106 PYTAŃ REKRUTACYJNYCH

dla Junior JavaScript Developera

devszczepaniak.pl

Przede wszystkim chciałbym Ci podziękować za zaufanie i za zostawienie mi swojego adresu e-mail. Mam nadzieję, że ten e-book przyniesie Ci realną wartość i pomoże rozpocząć pracę na stanowisku **Junior JavaScript Developera**.

Niniejszy e-book przeznaczony jest dla wszystkich programistów *wannabe* oraz tych, którzy dopiero zbierają swoje pierwsze zawodowe doświadczenie. W e-booku znajdziesz 106 przykładowych pytań, które możesz usłyszeć podczas rozmowy rekrutacyjnej. Zdecydowaną większość stanowią pytania techniczne, które mają za zadanie sprawdzić Twoją znajomość podstawowych pojęć, jakie Junior JavaScript Developerowi nie powinny być obce. Ponieważ JavaScript może być wykorzystany zarówno jako język frontendowy, jak i backendowy, w e-booku poruszone zostały zagadnienia z obu dziedzin. Nawet jeśli wiesz, że interesuje Cię tylko jeden z tych kierunków, to wiedza z tego drugiego obszaru i tak Ci się przyda. Będąc kandydatem na Junior Developera, często jeszcze nie wiadomo, w którym kierunku pójdzie Twoja kariera. Dlatego też gorąco zachęcam do przeczytania e-booka w całości.

Oprócz pytań teoretycznych znajdziesz tu też kilka szybkich zadań praktycznych. Zadania takie są wyraźnie oznaczone, a ich ukończenie nie powinno Ci zająć nie więcej niż kilka minut. Wiedza praktyczna znacznie częściej sprawdzana jest poprzez zadanie rekrutacyjne, które otrzymasz przed lub po rozmowie rekrutacyjnej. W tym e-booku nie skupiam się na tej części procesu rekrutacyjnego. Poniżej znajdziesz kilka propozycji serwisów, gdzie możesz potrenować swoje umiejętności rozwiązywania prostych zadań praktycznych (kliknij w link, aby otworzyć):

- Sphere online judge
- Codewars
- HackerRank
- edabit
- beecrowd

Na końcu e-booka znajdziesz też kilka pytań nietechnicznych, na które szczególnie warto znać odpowiedź. Odpowiedzi na nie zależą całkowicie od Ciebie, twoich planów, doświadczenia i ścieżki zawodowej, więc nie znajdziesz tu gotowej odpowiedzi, a jedynie wskazówki. Bardzo zachęcam, aby mieć na nie odpowiedź i nie dać się zaskoczyć podczas rozmowy.

E-book został opublikowany w III kwartale 2023 roku. Rozwój ekosystemu JavaScript jest dynamiczny i pewne pytania lub sposoby rozwiązywania zadań mogą ulec dezaktualizacji na przestrzeni czasu. Niemniej jednak zdecydowana większość pytań jest ponadczasowa.

Chcąc tworzyć dla Ciebie jak najlepsze treści, dołożyłem wszelkich starań, aby ten e-book stanowił produkt kompletny. Jeśli jednak widzisz, że coś jest niejasne lub warte poprawienia to zachęcam do kontaktu pod adresem **kontakt@devszczepaniak.pl**. Na stronie dedykowanej e-bookowi znajdziesz ewentualne erraty i aktualizacje treści.

Jeśli znalazłeś/aś ten e-book w Internecie lub otrzymałeś/aś go od kogoś, to będzie mi bardzo miło, jeśli zapiszesz się na mój mailing. Pozwoli mi to udostępniać Ci informacje o nowych wpisach oraz więcej materiałów takich jak ten e-book.

O autorze

Dominik Szczepaniak — pierwsze doświadczenie zawodowe zbierał jako WordPress developer, a od 2018 roku pracuje w CKSource jako Backend Software Engineer. Na co dzień tworzy, rozwija i utrzymuje aplikacje backendowe w Node.js i TypeScript. Zespół, w którym pracuje, zajmuje się dostarczaniem rozwiązania SaaS oraz On-Premises umożliwiającego kolaborację użytkowników w czasie rzeczywistym z wykorzystaniem edytora tekstu CKEditor 5.

Najbardziej lubi pracować nad trudnymi i nieszablonowymi problemami, wymagającymi niecodziennego podejścia. Wyzwania traktuje jako okazję do zdobycia nowych doświadczeń.

Po godzinach rozwija bloga programistycznego devszczepaniak.pl, gdzie dzieli się swoją wiedzą i doświadczeniem.

HTML

Przy pytaniach dotyczących HTML zdecydowałem się, aby pominąć pytania o absolutne podstawy. Zakładam, że przeznaczenie HTML nie jest Ci obce oraz że znasz najczęściej wykorzystywane znaczniki HTML. Zaproponowane pytania dla HTML są wyrywkową formą sprawdzenia jego znajomości.

1. Jakie kodowanie znaków jest rekomendowane dla stron WWW w j. polskim? W jaki sposób można je ustawić?

Rekomendowanym kodowaniem znaków dla języka polskiego jest *UTF-8*. To zapewnia, że wszystkie znaki specjalne, nie tylko dla j. polskiego, będą wyświetlone poprawnie. Kodowanie strony ustawiamy za pomocą znacznika *<meta charset="utf-8">* w sekcji *head*.

2. Co oznacza semantyczny HTML?

Semantyczny HTML oznacza, że oprócz definiowania struktury strony WWW, nadaje on znaczenie jej poszczególnym elementom. W przeszłości domyślnym elementem w strukturze HTML strony były elementy *div*, a jeszcze wcześniej tabelki i ramki. Obecnie, powszechnie występujące elementy stron WWW mają odpowiednie znaczniki, takie jak *header* dla nagłówka, *footer* dla stopki, czy *nav* dla nawigacji. Oczywiście takich znaczników jest więcej. Wykorzystanie dedykowanych elementów, z punktu widzenia semantyki jest bardziej poprawne niż wykorzystanie elementu ogólnego przeznaczenia (*div*).

HTML poprzez swoje elementy opisuje również znaczenie treści w nich zawartych. Na przykład, istotne jest, aby wykorzystywać odpowiednio hierarchię nagłówków od *h1* do *h6*. Innymi przykładami są: element cytatu (*blockquote*), wzmocnienia tekstu (*strong*) czy emfazy (*em*). Te elementy, oprócz wizualnej edycji wyświetlanej zawartości, niosą dodatkowe informacje o tekście w nich zawartym.

Istota semantycznego HTML ujawnia się przy wykorzystaniu oprogramowania interpretującego kod stron WWW. Przykładem takiego oprogramowania są czytniki ekranowe, a także boty odpowiedzialne np. za indeksowanie stron WWW. Semantycznie poprawny HTML pozytywnie wpływa na dostępność (*accessibility*) oraz SEO. Oprócz tego, wykorzystanie semantycznego HTML ułatwia pracę programistom utrzymującym stronę.

3. Czy znasz pojęcie dostępności stron WWW? Dlaczego jest istotne?

Dostępność stron internetowych odnosi się do tworzenia i projektowania stron WWW w taki sposób, aby były one użyteczne dla wszystkich użytkowników. W szczególności mowa tu o użytkownikach z niepełnosprawnościami wzrokowymi, słuchowymi, czy ruchu.

Dostępność stron WWW jest istotna z kilku powodów. Przede wszystkim, strona WWW powinna zapewniać równy dostęp do informacji wszystkim użytkownikom. Po drugie, dostępność stron WWW dla niektórych podmiotów wymagana jest przez prawo. Głównie dotyczy to podmiotów publicznych. Pełny wykaz podmiotów definiuje Ustawia z dnia 4 kwietnia 2019 r. o dostępności cyfrowej stron internetowych i aplikacji mobilnych podmiotów publicznych (Dz.U. 2019 poz. 848). Dostępna strona WWW pozwala także na dotarcie do szerszego grona odbiorców i zwiększa ogólny komfort użytkownika.

Przykładami działań zwiększających dostępność stron WWW są: wykorzystywanie kolorów z odpowiednim kontrastem, wykorzystanie odpowiedniego rozmiaru tekstu, stosowanie semantycznego HTML, nawigacja po stronie WWW z wykorzystaniem klawiatury, stosowanie opisów alternatywnych dla obrazów. Więcej przykładów zaleceń co do dostępności stron WWW można znaleźć w dokumencie Web Content Accessibility Guidelines (WCAG) 2.0.

4. Czym są atrybuty *data* i jak można je wykorzystać?

Atrybuty *data* można dodawać do znaczników HTML, aby przypisywać do nich dodatkowe informacje. Atrybuty *data* dodaje się w postaci *data-[nazwa_atrybutu]*. Przykładem wykorzystania może być przypadek, gdy chcemy wyświetlić na stronie WWW rekordy z bazy danych. Każdy rekord można wyświetlić w osobnym znaczniku *div*, a do znacznika dodać atrybut *data-id* wraz z identyfikatorem tego rekordu. Atrybut data-id można następnie wykorzystać w kodzie CSS lub JavaScript.

```
<div class="record" data-id="54759eb3c090d83494e2d804">foo</div>

<script>
const record = document.querySelector('.record');

console.log(record.dataset.id); // "54759eb3c090d83494e2d804"
</script>

<style>
.record[data-id='54759eb3c090d83494e2d804'] {
    color: red;
}
</style>
```

5. Jak spowodować, aby kliknięcie etykiety checkboxa powodowało jego zaznaczenie?

Teoretycznie można do tego wykorzystać JavaScript. Jednak znacznie prościej można to zrobić poprzez nadanie atrybutu *id* na znacznik *input* z typem *checkbox* oraz nadanie atrybutu *for* dla znacznika *label*. Wartość atrybutu *for* powinna być identyczna z *id* docelowego checkboxa. Analogicznie działa to dla inputów typu *radio*.

```
<label for="agreement">I agree</label>
<input type="checkbox" id="agreement">
```

6. Dlaczego stylowanie strony WWW elementami HTML takimi jak *<b>*, *<i>* i stylami inline to zły pomysł?

Stylowanie elementami HTML powoduje, że HTML zyskuje nadmiarową odpowiedzialność. Jedyną odpowiedzialnością HTML-a powinna być reprezentacja treści i struktury strony WWW. Dostosowanie wizualne strony powinno być odpowiedzialnością kodu CSS. Dodatkowym argumentem za zrezygnowaniem ze stylowania HTML-em jest fakt, że różne przeglądarki domyślnie wykorzystują różne style, przez co nie ma pewności, jak ostatecznie dany element będzie wyglądał.

Wykorzystanie stylów inline jest z kolei nieoptymalne. Style inline utrudniają ich ponowne wykorzystanie dla innych elementów. Ponadto style inline mają wyższy priorytet niż style zewnętrzne, co utrudnia debuggowanie i zwiększa ryzyko wystąpienia błędów na stronie.

7. Czym są encje w HTML?

Jeśli potrzebujemy wstawić w tekst na stronie WWW jakiś znak specjalny lub znak stanowiący element składni HTML, to możemy skorzystać z encji. Encję tworzy się poprzez znak &, następnie nazwa encji zakończona średnikiem. Na przykład encja znaku copyright (©) to *©*, a encje dla znaków < oraz > są to odpowiednio *<* i *>*.

CSS

Podobnie jak w przypadku pytań dotyczących HTML, szansa, że usłyszysz pytania o konkretne reguły CSS, jest bardzo mała. Nie oznacza to, że nie warto ich przećwiczyć, wręcz przeciwnie. Nie zdarzyło mi się jednak nigdy słyszeć pytań o nazwy czy właściwości konkretnych selektorów. Pytania w tym rozdziale poruszają praktyczne zagadnienia i problemy, z jakimi Web Developerzy spotykają się na co dzień. Celem tych pytań jest sprawdzenie przez rekrutera czy znasz charakterystykę CSS. Ostatnie z zadań jest zadaniem praktycznym, a przedstawione rozwiązanie jest tylko jednym z wielu poprawnych. Zachęcam do zrobienia tego zadania innym sposobem niż przedstawiony.

8. Czym jest RWD (Responsive Web Design)? Jak zrobić inny styl na mobile a inny na desktop?

Responsive Web Design jest podejściem do projektowania stron WWW, którego celem jest dostosowanie wyglądu i układu strony do każdego urządzenia. Obecnie, wykorzystanie urządzeń mobilnych jest na tyle powszechne, że projektowanie stron zgodnie z RWD jest praktycznie *must-have*.

Aby móc sterować wyglądem strony opierając się na parametrach takich jak orientacja czy szerokość ekranu można wykorzystać *@media queries*. Dzięki temu mechanizmowi można dobrać odpowiednie style w zależności od np. szerokości ekranu czy orientacji urządzenia.

```
@media screen and (min-width: 30em) and (orientation: portrait) {
  p {
    color: red;
  }
}
```

9. Co oznacza, że CSS jest kaskadowy?

Zamiast tego pytania może alternatywnie paść pytanie o hierarchiczność lub specificity CSS. Kaskadowość CSS wynika z algorytmu, jaki jest wykorzystywany do aplikowania stylów. Algorytm aplikacji stylów bierze pod uwagę kilka elementów.

