Helion

<packt>

Automatyzacja metodyki DevOps za pomocą potoków CI/CD GitLaba

Buduj efektywne potoki CI/CD do weryfikacji, zabezpieczania i wdrażania kodu, korzystając z rzeczywistych przykładów

CHRISTOPHER COWELL
NICHOLAS LOTZ
CHRIS TIMBERLAKE

Tytuł oryginału: Automating DevOps with GitLab CI/CD Pipelines: Build efficient CI/CD pipelines to verify, secure, and deploy your code using real-life examples

Tłumaczenie: Łukasz Wójcicki

ISBN: 978-83-289-0775-1

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

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

CZĘŚĆ 2. Automatyzacja etapów DevOps przy użyciu potoków CI/CD GitLaba

CZĘŚĆ 3. Następne kroki w doskonaleniu aplikacji za pomocą GitLaba

Zrozumienie okresu przed DevOps

Rozdział 1

Aby docenić potęgę **potoków CI/CD** i **metodyki DevOps** w rozwoju oprogramowania, musimy zrozumieć, w jaki sposób tworzono oprogramowanie przed pojawieniem się narzędzi takich jak GitLab. Choć w tym rozdziale nie zdobędziesz żadnej praktycznej wiedzy, to dowiesz się, jak powstały potoki GitLab CI/CD i uzyskasz jasny obraz problemów, jakie potoki GitLab CI/CD rozwiązują. Posiadanie pojęcia o tych kwestiach pozwoli Ci zrozumieć, dlaczego potoki GitLab CI/CD działają tak, jak działają, i otworzy Ci oczy na niesamowitą moc, jaką przynoszą one cyklowi życia oprogramowania. Krótko mówiąc, najlepszym sposobem na zrozumienie, jak teraz funkcjonujemy, jest zrozumienie, jak źle było wcześniej!

W tym rozdziale zostaniesz wprowadzony do fikcyjnej, ale realistycznej aplikacji internetowej o nazwie *Hats for Cats+*, która sprzedaje — zgadłeś — nakrycia głowy dla kotów. Dowiesz się szybko, co obejmuje proces przekształcania aplikacji od pomysłu w dobrze napisaną, przetestowaną i wdrożoną aplikację internetową. Zobaczysz, jak te zadania musiałyby być wykonywane w świecie, w którym nie istnieją potoki CI/CD, aby korzyści płynące z GitLaba były jeszcze bardziej oczywiste, gdy dowiesz się o nich w późniejszych rozdziałach.

W tym rozdziale omówimy następujące główne tematy:

- wprowadzenie do aplikacji internetowej *Hats for Cats*;
- ręczne tworzenie i weryfikacja kodu;
- ręczne przeprowadzanie testów bezpieczeństwa kodu;
- ręczne pakowanie i wdrażanie kodu;
- problemy z ręcznymi praktykami w cyklu życia oprogramowania;
- rozwiązywanie problemów za pomocą metodyki DevOps.

Wprowadzenie do aplikacji internetowej Hats for Cats

Hats for Cats to fikcyjna aplikacja internetowa służąca do sprzedaży czapek baseballowych, kapeluszy kowbojskich i meloników dla Twoich ulubionych futrzanych przyjaciół. Wyobraź sobie, że to standardowy sklep internetowy, podobny do setek lub tysięcy innych, z których korzystałeś. Pozwala użytkownikom przeglądać katalog kapeluszy, dodawać przedmioty do koszyka i wprowadzać informacje dotyczące płatności i dostawy.

Doświadczenie użytkownika lub graficzny design aplikacji nie ma znaczenia w kontekście tej książki. Framework aplikacji internetowej, na którym opiera się ta aplikacja, również nie ma znaczenia. Nawet język programowania, w którym jest napisana, też nie jest ważny. Powtórzę to, ponieważ jest to istotny, choć być może zaskakujący punkt: *ta książka jest bezstronna językowo*. Zawiera przykłady w kilku językach programowania, aby zwiększyć szanse, że przynajmniej niektóre z nich będą w tym, który jest Ci znany. Ale to, czy Twoje aplikacje lub aplikacja internetowa *Hats for Cats* są napisane w Javie, JavaScript, Pythonie, Ruby czy w innym języku, nie ma znaczenia. Ogólne zasady potoków CI/CD, opisane w tej książce, mają zastosowanie bez względu na to.

To, co ma znaczenie, to ogólne kroki, jakie musisz podjąć, aby upewnić się, że kod jest wysokiej jakości, zachowuje się zgodnie z oczekiwaniami, jest bezpieczny, ma odpowiednią wydajność, jest sensownie spakowany i wdrażany w odpowiednich środowiskach we właściwych momentach. Ta książka skupia się na tym, jak potoki GitLab CI/CD mogą ułatwić różne etapy **cyklu życia oprogramowania** (**SDLC** — ang. *software development life cycle*), czyniąc je prostszymi, szybszymi i bardziej niezawodnymi. Nie pokaże Ci, jak napisać aplikację internetową. Zakłada się, że całe programowanie odbywa się w tle, po czym zostanie pokazane, jak zbudować, zweryfikować, zabezpieczyć, spakować i wdrożyć ten kod.

Mając to na uwadze, przeanalizujmy ogólne kroki, które musiałbyś podjąć, aby przygotować swój kod dla użytkowników żyjących w świecie przed pojawieniem się GitLaba. Są to wszystkie manualne odpowiedniki tego, co potoki CI/CD mogą robić dla Ciebie automatycznie. Jednak zrozumienie ograniczeń procesów manualnych oraz trudności i monotonii związanej z ich stosowaniem pomoże Ci zrozumieć prawdziwą moc GitLaba.

Ręczne tworzenie i weryfikacja kodu

Zanim pojawiły się potoki CI/CD, konieczne było ręczne budowanie i weryfikowanie kodu. Było to często okropne doświadczenie z powodów, które tutaj opiszemy.

Ręczne budowanie kodu

Proces budowy kodu zależy od języka, w którym piszesz. Jeśli używasz języka interpretowanego, takiego jak Python czy Ruby, to procesu budowania nie będzie. Ale jeśli piszesz w języku kompilowanym, będziesz musiał zbudować swoją aplikację, kompilując jej kod źródłowy.

Wyobraź sobie, że korzystasz z języka Java. Oto tylko kilka różnych sposobów na kompilowanie kodu źródłowego Java na wykonywalne klasy Java:

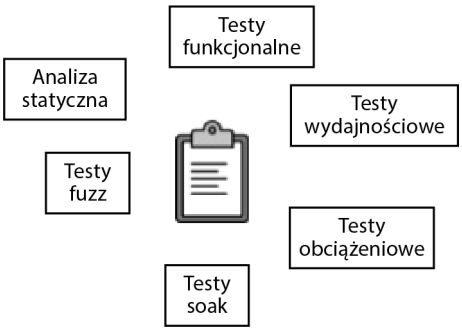
- Możesz użyć kompilatora `javac`, który jest dostarczany wraz z Java Development Kit.
- Możesz użyć narzędzia budowania Maven.
- Możesz użyć narzędzia budowania Gradle.

Istnieje wiele powodów, dla których ten proces ręcznego budowania jest uciążliwym i irytującym zadaniem, które większość programistów chętnie by zostawiła za sobą:

- Jest narażony na błędy użytkownika: ile razy zapomniałeś, czy powinieneś wskazywać kompilatorowi `javac` pakiet najwyższego poziomu, w którym znajdują się Twoje klasy, czy też poszczególne pliki klas?
- Jest wolny, trwa od kilku sekund do kilku minut, w zależności od tego, jak duża jest Twoja aplikacja. To może prowadzić do długiego przestoju.
- Łatwo o zapomnienie o czymś, co może powodować zamieszanie, gdy przypadkowo uruchomisz stary kod, który nie działa tak, jak myślałeś.
- Źle napisany kod może się nie skompilować, co sprawia, że wszyscy tracą czas, gdy inżynier budowy aplikacji odsyła kod do programistów w celu wprowadzenia poprawek i oczekuje na ich dostarczenie.

Ręczna weryfikacja kodu

Gdy już zbudujesz swój kod, musisz zweryfikować, czy działa poprawnie. Testowanie przyjmuje nieskończenie wiele form, a rodzajów testów jest więcej, niż moglibyśmy opisać w tej książce. Oto jednak niektóre z najczęstszych form testów, którym możesz poddać swój kod:



Rysunek 1.1. Testy weryfikujące kod

Testy funkcjonalne

Czy Twój program robi to, co powinien? Na to pytanie odpowiadają testy funkcjonalne. Większość projektów programistycznych zaczyna się od specyfikacji opisującej, jak oprogramowanie powinno działać: mając określone wejście, jakie wyjście powinno się pojawić? Programiści uważają swoją pracę za zakończoną, gdy kod, który napisali, odpowiada tym specyfikacjom. Jak się dowiedzą, że ich kod jest zgodny? Właśnie w tym miejscu pojawiają się testy funkcjonalne.

Podobnie jak ogólnie istnieje wiele form testowania, istnieje wiele podkategorii testowania, które razem tworzą testy funkcjonalne.

