która — jak widać na rysunku 2.1 — prosi użytkownika o wprowadzenie jego dochodów i wyświetla jako wynik kwotę należnego podatku. Na rysunku 2.1, po jego prawej stronie, zostały również pokazane różne przedziały podatkowe używane do obliczeń.

Aby przetestować, czy logika obliczania podatku działa zgodnie z oczekiwaniami, trzeba zidentyfikować pozytywne i negatywne przypadki testowe i wypróbować je jako dane wejściowe. Należy zauważyć, że dochód jest ciągłą wartością liczbową w zakresie od 0 do nieskończoności. Aby uzyskać pozytywne i negatywne przypadki testowe, należy logicznie zawęzić odpowiedni zestaw liczbowych wartości wejściowych i zweryfikować zwracane wyniki. Istnieją dwa schematy, które mogą nam w tym pomóc: podział klas równoważności i analiza wartości granicznych.



Rysunek 2.1. Przykład prostego kalkulatora podatkowego

Podział klasy równoważności

Mechanizm podziału klas równoważności sugeruje podzielenie danych wejściowych, które dają ten sam wynik lub podlegają podobnemu przetwarzaniu, na klasy. Aby całkowicie przetestować funkcjonalność, wystarczy wybrać z każdej klasy tylko jedną przykładową daną wejściową.

Z zastosowaniem tej sugestii do przykładu kalkulatora podatkowego, pierwszym zestawem klas podziału będą progi podatkowe: [0 – 5000], [5001 – 15 000] oraz [>15 000]. Te trzy klasy można uznać za **klasy równoważności**, ponieważ każda dana wejściowa w obrębie każdej z klas podlega tym samym regułom, a zweryfikowanie pozytywnych przypadków testowych wymaga przetestowania aplikacji z trzema punktami danych wejściowych — po jednym z każdej klasy. Na przykład do potwierdzenia pozytywnych przypadków testowych wystarczające będzie przetestowanie aplikacji dla wartości danych wejściowych 2000, 10 000 i 20 000. Ten sam schemat można zastosować również do wyznaczenia negatywnych przypadków testowych. Klasy danych wejściowych, które powinny spowodować błąd, to [*wartości ujemne*], [*litery*], [*symbole*] i tak dalej. Do przetestowania negatywnych przypadków testowych również wystarczy jedna wartość z każdej klasy.

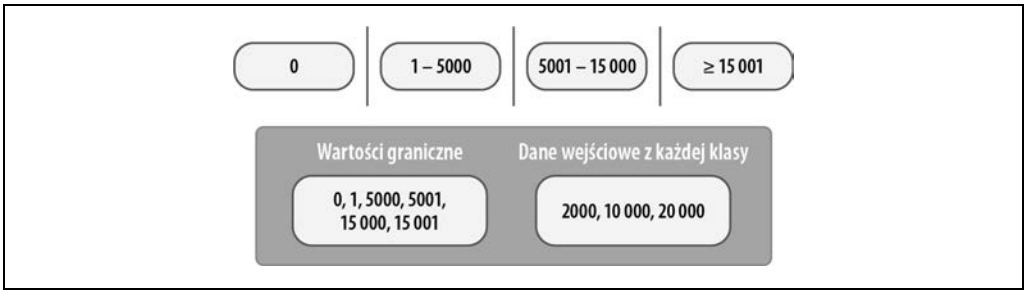
Oprócz ręcznych testów eksploracyjnych ten mechanizm jest pomocny w testach jednostkowych (które zostały omówione w rozdziale 3.). Można go również zastosować do dowolnego innego istotnego kontekstu w aplikacji, takiego jak testowanie wyników opartych na czasie (przed i po zdarzeniu), wewnętrznych stanów systemu i tak dalej.

Analiza wartości granicznych

Analiza wartości granicznych jest rozszerzeniem metody podziału klas równoważności. Polega na jawnym sprawdzaniu warunków brzegowych w każdej z klas. Jest to pomocne w znajdowaniu błędów, ponieważ warunki brzegowe są często niejasno zdefiniowane i niewłaściwie zaimplementowane. Na przykład w naszym przykładzie prostego kalkulatora podatkowego wymagania dotyczące przedziałów podatkowych mogły zostać określone jako „5% podatku dla dochodu poniżej 5000 USD, 10% dla dochodu między 5000 USD, a 15 000 USD i 30% podatku dla dochodu powyżej 15 000 USD”.

Nie pozwala to jednak jasno określić warunków brzegowych; tzn. do których klas powinny należeć wartości 5000 i 15 000. Mechanizm analizy wartości granicznych zwraca uwagę na takie problemy i pomaga je rozwiązać poprzez testowanie wartości granicznych z każdej klasy równoważności, a także wybieranie danych wejściowych wewnątrz zakresu klasy.

Zastosujmy ten mechanizm do przykładu kalkulatora podatkowego. Spróbujmy przeanalizować wartości graniczne w każdej z wyznaczonych wcześniej klas równoważności. W pierwszej z klas [0 – 5000] wartościami granicznymi są 0 i 5000. Jednak logiczne jest, że gdy dochód wynosi 0, nie powinno być żadnych podatków, więc można utworzyć nowe klasy równoważności: [0] i [1 – 5000]. Zatem wartości graniczne, które należy przetestować, aby objąć wszystkie pozytywne przypadki testowe, to [0, 1, 5000, 5001, 15 000, 15 001] — patrz rysunek 2.2.



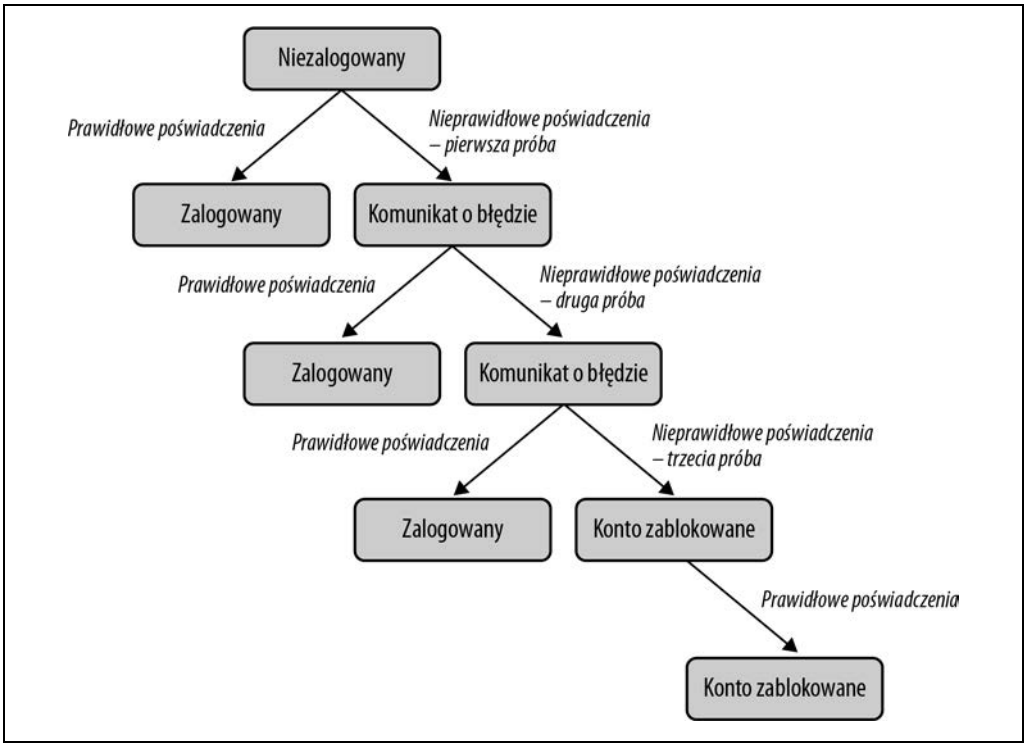
Rysunek 2.2. Klasy równoważności z warunkami brzegowymi

Jak pokazuje ten przykład, chociaż jest to mechanizm testowania, przynosi maksymalne korzyści zespołowi, gdy jest stosowany we wszystkich fazach dostarczania oprogramowania, począwszy od analizy.

Po omówieniu dwóch schematów, które pomagają ustrukturyzować wartości danych wejściowych pojedynczego pola, i przetłumaczenia ich na minimalny zestaw pozytywnych i negatywnych przypadków testowych, nadszedł czas, aby przejść do schematów dotyczących nieco bardziej złożonych scenariuszy — takich, w których wiele kombinacji danych wejściowych powoduje zwracanie różnych wyników. Aby pomóc nam w tej analizie, posłużę się klasycznym przykładem strony logowania przyjmującej dwie dane wejściowe: adres e-mail i hasło. Na podstawie tego przykładu omówię, w jaki sposób schematy: przejścia stanu, tabele decyzyjne i przyczyna-skutek pomagają w wizualizacji różnych przypadków testowych.

Przejścia stanu

Schemat przejść stanu pomaga w wyznaczaniu przypadków testowych w sytuacjach, w których zachowanie aplikacji zmienia się na podstawie historii danych wejściowych. Na przykład za pierwszym i drugim razem, gdy użytkownik wprowadzi nieprawidłowe hasło, strona logowania może wyświetlać komunikat o błędzie, ale przy trzeciej nieudanej próbie konto jest blokowane. W takich scenariuszach, aby określić przypadki testowe, można narysować drzewo przejść stanu (rysunek 2.3). W drzewie przejść każdy stan aplikacji jest przedstawiony jako węzeł. Możliwe wyniki działań są wyświetlane jako węzły potomne, a działania (zdarzenia) wyzwalające określone wyniki są oznaczone jako etykiety gałęzi.



Rysunek 2.3. Drzewo przejść stanu dla scenariusza nieprawidłowego logowania

To drzewo daje jasny obraz każdego przypadku testowego, wraz z jego stanem początkowym, działaniem, które zmienia stan aplikacji, oraz oczekiwanymi wynikami do sprawdzenia. Wizualizacja daje również realistyczne oszacowanie nakładu pracy wymaganego do przetestowania funkcji, ponieważ objaśnia liczbę stanów i przejść, co pomaga w fazie planowania.

Przejścia stanu mogą być znacznie bardziej skomplikowane — na przykład w systemie zarządzania zamówieniami, w którym zamówienia przechodzą przez stany takie jak płatność zakończona, oczekujące, wysłane, anulowane, zrealizowane i tak dalej. W takich przypadkach wizualizacja każdego stanu jako węzła i działań wprowadzających zamówienie do każdego możliwego następnego stanu dają jasny przegląd funkcjonalności.

Tabele decyzyjne

Gdy dane wejściowe generują wyniki poprzez ich powiązania logiczne (`AND`, `OR` itp.), do wyznaczania przypadków testowych mogą być użyte tabele decyzyjne. Pozwala to zaoszczędzić dużo czasu podczas testowania, ponieważ wszystkie możliwe kombinacje danych wejściowych i oczekiwanych wyników są z wyprzedzeniem wyraźnie wymienione w tabeli. W przykładzie logowania adres e-mail i hasło są logicznie powiązane operatorem `AND`. Oznacza to, że pomyślne zalogowania wymaga podania zarówno prawidłowego adresu e-mail, jak i hasła. Tabelę decyzyjną możliwą do utworzenia dla tego scenariusza przedstawiono w tabeli 2.1.

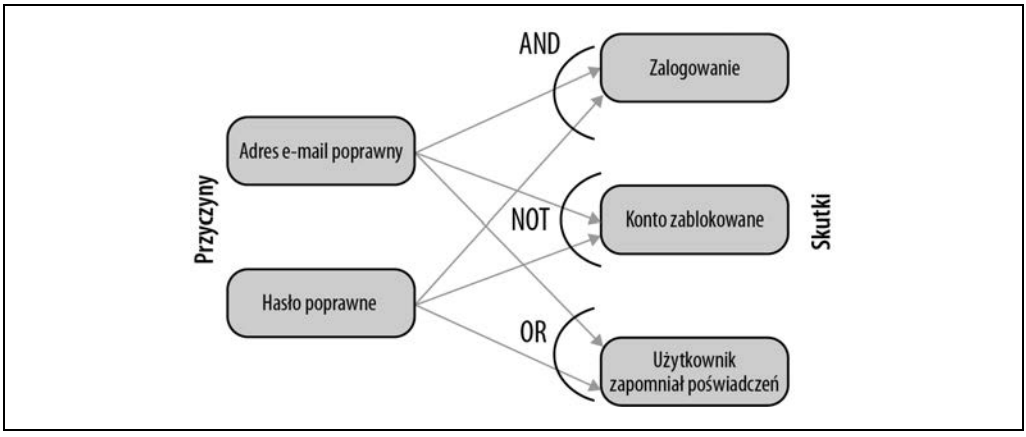
Tabela 2.1. Tabela decyzyjna dla scenariusza logowania

Tabela decyzyjna		Przypadek testowy 1	Przypadek testowy 2	Przypadek testowy 3	Przypadek testowy 4
Warunki	E-mail	Prawda	Fałsz	Fałsz	Prawda
	Hasło	Fałsz	Prawda	Fałsz	Prawda
Działania	Zalogowanie	-	-	-	Prawda
	Komunikat o błędzie	Prawda	Prawda	Prawda	-

Dzięki tej metodzie można również zaoszczędzić czas, ponieważ niektóre niepotrzebne przypadki testowe można wyeliminować. Na przykład w scenariuszu logowania można wyeliminować przypadek testowy 3, w którym obie dane wejściowe są niepoprawne, ponieważ jeśli nawet jedno z danych wejściowych będzie nieprawidłowe, logowanie się nie powiedzie.