Pierwszym z nich jest miejsce umieszczenia stylu. Style mogą być aplikowane na trzy sposoby: bezpośrednio we właściwościach stylowanego elementu, z wykorzystaniem znacznika *style* w pliku HTML oraz z wykorzystaniem zewnętrznego pliku. W przypadku, gdy pomiędzy stylami zdefiniowanymi w tych trzech miejscach wystąpi konflikt, priorytet otrzymają najpierw style inline, następnie style z pliku HTML, a na końcu style z zewnętrznych plików. Istotna jest również kolejność definiowania stylów. Jeśli dany parametr, np. kolor tekstu dla danego elementu został zdefiniowany podwójnie, np. w jednym pliku, to ostatecznym kolorem tekstu będzie ten zdefiniowany później. Pliki CSS interpretowane są z góry do dołu. Z hierarchią stylów wiąże się również dziedziczenie stylów. Jeżeli element dziecka nie ma zdefiniowanego konkretnego stylu, to przejmie on style rodzica.

Innym, równie istotnym zagadnieniem jest *specificity*. Określa ono priorytet, jaki otrzymuje określony styl. Jeśli element HTML stylowany jest względem jego klasy i nazwy, to w przypadku konfliktu, style zdefiniowane dla klasy będą miały wyższy priorytet. *Specificity* jest niezależne od umiejscowienia stylów, tzn. reguła kolejności stylów z poprzedniego akapitu nie ma tu zastosowania.

Specifity można liczyć punktowo, które się kumulują. Najmniej punktów daje *universal selector* (*), bo aż całe 0. Stylowanie po elementach lub pseudoelementach daje 1 punkt. Klasy, pseudoklasy i atrybuty dają 10 punktów. Style dla konkretnego *id* dają 100 punktów, a style inline 1000. Najwięcej punktów *specificity* dają style z flagą *!important*, aż 10000.

10. Czym są pseudoklasy i pseudoelementy?

Pseudoklasy w CSS to selektory, które pozwalają aplikować style do określonych stanów lub kontekstów elementów HTML. Dzięki nim możliwe jest stylowanie na podstawie interakcji elementu z użytkownikiem, jego pozycji w dokumencie i innych właściwościach.

Przykładami pseudoklas są:

- *:hover* — stosowana, gdy kursor myszy jest nad elementem;
- *:not(selector)* — stosowana, aby wybrać elementy, które nie pasują do określonego selektora;
- *:first-child*, *:last-child* — stosowane, aby wybrać pierwsze lub ostatnie dziecko danego rodzica.

Pseudoelementy to konstrukcje pozwalające na dodanie i ostylowanie części elementu, na który wskazuje dany selektor. Dzięki pseudo elementom możemy ostylować np. pierwszą linię tekstu (*::first-line*), pierwszą literę tekstu (*::first-letter*), a także to, co się znajduje przed (*::before*) i za elementem (*::after*).

11. Czym jest Grid i Flexbox? Kiedy z nich korzystać?

Grid i Flexbox to narzędzia CSS służące pozycjonowaniu elementów na stronie. Grid pozwala na pozycjonowanie względem dwuwymiarowej siatki z wierszami i kolumnami. Można definiować zarówno szerokość i wysokość wierszy, jak i kolumn oraz kontrolować ich miejsce na stronie. Flexbox pozwala pozycjonować elementy w jednym wymiarze, tzn. poziomo lub pionowo. Flexbox nada się do tworzenia layoutów, w których elementy mogą dostosowywać się do dostępnego miejsca, zmieniać swoje rozmiary i zachowanie w zależności od potrzeb. Szczególnie przydatny będzie do stylowania komponentów takich jak menu nawigacyjne, listy elementów, galerie zdjęć.

CIEKAWOSTKA:
Stylowanie z wykorzystaniem opisanych narzędzi możesz potrenować przez gry online: Flexbox Froggy oraz Grid Garden.

12. Czym jest normalizacja CSS i po co to robić?

Większość przeglądarek ma domyślne style dla elementów HTML. Poleganie na nich może powodować, że niektóre elementy mogą wyglądać inaczej na różnych przeglądarkach. Aby zapobiec temu zjawisku, można dokonać normalizacji CSS. Normalizacja CSS polega na ujednoliceniu domyślnego wyglądu elementów dla wszystkich przeglądarek. Alternatywą dla normalizacji jest reset stylów, czyli całkowite usunięcie domyślnych stylów z elementów HTML. Aby te mechanizmy działały poprawnie, style normalizujące lub resetujące powinny być załadowane jako pierwsze.

13. [ZADANIE PRAKTYCZNE] Zaprojektuj prosty layout, który będzie na desktopie miał kwadraty o wymiarach 100px na 100px w układzie 2x3 a na urządzeniach mobilnych 3x2.

W celu uproszczenia wszystkie style będą osadzone w pliku HTML w znaczniku *style*. Umownie, jako wartość graniczną między desktopem a urządzeniami mobilnymi przyjąłem wartość 768px. Aby zwiększyć czytelność layoutu, warto ustawić kolor kwadratów, ja wybrałem pomarańczowy. Do wykonania zadania wykorzystałem Grid omówiony w jednym z poprzednich pytań. Przykładowe rozwiązanie znajdziesz poniżej, ale zachęcam też do samodzielnego rozwiązania zadania.

```
<style>
  .container {
    width: 300px;
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    grid-template-rows: repeat(2, 1fr);
    grid-column-gap: 1px;
    grid-row-gap: 1px;
  }

  .container div {
    background-color: orange;
    width: 100px;
    height: 100px;
  }

  @media screen and (max-width: 768px) {
    .container {
      width: 200px;
      grid-template-columns: repeat(2, 1fr);
      grid-template-rows: repeat(3, 1fr);
    }
  }
</style>

<div class="container">
  <div></div>
  <div></div>
  <div></div>
  <div></div>
  <div></div>
  <div></div>
</div>
```

14. Jakie widzisz korzyści z wykorzystania zmiennych w CSS?

Wykorzystanie zmiennych w CSS przynosi kilka korzyści. Po pierwsze zwiększają czytelność kodu. Przykładem jest wykorzystanie zmiennych CSS do opisu palety kolorów. Drugą z zalet jest możliwość łatwiejszej i szybszej zmiany powtarzalnych elementów. Przykładowo, jeśli paleta kolorów na stronie musi zostać zmieniona, to gdy wykorzystywane są zmienne, jej zmiana potrwa znacznie szybciej niż w przypadku zmiany kolorów w każdym z miejsc z osobna. Zmniejsza to też ryzyko popełnienia błędu przy podmianie. Wykorzystanie zmiennych pomaga zachować spójność poprzez zdefiniowanie domyślnych wartości dla wybranych atrybutów CSS np. wspomniane już kolory, ale też wielkości odstępów, marginesów, rozmiaru tekstu itd. Zmienne zwiększają też komfort programisty, który nie musi szukać na własną rękę domyślnie wykorzystywanych wartości w plikach CSS.

JavaScript

Myślę, że Cię nie zaskoczę, gdy Ci powiem, że ten rozdział będzie największy. Oprócz pytań weryfikujących znajomość wybranych aspektów języka JavaScript zamieściłem też kilka pytań o wybrane API przeglądarek, takich jak Fetch API czy HTML Web Storage API i kilka zadań praktycznych sprawdzających sposób myślenia kandydata.

15. Jakie znasz typy danych? Na czym polega referencja?

W JavaScript wyróżniamy dwa typy danych: proste (prymitywne) oraz referencyjne. Zadaniem zmiennych typu prostego jest jedynie przechowanie konkretnej wartości. Do typów prostych zaliczają się: *number*, *boolean*, *string*, *null*, *undefined* i *symbol*.

Typ *object* jest typem referencyjnym. Do typu *object* zalicza się również tablice. Zmienne typu referencyjnego cechują się tym, że nie przechowują one bezpośrednio wartości, a jedynie referencję. Referencja to wskazanie miejsca w pamięci, w jakim znajduje się dana wartość. Powoduje to, że klonowanie zmiennych z wartościami typu referencyjnego jest nieco trudniejsze niż klonowanie wartości z typami prostymi. Zwykłe przypisanie powoduje jedynie przypisanie referencji wskazującej na ten sam obiekt. Klonując obiekty i tablice, należy pamiętać, że zagnieżdżone obiekty i tablice również mają swoje referencje.

CIEKAWOSTKA:
Czy wiesz, co będzie rezultatem wywołania *typeof null*?

16. Co to są menedżery pakietów i do czego się ich używa?

Menedżery pakietów pozwalają na zarządzanie zależnościami dodawanymi do projektu. Dzięki menedżerom można dodawać, usuwać czy aktualizować wykorzystywane pakiety. Menedżery umożliwiają też zarządzanie paczkami: publikowanie nowych wersji, dodawanie tagów itp. Do najpopularniejszych menadżerów należą npm, yarn i pnpm.

17. Jakie rodzaje zależności można zdefiniować w *package.json* i kiedy z nich korzystać?

Najbardziej podstawowe zależności, jakie można dodać dla projektu JavaScript, to zależności produkcyjne. Są to zależności wykorzystywane w kodzie produkcyjnym, a ich instalacja jest konieczna do poprawnego działania projektu. Definiowane są pod kluczem *dependencies*. Kolejnym typem są zależności developerskie, które wykorzystywane są np. do budowania paczek z kodem, uruchamiania testów itp. Zależności developerskie definiowane są w *devDependencies* i nie lądują w ostatecznym buildzie produkcyjnym. Trzecim interesującym typem zależności są *peerDependencies*. Te zależności nie są bezpośrednio instalowane, ale są wymagane przez kod produkcyjny. Ma to zastosowanie w przypadkach, gdy zainstalowanie zależności jako produkcyjnej jest niepotrzebne lub niemożliwe. Na przykład, tworząc plugin do jakiegoś frameworka, nie ma potrzeby instalowania go jako produkcyjnej zależności. Chcąc wykorzystać jego komponenty, można zdefiniować ten framework jako peer dependency. Jest to logiczne, ponieważ instalacja takiego pluginu bez instalacji frameworka mija się z celem. Oprócz opisanych istnieją także takie typy zależności jak *optionalDependencies* i *bundledDependencies*, ale ich wykorzystanie jest rzadko spotykane.

18. Czy dane przechowywane w Local Storage są widoczne w innych aplikacjach (pod innymi domenami)?

Odpowiedź brzmi, nie są. Przed dostępem do danych zapisanych w Local Storage w przez aplikacje w domenie chroni mechanizm *Same Origin Policy*. Aby uzyskać dostęp do danych z Local Storage, konieczna jest zgodność protokołu, domeny, subdomeny i portu.

19. Czym są *falsy values*?

Falsy values są to wartości, które przekazane do warunku klauzuli *if* dadzą wynik *false*. Wyróżnia się następujące falsy values: *false*, *null*, *undefined*, *0*, *NaN*, pusty *string*.

20. Czym się różni == od ===?

Porównanie z wykorzystaniem === od ==, oprócz sprawdzenia wartości zmiennej porównuje również jej typ. Najprościej można to zobrazować, wykonując porównania *0 == '0'* i *0 === '0'*. W pierwszym przykładzie, mimo niezgodności typów, rezultatem porównania będzie wartość *true*.

21. Do czego służy *optional chaining*?

Takie pytanie ma na celu sprawdzić, czy użytkownik jest na bieżąco z nowymi funkcjami języka. Alternatywnie może tu paść pytanie o dowolny inny konstrukt JavaScript, który został niedawno wprowadzony do standardu. *Optional chaining* pozwala na uproszczenie zapisu dostępu do opcjonalnych zagnieżdżonych właściwości obiektu. Dzięki *optional chaining* zapis *obj.first && obj.first.second* można zastąpić uproszczonym zapisem *obj.first?.second*.

22. Czym się różni *??* od *||*?

Operator *||* jest klasycznym operatorem *OR* znanym z większości języków programowania. Operator *??* (*nullish coalescing*) jest operatorem przypisywania wartości, ale różni się od operatora *||* w sposobie traktowania fałszywych wartości. Operator *??* zwraca pierwszą wartość, która nie jest ani *null*, ani *undefined*. Pozostałe *falsy values* zostaną uznane przez operator *nullish coalescing* za *truthy*.

23. Dlaczego skrypty podłączamy zwykle przed zamknięciem tagu body?

Domyślnie podczas ładowania skryptu za pomocą tagu *script* parsowanie HTML jest wstrzymane do czasu zakończenia działania skryptu. Takie działanie powoduje kilka komplikacji.