Testowanie ścieżki **Happy path** upewnia, że program działa zgodnie z oczekiwaniami, gdy otrzymuje powszechne, poprawne dane wejściowe. Na przykład, jeśli wprowadzisz 2 + 2 do kalkulatora, lepiej, żeby zwrócił 4! Testy *Happy path* wydają się najważniejszym

rodzajem testów, ponieważ sprawdzają zachowanie, z którym użytkownicy najprawdopodobniej się spotkają podczas korzystania z oprogramowania. W rzeczywistości jednak najczęstsze przypadki użycia można zazwyczaj objąć tylko kilkoma tego typu testami. Testy, które obejmują nietypowe lub niespodziewane przypadki, zwykle są znacznie liczniejsze.

Skoro już mówimy o nietypowych przypadkach, właśnie tutaj pojawia się **testowanie „przypadków brzegowych"** (ang. *edge-case testing*). Jeśli wyobrażasz sobie spektrum wartości wejściowych, większość wartości, które użytkownicy wprowadzą, znajdzie się w środku tego spektrum. Na przykład użytkownicy kalkulatora są bardziej skłonni wprowadzać coś takiego jak 56 : 209 (te wartości znajdują się w środku zakresu wartości, które kalkulator akceptuje) niż np. 0 + 0 lub 999 999 – 999 999 (ponieważ te wartości są na krawędziach zakresu). Testowanie „krawędziowych przypadków" daje gwarancję, że wartości wejściowe na skrajnych krawędziach akceptowalnego spektrum nie popsują Twojego oprogramowania. Czy można stworzyć nazwę użytkownika, która składa się z jednej litery? Czy można zamówić 9 999 kopii książki? Czy można wpłacić 1 centa na konto bankowe? Jeśli specyfikacje mówią, że Twoje oprogramowanie powinno sobie radzić z tymi skrajnymi przypadkami, lepiej się upewnić, że naprawdę to potrafi!

Jeśli testowanie „krawędziowych przypadków" daje pewność, że Twoje oprogramowanie potrafi obsłużyć wartość wejściową mieszczącą się na granicy akceptowalnych wartości, to **testowanie *corner-case*** potwierdza, że Twoje oprogramowanie potrafi obsłużyć dwa lub więcej jednoczesnych przypadków krawędziowych. Możesz to sobie wyobrazić jako testowanie „krawędziowych przypadków" w trybie turbo, które stawia oprogramowanie w jeszcze bardziej niewygodnych (ale nadal poprawnych) sytuacjach. Na przykład, czy Twoja aplikacja bankowa pozwala na zaplanowanie wypłaty najmniejszej poprawnej kwoty w najdalszej poprawnej przyszłej dacie? Nie trzeba ograniczać testowania *corner-case* do dwóch wartości wejściowych: jeśli Twoje oprogramowanie akceptuje trzy, dziesięć lub sto wartości wejściowych naraz, musisz się upewnić, że działa, gdy każde wejście zostanie przesunięte aż do skrajnej krawędzi zakresu wartości zgodnych ze specyfikacją.

To obejmuje przypadki, w których oprogramowanie otrzymuje poprawne wartości. Ale czy musisz się także upewnić, że zachowuje się poprawnie, gdy otrzymuje *nieprawidłowe* wartości? Oczywiście tak! Ten rodzaj testowania nazywany jest czasami testowaniem ***unhappy path*** i zwykle jest znacznie ciekawszy dla testerów, ponieważ bardziej prawdopodobne jest odkrycie błędów. Całe oprogramowanie musi odpowiednio obsługiwać niespodziewane, nieprawidłowe lub źle sformułowane dane, a Ty potrzebujesz testów, które to udowodnią. Wracając do wcześniejszych przykładów — musisz się upewnić, że Twój kalkulator się nie zawiesi, gdy poprosisz go o podzielenie liczby przez zero. Musisz sprawdzić, czy aplikacja bankowa przypadkiem nie wpłaci Ci depozytu, gdy poprosisz o wypłatę –6 dolarów. A Twój program do konwersji walut powinien dostarczyć sensowną wiadomość o błędzie, gdy zapytasz o kurs wymiany walut z 31 lutego 2020 roku.

Ponieważ zazwyczaj istnieje więcej sposobów na wprowadzenie błędnych niż poprawnych danych do aplikacji, programiści często skupiają się na poprawnym przetwarzaniu *oczekiwanych* danych, ale nie zastanawiają się nad rodzajami nieoczekiwanych, zniekształconych lub znajdujących się poza zakresem danych, które mogą wprowadzić użytkownicy. Programy muszą przewidywać i poprawnie obsługiwać *różne* rodzaje danych

— zarówno dobrych, jak i złych. Pisanie kompletnych zestawów testów zarówno *happy path*, jak i *unhappy path* to najlepszy sposób, aby upewnić się, że programista napisał kod, który zachowuje się dobrze, niezależnie od danych, które użytkownik wprowadza.

To są pewne rodzaje zachowań, które obejmują nie tylko poprawne, lecz także niepoprawne dane, które testy mogą sprawdzać. Ale istnieje jeszcze inny wymiar, który można wykorzystać do kategoryzowania testów: *rozmiar fragmentu kodu*, który test ma na celu.

W większości przypadków najmniejszą częścią kodu, którą test może sprawdzić, jest pojedyncza metoda lub funkcja. Na przykład możesz chcieć przetestować funkcję o nazwie *alphabetize*, która przyjmuje dowolną liczbę ciągów znaków jako dane wejściowe i zwraca te same ciągi, ale w porządku alfabetycznym. Aby przetestować tę funkcję, prawdopodobnie użyłbyś **testu jednostkowego** (ang. *unit test*). Testuje on pojedynczą *jednostkę* kodu — w tym przypadku jednostką jest pojedyncza funkcja. Możesz mieć kolekcję kilku testów jednostkowych, które obejmują tę funkcję, choć w różny sposób:

- Niektóre mogą obejmować testy *happy path*. Na przykład mogą przekazywać jako dane wejściowe ciągi `dog`, `cat` i `mouse`.
- Niektóre mogą obejmować przypadki krawędziowe lub *corner*. Na przykład mogą przekazywać funkcji pojedynczy pusty ciąg, ciągi składające się tylko z cyfr lub ciągi już uporządkowane alfabetycznie.
- Niektóre mogą obejmować testy *unhappy path*. Na przykład mogą przekazywać funkcji niespodziewany typ danych, taki jak wartości logiczne, zamiast oczekiwanego typu danych — ciągów znaków.

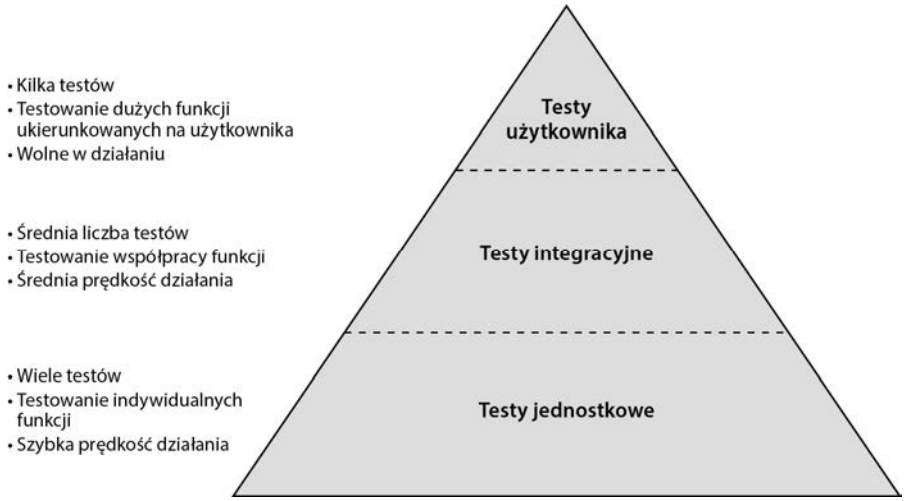
Aby zweryfikować zachowanie większych fragmentów kodu, można użyć **testów integracyjnych** (ang. *integration tests*). Te testy nie skupiają się na pojedynczych funkcjach, ale na tym, jak grupy funkcji współdziałają ze sobą. Na przykład wyobraź sobie, że Twoja aplikacja do konwersji walut ma cztery funkcje:

- `get_input`, która pobiera dane wejściowe od użytkownika w formie waluty źródłowej, kwoty źródłowej i waluty docelowej;
- `convert`, która konwertuje tę kwotę waluty źródłowej na odpowiednią kwotę waluty docelowej;
- `print_output`, która informuje użytkownika, ile waluty docelowej zostanie uzyskane w wyniku konwersji;
- `main`, która jest głównym punktem wejścia do Twojej aplikacji. To jest funkcja, która jest wywoływana podczas użytkowania Twojej aplikacji. Wywołuje ona trzy inne funkcje i przekazuje wynik każdej funkcji jako dane wejściowe do kolejnej.

Aby się upewnić, że te funkcje współdziałają ze sobą — czyli sprawdzić, czy się *integrują* — potrzebujesz testów integracyjnych, które wywołują funkcję `main`, w przeciwieństwie do testów jednostkowych, które wywołują `get_input`, `convert` i `print_output`. To pozwala przeprowadzać testy na wyższym poziomie abstrakcji, czyli na poziomie, który jest bliżej tego, w jaki sposób prawdziwy użytkownik korzystałby z Twojej aplikacji. W końcu użytkownik nie będzie wywoływał funkcji `get_input` w izolacji. Zamiast tego będzie wywoływać `main`, która z kolei wywoła trzy inne funkcje i skoordynuje przekazywanie wartości między nimi. Łatwo jest napisać funkcję, która działa zgodnie z oczekiwaniami

samodzielnie, ale trudniej jest sprawić, aby kolekcja funkcji współpracowała w celu zbudowania większego elementu logiki. Testy integracyjne wykrywają ten rodzaj problemu w taki sposób, którego czyste testy jednostkowe nie potrafią.