Grafy przyczynowo-skutkowe

Grafy przyczynowo-skutkowe to kolejny sposób wizualizacji logicznie powiązanych danych wejściowych i odpowiadających im możliwych wyników. Schemat pomaga zobaczyć szeroką perspektywę funkcjonalności, a zatem jest szczególnie przydatny w fazie analizy. Po utworzeniu grafu, aby uzyskać szczegółowe przypadki testowe, możesz przetłumaczyć go na tabelę decyzyjną. Graf przyczynowo-skutkowy dla zaprezentowanego wcześniej przykładu logowania został pokazany na rysunku 2.4.



Rysunek 2.4. Grafy przyczynowo-skutkowe dla scenariusza logowania

Z jednej strony zostały wymienione przyczyny, a z drugiej skutki. Ścieżki nawigacji między nimi zostały ułożone za pomocą operatorów logicznych.

Schematy postępowania, które opisałam do tej pory, przydają się do strukturyzowania powiązanych ze sobą danych wejściowych. W dalszej części rozdziału opiszę dwa schematy, które mogą być pomocne w sytuacji, gdy mamy do czynienia z wieloma niezależnymi zmiennymi i dużymi zbiorami danych.

Testowanie parami

W aplikacjach często mamy do czynienia z więcej niż jedną daną wejściową, a zarządzanie odmianami danych wejściowych i wnioskowanie przypadków testowych może sprawiać trudności. Testowanie parami, znane również jako **testowanie all-pair** (dosłownie: testowanie wszystkich par), to schemat, który w sytuacji, kiedy na wyniki wpływa wiele niezależnych zmiennych (danych wejściowych), pomaga w skondensowaniu przypadków testowych do minimum. Aby zilustrować, jak działa ten schemat, spróbujmy wykonać krótkie ćwiczenie.

Rozważmy formularz, w którym są wprowadzane trzy niezależne dane wejściowe: typ systemu operacyjnego, producent urządzenia i rozdzielczość. Pole systemu operacyjnego może przyjmować dwie wartości: *Android* lub *Windows*. Pole urządzenia może przyjmować trzy wartości: *Samsung*, *Google* lub *Oppo*. Na koniec pole rozdzielczości może przyjmować wartości: *Niska*, *Średnia* i *Wysoka*. Podczas testowania tego formularza, jak widać w tabeli 2.2, mamy więc $2 \times 3 \times 3 = 18$ kombinacji wejściowych.

Tabela 2.2. Przykładowe przypadki testowe bez zastosowania metody testowania parami

Przypadki testowe	Urządzenie	Rozdzielczość	System operacyjny
1	Samsung	Niska	Android
2	Samsung	Średnia	Android
3	Samsung	Wysoka	Android
4	Google	Niska	Android
5	Google	Średnia	Android
6	Google	Wysoka	Android
7	Oppo	Niska	Android
8	Oppo	Średnia	Android
9	Oppo	Wysoka	Android
10	Samsung	Niska	Windows
11	Samsung	Średnia	Windows
12	Samsung	Wysoka	Windows
13	Google	Niska	Windows
14	Google	Średnia	Windows
15	Google	Wysoka	Windows
16	Oppo	Niska	Windows
17	Oppo	Średnia	Windows
18	Oppo	Wysoka	Windows

Schemat testowania parami sugeruje, że dowolną parę wejść wystarczy przetestować raz, gdyż są to zmienne niezależne. Jak widać w tabeli 2.3, powoduje to skrócenie listy przypadków testowych do zaledwie dziewięciu.

Nowa, skondensowana tabela pozwoliła wyeliminować powtórzenia kilku par. Na przykład pary [*Google*, *Średnia*] i [*Google*, *Windows*] występują teraz tylko raz.

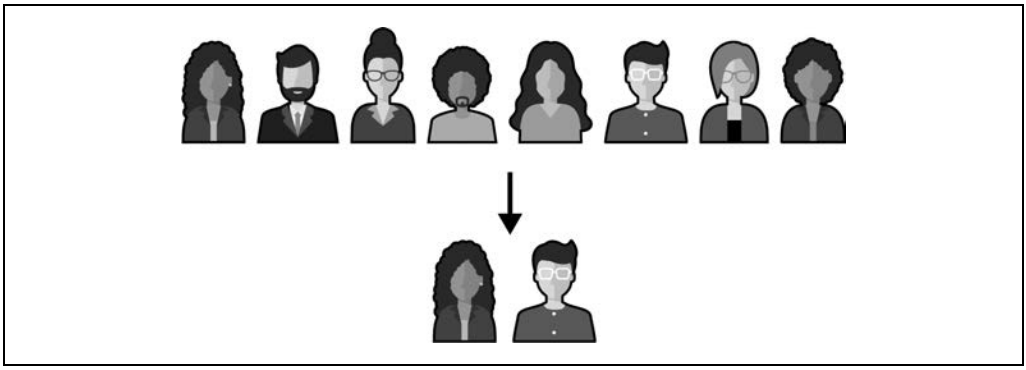
Tabela 2.3. Zredukowana liczba przypadków testowych w przypadku zastosowania metody testowania parami

Przypadki testowe	Urządzenie	Rozdzielczość	System operacyjny
1	Oppo	Niska	Android
2	Samsung	Niska	Windows
3	Google	Niska	Android
4	Oppo	Średnia	Windows
5	Samsung	Średnia	Android
6	Google	Średnia	Windows
7	Oppo	Wysoka	Android
8	Samsung	Wysoka	Windows
9	Google	Wysoka	Android/Windows

Próbkowanie

Do tej pory mieliśmy do czynienia z niezbyt obszernymi danymi wejściowymi. Takie dane można sobie łatwo wyobrazić bez pomocy narzędzi. Co jednak zrobić, jeśli trzeba przetestować duże zbiory danych? Załóżmy na przykład, że starszy system ubezpieczeniowy został przeniesiony do nowego systemu i trzeba sprawdzić, czy istniejące dane ubezpieczeń zostały do niego przeniesione prawidłowo. W starszym systemie mogą być dane milionów użytkowników, dlatego do tworzenia przypadków testowych nie można zastosować omówionych do tej pory schematów postępowania. Na przykład nie można określić klas równoważności, ponieważ każdy użytkownik może charakteryzować się innymi wartościami wieku, wysokości składek, czasu trwania umowy, rodzaju ubezpieczenia itp. Nie można także zastosować testowania parami, ponieważ istnieje zbyt wiele zmiennych, aby można było zidentyfikować powtarzające się pary i je wyeliminować. W takich przypadkach użyteczną techniką jest próbkowanie.

Próbkowanie, ogólnie rzecz biorąc, może być zastosowane do dowolnych danych wejściowych, które są ciągłe i obszerne. Polega, jak pokazano na rysunku 2.5, na wybraniu podzbioru wartości do wykorzystania w testach. Zwykle do tego celu stosuje się jedną z następujących technik: próbkowanie losowe lub próbkowanie według specyficznych kryteriów



Rysunek 2.5. Próbkowanie losowe lub próbkowanie na podstawie kryteriów

Próbkowanie losowe polega na wybraniu ze zbioru danych dowolnej próbki danych i zweryfikowaniu wyników. Na przykład jeśli mamy 1000 użytkowników, możemy wybrać losowo ze starszego systemu 50 – 100 użytkowników i porównać ich dane w tym systemie z danymi przechowywanymi w nowym systemie. Próbkowanie według specyficznych kryteriów polega na wybraniu ze zbioru danych próbek poprzez zidentyfikowanie w zbiorze danych pewnych wspólnych cech. Na przykład w systemie ubezpieczeniowym można by pobierać próbki na podstawie kryteriów specyficznych dla użytkownika, takich jak wiek, czas trwania umowy (liczba lat subskrypcji), sposób płatności, zawód itp., oraz dla polisy ubezpieczeniowej — interwały płatności, cena itp. Technikę tę można jeszcze bardziej udoskonalić poprzez stworzenie dla każdego kryterium pewnej liczby próbek proporcjonalnie do rzeczywistego rozkładu wartości w zbiorze danych. Stworzyłoby to reprezentatywny minizbiór danych i prawdopodobnie pokryłoby wszystkie rodzaje przypadków testowych.

Prowadzi to do ostatniego schematu postępowania, które omówię w tym rozdziale. Dotyczy on w większym stopniu ćwiczenia umiejętności analitycznego i logicznego myślenia niż postępowania zgodnie ze zbiorem ustalonych wytycznych. Przejdźmy do omówienia tego schematu postępowania.

Metoda odgadywania błędów

Metoda odgadywania błędów polega na przewidywaniu możliwych błędów na podstawie wcześniejszych doświadczeń. Mogą to być typowe problemy z integracją, sprawdzaniem poprawności danych wejściowych, przypadkami granicznymi itp. Chociaż w przewidywaniu prawdopodobnych przypadków błędów kluczową rolę odgrywają wcześniejsze doświadczenia, do tego celu możesz również wykorzystać swoją wiedzę techniczną i logiczne rozumowanie. Promowanie tego rodzaju myślenia poprawia w gruncie rzeczy umiejętności testowania eksploracyjnego.

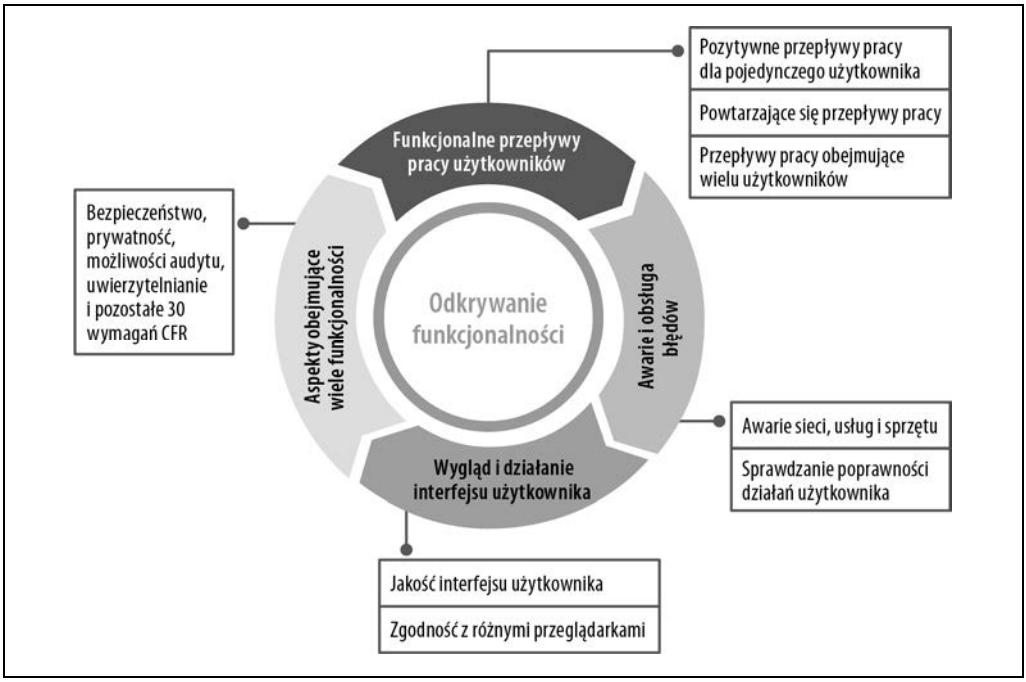
Oto kilka rodzajów błędów, które zgodnie z moimi doświadczeniami występują regularnie:

- Brakująca walidacja nieprawidłowych (pustych) danych wejściowych i brak odpowiednich komunikatów o błędach nakazujących użytkownikowi poprawienie danych wejściowych.
- Niejasne kody stanu HTTP zwracane w celu sprawdzania poprawności danych, techniczna lub biznesowa treść komunikatów o błędach (niektórym z nich przyjrzymy się w sekcji „Testowanie API” dalej w tym rozdziale).
- Nieobsłużone warunki brzegowe specyficzne dla domeny, typów danych, stanów itp.
- Błędy techniczne, takie jak niedostępność serwera, przekroczenie limitu czasu odpowiedzi itp., nieobsłużone po stronie interfejsu użytkownika.
- Problemy z interfejsem użytkownika podczas przejść, odświeżania danych i poruszania się po aplikacji (np. zablokowania i pozostałości).
- Używane zamiennie słowa kluczowe SQL `like` i `equals`, co całkowicie zmienia wyniki.
- Nieopróżniane pamięci podręcznej i niezdefiniowane limity czasu sesji.
- Ponowne publikowanie żądania, gdy użytkownik kliknie w przeglądarce przycisk *Wstecz*.
- Brak sprawdzania poprawności formatu pliku podczas przesyłania plików z różnych platform systemów operacyjnych.

Z opisanego zbioru ośmiu schematów postępowania w testowaniu eksploracyjnym możesz skorzystać w celu ustrukturyzowania procesu myślowego wokół eksploracji funkcjonalności i wnioskowania sensownych przypadków testowych. Należy zapamiętać, że wymienione schematy postępowania można zastosować do dowolnego kontekstu w aplikacji, niekoniecznie wyłącznie do danych wejściowych. Po zaprezentowaniu opisu potrzebnych narzędzi, a przed przejściem do praktycznych ćwiczeń, przyjrzyjmy się bliżej, co kryje się pod pojęciem odkrywania funkcjonalności.

Odkrywanie funkcjonalności

Załóżmy, że poproszono Cię o przeprowadzenie testów eksploracyjnych funkcji tworzenia zamówienia w aplikacji e-commerce. Od jakich ścieżek odkrywania zaczniesz swoją pracę? Na to pytanie spróbuję odpowiedzieć w tym podrozdziale. W tym celu przeanalizujemy cztery podstawowe ścieżki, które — jak pokazano na rysunku 2.6 — należy zbadać w dowolnej aplikacji.