Przede wszystkim opóźniamy ładowanie naszej strony. Ładując masywny skrypt, możemy spowodować, że odbiorca później zobaczy zawartość naszej witryny, co niekorzystnie wpływa na UX i SEO. Drugą komplikacją jest odwoływanie się do elementów drzewa DOM.

Dodając znaczniki *script* do elementu *head* powodujemy, że body się nie zdąży załadować, przez co próba jakichkolwiek operacji na elementach DOM będzie skutkowała błędami, ponieważ elementy DOM nie zostały jeszcze stworzone. Można sobie z tym poradzić i skorzystać ze zdarzenia *DOMContentLoaded* lub właśnie przenosząc tagi *script* przed zamknięcie tagu *body*.

24. Wyjaśnij różnicę pomiędzy *<script async>* i *<script defer>*.

Jest to pytanie ściśle powiązane z poprzednim. Aby uporać się z problemami opisanymi w poprzednim pytaniu, można skorzystać z importowania skryptów *async* lub *defer*. *Async* powoduje, że skrypty nie wstrzymują parsowania kodu HTML podczas ładowania. Wstrzymanie następuje jednak w momencie wykonywania skryptów. *Defer* również nie powoduje wstrzymania parsowania kodu HTML. Różny jest jednak moment wykonania skryptów. W tym przypadku kod wykona się dopiero po zakończeniu parsowania kodu HTML.

25. Jak reagować na zdarzenia w przeglądarce, np. kliknięcie w dany element HTML?

Aby móc reagować na zdarzenia w przeglądarce, oprócz wystąpienia zdarzenia potrzebna będzie referencja do elementu, którego zdarzenie chcemy obsłużyć. Referencję można pozyskać na kilka sposobów, np. z wykorzystaniem metody *querySelector()* wywołanej na obiekcie *document*. Mając referencję do obiektu, można na nim wywołać metodę *addEventListener()*, która umożliwia zdefiniowanie, jakie akcje mają się wykonać dla poszczególnych zdarzeń.

```
document.querySelector( '.container' ).addEventListener( 'click', () => {
    alert( 'clicked!' );
} );
```

26. Jaki będzie rezultat wywołania poniższego kodu? Dlaczego otrzymamy taki wynik?

```
function foo() {
  for( var i = 0; i < 10; i++ ) {
    setTimeout( function() {
      console.log( i );
    }, 300 );
  }
}

foo();
```

Jest to przykład kodu, który kilkukrotnie trafiłem w trakcie rozmów rekrutacyjnych. Odpowiedzią na to pytanie będzie wyświetlenie w konsoli 10 razy liczby 10. W pętli *for* zmienna *i* jest zadeklarowana za pomocą słowa kluczowego *var*. Zmienna *var* ma zasięg globalny w obrębie funkcji *foo()*. Po upływie 300 milisekund od rozpoczęcia pętli zostanie wywołane 10 callbacków. W każdym callbacku *console.log(i)* wypisze wartość zmiennej *i*, która w tym momencie wynosi 10. Dlatego otrzymamy 10 wypisań liczby 10. Intuicja podpowiada odpowiedź „liczby od 0 do 9”, na którą sam raz dałem się złapać. Byłoby to prawdą, gdyby słowo *var* zastąpić słowem kluczowym *let*. Wtedy po 300 milisekundach zostaną wyświetlone liczby od 0 do 9.

W miejsce przedstawionego fragmentu może się pojawić inny, ale jego celem wciąż będzie sprawdzenie, czy kandydat jest świadom istnienia wybranych mechanizmów języka JavaScript.

27. Wytłumacz na czym polega *hoisting*.

Hoisting występuje w przypadku zadeklarowania zmiennej lub funkcji słowem kluczowym *var*. *Hoisting* powoduje, że deklaracje zmiennych są wywoływane w pierwszej kolejności przed wykonaniem reszty kodu. Co istotne, *hoisting* jedynie deklaruje zmienne, a nie przypisuje im wartości. Oznacza to, że odwołanie się do zmiennej zadeklarowanej słowem kluczowym *var* będzie skutkować otrzymaniem wartości *undefined*.

Zmienne zadeklarowane słowami *let* oraz *const* również ulegają hoistingowi, lecz zamiast ich deklaracji, trafiają do tzw. *Temporal Dead Zone*. W przypadku odwołania się do zmiennej typu *let* czy *const* przed zadeklarowaniem otrzymamy błąd *ReferenceError: variable_name is not defined*.

28. Jaka jest różnica między *var*, *let* i *const*? Który jest preferowany?

Przede wszystkim to wcześniej wspomniane już zachowanie związane z hoistingiem. Oprócz tego zmienne typu *let* oraz *const* mogą tworzyć zasięg blokowy. Dodatkowo jak nazwa wskazuje, *const* jest „wartością stałą", czyli nie można takiej zmiennej ponownie przypisać wartości. Najbardziej preferowany w użyciu jest *const*, w drugiej kolejności let, a w przypadkach, gdy nie możemy ich użyć, pozostaje nam użycie *var*. Wynika to z dobrych praktyk i zwiększonej czytelności kodu.

29. Czym jest destrukturyzacja?

Za pomocą destrukturyzacji w prosty sposób można przypisywać właściwości obiektów do lokalnych zmiennych. Destrukturyzacja pomocna jest też w przypadku importowania zmiennych, klas i metod z innych plików i paczek.

```
const person = {
  name: 'John',
  surname: 'Doe'
};

const { surname } = person;

console.log( surname ); // Doe
```

Dzięki destrukturyzacji nie jest konieczne każdorazowe odwoływanie się do obiektu *person*. Zamiast tego można odwołać się do lokalnej zmiennej *surname*. Zwiększa to istotnie czytelność kodu.

30. Czym jest *pure function*?

Aby funkcję można było nazwać pure, musi ona spełniać dwa wymagania:

- Dla takiego samego zestawu danych wejściowych zawsze musi zwracać taki sam wynik,
- Pure function nie może modyfikować wartości (np. zmiennych poza swoim zakresem czy danych wejściowych).

Poniżej znajdziesz przykłady funkcji *pure* i *impure*:

```
let a = 2;

// Pure function
function add( a, b ) {
  return a + b;
}

// Impure function - modyfikuje zmienną spoza swojego zakresu
function increment() {
  return a++;
}

// Impure function - dla takich samych danych wejściowych
// wynik za każdym razem będzie inny
function random( a ) {
  return Math.random() * a;
}

console.log( add( 2, 4 ) );
increment();
console.log( a );
console.log( random( 3 ) );
```

31. Do czego służą wyrażenia regularne?

Wyrażenia regularne umożliwiają sprawdzenie, czy dana wartość jest zgodna ze zdefiniowanym wzorcem. Wyrażenia regularne nazywane są też regexami. Wyrażenia regularne mogą być wykorzystywane do wyszukiwania w tekście, walidacji danych czy modyfikowania tekstu.

32. Czym się różni function declaration od function expression?

Function declaration to sposób definiowania funkcji za pomocą słowa kluczowego *function*, nazwy funkcji i bloku kodu. *Function declaration* zostają załadowane, zanim kod zostanie wykonany. Natomiast *function expression* to *function declaration* przypisane do zmiennej. Powoduje to, że *function expression* zostaje załadowane dopiero w momencie, gdy interpreter napotka je w kodzie. Wywołanie *function expression* przed jego faktycznym wystąpieniem w kodzie będzie skutkować błędem, ponieważ albo zmienna uległa hoistingowi, lub też w przypadku *let* i *const* taka zmienna znajduje się w TDZ.

```
console.log( add( 2, 4 ) );
console.log( random( 3 ) );

//Function expression
var add = function( a, b ) {
  return a + b;
}

//Function declaration
function random( a ) {
  return Math.random() * a;
}
```

Uruchomienie powyższego fragmentu kodu spowoduje błąd. Przenosząc wywołanie funkcji pod *function expression*, kod zacznie działać poprawnie.

33. Czym są Local Storage, Session Storage oraz cookies?

Te trzy mechanizmy pozwalają na przechowywanie niewielkich porcji danych w przeglądarce w celu ich późniejszego użycia. Do takich danych można zaliczyć wszelkiego rodzaju identyfikatory, tokeny, konfiguracje itp.

Podstawową różnicą między Local Storage, Session Storage a cookies jest to, że cookies możemy odczytać z poziomu serwera. Zawartość Local Storage i Session Storage można odczytać tylko z poziomu przeglądarki.

Kolejną różnicą jest to, że cookies mają ograniczony czas życia. Można ustawić, aby czas życia ciasteczka wynosił np. 1 dzień lub 1 godzinę. Inaczej sprawa wygląda przy Local Storage i Session Storage. W przypadku pierwszego z nich, dane można usunąć poprzez wykorzystanie Web Storage API lub wyczyszczenie danych ręcznie. Session Storage traci dane po zakończeniu sesji strony.

Różnica jest też w maksymalnych rozmiarach. Cookies oferują maksymalnie 4 KB (niektóre przeglądarki oferują więcej), Local Storage 5 MB (wg specyfikacji HTML5) z możliwością rozszerzenia, a rozmiar Session Storage limituje rozmiar pamięci podręcznej.

34. Jakie znasz rodzaje pętli?

- *for* – pętla występująca w większości języków programowania. Jej zasada jest oparta na stworzeniu iteratora, zdefiniowaniu warunku zakończenia pętli oraz modyfikacji iteratora podczas każdej iteracji;
- *while* – kolejna klasyczna pętla. Pętla ta wykonuje kolejne iteracje, dopóki warunek w niej zawarty jest prawdziwy;
- *do...while* – zasada działania tej pętli jest identyczna z działaniem pętli while. Tym, co ją odróżnia, jest to, że ta pętla najpierw wykonuje kod, a dopiero potem sprawdza warunek;
- *for...in* – pętla ta pozwala nam na iterowanie po właściwościach obiektu;
- *for...of* – dzięki tej pętli można iterować po wartościach w typach danych, które umożliwiają iterowanie (np. tablice, stringi, obiekty typu *Map* czy *Set*).

35. Czym są *event bubbling* i *event capturing*?

Event bubbling można skojarzyć z bąbelkami w wodzie gazowanej. Bąbelki wypływają ku powierzchni i bardzo podobnie działa mechanizm event bubblingu. Zdarzenia są przekazywane od najbardziej wewnętrznego elementu, którego zdarzenie dotyczy, do najbardziej zewnętrznego. Przykładowo, mając element *section* z zagnieżdżonym elementem *div* i zdefiniowanym w nim elemencie *p*, event kliknięcia wykonany na elemencie *p* zostanie przekazany w następującej kolejności: *p > div > section*.

Drugim zagadnieniem jest *event capturing*. W przypadku event captiuringu sytuacja wygląda odwrotnie. W fazie przechwytywania zdarzeń przeglądarka rozpoczyna od najbardziej zewnętrznego elementu DOM i przechodzi w głąb hierarchii dokumentu, docierając do elementu docelowego, na którym zdarzenie zostało wywołane. Dla podanego wcześniej przykładu, flow wygląda następująco: *section > div > p*.

Domyślnym zachowaniem przeglądarek jest *event bubbling*. Możliwe jest też zatrzymanie propagacji zdarzeń przy użyciu metody *event.stopPropagation()*.

36. Czym jest IIFE?

IIFE jest akronimem od *Immediately Invoked Function Expression*. Jest to funkcja opakowana w dodatkowe okrągłe nawiasy, a za nimi jest dostawiona kolejna para nawiasów. Taki zabieg sprawia, że funkcja wewnątrz tych nawiasów jest wywoływana od razu, gdy interpreter napotka ją w kodzie. Przykład wykorzystania IIFE znajdziesz poniżej:

```
( () => {
  console.log( 'foobar' );
} )();
```

37. Czym jest *property descriptor*?

Property descriptor jest opisem właściwości danego obiektu. Deskryptor dla wybranej właściwości można podejrzeć za pomocą metody *Object.getOwnPropertyDescriptor(object , 'property')*.

```
const car = {
  model: 'Golf',
  brand: 'Volkswagen',
  engine: '1.9TDI'
};

console.log(Object.getOwnPropertyDescriptor(car, 'engine'));

// Expected output:
// [object Object] {
//   configurable: true,
//   enumerable: true,
//   value: "1.9TDI",
//   writable: true
// }
```

W rezultacie wywołania tej metody zwracany jest obiekt z czterema właściwościami: *configurable*, *enumerable*, *value* i *writable*. Ustawienie wartości *writable* na *false* sprawi, że wartość stanie się wartością tylko do odczytu. Wartość *enumerable* odpowiada, za to, czy można po niej iterować za pomocą np. *Object.keys()* czy pętli *for... in*. Właściwość *configurable* odpowiada za to, czy daną właściwość można usunąć oraz, czy można zmieniać pozostałe deskryptory. Właściwość *value* przechowuje wartość.

38. [ZADANIE PRAKTYCZNE] Wypisz wszystkie liczby od 1 do 100 ale gdy liczba jest podzielna przez trzy – wypisz „Fizz”, pięć – wypisz „Buzz”, trzy i pięć wypisz „FizzBuzz”.