Testerzy często myślą o różnych rodzajach testów jak o budowie piramidy. W tym modelu testy jednostkowe zajmują szeroką, niską podstawę piramidy: są one niskopoziomowe, ponieważ testują podstawowe fragmenty kodu, a jest ich wiele. Testy integracyjne zajmują środek piramidy: działają na wyższym poziomie abstrakcji niż testy jednostkowe i jest ich mniej. Na szczycie piramidy znajduje się trzecia kategoria testów, o której teraz porozmawiamy — testy użytkownika.



Rysunek 1.2. Piramida testów

Ostatni rodzaj testów, czyli testy użytkownika, symulują zachowanie użytkownika i testują oprogramowanie tak, jak robiłby to użytkownik. Na przykład, jeśli użytkownicy wchodzą w interakcję z aplikacją do wymiany walut, wprowadzają walutę źródłową, ilość waluty źródłowej i walutę docelową, a następnie oczekują zobaczenia wyniku w postaci ilości waluty docelowej, to dokładnie to zrobi test użytkownika. Może to oznaczać, że używa interfejsu graficznego aplikacji, klikając na przyciski i wprowadzając wartości w pola. Albo może to oznaczać, że wywołuje punkty końcowe interfejsu API REST aplikacji, przekazując wartości wejściowe i sprawdzając wynik dla wartości wyjściowej. Niezależnie od tego, jak oddziałuje z aplikacją, robi to w sposób jak najbardziej zbliżony do prawdziwego użytkownika. Podobnie jak w przypadku testów jednostkowych i integracyjnych, testy użytkownika mogą obejmować testy *happy path*, *corner* oraz *unhappy path*, aby uwzględnić wszystkie scenariusze opisane w specyfikacjach oprogramowania, a także wszelkie inne scenariusze, które projektant testów może stworzyć.

Jak do tej pory, wyjaśniliśmy różne cele testów jednostkowych, integracyjnych i użytkownika, ale jeszcze nie opisaliśmy fundamentalnej różnicy. Testy jednostkowe i integracyjne są niemal zawsze zautomatyzowane. Oznacza to, że są to programy komputerowe testujące inne programy komputerowe. Testy użytkownika są automatyzowane, jeśli to tylko możliwe, ale pojawia się wystarczająco dużo trudności związanych z pisaniem

niezawodnych i reprodukowalnych testów, które oddziałują z interfejsem graficznym aplikacji, że wiele testów użytkownika musi być uruchamianych ręcznie. Aplikacje internetowe są niezwykle trudne do przetestowania ze względu na nieprzewidywalne zachowanie w czasie ładowania, niekompletne renderowanie stron, brakujące lub niekompletnie wczytane pliki CSS oraz przeciążenie sieci. Oznacza to, że chociaż zespoły tworzenia oprogramowania często próbują zautomatyzować testy użytkownika dla aplikacji internetowych, to często kończy się to hybrydą zautomatyzowanych i ręcznych testów użytkownika. Jak się pewnie domyślasz, ręczne testy użytkownika są niezwykle kosztowne zarówno jeśli chodzi o czas, jak i o morale testerów.

Testy wydajności

Po tej ogólnokształcącej podróży po teście funkcjonalnym możesz myśleć, że omówiliśmy już wszystkie obszary testów. Ale dopiero zaczynamy. Innym aspektem Twojej aplikacji, który prawdopodobnie powinien być przetestowany, jest jej wydajność: czy robi to, co powinna, wystarczająco szybko, aby użytkownicy nie byli sfrustrowani? Czy spełnia wymogi dotyczące specyfikacji wydajności, które deweloperzy mogli dostać przed rozpoczęciem kodowania? Czy wydajność aplikacji jest znacznie lepsza lub gorsza niż wydajność konkurentów? To są pytania, na które mają odpowiedzieć testy wydajności.

Testowanie wydajności jest niezwykle trudne do zaprojektowania i przeprowadzenia. Istnieje wiele zmiennych do uwzględnienia podczas mierzenia, jak szybko działa twoja aplikacja:

- W jakim środowisku powinna działać podczas testów? Tworzenie środowiska identycznego z produkcyjnym jest często zbyt kosztowne, ale na jakie skróty można pójść w środowisku testowym, aby nie zniekształcić wyników testów wydajności?
- Jakie wartości wejściowe powinny być używane w testach wydajności? W zależności od aplikacji, przetwarzanie niektórych wartości wejściowych może zajmować znacznie więcej czasu niż innych.
- Jeśli twoja aplikacja jest konfigurowalna, jakich ustawień konfiguracji powinieneś użyć? To jest szczególnie ważne, jeśli nie ma standardowej konfiguracji, na którą większość użytkowników się decyduje.

Nawet jeśli uda ci się wymyślić, jak zaprojektować przydatne testy wydajności, często zajmują one dużo czasu i w niektórych przypadkach dają niejednorodne wyniki. To prowadzi do częstego ponawiania tych testów, co jeszcze bardziej je wydłuża. Testy wydajności należą więc do najważniejszych, ale też najdroższych rodzajów testów.

Testy obciążeniowe

Testy wydajności mają bliskiego krewnego zwanego **testami obciążeniowymi**. Podczas gdy testy wydajności określają, jak szybko Twoje oprogramowanie może wykonać jedną operację (np. pojedynczą konwersję waluty, pojedynczy depozyt bankowy lub pojedyncze zadanie arytmetyczne), testy obciążeniowe określają, jak dobrze Twoja aplikacja sobie radzi, gdy wielu użytkowników oddziałuje na nią jednocześnie. Testy obciążeniowe borykają się z wieloma trudnościami projektowymi, podobnie jak testy wydajności, i mogą dawać równie niejednolite wyniki. Mogą być jeszcze bardziej kosztowne do skonfigurowania, ponieważ wymagają symulowania setek lub tysięcy użytkowników.

Testy soak

Kiedy Twoja aplikacja działa przez godziny lub dni, czy alokuje pamięć, której nigdy nie odzyskuje? Czy zużywa ogromne ilości przestrzeni dyskowej z nadmiernym zapisywaniem? Czy uruchamia procesy w tle, które nigdy nie są wyłączane? Jeśli cierpi na którekolwiek z tych wycieków zasobów, może to spowodować utratę wydajności lub nawet awarię w miarę kończenia się pamięci, przestrzeni dyskowej lub przeznaczonych na to cyklów CPU. Te problemy można wykryć za pomocą **testów *soak***, które po prostu testują Twoje oprogramowanie przez długi okres, jednocześnie monitorując jego stabilność i wydajność. Oczywiście testy *soak* są nadzwyczaj kosztowne pod względem czasu i zasobów sprzętowych, które trzeba poświęcić na ich przeprowadzenie i monitorowanie.

Testy fuzz

Mało wykorzystywany, ale potężny rodzaj testowania nosi nazwę **testowania *fuzz***. To podejście wysyła do Twojego oprogramowania poprawne, ale dziwaczne dane wejściowe, aby ujawnić błędy, które tradycyjne testy funkcjonalne mogły przeoczyć. Można to porównać do testowania *happy path* w stanie upojenia. Zamiast próby stworzenia konta z nazwą użytkownika „Sam", spróbuj użyć nazwy użytkownika składającej się z 1000 liter. Lub spróbuj stworzyć nazwę użytkownika, która składa się wyłącznie z odstępów. Albo dołącz znaki Unicode z alfabetu klingońskiego w adresie wysyłki.

Testowanie *fuzz* wprowadza silny element losowości: wartości wejściowe, które wysyła do Twojego oprogramowania, albo są całkowicie losowo generowane, albo są losowymi permutacjami wartości wejściowych, które są znane jako nieproblemowe dla Twojego kodu. Na przykład, jeśli Twój kod przekształca pliki PDF na pliki HTML, test *fuzz* może zacząć od wysyłania lekko zmienionych wersji poprawnych, łatwo obsługiwanych plików PDF, a następnie przejść do proszenia Twojego oprogramowania o konwersję zupełnie losowych ciągów znaków, które w ogóle nie przypominają plików PDF. Ponieważ test *fuzz* może wysyłać wiele tysięcy losowych wartości wejściowych, zanim natrafi na wartość wejściową, która spowoduje awarię lub inny błąd, testy *fuzz* muszą być zautomatyzowane. Są one po prostu zbyt niewygodne do ręcznego uruchamiania.

Statyczna analiza kodu

Innym ściśle zautomatyzowanym rodzajem testowania jest **analiza statycznego kodu**. W przeciwieństwie do innych wspomnianych już testów, które próbują znaleźć problemy w Twoim kodzie w trakcie jego działania, analiza statycznego kodu sprawdza Twój kod źródłowy bez jego wykonywania. Może szukać różnych problemów, ale ogólnie rzecz biorąc, sprawdza, czy stosujesz uznaną praktykę kodowania i idiomy języka. Mogą to być praktyki ustalone przez Twój zespół, przez twórców języka lub przez inne autorytety programistyczne.