Rysunek 2.6. Cztery podstawowe ścieżki odkrywania podczas eksploracji funkcjonalności

Funkcjonalne przepływy pracy użytkowników

Funkcjonalne przepływy pracy użytkownika aplikacji dotyczą podejmowanych przez docelowego użytkownika podczas korzystania z aplikacji działań, takich jak logowanie, wyszukiwanie produktu, dodawanie go do koszyka, podawanie adresu wysyłki, wybór opcji dostawy czy zapłata za zamówienie, a w końcu potwierdzenie zamówienia. Jest to pozytywny przepływ pracy pojedynczego użytkownika, który należy zweryfikować w pierwszej kolejności. Ten pozytywny przepływ pracy, aby upewnić się, że działa w sposób kompletny, trzeba zbadać z wykorzystaniem różnych adresów wysyłki, sposobów płatności i metod dostawy, a także z różnymi kombinacjami towarów.

Podczas eksploracji możesz odkryć, że korzystasz z niektórych omówionych wcześniej schematów postępowania w testach eksploracyjnych. Na przykład aby sprawdzić, czy aplikacja dodaje odpowiednią kwotę podatku do łącznej ceny, możesz użyć metody podziału klas równoważności oraz analizy wartości granicznych. Z kolei aby wyznaczyć przypadki testowe dotyczące adresu wysyłki i metod dostawy dostępnych dla danej kombinacji adresów, możesz skorzystać z drzewa przejść. Gdy upewnisz się, że ten pozytywny przepływ pracy dla jednego użytkownika działa idealnie, możesz rozpocząć badanie dwóch innych rodzajów przepływów:

Powtarzające się przepływy pracy

Użytkownicy aplikacji często powtarzają ten sam przepływ pracy (lub jego część) wiele razy — na przykład wyszukują różne produkty i dodają je do koszyka. Zwykle jednak dany przepływ pracy użytkownika jest testowany raz, a jeśli działa, to przy założeniu, że zachowanie pozostaje takie samo, powtarzające się przepływy pracy są ignorowane. Jednak w praktyce to założenie może, ale nie musi być prawdziwe. Na przykład jeśli użytkownik spróbuje ponownie dodać ten sam produkt do koszyka, interfejs użytkownika może wyświetlić komunikat informujący, że produkt został już dodany i zapytać, czy chce zwiększyć ilość. Powtarzające się przepływy pracy również trzeba przetestować.

Przepływy pracy obejmujące wielu użytkowników

Funkcjonalność może działać idealnie z punktu widzenia jednego użytkownika, ale może tak nie być, jeśli w interakcję z aplikacją wchodzi jednocześnie kilku użytkowników. Dlatego ważne jest zbadanie możliwych scenariuszy kolizji — sytuacji, kiedy działania jednego użytkownika wpływają na drugiego. Na przykład co się stanie, gdy dwóch różnych użytkowników w tym samym momencie doda do swoich koszyków ostatni dostępny produkt?

Krótko mówiąc, funkcjonalne przepływy użytkowników są często wybierane jako pierwsza ścieżka odkrywania, którą należy zbadać w aplikacji, ale w ramach tej ścieżki czasami trzeba zbadać kilka gałęzi podrzędnych. Do podstawowych podgałęzi należą pozytywne przepływy pracy dla jednego użytkownika, przepływy powtarzające się i przepływy obejmujące wielu użytkowników.

Awarie i obsługa błędów

Jak wspomniałam na początku tego rozdziału, testy eksploracyjne są przeprowadzane w środowisku testowym. Obejmują interakcje z komponentami aplikacji w celu zasymulowania realizowanych w czasie rzeczywistym scenariuszy i obserwowania zachowania aplikacji. W tym zdaniu znajdują się dwie frazy, które ze względu na to, że stanowią rdzeń testów eksploracyjnych, zasługują na uwagę: *interakcje z komponentami aplikacji* i *scenariusze realizowane w czasie rzeczywistym*. Przy rozważaniu scenariuszy realizowanych w czasie rzeczywistym należy również pomyśleć o wszystkich możliwych awariach, ponieważ awarie są praktycznie nieuniknione. Na przykład pomiędzy komponentami aplikacji może wystąpić awaria połączenia sieciowego uniemożliwiająca wysłanie odpowiedzi do użytkownika lub sieć pomiędzy docelowym użytkownikiem a serwerem aplikacji może działać powoli, co jest przyczyną opóźnień, albo usługi aplikacji mogą być wyłączone z powodu awarii sprzętowej. Wszystkie te, a także inne awarie należy przewidzieć podczas testów eksploracyjnych i zasymulować w środowisku testowym.

Oprócz wyżej wymienionych awarii sieci, usług i sprzętu mogą również wystąpić błędy spowodowane nieprawidłowymi działaniami użytkownika. Funkcjonalność można uznać za kompletną tylko wtedy, gdy ma wbudowane mechanizmy walidacji służące do obsługi takich przypadków. W funkcjonalności tworzenia zamówień kilka miejsc wymaga zbadania tych walidacji. Na przykład, jak wspomniałam wcześniej, strona logowania powinna zawierać mechanizmy walidacji adresów e-mail i haseł, a tekst do wyszukiwania wprowadzony w formularzu wyszukiwania produktu powinien zostać zweryfikowany pod kątem nieprawidłowych danych wejściowych, dostępności towaru i tak dalej. Pozostałe miejsca to adres wysyłki i szczegóły płatności, dodawanie towarów do koszyka itp.

Testy eksploracyjne powinny kłaść szczególny nacisk na identyfikację możliwych awarii i obsługę błędów, w ramach której funkcjonalność powinna informować użytkowników o popełnionych pomyłkach oraz sugerować możliwe naprawy poprzez wyświetlenie opisowych komunikatów o błędach.

Wygląd i działanie interfejsu użytkownika

Interfejs użytkownika jest tym, co docelowy użytkownik widzi, i z oczywistych względów nie mogą wystąpić w nim problemy jakościowe. Wygląd i działanie interfejsu użytkownika to zatem kolejna ważna ścieżka do odkrycia. Oto kilka przykładów: przypadki testowe związane z jakością interfejsu użytkownika w funkcji tworzenia zamówień mogą obejmować zapewnienie odpowiedniej ilości miejsca na adresy wysyłki (nie tak mało, aby nie wystarczyło miejsca na wyświetlenie długiej nazwy ulicy, lub nie tak dużo, aby pozostawała ogromna ilość pustego miejsca, gdy ten adres jest krótki) oraz wyświetlanie zdjęć produktów w odpowiednio wysokiej jakości. Docelowi użytkownicy powinni mieć możliwość bezproblemowej obsługi aplikacji z preferowanych przeglądarek, a w przypadku opóźnień powinna być wyświetlana ikona ładowania. Ustrukturyzowane podejście do testowania jakości interfejsu użytkownika zostało szczegółowo omówione w rozdziale 6.

Aspekty obejmujące wiele funkcjonalności

W danej funkcjonalności może istnieć kilka aspektów obejmujących wiele funkcjonalności, takich jak bezpieczeństwo, wydajność, dostępność, uwierzytelnianie, autoryzacja, możliwość audytu, prywatność i tak dalej, wymagających podczas wykonywania testów eksploracyjnych szczególnej koncentracji. Wielu spośród tych aspektów, ze względu na ich znaczenie, poświęcono osobny rozdział tej książki. Oto krótka lista wymagań wielofunkcyjnych, które warto zbadać z punktu widzenia eksploracji funkcjonalności tworzenia zamówień:

Bezpieczeństwo

> W przepływie pracy tworzenia zamówień złośliwy użytkownik mógłby wprowadzać zapytania SQL w polach wejściowych interfejsu użytkownika i próbować włamać się do aplikacji. Aplikacja powinna posiadać mechanizmy walidacji do obsługi tego rodzaju prób. Na podobnej zasadzie dane kart kredytowych użytkowników nie powinny być przechowywane w postaci zwykłego tekstu w bazie danych aplikacji i nie powinny być rejestrowane w postaci zwykłego tekstu w logach aplikacji, tak aby były bezpieczne nawet w przypadku potencjalnego naruszenia zabezpieczeń. Wszystkie wymienione aspekty testowania bezpieczeństwa oraz aspekty dodatkowe zostały szczegółowo omówione w rozdziale 7.

Prywatność

Prywatne dane użytkowników, takie jak dane kart kredytowych czy adresy wysyłki, nie powinny być bez zgody użytkowników przechowywane w bazie danych aplikacji. Ponadto użytkownicy powinni zostać poinformowani o sposobach możliwego wykorzystania ich danych do analiz lub o tym, czy dane te zostaną wysłane do usług podmiotów zewnętrznych w celu przetwarzania. Niektóre klauzule prywatności danych są również egzekwowane przez przepisy prawne. Kwestie te zostaną omówione w rozdziale 10.

Uwierzytelnianie (autoryzacja)

Większość witryn internetowych ma funkcję uwierzytelniania użytkowników, która wymaga zbadania przypadków testowych związanych z uwierzytelnianiem, takich jak logowanie jednokrotne, uwierzytelnianie dwuskładnikowe, wygaśnięcie sesji, blokowanie i odblokowywanie konta itp. W aplikacji e-commerce użytkownicy często mają możliwość przeglądania katalogu produktów bez logowania, ale bez logowania nie mogą składać zamówień.

Mogą też istnieć role (np. administrator, kierownik obsługi klienta) oraz uprawnienia (np. edytowanie zamówienia) przypisane do różnych użytkowników, co wymaga zbadania przypadków testowych związanych z autoryzacją, takich jak wiele nakładających się na siebie ról, dodawanie nowych uprawnień do tych istniejących, obserwowanie zachowania aplikacji, gdy operacja jest wykonywana bez odpowiednich uprawnień itp.

To tylko kilka przykładów. Około 30 różnych aspektów obejmujących wiele funkcjonalności oraz sposobów ich testowania omówiono w rozdziale 10.

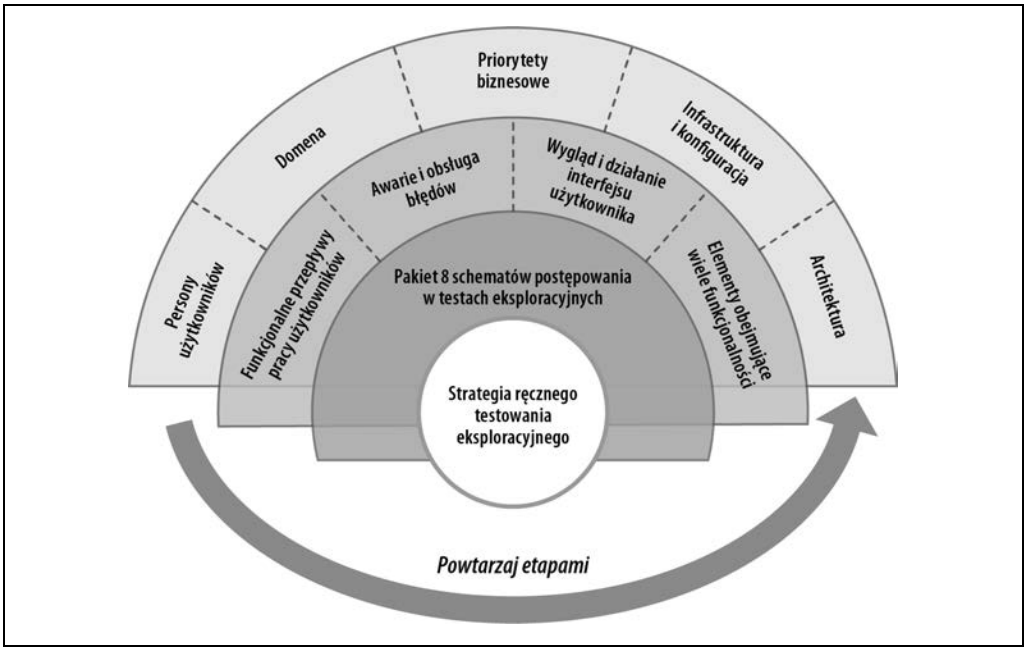
Opisane tutaj cztery ścieżki odkrywania powinny prowadzić do dobrego pokrycia testami dowolnej funkcjonalności. Warto zwrócić uwagę, że podczas eksploracji każdej z tych ścieżek możemy wpadać na nowe pomysły i opracowywać przypadki testowe, które nie należą do określonej ścieżki. Warto je zanotować, aby można było do nich później wrócić lub wykorzystać do eksploracji innej ścieżki.

Strategia ręcznego testowania eksploracyjnego

Strategia ręcznego testowania eksploracyjnego (patrz rysunek 2.7) łączy ze sobą wszystko, co omówiliśmy do tej pory, a ponadto obejmuje procesy zespołowe. W ten sposób możemy uzyskać praktyczny szablon przeprowadzania testów eksploracyjnych w codziennej pracy nad projektem. Zacznijmy od zewnętrznego półkola i spróbujmy przechodzić do wewnątrz.

Zrozumienie aplikacji

Zewnętrzny półokrąg podkreśla zrozumienie szczegółów aplikacji i wskazuje pięć obszernych obszarów aplikacji, na których należy się skupić. Zebranie szczegółów na ich temat pomoże Ci rozpocząć testy eksploracyjne, ale jak wspomniałam wcześniej, podczas eksploracji z pewnością odkryjesz nowe informacje o aplikacji.



