



USŁUGI SIECIOWE REST API W TECHNOLOGII SPRING

MARIUSZ PASTUSZKA

REST - A CO TO?

- Styl architektoniczny REST opisuje sześć ograniczeń. Ograniczenia te, zastosowane do architektury, zostały pierwotnie przedstawione przez Roya Fieldinga w jego pracy doktorskiej (patrz https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm) i definiując podstawy stylu REST-ful.

CECHY WZORCA

- Ujednolicony interfejs - Uniform Interface
- Bezstanowość - Stateless
- Możliwość buforowania - Cacheable
- Klient-Serwer - Client-Server
- System warstwowy - Layered System
- Kod na żądanie (opcjonalnie) - Code on Demand

OMÓWIENIE CECH – KOD NA ŻĄDANIE

- Serwery są w stanie tymczasowo rozszerzyć lub dostosować funkcjonalność klienta, przesyłając do niego logikę, którą może on wykonać. Przykładem tego mogą być skompilowane komponenty, takie jak aplety Java i skrypty po stronie klienta, takie jak JavaScript.
- Jedyne opcjonalne wymaganie

OMÓWIENIE CECH – SYSTEM WARSTWOWY

- Klient nie może zwykle stwierdzić, czy jest połączony bezpośrednio z serwerem końcowym, czy z pośrednikiem po drodze.
- Serwery pośredniczące mogą poprawić skalowalność systemu, umożliwiając równoważenie obciążenia i udostępniając współdzielone pamięci podręczne.
- Warstwy mogą również egzekwować zasady bezpieczeństwa.

OMÓWIENIE CECH – Klient - SERWER

- Jednolity interfejs oddziela klientów od serwera.
- Klienci nie są zainteresowani przechowywaniem danych, które pozostaje wewnętrzne dla każdego serwera, dzięki czemu przenośność kodu klienta jest lepsza.
- Serwery nie są związane z interfejsem użytkownika lub stanem użytkownika, dzięki czemu serwery mogą być prostsze i bardziej skalowalne.
- Serwery i klienci mogą być również wymieniane i rozwijane niezależnie, o ile interfejs nie jest zmieniany.