Na przykład analiza statycznego kodu pozwoli zauważyć, że deklarujesz zmienną bez przypisywania jej wartości. Albo może wskazać, że przypisałeś wartość zmiennej, ale nigdy nie odnosisz się do tej zmiennej. Może zidentyfikować niedostępny kod, kod korzystający z wzorców kodowania, które są wolniejsze niż alternatywne, ale funkcjonalnie równoważne wzorce, lub kod korzystający z odstępów w nietypowy sposób. To wszystkie

praktyki, które może nie spowodują bezpośredniego przerwania kodu, ale mogą sprawić, że Twój kod nie będzie tak czytelny, łatwy w utrzymaniu lub szybki, jak mógłby być.

Dodatkowe wyzwania związane z weryfikacją kodu

Dotychczas opisaliśmy tylko niektóre sposoby weryfikowania zachowania, wydajności i jakości Twojego kodu. Ale po zakończeniu wszystkich tych testów różnego rodzaju stajesz przed potencjalnie trudnym pytaniem o to, jak przeanalizować, przetworzyć i raportować wyniki. Jeśli masz szczęście, narzędzia do testowania generują raporty w standardowym formacie, który można zintegrować z automatycznie aktualizowanym panelem sterowania. Ale prawdopodobnie będziesz musiał korzystać przynajmniej z jednego narzędzia lub frameworku do testowania, którego nie da się łatwo wkomponować w normalną strukturę raportowania i które trzeba będzie ręcznie przeskanować, oczyścić i doprowadzić do czytelnej i łatwo przyswajalnej formy.

Już wspomnieliśmy, że zwłaszcza testy wydajnościowe często wymagają wielokrotnego uruchamiania. W rzeczywistości jednak *wszystkie* testy muszą być uruchamiane wielokrotnie, aby wykrywać regresje lub wygładzać tzw. migające testy, które czasami kończą się pomyślnie, a czasami nie, w zależności od warunków sieciowych, obciążenia serwera lub niezliczonych innych nieprzewidywalnych czynników. Oznacza to, że obciążenie związane z ręcznym uruchamianiem testów lub zarządzaniem i uruchamianiem testów automatycznych jest znacznie większe, niż się to początkowo wydaje. Jeśli planujesz uruchamiać testy wielokrotnie, musisz ustalić, kiedy i jak często to robić; powinieneś się też upewnić, że odpowiedni sprzęt lub środowiska testowe są dostępne we właściwych momentach. Musisz być również wystarczająco elastyczny, aby zmienić rytm testowania, gdy zmieniają się warunki lub gdy zarząd prosi o bardziej aktualne wyniki. Wynika z tego morał, że testowanie jest trudne, czasochłonne i podatne na błędy, a wszystkie te trudności są wyolbrzymiane za każdym razem, gdy ludzie muszą się zaangażować w upewnianie się, że testy są przeprowadzane we właściwy sposób we właściwym czasie.

Mimo że właśnie powiedzieliśmy, że testy zazwyczaj *powinny* być uruchamiane, a następnie wielokrotnie ponawiane, istnieje jeszcze jeden problem. Ponieważ wykonywanie testów jest kosztowne i trudne, istnieje tendencja do uruchamiania ich tak rzadko, jak to tylko możliwe. Ta tendencja jest wprowadzana przez powszechny model rozwoju, w którym programiści tworzą funkcję (lub czasami cały produkt), a następnie *przekazują kod zespołowi* **Kontroli Jakości** (**QA** — ang. *quality assurance*) w celu walidacji. Ścisłe podziały między tworzeniem a testowaniem kodu oznaczają, że w wielu zespołach testy są uruchamiane tylko pod koniec cyklu rozwojowego — pod koniec dwutygodniowego sprintu, pod koniec rocznego projektu lub gdzieś pośrodku.

Praktyka rzadkiego lub opóźnionego testowania prowadzi do ogromnego problemu: kiedy programiści przekazują duży pakiet kodu do testowania — tysiące lub dziesiątki tysięcy linii kodu, które zostały opracowane przez różne osoby, korzystające z różnych stylów kodowania i idiomów przez tygodnie lub miesiące — zdiagnozowanie źródła wszelkich błędów, które testy ujawniają, może być niezwykle trudne. To z kolei oznacza, że trudno jest naprawić te błędy. Podobnie jak duże stogi siana skuteczniej ukrywają

igły niż małe stogi siana, duże partie kodu utrudniają znalezienie, zrozumienie i poprawienie wszelkich błędów, które mogą zawierać. Im dłużej zespół rozwojowy czeka, zanim przekaże kod zespołowi QA, tym większy staje się ten problem.

To kończy nasz błyskawiczny przegląd testów funkcjonalnych, testów obciążeniowych, testów *soak*, testów *fuzz* i analizy statycznego kodu. Ponadto wyjaśniliśmy niektóre ukryte trudności związane z przeprowadzaniem wszystkich tych różnych rodzajów testów. Możesz się zastanawiać, dlaczego w ogóle rozmawialiśmy o testowaniu. Powodem jest to, że zrozumienie wyzwań testowania — nabranie wyczucia dla liczby sposobów weryfikacji kodu, zrozumienie ważności różnych form testów, zdawanie sobie sprawy z czasochłonności konfigurowania środowisk testowych, uciążliwości ręcznego wykonywania testów użytkownika, trudności w przetwarzaniu i raportowaniu wyników oraz trudności w znajdowaniu i naprawianiu błędów ukrytych w dużym pakiecie kodu — ułatwi zrozumienie, jak trudne było tworzenie oprogramowania przed pojawieniem się DevOps. W dalszej części tej książki, gdy zobaczysz, jak potoki CI/CD GitLaba upraszczają proces uruchamiania różnych rodzajów testów i przeglądania ich wyników, oraz gdy zrozumiesz, jak testy uruchamiane wcześnie i często ułatwiają wykrywanie problemów i są tańsze w naprawie, będziesz mógł spojrzeć wstecz na te kłopotliwe procedury testowe i poczuć sympatię dla biednych programistów, którzy musieli przemierzać tę część cyklu życia oprogramowania przed pojawieniem się GitLaba. Życie jest o wiele lepsze w jego erze!

Ręczne przeprowadzanie testów bezpieczeństwa kodu

Wspomnieliśmy, że testowanie funkcjonalne to tylko jedna forma testowania. Kolejną ważną formą jest **testowanie zabezpieczeń**. Jest ono tak istotne i trudne do wykonania, że zazwyczaj przeprowadzają je specjalistyczne zespoły, oddzielone od tradycyjnych działów QA. Istnieje wiele sposobów podejścia do testowania zabezpieczeń, ale większość sprowadza się do jednej z trzech kategorii, którymi są:

- inspekcja kodu źródłowego,
- interakcja z działającym kodem,
- inspekcja zależności firm trzecich używanych przez Twój projekt.

Ponadto istnieją różne rodzaje problemów, które mogą badać testy zabezpieczeń. Na pierwszy rzut oka niektóre z tych problemów mogą nie dotyczyć obszaru bezpieczeństwa, ale wszystkie przyczyniają się do potencjalnej utraty danych lub manipulacji oprogramowaniem przez złośliwych użytkowników. Są to:

- nietypowe praktyki kodowania,
- niebezpieczne praktyki kodowania,
- zależności od kodu źródłowego zawierającego znane podatności.

Przejrzyjmy niektóre konkretne odmiany testów zabezpieczeń i zobaczmy, jak korzystają z różnych technik, aby szukać problemów różnego rodzaju.

Statyczna analiza kodu

Nierzadko można znaleźć niebezpieczne praktyki kodowania, po prostu prosząc eksperta ds. zabezpieczeń o przejrzenie Twojego kodu źródłowego. Na przykład, jeśli pobierasz dane od użytkownika, a następnie używasz tych danych do zapytania bazy danych, sprytny użytkownik może przeprowadzić tzw. atak *SQL injection*, wprowadzając polecenia bazodanowe w swoich danych wejściowych. Kompetentni recenzenci kodu szybko zauważą ten rodzaj problemu i często mogą zaproponować łatwe do wdrożenia rozwiązania.

Na przykład poniższy pseudokod akceptuje dane od użytkownika, ale nie sprawdza ich poprawności przed użyciem w poleceniu SQL. Sprytny użytkownik mógłby wprowadzić złośliwą wartość, na przykład `Smith OR (0 = 0)`, co spowodowałoby ujawnienie większej ilości informacji, niż zamierzał deweloper:

```
employee_name = get_user_input()
sql = "SELECT salary FROM employee_records WHERE employee_name = $employee_name"
call_database(sql)
```

Przeglądy kodu również mogą pomóc zidentyfikować kod, który może nie wydaje się oczywiście niebezpieczny, ale używa nietypowych idiomów, nietypowego formatowania lub niezręcznej struktury programu, co utrudnia czytanie i konserwację kodu innym członkom zespołu (nawet autorowi). To może pośrednio sprawić, że kod stanie się bardziej podatny na problemy związane z bezpieczeństwem w przyszłości lub przynajmniej utrudni osobom przeglądającym kod znalezienie przyszłych problemów z bezpieczeństwem.

Na przykład poniższa funkcja napisana w języku Python przyjmuje niezwykle dużą liczbę parametrów, a następnie ignoruje większość z nich. Obie te cechy są uważane za złe praktyki programistyczne, nawet jeśli żadna z nich nie zagraża zachowaniu ani bezpieczeństwu kodu:

```
def sum(i, j, k, l, m, n, o, p, q, r):
    return i + j
```

Statyczna analiza kodu czasami może działać automatycznie. Wiele środowisk programistycznych oferuje statyczną analizę kodu jako wbudowaną funkcję: podkreślają czerwonymi liniami ostrzeżenia pod niestandardowym lub niebezpiecznym kodem, który wykryją. To może być ogromną pomocą, ale lepiej traktować to jako uzupełnienie ręcznych przeglądów kodu, a nie ich pełny zamiennik.