Rysunek 2.7. Strategia ręcznego testowania eksploracyjnego

Czasami testy eksploracyjne są mylone z tzw. **testowaniem małpim** (ang. *monkey testing*) — czyli podejściem, w którym aplikacja jest testowana z losowymi danymi wejściowymi i zerową wiedzą na temat funkcjonalności. Należy zapamiętać, że gdy przeprowadzasz testy eksploracyjne, powinieneś dokładnie rozumieć testowaną funkcjonalność i testować ją z nastawieniem na odkrywanie nieznanych aspektów.

Oto krótki opis pięciu rozległych obszarów, na których powinieneś się skupić podczas prób zrozumienia aplikacji:

Persony użytkowników

Persona (*https://oreil.ly/QtpCm*) to postać reprezentująca zbiór docelowych użytkowników o podobnych atrybutach. W zespołach programistycznych takie persony użytkowników są tworzone na początku projektu po to, aby ich specyficzne potrzeby mogły zostać wprowadzone na wszystkich etapach cyklu życia dostarczania oprogramowania, począwszy od fazy projektowania. Przykładem persony użytkowników mającej wpływ na funkcje aplikacji jest serwis społecznościowy, od którego młodzi dorośli oczekują ekstrawagancji, podczas gdy seniorzy oczekują czytelnych i klarownych interakcji. Testowanie sprowadza się do „noszenia kapelusza" docelowego użytkownika, więc znajomość zestawu person użytkowników, które aplikacja zamierza obsługiwać, oraz badanie sposobu, w jaki każda persona postrzega aplikację i wchodzi z nią w interakcje, jest niezbędna.

Dziedzina

Każda dziedzina — sieci społecznościowe, transport, zdrowie itp. — ma spersonalizowany przepływ pracy, proces oraz odpowiednią terminologię lub żargon, który należy rozumieć, aby rozpocząć jej eksplorację. Doskonałym przykładem jest dziedzina e-commerce, gdzie wiedza dziedzinowa ma kluczowe znaczenie dla testowania. Na przykład zamówienie, po jego utworzeniu, przechodzi przez zdefiniowany przepływ pracy: odczytanie, obietnica, potwierdzenie i tak dalej. Przepływ pracy realizacji zamówienia musi wchodzić w interakcje z wieloma podmiotami, takimi jak magazyn, w którym towary są przechowywane, partnerska firma kurierska zajmująca się transportem towarów z magazynu do odbiorcy, oraz dostawcy regularnie uzupełniający towary. Zatem w czasie obserwowania zachowania aplikacji podczas testów eksploracyjnych trzeba wiedzieć, jak podejść do badania wszystkich ścieżek aplikacji. Bez podstawowej wiedzy dziedzinowej może to być trudne.

Priorytety biznesowe

Rozważmy scenariusz, w którym priorytetem biznesowym jest zaprojektowanie rozwiązania jako platformy (*https://oreil.ly/dEd9N*) pod kątem późniejszej rozszerzalności i skalowalności. W takich przypadkach przetestowanie wyłącznie funkcjonalnego przepływu użytkownika za pośrednictwem interfejsu użytkownika może być niewystarczające. Należy go zbadać również z punktu widzenia „platformy" — na przykład zaobserwować, czy interfejs użytkownika i usługi sieciowe są ze sobą ściśle powiązane, czy też usługi sieciowe są niezależne i mogą być zintegrowane z innymi systemami.

Infrastruktura i konfiguracja

Jak wspomniałam wcześniej, testy eksploracyjne obejmują interakcje ze środowiskiem testowym w celu symulacji w czasie rzeczywistym różnych scenariuszy, włącznie z awariami. Posiadanie informacji o tym, gdzie i które składniki aplikacji zostały wdrożone, oraz znajomość konfigurowalnych ustawień aplikacji dostarczają kluczowych wskazówek dotyczących znajdowania nowych ścieżek odkrywania. Na przykład usługi sieciowe mogą być skonfigurowane z maksymalną liczbą dostępów, które mogą obsługiwać w danym okresie, co jest znane jako **ograniczanie przepustowości** (*https://oreil.ly/TYa3z*). W przypadku takich aplikacji może być konieczne obserwowanie ich zachowania po przekroczeniu limitu przepustowości. Zebranie podstawowych informacji o infrastrukturze i konfiguracji, na przykład o sposobie wdrożenia usług i bazy danych (na jednym komputerze lub w trybie rozproszonym na wielu komputerach), ustawień ograniczeń przepustowości, bramy interfejsu API i tym podobnych, pomoże Ci odkryć ważne przypadki testowe.

Architektura aplikacji

Znajomość architektury aplikacji dodaje w sesji testowania eksploracyjnego nowe gałęzie do ścieżek odkrywania. Na przykład jeśli architektura obejmuje usługi sieciowe, może być konieczne przeprowadzenie testów eksploracyjnych interfejsu API (omówionych w punkcie „Testowanie API" dalej w tym rozdziale) zamiast eksplorowania wyłącznie interfejsu użytkownika. Podobnie — jeśli aplikacja zawiera strumienie zdarzeń (omówione w rozdziale 5.), ważne staje się zbadanie przypadków związanych z komunikacją asynchroniczną. Wysokopoziomowe zrozumienie architektury pomoże Ci znaleźć ścieżki odkrywania w kontekście integracji wewnętrznych komponentów, przepływu danych między komponentami, integracji podmiotów zewnętrznych i obsługi błędów. Kilka z tych aspektów również omówiono w innych częściach tej książki.

Po zebraniu wystarczających informacji na temat tych pięciu obszarów aplikacji jesteś gotowy do przystąpienia do faktycznej realizacji fazy testów eksploracyjnych.

Jeśli to brzmi trochę przytłaczająco — szczególnie w części dotyczącej architektury i infrastruktury — na razie nie martw się zbytnio o te szczegóły. Nie ma niczego złego w podejściu do testów eksploracyjnych z funkcjonalnego punktu widzenia i stopniowe uczenie się zadawania dodatkowych pytań.

Eksploracja częściami

Kolejny półokrąg na diagramie strategii ręcznego testowania eksploracyjnego na rysunku 2.7 wskazuje na eksplorację częściami.

James Bach w swoim artykule z 2003 roku *Exploratory Testing Explained* (*https://oreil.ly/B7jaO*) definiuje tę praktykę jako „jednoczesne uczenie się, projektowanie testów i ich wykonywanie". Obecnie jest to jedna z najczęściej stosowanych definicji testów eksploracyjnych. Mówiąc dokładniej, testowanie eksploracyjne polega na wykonywaniu w aplikacji szeregu działań przy jednoczesnym obserwowaniu jej zachowania, a tym samym polega na zdobywaniu wiedzy o aplikacji i stopniowym jej odkrywaniu. Taki proces wymaga od naszych umysłów ciągłej czujności i przeprowadzania *dogłębnej analizy*. Ludzie są intensywnie skupieni na jakimś temacie i naprawdę się w niego zagłębiają w przypadku, gdy jego zakres jest dość niewielki. Dlatego powinniśmy badać aplikacje częściami! Części te mogą być dowolnymi spośród omówionych wcześniej ścieżek odkrywania lub mogą stanowić dowolną podgałąź w ścieżce, taką jak przepływ pracy użytkownika, funkcjonalność lub aspekt obejmujący wiele funkcji, na przykład zabezpieczenia.

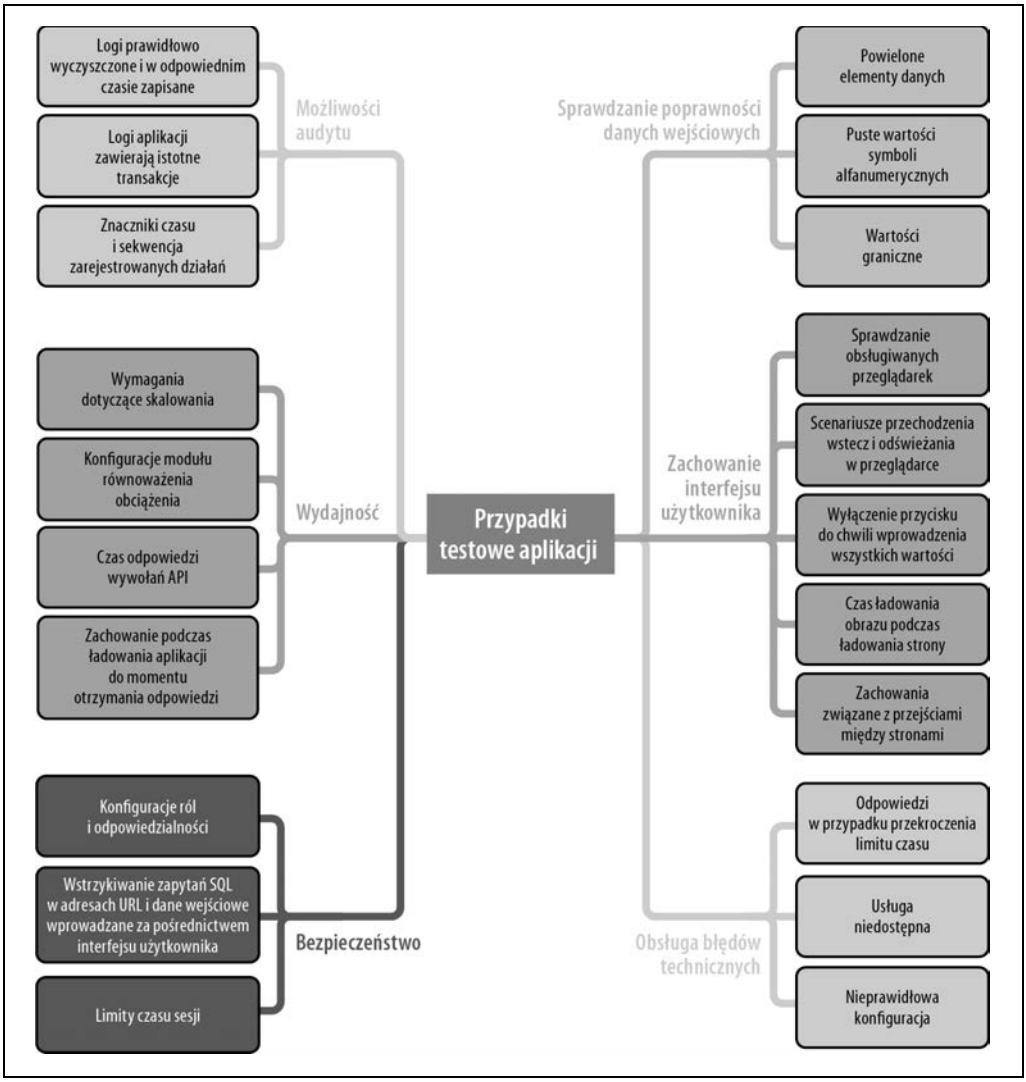
Śledzenie wszystkich tych ścieżek i podgałęzi podczas dogłębnej eksploracji może być trudne. Jedną ze strategii postępowania w takim przypadku jest skorzystanie z mapy myśli (ang. *mind map*)[1] podobnej do tej z rysunku 2.8. Można ją udostępnić całemu zespołowi.

Podczas tej fazy możesz również potrzebować pakietu ośmiu schematów postępowania podczas testowania eksploracyjnego, reprezentowanych przez wewnętrzny półokrąg z rysunku 2.7.

Powtarzanie testów eksploracyjnych w wielu fazach

Testowanie eksploracyjne nie może być jednorazowym działaniem. Zespół stale dodaje nowy kod, implementuje nowe funkcjonalności i przeprowadza nowe integracje, co skutkuje zmianami w zachowaniu aplikacji, a tym samym potrzebą ponownej eksploracji. Potraktowanie testów eksploracyjnych jako ciągłego procesu pozwala ustrukturyzować zakres, który powinien być dogłębnie zbadany w określonym czasie lub w określonej fazie życia aplikacji. Na przykład niektóre zespoły Agile stosują **testowanie dev-box**, polegające na tym, że przedstawiciele biznesowi wspólnie z testerami

[1] Mapa myśli to technika wizualizacji, w której są przechwytywane główne idee wraz z ich rozgałęzieniami. Do narysowania mapy myśli można użyć takich narzędzi jak Coggle (*https://coggle.it*) i XMind (*https://www.xmind.net*).



Rysunek. 2.8. Mapa myśli do testowania eksploracyjnego

wykonują na komputerze programisty ograniczone czasowo testy eksploracyjne świeżo zaimplementowanych historyjek użytkownika. W tym przypadku można ograniczyć zakres do wyłącznie pozytywnych przepływów pracy użytkowników, walidacji oraz wyglądu i działania interfejsu użytkownika. Kolejną fazą sprzyjającą eksploracji jest faza testowania historyjek użytkownika po ich zaprogramowaniu. W tej fazie można rozszerzyć zakres eksploracji i uwzględnić niektóre aspekty dotyczące wielu przeglądarek oraz wielu funkcjonalności. Ponadto niektóre zespoły Agile przeprowadzają regularne **polowania na błędy** (ang. *bug bashes*), podczas których wszyscy członkowie zespołu spotykają się w celu eksplorowania już zaprogramowanych funkcjonalności aplikacji. I na końcu, w fazie testowania wersji do publikacji, możemy skupić się na aspektach obejmujących wiele funkcjonalności,

takich jak wydajność, niezawodność i skalowalność, a także badać pozytywne przepływy pracy użytkowników i integracje na nieco wyższym poziomie. Planowanie z wyprzedzeniem faz testów eksploracyjnych pomaga zespołowi nieprzerwanie uzyskiwać informacje zwrotne, a tym samym dostarcza przestrzeni do ciągłego doskonalenia.