FizzBuzz jest klasykiem zadań rekrutacyjnych. Jego zadaniem jest sprawdzenie, czy kandydat potrafi napisać prosty program w języku, którego dotyczy rozmowa. Podejście do rozwiązania tego zadania pozwala też zweryfikować sposób myślenia kandydata. Przedstawione rozwiązanie jest tylko jednym z wielu i zachęcam Cię do samodzielnego znalezienia innego rozwiązania:

```
for ( let i = 1; i <= 100; i++ ) {
  if ( i % 3 === 0 && i % 5 === 0 ) {
    console.log( 'FizzBuzz' );
  } else if ( i % 3 === 0 ) {
    console.log( 'Fizz' );
  } else if ( i % 5 === 0 ) {
    console.log( 'Buzz' );
  } else {
    console.log( i );
  }
}
```

Pozostawiłem w powyższym kodzie pole do usprawnień, co można zrobić tu lepiej? :)

39. [ZADANIE PRAKTYCZNE] Napisz funkcję, która przyjmie ciąg znaków (zdanie) i ciąg znaków z odwróconą kolejnością słów. Przykład - dla „Ala ma kota" funkcja ma zwrócić „kota ma Ala".

Celem tego zadania jest sprawdzenie umiejętności znajomości podstawowych funkcji i mechanizmów języka JavaScript. Pierwsza z wykorzystanych funkcji to *split()*. Funkcja split podzieli stringa na poszczególne słowa oraz każde ze słów umieści jako osobny element w tablicy. Funkcja ta jako parametr przyjmuje string, który posłuży nam do rozdzielenia elementów – w naszym wypadku będzie to spacja. Drugą z wykorzystanych funkcji jest *reverse()*. Służy ona do odwrócenia naszej nowo powstałej tablicy. Ostatnią wykorzystaną funkcją będzie *join()* łączący tablicę w całość. Domyślnie *join()* do łączenia używa przecinka, tak konieczne jest podanie stringa ze spacją jako parametr, aby powstało zdanie. Proponowane rozwiązanie prezentuje się następująco:

```
const sentence = 'Ala ma kota';

function reverseSentence( str ) {
  return str.split( ' ' ).reverse().join( ' ' );
}

console.log( reverseSentence( sentence ) );
```

40. [ZADANIE PRAKTYCZNE] Jakie znasz sposoby wyczyszczenia tablicy z elementów?

Pokażę dwa przykładowe rozwiązania. Pierwsze z nich polega na nadpisaniu tablicy pustą tablicą. Jest to rozwiązanie dość oczywiste, ale wymaga, by tablica była zadeklarowana z wykorzystaniem słów kluczowych *var* lub *let*. Drugim ze sposobów jest ustawienie właściwości *length* tablicy na wartość 0. Oba te działania w rezultacie dadzą nam pustą tablicę.

```
//Metoda nr 1
let arr1 = [ 'hello', 'world' ];
arr1 = [];
console.log( arr1 );

//Metoda nr 2
const arr2 = [ 'hello', 'world' ];
arr2.length = 0;
console.log( arr2 );
```

41. [ZADANIE PRAKTYCZNE] Jak pozbyć się duplikatów z tablicy w JS?

Przedstawię dwa przykładowe rozwiązania. Pierwsze z nich to skorzystanie z obiektu *Set*. Cechą *Set* jest to, że nie pozwala na przechowywanie duplikatów. W połączeniu z wykorzystaniem *spread* operatora, rozwiązanie mamy prawie gotowe. Tak otrzymany *Set* traktujemy jeszcze raz *spread* operatorem i otrzymujemy tablicę bez duplikatów:

```
const arr = [ 'foo', 'bar', 'baz', 'bar', 'bar', 'baz' ];

console.log( [ ...new Set( [ ...arr ] ) ] );
```

Drugim sposobem jest przeiterowanie po tablicy pierwotnej i przenoszenie wartości z jednej tablicy do drugiej.

```
const arr = [ 'foo', 'bar', 'baz', 'bar', 'bar', 'baz' ];

const noDuplicates = [];

for( const el of arr ) {
  if( !noDuplicates.includes( el ) ) {
    noDuplicates.push( el );
  }
}

console.log( noDuplicates );
```

Trzeba tutaj wspomnieć, że obie z tych metod zadziałają skutecznie jedynie w przypadku tablic składających się z typów prostych. W typach referencyjnych porównywane są referencje, a nie wartości. Dwa obiekty o różnych referencjach, lecz identycznej strukturze nie będą traktowane jako duplikaty. Zachęcam, aby przeanalizować poniższy przykład:

```
const obj = { a: 2 };
const arr = [
  'foo', 'bar', 'baz', 'bar', 'bar', 'baz', obj, obj, { a: 1 }, { a: 1 }
];

console.log( [ ...new Set( [ ...arr ] ) ] );
```

42. [ZADANIE PRAKTYCZNE] Zaimplementuj stos (Stack)

Zaproponowana implementacja stosu będzie zawierała dwie funkcje — jedna do wrzucania elementu na stos oraz droga do zdejmowania elementu ze stosu. Stos jest strukturą LIFO, czyli *Last In First Out*. Oznacza to, że ostatni umieszczony na stosie element jest z niego zdejmowany w pierwszej kolejności. Przykładowa implementacja będzie wyglądała następująco:

```
const stack = {
  items: [],
  push: item => stack.items.unshift( item ),
  pop: () => {
    if ( !stack.items.length ) {
      console.warn( 'no items to pop' );

      return;
    }

    stack.items.shift();
  }
}

stack.push( 9 );
stack.push( 4 );
stack.push( 1 );
stack.push( 5 );
stack.push( 5 );
stack.push( 16 );
stack.push( 2 );
stack.pop();
stack.pop();
console.log( stack.items ); // expected output: [5, 5, 1, 4, 9]
```

43. [ZADANIE PRAKTYCZNE] Do wykonania zadanie polegające na wysłaniu Ajaxem obiektu i zapisaniu w Local Storage tokena, który dostanie się w ramach odpowiedzi. W przypadku błędu wyświetlić komunikat o błędzie.

Z tego typu zadaniami spotkasz się w swojej codziennej pracy. Do wykonania tego zadania możesz wykorzystać *Fetch API* lub dowolnego innego klienta HTTP. Wykorzystując *Fetch API*, nie jest konieczna instalacja dodatkowych zależności. W proponowanym rozwiązaniu metoda *fetch()* przyjmie dwa parametry: ścieżkę do zasobu oraz obiekt, w którym zdefiniowana zostanie metoda HTTP i obiekt z danymi, który otrzymamy w zadaniu.

```
fetch( 'https://example.com/token', {
    method: 'POST',
    body: JSON.stringify(obj)
} )
    .then( response => response.json() )
    .then( response => {
        localStorage.setItem( 'token', response.data.token );
} )
    .catch( error => console.error( 'An error occured:' , error ) );
```

44. [ZADANIE PRAKTYCZNE] Sprowadź następującą strukturę *[['a'], ['b'], [['c', 'd']]]* do postaci płaskiej tablicy z elementami *['a', 'b', 'c', 'd']*.

Jest to kolejne z dość prostych zadań sprawdzających sposób myślenia kandydata i jego znajomość języka. Można je rozwiązać na wiele sposobów. Zaproponowane rozwiązanie jest jednym z prostszych i wykorzystuje wbudowane metody języka JavaScript. Alternatywnie można próbować rozwiązać to zadanie z wykorzystaniem np. funkcji rekurencyjnej.

```
const nestedArray = [ [ 'a' ], [ 'b' ], [ [ 'c', 'd' ] ] ];

console.log( nestedArray.flat( Infinity ) );
```

TypeScript

TypeScript staje się już praktycznie standardem w ekosystemie JavaScriptu. Większość popularnych rozwiązań JavaScriptowych udostępnia typy, które można wykorzystać w projektach TypeScriptowych. Oprócz silnego typowania TypeScript daje szereg innych możliwości, które znacznie usprawniają pracę. W tym rozdziale zawarłem kilka pytań dotyczących TypeScripta, które sprawdzają wiedzę kandydata o przeznaczeniu i jego podstawowych mechanizmach. Do uruchamiania kodu z tego rozdziału możesz wykorzystać środowisko TypeScript Playground.

45. Czym się różni typ *any* od typu *unknown*? Dlaczego używanie *any* jest złą praktyką?

Typ *any* mówi, że zmienna tego typu może być czymkolwiek. Oznacza to, że gdy w danym miejscu kompilator spodziewa się przekazania konkretnego typu, to można tam również przekazać wartość typu *any* i nie spowoduje to błędu transpilacji.

```
const anyValue: any = true;

function x( a: string ): void {
   // code
}

function y( a: number ): void {
   // code
}

x( anyValue );
y( anyValue );
```

W obu przypadkach kompilator TypeScript nie zasygnalizuje błędu. Korzystanie z typu *any* powinno się zredukować do minimum. Jedną z głównych zalet TypeScripta jest właśnie typowanie, więc korzystanie z *any* jest złą praktyką. Typ *unknown* jest bardzo podobnym typem do *any*. Jak nazwa *any* sugeruje, że wartość może być czymkolwiek, tak typ *unknown* sugeruje, że typ wartości pozostaje nieznany.

Typ *unknown* nie pozwala na pewne rzeczy, na które pozwala typ *any*. Po pierwsze, do zmiennej określonego typu, np. *number* można przypisać wartość typu *any*, lecz nie można przypisać wartości typu *unknown*. Po drugie, na wartościach typu unknown nie można wywoływać metod zarezerwowanych dla innych typów np. *toString* czy *isNan*. Oczywiście dotyczy to też bardziej złożonych czy typów definowanych przez nas samych.

```
const a: any = true;
const b: unknown = true;

let c: number;

c = a; // Ok
c = b; // Błąd - Object is of type 'unknown'.(2571)

a.toString(); // Ok
b.toString(); // Błąd - Object is of type 'unknown'.(2571)
```

Oczywiście istnieje możliwość na wywołanie metody *toString* na wartości typu *unknown* poprzez skorzystanie z operatora *as* i casting na inny typ zawierający tę metodę, ale to również nie należy do dobrych praktyk. Z obu typów, zdecydowanie lepiej jest wybrać typ *unknown*, ale jeszcze lepiej jest redukować użycie obu typów do niezbędnego minimum, co oczywiście nie zawsze jest możliwe.

46. Czym są *type guards* i w jakim celu się je stosuje?

Type guards to konstrukcja wykorzystywana do doprecyzowania typu w przypadku gdy zmienna może przyjmować więcej niż jeden typ.

```
const a: string | number = getValue();

function isNumber( a: string | number ): a is number {
  return typeof a === "number";
}
```

Metoda *getValue* może zwrócić zarówno typ *string*, jak i *number*. Użycie *type guard* pozwala zweryfikować jaki typ został użyty. Oczywiście można pokusić się o skorzystanie tylko z operatora *typeof*, natomiast użycie *type guard* powoduje zapamiętanie informacji o typie zmiennej przez kompilator, przez co nie ma konieczności każdorazowego sprawdzania jej typu.

47. W jaki sposób można „wyłączyć/zawiesić" kompilator TypeScript dla konkretnej linii kodu? Kiedy ma to sens?

Odpowiedź brzmi – tak, da się. TypeScript oferuje kilka klauzul pozwalających na dodawanie wyjątków dla kompilatora:

- *@ts-ignore* – w przypadku gdy kompilator TypeScript zasygnalizuje wystąpienie błędu, umieszczenie tej klauzuli linię wyżej, pozwala na zignorowanie tego błędu przez kompilator;
- *@ts-expect-error* – klauzula umieszczana w miejscu, gdzie programista spodziewa się wystąpienia błędu. Jeżeli błąd nie wystąpi, to samo wystąpienie klauzuli w kodzie będzie interpretowane jako błąd;
- *@ts-nocheck* – kod poniżej tej klauzuli nie będzie sprawdzany przez kompilator TypeScript. Jest to blokowy odpowiednik dla *@ts-ignore*;
- *@ts-check* – odwrotność powyższej klauzuli pozwalający na przywrócenie sprawdzania kodu.

Stosowanie tych klauzul powinno być ostatecznością. Zdecydowanie lepiej jest znaleźć przyczynę problemu i naprawić problem niż korzystać z tych klazul. Uzasadnionym przypadkiem użycia może być wykorzystanie błędnie otypowanych bibliotek. Oczywiście zawsze można spróbować wystawić pull requesta w repozytorium takiej biblioteki, ale nie każda biblioteka jest utrzymywana i nie zawsze jest na to czas.