Wykrywanie tajemnic

Możesz myśleć o **wykrywaniu tajemnic** jako o specjalnym rodzaju statycznej analizy kodu. Istnieje wiele typów poufnych danych, które nie powinny się znajdować w kodzie źródłowym oprogramowania. Łatwo jest podać ich przykłady:

- hasła,
- klucze wdrożenia,
- publiczne klucze SSH lub GPG,
- numery ubezpieczenia społecznego,
- unikalne osobiste numery identyfikacyjne używane przez inne kraje.

Podobnie jak statyczna analiza kodu przeszukuje kod źródłowy w celu znalezienia problemów programistycznych lub związanych z bezpieczeństwem, wykrywanie tajemnic skanuje kod źródłowy w poszukiwaniu poufnych informacji, które powinny być usunięte i przechowywane w bardziej bezpiecznym miejscu. Na przykład poniższy kod Java zawiera numer ubezpieczenia społecznego, który może być widoczny dla każdej osoby mającej dostęp do kodu:

```
String bethSSN = "555-12-1212";
if (customerSSN.equals(bethSSN))) {
    System.out.println("Witaj, Beth!");
}
```

Analiza dynamiczna

Przeglądanie kodu źródłowego jest przydatne, ale istnieje wiele kategorii defektów oprogramowania, które można łatwiej znaleźć, oddziałując z wykonywanym kodem. Ta interakcja może przybierać formę korzystania z interfejsu użytkownika aplikacji, tak jak zrobiłby to człowiek — czyli wysyłania żądań do punktu końcowego interfejsu API REST lub odwiedzania różnych adresów URL aplikacji internetowej z różnymi wartościami w zapytaniach.

Na przykład Twój serwer internetowy może być skonfigurowany w taki sposób, żeby w nagłówkach każdej odpowiedzi zawierał numer wersji. To może wydawać się niewinne, ale może dostarczyć wskazówek złośliwym użytkownikom na temat tego, które ataki ukierunkowane na serwer internetowy są prawdopodobnie skuteczne wobec Twojej witryny, a które ataki są prawdopodobnie nieskuteczne.

Kolejnym przykładem jest złożona logika w Twoim kodzie, która zaciemnia fakt, że można wywołać niewłaściwe działanie dzielenia przez zero, wprowadzając określony zestaw wartości wejściowych. Jak wcześniej omawiano, problemy tego rodzaju mogą początkowo nie wydawać się ryzykowne z punktu widzenia bezpieczeństwa, ale sprytny haker często potrafi znaleźć sposoby na wykorzystanie prostych błędów w sposób, który ujawnia dane, powoduje utratę danych lub prowadzi do ataków typu odmowa usługi.

Na przykład poniższy kod Ruby może generować **instancję błędu dzielenia przez zero** podczas działania, co w konsekwencji może spowodować awarię programu:

```
puts 'ile masz kapeluszy?'
num_hats = gets.to_i
puts 'ile masz kotów?'
num_cats = gets.to_i
puts "masz #{num_hats / num_cats} kapeluszy na kota"
```

Skanowanie zależności

Skanowanie zależności to praktyka porównywania nazw i numerów wersji każdej zależności Twojego produktu z bazą danych znanych podatności i identyfikowanie tych zależności, które powinny być zaktualizowane do nowszej wersji lub usunięte całkowicie w celu poprawy bezpieczeństwa oprogramowania. Prawie każdy złożony program komputerowy napisany dzisiaj opiera się na dziesiątkach, setkach lub tysiącach bibliotek

zewnętrznych o otwartym kodzie źródłowym. Kod źródłowy najpopularniejszych bibliotek jest badany przez hakerów typu Black Hat w poszukiwaniu potencjalnych luk. Te luki są często szybko naprawiane przez osoby utrzymujące bibliotekę, ale jeśli Twój projekt korzysta z przestarzałych, niezałatanych wersji tych bibliotek, skanowanie zależności poinformuje cię, że Twój kod może być podatny na znane ataki.

Doskonałym przykładem potwierdzającym potrzebę tego typu skanowania bezpieczeństwa jest wiadomość, która została podana w chwili pisania książki. Wiele projektów w języku Java opiera się na otwartej bibliotece Java o nazwie Log4j, która umożliwia wygodne rejestrowanie komunikatów informacyjnych, ostrzeżeń lub błędów. Odkryto niedawno podatność, która umożliwia hakerom zdalne wykonywanie poleceń lub instalowanie złośliwego oprogramowania na dowolnym komputerze, na którym działa Log4j. To ogromny problem! Na szczęście to dokładnie rodzaj problemu, który może być wykryty podczas skanowania zależności. Każde aktualne narzędzie do skanowania zależności poinformuje cię, czy Twój program jest zależny — bezpośrednio lub poprzez inne zależności — od niezałatanej wersji Log4j, i doradzi, do jakiej wersji Log4j powinieneś dokonać aktualizacji.

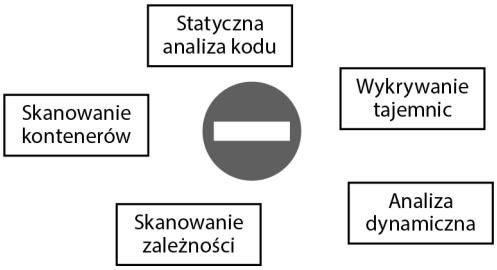
Skanowanie kontenerów

W dzisiejszych czasach wiele produktów oprogramowania jest dostarczanych jako obrazy **Docker**. Najprościej opisać obraz Docker jako dystrybucję Linuksa, na której zainstalowana jest Twoja aplikacja, a następnie jest ona pakowana w format *obrazu*, który może być uruchomiony przez Dockera lub podobne narzędzia. Jeśli zbudujesz obraz Docker, który zawiera przestarzałą wersję dystrybucji Linuksa z luką bezpieczeństwa, Twoja aplikacja nie będzie tak bezpieczna, jak mogłaby być.

Skanowanie kontenerów analizuje bazowy obraz Linuksa, na którym jest zainstalowana Twoja *zdockerowana* aplikacja, i sprawdza bazę danych znanych podatności bezpieczeństwa, aby ocenić, czy Twoja spakowana aplikacja może być podatna na ataki. Ponieważ CentOS 6 przestał być wspierany w 2020 roku, biblioteki, które zawiera, mają wiele poważnych luk bezpieczeństwa. Skanowanie kontenerów ostrzegłoby Cię o tym problemie i zasugerowałoby, abyś rozważył aktualizację obrazu Docker Twojej aplikacji do wersji CentOS 7 lub późniejszej jako obrazu bazowego.

Podsumowanie ręcznych testów bezpieczeństwa

Przeanalizowaliśmy różne testy zaprojektowane do wykrywania podatności bezpieczeństwa lub problemów związanych z bezpieczeństwem, takich jak niestosowanie się do najlepszych praktyk kodowania. Chociaż może się wydawać, że istnieje wiele kroków do przejścia przed wdrożeniem prostej aplikacji internetowej, nigdy wcześniej nie było tylu sposobów na kradzież informacji lub wyłączenie usługi, i nie ma powodu, aby przypuszczać, że ten trend odwróci się w najbliższym czasie. Więc, czy ci się to podoba, czy nie, odpowiedzialni programiści muszą myśleć o wszystkich tych różnych testach bezpieczeństwa i prawdopodobnie je wdrożyć.



Rysunek 1.3. Niektóre z wielu rodzajów testów związanych z bezpieczeństwem

Niektóre z tych testów muszą być wykonywane ręcznie. Inne posiadają automatyczne narzędzia, które pomagają w tym procesie. Jednakże testy automatyczne wciąż są obciążające: musisz zainstalować narzędzia lub frameworki do testowania bezpieczeństwa, skonfigurować narzędzia do testowania, aktualizować frameworki testowe i zależności, tworzyć i utrzymywać środowiska testowe, przetwarzać raporty oraz prezentować je w zintegrowany sposób. Jeśli spróbujesz uprościć te sprawy, przekazując część tych zadań zewnętrznym firmom lub narzędziom typu **SaaS** (ang. *Software-as-a-Service*), będziesz musiał nauczyć się osobnych interfejsów użytkownika dla każdego narzędzia, utrzymywać różne konta użytkowników dla każdej usługi, zarządzać wieloma licencjami i wykonywać wiele innych zadań, aby utrzymać płynność działania testów.

W tym punkcie przekazaliśmy Ci więcej informacji o tym, jak trudne było życie zespołów programistycznych przed pojawieniem się GitLaba. Jak dowiesz się w następnym rozdziale, potoki CI/CD GitLaba zastępują niezręczne, wieloetapowe procesy testowania bezpieczeństwa — szybkimi, automatycznymi skanerami bezpieczeństwa, które konfigurujesz raz i z których korzystasz przez cały czas, kiedy kontynuujesz rozwijanie swojego projektu oprogramowania. Wrócimy do tego tematu w późniejszym etapie znacznie bardziej szczegółowo.