Testowanie eksploracyjne ma charakter organiczny. W związku z tym możesz odkrywać nowe ścieżki, których nie planowałeś zgłębiać, i które mogą pochłonąć Twój czas przydzielony w iteracji. Należy się tego spodziewać. W takich sytuacjach warto zastanowić się, czy określona ścieżka może zostać uwzględniona w następnej fazie testowania historyjki użytkownika lub podczas sesji warsztatów *bug bash*, a jeśli tak, to należy to zaplanować i przejść do dalszego testowania.

Podsumowując strategię testów eksploracyjnych: najpierw poznaj szczegóły aplikacji, a następnie zanotuj poszczególne ścieżki do zbadania. Następnie kontynuuj eksplorację różnych ścieżek w kolejnych fazach cyklu dostarczania aplikacji, aby zapewnić zespołowi ciągłe informacje zwrotne.

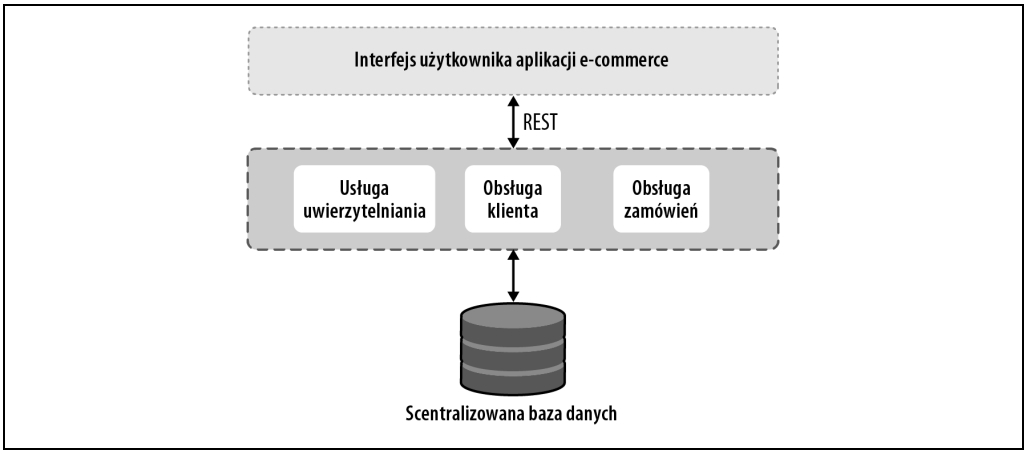
Ćwiczenia

Do tej pory omówiliśmy wiele teorii na temat schematów postępowania i dostępnych strategii. Aby zastosować je do eksploracji ścieżek odkrywania aplikacji, może być konieczne nauczenie się kilku narzędzi, na przykład języka SQL do eksploracji bazy danych (co zostało omówione w rozdziale 5.), narzędzia Postman do eksploracji interfejsów API i tak dalej. Kilka takich narzędzi zostało opisanych w różnych częściach tej książki. W tym podrozdziale omówię narzędzia do testowania eksploracyjnego interfejsu API oraz webowego interfejsu użytkownika.

Testowanie API

Interfejs programowania aplikacji (ang. *application programming interface*), czyli interfejs API (*https://oreil.ly/jNiSY*), umożliwia systemom komunikowanie się ze sobą. Ogólnie rzecz biorąc, interfejsy API abstrahują od wewnętrznej złożoności systemu i upraszczają wymianę informacji w sieci z wykorzystaniem protokołu HTTP w formatach XML, JSON lub w formacie zwykłego tekstu. Aby ustandaryzować wymianę informacji, wynaleziono takie protokoły jak SOAP i specyfikacje takie jak REST. Obecnie interfejsy API RESTful są bardziej rozpowszechnione niż SOAP, a starsze systemy korzystające z SOAP są przepisywane na specyfikacje REST. Aby ułatwić zrozumienie interfejsów API REST, rozważmy prostą aplikację e-commerce, taką jak pokazana na rysunku 2.9, składającą się z trzech usług REST (zamówienia, uwierzytelnianie i obsługa klienta), interfejsu użytkownika i bazy danych.

Usługa sieciowa to komponent, który spełnia w aplikacji niezależny cel. Na przykład usługa zamawiania w przykładowej aplikacji e-commerce może być odpowiedzialna za zarządzanie (tworzenie, aktualizowanie i usuwanie) zamówień, podczas gdy usługa obsługi klienta jest odpowiedzialna



Rysunek 2.9. Przykładowa aplikacja e-commerce o architekturze zorientowanej na usługi

za utrzymywanie danych klientów. Podział na osobne usługi sieciowe ułatwia wymianę informacji, ponieważ inne komponenty aplikacji, takie jak interfejs użytkownika lub inne usługi, aby uzyskać potrzebne informacje, mogą sięgać do interfejsu API odpowiedniej usługi.

Architektura zorientowana na usługi to taka, w której podstawowe funkcjonalności aplikacji są zaimplementowane w postaci usług sieciowych tak, jak pokazano na rysunku 2.9.

Dla przykładu załóżmy, że docelowy użytkownik chce zapłacić za zamówienie za pośrednictwem interfejsu użytkownika aplikacji e-commerce. Jak pokazano na listingu 2.1, interfejs użytkownika natychmiast wysyła do usługi zamówień żądanie utworzenia zamówienia zawierające jego szczegóły, a usługa obsługi zamówień przetwarza żądanie i wysyła odpowiedź do interfejsu użytkownika. Interfejs użytkownika e-commerce w tym kontekście jest nazywany *klientem*.

Listing 2.1. Przykładowe żądanie i odpowiedź REST

```
// Żądanie

POST method: http://eCommerce.com/orders/new
{
   "name":"V-Neck Tshirt",
   "sku":"ABCD1234",
   "color":"Red",
   "size":"M"
}

// Odpowiedź

Status Code: 200 OK
Response Body:
{
   "Msg": "utworzono pomyślnie",
   "ID": "Order1234227891"
}
```

Jeśli spojrzysz na żądanie z listingu 2.1, zauważysz, że za pośrednictwem metody HTTP `POST` trafia ono do interfejsu API `/orders/new`. Zazwyczaj żądania `POST` są wykorzystywane do tworzenia lub dodawania nowych informacji, natomiast żądania `GET` są używane do pobierania informacji — na przykład pobierania listy złożonych przez klienta zamówień. Istnieją również metody `PUT` i `DELETE`, używane odpowiednio do działań aktualizacji i usuwania. W treści żądania, które ma format obiektu JSON, są umieszczone szczegóły zamówienia — w tym przypadku nazwa towaru, kod SKU, kolor i rozmiar. Cała ta struktura jest określana jako *kontrakt*. Jeśli klient nie dotrzyma tego kontraktu, usługa nie przetworzy przesłanego żądania.

Na podobnej zasadzie, tzn. zgodnie z kontraktem, jest również przesyłana odpowiedź: zawiera kod statusu operacji wskazujący na jej powodzenie lub niepowodzenie, a także treść odpowiedzi zawierająca więcej informacji dotyczących operacji. Na listingu 2.1 kod statusu odpowiedzi to `200 OK`, co oznacza sukces, a treść odpowiedzi zawiera komunikat „utworzono pomyślnie" wraz z identyfikatorem zamówienia wygenerowanym przez usługę obsługi zamówień. Po otrzymaniu tej odpowiedzi interfejs użytkownika aplikacji e-commerce przeniesie użytkownika na stronę potwierdzenia zamówienia i wyświetli na niej identyfikator zamówienia. Należy zapamiętać, że wszystkie te działania są wykonywane synchronicznie — innymi słowy, interfejs użytkownika aplikacji e-commerce, zanim przejdzie do strony potwierdzenia zamówienia, poczeka, aż otrzyma odpowiedź.

Po zdobyciu podstawowej wiedzy na temat działania interfejsów API można by zadać pytanie: dlaczego trzeba eksplorować interfejsy API osobno, skoro istnieje możliwość przetestowania z webowego interfejsu użytkownika funkcjonalności tworzenia zamówień? Na to pytanie można zwięźle odpowiedzieć, że dziś interfejsy API same stały się produktami! W nieco pełniejszej odpowiedzi należałoby zaznaczyć, że interfejsy API obejmują całą logikę biznesową i wszystkie walidacje, co czyni je samodzielnymi produktami do wielokrotnego wykorzystania przez inne wewnętrzne i zewnętrzne komponenty. Na przykład w naszej hipotetycznej firmie e-commerce moglibyśmy zbudować nową mobilną aplikację zakupową i skorzystać z tego samego interfejsu API do tworzenia zamówień, albo zbudować portal obsługi klienta na bazie tego samego interfejsu API obsługi klienta. Interfejsy API mogą nawet rozgałęzić się na zupełnie nową dziedzinę, w której można by ponownie wykorzystać interfejsy API usługi uwierzytelniania w celu zbudowania w tym nowym produkcie funkcjonalności logowania. Tak więc w dzisiejszym cyfrowym świecie eksploracja interfejsów API jako samodzielnych produktów jest bardzo ważna.

Oto niektóre, oprócz podstawowej logiki biznesowej, ścieżki odkrywania, na które należy zwrócić uwagę podczas eksploracji interfejsów API:

Walidacja kontraktu żądania

Walidacja powinna być wykonana po to, aby w przypadku, gdy nieuczciwy klient utworzy nowe zamówienie z nieprawidłowym formatem danych, usługa zamówienia odrzuciła to żądanie.

Uwierzytelnianie

W większości przypadków interfejsy API ze względów bezpieczeństwa są chronione za pomocą pewnych mechanizmów uwierzytelniania, na przykład przez wysyłanie w nagłówku żądania tokena (długiego, zaszyfrowanego ciągu). Jest to ważna ścieżka odkrywania, którą warto zbadać.

Uprawnienia

Interfejsy API mogą mieć ograniczenia dotyczące operacji wykonywanych na rzecz swoich aplikacji klienckich. Na przykład administrator może mieć prawo do edytowania istniejącego zamówienia, ale kierownika ds. klientów mogą obowiązywać ograniczenia i może on być uprawniony wyłącznie do przeglądania zamówienia.

Zgodność z poprzednimi wersjami

Czasami, wraz z rozwojem produktu, kontrakty API również mogą wymagać zmiany. Ponieważ jednak mogą istnieć aplikacje klienckie korzystające z interfejsów API, może powstać potrzeba stworzenia nowych wersji i utrzymywania ich równolegle ze starymi. Aplikacja powinna być przetestowana z obiema wersjami interfejsu API.

Kody odpowiedzi HTTP

Kody odpowiedzi zwracane w przypadku awarii technicznych i biznesowych powinny być sensowne. Najczęściej wykorzystywane kody zestawiono w tabeli 2.4.

Tabela 2.4. Kody odpowiedzi HTTP i ich znaczenie

Kod odpowiedzi	Znaczenie
`200 OK`	Wskazuje na powodzenie żądań `GET`, `PUT` lub `POST`.
`201 Created`	Wskazuje na utworzenie nowego obiektu, na przykład nowego zamówienia.
`400 Bad Request`	Wskazuje na nieprawidłowy format żądania.
`401 Unauthorized`	Wskazuje, że klient nie może uzyskać dostępu do żądanego zasobu i powinien ponownie wysłać żądanie z wymaganymi poświadczeniami.
`403 Forbidden`	Wskazuje, że żądanie jest prawidłowe, a klient został uwierzytelniony, ale z jakiegoś powodu nie ma dostępu do żądanej strony lub zasobu.
`404 Not Found`	Wskazuje, że żądany zasób nie jest dostępny.
`500 Internal Server Error`	Wskazuje, że żądanie jest prawidłowe, ale serwer nie może go obsłużyć, prawdopodobnie z powodu wewnętrznych błędów.
`503 Service Unavailable`	Wskazuje, że serwer jest wyłączony (na przykład podczas konserwacji).

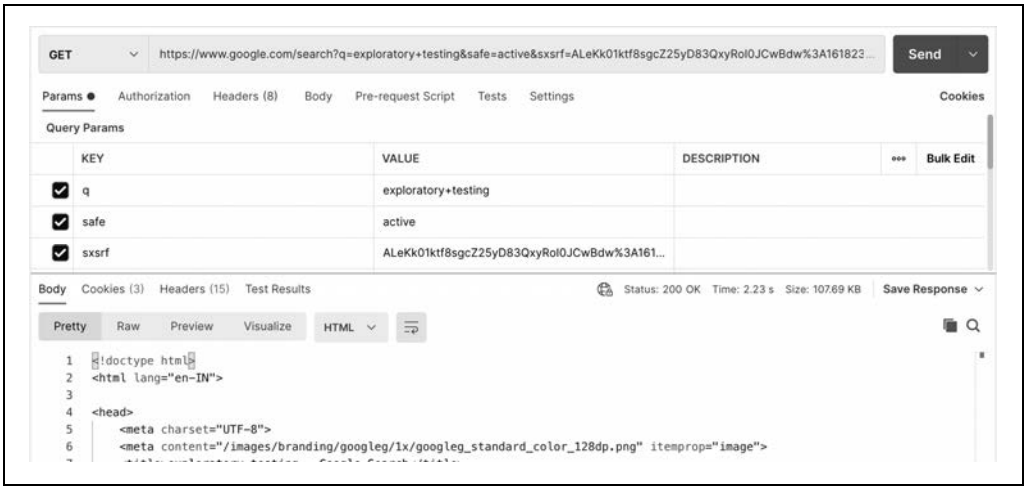
Aby zbadać wszystkie te ścieżki odkrywania, potrzebne są odpowiednie narzędzia. Kilka z nich przedstawię w następnych punktach.

