

Odpowiedzi na pytania sprawdzające Twoją wiedzę

Dodatek

Ten dodatek zawiera odpowiedzi na pytania z sekcji *Sprawdź swoją wiedzę* znajdujące się pod koniec każdego rozdziału.

Rozdział 1. Wprowadzenie do narzędzi i umiejętności dla .NET

Oto proponowane odpowiedzi na te pytania:

1. Jak postrzegani są deweloperzy .NET, którzy wybierają Ridera, w porównaniu do tych, którzy używali wyłącznie Visual Studio?

Odpowiedź: Postrzeganie deweloperów .NET używających JetBrains Ridera w porównaniu do tych, którzy pozostają przy Visual Studio, może się znacznie różnić w różnych kręgach społeczności technologicznej. Oba środowiska programistyczne (IDE) mają swoich zwolenników i przeciwników, a wybór często odzwierciedla osobiste preferencje, specyfikę projektu i ogólne trendy wśród deweloperów.

Visual Studio istnieje od końca lat 90. XX w. i jest często uważane za domyślne narzędzie IDE dla programistów .NET. Jest głęboko zintegrowane z ekosystemem Windowsa i narzędziami deweloperskimi Microsoftu. Deweloperzy korzystający z Visual Studio są często postrzegani jako trzymający się tradycyjnych, być może bezpieczniejszych rozwiązań. To IDE jest rozbudowane, oferuje wiele funkcji od razu po instalacji i jest bezpośrednio wspierane przez Microsoft, co zapewnia wysoką kompatybilność ze wszystkimi aktualizacjami .NET oraz płynną integrację z narzędziami i usługami Microsoftu.

Z kolei Rider jest produktem firmy JetBrains, znanej z narzędzi programistycznych, takich jak IntelliJ IDEA i ReSharper. Rider jest chwalony za możliwość działania na różnych systemach operacyjnych, umożliwiając deweloperom pracę nad aplikacjami .NET w systemie macOS, a także na Linuksie i Windowsie.

Ta elastyczność jest bardzo pożądana we współczesnych, zróżnicowanych środowiskach programistycznych. Rider zawiera już narzędzie do analizy kodu i refaktoryzacji ReSharper, co niektórzy programiści uważają za podstawę zwiększenia swojej produktywności i poprawy jakości kodu.

Programiści wybierający Ridera mogą być postrzegani jako bardziej skłonni do eksplorowania nowych narzędzi i innowacji spoza ekosystemu Microsoftu. Mogą być również uważani za osoby ceniące elastyczność i wydajność, biorąc pod uwagę reputację Ridera jako szybszego i mniej zasobożernego narzędzia w porównaniu do Visual Studio. Użytkownicy Ridera są często kojarzeni ze współczesnymi praktykami programistycznymi, szczególnie w środowiskach ceniących wieloplatformowość i nieograniczających się wyłącznie do Windowsa. Korzystanie z Ridera może także sygnalizować otwartość programisty na szeroki wachlarz technologii i zdolność do wybierania narzędzi optymalnych, a nie tylko standardowych.

Wybór środowiska IDE nie jest bezpośrednio powiązany z poziomem umiejętności programisty. Wysoka biegłość w .NET może być osiągnięta w obu środowiskach, a decyzja o wyborze danego IDE często sprowadza się do osobistych preferencji, wymagań projektu lub polityki firmy. Jednak osoby znające więcej niż jedno IDE są często postrzegane jako bardziej wszechstronne.

W niektórych środowiskach korporacyjnych, zwłaszcza tych ściśle związanych z produktami Microsoftu, używanie Visual Studio może być niepisaną normą, podczas gdy start-upy lub firmy z bardziej zróżnicowanym stosem technologicznym mogą doceniać elastyczność oferowaną przez Ridera. W kontekście rekrutacji czy pracy zespołowej znajomość obu tych narzędzi może być Twoim atutem, ponieważ wskazuje zdolność adaptacji i gotowość do korzystania z najlepszego narzędzia do danego zadania.

W ostatecznym rozrachunku największe znaczenie mają umiejętności, doświadczenie i efektywne wykorzystanie dostępnych narzędzi.

2. Jakiej frazy warto używać podczas konstruowania lepszych promptów w ChatGPT?

Odpowiedź: W przypadku złożonych odpowiedzi podczas konstruowania promptów dla ChatGPT należy używać frazy „myśl krok po kroku” („think step-by-step”).

3. O czym należy pamiętać, zadając pytanie na forum lub kanale Discorda?

Odpowiedź: Przed zadaniem pytania warto najpierw samodzielnie spróbować zapoznać się z danym tematem, a następnie uwzględnić znalezione informacje w treści pytania. Pytanie powinno zachować konkretną i zwięzłą formę. Warto także pokazać swoje dotychczasowe próby rozwiązania problemu, aby inni widzieli, co już zostało sprawdzone. Nie można też zapominać o uprzejmości i cierpliwości.

4. Po wydaniu .NET 9 w listopadzie 2024 r. pobierasz i instalujesz SDK. Jak możesz skonfigurować swoje projekty, aby nadal były kompilowane dla .NET 8, a jednocześnie korzystały z zalet kompilatora C# 13?

Odpowiedź: Można skonfigurować projekt tak, aby docelową platformą pozostała .NET 8, a jednocześnie korzystać z nowości kompilatora C# 13. W tym celu w pliku projektu trzeba przypisać elementowi `<TargetFramework>` wartość `net8.0` i dodać element `<LangVersion>` z przypisaną wartością 13, tak jak w poniższym fragmencie kodu:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <LangVersion>13</LangVersion> <!--Wymaga .NET 9 SDK GA-->
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

5. Do czego służy repozytorium GitHuba tej książki?

Odpowiedź: Repozytorium GitHuba tej książki służy do przechowywania kodu rozwiązań zawartych w książce, udostępniania dodatkowych materiałów rozbudowujących jej treść, takich jak poprawki błędów, drobne ulepszenia i listy przydatnych linków, a także umożliwia czytelnikom kontakt ze mną w przypadku pytań lub problemów związanych z książką.

Rozdział 2. Pełne wykorzystanie edytora kodu

Oto sugerowane odpowiedzi na te pytania:

1. Opisz kroki korzystania z funkcji zmiany sygnatury metody w Visual Studio. Uwzględnij w odpowiedzi sposób uruchomienia tej funkcji oraz opcje, jakie są dostępne podczas tego procesu.

Odpowiedź: Aby w Visual Studio skorzystać z funkcji refaktoryzacji *Zmień sygnaturę*, wykonaj następujące kroki:

- 1) **Znajdź metodę.** Najpierw przejdź do metody, której sygnaturę chcesz zmienić. Możesz to zrobić, przewijając kod lub korzystając z funkcji *Idź do* (*Ctrl+T*), aby wyszukać metodę po nazwie.
- 2) **Aktywuj funkcję.** Gdy znajdziesz metodę, kliknij, aby umieścić kursor w dowolnym miejscu nazwy lub deklaracji metody. Kliknij prawym przyciskiem myszy, aby otworzyć menu kontekstowe, a następnie wybierz pozycję *Refaktoryzuj/Zmień sygnaturę...* Możesz bezpośrednio wywołać tę funkcję, naciskając klawisze *Ctrl+R*, *Ctrl+V*.
- 3) **Zmodyfikuj sygnaturę.** W oknie dialogowym *Zmień sygnaturę* możesz wprowadzać różne zmiany, podane na poniższej liście:
 - **Dodawanie nowych parametrów.** Możesz dodać do sygnatury nowe parametry, klikając przycisk *Dodaj*. Pozwala to wprowadzić typ, nazwę i domyślną wartość parametru.

- **Usuwanie parametrów.** Parametry można też usunąć, zaznaczając je i klikając przycisk *Usuń*.
 - **Zmiana kolejności parametrów.** Parametry można przestawiać, zaznaczając parametr i używając przycisków ze strzałką w górę lub w dół.
 - **Zmiana nazw parametrów.** Aby zmienić nazwę dowolnego parametru, wystarczy edytować ją bezpośrednio na liście.
 - **Zmiana typów parametrów.** Możesz również zmienić typ dowolnego parametru, edytując go bezpośrednio na liście.
- 4) **Podgląd zmian.** Visual Studio umożliwia podgląd wszystkich zmian, które zostaną wprowadzone w całej bazie kodu w wyniku refaktoryzacji. Pozwala to zobaczyć, jak zmiany wpłyną na inne części kodu korzystające z danej metody.
- 5) **Zastosowanie zmian.** Po dokonaniu niezbędnych modyfikacji kliknij przycisk *OK*, aby zastosować zmiany. Visual Studio automatycznie zaktualizuje wszystkie odniesienia do metody w całym rozwiązaniu, aby odzwierciedlić nową sygnaturę.
2. Wyjaśnij, jak plik *.editorconfig* jest używany do standaryzowania stylów formatowania kodu. Opisz, czym jest plik *.editorconfig*, i podaj przykład ustalania stylu wcięcia na cztery spacje.

Odpowiedź: Plik *.editorconfig* służy do definiowania i utrzymywania spójnych stylów kodowania oraz konfiguracji dla projektu w różnych edytorach kodu i środowiskach IDE. Jest on umieszczany w katalogu głównym projektu i służy do zmieniania globalnych ustawień Visual Studio dla danego projektu na podstawie ustawień określonych w pliku.

Aby skorzystać z pliku *.editorconfig*, należy wykonać poniższe kroki:

- 1) W katalogu głównym projektu utwórz nowy plik tekstowy o nazwie *.editorconfig*.
- 2) Aby ustawić styl wcięć na cztery spacje, należy dopisać do tego pliku następujące informacje:

```
root = true
[*]
indent_style = space
indent_size = 4
```

 - Zapis `root = true` wskazuje, że jest to główny plik *.editorconfig* i konfiguracje z plików spoza tego katalogu nie powinny być dziedziczone.
 - Zapis `[*]` oznacza, że poniższe ustawienia mają zastosowanie do wszystkich plików.
 - Zapis `indent_style = space` wybiera spację jako styl wcięć.
 - Zapis `indent_size = 4` definiuje rozmiar każdego poziomego wcięcia na 4 spacje.
3. Na czym polega różnica między fragmentem kodu typu `expansion` a `surrounds with`? Wyjaśnij, jak skorzystać z obu typów, i podaj przykład każdego z nich.

Odpowiedź: Fragmenty kodu typu `expansion` to szablony, które wstawiają kod w miejscu kursora. Są często używane do szybkiego dodawania często wykorzystywanych struktur kodu, takich jak pętle, instrukcje warunkowe, definicje klas czy całe metody. Aby użyć fragmentu typu `expansion`, zazwyczaj wpisuje się krótką nazwę reprezentującą fragment, a następnie naciska klawisz `Tab`, by rozwinąć fragment do pełnego szablonu kodu. Na przykład w pliku C# wpisanie `for` i naciśnięcie klawisza `Tab` spowoduje wstawienie szablonu pętli `for`, który można następnie dostosować do własnych potrzeb.

Fragmenty kodu typu `surrounds with` służą do opakowywania istniejącego kodu w nową strukturę. Są szczególnie przydatne do obejmowania zaznaczonych bloków kodu w takie konstrukcje jak instrukcje `if`, bloki `try-catch` czy znaczniki HTML-a w kontekście programowania stron WWW. Aby użyć fragmentu `surrounds with`, należy zaznaczyć blok kodu, który chce się objąć strukturą, kliknąć prawym przyciskiem myszy, by otworzyć menu kontekstowe i wybrać z niego pozycję *Fragmenty kodu/Otocz...*, albo użyć skrótu klawiaturowego `Ctrl+K`, `Ctrl+S`. Na przykład jeżeli mamy fragment kodu, który powinien być wykonywany warunkowo, możemy go zaznaczyć, aktywować funkcję *Otocz przez...* i wybrać fragment `if`, a zaznaczony kod zostanie objęty instrukcją `if`.

4. Który atrybut elementu `<Code>` jest wymagany przy definiowaniu fragmentu kodu? Wymień często używane wartości dla tego atrybutu.