Ręczne pakowanie i wdrażanie kodu

Teraz, gdy Twoje oprogramowanie zostało zbudowane, zweryfikowane i jest bezpieczne, nadszedł czas, aby pomyśleć o jego pakowaniu i wdrażaniu. Podobnie jak w przypadku innych kroków, o których rozmawialiśmy, ten proces może być irytującym obciążeniem, gdy jest wykonywany ręcznie. Sposób pakowania aplikacji do stanu gotowego do wdrożenia zależy nie tylko od języka programowania, w jakim jest napisana, lecz także od narzędzia do zarządzania budową, które jest używane. Na przykład, jeśli używasz narzędzia Maven do zarządzania produktem w języku Java, będziesz musiał uruchomić inną serię poleceń niż w przypadku użycia narzędzia Gradle. Spakowanie kodu Ruby do gema wymaga zupełnie innego procesu. Pakowanie często polega na zbieraniu dziesiątek, setek lub tysięcy plików, pakowaniu ich za pomocą narzędzia odpowiedniego dla danego języka, podwójnym sprawdzaniu, czy dokumentacja i pliki licencji są kompletne i czy znajdują się w odpowiednim miejscu, a także ewentualnie na kryptograficznym podpisywaniu spakowanego kodu, aby pokazać, że pochodzi on od zaufanego źródła.

Już wcześniej wspomnieliśmy o zadaniu określenia, na jakiej licencji jest udostępniany Twój kod. To prowadzi do kolejnego rodzaju testów, które trzeba przeprowadzić przed wdrożeniem kodu w produkcji: **skanowania zgodności licencji**.

Skanowanie zgodności licencji

Większość otwartych bibliotek innych firm jest udostępniana na zasadzie określonej licencji oprogramowania. Istnieje niezliczona liczba licencji, z których deweloperzy mogą wybierać, ale większość otwartych bibliotek korzysta tylko z garści z nich, w tym Licencji MIT, Licencji GNU **GPL** (ang. *General Public License*) oraz Licencji Apache. Ważne jest, aby wiedzieć, jakie licencje są używane przez Twoje zależności, ponieważ nie możesz zgodnie z prawem korzystać z zależności, które używają licencji niezgodnych z ogólną licencją Twojego projektu.

Co może sprawić, że dwie licencje są niezgodne? Niektóre licencje, takie jak Peaceful Open Source License, wyraźnie zabraniają korzystania z oprogramowania przez wojsko. Inną, bardziej powszechną przyczyną konfliktów licencyjnych jest konflikt pomiędzy tzw. licencjami typu Copyleft a licencjami własnościowymi. Licencje *Copyleft*, takie jak GPL, określają, że oprogramowanie korzystające z bibliotek objętych licencją GPL musi używać licencji GPL. Licencje Copyleft czasami nazywa się **licencjami wirusowymi**, ponieważ nakładają swoje ograniczenia licencyjne na oprogramowanie korzystające z zależności objętych tego typu licencjami.

Ponieważ jesteś prawnie zobowiązany do upewnienia się, że główna licencja Twojego projektu jest zgodna z licencjami każdej zależności zewnętrznej, którą wykorzystujesz, musisz dodać krok skanowania licencji do swojego procesu pakowania i wdrażania. Niezależnie od tego, czy jest to robione ręcznie, czy za pomocą narzędzia automatycznego, musisz zidentyfikować i zastąpić wszelkie zależności, których nie wolno Ci używać.

Wdrażanie oprogramowania

Gdy Twoje oprogramowanie zostało spakowane, a Ty dwa razy sprawdziłeś licencje swoich zależności, napotykasz na przeszkodę w postaci wdrażania kodu we właściwym miejscu o właściwym czasie.

Większość zespołów deweloperskich wdraża kod w kilku środowiskach. Każda firma organizuje to inaczej, ale typowa (choć minimalna) struktura środowisk może wyglądać tak:

- Jedno lub więcej **środowisk testowych**.
- **Środowisko staging** lub **przedprodukcyjne**, które jest konfigurowane możliwie podobnie do środowiska produkcyjnego, ale zazwyczaj w o wiele mniejszej skali.
- **Środowisko produkcyjne**.

Omówimy korzystanie z tych różnych środowisk później, ale teraz musisz tylko zrozumieć, jak każde z tych środowisk jest wykorzystywane w ramach podstawowego procesu wdrażania. Gdy kod jest rozwijany, zwykle wdraża się go w środowisku testowym, aby zespół QA lub **inżynierowie ds. wersji** mogli się upewnić, że działa zgodnie z oczekiwaniami

i integruje się z istniejącym kodem bez powodowania problemów. Gdy nowy kod jest gotowy do dodania do kodu źródłowego wersji produkcyjnej, tradycyjnie wdrażany jest w środowisku staging, aby można było przeprowadzić ostatnią rundę testów i upewnić się, że nie ma niezgodności między nowym kodem a środowiskiem, w którym ostatecznie będzie działał. Jeśli te testy przebiegną pomyślnie, kod jest w końcu wdrażany w środowisku produkcyjnym, gdzie prawdziwi użytkownicy mogą skorzystać z nowych funkcji, poprawek błędów lub innych usprawnień, które nowy kod wprowadził.

Jak sobie pewnie wyobrażasz, upewnienie się, że właściwy kod zostaje wdrożony we właściwym środowisku we właściwym czasie, to zadanie trudne, ale niezwykle ważne. A wdrażanie to tylko połowa bitwy! Druga połowa polega na sprawdzeniu, czy różne środowiska są dostępne i sprawne. Muszą działać na właściwym rodzaju i skali sprzętu, muszą być zaopatrzone w odpowiednie konta użytkowników, muszą mieć poprawnie skonfigurowane polityki sieciowe i bezpieczeństwa oraz muszą mieć poprawne wersje systemów operacyjnych, narzędzi i innych aplikacji infrastrukturalnych. Oczywiście istnieją zadania związane z konserwacją, aktualizacjami i innymi pracami rekonfiguracyjnymi systemu, które muszą być zaplanowane, przeprowadzone i naprawione w przypadku awarii. Ogromny zakres i złożoność tych zadań są powodem, dla którego duże organizacje mają duże zespoły inżynierów ds. wersji, które dbają o to, żeby wszystko działało płynnie, i gorączkowo rozwiązują problemy, gdy tak się nie dzieje.

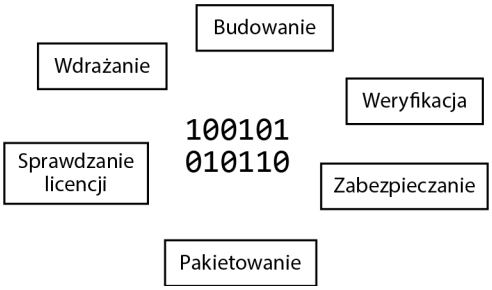
To kończy naszą wycieczkę przez najczęstsze zadania w cyklu życia oprogramowania (SDLC), które mają miejsce po wprowadzeniu nowego kodu:

1. Zbuduj kod.
2. Zweryfikuj funkcjonalność, wydajność, wykorzystanie zasobów i inne aspekty kodu za pomocą różnych testów.
3. Upewnij się, że kod nie ma luk w zabezpieczeniach, korzystając z jeszcze większej liczby testów.
4. Spakuj kod w formacie gotowym do wdrożenia.
5. Szukaj i usuń wszelkie problemy z niezgodnymi licencjami.
6. Wdrażaj kod w odpowiednim środowisku.

Teraz pewnie dostrzegasz pewien motyw: życie przed GitLabem było skomplikowane, podatne na błędy i powolne. Te przymiotniki z pewnością odnoszą się do zadań związanych z pakowaniem, skanowaniem licencji i wersjonowaniem, które występują w końcowym etapie cyklu życia oprogramowania. Ale, jak się dowiesz bardziej szczegółowo w późniejszym rozdziale, potoki CI/CD GitLaba zajmują się najbardziej uciążliwymi aspektami tych zadań za Ciebie. Pozwalając potokowi obsługiwać nudne i powtarzalne czynności, możesz się skupić na bardziej kreatywnych i satysfakcjonujących częściach pisania oprogramowania.

Problemy z ręcznymi praktykami w cyklu życia oprogramowania

Teraz, gdy masz ogólny obraz tego, co się dzieje z oprogramowaniem od momentu, gdy deweloperzy zakończą jego pisanie, aż do momentu, gdy użytkownicy mogą się z nim zapoznać, zaczynasz rozumieć, jak trudny może być ten proces. Wiele zadań musi zostać wykonanych na tej ścieżce dostarczania bezpiecznego, działającego kodu.



Rysunek 1.4. Główne zadania w cyklu życia oprogramowania (SDLC)

Niektóre z tych zadań są zazwyczaj wykonywane ręcznie, podczas gdy inne mogą być częściowo lub całkowicie zautomatyzowane. Z obydwoma podejściami są jednak związane trudności, które sprawiają, że każde zadanie staje się potencjalnym problemem.