Postman

Postman (*https://www.postman.com*) to popularne narzędzie do testowania interfejsów API. Pliki binarne instalacji dla wersji przeznaczonej na komputery desktop są dostępne za darmo. Dostępna jest również webowa darmowa wersja próbna. Oto krótki przewodnik dla desktopowej wersji aplikacji:

1. Pobierz z oficjalnej strony produktu (*https://www.postman.com/downloads*) binarny plik instalacyjny dla swojego systemu operacyjnego.
2. Otwórz Postmana i wybierz polecenie *New/HTTP Request*. Spowoduje to przejście do okna nowego żądania.

3. Poszukaj w Google frazy „exploratory testing”, a następnie skopiuj adres URL i wklej go w polu adresu URL w oknie nowego żądania w programie Postman, jak pokazano na rysunku 2.10. Zwróć uwagę, że na rozwijanej liście obok tego pola metoda HTTP została automatycznie ustawiona na `GET`.
4. W zakładce *Params* zostaną automatycznie wypełnione parametry żądania odpowiadające wyszukiwaniu w Google. Parametr zapytania `q` będzie miał wartość `exploratory+testing`. Aby wyszukać inne słowo kluczowe, możesz je dowolnie zmienić.
5. Aby dokończyć żądanie, naciśnij przycisk *Send*.
6. W dolnym panelu wyświetli się kod statusu odpowiedzi, nagłówki, treść i pliki cookie (patrz rysunek 2.10).



Rysunek 2.10. Tworzenie nowego żądania i weryfikacja odpowiedzi za pomocą narzędzia Postman

W tym przypadku odpowiedź ma format HTML. Jeśli klikniesz przycisk *Preview*, wyświetli się dokładnie taka sama strona wyników wyszukiwania, jak wyświetla się w przeglądarce. W wielu przypadkach, jak widzieliśmy na listingu 2.1, odpowiedź będzie miała format obiektu JSON. Interfejs użytkownika sparsuje kod JSON i wyświetli właściwe informacje.

Wyszukiwanie w Google było żądaniem `GET`, ale czynności wymagane do stworzenia żądania POST są podobne: wybierz z rozwijanej listy `POST` jako metodę HTTP, wprowadź w polu *URL* żądanie API i treść żądania w zakładce *Body*, a następnie, aby zobaczyć odpowiedź, naciśnij *Send*. Jeśli chcesz poćwiczyć trochę więcej, możesz skorzystać z narzędzia Any API (*https://any-api.com*), zawierającego listę 1 400 publicznie hostowanych interfejsów API REST dostępnych do wypróbowania.

Aplikacja Postman domyślnie zapisuje swój obszar roboczy w chmurze na koncie Postman użytkownika. Funkcjonalność ta może być przydatna w przypadku konieczności zsynchronizowania pracy (*https://oreil.ly/Yl5v9*) na nowym komputerze. Należy się jednak upewnić, że synchronizacja z chmurą Postman nie narusza umów o zachowaniu poufności (NDA) z klientami ani wewnętrznych zasad IT.

Postman dostarcza kilku innych mechanizmów eksploracji interfejsów API. Oto kilka powszechnie wykorzystywanych:

- Dodanie tokena w zakładce *Authorization* spowoduje wysłanie wraz z żądaniem tokena uwierzytelniającego. Możesz wprowadzić tam nieprawidłowe ciągi, aby sprawdzić, czy żądanie zakończy się niepowodzeniem.
- Podobnie, aby wraz z żądaniem wysłać pliki cookie, można je dodać na zakładce *Cookies* (zakładka ta znajduje się pod przyciskiem *Send*).
- Postman rejestruje czas potrzebny na odebranie odpowiedzi, co można zaobserwować na rysunku 2.10 obok kodu statusu odpowiedzi. Jest to pomocne w celu szybkiego zbadania, czy dla różnych danych wejściowych zmienia się wydajność.
- Zamiast tworzyć żądania ręcznie, możesz bezpośrednio zaimportować specyfikacje API za pomocą linków do narzędzi Swagger, OpenAPI itp.

Oprócz możliwości testowania usług REST Postman zapewnia również obsługę testowania usług GraphQL i SOAP.

WireMock

WireMock (*http://wiremock.org*) to narzędzie do tworzenia i modyfikowania **makiet** (ang. *stubs*), czasami nazywanych także **zaślepkami**, czyli komponentów oprogramowania emulujących zachowanie innych komponentów. Makiety są szczególnie przydatne podczas tworzenia i testowania złożonych aplikacji obejmujących wiele integracji, zwłaszcza gdy nie wszystkie usługi integracyjne są gotowe. Zespoły uzgadniają umowę serwisową i po utworzeniu makiet integrowanych usług mogą kontynuować programowanie. Makiety tworzy się poprzez jawne zaprogramowanie ich w taki sposób, aby na określone żądania odpowiadały wysłaniem zdefiniowanych wyników. Tę funkcjonalność można wykorzystać w testach eksploracyjnych w celu skonfigurowania różnych pozytywnych i negatywnych przypadków testów integracji. Oczywiście po pojawieniu się rzeczywistych komponentów konieczne jest ponowne przetestowanie kompletnej funkcjonalności.

Skonfigurowaniem serwera-zaślepki i ustawieniem aplikacji w taki sposób, aby wskazywała na tę zaślepkę, może zająć się inżynier DevOps lub wybrany programista w zespole. Testerzy muszą jednak wiedzieć, jak modyfikować zaślepkę, aby móc symulować różne przypadki testowe. To ćwiczenie zostało dołączone specjalnie w tym celu.

Aby zilustrować wykorzystanie programu WireMock, wróćmy do przykładowej aplikacji e-commerce. Załóżmy, że nie mamy rzeczywistej usługi płatniczej, którą moglibyśmy zintegrować z aplikacją, ale znamy kontrakt żądanie-odpowiedź punktu końcowego `/makePayment` używanego przez interfejs użytkownika aplikacji e-commerce do wysyłania płatności. Aby zbadać różne przypadki testowe tej integracji, należy skonfigurować zaślepkę punktu końcowego `/makePayment` z pozytywnymi i negatywnymi odpowiedziami. Oto czynności, które należy wykonać w tym celu:

1. Pobierz z oficjalnej strony internetowej narzędzia WireMock (*https://oreil.ly/qsBOh*) samodzielne archiwum JAR.

2. Otwórz terminal i uruchom dla pobranej wersji archiwum JAR następujące polecenie:

   ```
   $ java -jar wiremock-jre8-standalone-x.x.x.jar
   ```

   Powyższe polecenie uruchamia serwer WireMock w porcie 8080.

3. Aby utworzyć nową zaślepkę, skonstruuj kontrakt interfejsu API `/makePayment` tak, jak pokazano na listingu 2.2 i za pomocą Postmana wyślij żądanie `POST` do punktu końcowego *http://localhost:8080/__admin/mappings/new* (w oknie nowego żądania w aplikacji Postman ustaw w rozwijanym menu metodę HTTP na wartość `POST`, w polu *URL* wprowadź adres URL, a w polu *Body/raw* wpisz kod JSON z listingu 2.2; następnie naciśnij *Send*).

   Listing 2.2. Przykładowa zaślepka utworzona za pomocą narzędzia WireMock

   ```
   {
      "request": {
          "method": "POST",
          "url": "/makePayment"
      },
      "response": {
           "status": 200,
           "body": "Płatność powiodła się"
      }
   }
   ```

4. Teraz sprawdź, czy zaślepka działa. W tym celu utwórz kolejne żądanie `POST` do adresu URL *http://localhost:8080/makePayment*. Powinieneś otrzymać odpowiedź z kodem statusu `200 OK` i komunikatem „Płatność powiodła się”, zgodnie z ustawieniami w zaślepce. Po otrzymaniu tej odpowiedzi wyobrażony interfejs użytkownika przykładowej aplikacji e-commerce powinien wyświetlić stronę z potwierdzeniem zamówienia.

5. Aby zmienić zaślepkę w taki sposób, żeby zwracała odpowiedź negatywną, zmień treść odpowiedzi na listingu 2.2 na pokazaną poniżej i wyślij żądanie `POST` do tego samego punktu końcowego `/mappings/new`:

   ```
       "response": {
           "status": 401,
           "body": "Płatność nieudana"
   }
   ```

   Po otrzymaniu tej odpowiedzi interfejs użytkownika aplikacji e-commerce powinien wyświetlić komunikat o błędzie.

W podobny sposób można skonfigurować inne przypadki testowe (nieprawidłowe żądania, scenariusze niedostępności usługi itp.). Należy ustawić odpowiednie kody statusu HTTP w treści odpowiedzi i obserwować, czy interfejs użytkownika odpowiednio je obsługuje. Jak można zobaczyć, zaślepki pomagają w testowaniu eksploracyjnym API w sytuacji, gdy rzeczywiste usługi integracyjne nie są dostępne do testowania.

Teraz możemy przejść do narzędzi eksploracyjnego testowania interfejsu użytkownika.

Testowanie interfejsu webowego

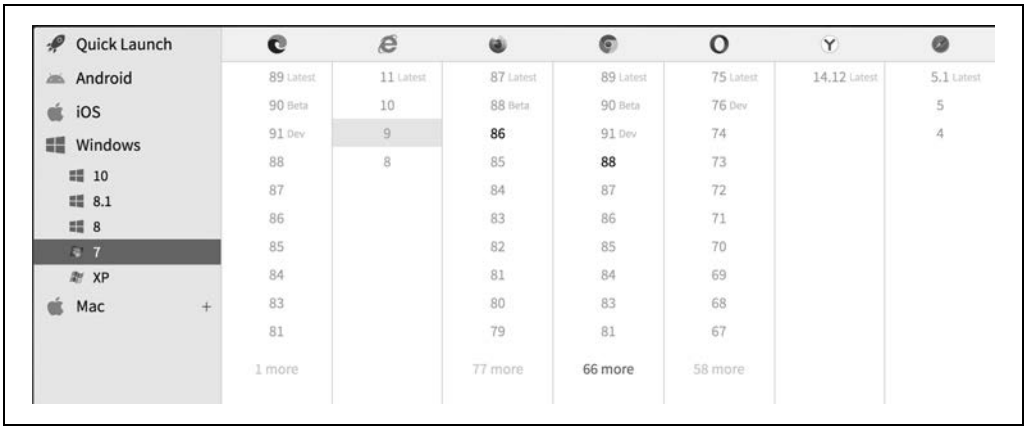
W tym punkcie zostały zaprezentowane trzy podstawowe narzędzia do testowania interfejsu użytkownika: przeglądarki, program Bug Magnet i zestaw narzędzi Chrome DevTools.

Przeglądarki

Pierwszym i najważniejszym narzędziem do eksploracji webowego interfejsu użytkownika jest przeglądarka. Najlepszą praktyką testowania jest pokrycie co najmniej 85% bazy użytkowników aplikacji. Zgodnie z najnowszymi — w chwili pisania tego tekstu — statystykami opublikowanymi w witrynie *gs.statcounter.com* (*https://gs.statcounter.com*) dotyczącymi globalnego rozkładu wykorzystania przeglądarek przeglądarka Chrome miała około 64,5% udziału w rynku, następna była przeglądarka Safari z udziałem 18,8%, następnie Edge z 4,05%, Firefox 3,4% i Samsung Internet z udziałem 2,8%. Statystyki te wskazują, że w testach koniecznie trzeba uwzględnić przeglądarki Chrome i Safari. Na trzecim miejscu niemal *ex aequo* są przeglądarki Edge i Firefox, dlatego w testach jakości interfejsu użytkownika wskazane jest uwzględnienie ich obu. Każdą z tych przeglądarek, dla dowolnego systemu operacyjnego, możesz pobrać na swój lokalny komputer.

Czasami konieczne jest przetestowanie aplikacji na starszych przeglądarkach, na przykład Internet Explorer 11 lub Edge Legacy, chociaż firma Microsoft oficjalnie zakończyła wsparcie dla tych wersji. Jednym ze sposobów poradzenia sobie w tej sytuacji jest pobranie na swój komputer maszyny wirtualnej z systemem Windows (*https://oreil.ly/IOUWS*).

Można również skorzystać z platform testowych hostowanych w chmurze, takich jak BrowserStack (*https://www.browserstack.com*) i Sauce Labs (*https://saucelabs.com*), które zwalniają nas z konieczności instalowania różnych wersji przeglądarek i systemów operacyjnych na komputerach lokalnych. Platformy te ,za opłatą, zapewniają wirtualny dostęp do przeglądarek internetowych działających na różnych systemach operacyjnych. Proces jest bardzo prosty: należy zapłacić za subskrypcję (dostępne są również bezpłatne wersje próbne), zalogować się do portalu, wybrać kombinację wersji przeglądarki i systemu operacyjnego (tak jak pokazano na rysunku 2.11) i przetestować swoją aplikację.