Odpowiedź: Jedynym wymaganym atrybutem elementu `<Code>` jest atrybut `Language`, a najczęstsze wartości tego atrybutu to `CSharp`, `XAML`, `XML`, `JavaScript`, `SQL` i `HTML`.

5. Który szablon projektu należy zainstalować, aby zyskać możliwość tworzenia niestandardowych szablonów projektów dla .NET SDK?

Odpowiedź: Aby tworzyć własne szablony projektów dla .NET SDK, należy zainstalować szablon projektu `Microsoft.TemplateEngine.Authoring.Templates`, używając do tego poniższego polecenia:

```
dotnet new install Microsoft.TemplateEngine.Authoring.Templates
```

Rozdział 3. Zarządzanie kodem źródłowym za pomocą Gita

Oto sugerowane odpowiedzi na pytania:

1. Wymień pięć ważnych cech systemu zarządzania kodem źródłowym.

Odpowiedź: Do ważnych cech systemu zarządzania kodem źródłowym należą:

- **Kontrola wersji.** Pozwala programistom zarządzać zmianami w bazie kodu. Systemy kontroli wersji umożliwiają zapisywanie migawek ze zmianami, przywracanie wcześniejszych wersji plików oraz utrzymywanie historii z informacjami o tym, kto, co i kiedy zmienił.

- **Tworzenie i scalanie gałęzi.** Tworzenie gałęzi daje programistom możliwość odejścia od głównej linii rozwoju i równoległej pracy nad różnymi funkcjami lub projektami bez ingerencji w główną gałąź kodu. Po zakończeniu i przetestowaniu pracy nad daną gałęzią można ją scalić z główną.
 - **Zarządzanie współbieżnością.** W środowiskach, gdzie wielu programistów pracuje na tym samym zbiorze plików, systemy kontroli wersji muszą efektywnie obsługiwać równoczesny dostęp do nich. Wymusza to stosowanie mechanizmów blokowania zapobiegających konfliktom oraz bardziej zaawansowane mechanizmy scalania pomagające rozwiązywać konflikty powstające w sytuacji, gdy dwóch programistów modyfikuje ten sam plik.
 - **Śledzenie zmian i prowadzenie audytów.** Systemy kontroli wersji powinny zapewniać szczegółowe dzienniki i ścieżki audytowe rejestrujące każdą zmianę w kodzie źródłowym. Powinny one zawierać informacje o tym, kto dokonał zmiany, kiedy została dokonana i co dokładnie zostało zmienione (często z komunikatem commitu wyjaśniającym powód zmiany).
 - **Integracja z narzędziami programistycznymi.** Solidny system kontroli wersji powinien bezproblemowo integrować się z innymi narzędziami używanymi w procesie tworzenia oprogramowania, takimi jak edytory kodu, systemy kompilacji, frameworki testowe i narzędzia wdrożeniowe. Ta integracja pozwala usprawnić pracę, umożliwiając automatyzację zadań, np. uruchamianie kompilacji, testów lub procesów wdrożeniowych po zatwierdzeniu zmian w repozytorium.
2. Dlaczego tak trudno nauczyć się pracy z Gitem?
- Odpowiedź:** Korzystanie z Gita, zwłaszcza praca z nim z poziomu wiersza poleceń, jest wyzwaniem. Co prawda podstawowe polecenia, takie jak zatwierdzanie i wypychanie zmian, są proste, ale już wybrnięcie z takich sytuacji jak przypadkowe zatwierdzenia czy wypchnięcia może być naprawdę trudne, jeżeli nie zrozumiesz modelu grafu leżącego u podstaw całego systemu kontroli wersji.
3. Plik w katalogu używanym jako repozytorium Gita może znajdować się w jednym z czterech stanów. Jakie to stany?
- Odpowiedź:** Cztery stany, w jakich może znajdować się plik w katalogu repozytorium Gita, to: nieśledzony, gotowy do zatwierdzenia, niemodyfikowany lub zmodyfikowany.
4. Kiedy można użyć polecenia `git diff`?
- Odpowiedź:** Polecenie `git diff` służy do wyświetlania różnic między różnymi commitami, gałęziami, plikami i innymi elementami. Podaje ono informacje na temat zmian w kodzie, jest zatem jednym z ważniejszych narzędzi dla programistów.
- Oto kilka typowych sytuacji, w których można użyć polecenia `git diff`:
- **Przeglądanie zmian przed zatwierdzeniem.** Przed zatwierdzeniem zmian do repozytorium warto sprawdzić, co dokładnie zostało zmodyfikowane. Użycie polecenia `git diff` bez dodatkowych parametrów

wyświetla różnice między katalogiem roboczym a obszarem indeksowania. Pozwala to zobaczyć zmiany przygotowane do zatwierdzenia w porównaniu do ostatniego commitu.

- **Porównywanie zmian gotowych do zatwierdzenia z ostatnim commitem.** Aby przejrzeć zmiany przygotowane do zatwierdzenia w porównaniu do ostatniego commitu, można użyć polecenia `git diff --staged` (lub `git diff --cached`). Pozwala to przeprowadzić ostatnią kontrolę przed zatwierdzeniem, aby się upewnić, że wszystkie zmiany są zamierzone i poprawnie przygotowane.
- **Analiza różnic między gałęziami.** Aby poznać różnice między dwiema gałęziami, można użyć polecenia `git diff`, podając mu nazwy obu gałęzi. Na przykład polecenie `git diff main moja-galaz` pokaże, co się zmieniło między główną gałęzią a gałęzią z Twoimi zmianami. Jest to szczególnie przydatne podczas przygotowywania do scalania jednej gałęzi z drugą czy podczas sprawdzania potencjalnych konfliktów.
- **Badanie zmian między commitami.** Polecenie `git diff` można wykorzystać do przeglądania zmian między dwoma konkretnymi commitami. Podając hasze commitów, można zobaczyć różnice wprowadzone między wybranymi punktami w historii. Na przykład polecenie `git diff 1a2b3c4d 5e6f7g8h` porównuje stan repozytorium w commitcie `1a2b3c4d` ze stanem w commitcie `5e6f7g8h`.
- **Sprawdzanie różnic pojawiających się w konkretnym pliku.** Aby zobaczyć, jak konkretny plik zmieniał się w czasie, można podać jego nazwę wraz z referencjami do commitów. Na przykład polecenie `git diff HEAD~1 HEAD mojplik.txt` pokazuje, jak plik *mojplik.txt* zmienił się od ostatniego commitu do aktualnego wskazywanego przez HEAD.

5. Jakie są typowe zastosowania gałęzi w Gicie?

Odpowiedź: Oto kilka częstych sposobów użycia gałęzi w Gicie:

- **Tworzenie nowych funkcji aplikacji.** Gałęzie w Gicie najczęściej używane są do rozwijania nowych funkcji w aplikacji. Programiści tworzą nowe gałęzie wywodzące się z gałęzi głównej, aby pracować nad nową funkcją bez zakłócania podstawowej bazy kodu.
- **Naprawianie błędów.** Gałęzie są również używane do naprawiania błędów. Osobną gałąź tworzy się na potrzeby poprawki, aby uniknąć wpływu na prace prowadzone w głównej gałęzi.
- **Eksperymentowanie.** Gałęzie umożliwiają eksperymentowanie z nowymi technologiami, architekturami czy koncepcjami bez wpływania na stabilną wersję oprogramowania.
- **Zarządzanie wydaniem.** Gdy wersja oprogramowania jest gotowa do wydania, można utworzyć dla niej specjalną gałąź, gdzie będą wprowadzane tylko poprawki i aktualizacje dokumentacji.
- **Szybkie poprawki.** Gałęzie typu hotfix są używane do natychmiastowego rozwiązywania krytycznych błędów w wersjach produkcyjnych.

- **Współpraca.** Gałęzie ułatwiają współpracę w zespole, umożliwiając równoczesną pracę nad różnymi częściami projektu.
- **Przegląd kodu.** Przed scaleniem nowej funkcji z główną gałęzią programiści mogą przesłać swoje gałęzie do repozytorium, aby inni członkowie zespołu mogli przeprowadzić przegląd kodu, uruchomić testy i przedstawić swoje opinie.

Rozdział 4. Debugowanie i rozwiązywanie problemów z pamięcią

Oto sugerowane odpowiedzi na pytania:

1. Wymień kilka ważnych strategii debugowania.

Odpowiedź: Do ważnych strategii debugowania można zaliczyć:

- Zrozumienie problemu poprzez odtworzenie błędu.
- Zebranie jak największej ilości informacji o błędzie, w tym śladów stosu, protokołów oraz komunikatów o błędach.
- Użycie debuggera do wyizolowania problemu.
- Pisanie testów jednostkowych w celu weryfikacji, czy poszczególne części kodu działają zgodnie z oczekiwaniami, oraz stosowanie asercji do sprawdzania założeń dotyczących kodu.
- Prowadzenie protokołu prób rozwiązania problemu i uzyskanych wyników.
- Zachowanie spokoju i metodyczna praca. Czasami oderwanie się od problemu na jakiś czas pozwala lepiej dostrzec rozwiązanie.

2. Jaka jest różnica między oknami *Lokalne* i *Wyrażenie kontrolne* podczas debugowania?

Odpowiedź: Podczas debugowania w Visual Studio okna *Lokalne* i *Wyrażenia kontrolne* mają następujące zastosowania:

Okno *Lokalne* automatycznie wyświetla zmienne lokalne dla bieżącego kontekstu wykonania lub zakresu. Aktualizuje się w czasie rzeczywistym podczas przechodzenia przez kod, wyświetla zmienne i ich wartości dla aktualnie wykonywanej metody lub zakresu, w którym zatrzymano działanie podczas sesji debugowania. Automatycznie wyświetla wszystkie zmienne lokalne w bieżącej metodzie lub zakresie bez konieczności dodawania ich ręcznie. Pokazuje jedynie zmienne dostępne dla obecnego zakresu, co oznacza, że zmienia się podczas przechodzenia przez różne części kodu. Umożliwia szybki i łatwy podgląd stanu wykonania dokładnie w miejscu, w którym przerwano jego działanie, bez potrzeby wcześniejszej konfiguracji. Okno *Lokalne* przydaje się w sytuacjach, gdy kod jest głęboko zagnieżdżony w funkcji lub metodzie, a użytkownik chce szybko przeanalizować stan wszystkich lokalnych danych.

Okno *Wyrażenia kontrolne* służy do ręcznego monitorowania określonych zmiennych lub wyrażeń podczas sesji debugowania. W przeciwieństwie do okna *Lokalne* nie wypełnia się automatycznie, ale wymaga ręcznego dodania zmiennych lub wyrażeń, które mają być śledzone. Można dodać do niego zmienne lub wyrażenia pochodzące z dowolnego zakresu lub kontekstu dostępnego w bieżącym punkcie debugowania. Pozwala to monitorować wartości zmiennych lub wyrażeń, które nie muszą być lokalne dla aktualnego zakresu. Można dodać dowolne wyrażenie poprawne w bieżącym kontekście, a także wprowadzać wyrażenia składające się z wywołań metod i obliczeń. Po dodaniu zmiennych lub wyrażeń do okna *Wyrażenia kontrolne* pozostają one widoczne przez całą sesję debugowania (lub do momentu ich usunięcia), a użytkownik może obserwować, jak ich wartości zmieniają się w czasie wykonywania różnych części kodu. Jest to szczególnie przydatne, gdy konieczne jest ciągłe monitorowanie określonych zmiennych lub ocena wyrażeń związanych z różnymi zakresami. Na przykład w trakcie debugowania pętli można obserwować, jak zmienia się wartość zmiennej w każdej iteracji, nawet w przypadku, gdy nie jest to zmienna lokalna. Można też monitorować wynik wywołania funkcji albo stan złożonego wyrażenia.

3. W jaki sposób można kontrolować, co pojawia się w oknach debugowania, takich jak *Lokalne*?

Odpowiedź: Sposób wyświetlania różnych typów w tych panelach można kontrolować, dekorując kod specjalnymi atrybutami, takimi jak `DebuggerDisplay` i `DebuggerBrowsable`.