Warto także rozważyć włączenie reguły *@typescript-eslint/ban-ts-comment* w swojej konfiguracji ESLint z wartością *allow-with-description* oraz ustawionym parametrem *minimumDescriptionLength*. Sprawi to, że ESLint zaakceptuje powyższe klauzule, ale tylko jeżeli będą zawierały wyjaśnienie o pewnej wymaganej długości znaków. Dzięki temu, każde użycie klauzul będzie nieco bardziej przemyślane i wyjaśnione szerzej niż *bug* czy *issue*.

48. Czym są tzw. „generyki”? Podaj przykład zastosowania.

„Generyki” lub też typy generyczne pozwalają na dynamiczne przekazanie typu do konstrukcji w kodzie, na przykład funkcji czy interfejsów. Załóżmy, że mamy API, które zwraca listę kotków oraz piesków. Dodatkowo lista ta wspiera paginację. Zwrócony obiekt z API będzie zawierał: numer strony, kursor do poprzedniej strony, kursor do następnej strony, oraz listę zwróconych kotków lub piesków. Poniższy przykład pokazuje, jak można stworzyć reużywalny interfejs, który będzie w stanie obsłużyć zarówno kotki, jak i pieski:

```
interface ICat {
   // Properties
}

interface IDog {
   // Properties
}

interface IPaginationResult<T> {
   page: number;
   prev_page: null | string;
   next_page: null | string;
   items: T[];
}

function getDogs(): IPaginationResult<IDog> {
   // code
}

function getCats(): IPaginationResult<ICat> {
   // code
}
```

Dzięki użyciu generyków, za pomocą jednego interfejsu zdefiniowano kształt zwracanego obiektu dla dwóch metod. Co więcej, nic nie stoi na przeszkodzie, aby użyć tego interfejsu do zwrócenia np. ptaszków czy rybek w przyszłości.

49. Do czego służą typy *enum* oraz *tuple*?

Typ *enum*, czyli inaczej typ wyliczeniowy pozwala na zdefiniowanie zestawu nazwanych stałych, zarówno w formie numerycznej, jak i formie łańcucha znaków.

```
enum Colors {
  Red,
  Green,
  Blue
}

console.log( Colors.Red ); // 0
```

Tak zdefiniowany typ wyliczeniowy, przypisze rosnąco liczby całkowite, zaczynając od 0 dla każdego zdefiniowanego elementu. Oczywiście można również zdefiniować te liczby ręcznie. Możliwe również jest zdefiniowanie wartości jako łańcucha znaków.

```
enum Colors {
  Red = 1,
  Green = 2,
  Blue = 3
}

console.log( Colors.Red ); // 1

enum Colors {
  Red = '#FF0000',
  Green = '#00FF00',
  Blue = '#0000FF'
}

console.log( Colors.Red ); // '#FF0000'
```

Typ *tuple* lub też inaczej krotka jest strukturą danych pozwalającą na zdefiniowanie listy z wartościami o konkretnych typach w konkretnych polach. Można doszukać się tu analogii do struktur tabel i rekordów w relacyjnych bazach danych.

```
let user: [ string, number, string ]; // Imię, wiek, zawód

user = [ 'Steve', 21, 'Programmer' ];
```

50. Jakie typowanie wykorzystuje TypeScript i jakie niesie to konsekwencje?

TypeScript wykorzystuje typowanie strukturalne, co niesie za sobą kilka konsekwencji – zarówno pozytywnych, jak i negatywnych. Pierwszą konsekwencją jest to, że pozwala to na pewną dowolność w przypisywaniu wartości do zmiennych o określonym typie. Kompilator TypeScript sprawdza jedynie, czy struktura przypisywanego obiektu jest zgodna ze strukturą oczekiwaną.

```
interface Pet {
   name: string;
}

interface Person {
   name: string;
}

let steve: Person;

const nemo: Pet = {
   name: 'nemo'
}

steve = nemo;
```

Jest to broń obosieczna, z uwagi na to, że programista zyskuje większą elastyczność, lecz kosztem sytuacji gdzie możemy przypisać obiekt tylko na podstawie zgodności interfejsów. W przykładzie powyżej osoba *steve* została zwierzakiem o imieniu *nemo*.

Kolejną konsekwencją typowania strukturalnego jest to, że w przypadku typowania przy użyciu klas sprawdzane są wszystkie pola klasy, również te prywatne. Jest to w pewien sposób zrozumiałe i oczekiwane zachowanie z punktu widzenia typowania strukturalnego. Niestety w większych projektach, gdzie wykorzystane są moduły w różnych wersjach z subtelnymi zmianami w klasach, może to powodować sporo problemów. Co więcej, prywatne właściwości nie są dostępne spoza klasy, a mimo to wciąż mają swój udział w typowaniu co również może powodować spore problemy. Stąd też zdecydowanie odradzam typowanie przy użyciu klas na rzecz interfejsów.

```
class Person {
   private _age: number;

   constructor( private readonly _name: string ) {
      this._age = 21;
   }
}

class Pet {
   constructor( private readonly _name: string ) {}
}

let steve: Person;

const nemo: Pet = new Pet( 'nemo' );

steve = nemo; // Błąd - Property '_age' is missing in type 'Pet' but required in typ
```

51. Wymień główne korzyści, jakie niesie ze sobą TypeScript.

Przede wszystkim są to typy, zarówno wbudowane, jak i możliwość tworzenia własnych. TypeScript oferuje wiele możliwości w zakresie typowania, co pozwala tworzyć o wiele czystszy kod niż w JavaScript. Typy pozwalają wyeliminować lub zredukować sporo błędów, które można popełnić w JavaScript, a na które nie pozwoli kompilator TypeScript. Kod w TypeScript jest znacznie bardziej przewidywalny i łatwiejszy w debugowaniu. Kolejna z zalet to domyślna implementacja eksperymentalnych funkcji tj. dekoratory. Oprócz tego TypeScript oferuje modyfikatory dostępu, które w czystym JavaScript są od niedawna. TypeScript pozwala też na wykorzystywanie kodu JavaScript, co oznacza, że nie jest konieczne przepisywanie całego codebase od razu. Można to robić stopniowo.

Git

Git jest jednym z najpopularniejszych systemów kontroli wersji. Obecnie, coraz ciężej jest znaleźć projekt czy firmę, gdzie nie wykorzystuje się Gita. Popularność gita wynikająca z ogromu możliwości i istotnego zwiększenia komfortu pracy sprawia, że bez znajomości Gita ciężko będzie Ci znaleźć pracę w sensownym projekcie. Przedstawione pytania sprawdzają podstawową znajomość Gita oraz zdolność kandydata do rozwiązywania podstawowych problemów, jakie mogą wystąpić podczas pracy z Gitem.

52. Czym jest Git i systemy kontroli wersji? Do czego je wykorzystujemy?

Odpowiedź na pytanie, czym jest Git, zawarłem we wstępie do tego rozdziału. Z kolei systemy kontroli wersji to narzędzia rozwiązujące kilka istotnych problemów:

- Przede wszystkim systemy kontroli wersji umożliwiają to, co sama nazwa wskazuje, czyli wersjonowanie projektu. Mowa tu o możliwości prześledzenia każdej zmiany, która zaistniała w projekcie. Oznacza to również możliwość cofnięcia się do dowolnego poprzedniego stanu projektu.
- Systemy kontroli wersji pomagają dokumentować zmiany w projekcie. W Gicie podstawowym bytem jest commit, którego elementem składowym jest opis. Na opis może składać się informacja, co zostało zmienione, ale może on też zawierać cel tej zmiany. Zachęcam do zapoznania się z poradnikiem jak pisać dobre opisy commitów, bo wbrew pozorom nie jest to łatwe zadanie.
- Dzięki systemom kontroli wersji możliwa jest praca nad kilkoma zadaniami jednocześnie, niezależnie od siebie. Do tego celu został stworzony mechanizm gałęzi (branchy). Gałęzie jednocześnie zwiększają wygodę pracy nad projektem w zespołach. Mechanizm gałęzi pozwala na pracę wielu programistów nad tym samym kodem bez przeszkadzania sobie nawzajem.

53. Jaka jest różnica między Gitem a GitHubem?

Pytanie nieco żartobliwe, ale jego cel jest taki sam jak poprzedniego pytania. Można na nie odpowiedzieć żartobliwie na przykład: *„taka jak pomiędzy krzesłem a krzesłem elektrycznym".*

Można także odpowiedzieć całkowicie poważnie. Git jest systemem kontroli wersji, natomiast GitHub jest platformą zajmującą się hostingiem kodu źródłowego wykorzystującą Gita oraz dostarczającą cały zestaw udogodnień i narzędzi do pracy nad projektem.

54. Opisz podstawowe flow pracy z Gitem.

Zakładając, ze zaczynamy pracę z istniejącym projektem, trzeba go najpierw sklonować poleceniem *git clone*. Przed rozpoczęciem pracy dobrą praktyką jest rozpoczęcie pracy na osobnej gałęzi. Do tego można wykorzystać polecenie *git checkout -b nazwa_gałęzi*. Polecenie stworzy nową gałąź i ustawi ją jako aktywną. Kolejnym etapem jest dodanie zmian w kodzie i zindeksowanie ich poleceniem *git add*. Aby z zindeksowanych zmian utworzyć commit, czyli punkt kontrolny w historii projektu, który zawiera informacje o wprowadzonych zmianach. W tym celu należy użyć polecenia *git commit -m "opis zmian"*. Ostatnim etapem jest wysłanie zmian poleceniem *git push nazwa_zdalnego_repozytorium nazwa_gałęzi*.

55. Jak można przeglądać historię zmian w Gicie?

Podstawowym poleceniem do wyświetlania historii zmian jest polecenie *git log*. Można do tego celu posłużyć się również poleceniem *git shortlog*, które w przejrzystej formie wyświetli listę kontrybutorów wraz z commitami.

56. Czym się różni *git fetch* od *git pull*?

Polecenie *git fetch* jedynie pobiera dane o zmianach ze zdalnego repozytorium. Polecenie *git pull* oprócz pobrania zmian ze zdalnego repozytorium podejmuje próbę scalenia ich (*merge*) z aktywną gałęzią. Podczas scalania może wystąpić konflikt, który trzeba będzie rozwiązać.

57. Czym jest *squash* commitów?

Squash commitów jest scaleniem kilku commitów w jeden. Na przykład, w serwisie GitHub takie zachowanie można zdefiniować w momencie scalania pull requestów.

W takim przypadku wszystkie commity ze scalanego pull requesta zostaną połączone w jeden, a następnie scalone z gałęzią docelową. W samym Gicie można taki rezultat uzyskać dzięki poleceniu *git rebase*.

58. Do czego służy Git LFS?

Git nie jest najlepszy miejscem do przechowywania dużych plików, czy plików binarnych tj. obrazy, filmy czy pliki audio. W tym celu powstało narzędzie o nazwie Git LFS (Large File System). Dzięki temu, w repozytorium nie są przechowywane pliki a jedynie wskaźniki (referencje) do nich. Typy plików, przechowywanych przez Git LFS można zdefiniować w pliku *.gitattributes*.

59. Opisz stany, w jakich może znajdować się zmiana w Gicie.

Wyróżniamy trzy stany, w jakich może znajdować się zmiana w repozytorium. Pierwszym stanem jest stan *modified*. W tym stanie są domyślnie wszystkie modyfikacje poczynione od ostatniego commita. Drugim stanem jest stan *staged*. Aby zmiana znalazła się w tym stanie, należy skorzystać z polecenia *git add* i wybrać zmiany. Przy poleceniu *git commit* pod uwagę brane są tylko zmiany będące w stanie *staged*. Ostatnim stanem jest stan *commited*, czyli stan, w którym zmiana została zapisana w repozytorium. Przy commitowaniu możliwe jest pominięcie stanu *staged* poprzez skorzystanie z odpowiedniej flagi.

60. Do czego służą gałęzie w Gicie?

Gałęzie w Gicie pozwalają na pracę nad różnymi funkcjonalnościami, naprawami błędów lub eksperymentalnymi zmianami w izolowanych środowiskach, nie ingerując w główną linię kodu (gałąź główną). Dzięki gałęziom można bezpiecznie rozwijać projekt, a następnie po przetestowaniu i zaakceptowaniu zmian, łączyć je z gałęzią główną.

61. Jak się nazywa mechanizm pozwalający skopiować *commit* na inną gałąź?

Do kopiowania commitów i dodawania ich na inne gałęzie służy mechanizm cherry pick.

Wykonanie polecenia *git cherry-pick sha_commita* spowoduje kopię commita z podaną referencją i dodaniem go do aktywnej gałęzi. Referencję do commita możemy uzyskać za pomocą polecenia *git log*.

62. Czym jest konflikt w Gicie i jak może zostać rozwiązany?

Konflikt w Gicie może zaistnieć w następującej sytuacji:

1. Programista A tworzy branch *feature* i wprowadza na nim zmiany w istniejącym już wcześniej pliku.
2. W międzyczasie programista B na branchu *master* wprowadza zmiany w tym samym pliku.
3. Programista A kończy pracę na branchu *feature*, pushuje wprowadzone zmiany i przełącza się na *mastera*.
4. Programista A chce zmerge'ować mastera z branchem *feature*...
5. ... i w tym momencie mamy konflikt. Programista B edytował ten sam plik, co programista A i system kontroli wersji nie wie, która wersja jest poprawna.