Jakie są trudności związane z ręcznym wykonywaniem tych zadań? Spójrzmy na to bliżej:

- **Wymagają czasu.** Często zajmują znacznie więcej czasu niż przewidujesz, nawet jeśli miałeś doświadczenie w ręcznym wykonywaniu ich w przeszłości. Istnieje nieskończona liczba sytuacji, w których coś może pójść nie tak podczas wykonywania któregokolwiek z tych zadań, co wymaga żmudnego rozwiązywania problemów i naprawiania. Nawet gdy wszystko idzie dobrze, po prostu jest dużo pracy związanej z każdym z tych zadań. I pamiętaj o prawie z 1979 roku zaproponowanym przez fizyka Douglasa Hofstadtera: *Zawsze zajmuje to więcej czasu, niż się spodziewasz, nawet jeśli weźmiesz pod uwagę prawo Hofstadtera*.
- **Są podatne na błędy**. Ponieważ polegasz na ludziach, którzy je wykonują — ludziach, którzy mogą być zmęczeni, znudzeni lub rozproszeni — wszystkie są podatne na błędy konfiguracji, błędy we wprowadzaniu danych lub kroki, które zostały pominięte lub zastosowane w niewłaściwej kolejności, aby wymienić tylko kilka błędów, które mogą prowadzić do problemów.
- **Są trudne dla morale pracowników.** Nikt nie lubi wykonywać rutynowej, powtarzalnej pracy, zwłaszcza gdy stawki są wysokie i musisz zrobić to dobrze. Perspektywa przeprowadzania standardowego dwugodzinnego zestawu testów ręcznych po raz dwudziesty w ciągu 2 tygodni skłoniła już wielu inżynierów ds. jakości (QA) do zastanowienia się, czy może praca w dziedzinie oprogramowania to jednak nie najlepszy wybór ścieżki kariery.
- **Istnieje duże ryzyko niedokładnej komunikacji lub raportowania**. Kiedy tester ręczny zakończy swoje wyczerpujące dwugodzinne testy, czy ma wystarczająco

dużo komórek mózgowych, aby dokładnie zarejestrować, co działało, a co nie? Wszystkie te testy są bezcelowe, jeśli nie możemy polegać na dokładności zarejestrowanych wyników, ale każdy, kto wykonywał trudny plan testów manualnych, wie, ile niejasności może być w wynikach, ile nieoczekiwanych warunków może wpłynąć na potencjalnie zakłócenia testów i jak trudno wytłumaczyć te czynniki ludziom, którzy polegają na tych raportach. A to nie uwzględnia bardzo dużej możliwości po prostu niepoprawnego zarejestrowania wyników, nawet gdy są one jednoznaczne.

Z tych wszystkich powodów możesz zobaczyć, jak kosztowne — pod względem czasu, pieniędzy i dobrej woli pracowników — są ręczne zadania.

Ale jeśli niektóre z opisanych zadań można zautomatyzować, czy to wyeliminowałoby to problemy, z którymi mierzymy się w przypadku zadań manualnych? Cóż, rozwiązałoby to pewne problemy. Jednak dodanie serii narzędzi automatyzacji do cyklu życia oprogramowania (SDLC) wprowadziłoby zupełnie nowy zestaw problemów. Rozważmy cały dodatkowy wysiłek i koszty z tym związane oraz wszystkie zadania, jakie pociąga za sobą budowanie niestandardowych narzędzi:

- Badanie i wybór narzędzi do każdego zadania podlegającego automatyzacji.
- Zakup i odnawianie licencji dla każdego narzędzia.
- Wybór rozwiązania hostingowego dla każdego narzędzia.
- Przypisywanie użytkowników do każdego narzędzia.
- Poznanie różnych interfejsów graficznych (GUI) dla każdego narzędzia.
- Zarządzanie bazami danych i inną infrastrukturą dla każdego narzędzia.
- Integracja każdego narzędzia z innymi narzędziami w SDLC.
- Zrozumienie, jak wyświetlać status i wyniki każdego narzędzia w jednym centralnym miejscu, jeśli jest to w ogóle możliwe.
- Radzenie sobie z narzędziami, które są błędne, stają się przestarzałe lub stają się mniej przekonujące w miarę pojawiania się lepszych alternatyw na rynku.

Nawet po rozwiązaniu wszystkich problemów związanych z zadaniami manualnymi lub zautomatyzowanymi pozostaje jeden duży problem, którego nie da się uniknąć w zespołach korzystających z tego modelu: jest to sekwencyjny przepływ pracy. Kroki następują po sobie. Jeden zespół pisze oprogramowanie, a następnie przekazuje kod innej grupie, która go buduje. Ta grupa z kolei przekazuje kod trzeciej grupie odpowiedzialnej za walidację oprogramowania. Gdy skończą, zazwyczaj przesyłają go do kolejnej grupy inżynierów, którzy przeprowadzają testy bezpieczeństwa. Wreszcie, zespół ds. wydań przejmuje kontrolę nad kodem, aby można go było wdrożyć we właściwym miejscu. Proces ten może odbiegać od tego opisu na wiele sposobów, ale podstawowa koncepcja wykonywania jednego kroku na raz i przekazywania kodu do kolejnego kroku dopiero po zakończeniu poprzednich kroków, jest cechą wspólną dla wielu, wielu zespołów deweloperskich.

Dotąd może nie być oczywiste, dlaczego sekwencyjne przepływy pracy stwarzają problem, więc wyjaśnijmy to. Ze względu na trudność wykonywania tych kroków ręcznie lub ze względu na kłopoty z utrzymaniem zautomatyzowanych kroków w płynnym i niezawodnym działaniu w różnych narzędziach, ten przepływ pracy zazwyczaj ma charakter sporadyczny. Częstotliwość wykonywania tych kroków różni się w zależności od zespołu,

ale czas i koszty związane z tym oznaczają, że zmiany w kodzie zazwyczaj gromadzą się przez dni, tygodnie, a czasem nawet miesiące, zanim zostaną zbudowane, zweryfikowane, zabezpieczone i odpowiednio wdrożone. A to z kolei oznacza, że problemy wykryte w trakcie tego procesu są kosztowne do naprawienia. Jeśli test funkcjonalny skończy się niepowodzeniem, test bezpieczeństwa wykryje podatność albo test integracyjny pokaże, że kod nie działa dobrze razem po wdrożeniu w tym samym środowisku, zidentyfikowanie, który fragment kodu jest źródłem problemu, jest jak znalezienie igły w bardzo dużym stogu siana. Jeśli zmieniło się 5000 linii kodu w 25 klasach, zaktualizowano 16 zależności do nowszych wersji, wersja Javy zmieniła się z 16 na 17, a środowisko testowe działa na innej wersji Ubuntu, to jest wiele zmiennych do zbadania, gdy próbujesz namierzyć źródło problemu i znaleźć sposób jego naprawienia.

W tym momencie masz wystarczającą wiedzę na temat tradycyjnego, przed-DevOpsowego rozwoju oprogramowania. Możemy podsumować jego największy problem w jednym zdaniu: *sekwencyjne przepływy pracy, które obejmują zadania manualne lub zadania zautomatyzowane wykonywane przy użyciu różnych narzędzi, powodują, że rozwój jest powolny, wydania są niezbyt liczne, a oprogramowanie jest niższej jakości, niż mogłoby to być*.

Ale są dobre wieści. Aby rozwiązać te problemy, wynaleziono DevOps. A potem powstały potoki GitLaba CI/CD, aby ułatwić korzystanie z DevOps. Wkrótce omówimy obie te rzeczy.

Rozwiązywanie problemów za pomocą DevOps

Co rozumiemy przez **DevOps**? Mimo że termin ten jest używany przez społeczność programistyczną od co najmniej 10 lat (pierwsza sesja **devopsdays**, obecnie największej konferencji skupiającej się na DevOps, odbyła się w 2009 roku), obecnie nie ma jednej, standardowej definicji, na którą wszyscy by się zgadzali.

Kiedy GitLab mówi o DevOps, odnosi się do nowego sposobu myślenia o cyklu życia oprogramowania (SDLC), który koncentruje się na czterech aspektach:

- automatyzacja,
- współpraca,
- szybka informacja zwrotna,
- ulepszanie iteracyjne.

Przyjrzyjmy się każdemu z tych aspektów bardziej szczegółowo.

Głównym celem DevOps jest **zautomatyzowanie** jak największej liczby zadań związanych z rozwojem oprogramowania. Eliminuje to wyzwania związane z ręcznym tworzeniem, testowaniem, zabezpieczaniem i wdrażaniem. Ale ma to ograniczoną wartość, jeśli zamienia te wyzwania na kłopoty i koszty związane z łączeniem zbioru narzędzi manualnych. Zobaczymy, jak GitLab rozwiązuje ten problem, ale na razie zrozum, że prawidłowy przepływ pracy DevOps jest w pełni zautomatyzowany.

Poprzez **promowanie współpracy** między wszystkimi zespołami zaangażowanymi w tworzenie oprogramowania, a także między wszystkimi członkami każdego zespołu, DevOps

pomaga zidentyfikować newralgiczne punkty i rozwiązać potencjalne problemy, które występują za każdym razem, gdy kod jest przenoszony z jednego zespołu do drugiego. Jeśli nie ma „murów" na drodze przekazywania kodu — jeśli każdy krok w procesie jest przezroczysty dla wszystkich zaangażowanych w tworzenie i dostarczanie oprogramowania — każdy czuje się zaangażowany w ogólną jakość kodu i ma wrażenie, że gra z pozostałymi w jednym zespole. Różne osoby wciąż ponoszą główną odpowiedzialność za konkretne zadania, ale ogólna kultura zmierza ku wspólnej własności kodu i wspólnym celom.