4. W oknach debugowania wartości typu `string` domyślnie są otoczone cudzysłowami. Jak można usunąć znaki cudzysłowu?

Odpowiedź: Aby usunąć cudzysłowy wokół nazwy pola, należy za nazwą pola dodać parametr `,nq`, tak jak w poniższym kodzie:

```
[DebuggerDisplay("{ProductId}: {ProductName,nq}")]
```

5. Jakie narzędzia dostępne w Visual Studio i JetBrains zostały przygotowane do analizy wspomaganie wydajności i użycia pamięci aplikacji?

Odpowiedź: W celu analizy wydajności aplikacji i wykorzystania pamięci Visual Studio udostępnia w debuggerze takie narzędzia jak profiler wydajności, analizator użycia pamięci lub profiler procesora. JetBrains oferuje takie narzędzia jak `dotPeek`, `dotTrace`, `dotMemory` i `dotCover`.

Rozdział 5. Protokołowanie, śledzenie i metryki obserwowalności

Oto proponowane odpowiedzi na pytania:

1. Jakie strategie można zastosować, aby zoptymalizować protokołowanie i obserwowalność w aplikacjach .NET?

Odpowiedź: Aby zoptymalizować protokołowanie i obserwowalność w aplikacjach .NET, należy stosować protokołowanie strukturalne, scentralizować procesy protokołowania, przyjąć identyfikator korelacyjny na potrzeby śledzenia żądań, wykorzystywać narzędzia obserwowalności, takie jak OpenTelemetry (OTel), wdrożyć system powiadomień i monitorowania oraz edukować cały zespół programistów.

2. Co programista .NET powinien wiedzieć o interfejsie ILogger?

Odpowiedź: Interfejs ILogger tworzy warstwę abstrakcji nad różnymi dostawcami mechanizmów protokołowania. Programiści mogą rozbudowywać funkcje protokołowania, implementując własnych dostawców. Interfejs ILogger obsługuje różne poziomy protokołowania komunikatów, co pozwala na kategoryzowanie ich według znaczenia. Pozwala też stosować protokołowanie strukturalne, co umożliwia tworzenie komunikatów z wyznaczonymi miejscami na wartości przekazywane osobno. Interfejs ILogger pozwala również definiować zakresy protokołowania, co ułatwia grupowanie operacji logicznie ze sobą powiązanych. Przydaje się to do wiązania wpisów na temat konkretnej operacji, żądania lub transakcji, co ułatwia śledzenie dzienników powiązanych z tym samym przepływem pracy lub żądaniem. Interfejs ILogger jest domyślnie zintegrowany z wbudowanym w .NET mechanizmem wstrzykiwania zależności. Ta integracja pozwala na wstrzykiwanie do klas obiektów typu `ILogger<T>`, gdzie T oznacza klasę, do której ILogger jest wstrzykiwany.

3. Dlaczego ważne jest, aby zrozumieć, że parametry message w metodach LogX są tylko szablonami komunikatów, a nie właściwymi komunikatami?

Odpowiedź: Informacja o tym, że parametr komunikatu w metodach LogX to szablon komunikatu, a nie pojedynczy komunikat, jest niezwykle ważna, ponieważ traktowanie tego parametru jako komunikatu i przekazywanie za każdym razem różnych wartości tekstowych powoduje alokację nowego łańcucha znaków przy każdym wywołaniu metody. Skutkuje to niepotrzebnym zużyciem pamięci i obniża wydajność. Aby tego uniknąć, należy definiować ograniczoną liczbę szablonów komunikatów i przekazywać różne wartości za pomocą parametru args.

4. Jaka jest różnica między protokołami, metrykami i alertami?

Odpowiedź: Protokoły to szczegółowe zapisy w historii zdarzeń, operacji lub transakcji w aplikacjach. Tworzą one chronologiczny i dokładny zapis tego, co i kiedy się wydarzyło. Protokoły zazwyczaj zawierają takie dane jak znaczniki czasu, komunikaty zdarzeń, komunikaty o błędach i informacje kontekstowe dotyczące stanu aplikacji w momencie rejestrowania komunikatu. Są użyteczne przy diagnozowaniu problemów. Mogą także pomagać administratorom i programistom w analizie sekwencji zdarzeń prowadzących do powstania konkretnego problemu. Ponadto mogą służyć do śledzenia aktywności użytkowników i zmian pojawiających się w systemie, co jest niezbędne do wypełnienia wymagań prawnych.

Metryki to miary ilościowe monitorujące różne aspekty wydajności i stanu aplikacji. Są to zazwyczaj dane liczbowe zbierane i agregowane w określonych przedziałach czasu. Przykładowe metryki to poziom wykorzystania procesora,

zużycie pamięci, czasy odpowiedzi i przepustowość. Metryki są niezbędnym elementem monitorowania wydajności aplikacji, a ich analiza jest podstawą efektywnego wykorzystania zasobów. Metryki długoterminowe pomagają w przewidywaniu trendów i podejmowaniu decyzji dotyczących zapotrzebowania na zasoby w przyszłości lub planów skalowania systemu.

Alerty to powiadomienia wywoływane przez określone warunki lub w wyniku przekroczenia progów monitorowanych danych, takich jak protokoły lub metryki. Alerty konfiguruje się na podstawie zasad. Na przykład można nakazać wygenerowanie alertu, jeżeli użycie procesora przez kilka minut będzie przekraczać poziom 90%. Alerty mają na celu zwrócenie natychmiastowej uwagi na potencjalne problemy wymagające pilnej reakcji. Pozwalają zespołom szybko reagować na wykrywane problemy, jeszcze zanim te urosną do poziomu krytycznego, jednocześnie redukując konieczność ręcznego monitorowania systemu.

Podsumowując: protokoły dostarczają informacji „dlaczego” — szczegółów dotyczących zdarzeń i kontekstu. Metryki odpowiadają na pytania „co” i „ile”, tworząc wysokopoziomowe zestawienie wydajności i trendów. Powiadomienia informują o tym, „kiedy” — natychmiastowo powiadamiają o problemach wymagających naszej reakcji.

5. Czym jest OpenTelemetry i dlaczego Microsoft zaleca jego używanie w projektach .NET?

Odpowiedź: OpenTelemetry (OTel) to framework obserwowalności. Udostępnia on API, pakiet SDK oraz współpracujące z nim narzędzia w celu generowania i zbierania danych telemetrycznych, takich jak metryki i protokoły. Kluczowe zalety stosowania OTel w projektach .NET to wspólny mechanizm zbierania danych, ujednolicone schematy i semantyka danych telemetrycznych oraz API umożliwiające integrację systemów analizy danych z frameworkiem OTel. Dzięki temu, że .NET definiuje własne API do protokołowania, nie trzeba uczyć się dodatkowego API, po prostu framework OpenTelemetry integruje się z interfejsem ILogger.

Rozdział 6. Dokumentowanie kodu, API i serwisów

Oto proponowane odpowiedzi na pytania:

1. Wymień dobre praktyki dodawania komentarzy do kodu źródłowego.

Odpowiedź: Wśród dobrych praktyk komentowania kodu źródłowego można wymienić:

- **Ciągle aktualizowanie komentarzy.** Nieaktualne komentarze mogą wprowadzać w błąd, co bywa gorsze niż brak komentarzy. Należy się upewnić, że komentarze są aktualizowane wraz z kodem, który opisują.
- **Unikanie zbędnych komentarzy.** Nie należy opisywać oczywistych rzeczy. Komentarze powinny wносить dodatkowe informacje, wykraczające poza to, co jest już jasne na podstawie kodu.

- **Wykorzystywanie kodu jako dokumentacji.** Należy dążyć do tego, aby kod był samodokumentujący. Używanie czytelnych konwencji nazewniczych, refaktoryzacja złożonych bloków kodu do dobrze nazwanych funkcji oraz intuicyjna struktura kodu ułatwiają zrozumienie intencji całego kodu.
 - **Wykorzystywanie komentarzy dokumentacyjnych.** Wiele języków obsługuje specjalne formaty komentarzy dokumentacyjnych, które mogą być używane do automatycznego generowania zewnętrznej dokumentacji. Należy stosować je dla publicznych interfejsów oraz API.
2. Jak udokumentować parametr metody za pomocą komentarzy XML-a?
- Odpowiedź:** Aby udokumentować parametr metody przy użyciu komentarzy XML-a, należy dodać element `<param>` z atrybutem `name` i opisem, tak jak w poniższym kodzie:
- ```
/// <param name="culture">Definiuje kod kultury ISO, np. pl-PL, en-GB
/// lub es-AR.</param>
/// <param name="useComputerCulture">Nadaj mu wartość true, aby zmienić kulturę na
/// kulturę lokalnego komputera.</param>
public static void ConfigureConsole(string culture = "en-US",
bool useComputerCulture = false)
```
3. Czy można zastosować style w tekstach odpowiedzi z komentarzy XML-a?
- Odpowiedź:** W odpowiedziach nie można stosować złożonych stylizacji, ale można używać znaczników `<b>`, `<i>` i `<u>`, aby wprowadzić pogrubienie, kursywę i podkreślenie tekstu. Można również wyróżnić odniesienia do kodu, stosując znacznik `<c>` dla kodu w tekście oraz `<code>` dla całych bloków kodu. Pomaga to odróżnić fragmenty kodu lub nazwy zmiennych od reszty tekstu w dokumentacji XML-a.
4. Czym jest DocFX?
- Odpowiedź:** DocFX generuje statyczną stronę internetową na podstawie kodu źródłowego (na potrzeby dokumentowania publicznych interfejsów API) oraz plików Markdown (są one źródłem dodatkowych, spersonalizowanych stron dokumentacji). Układ i styl strony można dostosować za pomocą szablonów.
5. Jak pokazać dziedziczenie w diagramie klas Mermaid?

**Odpowiedź:** W diagramie klas Mermaid dziedziczenie przedstawia się za pomocą symbolu `<|--`, tak jak w poniższym zapisie:

```
classDiagram
 class Stream {
 <<abstract>>
 ...
 }

 class MemoryStream {
 ...
 }
```

```
class FileStream {
 ...
}

Stream <|-- MemoryStream
Stream <|-- FileStream
```

## Rozdział 7. Obserwowanie kodu i dynamiczne wpływanie na jego wykonanie

Oto sugerowane odpowiedzi na pytania:

1. Wymień cztery części składowe zestawu .NET i powiedz, które z nich są opcjonalne.

**Odpowiedź:** Zestaw składa się z trzech obowiązkowych części i jednej opcjonalnej:

- **Manifest i metadane zestawu** — nazwa, zestaw, wersja pliku, wykorzystywane zestawy itd.
- **Metadane typów** — informacje o typach, ich składnikach itd.
- **Kod IL** — implementacja metod, właściwości, konstruktorów itd.
- **Osadzone zasoby** (opcjonalne) — obrazy, ciągi znaków, JavaScript itd.

2. Do czego można zastosować atrybut?

**Odpowiedź:** Istnieje typ wyliczeniowy definiujący elementy, do których można zastosować atrybut:

```
namespace System
{
 [Flags]
 public enum AttributeTargets
 {
 Assembly = 1,
 Module = 2,
 Class = 4,
 Struct = 8,
 Enum = 16,
 Constructor = 32,
 Method = 64,
 Property = 128,
 Field = 256,
 Event = 512,
 Interface = 1024,
 Parameter = 2048,
 Delegate = 4096,
 ReturnValue = 8192,
 GenericParameter = 16384,
 All = 32767
 }
}
```

3. Jak się nazywają i co oznaczają części numeru wersji, jeżeli taki numer przestrzega zasad semantycznego wersjonowania?

**Odpowiedź:** Numery wersji w .NET są kombinacją trzech liczb z dwoma opcjonalnymi dodatkami: wersją wstępną (ang. *Prerelease*) i numerem kompilacji (ang. *Build number*). Zgodnie z zasadami wersjonowania semantycznego wspomniane trzy liczby oznaczają:

- **Główna** — zmiany powodujące niezgodność wsteczną.
- **Poboczna** — zmiany niepowodujące niezgodności wstecznej, w tym nowe funkcjonalności i często poprawki błędów.
- **Poprawka** — poprawki błędów niepowodujące niezgodności wstecznej.