OMÓWIENIE CECH – MOŻLIWOŚĆ BUFOROWANIA

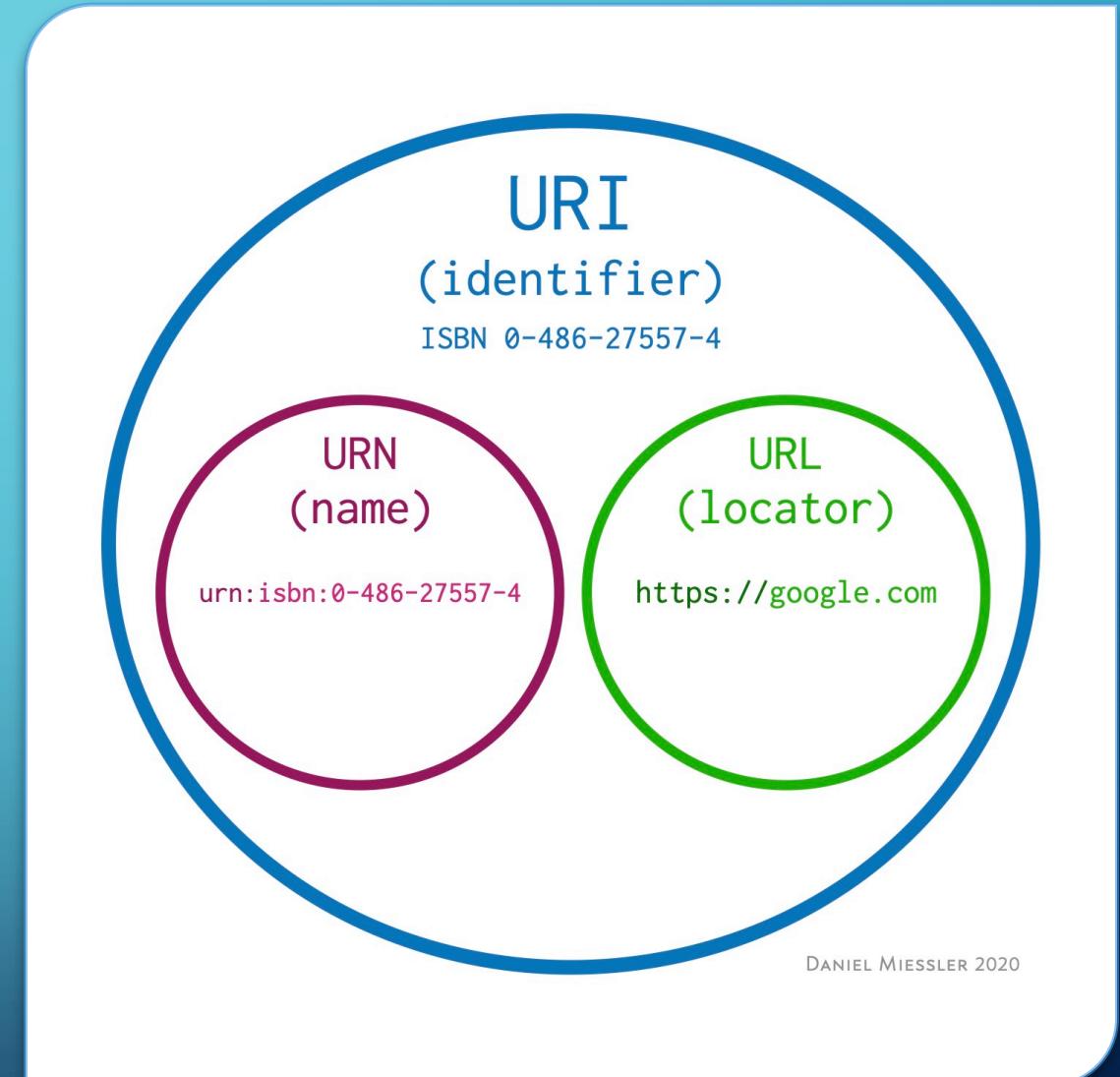
- Podobnie jak w sieci World Wide Web, klienci mogą buforować odpowiedzi.
- Odpowiedzi muszą zatem, niejawnie lub jawnie, określać się jako buforowalne lub nie, aby zapobiec ponownemu wykorzystaniu przez klientów nieaktualnych lub nieodpowiednich danych w odpowiedzi na kolejne żądania.
- Dobrze zarządzane buforowanie częściowo lub całkowicie eliminuje niektóre interakcje klient-serwer, dodatkowo poprawiając skalowalność i wydajność.

OMÓWIENIE CECH – BEZSTANOWOŚĆ

- Jako że REST jest akronimem od REpresentational State Transfer, bezpieczeństwość jest kluczem.
- Stan niezbędny do obsługi żądania jest zawarty w samym żądaniu, czy to jako część URI, parametrów ciągu zapytania, ciała, czy nagłówków.
- URI jednoznacznie identyfikuje zasób, a ciało zawiera stan (lub zmianę stanu) tego zasobu.
- Następnie, po tym jak serwer wykona swoje przetwarzanie, odpowiedni stan, lub fragment(y) stanu, które mają znaczenie, są przekazywane z powrotem do klienta poprzez nagłówki, status i treść odpowiedzi.

URI VS URL VS URN

[HTTPS://DANIELMIESSLER.COM/STUDY/DIFFERENCE-BETWEEN-URI-URL/](https://danielmiessler.com/study/difference-between-uri-url/)



OMÓWIENIE CECH – BEZSTANOWOŚĆ CD.

- Większość z nas, którzy są w branży od jakiegoś czasu, jest przyzwyczajona do programowania w kontenerze, który zapewnia nam koncepcję "sesji", która utrzymuje stan przez wiele żądań HTTP.
- W REST klient musi zawrzeć wszystkie informacje, aby serwer mógł spełnić żądanie, przesyłając ponownie stan w razie potrzeby, jeśli ten stan musi obejmować wiele żądań.
- Bezpaństwość umożliwia większą skalowalność, ponieważ serwer nie musi utrzymywać, aktualizować ani przekazywać stanu sesji.
- Dodatkowo, load balancery nie muszą się martwić o powiązania sesji dla systemów bezpaństwowych.