W tym momencie przed programistą A stoi zadanie na wybraniu poprawnej wersji. Brzmi to skomplikowanie, ale w praktyce jest całkiem proste. Wystarczy skasować niechciany kod a zostawić właściwy i... gotowe!

Kolejna książka o Gicie

Pytania powyżej to dopiero początek. Jeśli chcesz opanować Gita naprawdę dobrze, polecam "Kolejną książkę o Gicie". To praktyczny przewodnik, który przeprowadzi Cię od podstaw aż do zaawansowanych technik używanych w realnych projektach.

Na dobry start mam dla Ciebie kod rabatowy **NAUKA10**, który obniża cenę ebooka.

Docker

Podobnie jak Git, konteneryzacja już na dobre zagościła w wielu projektach i firmach. Nawet jeśli wiążesz swoją przyszłość z frontendem, to podstawowa wiedza o konteneryzacji i tak Ci się przyda. Docker jest jednym z najpopularniejszych narzędzi do pracy z kontenerami. Pytania z tego rozdziału sprawdzą Twoją wiedzę o podstawach konteneryzacji i znajomości podstaw Dockera.

63. Czym jest Docker i do czego służy?

Docker jest narzędziem służącym do uruchamiania kontenerów. Dzięki kontenerom można w prosty sposób uruchamiać aplikacje bez fizycznego instalowania ich na wykorzystywanej maszynie. Przykładowo, chcąc uruchomić bazę danych, nie trzeba instalować jej lokalnie. Wystarczy, że uruchomiony zostanie kontener, bazujący na obrazie wybranej bazy danych oraz zostanie zdefiniowane, jak można się z nim komunikować. W ten sam sposób można dokonać konteneryzacji dowolnej aplikacji. Istotnym aspektem konteneryzacji z wykorzystaniem Dockera jest to, że kontenery nic o sobie nie wiedzą, dopóki użytkownik jawnie nie zadeklaruje, że ma być inaczej. Czyli posiadając kontener z bazą danych i kontener z rozwijaną aplikacją należy jeszcze zdefiniować, jak te dwa kontenery powinny się ze sobą komunikować.

64. Jaka jest różnica między obrazem i kontenerem?

Obrazem można nazwać *„snapshot"* aplikacji, która będzie uruchamiana. Używając analogii z programowania obiektowego, obraz można przyrównać do klasy. Oznacza to, że mając jeden obraz, można powołać do życia wiele kontenerów. Aby uruchomić własną aplikację z użyciem Dockera niezbędne będzie utworzenie dla niej *Dockerfile*. Plik *Dockerfile* zawiera w sobie listę kroków niezbędną do zbudowania obrazu aplikacji. Natomiast kontener jest instancją aplikacji uruchomionej na podstawie obrazu. Przyrównując to do programowania obiektowego, kontener byłby obiektem stworzonym na podstawie klasy (obrazu).

65. Czym jest *Docker Compose*?

Docker Compose jest narzędziem usprawniającym pracę z wieloma kontenerami. Dzięki *Docker Compose* użytkownik może uruchomić środowisko z wieloma kontenerami za pomocą jednej komendy. Aby skorzystać z *Docker Compose*, niezbędne jest stworzenie pliku konfiguracyjnego (domyślnie plik *docker-compose.yml*). Dzięki temu narzędziu można na przykład przekazać do kontenerów zmienne środowiskowe, zdefiniować i skonfigurować sieć do komunikacji między kontenerami czy wymusić ich uruchamianie w określonej kolejności i na podstawie konkretnych wytycznych.

66. Do czego służą wolumeny?

Dzięki wykorzystaniu wolumenów (*volume*) tworzony jest link między wskazaną lokalizacją w kontenerze a wskazaną lokalizacją na lokalnej maszynie. Taki zabieg pozwala, np. na bieżąco obserwować jak zmiany w kodzie źródłowym wpływają na działanie aplikacji bez potrzeby przebudowywania obrazu po każdej zmianie. Innym wykorzystaniem wolumenów może być np. zapis logów z aplikacji do pliku czy przekazywanie inicjalnego stanu bazy danych do aplikacji przy starcie aplikacji.

67. Czym jest Open Container Initiative?

Open Container Initiative (OCI) jest odpowiedzialne za standaryzację w świecie konteneryzacji. Podążając za standardami wytyczanymi przez OCI, w przypadku potrzeby zastąpienia jednego z rozwiązań służących do konteneryzacji innym, migracja odbywa się w zdecydowanie mniej bolesny sposób. Można tu przywołać analogię do kontenerów używanych na statkach. Wymiary kontenerów używanych w transporcie morskim są standaryzowane. Dzięki temu doskonale wiadomo, jakie wymiary ma kontener, ile się do niego zmieści oraz ile kontenerów może zostać załadowanych na statek. Dzięki OCI w prosty sposób możesz podmienić w swoim projekcie Dockera np. na Podmana.

68. Jakie korzyści wynikają z konteneryzacji?

Najważniejsze zalety konteneryzacji to:

- brak konieczności uruchamiania aplikacji na lokalnej maszynie. Oznacza to też brak konieczności instalacji dodatkowych komponentów tj. serwer HTTP czy bazy danych;
- jeden standard wynikający z pracy OCI;
- przenaszalność (*portability*) – kontenery w łatwy sposób można uruchomić na każdej maszynie umożliwiającej uruchomienie Dockera. Co istotne, na każdej maszynie kontener bazujący na tym samym obrazie w tej samej konfiguracji będzie działał tak samo (chyba że na maszynie docelowej zabraknie zasobów, ale uważam to za *edge case*);
- izolacja – wystawienie endpointu dostępnego z zewnątrz dla kontenera wymaga jawnego zadeklarowania tego faktu. Domyślnie, kontenery są od siebie odizolowane w maksymalnym możliwym stopniu od maszyny, na której są uruchamiane;
- szybkie dostarczanie – gotowe obrazy można w szybki i prosty sposób udostępnić klientom i innym programistom. Ich uruchomienie przez odbiorcę jest równie szybkie i proste;
- konteneryzacja nie wymaga tylu zasobów co korzystanie z maszyn wirtualnych.

69. Jakie są dobre praktyki z pracą z kontenerami?

Lista dobrych praktyk jest znacznie dłuższa i jeśli znasz inne dobre praktyki to nie wahaj się o nich wspomnieć gdy padnie to pytanie.

- nie korzystaj z konta *root* w kontenerze. Korzystanie z konta *root* jest złą praktyką z punktu widzenia bezpieczeństwa. Niektóre obrazy, np. obraz dla Node.js oferują wbudowanego użytkownika;
- nie instaluj zbędnych zależności – np. nie potrzebujesz w obrazie produkcyjnym developerskich zależności;
- używaj ściśle zdefiniowanej wersji obrazów (z tagiem z konkretną wersją). Dodatkowo, nie korzystaj z obrazów oznaczonych tagiem *latest*. Pozwoli to na zachowanie determinizmu przy pracy z zewnętrznym obrazami. W innym razie nigdy nie ma pewności, która wersja obrazu zostanie wykorzystana;
- regularnie skanuj swoje obrazy w poszukiwaniu podatności;

- używaj lżejszych wersji zewnętrznych obrazów jeśli to możliwe. Przykładowo, obraz dla Node.js z tagiem *erbium-buster* dostępny w Docker Hub waży ponad 300MB, gdzie obraz *erbium-alpine3.14* waży około 30MB! Przy wyborze obrazu należy zadać sobie pytanie, czy któryś z lżejszych wariantów obrazów jest wystarczający;
- korzystaj z *multistage builds*. Przede wszystkim pozwala to na zmniejszenie rozmiaru ostatecznego obrazu. Oprócz tego, jest to dobra praktyka bezpieczeństwa w momencie, gdy przy budowaniu używasz tokenów bezpieczeństwa. W obrazie zapisane są informacje o każdej jego warstwie, co może spowodować wyciek tokenów bezpieczeństwa, które wykorzystano podczas budowania obrazu. *Multistage build* pozwala temu zapobiec. Należy jednak pamiętać, aby wrażliwe wartości nie były używane podczas ostatniego etapu budowania;
- ignoruj pliki, które nie powinny znaleźć się w obrazie za pomocą *.dockerignore;*
- minimalizuj liczbę warstw w obrazie. Ponadto, Docker potrafi cache'ować poszczególne warstwy, przez co umiejętne konstruowanie definicji obrazów pozwala znacznie skrócić czas przebudowania obrazu.

Bazy danych

Podobnie jak w przypadku Dockera, nawet jeśli nie wiążesz swojej przyszłości z backendem, ani nie planujesz zostać bazodanowym guru, to podstawy znajomości baz danych prędzej czy później Ci się przyda. W odpowiedziach założyłem, że wykorzystywanym system zarządzania baz danych jest MySQL, ale dla innych relacyjnych baz danych odpowiedzi będą bardzo podobne.

70. W jaki sposób można dowiedzieć się więcej o strukturze bazy danych oraz tabelach?

Do odkrywania elementów składowych bazy danych oraz struktury tabel wykorzystać można następujące zapytania:

- *SHOW DATABASES* — zwraca listę dostępnych baz danych;
- *SHOW TABLES* — zwraca listę tabel dla obecnie używanej bazy danych;
- *DESCRIBE nazwa_tabeli* — zwraca informacje o kolumnach we wskazanej tabeli. W celu uzyskania nieco większej ilości szczegółów można wykorzystać zapytanie *SHOW FULL COLUMNS FROM nazwa_tabeli*.

71. Do czego służy klauzula *JOIN*? Jakie znasz rodzaje *JOIN-ów*?

Klauzula *JOIN* jest opcjonalnym elementem klauzuli *SELECT*. Dzięki klauzuli *JOIN* możliwe jest łączenie danych z (najczęściej) kilku zbiorów. Klauzule *JOIN* opierają się na identyfikatorach zasobów, gdzie każdy zbiór wykorzystany w zapytaniu powinien zawierać kolumnę, po której będzie odbywać się łączenie.

Wybrane rodzaje *JOIN-ów*:

- *LEFT JOIN* — zwróci wyniki dla lewego zbioru oraz część wspólną dla obu zbiorów;
- *RIGHT JOIN* — zwrócona zawartość zboru dołączonego (prawego) oraz część wspólna;
- *INNER JOIN* — zwrócona zostaje jedynie wspólna część zbioru;
- *FULL JOIN* — powoduje złączenie całości zbiorów lewego oraz prawego;
- *SELF JOIN* — zbiór jest łączony sam ze sobą i znajduje zastosowanie, gdy elementy zbioru mają w sobie referencje do innych elementów z tego samego zbioru.

72. Do czego służy klauzula *EXPLAIN*?

Klauzula *EXPLAIN* umieszczona przed zapytaniem SELECT, DELETE, INSERT, REPLACE, lub UPDATE sprawia, że zamiast wyników, zapytanie zwraca opis przygotowanego zapytania. W opisie zawarte są informacje o tym, jak łączone są tabele i w jakiej kolejności, jakie indeksy zostały wykorzystane itp. Dzięki klauzuli *EXPLAIN* można łatwiej zdiagnozować nieefektywne zapytania oraz dowiedzieć się, co potencjalnie można w nich poprawić.

73. Na czym polega normalizacja? Jakie znasz postaci normalne?

Normalizacja baz danych polega na sprowadzeniu struktury tabel w bazie danych do takiej postaci, aby spełniały one założenia postaci normalnych. Najczęściej spotykane i stosowane postaci normalne to:

- pierwsza postać normalna (1NF) — każda wartość w bazie danych powinna być atomowa (inaczej mówiąc niepodzielna);
- druga postać normalna (2NF) — pierwszy warunek konieczny to spełnienie warunków pierwszej postaci normalnej. Drugi warunek konieczny mówi o tym, że wszystkie kolumny w tabeli muszą zależeć od klucza głównego;
- trzecia postać normalna (3NF) — pierwszym warunkiem koniecznym jest spełnienie warunków drugiej postaci normalnej. Drugi warunek konieczny do spełnienia mówi o tym, że niekluczowa kolumna nie może zależeć od innej niekluczowej kolumny.

O ile teorię normalizacji warto znać, tak w praktyce postaci normalne nie zawsze są restrykcyjnie wdrażane. W wielu przypadkach sprowadzanie wartości do bardziej restrykcyjnych postaci normalnych może być po prostu zbędne. Niemniej jednak znajomość postaci normalnych jest przydatną wiedzą i może Ci się przydać.