4. Jak uzyskać referencję zestawu aktualnej aplikacji konsoli?

**Odpowiedź:** Aby uzyskać referencję zestawu aktualnie wykonywanej aplikacji konsoli, należy wywołać metodę `Assembly.GetEntryAssembly()`.

5. Jak uzyskać wszystkie atrybuty zastosowane dla zestawu?

**Odpowiedź:** Aby pobrać wszystkie atrybuty zastosowane wobec zestawu, należy wywołać metodę `GetCustomAttributes()`, podając jej referencję zestawu.

6. Jak można utworzyć własny atrybut?

**Odpowiedź:** Aby utworzyć własny atrybut, należy utworzyć klasę dziedziczącą po klasie `Attribute`. Do takiej klasy należy dodać atrybut `[AttributeUsage]`, który pozwala określić, gdzie można stosować nowy atrybut. Szczegóły dotyczące miejsc stosowania atrybutów znajdują się w odpowiedzi na pytanie 2.

7. Po jakiej klasie należy dziedziczyć, aby umożliwić dynamiczne ładowanie zestawu?

**Odpowiedź:** Aby umożliwić dynamiczne ładowanie zestawów, należy dziedziczyć po klasie `AssemblyLoadContext`.

8. Czym jest drzewo wyrażeń?

**Odpowiedź:** Drzewa wyrażeń reprezentują kod jako strukturę, którą można analizować lub wykonywać.

9. Czym jest generator kodu?

**Odpowiedź:** Generatory kodu źródłowego pozwalają uzyskać obiekt kompilacji reprezentujący cały kod poddawany kompilacji, a następnie dynamicznie generować dodatkowe pliki kodu i je również kompilować.

10. Jaki interfejs musi implementować klasa generatora kodu i jakie metody wchodzi w skład tego interfejsu?

**Odpowiedź:** Klasa generatora kodu źródłowego musi implementować interfejs `ISourceGenerator`, który zawiera metody `Initialize` oraz `Execute`.

## Rozdział 8. Ochrona danych i aplikacji za pomocą kryptografii

Oto sugerowane odpowiedzi na pytania:

1. Który z algorytmów szyfrowania oferowanych przez .NET jest najlepszym wyborem w przypadku szyfrowania symetrycznego?

**Odpowiedź:** W przypadku szyfrowania symetrycznego najlepszym wyborem jest algorytm AES.

2. Który z algorytmów szyfrowania oferowanych przez .NET jest najlepszym wyborem w przypadku szyfrowania asymetrycznego?

**Odpowiedź:** W przypadku szyfrowania asymetrycznego najlepszym wyborem jest algorytm RSA.

3. Co to jest atak tęczowy?

**Odpowiedź:** Atak tęczowy wykorzystuje tabelę wstępnie obliczonych skrótów haseł. Jeżeli baza danych skrótów haseł zostanie skradziona, atakujący może szybko porównać je z wartościami w tabeli tęczowej i odszukać oryginalne hasła.

4. Czy w algorytmach szyfrowania lepszy jest większy, czy mniejszy rozmiar bloku?

**Odpowiedź:** W przypadku algorytmów szyfrowania lepiej jest mieć mniejszy rozmiar bloku.

5. Co to jest kryptograficzna funkcja haszująca?

**Odpowiedź:** Kryptograficzna funkcja haszująca daje wynik o stałym rozmiarze, uzyskany poprzez przetworzenie danych wejściowych o dowolnym rozmiarze przez funkcję skrótu. Funkcje skrótu są jednokierunkowe, co oznacza, że jedynym sposobem na odtworzenie oryginalnych danych jest sprawdzenie wszystkich możliwych wartości wejścia i porównanie wyników.

6. Co to jest podpis kryptograficzny?

**Odpowiedź:** Podpis kryptograficzny to wartość dołączona do cyfrowego dokumentu w celu potwierdzenia jego autentyczności. Poprawny podpis informuje odbiorcę, że dokument został utworzony przez znanego mu nadawcę i nie został zmodyfikowany.

7. Jaka jest różnica między szyfrowaniem symetrycznym a asymetrycznym?

**Odpowiedź:** Szyfrowanie symetryczne wykorzystuje współdzielony tajny klucz i używa go zarówno do szyfrowania, jak i odszyfrowywania wiadomości. Szyfrowanie asymetryczne wykorzystuje klucz publiczny do szyfrowania i klucz prywatny do odszyfrowywania.

8. Co oznacza skrót RSA?

**Odpowiedź:** RSA to skrót od nazwisk trzech badaczy, którzy opisali ten algorytm w 1978 r. — Rivest, Shamir i Adlema.

9. Dlaczego hasła powinny być solone przed zapisaniem?

**Odpowiedź:** Hasła należy solić przed zapisaniem, aby spowolnić ataki słownikowe wykorzystujące tablice tęczowe.

10. SHA-1 to algorytm haszujący zaprojektowany przez amerykańską Agencję Bezpieczeństwa Narodowego (NSA). Dlaczego nie należy go używać?

**Odpowiedź:** Algorytmu SHA-1 nie należy używać, ponieważ nie jest już bezpieczny. Wszystkie nowoczesne przeglądarki przestały akceptować certyfikaty SSL bazujące na SHA-1.

## Rozdział 9. Tworzenie chatu używającego modelu LLM

Oto sugerowane odpowiedzi na pytania:

1. Czym jest Semantic Kernel?

**Odpowiedź:** Semantic Kernel to otwartoźródłowy pakiet SDK firmy Microsoft, który umożliwia łatwe tworzenie agentów mogących wywoływać istniejący kod. Semantic Kernel można używać z modelami OpenAI, Azure OpenAI, Hugging Face i innych dostawców. Łącząc istniejące projekty C# i .NET z tymi modelami, można budować agenty odpowiadające na pytania lub automatyzujące procesy.

2. Co robi metoda `CreateFunctionFromMethod`?

**Odpowiedź:** Metoda `CreateFunctionFromMethod` pozwala rozszerzać ogólne możliwości modelu LLM o własne funkcje. Każda funkcja jest opisana w języku naturalnym w pliku JSON, aby model LLM wiedział, do czego służy i jakich wymaga parametrów. Przy wywoływaniu funkcji trzeba wykonać następującą sekwencję operacji:

- Użytkownik wysyła wiadomość wraz z zestawem funkcji powiązanych z metodami .NET.
- Model LLM może wybrać wywołanie jednej lub kilku funkcji albo skorzystać ze swojego standardowego modelu.
- Aplikacja wykonuje metodę i przesyła jej wynik do modelu LLM.
- Model LLM wykorzystuje wynik metody jako dodatkowy kontekst do wygenerowania odpowiedzi.

3. Domyślnie każdy prompt wysyłany do modelu LLM za pomocą biblioteki Semantic Kernel jest niezależny od pozostałych promptów. Jak dodać pamięć sesji do chatu, aby model zapamiętywał wcześniejsze prompty?

**Odpowiedź:** Aby dodać pamięć sesji do chatu, można utworzyć obiekt klasy `ChatHistory` i wywoływać metodę `AddUserMessage`, przekazując jej każdą kolejną wiadomość. Następnie należy przekazać obiekt `ChatHistory` do metody `GetChatMessageContentAsync`, a cała historia będzie wysyłana do modelu LLM wraz z każdym kolejnym żądaniem. Dodatkowo po uzyskaniu odpowiedzi z modelu należy wywołać metodę `AddAssistantMessage`, przekazując jej otrzymaną wiadomość.



4. Domyślnie musisz czekać, aż cała odpowiedź zostanie zwrócona przez model LLM. Jak włączyć strumieniowanie odpowiedzi?

**Odpowiedź:** Aby włączyć strumieniowanie, można użyć asynchronicznej pętli `foreach`, tak jak w poniższym kodzie:

```
await foreach (StreamingChatMessageContent message
 in completion.GetStreamingChatMessageContentsAsync(history))
```

5. Czym jest Hugging Face?

**Odpowiedź:** Hugging Face to firma znana z innowacji w dziedzinie sztucznej inteligencji, szczególnie w zakresie przetwarzania języka naturalnego. Zdobyła dużą popularność dzięki wkładowi w otwarte oprogramowanie oraz rozwijaniu biblioteki Transformers, która udostępnia najnowocześniejsze modele uczenia maszynowego do różnych zadań NLP.

Biblioteka Transformers jest często stosowana w społeczności AI. Oferuje wstępnie wytrenowane modele, które mogą być wykorzystywane do takich zadań jak klasyfikacja tekstu, ekstrakcja informacji, odpowiadanie na pytania, podsumowywanie oraz tłumaczenie. Oprócz biblioteki Transformers firma Hugging Face prowadzi hub modeli, w którym znajdują się tysiące gotowych modeli udostępnionych przez społeczność. Te modele można łatwo pobierać i wykorzystywać w różnych projektach NLP.

## Rozdział 10. Wstrzykiwanie zależności, kontenery i czas życia serwisów

Oto sugerowane odpowiedzi na pytania:

1. Jaka jest główna idea wstrzykiwania zależności?

**Odpowiedź:** Główną ideą wstrzykiwania zależności jest oddzielenie tworzenia zależności obiektu od jego własnego zachowania, co pozwala na uzyskanie kodu bardziej modułowego, testowalnego i łatwiejszego w utrzymaniu. Obiekty nie tworzą zależności samodzielnie, ale otrzymują je w trakcie działania aplikacji, często za pośrednictwem zewnętrznego frameworka lub kontenera.

2. Jakie są trzy główne rodzaje wstrzykiwania zależności? Który rodzaj nie jest bezpośrednio obsługiwany w projektach .NET poza specjalnymi scenariuszami?

**Odpowiedź:** Istnieją trzy główne sposoby wstrzykiwania zależności:

- 1) **Wstrzykiwanie przez konstruktor.** Zależności są przekazywane do konstruktora klasy. Jest to najlepsza praktyka, ponieważ umożliwia najłatwiejsze tworzenie atrap (mocking) usług podczas testowania.
- 2) **Wstrzykiwanie przez metodę.** Zależności są przekazywane jako argumenty metod.
- 3) **Wstrzykiwanie przez właściwość.** Zależności są przypisywane właściwościom klasy. Ten typ wstrzykiwania nie jest bezpośrednio obsługiwany w projektach .NET, ale można go wprowadzić przy użyciu takich narzędzi jak Autofac.

3. W .NET możesz rejestrować zależności z różnymi czasami życia. Jakimi?

**Odpowiedź:** Cykle życia usług zależności to:

- **Transient.** Usługi są tworzone przy każdym żądaniu. Ten rodzaj usług powinien być lekki i bezstanowy.
- **Scoped.** Usługi są tworzone raz na każde żądanie otrzymane od klienta i usuwane po zakończeniu przygotowywania odpowiedzi dla klienta.
- **Singleton.** Usługi są zwykle tworzone przy pierwszym żądaniu i później są współdzielone. Można też przekazać instancję usługi już podczas rejestracji.

4. Jakich zdarzeń możesz nasłuchiwać w usługach hosta?

**Odpowiedź:** Interfejs `IHostedService` definiuje dwie metody, które działają jak zdarzenia: `StartAsync` i `StopAsync`. Można również przygotować obsługę zdarzeń `OnStarted`, `OnStopping` i `OnStopped`.

5. Kiedy są uruchamiane usługi hosta?

**Odpowiedź:** Serwisy hosta uruchamiają się po wywołaniu metody `RunAsync`.

## Rozdział 11. Testowanie i mockowanie

Oto sugerowane odpowiedzi na pytania:

1. Jakie kryteria powinien spełniać dobry test jednostkowy?

**Odpowiedź:** Dobry test jednostkowy powinien spełniać następujące kryteria:

- Weryfikuje pojedynczą jednostkę zachowania, np. metodę implementującą logikę biznesową.
- Wykonuje się możliwie najszybciej. Na przykład zamiast produkcyjnej bazy danych może używać bazy w pamięci, co przyspiesza testowanie logiki biznesowej. Dobry framework testowy pozwala również ustawić limit czasu wykonywania testu.
- Działa w izolacji od innych testów (oraz opcjonalnie od swoich zależności).

2. Czym jest MUT?

**Odpowiedź:** MUT (ang. *Method Under Test*) to metoda w ramach SUT (ang. *System Under Test*), która jest poddawana testom. Skrót SUT odnosi się do testowanego systemu lub typu. Często tworzymy klasę testową zawierającą wiele metod testowych, aby grupować testy dla danego SUT.