Rysunek 2.11. BrowserStack i podobne usługi pozwalają testować różne kombinacje wersji systemu operacyjnego i przeglądarki

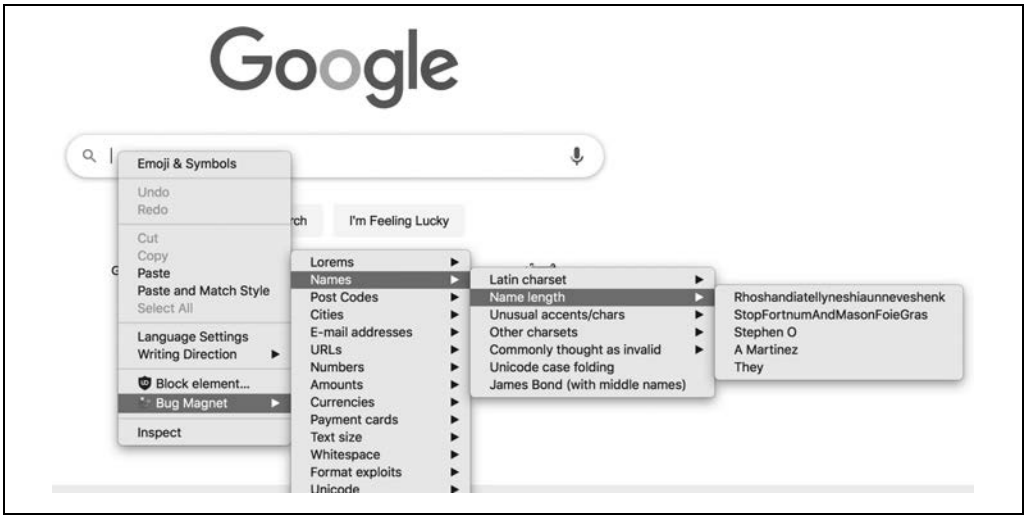
Aplikacja BrowserStack umożliwia również lokalne testowanie (*https://oreil.ly/6DLat*) aplikacji prywatnych, hostowanych w środowiskach testowych (ang. *quality assurance* — QA) lub przedprodukcyjnych (ang. *staging environments*). W zależności od potrzeb testowania zasubskrybowanie

takiej usługi może być opłacalne. Wymienione platformy mogą okazać się cenne, zwłaszcza gdy istnieje konieczność przetestowania aplikacji na szerokiej gamie starszych przeglądarek.

Bug Magnet

Bug Magnet (*https://bugmagnet.org*) to wtyczka dostępna dla przeglądarek Chrome i Firefox, umożliwiająca testowanie przypadków brzegowych w aplikacji. Zawiera listę typowych przypadków testowych i odpowiednich wartości, które należy wprowadzić dla każdego przypadku testowego w edytowalnych elementach aplikacji. Narzędzie można wykorzystać głównie w formie listy kontrolnej na potrzeby testów eksploracyjnych. Aby je wypróbować:

1. Zainstaluj wtyczkę (*https://oreil.ly/5sbqz*) w przeglądarce Chrome.
2. Otwórz wyszukiwarkę Google (*https://www.google.com*) i kliknij prawym przyciskiem myszy pole tekstowe wyszukiwania.
3. Polecenie wywołania programu Bug Magnet znajduje się w menu podręcznym, dostępnym pod prawym przyciskiem myszy, co widać na rysunku 2.12. Jak widać, sugeruje wiele przypadków brzegowych, z których można wybrać jeden. Możesz na przykład wybrać polecenie *Names/Name Length* i wybrać imię. W polu tekstowym wyszukiwarki Google zostanie wypełniona długa nazwa. Jeśli aplikacja obejmuje mechanizmy walidacji długości wejściowego ciągu, to powinien wyświetlić się odpowiedni komunikat o błędzie.



Rysunek 2.12. Podczas ręcznych testów eksploracyjnych możesz użyć wtyczki Bug Magnet w roli przewodnika

Oprócz Bug Magnet dostępnych jest również kilka innych heurystycznych ściągawek do testowania eksploracyjnego (*https://oreil.ly/O29Em*), których możesz użyć, aby nie przegapić przypadków testowych. Są one szczególnie przydatne dla początkujących.

Wtyczka DevTools przeglądarki Chrome

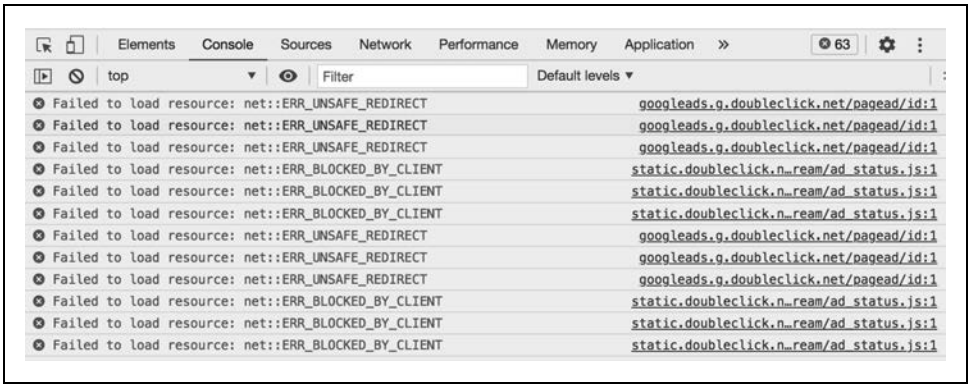
DevTools (*https://oreil.ly/T0rlU*) to „uniwersalny szwajcarski scyzoryk" przeglądarki Chrome. Zawiera wiele mechanizmów, które mogą być pomocne w testach eksploracyjnych, testach bezpieczeństwa, testach wydajności i innych. Okienko tego narzędzia zobaczysz w wielu rozdziałach tej książki, w różnych miejscach. Aby zorientować się, co oferuje, wykonaj następujące działania:

1. Otwórz przeglądarkę Chrome i wyszukaj frazę „testy eksploracyjne".
2. Kliknij prawym przyciskiem myszy stronę wyników wyszukiwania i wybierz opcję *Zbadaj*. Okienko narzędzia DevTools otworzy się natychmiast. Aby otworzyć narzędzie DevTools, możesz także użyć skrótów klawiaturowych — *Cmd-Option-C* lub *Cmd-Option-I* w systemie macOS albo *Shift-Ctrl-J* w systemie Windows.

W oknie narzędzia DevTools możesz odkrywać szereg elementów takich jak:

Błędy strony

Jak widać na rysunku 2.13, w zakładce *Konsola* wyświetlają się błędy występujące na stronie internetowej. Ogólnie rzecz biorąc, strona internetowa powinna być wolna od błędów, dlatego dobrym pomysłem jest sprawdzenie tej zakładki podczas przechodzenia na kolejne strony testowanej aplikacji. Błędy wyświetlane na tej zakładce mogą również pomóc w debugowaniu wszelkich problemów znalezionych na stronie internetowej. Na przykład jeśli zauważysz, że brakuje ilustracji, możesz sprawdzić zakładkę *Konsola* i dołączyć wyświetlany tam błąd do raportu o błędach.



Rysunek 2.13. Zakładka Konsola wyświetlająca błędy na stronie internetowej

Liczba żądań wysłanych ze strony

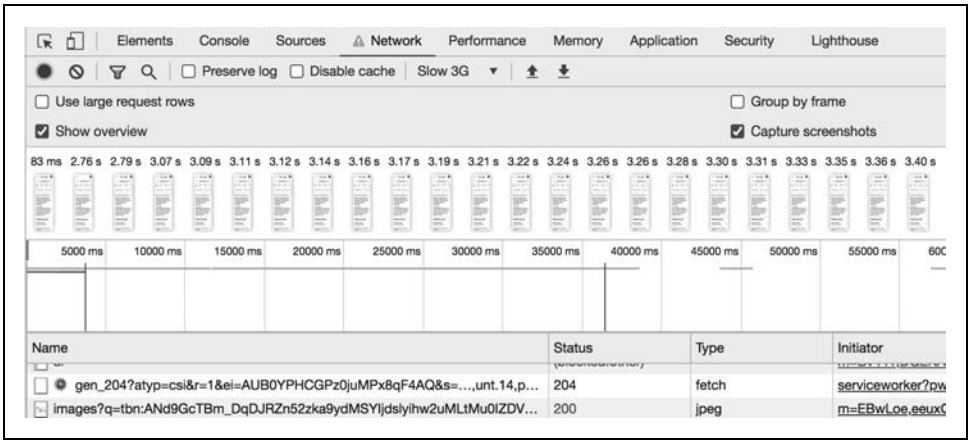
Czasami z powodu błędów w logice aplikacji strona internetowa może wykonywać wiele niechcianych wywołań API (*https://oreil.ly/wUswC*), co może wpłynąć na spowolnienie jej działania. Takie problemy da się wychwycić w zakładce *Sieć*, w lewym dolnym rogu strony, która wyświetla całkowitą liczbę żądań wysłanych z tej strony.

Zachowanie dla nowego użytkownika

Podczas wielokrotnego testowania tej samej aplikacji niektóre zasoby (na przykład ilustracje na stronach internetowych) są zapisywane w pamięci podręcznej. Zatem zmiana ilustracji podczas programowania może pozostać niezauważona. Aby wyczyścić pamięć podręczną i ponownie wyświetlić stronę, skorzystaj z pola wyboru *Wyłącz pamięć podręczną* w zakładce *Sieć*. Należy również pamiętać, że pamięć podręczna podobnie działa w przypadku docelowych użytkowników, więc ten mechanizm pomaga w eksploracji środowiska aplikacji dla użytkownika odwiedzającego stronę po raz pierwszy.

Zachowanie interfejsu użytkownika w powolnych sieciach

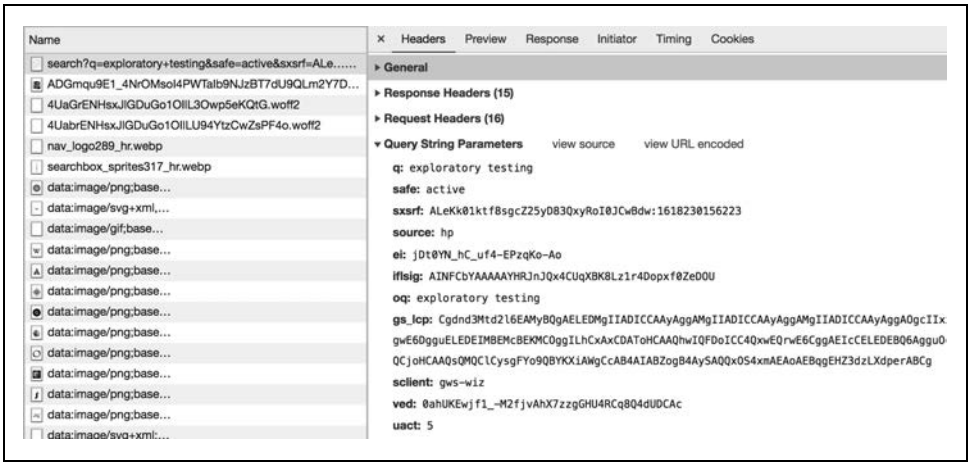
Aby poznać wrażenia docelowego użytkownika z korzystania z aplikacji w warunkach sieci o ograniczonej przepustowości, można ograniczyć przepustowość za pomocą zakładki *Sieć*, po czym obserwować zachowanie interfejsu użytkownika. Jak widać na rysunku 2.14, obok pola wyboru *Wyłącz pamięć podręczną* znajduje się rozwijane menu, które pozwala symulować warunki sieciowe 2G, 3G i 4G. Aby przetestować różne ustawienia, wybierz opcję z listy rozwijanej, wyczyść pamięć podręczną przeglądarki i ponownie załaduj stronę. Narzędzie DevTools wyświetli zbiór zrzutów ekranu, pokazanych na rysunku 2.14, przedstawiających historię stopniowego ładowania się aplikacji przy określonej przepustowości. Progresywne aplikacje webowe (omówione w rozdziale 11.) działają nawet w trybie offline, a rozwijana lista ograniczania przepustowości sieci zawiera również opcję przejścia do trybu offline w celu zweryfikowania tego zachowania.



Rysunek 2.14. Ograniczanie przepustowości sieci przy użyciu narzędzia DevTools przeglądarki Chrome

Integracja interfejsu użytkownika i interfejsu API

Zakładka *Sieć* przechwytuje wszystkie wywołania sieciowe na stronie internetowej, w tym wszystkie wywołania usług sieciowych z interfejsu użytkownika. Dla każdego żądania rejestruje nagłówki żądań i odpowiedzi (w tym tokeny uwierzytelniania), parametry zapytań, odpowiedzi i inne przydatne informacje (patrz rysunek 2.15).



Rysunek 2.15. Szczegóły żądania i odpowiedzi w zakładce Sieć

Uzyskane informacje o żądaniach (odpowiedziach) mogą być użyte do eksploracji integracji interfejsu użytkownika (interfejsu API). Na przykład można sprawdzić, czy interfejs użytkownika przekazuje poprawne parametry zapytania wprowadzone przez docelowego użytkownika oraz czy kieruje je do właściwego punktu końcowego. Ponadto można obserwować zachowanie interfejsu użytkownika dla różnych odpowiedzi z usługi. Na przykład gdy interfejs API dostępności produktów zwraca kod odpowiedzi 404, interfejs użytkownika powinien wyświetlić komunikat o błędzie „Towar niedostępny".