74. Do czego służy indeks w bazie danych?

Indeks jest mechanizmem usprawniającym wyszukiwanie, sortowanie i filtrowanie rekordów z wykorzystaniem wielu pól. Indeks tworzy nową strukturę danych, która przechowuje wartość pola oraz wskaźnik do rekordu. Przy odpytywaniu bazy danych, do wyszukiwania rekordów można wykorzystać indeks, co może przyspieszyć czas wykonywania zapytań. Indeks z baz danych można przyrównać do indeksów znajdujących się na końcu książek. W takich indeksach poszczególnym słowom-kluczom przypisane są odpowiednie numery stron, gdzie występują szukane frazy.

75. Na czym polegają relacje w relacyjnych bazach danych?

Aby lepiej zrozumieć pojęcie relacji, trzeba wcześniej wprowadzić pojęcia kluczy głównych i kluczy obcych. Klucz główny jest wartością pozwalającą na jednoznaczną identyfikację każdej encji. Każda encja musi posiadać klucz główny. Klucz główny jest unikalny w zasięgu pojedynczej tabeli i zwykle występuje pod postacią liczby lub ciągu znaków np. UUID. Klucz obcy jest referencją do klucza głównego innej tabeli. Dzięki niemu możliwe jest powiązanie encji z dwóch tabel. Przykładowo, w tabeli opisującej sprzęt firmowy, każdy sprzęt ma przypisanego pracownika. Kluczem obcy byłby w takim przypadku identyfikator przypisanego pracownika z tabeli z pracownikami.

Relacje pozwalają na odzwierciedlenie powiązań między encjami w relacyjnej bazie danych. Do stworzenia relacji, można wykorzystać klucze obce. Wyróżniamy kilka typów relacji:

- jeden do jednego (*1:1*) – jednej encji z tabeli A przypisana jest jedna encja z tabeli B. Przykładem może być relacja rodzina-dom. Jedna rodzina zwykle ma jeden dom. W jednym domu zamieszkuje zwykle jedna rodzina;
- jeden do wielu (*1:n*) – jednej encji z tabeli A przypisane jest wiele encji z tabeli B. Jest to chyba najczęściej spotykany typ relacji. Jednym z przykładów jest relacja, gdzie 1 klient ma wiele zamówień w sklepie. Każde zamówienie ma tylko jednego klienta;
- wiele do wielu (*n:m*) – wielu encjom z tabeli A przypisane jest wiele encji z tabeli B. Ten typ relacji zwykle implementowany jest z wykorzystaniem tabeli pośredniej, gdzie wykorzystane są klucze główne z tabel A i B jako klucze obce referujące do tabel A i B. Przykładem wykorzystania takiej relacji może być baza danych biblioteki z trzema tabelami: czytelnicy, książki i wypożyczenia. Każdy czytelnik może wypożyczyć n książek, a każda książka może być wypożyczona przez m czytelników. Tabela z wypożyczeniami w opisywanym przykładzie jest tabelą pośrednią.

76. Jakie znasz dobre praktyki bezpieczeństwa baz danych?

Do najbardziej podstawowych praktyk bezpieczeństwa związanych z bazami danych należą:

- nadawaj użytkownikom tylko te uprawnienia, które są konieczne;
- zadbaj o backupy bazy danych. Koniecznie sprawdzaj co jakiś czas czy kopie zapasowe działają poprawnie;

- sprawdź konfigurację bazy danych, usuń domyślnie tworzone elementy bazy (np. użytkownicy, tabele), rozważ zmianę domyślnego portu;
- nie korzystaj z konta root;
- ustaw silne hasła dla kont użytkowników;
- korzystaj z kont z uprawnieniami *read-only*, gdy tylko potrzebujesz coś sprawdzić w bazie danych. Zapobiegnie to przypadkowemu usunięciu danych. Brzmi niewiarygodnie, ale takie sytuacje się zdarzają;
- korzystaj z bezpiecznego połączenia SSL;
- korzystaj z szyfrowania;
- nie przechowuj haseł w postaci tekstowej ani zaszyfrowanej. Zamiast tego, w bazie danych zapisuj wyniki funkcji skrótu.

77. Czym są nierelacyjne bazy danych?

Istnieją systemy ze specyficznymi potrzebami i wymaganiami, gdzie relacyjne bazy danych się nie sprawdzają. W tym celu powstały liczne nierelacyjne bazy danych. Każda z baz służy określonym potrzebom i ma swoją specyfikę. Przykładowo, dokumentowa baza MongoDB będzie dobrym wyborem przy nieuporządkowanych strukturach danych. Z kolei Redis, będący reprezentantem baz typu klucz-wartość, dobrze sprawdzi się jako silnik cache. Wyróżnić można wiele typów, z których do najpopularniejszych należą bazy dokumentowe (MongoDB, Elasticsearch), bazy klucz-wartość (Redis, Amazon DynamoDB) czy bazy grafowe (Neo4J). Pełna lista nierelacyjnych baz danych jest znacznie większa.

78. Czym jest transakcja?

Transakcja to mechanizm, który oczekuje zestawu operacji (zapytań) do wykonania. Do zaistnienia transakcji konieczne jest pomyślne wykonanie wszystkich operacji. Po pomyślnym wykonaniu wszystkich operacji następuje ich zatwierdzenie (*commit*) W przypadku niepowodzenia, wszystkie operacje są odrzucane, a wykonane operacje są cofane (*rollback*).

79. Czym jest ACID?

ACID jest zbiorem zasad jakie powinny spełniać transakcje.

- *atomicity* — oznacza, że każda operacja na bazie danych jest traktowana jako osobny, niepodzielny byt. Dodatkowo albo wszystkie operacje zakończą się pomyślnie, albo żadna z nich nie zostanie zaaplikowana;
- *consistency* — każda transakcja zmienia stan bazy z jednego poprawnego stanu na inny stan, również poprawny;
- isolation — równolegle uruchomione transakcje nie powinny wpływać na siebie nawzajem;
- durability — zmiany wykonane w trakcie transakcji są permanentne. Oznacza to, że rezultaty transakcji są trwale zapisane w bazie danych niezależnie od dalszych awarii.

80. [ZADANIE PRAKTYCZNE] Dla poniższej tabeli przygotuj zapytanie, które zwróci 10 najnowszych rekordów (kolumna *created_at*). Zwróć tylko wartość *serial_number* jako S/N. Uwzględnij wyłącznie wiersze, dla których *is_valid* jest równe *true*.

id	firstname	lastname	serial_number	is_valid	created_at
1	Bob	Smith	1CC6jRxAEO	true	2022-01-02
...	...	...	...	...	...
12876	John	Snow	CObhq4iujM	false	2023-03-13

To zadanie sprawdza znajomość podstawowych operatorów SQL:

- *AS* – pozwala zwrócić nazwę kolumny pod zmienioną nazwą. Jest szczególnie przydatne, gdy chcemy zwrócić wyniki funkcji, np. operacji agregujących, pod przyjazną dla człowieka nazwą;
- *WHERE* – pozwala wybrać z tabeli tylko te rezultaty, gdzie spełniony został warunek po słowie kluczowym;

- *ORDER BY column_name DESC* – klauzula umożliwia sortowanie wartości po wybranej kolumnie malejąco. Aby posortować rosnąco, wystarczy zamienić *DESC* na *ASC*. W MySQL, jeśli sortujemy rosnąco, kierunek sortowania można pominąć (*ASC* jest domyślny);
- *LIMIT* – zdefiniowanie limitu pozwala na zwrócenie jedynie n pierwszych rezultatów. Szczególnie użyteczne, gdy liczba wyników jest ogromna.

```sql
SELECT
    serial_number AS "S/N"
FROM
    table_name
WHERE
    is_valid = true
ORDER BY
    created_at
DESC
LIMIT
    10
;
```

81. [ZADANIE PRAKTYCZNE] Dla podanej tabeli (*employees*) przygotuj zapytanie, które zwróci nazwę stanowiska oraz maksymalną pensję dla danego stanowiska.

id	firstname	lastname	position	salary
1	Bob	Smith	CEO	25000
...	...	...	...	...
154	John	Snow	Accountant	9500

W rozwiązaniu tego zadania przydatny będzie operator *GROUP BY*. Dzięki niemu możliwe jest pogrupowanie rezultatów względem wybranej kolumny oraz zwrócenie poprawnego rezultatu dla funkcji agregującej *MAX*. W podanym przykładzie grupowanie następuje względem kolumny position. Rozwiązanie zadania wygląda następująco:

```sql
SELECT
    position, MAX(salary) AS max_salary
FROM
    employees
GROUP BY
    position
;
```

82. [ZADANIE PRAKTYCZNE] Dla podanej tabeli (*employees*) zwróć rezultaty, gdzie pole *department* jest równe Warsaw, Amsterdam, Paris lub Berlin.

id	firstname	lastname	position	department
1	Bob	Smith	CEO	Warsaw
...	...	...	...	...
2504	John	Snow	Accountant	Washington

Do rozwiązania tego zadania można podejść na dwa sposoby. Najbardziej trywialny sposób to wykorzystanie operatora klauzuli *WHERE...OR*. Niestety takie rozwiązanie jest nieeleganckie i nie skaluje się zbyt dobrze. Takie podejście przy 20 czy 50 wariantach zakończyłoby się, w najlepszym przypadku, bólem głowy. Znacznie milej widzianym rozwiązaniem jest wykorzystanie operatora *IN.*

```sql
SELECT
    *
FROM
    employees
WHERE
    department IN("Warsaw","Amsterdam","Paris","Berlin")
;
```

Pozostałe pytania techniczne

W tym rozdziale zawarłem wszystkie pytania, których nie byłem w stanie przypisać do żadnej konkretnej kategorii. Dodałem tu również pytania o ogólną wiedzę programistyczną, niezależną od wykorzystywanego stosu technologicznego.

83. Czym są wzorce projektowe?

Wzorce projektowe są sprawdzonymi i powtarzalnymi rozwiązaniami dla powszechnie występujących problemów w wytwarzaniu oprogramowania. Wzorce służą zapewnieniu efektywnych i optymalnych rozwiązań dla konkretnych sytuacji i pozwalają na tworzenie bardziej elastycznego, skalowalnego i zrozumiałego kodu. Praca z wykorzystaniem wzorców ma również pozytywny wpływ na komunikację w zespole. Przykładowo, podczas analizy danego problemu, *fabryka* czy *budowniczy* powie zespołowi więcej niż skomplikowany opis rozwiązania, czy próby wynalezienia koła na nowo.

84. Opisz dowolny wzorzec projektowy.

Wybór jest naprawdę spory, ale rekomenduję opisanie wzorca, który znasz bardzo dobrze. Pytanie o wzorzec jest dobrym punktem wyjścia, aby zboczyć nieco z podejścia opartego na pytaniach rekrutacyjnych i rozpocząć ogólną dyskusję na temat dobrych praktyk w programowaniu. Podczas tej dyskusji rekruter prawdopodobnie będzie badał poziom Twojego zaawansowania i zrozumienia tematu. Stanowczo odradzam uczenia się wzorców na pamięć, ponieważ istnieje duża szansa, że twój rozmówca to zauważy.

85. Czym jest DRY?

DRY oznacza *Don't Repeat Yourself*, czyli „nie powtarzaj się” i jest jedną z dobrych praktyk programowania. Stosowanie się do DRY niesie ze sobą szereg zalet:

- łatwość utrzymania kodu – wprowadzanie zmian jest łatwiejsze, gdy trzeba to zrobić w jednym miejscu;
- łatwiejsza detekcja i poprawianie błędów;
- mniejsze ryzyko popełnienia błędu – jeżeli zmieniane jest kilka fragmentów kodu, istnieje większe ryzyko popełnienia błędu.

86. Czym jest SOLID?

SOLID opisuje pięć rekomendacji programowania obiektowego:

1. *Single Responsibility Principle* – zasada jednej odpowiedzialności. Klasa lub moduł powinny mieć tylko jedną odpowiedzialność. Można również powiedzieć, że klasa lub moduł powinny mieć jeden powód do zmiany.
2. *Open/Closed Principle* – zasada otwarte-zamknięte. Klasa powinna być otwarta na rozbudowę, lecz zamknięta na modyfikację.
3. *Liskov Substitution Principle* – zasada podstawienia Liskov. Jeżeli w danej funkcji będzie wykorzystany obiekt klasy potomnej, to wywołanie na nim metody pierwotnie zdefiniowanej w klasie bazowej powinno dać te same rezultaty.
4. *Interface Segregation Principle* – zasada segregacji interfejsów. Klient wykorzystujący dany interfejs nie powinien być zmuszony do obsłużenia metod, których nie używa.
5. *Dependency Inversion Principle* – zasada odwrócenia zależności. Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych. Co więcej, abstrakcje nie powinny być zależne od implementacji, tylko to implementacje powinny być zależne od abstrakcji.

Są to tylko teoretyczne wyjaśnienia tych zasad. Im bardziej praktycznie podejdziesz do tematu, tym lepiej. Podobnie jak w przypadku wzorców projektowych, zachęcam do praktycznego zrozumienia tych wytycznych, a odpowiedź na to pytanie będzie dla Ciebie naturalna.