OMÓWIENIE CECH – BEZSTANOWOŚĆ CD.

- Jaka jest więc różnica pomiędzy stanem a zasobem?
- Stan aplikacji, to to, o co dba serwer, aby zrealizować żądanie - dane niezbędne do bieżcej sesji lub żądania.
- Zasób, lub stan zasobu, to dane, które definiują reprezentację zasobu - dane przechowywane na przykład w bazie danych.
- Rozważmy stan aplikacji jako dane, które mogą się zmieniać w zależności od klienta i żądania. Z drugiej strony, stan zasobu jest stały dla każdego klienta, który go zażąda.

OMÓWIENIE CECH – BEZSTANOWOŚĆ CD.

- Należy dołożyć wszelkich starań, aby stan aplikacji nie obejmował wielu żądań Twojej usługi (usług).

OMÓWIENIE CECH – UJEDNOLICONY INTERFEJS

- Ograniczenie jednolitego interfejsu definiuje interfejs pomiędzy klientami i serwerami.
- Upraszczza i rozdziela architekturę, co pozwala każdej części rozwijać się niezależnie.

OMÓWIENIE CECH – UJEDNOLICONY INTERFEJS CD.

- Przetwarzanie w oparciu o zasoby
 - Poszczególne zasoby są identyfikowane w żądaniach przy użyciu URI jako identyfikatorów zasobów.
 - Same zasoby są koncepcyjnie oddzielone od reprezentacji, które są zwracane do klienta.
 - Serwer nie wysyła swojej bazy danych, ale raczej pewien HTML, XML lub JSON, który reprezentuje niektóre rekordy bazy danych wyrażone na przykład w języku polskim i zakodowane w UTF-8, w zależności od szczegółów żądania i implementacji serwera.

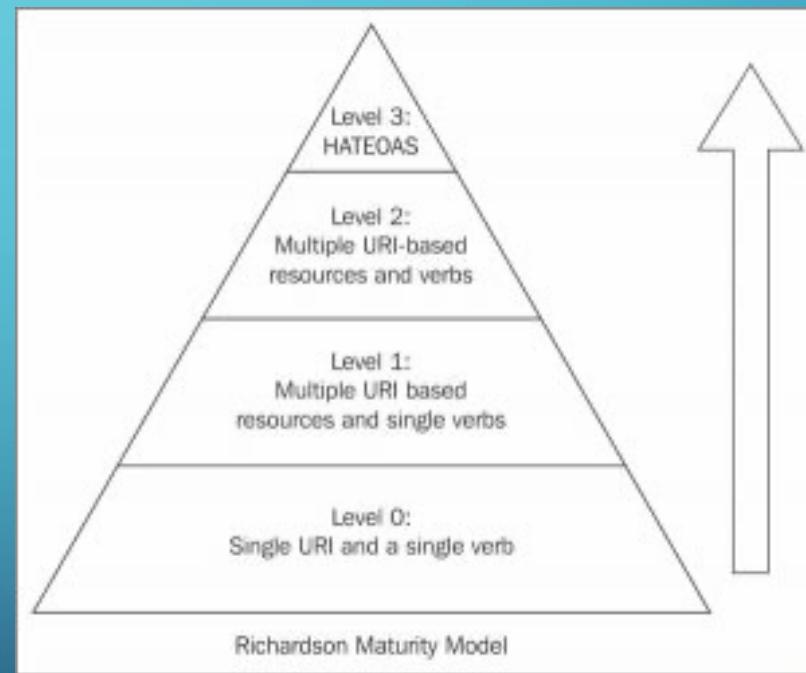
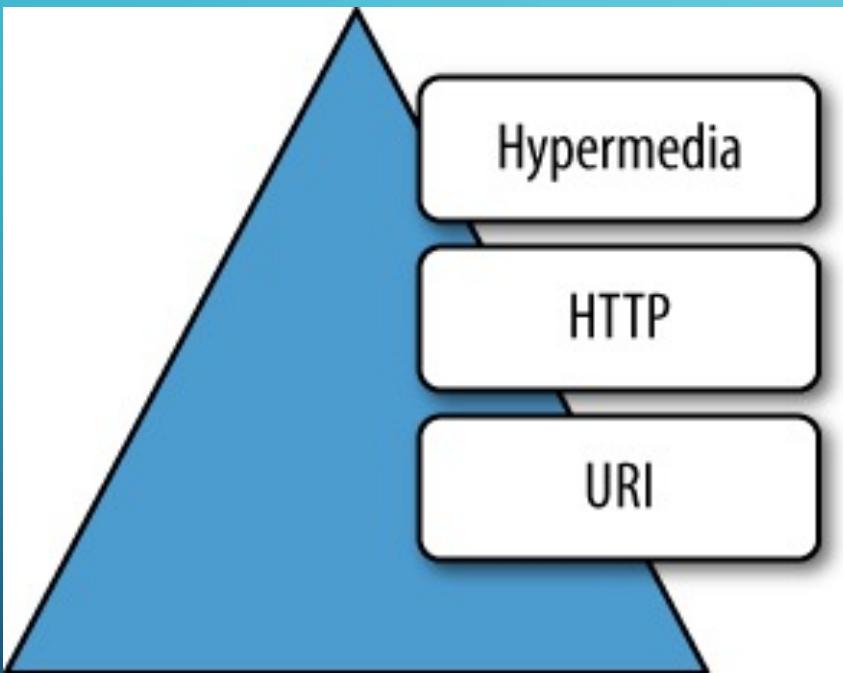
OMÓWIENIE CECH – UJEDNOLICONY INTERFEJS – CD.