3. Proces *Test-Driven Development* (TDD) jest opisywany jako „czerwony — zielony — refaktoryzacja”. Co oznaczają te trzy kroki?

**Odpowiedź:** Trzy kroki w podejściu TDD, znane jako „czerwony — zielony — refaktoryzacja”, to:

- Czerwony — napisanie testu dla nowej funkcji. Test powinien początkowo nie przechodzić, ponieważ testowana funkcja jeszcze nie istnieje. Ten krok sprawia, że testy mają znaczenie i rzeczywiście weryfikują zamierzoną funkcję systemu.

- Zielony — napisanie minimalnej ilości kodu koniecznej do przejścia testu. Zachęca to do prostoty i skupienia się tylko na wymaganej funkcji.
- Refaktoryzacja — poprawianie napisanego przed chwilą kodu, aby lepiej pasował do istniejącej bazy kodu, był zgodny z dobrymi praktykami, a także czytelny i łatwy w utrzymaniu. Ważne jest, aby po refaktoryzacji wszystkie testy nadal przechodziły, co daje pewność braku regresji.

4. Jak konfiguruje się testy w xUnit?

**Odpowiedź:** Testy w xUnit konfigurowane są za pomocą atrybutów .NET, dzięki czemu kod testowy jest czytelny i łatwy do zrozumienia. Do standardowych przypadków testowych używa się atrybutu `[Fact]`, a do testów parametryzowanych atrybutów `[Theory]` z atrybutami `[InlineData]`, `[ClassData]` lub `[MemberData]`, które umożliwiają przeprowadzanie testów wykorzystujących różne dane.

5. Jaki parametr w xUnit mają atrybuty `[Fact]` i `[Theory]`, którego definiowanie uznawane jest za dobrą praktykę? Dlaczego?

**Odpowiedź:** Atrybuty `[Fact]` i `[Theory]` umożliwiają ustawienie limitu czasu wykonywania testu w milisekundach, tak jak w poniższym kodzie:

```
[Fact(Timeout = 3000)] // Test zakończy się po 3 sekundach.
```

Zdefiniowanie limitu czasu zapobiega zbyt długiemu wykonywaniu testów.

6. Aby dostarczyć dane do testu z atrybutem `[Theory]`, możesz użyć atrybutu `[ClassData]` i wskazać klasę. Jakie są wymagania wobec tej klasy?

**Odpowiedź:** Klasa określona w atrybucie `[ClassData]` musi implementować interfejs `IEnumerable<object[]>`, co oznacza, że musi zawierać zarówno generyczną, jak i niegeneryczną metodę `GetEnumerator`, zwracającą sekwencję tablic obiektów. Klasa może też być silnie typowana poprzez dziedziczenie z klasy `TheoryData`.

7. Jakie są dobre praktyki dotyczące obsługi pozytywnych i negatywnych wyników testów?

**Odpowiedź:** Dobrą praktyką jest rozdzielanie pozytywnych i negatywnych wyników testów do osobnych metod testowych z odpowiednimi zestawami parametrów. Jeżeli scenariusz jest zbyt skomplikowany i nie można łatwo zrozumieć znaczenia wartości wejściowych na podstawie samych parametrów, warto utworzyć osobną metodę testową z atrybutem `[Fact]` dla tego konkretnego przypadku.

8. Jakie sygnały ostrzegawcze powinny zwrócić naszą uwagę podczas pisania lub przeglądania testów jednostkowych?

**Odpowiedź:** Oto potencjalne problemy:

- Używanie instrukcji `if` lub `switch` w testach to antywzorzec, ponieważ test jednostkowy nie powinien zawierać rozgałęzień.
- Sekcja *Arrange* (przygotowanie testu) jest zbyt rozbudowana. Warto wtedy wydzielić osobne metody pomocnicze.

- Sekcja *Act* (wywołanie testowanego kodu) powinna zazwyczaj zawierać jedno wywołanie. Jeżeli test wymaga wielu wywołań metod, może to oznaczać problem z projektowaniem interfejsu API.
  - Sekcja *Assert* (weryfikacja wyniku) powinna zwykle zawierać jedno wyrażenie. Można używać wielu asercji, gdy jednostka zachowania zwraca wiele wartości, ale warto rozważyć refaktoryzację API w taki sposób, aby zwracana była jedna wartość zbiorcza, np. rekord zawierający wszystkie wymagane wartości.
9. Jak w xUnit można zobaczyć dane wyjściowe podczas wykonywania testów?
- Odpowiedź:** W xUnit nie można używać metody `Console.WriteLine`. Zamiast niej należy użyć interfejsu `ITestOutputHelper`, który jest wstrzykiwany do konstruktora klasy testowej.
10. Jak skonfigurować wartość zwracaną przez zmockowaną metodę przy użyciu `NSubstitute`?
- Odpowiedź:** W `NSubstitute` konfigurację wartości zwracanej przez metodę można zdefiniować za pomocą metody `Returns`, tak jak w poniższym przykładzie:
- ```
ICalculator calc = Substitute.For<ICalculator>();  
calc.Add(2, 3).Returns(5);
```

Rozdział 12. Testy integracyjne i testy bezpieczeństwa

Oto sugerowane odpowiedzi na pytania:

1. Czym różnią się od siebie dublery testowe, mocki, szpiegi, atrapy, fałszywki i manekiny?

Odpowiedź: Testowy dubler to ogólny termin odnoszący się do każdej fałszywej zależności w teście. Są one wykorzystywane w testach zamiast rzeczywistych zależności, które byłyby trudne do konsekwentnego skonfigurowania w porównaniu do dublerów. Istnieje wiele typów dublerów. Najczęściej spotykane to mocki i atrapy:

- Mocki to dublery dla interakcji wychodzących. Na przykład test może wywołać zmockowaną zależność, która symuluje wysyłanie e-maila podczas rejestracji użytkownika. Stan w zewnętrznym systemie może zostać zmieniony. Mocki zwykle tworzy się za pomocą frameworku do mockowania. Gdy są tworzone ręcznie, czasami nazywa się je szpiegami.
- Atrapy to dublery dla interakcji przychodzących. Na przykład test może wywołać atrapę zależności, która pobiera informacje o produkcie z bazy danych. Stan w zewnętrznym systemie nie jest zmieniany. Gdy zależność jeszcze nie istnieje, np. w przypadku stosowania techniki TDD, wtedy atrapa nazywana jest fałszywką. Gdy atrapa jest prostą wartością i nie wpływa na wynik, nazywa się ją manekinem.

2. W przypadku testów integracyjnych jakie typy systemów zewnętrznych należy testować, a które należy symulować?

Odpowiedź: Zewnętrzne systemy dzielą się na dwa typy: te, które znajdują się pod naszą kontrolą, oraz te, które są poza naszą kontrolą. Zewnętrzne systemy znajdujące się pod naszą kontrolą to magazyny danych, do których dostęp ma tylko nasz projekt. Żaden inny system nie zmienia tych danych. Zewnętrzne systemy poza naszą kontrolą to systemy e-mailowe oraz usługi publiczne, takie jak systemy pogodowe lub rządowe. Powinniśmy bezpośrednio używać zewnętrznych systemów, które znajdują się pod naszą kontrolą, natomiast mockować te, które są poza naszą kontrolą.

3. Czym są tunele deweloperskie i jak mogą z nich korzystać programiści .NET?

Odpowiedź: Tunele deweloperskie pozwalają programistom na bezpieczne udostępnianie swojego lokalnego środowiska deweloperskiego w publicznym internecie. Taka konfiguracja jest szczególnie przydatna, gdy musisz udostępnić swoją pracę kolegom, klientom lub usługom zewnętrznym, ale nie chcesz wdrażać jej na publicznym środowisku pośrednim. Dla programistów .NET pracujących z Visual Studio istnieją nawet mechanizmy bezpośredniej integracji z niektórymi narzędziami do tunelowania, co upraszcza cały proces konfiguracji. Możliwość szybkiego udostępnienia lokalnie rozwijanej witryny, a następnie poprawianie jej w czasie rzeczywistym w reakcji na uzyskane komentarze, może znacznie przyspieszyć cykle deweloperskie i poprawić jakość ostatecznego produktu.

4. Od czego należy zacząć testowanie bezpieczeństwa?

Odpowiedź: W przypadku testowania bezpieczeństwa profesjonalny programista .NET powinien zacząć od listy OWASP Top 10 (Open Web Application Security Project), która przedstawia najistotniejsze ryzyka związane z bezpieczeństwem aplikacji webowych. Powinien przestrzegać zasad bezpiecznego tworzenia kodu, które pomagają unikać powstawania podatności. Dla programistów .NET oznacza to umiejętność bezpiecznej obsługi danych wejściowych od użytkownika, wdrażania uwierzytelniania i autoryzacji, bezpiecznego zarządzania sesjami oraz szyfrowania wrażliwych danych.

5. Czym jest Microsoft Security Development Lifecycle?

Odpowiedź: Microsoft Security Development Lifecycle (SDL) to proces tworzenia oprogramowania, który pomaga programistom w budowaniu bezpieczniejszego oprogramowania oraz w spełnianiu wymagań dotyczących zgodności z bezpieczeństwem, przy jednoczesnej redukcji kosztów całego procesu. SDL oferuje wskazówki, praktyki i narzędzia dla wszystkich etapów tworzenia oprogramowania, w tym modelowania zagrożeń.

Rozdział 13. Mierzenie wydajności i testy obciążeniowe

Oto sugerowane odpowiedzi na pytania:

1. Wyjaśnij, czym jest notacja dużego O.

Odpowiedź: Notacja dużego O jest notacją matematyczną używaną do opisywania wydajności lub złożoności algorytmu. Podaje ona złożoność czasową lub pamięciową w zależności od rozmiaru danych wejściowych (oznaczanych jako n). Notacja ta informuje o górnej granicy dla czasu lub pamięci wymaganych przez algorytm, dzięki czemu umożliwia porównanie efektywności algorytmów bez konieczności uwzględniania szczegółów implementacyjnych czy specyfikacji sprzętowych. Notacja dużego O zazwyczaj opisuje złożoność w najgorszym przypadku, co pomaga poznać maksymalną ilość zasobów, której może wymagać algorytm. Opisuje również, jak algorytm zachowuje się w miarę wzrostu rozmiaru danych wejściowych. Dzięki temu można ocenić skalowalność i wydajność algorytmów. Stosowanie notacji dużego O pozwala programistom i specjalistom od informatyki podejmować decyzje dotyczące wyboru algorytmów najlepiej dopasowanych do konkretnych problemów, zwłaszcza w kontekście skalowania danych.

2. Opisz powszechny błąd w testach wydajności i podaj narzędzia, jakie udostępniła nam biblioteka *BenchmarkDotNet*, aby go unikać.

Odpowiedź: Do częstych błędów podczas testowania wydajności należą:

- Brak izolacji kodu poddanego testowaniu od logiki inicjalizującej lub sprzątającej. Biblioteka *BenchmarkDotNet* udostępnia atrybuty `[GlobalSetup]` oraz `[GlobalCleanup]`, które pozwalają oddzielić kod odpowiedzialny za inicjalizację i sprząatanie od właściwego kodu testowego.
 - Optymalizacje wykonywane przez kompilator JIT (*Just-In-Time*) w trakcie działania programu, które mogą wpływać na wyniki testów, jeżeli nie zostaną odpowiednio uwzględnione. Biblioteka *BenchmarkDotNet* automatycznie zarządza tzw. rozgrzewaniem kompilatora JIT.
 - Porównywanie wyników uzyskanych na różnych maszynach, w różnych środowiskach lub konfiguracjach, co może prowadzić do mylnych wniosków. Biblioteka *BenchmarkDotNet* umożliwia eksportowanie i porównywanie wyników, ale ważne jest, aby porównania były wykonywane w zbliżonych warunkach oraz by uwzględniać różnice w sprzęcie czy konfiguracji środowiska.
3. Na czym polega różnica między testami obciążeniowymi a testami wydajnościowymi?