87. Czym jest złożoność obliczeniowa?

Złożoność obliczeniowa określa, jak szybko rośnie lub maleje ilość zasobów obliczeniowych wymaganych do wykonania algorytmu w zależności od rozmiaru danych wejściowych. Złożoność pozwala na opisanie, jak efektywnie działa dany algorytm w zależności od wielkości problemu, na którym jest stosowany.

88. Czym jest wstrzyknięcie zależności (*Dependency Injection*)?

Wstrzyknięcie zależności to wzorzec projektowy stosowany w programowaniu, który polega na przekazaniu zależności obiektu do innego obiektu zamiast pozwalania mu na samodzielne ich tworzenie. Wykorzystanie *Dependency Injection* ma kilka zalet:

- umożliwia rozdzielenie logiki od tworzenia obiektów ułatwia zarządzanie zależnościami między komponentami w aplikacji;
- ułatwia testowanie dzięki zastąpieniu wstrzykiwanej zależności mockiem lub stubem;
- sprzyja zasadzie pojedynczej odpowiedzialności. Dodatkowa odpowiedzialność zostaje wydzielona do wstrzykiwanej zależności.

89. Czym są *Progressive Web Apps* (PWA)?

Progressive Web Apps (PWA) to aplikacje wykonane z wykorzystaniem technologii webowych, których założeniem jest zbliżenie doświadczeń z jej użytkowania do doświadczeń z tradycyjnych aplikacji mobilnych. PWA mogą działać na różnych platformach i umożliwiają m.in. jej instalację, pracę w trybie offline i w tle, a także integrację z urządzeniem i innymi aplikacjami.

90. Czym jest *REST API*?

REST API to interfejs programistyczny aplikacji zgodny z zasadami architektury REST:

- odseparowanie interfejsu użytkownika od operacji na serwerze;
- serwer nie przechowuje stanu o sesji użytkownika;
- REST zakłada wykorzystanie *cache*. Odpowiedź, którą użytkownik otrzyma z REST API, musi jasno definiować czy ma ona być *cacheable*, czy *non-cacheable*;
- endpointy, czyli adresy zasobów, powinny jednoznacznie wskazywać, do jakiego zasobu się odwołują. Struktura REST API musi być niezależna od schematu bazy danych wykorzystywanej aplikacji;
- separacja warstw. Oznacza, że należy oddzielić warstwy dostępu do danych, logiki biznesowej oraz prezentacji;
- możliwość udostępniania skryptów wykonywalnych użytkownikom (opcjonalnie).

REST API najczęściej opiera się o protokół HTTP i operuje na formacie danych JSON.

91. Jaka jest różnica między klasą a obiektem?

Klasę może zdefiniować schemat obiektu do inicjalizacji, natomiast obiekt jest to powołany do życia byt na podstawie stworzonego wcześniej schematu (klasy).

Przykładowo tworząc klasę *Car* opisującą samochód, będzie miał on cechy np. kolor, marka, rodzaj paliwa oraz metody np. jedź, hamuj, skręcaj. Tworząc obiekt na podstawie klasy, definiuje się już konkretne właściwości, jakie ma mieć ten samochód. W przypadku *Car* może być to na przykład czerwone Ferrari z silnikiem benzynowym.

92. Jakie znasz struktury danych?

Do najbardziej podstawowych struktur danych należą: stos, kolejka, lista, drzewo binarne i graf.

Samych struktura danych jest oczywiście więcej. Tak jak w przypadku wzorców, praktyka zdecydowanie ponad teorię. Znanie definicji stosu nic Ci nie da, jeśli nie będziesz w stanie go wykorzystać w praktyce.

93. Podaj sposoby, jakimi optymalizował(a)byś wydajność strony WWW.

Poniżej przedstawiam kilka propozycji odpowiedzi na to pytanie:

- kompresja zdjęć, plików audio i video zamieszczanych na stronie;
- lazy loading dla zdjęć i video;
- korzystanie z CDN-ów;
- optymalizacja na poziomie kodu – usunięcie zbędnego i nieużywanego kodu, refaktoryzacja i optymalizacja używanego kodu;
- minifikacja i obfuskacja kodu;
- włączenie kompresji Gzip/Brotli;
- usunięcie blokującego JavaScript;
- umiejętne cache-owanie zasobów;
- redukcja liczby zapytań HTTP.

94. Jakie znasz rodzaje testów automatycznych?

Po raz kolejny zachęcam do odwołania się do praktyki, a nie teoretycznego wykucia przeznaczenia i rodzajów testów. Możesz dostać zadanie opisania przykładowego testu, co przy jedynie teoretycznej znajomości pojęcia może być trudne. Najprostszą odpowiedzią na to pytanie będzie np.:

- testy jednostkowe testy dla unitu, czyli małego wydzielonego fragmentu kodu np. funkcji czy metody;
- integracyjne, które sprawdzają współpracę między różnymi jednostkami kodu;
- end-to-end symulujące rzeczywiste interakcje użytkownika z aplikacją.

Warto również w swojej odpowiedzi wspomnieć o piramidzie testów. Można również rozwinąć swoją wypowiedź i wspomnieć o testach funkcjonalnych, akceptacyjnych, wydajności, obciążenia, bezpieczeństwa, GUI czy regresji.

95. Czym jest Garbage Collector?

Garbage Collector to mechanizm w językach programowania, który automatycznie zarządza pamięcią, usuwając nieużywane lub nieosiągalne obiekty z pamięci komputera. Obiekt można uznać za nieosiągalny, gdy nie istnieje do niego żadna referencja w aplikacji. Działa on w tle, monitorując i śledząc obiekty alokowane w pamięci.

96. Jaka jest różnica między szyfrowaniem a haszowaniem?

Najważniejszą różnicą między szyfrowaniem a haszowaniem jest to, że szyfrowanie jest procesem odwracalnym a haszowanie nie. W procesie szyfrowania otrzymuje się szyfrogram, a rezultatem haszowania jest wynik funkcji skrótu.

97. Co to jest JWT?

JWT (*JSON Web Token*) to otwarty standard dla tokenów, który jest używany do przesyłania informacji między stronami w formacie JSON. JWT składa się z trzech części oddzielonych kropkami: nagłówka (*header*), dane (*payload*) i podpis (*signature*).

Nagłówek zawiera informacje o typie tokena i używanym algorytmie do generowania podpisu. W danych zamieszczana jest faktyczna informacja. Podpis pozwala na weryfikację autentyczności tokena.

98. Opisz w skrócie jakie znasz rodzaje ataków/podatności jakie mogą wystąpić w aplikacjach.

Z podatności, jakie mogą wystąpić w aplikacjach webowych, warto wspomnieć o *SQL Injection*, *Cross-Site Scripting (XSS)*, *Cross-Site Request Forgery (CSRF)*, wykorzystywaniu podatnych komponentów czy błędach konfiguracji.

Warto znać ogólną istotę tych podatności, sposoby wykorzystania, konsekwencje oraz sposoby na zabezpieczanie się przed nimi. O ile nie aplikujesz na stanowisko związane z bezpieczeństwem, raczej nikt nie będzie wymagał zbytniego zagłębiania się w szczegóły. W zależności od rodzaju firmy, do jakiej aplikujesz, warto znać ataki specyficzne dla danej branży. Przykładowo, w firmach wykorzystujących technologię *blockchain* lub AI warto wspomnieć o podatnościach specyficznych dla tych technologii.

99. Czy wiesz, czym są serwisy ciągłej integracji? Jakie widzisz korzyści z ich zastosowania?

Serwisy ciągłej integracji (*Continuous Integration*) to praktyka wytwarzania oprogramowania, w której kod jest regularnie integrowany w centralnym repozytorium, a następnie automatycznie budowany i testowany. Pozwala to na szybsze wykrywanie błędów i polepszenie jakości wdrażanego oprogramowania.

Pojęcie ciągłej integracji jest nieco bardziej zaawansowane, więc z pewnością zaplusujesz, jeśli masz jakieś doświadczenie z CI.

Pytania nietechniczne

Na koniec zostawiłem kilka pytań nietechnicznych, na które szczególnie warto przemyśleć odpowiedź. Odpowiedź na nie może być niekiedy nawet ważniejsza niż odpowiedzi na pytania dotyczące zagadnień technicznych. Część z tych pytań możesz usłyszeć, zanim jeszcze dojdzie do technicznej części rozmowy. Pytania są na tyle specyficzne, a odpowiedzi na tyle indywidualne, że podawanie przykładowych odpowiedzi mija się z celem.

100. Jaki ciekawy materiał (np. artykuł/książka/reportaż/film/vlog) związany z branżą IT ostatnio czytałeś(aś)/oglądałeś(aś)? Czego się tam dowiedziałeś(aś)?

Celem pytania jest dowiedzenie się, jak bardzo interesuje Cię branża IT i czy jesteś na bieżąco z nowymi trendami. Dzięki temu pytaniu rozmówca pozna też obszary Twoich zainteresowań.

101. Opowiedz coś więcej o wybranym projekcie, w którym brałeś(aś) udział.

Pytanie ma na celu zweryfikować Twoje doświadczenie praktyczne. Nie musi to być projekt komercyjny. Możesz opowiedzieć np. o jakimś projekcie na potrzeby rozwoju czy projekcie ze studiów. Warto opowiedzieć nie tylko o założeniach projektu, ale też o wyzwaniach, jakie przy nim napotkałeś(aś), wykorzystanych technologiach oraz czego się dzięki niemu nauczyłeś(aś).

102. Jak widzisz swój kierunek rozwoju?

Tutaj warto być szczerym. Pomoże to nie tylko Tobie, ale też twojemu potencjalnemu przyszłemu pracodawcy. Jeśli Twoja wizja rozwoju nie odpowiada profilowi kandydata, którego szuka twój współrozmówca, to lepiej by to wyszło przed podpisaniem umowy.

103. Czego oczekujesz od pracodawcy?

Podobnie jak w poprzednim pytaniu, postaw na szczerość. Przedstawienie oczekiwań na etapie rozmowy pozwala na uniknięcie nieporozumień, gdyby pracodawca nie był w stanie ich spełnić.

104. Co wiesz o naszej firmie? Dlaczego wysłałeś(aś) aplikację akurat do nas?

Tym pytaniem pracodawca może sprawdzić, jak bardzo zależy Ci na pracy w danym miejscu. Pytanie to pozwala sprawdzić, czy kandydat przemyślał wysłanie swojej aplikacji, czy też wysyłał swoje CV na wszystkie możliwe ogłoszenia.

105. Tell me something about your hobbies.

Jeżeli w ogłoszeniu o pracę widnieje wymóg znajomości języka angielskiego, najprawdopodobniej część rozmowy będzie prowadzona w języku angielskim. W rekrutacjach, w których brałem udział, zwykle była to luźna rozmowa o zainteresowaniach czy prośba o opowiedzenie o sobie. Samo pytanie o hobby również jest dobrym sposobem na sprawdzenie, czy kandydat wpasuje się w zespół.

106. Jakie TY masz pytania?

Jedno z najważniejszych pytań, jakie możesz usłyszeć i które na 99% padnie. Koniecznie zastanów się, czego chciał(a)byś się dowiedzieć jeszcze przed samą rozmową. Nie tylko rozwiejesz ewentualne wątpliwości, ale niektórymi pytaniami można BARDZO zaplusować u rekrutera :)

Na zakończenie

Na koniec mam dla Ciebie kilka dodatkowych rad:

- odpocznij przed rozmową. Na rozmowę przyjdź wyspany i wypoczęty;
- nie bój się odpowiedzi „nie wiem”. Zdecydowanie lepiej jest przyznać się do niewiedzy niż kłamać. Kłamanie na rozmowie może przekreślić Twoją kandydaturę, a w niektórych przypadkach wręcz zamknąć ścieżkę kariery w danej firmie;
- jeśli nie znałeś/aś odpowiedzi na zadane pytanie, nie bój się poprosić o odpowiedź. Rekruterzy bardzo chętnie objaśniają zagadnienia, o które pytają;
- wyciągaj lekcje z przeprowadzonych rozmów. Nawet jeśli nie otrzymasz propozycji pracy, to każda kolejna odbyta rozmowa jest okazją do zebrania cennej wiedzy.

Pozostaje mi życzyć Ci powodzenia na rozmowie! Gratuluję Ci też dotarcia do końca tego e-booka. Wiem, że analiza ponad setki pytań nie jest prostą sprawą.

Mam nadzieję, że przygotowana przeze mnie lista pytań będzie dla Ciebie przydatna. W ramach podziękowania za ten e-book będę wdzięczny, jeśli dasz o nim znać swoim znajomym i zachęcisz ich do zapisu na mój mailing :)

Możesz mi również postawić kawę w serwisie buycoffee klikając w przycisk poniżej lub w ten link.

Postaw kawę