- Samoopisowe komunikaty
 - Każdy komunikat zawiera wystarczającą ilość informacji, aby opisać, jak go przetworzyć.
 - Na przykład, który parser należy wywołać może być określony przez Internet Media Type (poprzednio znany jako MIME Type).
 - Odpowiedzi również jednoznacznie wskazują na możliwość ich buforowania.

OMÓWIENIE CECH – UJEDNOLICONY INTERFEJS – CD.

- Hipermedia jako silnik stanu aplikacji (HATEOAS)
 - Klienci dostarczają stan za pomocą zawartości ciała, parametrów ciągu zapytania, nagłówków żądania i żądanego URI (nazwa zasobu). Usługi dostarczają stan do klientów poprzez zawartość ciała, kody odpowiedzi i nagłówki odpowiedzi.
 - HATEOS oznacza również, że tam gdzie jest to konieczne, linki są zawarte w zwracanym ciele (lub nagłówkach), aby dostarczyć URI do pobrania samego obiektu lub obiektów powiązanych.
 - Jednolity interfejs, który musi zapewnić każda usługa REST jest fundamentalny dla jej projektu.

MODEL DOJRZAŁOŚCI RICHARDSONA



MODEL DOJRZAŁOŚCI RICHARDSONA

- **Poziom zero**

- Poziom zerowy dojrzałości nie wykorzystuje żadnych możliwości URI, metod HTTP i HATEOAS.
- Usługi te mają pojedynczy URI i używają pojedynczej metody HTTP (zazwyczaj POST). Na przykład, większość usług opartych na Web Services (WS-*) używa pojedynczego URI do identyfikacji punktu końcowego oraz HTTP POST do przesyłania zawartości opartej na SOAP, efektywnie ignorując pozostałe czasowniki HTTP.

MODEL DOJRZAŁOŚCI RICHARDSONA

- Poziom pierwszy

- Pierwszy poziom dojrzałości wykorzystuje URI
- Usługi te wykorzystują wiele URI, ale tylko jeden verb HTTP - zazwyczaj HTTP POST. Nadają one każdemu zasobowi w swoim wszechświecie URI. Unikalny URI oddziennie identyfikuje jeden unikalny zasób - i to czyni je lepszymi niż poziom zero.