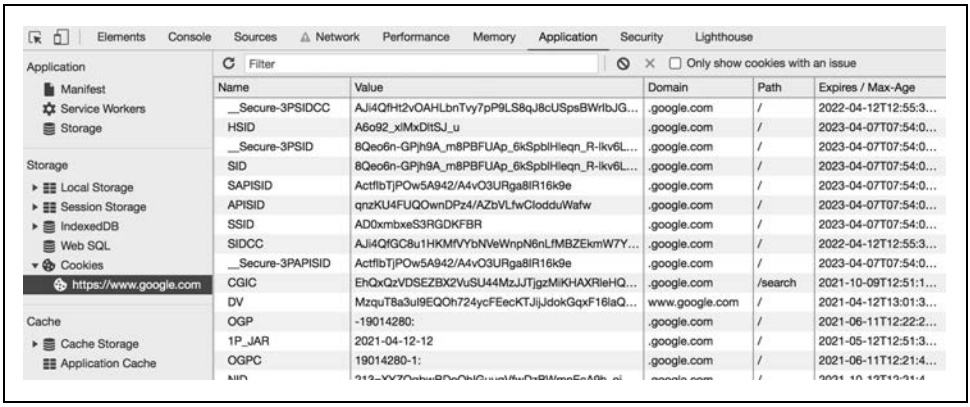
Zachowania związane z niedostępnością usługi

Gdy trzeba zasymulować przypadki testowe dla niepowodzeń żądań, można zablokować określone żądanie w zakładce *Sieć* i obserwować zachowanie interfejsu użytkownika. Dla przykładu — znajdź w zakładce *Sieć,* w wynikach wyszukiwania Google „testowanie eksploracyjne", pierwszy adres URL ładujący ilustrację, kliknij go prawym przyciskiem myszy, wybierz polecenie *Zablokuj adres URL żądania* i ponownie załaduj stronę. Spowoduje to zablokowanie ładowania w interfejsie użytkownika odpowiedniej ilustracji. Tę funkcjonalność można wykorzystać do przetestowania scenariusza „usługa wyłączona", bez faktycznego jej wyłączania.

Pliki cookie

Pliki cookie w aplikacji służą przede wszystkim do przechowywania informacji o sesji. Listę przechowywanych plików cookie wraz z dotyczącymi ich szczegółami wyświetla zakładka *Aplikacja*, co pokazano na rysunku 2.16. Podczas eksploracji możesz również z tego miejsca edytować lub usuwać wartości plików cookie i obserwować zachowanie aplikacji.

Aby uzyskać więcej szczegółowych informacji o wszystkich funkcjonalnościach narzędzia DevTools przeglądarki Chrome, odwiedź oficjalną stronę tego programu (*https://oreil.ly/J34ry*).



Rysunek 2.16. Za pomocą zakładki Aplikacje można edytować i usuwać pliki cookie

Właśnie poznałeś kilka narzędzi ułatwiających rozpoczęcie eksploracji interfejsów API i webowych interfejsów użytkownika. Ale jest jeszcze jeden kluczowy temat na drodze do osiągnięcia celów związanych z testami eksploracyjnymi: utrzymanie właściwej higieny w środowisku testowym. Omówimy to w dalszych punktach.

Perspektywy: higiena środowiska testowego

Środowisko testowe jest rzeczywistym „placem ćwiczeń", na którym testerzy wykorzystują swoje umiejętności testowania eksploracyjnego. Jego niewłaściwie utrzymywanie ma bezpośredni wpływ na testerów i uzyskiwane przez nich wyniki. Poniższa lista opisuje kilka związanych z utrzymaniem środowiska „zapachów", z którymi możesz się spotkać, oraz ich znaczenie, a także możliwe do podjęcia środki zaradcze:

Środowiska testowe współdzielone kontra dedykowane

W dużych zespołach często występuje sytuacja, kiedy jedno środowisko testowe jest współdzielone przez wiele mniejszych zespołów. Nakłada to poważne ograniczenia na zdolności testerów poszczególnych grup do wykonywania działań w środowisku w taki sposób, jaki dyktują im ich ścieżki odkrywania. Na przykład jeśli na chwilę muszą wyłączyć usługę, potrzebują zgody innych zespołów. Co gorsza, w celu przetestowania najnowszego kodu muszą skoordynować działania z innymi zespołami lub poczekać na następne zaplanowane wdrożenie, które może odbywać się raz dziennie lub raz w tygodniu. Stworzenie dedykowanego środowiska testowego, zawierającego komponenty wchodzące w zakres indywidualnych kompetencji zespołu należącego do większej grupy, da testerom swobodę wykonywania testów eksploracyjnych.

Higiena wdrażania

Gdy zespół posiada własne, dedykowane środowisko, odpowiednim podejściem jest ręczne wyzwalanie nowych wdrożeń zamiast automatyzowania ich za pośrednictwem potoku ciągłej integracji, ponieważ automatyzacja może bez ostrzeżenia zmienić konfiguracje testera w jego

środowisku. Ponadto kompilacja powinna być udostępniona do wdrożenia w potoku ciągłej integracji dopiero po pomyślnym przejściu etapu automatycznych testów. Ma to na celu uzyskanie pewności, że w najnowszym kodzie nie ma otwartych defektów, które mogłyby blokować testy eksploracyjne. Więcej informacji na temat strategii ciągłej integracji i wdrażania znajdziesz w rozdziale 4.

Ponadto środowisko testowe w ramach wdrożenia powinno być skonfigurowane w sposób jak najbardziej przypominający środowisko produkcyjne, łącznie z zaporami firewall, rozdzielonymi warstwami (komponentami), konfiguracjami limitów szybkości itp. Tylko wtedy można szczegółowo zbadać omówione wcześniej przypadki testowe awarii.

Higiena danych testowych

Dane testowe wchodzą w zakres kompetencji testerów. Aby zyskać pewność, że nieumyślnie nie wykoleją własnej eksploracji, powinni oni przestrzegać określonych praktyk. W szczególności podczas kontynuowania testowania nowej funkcjonalności w tym samym wdrożeniu należy uważać na zdezaktualizowane dane i konfiguracje. Aby uniknąć takich komplikacji, najlepiej zapewnić możliwość wdrożenia nowej kompilacji przy rozpoczynaniu każdej nowej historyjki użytkownika (z założeniem, że nowe wdrożenie usunie stare dane i konfiguracje oraz przywróci aplikację do początkowego stanu). Inną opcją jest utworzenie nowego zestawu danych testowych dla każdej historyjki użytkownika, na przykład lepiej stworzyć nowego użytkownika, zamiast eksplorować aplikację z istniejącymi użytkownikami, których dane mogą mieć różne stany.

W przypadku, gdy aplikacja zawiera setki powiązanych ze sobą tabel, tworzenie danych testowych może być skomplikowane. W takich sytuacjach nowe wdrożenie może usuwać stare dane i zastępować je standardowym zestawem danych testowych. Można również skorzystać ze skryptu SQL, który w ramach wdrożenia utworzy odpowiednie nowe dane testowe. Inna możliwość polega na anonimizacji produkcyjnych danych i wykorzystaniu ich w środowisku testowym. Jednak w przypadku niezachowania należytej staranności może to budzić obawy z punktu widzenia bezpieczeństwa.

Autonomiczne zespoły

Najczęściej dostęp do środowiska testowego jest ograniczony. Członkowie zespołu czasem nie mają poświadczeń logowania lub wymaganych uprawnień do aktualizowania konfiguracji, przeglądania logów aplikacji lub konfigurowania skrótów. Aby wykonać takie działania, muszą zwrócić się o pomoc do zespołu DevOps lub zespołu odpowiedzialnego za utrzymanie systemu. Jest to szczególnie frustrujące podczas testów eksploracyjnych, w których testerzy czasami potrzebują dostępu do wszystkich komponentów aplikacji. Zapewnienie autonomiczności zespołu oraz dostępu do wszystkiego, czego potrzebuje, zmniejszy opóźnienia spowodowane zależnościami zewnętrznymi i umożliwi płynną dostawę.

Konfiguracja usług zewnętrznych

Zazwyczaj podczas konfiguracji środowiska testowego usługi podmiotów zewnętrznych są pomijane. Przyjmuje się bowiem założenie, że integracje mogą być testowane bezpośrednio w środowisku produkcyjnym. Może to spowodować niepożądane blokady, zwłaszcza gdy problemy zostaną wykryte zbyt późno w cyklu dostarczania. Dlatego podczas konfigurowania

środowiska testowego ważne jest zadbanie o zapewnienie jakiegoś sposobu na zbadanie integracji z usługami podmiotów zewnętrznych — albo poprzez wykorzystanie zaślepek, albo poprzez opłacenie ograniczonego dostępu do tych usług.

Po omówieniu tematów związanych z tym co, dlaczego i jak w ręcznych testach eksploracyjnych, warto podkreślić, że testowanie eksploracyjne jest sztuką, która czerpie energię z analitycznych i obserwacyjnych umiejętności jednostki. Ze względu na tę indywidualistyczną naturę nie istnieje ustalony sposób walidacji wyników testów eksploracyjnych. Innymi słowy, Twój analityczny mózg może dzisiaj zainicjować ścieżkę odkrywania, a ta doprowadzi Cię do wykrycia błędów, ale wykonanie tego samego jutro może nie być możliwe. Ze względu na tę nieprzewidywalność ważne jest zachowanie zdyscyplinowanego podejścia do testów eksploracyjnych, zgodnie z pojęciami zaprezentowanymi w tym rozdziale.

Kluczowe wnioski

Oto najważniejsze wnioski z tego rozdziału:

- Ręczne testowanie eksploracyjne polega na „wędrowaniu" po aplikacji testowej z zamiarem zbadania i zrozumienia zachowania aplikacji. Takie działania mogą doprowadzić do odkrycia nowych przepływów pracy użytkowników oraz błędów w istniejących przepływach użytkowników.
- Ręczne testy eksploracyjne różnią się od testów ręcznych tym, że te drugie polegają na sprawdzaniu listy specyfikacji, podczas gdy te pierwsze opierają się na indywidualnej analizie aplikacji i umiejętnościach obserwacyjnych testera.
- Testy eksploracyjne łączą potrzeby biznesowe, implementację techniczną i perspektywę docelowego użytkownika, a jednocześnie kwestionują to, co jest uznawane za prawdę z każdego z tych punktów widzenia.
- Omówiliśmy zbiór ośmiu schematów postępowania w testach eksploracyjnych, które mogą pomóc w strukturyzacji procesów myślowych testera i opracowaniu sensownych przypadków testowych.
- Strategia ręcznego testowania eksploracyjnego kładzie nacisk na zrozumienie szczegółów aplikacji w pięciu szerokich obszarach i pozwala rozpocząć eksplorację czterech podstawowych ścieżek: funkcjonalnych przepływów pracy użytkownika, awarii i obsługi błędów, wyglądu i działania interfejsu użytkownika oraz aspektów obejmujących wiele funkcjonalności.
- Testowanie eksploracyjne powinno być procesem ciągłym. Można zaplanować powtarzanie go w różnych fazach cyklu życia dostarczania oprogramowania, na przykład testowanie *dev-box*, testowanie historyjek użytkownika, warsztaty „polowań na błędy" oraz testowanie wersji.
- Aby zbadać różne ścieżki odkrywania w aplikacji, może być niezbędne nauczenie się korzystania z nowych narzędzi. W tym rozdziale omówiłam narzędzia do testowania eksploracyjnego interfejsu API i interfejsu użytkownika, takie jak Postman, WireMock, Bug Magnet oraz Chrome DevTools.

- Środowisko testowe jest placem ćwiczeń dla ręcznych testów eksploracyjnych, a utrzymanie jego higieny ma kluczowe znaczenie dla osiągnięcia celów tych testów. W rozdziale omówiliśmy kilka typowych problemów związanych z utrzymaniem środowiska testowego i środkami zaradczymi, aby je przezwyciężyć.
- Ręczne testy eksploracyjne to wysoce zindywidualizowany proces, opierający się na umiejętnościach analitycznych i obserwacyjnych. Strukturyzacja podejścia do testów eksploracyjnych ma kluczowe znaczenie dla uproszczenia wyników.

Po pierwsze, po drugie i po trzecie: testuj swój kod!

Wysoka jakość aplikacji nie bierze się znikąd! Aby ją zapewnić, testowanie musi być integralnym aspektem inżynierii oprogramowania, wplecionym w każdy etap cyklu jego dostarczania. Błędy czy niedoskonałości w kodzie, pominięte z powodu niedokładnego testowania, mogą się okazać niezwykle kosztowne, jeśli ujawnią się w środowisku produkcyjnym. Oznacza to, że wdrożenie mądrej strategii testowania jest warunkiem sukcesu w branży dostarczania oprogramowania.

Ten praktyczny przewodnik zawiera szeroki przegląd strategii, wzorców i form testowania oprogramowania, ułatwiających dobór ścieżek i podejść do konkretnych projektów w zależności od zakresu, budżetu i ram czasowych. Książka uczy przydatnych umiejętności w zakresie przeprowadzania testów wydajności, bezpieczeństwa i dostępności, w tym testów eksploracyjnych, automatyzacji testów, testów wielofunkcyjnych, testowania danych, testowania mobilnego i wielu innych. Zaprezentowano tu także łączenie testów w potokach ciągłej integracji, co pozwala na szybkie otrzymywanie informacji zwrotnych. W ten sposób łatwiejsze staje się kontrolowanie trudnych przepływów pracy programistycznej i uzyskiwanie wysokiej jakości aplikacji.

Najciekawsze zagadnienia:

- ponad 40 narzędzi do testowania kodu
- zasady dobierania testów pod kątem jakości oprogramowania
- strategie i koncepcje dotyczące testowania
- praktyczne korzystanie z poszczególnych narzędzi do testowania
- najlepsze praktyki przeprowadzania testów

Gayathri Mohan jest główną konsultantką w firmie Thoughtworks, wcześniej zarządzała w niej dużymi zespołami walidacyjnymi. Ma bogate doświadczenie w tworzeniu oprogramowania dla różnych branż przemysłu. Jako liderka techniczna jest organizatorką wielu wydarzeń cieszących się dużą popularnością wśród inżynierów.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-015-4
9 788383 220154
Cena: 89,00 zł