Odpowiedź: Testy obciążeniowe koncentrują się na analizowaniu wydajności systemu pod określonym, oczekiwanym obciążeniem. Obciążenie to odnosi się do typowych warunków, które system ma obsługiwać w rzeczywistym scenariuszu, takich jak liczba jednoczesnych użytkowników lub wolumen transakcji. Podczas testowania obciążenia system jest poddawany rosnącemu

obciążeniu, np. większej liczbie żądań użytkowników, operacji bazodanowych czy procesów, aby zasymulować standardowe warunki działania. Celem jest określenie momentu, w którym wydajność systemu zaczyna się pogarszać lub w którym system przestaje spełniać oczekiwania przy przewidywanym obciążeniu. Przykładem może być komercyjna witryna przygotowująca się na wyprzedaż z okazji Black Friday, gdzie spodziewana liczba użytkowników i transakcji jest większa niż zwykle, ale nadal mieści się w przewidywanych granicach.

Testy wydajnościowe mają na celu określenie wytrzymałości systemu i sposobu jego reagowania na ekstremalne warunki. Są one projektowane w taki sposób, aby znaleźć punkt krytyczny systemu poprzez poddanie go obciążeniu znacznie przekraczającemu normalne możliwości operacyjne. Ten rodzaj testowania polega na zastosowaniu nierealistycznie wysokiego obciążenia, które prawdopodobnie doprowadzi do awarii systemu. Celem jest przede wszystkim określenie momentu wystąpienia awarii, jak również zbadanie, w jaki sposób system ulega awarii i jak się po niej regeneruje, np. w kontekście wycieków pamięci, problemów z synchronizacją czy uszkodzeń danych. Testowanie wydajnościowe może polegać na symulowaniu nagłych skoków ruchu użytkowników, znacznie przewyższających normalne wolumeny, lub intensywnym obciążaniu zasobów obliczeniowych. Ma to na celu sprawdzenie, czy system jest w stanie obsłużyć niespodziewane wzrosty obciążenia i powrócić do normalnej pracy.

Testowanie obciążeniowe ma sprawdzać, czy aplikacja może obsłużyć spodziewany ruch zgodnie ze specyfikacją, natomiast testowanie wydajnościowe pozwala określić granice systemu i jego zachowanie w warunkach skrajnych. Testy obciążeniowe analizują wydajność systemu w typowych warunkach, podczas gdy testy odpornościowe zmuszają system do wyjścia poza jego normalne zdolności operacyjne, aby sprawdzić, czy jest w stanie przetrwać nagłe lub rzadkie skoki obciążenia. Głównym celem testów obciążeniowych jest optymalizacja wydajności w standardowych warunkach, aby zapewnić płynną realizację wymagań użytkowników. Z kolei testy wydajnościowe koncentrują się na sprawdzaniu odporności i stabilności systemu, aby się upewnić, że nie ulega on katastrofalnej awarii pod ekstremalnym obciążeniem i potrafi wrócić do normalnej pracy.

4. Czym jest Bombardier i dlaczego jest użyteczny dla programistów?

Odpowiedź: Bombardier to wysokowydajne wielopatformowe narzędzie do testowania wydajności napisane w języku Go. Zostało zaprojektowane do generowania znacznego obciążenia w celu testowania wydajności serwerów i serwisów webowych, minimalizując przy tym wpływ na system. Bombardier może wysyłać dużą liczbę żądań na sekundę, co czyni go odpowiednim narzędziem zarówno do testowania wydajnościowego, jak i do testowania obciążenia aplikacji webowych. Pozwala to na analizę zachowania systemu przy dużym ruchu generowanym przez użytkowników.

5. Z czego składa się scenariusz w NBomberze?

Odpowiedź: Scenariusz w NBomberze składa się z następujących elementów:

- **Kroki** (ang. *Steps*). Są to pojedyncze operacje lub akcje, które będą wykonywane. Na przykład w kontekście aplikacji webowej krokiem może być konkretne żądanie HTTP wysyłane do punktu końcowego.
- **Symulacje obciążenia** (ang. *Load simulations*). Określają sposób generowania obciążenia, precyzując liczbę jednoczesnych użytkowników lub żądań oraz czas trwania testu. NBomber obsługuje różne strategie symulacji obciążenia, takie jak symulacja stabilnego ruchu przez określony czas lub stopniowe zwiększanie bądź zmniejszanie obciążenia.
- **Ustawienia scenariusza** (ang. *Scenario settings*). Dodatkowe ustawienia scenariusza, takie jak globalne nagłówki dla żądań HTTP czy operacje przygotowawcze i sprzątające, które powinny zostać wykonane zarówno przed uruchomieniem scenariusza, jak i po uruchomieniu.

Rozdział 14. Testy funkcjonalne i end-to-end

Oto sugerowane odpowiedzi na pytania:

1. Jakich przeglądarek używa Playwright do uruchamiania testów?

Odpowiedź: Playwright używa otwartoźródłowych wersji Chromium. Może również działać z innymi przeglądarkami z tej rodziny dostępnymi na danym komputerze. W szczególności aktualna wersja Playwrighta obsługuje kanały Stable i Beta przeglądarek z rodziny Chromium. Wersja Playwrighta dla Firefoksa używa najnowszego wydania Firefox Stable. Wersja Playwrighta dla WebKity korzysta z najnowszego wydania WebKit trunk, jeszcze zanim zostanie ono użyte w Apple Safari i innych przeglądarkach bazujących na silniku WebKity. Playwright nie współpracuje z niestandardowymi wersjami Firefoksa ani Safari, ponieważ wykorzystują one dodatkowe łatki na standardową przeglądarkę.

2. Wymień główne interfejsy reprezentujące ważne obiekty używane podczas pisania testów dla Playwrighta.

Odpowiedź: Główne interfejsy reprezentujące istotne obiekty podczas pisania testów w Playwrightcie to `IPlaywright`, `IBrowser`, `IBrowserContext`, `IResponse`, `IPage` oraz `ILocator`.

3. Jakie metody udostępniane przez Playwrighta pozwalają uzyskać przynajmniej jeden element ze strony WWW?

Odpowiedź: Playwright udostępnia przydatne metody do pobierania jednego lub więcej elementów na stronie internetowej, takie jak `GetByRole`, `GetByLabel`, `GetByPlaceholder`, `GetByTestId`, `GetByText`, `GetByTitle` oraz `GetByAltText`.

4. Co się stanie, jeżeli więcej niż jeden element pasuje do podanego selektora, a wywołasz metodę zakładającą pojedynczy element DOM, taką jak `ClickAsync`?

Odpowiedź: Wszystkie operacje na lokalizatorach zakładające pojedynczy element DOM zgłoszą wyjątek, jeżeli do zapytania będzie pasował więcej niż jeden element. Na przykład jeżeli użyjesz lokalizatora pasującego

do wszystkich przycisków na stronie i wywołasz metodę `ClickAsync`, to zostanie rzucony wyjątek.

5. Do czego służy program Playwright Inspector?

Odpowiedź: Program Playwright Inspector umożliwia tworzenie kompleksowych skryptów testowych w ułamku czasu potrzebnego do przygotowania ich ręcznie. Precyzyjnie rejestruje działania użytkownika, zmniejszając ryzyko błędów, które mogą wystąpić podczas ręcznego pisania testów.

Rozdział 15. Konteneryzacja przy użyciu Dockera

Oto sugerowane odpowiedzi na pytania:

1. Co sprawia, że kontenery są lepsze w porównaniu do maszyn wirtualnych?

Odpowiedź: Kontenery działają na jądrze systemu operacyjnego (OS) jednej maszyny i współdzielą je z innymi kontenerami. Są lepsze, ponieważ nie wymagają dodatkowego obciążenia w postaci hipernadzorcy zarządzającego maszynami wirtualnymi. Kontenery działają bezpośrednio w jądrze systemu hosta. Dzięki temu są bardziej wydajne, szybsze i mniej zasobożerne niż tradycyjne maszyny wirtualne, które wymagają pełnego systemu operacyjnego dla każdej z nich.

2. Jaki związek łączy Dockera i Kubernetesa?

Odpowiedź: Docker i Kubernetes to ściśle powiązane technologie, które odgrywają kluczowe role w obszarze konteneryzacji i orkiestracji we współczesnym tworzeniu i wdrażaniu oprogramowania. Docker to platforma umożliwiająca twórcom oprogramowania pakowanie aplikacji w kontenery — ustandaryzowane, wykonywalne komponenty łączące kod źródłowy aplikacji z bibliotekami systemowymi i zależnościami wymaganymi do uruchomienia tego kodu w dowolnym środowisku. Docker upraszcza tworzenie, wdrażanie i uruchamianie aplikacji z użyciem kontenerów.

Kubernetes to platforma o otwartych źródłach, która została zaprojektowana do automatyzowania wdrażania kontenerów aplikacji, ich skalowania i zarządzania nimi. Powstała, by radzić sobie ze złożonością wynikającą z używania na dużą skalę konteneryzowanych aplikacji, szczególnie w zakresie zarządzania wieloma kontenerami rozproszonymi na różnych maszynach.

Docker i Kubernetes uzupełniają się wzajemnie. Docker zapewnia prosty i wydajny sposób konteneryzacji aplikacji, natomiast Kubernetes jest metodą uruchamiania tych kontenerów w środowisku produkcyjnym na dużą skalę. Kubernetes często korzysta z Dockera jako środowiska uruchomieniowego kontenerów. Zazwyczaj twórca oprogramowania pisze aplikację i pakuje ją w kontener Dockera. Po przetestowaniu kontener może zostać przesłany do rejestru Dockera (takiego jak Docker Hub). Kubernetes może następnie pobrać ten kontener z rejestru i zarządzać jego wdrożeniem w klastrze, obsługując jego replikację, równoważenie obciążenia i przełączenia awaryjne.

Podsumowując: Docker upraszcza początkowe etapy pakowania i testowania aplikacji. Kubernetes obsługuje złożone zadania związane z zarządzaniem konteneryzowanymi aplikacjami na dużą skalę, takie jak sieci, orkiestracja pamięci masowej i zarządzanie cyklem życia kontenerów.

3. Jakie relacje istnieją między rejestrem Dockera, obrazem Dockera a kontenerem Dockera?

Odpowiedź: Obraz Dockera to statyczny rzut konfiguracji kontenera Dockera. Obraz ten zawiera wszystko, co jest potrzebne do uruchomienia aplikacji — kod, środowisko uruchomieniowe, biblioteki, zmienne środowiskowe i pliki konfiguracyjne. Obrazy służą do tworzenia kontenerów Dockera.

Kontener Dockera to instancja uruchomieniowa obrazu Dockera. Gdy uruchamiasz obraz, Docker tworzy z niego kontener. Kontenery to odizolowane środowiska, które posiadają własne systemy plików dostarczane przez obraz. Kontener uruchamia aplikację zgodnie z definicją w obrazie i dlatego może być łatwo uruchamiany, zatrzymywany, przenoszony oraz usuwany.

Rejestr Dockera to system przechowywania i dostarczania treści, który przechowuje nazwane obrazy Dockera dostępne w różnych wersjach. Użytkownicy współpracują z rejestrem za pomocą poleceń `push` i `pull` używanych do przesyłania i pobierania obrazów. Docker Hub to popularny publiczny rejestr, z którego może korzystać każdy. Dostępne są także rejestry prywatne, takie jak Amazon Elastic Container Registry (ECR) lub Azure Container Registry (ACR).

4. Na czym polega różnica między poleceniami `docker start` a `docker run`?

Odpowiedź: Polecenie `docker start` służy do uruchamiania wcześniej utworzonego i zatrzymanego kontenera. Nie tworzy nowego kontenera, ale po prostu uruchamia jeden lub więcej zatrzymanych kontenerów. Polecenie to jest przydatne, gdy masz kontenery, które zostały zatrzymane i trzeba je ponownie uruchomić, bez zmiany ich konfiguracji lub utraty danych znajdujących się w kontenerze. Utrzymuje ono konfigurację kontenera oraz zmiany jego stanu wewnętrznego od momentu utworzenia. Podczas uruchamiania zatrzymanego kontenera nie są tworzone nowe warstwy obrazu Dockera.