Szybka informacja zwrotna może być najważniejszym i rewolucyjnym elementem DevOps. Można ją uznać za wynik dwóch wcześniej omówionych koncepcji: równoczesnych przepływów pracy i „przesunięcia w lewo". Gdy się nad tym zastanowisz, te dwa terminy sprowadzają się do tego samego: wykonaj wszystkie zadania związane z tworzeniem, weryfikacją i zabezpieczaniem tak szybko, jak to możliwe dla każdej partii kodu, którą programiści dodają. Wykonując wszystkie te zadania równocześnie zamiast sekwencyjnie, uzyskasz pewność, że zachodzą one na bardzo wczesnym etapie cyklu życia oprogramowania. I wykonuj wszystkie te zadania natychmiast dla każdej porcji nowego kodu, która jest dodawana, bez względu na to, jak mała ona jest. Poprzez wcześniejsze i częstsze wykonywanie tych zadań minimalizujesz rozmiar zmian w kodzie, które są testowane, co sprawia, że tańsze i łatwiejsze staje się rozwiązywanie ewentualnych błędów w oprogramowaniu, problemów konfiguracji lub podatności na zagrożenia, które zostaną wykryte przez testy.

Jeśli szybko znajdujesz i naprawiasz problemy, będziesz w stanie częściej udostępniać oprogramowanie klientom. Poprzez wcześniejsze dostarczanie nowych funkcji i poprawek błędów, pomagasz im czerpać korzyści z **iteracyjnej poprawy produktu**. Wprowadzając mniejsze zmiany w kodzie w krótszych odstępach czasu, przy mniejszym ryzyku popsucia czegoś i potrzeby cofnięcia zmian, wpisujesz się w hasło „tworzenia nudnych wydań". W tym przypadku nuda to dobra rzecz: większość klientów wolałaby częste, małe aktualizacje, które niosą niewielkie ryzyko, niż rzadkie, ogromne zmiany, które mogą wprowadzić chaos i skutkować koniecznością cofnięcia zmian.

Korzystając z automatyzacji, współpracy, szybkiej informacji zwrotnej i iteracyjnej poprawy, praktyki DevOps pozwalają tworzyć kod o wyższej jakości i tańszy w produkcji, który jest częściej dostarczany użytkownikom.

Jak GitLab implementuje DevOps

GitLab to narzędzie umożliwiające wykonanie wszystkich zadań związanych z rozwojem oprogramowania, o których rozmawialiśmy, z wykorzystaniem zasad DevOps, które przedstawiliśmy. Najważniejszą cechą GitLaba jest to, że jest to jedno narzędzie, które łączy wszystkie etapy cyklu życia oprogramowania

Pamiętasz, jak przechodzenie z procesów manualnych na procesy zautomatyzowane rozwiązało pewne problemy, ale spowodowało powstanie nowych, związanych z automatyzacją? Jednonarzędziowe podejście GitLaba rozwiązuje także te trudności. Przyjrzyjmy się korzyściom płynącym z następującego podejścia jednego, zintegrowanego środowiska narzędziowego:

- jeden licencjonowany zakup (chyba że Twój zespół korzysta z darmowej, ograniczonej funkcjonalności wersji GitLaba, wtedy nie trzeba kupować licencji);
- jedna aplikacja do utrzymania i aktualizacji;
- jedna grupa kont użytkowników do utworzenia;
- jeden zarządzany zbiór danych;
- jeden interfejs graficzny do nauczenia się;
- jedno miejsce do sprawdzania — aby zobaczyć raporty i statusy wszystkich kroków budowy, weryfikacji, zabezpieczeń, pakowania i wdrażania.

To, że GitLab jest jednym narzędziem, rozwiązuje problemy wynikające z używania rozproszonych narzędzi automatyzacyjnych. Co więcej, fakt, że używa on jednego zestawu komponentów i encji, które współpracują i dobrze się komunikują między sobą, umożliwia i zachęca do współpracy, równoczesności, przejrzystości i współwłasności, które są tak istotne w kontekście DevOps. Gdy masz równoczesne zadania, uzyskujesz szybką informację zwrotną. A to z kolei pozwala na iteracyjną poprawę poprzez tworzenie „nudnych wydań".

Większość pozostałej części tej książki zajmuje się techniką, jaką GitLab wykorzystuje do praktycznego wdrożenia zasad DevOps: **potokami CI/CD**. Jeszcze teraz nie zdefiniujemy, co oznacza ten termin, ale dowiesz się wszystkiego z dalszych rozdziałów. Na razie musisz wiedzieć, że potoki CI/CD to miejsce, gdzie gumowe koło GitLaba spotyka się z drogą DevOps: to, jak model jednonarzędziowy GitLaba wykonuje całe budowanie, weryfikowanie, zabezpieczanie, pakowanie i wdrażanie, przez które musi przejść Twój kod.

Nie możemy pominąć tego, że duża część GitLaba służy pomocy w planowaniu pracy, przydzielaniu jej i zarządzaniu nią. Ale te zagadnienia nie należą do potoków CI/CD i w związku z tym wykraczają poza zakres tej książki. Będziemy czasami poruszać inne tematy po prostu dlatego, że wszystko w GitLabie jest tak ze sobą powiązane, że nie ma możliwości pozostawania wyłącznie w granicach potoków CI/CD. Pozostała części tej książki wyjaśni, co potrafią potoki GitLaba i jak z nich korzystać.

Podsumowanie

Osoby, które nie pracują w firmach zajmujących się oprogramowaniem, mogą nie zdawać sobie sprawy, że tworzenie oprogramowania to nie tylko... *pisanie oprogramowania*. Po wprowadzeniu go trzeba przejść przez długi i skomplikowany szereg kroków, aby zbudować, zweryfikować, zabezpieczyć, spakować i wdrożyć kod, zanim użytkownicy będą mogli z niego korzystać. Wszystkie te kroki można wykonać ręcznie, a niektóre z nich — w określonych warunkach — można zautomatyzować. Ale zarówno manualne, jak i zautomatyzowane podejście do przygotowania oprogramowania stwarza problemy.

DevOps to stosunkowo nowe podejście do realizacji tych kroków. Łączy ono automatyzację, współpracę, szybką informację zwrotną i iteracyjną poprawę w sposób, który pozwala zespołom robić oprogramowanie — lepiej, szybciej i taniej.

GitLab to narzędzie DevOps, które gromadzi wszystkie te zadania w jednym miejscu, pozwalając zespołowi do rozwoju oprogramowania wykonać wszystko za pomocą jednego

narzędzia, używając jednego interfejsu graficznego, z wynikami testów i statusem wdrożeń wyświetlanymi w jednym miejscu. Skupienie na automatyzacji rozwiązuje problemy stworzone przez procesy manualne. Jego model jednonarzędziowy rozwiązuje problemy stworzone przez procesy zautomatyzowane.

GitLab wciela wszystkie zasady DevOps w życie poprzez wykorzystanie potoków CI/CD, które będą głównym tematem pozostałej części tej książki.

Ale zanim zajmiemy się potokami CI/CD, musimy na krótko, w jednym rozdziale, zanurzyć się w Gicie – narzędziu, na którym opiera się GitLab. Bez znajomości podstaw Gita prawdopodobnie wiele koncepcji i terminologii GitLaba będzie dla Ciebie mylących. Więc przygotuj się, sięgnij po dużą filiżankę swojego ulubionego napoju kofeinowego – czas na Gita.

DevOps.
Wdrażaj zmiany szybko i bezpiecznie!

Obecnie zasady i wzorce DevOps pozwalają na ograniczenie ryzyka związanego z budowaniem, zabezpieczaniem i wdrażaniem kodu. Programiści i wdrożeniowcy wiedzą, że zapewnienie funkcjonalnej poprawności, efektywności i bezpieczeństwa kodu jest czasochłonne i skomplikowane. Dużym ułatwieniem w tym zakresie mogą być potoki CI/CD GitLaba.

Dzięki tej książce poznasz od podstaw Git i GitLab. Dowiesz się, jak skonfigurować runnery GitLaba, a także jak tworzyć i konfigurować potoki dla różnych etapów cyklu rozwoju oprogramowania. Poznasz zasady interpretacji wyników potoków w GitLabie. Nauczysz się też wdrażania kodu w różnych środowiskach i korzystania z wielu zaawansowanych funkcji, takich jak łączenie GitLaba z Terraformem, klastrami Kubernetes czy uruchamianie i poprawa wydajności potoków. Skorzystasz ponadto z licznych przykładów i studiów przypadku, dzięki którym za pomocą potoków CI/CD zautomatyzujesz wszystkie etapy DevOps do budowy i wdrażania kodu o wysokiej jakości.

Najciekawsze zagadnienia:

- podstawy Gita, GitLaba i DevOps
- tworzenie, przeglądanie i uruchamianie potoków CI/CD GitLaba
- weryfikacja, zabezpieczanie i wdrażanie kodu za pomocą potoków CI/CD GitLaba
- runnery, DAG-i i logika warunkowa GitLaba
- najlepsze praktyki i metody rozwiązywania problemów w potokach CI/CD GitLaba
- przykłady cykli życia procesów rozwoju oprogramowania

Christopher Cowell tworzy treści edukacyjne w Instabase, wcześniej był trenerem w GitLabie. Przez dwie dekady pracował również jako naukowiec i inżynier QA w takich firmach jak Accenture, Oracle i Puppet.

Nicholas Lotz jest trenerem technicznym w GitLabie. Wcześniej pracował jako inżynier systemowy i konsultant w branży infrastruktury oprogramowania.

Chris Timberlake jest starszym architektem rozwiązań w GitLabie. Wcześniej pracował w Red Hat. Ma spore doświadczenie w obszarze bezpieczeństwa i pracy w organach ścigania.

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 978-83-289-0775-1

9 788328 907751

Cena: 79,00 zł