Polecenie `docker run` służy do utworzenia nowego kontenera z określonego obrazu Dockera i jego uruchomienia. Jeżeli wykorzystasz to polecenie, a obraz nie znajduje się lokalnie w systemie, to Docker najpierw pobierze obraz z rejestru Dockera, a następnie utworzy i uruchomi kontener.

Polecenie `docker run` łączy kilka działań: tworzy kontener, uruchamia go i może przydzielić kontenerowi pseudoterminal, a w zależności od użytych opcji może połączyć się ze standardowym wejściem i wyjściem oraz ze standardowym wyjściem błędów. W ramach tego polecenia można zdefiniować różne opcje, takie jak mapowanie portów, podłączanie woluminów, zmienne środowiskowe i ustawienia sieci.

Podsumowując: polecenie `docker run` służy do tworzenia i uruchamiania nowego kontenera, podczas gdy polecenie `docker start` po prostu uruchamia wcześniej

zatrzymany kontener. Użyj polecenia `docker run`, gdy potrzebujesz zainicjować nową instancję kontenera. Z kolei polecenia `docker start` używaj, gdy chcesz ponownie uruchomić kontener, który został wcześniej zatrzymany, zachowując jego stan i konfigurację. Wprowadzając polecenie `docker run`, możesz określić początkowe opcje konfiguracyjne. Pamiętaj, że polecenie `docker start` nie pozwala na zmianę konfiguracji — uruchamia kontener w takiej postaci, w jakiej został zatrzymany.

5. Jak wybiera się obraz bazowy w pliku *Dockerfile*?

Odpowiedź: W pliku *Dockerfile* instrukcja `FROM` rozpoczyna się od nazwy obrazu bazowego zawierającego środowisko uruchomieniowe .NET, tak jak w poniższym kodzie:

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0
```

Rozdział 16. Oprogramowanie chmurowe z .NET Aspire

Oto sugerowane odpowiedzi na pytania:

1. Wymień trzy główne rodzaje zasobów Aspire.

Odpowiedź: Wbudowane typy zasobów Aspire to:

- **Projekt** — projekt .NET. Na przykład serwis webowy lub witryna ASP.NET Core.
- **Kontener** — obraz kontenera. Na przykład obraz Dockera Redis lub RabbitMQ.
- **Plik wykonywalny** — plik wykonywalny.

2. Jaką funkcję pełni projekt *AppHost* w rozwiązaniu Aspire?

Odpowiedź: Projekt *AppHost* to aplikacja konsoli, która uruchamia wszystkie inne projekty i zapewnia poprawną konfigurację wszystkich zasobów i punktów końcowych. Projekt *AppHost* definiuje zasoby tworzące aplikację, w tym projekty .NET, kontenery, pliki wykonywalne i zasoby chmurowe. Projekt *AppHost* opisuje sposób, w jaki różne zasoby komunikują się ze sobą.

3. Jaką funkcję pełni projekt *ServiceDefaults* w rozwiązaniu Aspire?

Odpowiedź: Projekt *ServiceDefaults* to biblioteka klas, która centralizuje konfigurację wszystkich zasobów Aspire, w tym komponentów takich jak bazy danych i projekty .NET.

4. Co robi metoda `AddProject`?

Odpowiedź: Metoda `AddProject` dodaje zasób projektu .NET do modelu aplikacji Aspire. Dynamicznie skanuje ona wskazany projekt w poszukiwaniu informacji o konfiguracji zapisanych w pliku *launchSettings.json*, takich jak adresy URL. Na przykład można wywołać metodę `AddProject`, aby dodać projekt serwisu ASP.NET Core Web API, tak jak w poniższym kodzie:

```
var apiService = builder.AddProject  
    <Projects.AspireStarter_ApiService>("apiservice");
```

5. Co robi metoda `WithReference`?

Odpowiedź: Metoda `WithReference` łączy ze sobą zasoby Aspire, podając im poprawną konfigurację komunikacji, taką jak informacje niezbędne do wykrywania usług w projektach, do których tworzone jest odwołanie. Na przykład aby dodać projekt frontendowy, który będzie wywoływał serwis webowy i kontener Redisa, można użyć poniższego kodu:

```
builder.AddProject<Projects.AspireStarter_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(cache)
    .WithReference(apiService);
```

6. Co można zobaczyć na pulpicie deweloperskim Aspire?

Odpowiedź: Pulpit deweloperski Aspire składa się z pięciu sekcji: *Zasoby* (kontenery, projekty, pliki wykonywalne), *Konsola* (protokołowanie), *Strukturalne* (protokołowanie), *Ślady* oraz *Metryki*.

7. Jakie korzyści daje odwoływanie się do pakietu komponentów Aspire zamiast standardowego pakietu, takiego jak Redis?

Odpowiedź: Komponenty Aspire to biblioteki klas typu wrapper, które konfigurują poszczególne funkcje systemu, takie jak serwis Redisa, do poprawnego działania w środowisku chmurowym. Komponenty Aspire zostały zaprojektowane tak, żeby usunąć największą przeszkodę w rozpoczęciu pracy z tworzeniem aplikacji chmurowych. Tą przeszkodą jest zbyt duża liczba konfiguracji, które należy przygotować, przy czym często nie do końca wiadomo, od czego zacząć. Aspire pomaga w tym procesie, jednoznacznie określając zakres informacji, które dany komponent musi zapewniać, i wymagając, aby wszystkie komponenty zapewniały co najmniej domyślne ustawienia odporności, konfigurację telemetrii oraz integrację z kontenerem wstrzykiwania zależności (DI).

8. Jak można porównać ze sobą Aspire, Dapr i Orleans?

Odpowiedź: Systemów Dapr, Orleans i Aspire można używać razem. Każdy z nich zapewnia powiązaną, ale uzupełniającą funkcjonalność. Dapr (Distributed Application Runtime) to projekt open source zaprojektowany w celu ułatwienia tworzenia stabilnych aplikacji, zorientowanych na mikrousługi, które są niezależne od platformy. Orleans to framework do tworzenia rozproszonych aplikacji o dużej skali. Orleans wykorzystuje model aktora, ale wprowadza koncepcję wirtualnych aktorów, nazywanych we frameworku ziarnami (ang. *grains*). Te ziarna są podstawowymi jednostkami obliczeniowymi i przechowywania stanu.

9. Jakie technologie kontenerowe obsługuje Aspire?

Odpowiedź: Aspire obsługuje Dockera i Podmana. Domyślnie używany jest Docker.

10. Jak nakazać stosowanie stabilnego hasła do baz danych takich jak PostgreSQL?

Odpowiedź: Niektóre rozwiązania Aspire wymagają ustawienia stabilnego hasła zamiast polegania na domyślnym, generowanym za każdym razem przy

uruchamianiu rozwiązania. Najprostszym sposobem wyeliminowania tego problemu jest umieszczenie hasła w sekrecie użytkownika dla projektu *AppHost* przy użyciu klucza `Parameters:mypassword`.

Rozdział 17. Wzorce i zasady projektowe

Oto sugerowane odpowiedzi na pytania:

1. Co oznacza akronim SOLID?

Odpowiedź: SOLID oznacza następujące zasady:

- **Zasada pojedynczej odpowiedzialności** (ang. *Single Responsibility Principle* — SRP).
- **Zasada otwarte — zamknięte** (ang. *Open/Closed Principle* — OCP).
- **Zasada podstawienia Liskov** (ang. *Liskov Substitution Principle* — LSP).
- **Zasada segregacji interfejsów** (ang. *Interface Segregation Principle* — ISP).
- **Zasada odwrócenia zależności** (ang. *Dependency Inversion Principle* — DIP).

2. Wymień trzy główne rodzaje wzorców projektowych.

Odpowiedź: Trzy główne rodzaje wzorców projektowych to: kreacyjne, strukturalne i behawioralne.

3. Na czym polega wzorzec projektowy Strategia? Podaj przykład jego użycia w bibliotece klas .NET.

Odpowiedź: Wzorzec projektowy Strategia definiuje rodzinę algorytmów, hermetyzuje każdy z nich i umożliwia ich wymienne stosowanie. Strategia pozwala, by algorytm zmieniał się niezależnie od klientów, które z niego korzystają. Przykładem w bibliotece klas bazowych .NET są interfejsy `IComparer` i `IComparable`.

4. Wzorzec projektowy Singleton jest często implementowany w .NET przy użyciu klasy statycznej z właściwością `Instance` i prywatnym konstruktorem. Czy istnieje lepszy sposób implementowania tego wzorca?

Odpowiedź: Kontener DI wbudowany w ASP.NET Core pozwala na konfigurowanie usług z różnymi cyklami życia, w tym z czasem `Singleton`. Gdy usługa jest zarejestrowana jako `Singleton` w aplikacji ASP.NET Core, kontener DI zapewnia, że zostanie utworzona tylko jedna instancja tej usługi i będzie ona współdzielona w całej aplikacji. Jest to zgodne z celem tradycyjnego wzorca `Singleton`, czyli ograniczeniem klasy do jednej instancji.

5. Na czym polega zasada projektowa znana jako prawo Demeter?

Odpowiedź: Prawo Demeter (ang. *Law of Demeter* — LoD), znane również jako zasada najmniejszej wiedzy, to wytyczna dotycząca projektowania systemów obiektowych. Zakłada, że moduł powinien mieć wiedzę tylko o ściśle powiązanych modułach i komunikować się bezpośrednio jedynie z nimi. W praktyce oznacza to, że obiekt powinien unikać wywoływania metod na obiekcie zwracanym

(będącym wynikiem innego wywołania), gdyż prowadzi to do powstania silnie powiązanego kodu. Prawo Demeter można podsumować jako „rozmawiaj tylko ze swoimi bezpośrednimi przyjaciółmi”.

Rozdział 18. Podstawy architektury rozwiązań i oprogramowania

Oto sugerowane odpowiedzi na pytania:

1. Na czym polega różnica między „architekturą oprogramowania” a „architekturą rozwiązań”?

Odpowiedź: Architektura oprogramowania dotyczy przede wszystkim struktury i projektowania systemów programistycznych. Obejmuje definiowanie rozwiązania spełniającego wszystkie wymagania techniczne i operacyjne, przy jednoczesnej optymalizacji typowych atrybutów jakościowych, takich jak wydajność, bezpieczeństwo i łatwość zarządzania. Architektura rozwiązania ma charakter bardziej holistyczny i ukierunkowany na wymagania biznesowe. Obejmuje nie tylko samo oprogramowanie, ale również sprzęt, zasoby ludzkie i procesy potrzebne do rozwiązania problemu biznesowego lub spełnienia konkretnego wymagania.

2. Opisz krótko trzy style architektury oprogramowania.

Odpowiedź: Trzy style architektury oprogramowania to:

- **Modułowy monolit.** Jest ewolucją klasycznej architektury monolitycznej, zaprojektowaną w celu złagodzenia części jej wad przy zachowaniu korzyści. Stara się rozwiązać te problemy poprzez organizację aplikacji monolitycznej w moduły lub komponenty. Każdy moduł koncentruje się na konkretnym obszarze biznesowym lub funkcjonalności i jest oddzielony od innych modułów. Mimo że aplikacja jest nadal wdrażana jako jedna całość, poszczególne moduły mogą być projektowane tak, by skalowały się bardziej niezależnie w ramach aplikacji, np. z wykorzystaniem wielowątkowości. Proces wdrażania pozostaje tak samo prosty jak w klasycznym monolicie, ale dzięki modułowości możliwe jest skrócenie czasu kompilacji i bardziej ukierunkowana optymalizacja.
- **Mikroserwisy.** Wyobraź sobie skomplikowaną maszynę złożoną z niezależnych komponentów, z których każdy odpowiada za określoną funkcję i może działać samodzielnie. Taka właśnie jest architektura mikroserwisów. Tworzy aplikację jako zbiór serwisów, które są łatwe do utrzymania i testowania, są ze sobą luźno powiązane, wdrażane niezależnie od siebie i zorganizowane wokół konkretnych funkcji biznesowych. Ta architektura świetnie sprawdza się w przypadku dużych, złożonych aplikacji, które wymagają wysokiej skalowalności i elastyczności. Netflix i Amazon słyną z wykorzystania mikroserwisów do skalowania swoich globalnych usług.

- **Czysta architektura.** Czysta architektura ma na celu tworzenie systemów, które są łatwe do utrzymania, przetestowania i modyfikowania, ponieważ skupia się ona na rozdzieleniu odpowiedzialności. Wymaga staranności w zachowaniu oddzielenia warstw oprogramowania, szczególnie logiki biznesowej, od zewnętrznych zależności. Choć może się wydawać skomplikowana lub przesadna w przypadku małych projektów, zalety czystej architektury stają się bardziej oczywiste w miarę wzrostu rozmiaru i złożoności projektów. Jest wysoko ceniona w społeczności deweloperskiej.

3. Czym jest Domain-Driven Design (DDD)?

Odpowiedź: Domain-Driven Design (DDD) to podejście do projektowania oprogramowania, które koncentruje się na poznawaniu i modelowaniu domeny problemowej, w której działa system oprogramowania. DDD nie skupia się na samej technologii, ale umieszcza domenę w centrum procesu projektowego. Oto jego najważniejsze elementy:

- DDD zachęca programistów do konstruowania modelu domeny, czyli systemu abstrakcji opisujących jej wybrane aspekty. Taki model pomaga rozwiązywać problemy związane z daną domeną. Model domeny zawiera podstawowe pojęcia, zasady i zależności funkcjonujące w obrębie tej domeny. Służy jako pomost łączący rzeczywistość biznesową z kodem.
- Jednym z filarów DDD jest koncepcja języka wszechobecnego. Język wszechobecny to wspólny słownik używany przez ekspertów domenowych, użytkowników i programistów. Zapewnia on, że wszyscy zaangażowani w projekt posługują się spójnym nazewnictwem. Język ten stosuje się zarówno w modelu domeny, jak i przy opisywaniu wymagań systemowych.
- DDD kładzie nacisk na ścisłą współpracę między ekspertami technicznymi (programistami) a ekspertami domenowymi (osobami rozumiejącymi domenę biznesową). Dzięki wspólnej pracy iteracyjnie dopracowują oni model koncepcyjny rozwiązujący konkretne problemy domenowe. Taka współpraca pomaga zapewnić, że oprogramowanie dokładnie odzwierciedla rzeczywistość domenową.
- W obiektowo zorientowanej architekturze warstwowej warstwa domeny jest jedną z powszechnie stosowanych warstw. Warstwa ta zawiera model domeny i hermetyzuje podstawową logikę biznesową.
- DDD definiuje różne rodzaje modeli. Na przykład encja to obiekt zdefiniowany przez zbiór swoich cech. Encjami są m.in. miejsca w samolocie, ponieważ każde miejsce ma unikalną tożsamość (np. numer siedzenia). Obiekt wartości to niezmienny obiekt zawierający atrybuty, ale pozbawiony koncepcyjnej tożsamości. Na przykład podczas wymiany wizytówek ludzi interesuje zawartość karty (jej atrybuty), a nie rozróżnienie poszczególnych fizycznych kart.

4. Co to jest CQRS?

Odpowiedź: Rozdzielenie odpowiedzialności poleceń i zapytań (ang. *Command Query Responsibility Segregation* — CQRS) to wzorzec projektowy i styl architektoniczny, który wydziela osobne modele do odczytu i do aktualizacji

danych. Podejście to wywodzi się z fundamentalnej zasady informatyki znanej jako rozdzielenie poleceń i zapytań (ang. *Command-Query Separation* — CQS), która głosi, że każda metoda powinna być albo poleceniem wykonującym jakąś akcję, albo zapytaniem zwracającym dane do wywołującego, ale nie powinna wypełniać obu tych funkcji. CQRS idzie o krok dalej, wprowadzając taki sam podział na poziomie architektury.

5. Kim jest Wujek Bob i dlaczego jest tak ważną postacią dla programistów .NET?

Odpowiedź: Koncepcja czystej architektury została przedstawiona przez Roberta C. Martina (znanego jako Wujek Bob), który opracował uporządkowany sposób myślenia o architekturze oprogramowania, promujący stosowanie praktyk, w wyniku których powstaje łatwiejszy w utrzymaniu kod. Czysta architektura to filozofia projektowania oprogramowania, która kładzie nacisk na rozdzielenie odpowiedzialności pomiędzy różne elementy aplikacji. Podejście to ma na celu tworzenie systemów niezależnych od frameworków, interfejsu użytkownika, bazy danych czy jakichkolwiek zewnętrznych zależności.

Rozdział 19. Kariera, praca zespołowa i rozmowy kwalifikacyjne

Oto sugerowane odpowiedzi na pytania:

1. Na czym polegają obowiązki analityka biznesowego i w jaki sposób inżynier oprogramowania .NET może z nim współpracować?

Odpowiedź: Do obowiązków analityka biznesowego należy zbieranie informacji od interesariuszy i definiowanie wymagań biznesowych, tłumaczenie potrzeb biznesowych na specyfikacje techniczne oraz weryfikowanie zaimplementowanych funkcji pod kątem zgodności z wymaganiami biznesowymi. Inżynier oprogramowania .NET często musi zrozumieć logikę biznesową opisaną przez analityka, aby się upewnić, że rozwiązania techniczne są dokładnym odzwierciedleniem potrzeb biznesowych. Może także wspierać analityka w ocenie wykonalności proponowanych rozwiązań.

2. Czym jest programowanie w parach?

Odpowiedź: Programowanie w parach to technika wytwarzania oprogramowania, w której dwóch programistów współpracuje przy jednym stanowisku pracy. Jeden z nich (tzw. kierowca) pisze kod, podczas gdy drugi („obserwator” lub „nawigator”) na bieżąco przegląda każdy wiersz wpisywanego kodu. Role te są regularnie zamieniane, co sprzyja współpracy i bardziej kreatywnemu rozwiązywaniu problemów.

3. Czy model językowy, taki jak ChatGPT lub Llama3, mógłby zastąpić inżyniera oprogramowania .NET?

Odpowiedź: Llama3 i ChatGPT jako modele językowe LLM są potężnymi narzędziami do rozumienia i generowania języka naturalnego. Mają jednak inne role i możliwości niż inżynier oprogramowania .NET. LLM-y są

zaprojektowane do zadań związanych z przetwarzaniem języka naturalnego, takich jak tworzenie odpowiedzi przypominających ludzkie na podstawie podanych poleceń, udzielanie odpowiedzi na pytania faktograficzne, tłumaczenie tekstów między językami, skracanie długich tekstów do postaci streszczenia czy określanie tonu emocjonalnego tekstu.

Inżynier oprogramowania .NET to specjalista zajmujący się tworzeniem oprogramowania z wykorzystaniem platformy .NET (w tym C#, ASP.NET i innych powiązanych technologii). Inżynierowie .NET ściśle współpracują z ekspertami dziedzinowymi, kierownikami projektów i innymi członkami zespołu, aby tworzyć solidne i łatwe w utrzymaniu rozwiązania programistyczne.

Modele LLM mogą wspomagać inżynierów oprogramowania poprzez generowanie dokumentacji, pisanie komentarzy do kodu i dostarczanie wskazówek na temat wymagań zapisanych w języku naturalnym. Inżynierowie oprogramowania .NET wnoszą jednak wiedzę techniczną, znajomość domeny i umiejętności rozwiązywania problemów, niezbędne przy tworzeniu złożonych, funkcjonalnych systemów.

Chociaż modele LLM mogą automatyzować niektóre zadania, nie są w stanie całkowicie zastąpić inżynierów oprogramowania .NET. Inżynierowie rozumieją specyficzną domenę biznesową i jej niuanse, co jest podstawą budowania skutecznych rozwiązań. Modele LLM nie potrafią pisać skomplikowanych algorytmów, optymalizować kodu pod względem wydajności ani obsługiwać szczegółów niskopoziomowych systemów. Inżynierowie współpracują z bazami danych, punktami końcowymi API, bibliotekami zewnętrznymi i innymi komponentami w celu stworzenia spójnych systemów.

Tworzenie oprogramowania często wiąże się ze współpracą, komunikacją i podejmowaniem decyzji, a to właśnie w tych obszarach LLM-y zawodzą.

Podsumowując: mimo że modele LLM takie jak Llama3 i ChatGPT są wartościowymi narzędziami, nie są w stanie w pełni zastąpić wiedzy i umiejętności inżyniera oprogramowania .NET. Mogą natomiast wspierać proces tworzenia oprogramowania, oferując pomoc językową i automatyzując powtarzalne zadania.

4. Jakich pytań możesz się spodziewać podczas rozmowy kwalifikacyjnej?

Odpowiedź: Pytania zadawane podczas rozmowy kwalifikacyjnej zazwyczaj mają na celu ocenę zakresu wiedzy i umiejętności technicznych kandydata, a także jego dopasowania do kultury firmy. Można spodziewać się pytań dotyczących:

- **Dopasowania do danego stanowiska.** Pytania są dobierane na podstawie konkretnych umiejętności i wiedzy wymaganych na danym stanowisku.
- **Umiejętności rozwiązywania problemów.** Pracodawcy często wybierają pytania oceniające zdolność kandydata do rozwiązywania problemów, w szczególności do rozwiązywania problemów pod presją.
- **Potencjału do rozwoju.** Pytania mogą również dotyczyć zdolności kandydata do nauki i rozwoju.

- **Dopasowania kulturowego.** Firmy zadają także pytania, które pomagają określić, czy kandydat pasuje do firmowych wartości, etyki pracy i dynamiki zespołu w organizacji.
5. Dlaczego podczas nauki warto zobaczyć zarówno błędne, jak i poprawne odpowiedzi?

Odpowiedź: Przeglądanie zarówno błędnych, jak i poprawnych odpowiedzi podczas nauki danego tematu daje wiele korzyści:

- **Kontrast.** Zestawiając błędne odpowiedzi z poprawnymi, możesz je ze sobą porównać. Pomaga to zrozumieć subtelności i granice danego tematu.
- **Typowe błędy.** Analizując błędne odpowiedzi, poznajesz typowe nieporozumienia lub błędy popełniane przez innych. Świadomość istnienia tych pułapek może uchronić Cię przed ich powielaniem.
- **Myślenie krytyczne.** Ocena błędnych odpowiedzi pobudza myślenie krytyczne. Zaczynasz kwestionować założenia i logikę stojącą za każdą odpowiedzią.
- **Przyczyny błędów.** Niepoprawne odpowiedzi często ujawniają luki w znajomości danego tematu. Gdy już dowiesz się, dlaczego dana odpowiedź jest błędna, możesz wyznaczyć sobie tematy wymagające dodatkowego zgłębienia.
- **Powtórka rozłożona w czasie.** Późniejsze wracanie zarówno do poprawnych, jak i błędnych odpowiedzi podnosi skuteczność zapamiętywania. Trudności pojawiające się przy poprawianiu błędów jeszcze bardziej wzmacniają proces uczenia się.
- **Aktywne przywoływanie.** Przeglądanie błędnych odpowiedzi zmusza mózg do aktywnej pracy. Proces ten wzmacnia połączenia pamięciowe.
- **Rzeczywiste przypadki.** W praktyce raczej nie spotykasz się z idealnie sformułowanymi pytaniami i jednoznacznymi odpowiedziami. Przeglądanie błędnych odpowiedzi przygotowuje na niejednoznaczności istniejące w rzeczywistych przypadkach.
- **Informacja zwrotna.** Błędne odpowiedzi dostarczają nam dodatkowych informacji. Wskazują obszary wymagające dalszego ćwiczenia.
- **Unikanie nadmiernej pewności siebie.** Obserwowanie błędnych odpowiedzi przypomina, że nikt nie jest nieomylny. Zmusza to do pokory i zapobiega przesadnej pewności siebie.
- **Otwartość na poprawki.** Gotowość do uczenia się na błędach jest przejawem dążenia do samodoskonalenia. Stajesz się bardziej otwarty na korygowanie własnego zasobu wiedzy.

Podsumowując: zetknięcie się z błędnymi i poprawnymi odpowiedziami wzbogaca proces uczenia się, wymusza krytyczny sposób myślenia i pomaga lepiej poznać analizowany temat.