MODEL DOJRZAŁOŚCI RICHARDSONA

- Poziom drugi

- Drugi poziom dojrzałości wykorzystuje URI i metody HTTP
- Usługi poziomu drugiego hostują liczne zasoby, do których można zaadresować URI. Takie usługi obsługują kilka z czasowników HTTP na każdym eksponowanym zasobie - są to usługi typu Create, Read, Update i Delete (CRUD). W tym przypadku stan zasobów, zwykle reprezentujących podmioty biznesowe, może być manipulowany przez sieć.
- Poziom 2 to doskonały przypadek użycia zasad REST.

MODEL DOJRZAŁOŚCI RICHARDSONA

- **Poziom trzeci**

- Trzeci poziom dojrzałości wykorzystuje wszystkie trzy, tj. URI oraz HTTP i HATEOAS.
- Ten poziom jest najbardziej dojrzałym poziomem modelu Richardsona, który zachęca do łatwej wykrywalności.
- Na tym poziomie odpowiedzi stają się łatwe do zrozumienia dzięki zastosowaniu HATEOAS.
- Usługa prowadzi konsumentów przez ścieżkę zasobów, powodując w rezultacie przejście stanu aplikacji.

KRÓTKIE WSKAZÓWKI DOTYCZĄCE REST API

- Użyj czasowników HTTP, aby twoje żądania coś znaczyły
- GET - Odczytaj określony zasób (poprzez identyfikator) lub kolekcję zasobów.
- DELETE - Usuń/usuń określony zasób według identyfikatora.
- POST - Tworzy nowy zasób. Również uniwersalny czasownik dla operacji, które nie pasują do innych kategorii.
 - PUT – Tworzy nowy zasób – całkowita zamiana.
 - PATCH – Update zasobu – częściowy.

KRÓTKIE WSKAZÓWKI DOTYCZĄCE REST API

- Zapewnij rozsądne nazwy zasobów
 - Stworzenie świetnego API to w 80% sztuka, a w 20% nauka. Tworzenie hierarchii adresów URL reprezentujących sensowne zasoby jest częścią sztuki. Posiadanie sensownych nazw zasobów (które są po prostu ścieżkami URL, takimi jak `/customers/12345/orders`) poprawia przejrzystość tego, co dane żądanie robi.
 - Odpowiednie nazwy zasobów zapewniają kontekst dla żądania usługi, zwiększając zrozumiałość interfejsu API. Zasoby są postrzegane hierarchicznie poprzez ich nazwy URI, oferując konsumentom przyjazną, łatwą do zrozumienia hierarchię zasobów, którą mogą wykorzystać w swoich aplikacjach.

KRÓTKIE WSKAZÓWKI DOTYCZĄCE REST API

- Używaj identyfikatorów w adresach URL zamiast w query-string. Używanie parametrów URL query-string jest świetne do filtrowania, ale nie do nazw zasobów.
 - Dobrze: /users/12345
 - Słabo: /api?type=user&id=23
- Wykorzystaj hierarchiczną naturę adresu URL, aby zasugerować strukturę.
- Projektuj dla swoich klientów, nie dla swoich danych.
- Nazwy zasobów powinny być rzeczownikami. Unikaj czasowników jako nazw zasobów, aby poprawić przejrzystość. Użyj metod HTTP do określenia części żądania zawierającej czasowniki.

KRÓTKIE WSKAZÓWKI DOTYCZĄCE REST API

- Użyj liczby mnogiej w segmentach URL, aby utrzymać URI API spójne we wszystkich metodach HTTP, używając metafory kolekcji.
 - Zalecane: `/customers/33245/orders/8769/lineitems/1`
 - Niezalecane: `/customer/33245/order/8769/lineitem/1`
- Unikaj używania w adresach URL sformułowań dotyczących kolekcji. Na przykład '`customer_list`' jako zasób. Używaj pluralizacji, aby wskazać metaforę kolekcji (np. `customers` vs `customer_list`).

KRÓTKIE WSKAZÓWKI DOTYCZĄCE REST API

- W segmentach adresu URL używaj małych liter, oddzielając słowa podkreśnikami ("_") lub myślnikami ("-"). Niektóre serwery ignorują wielkość liter, więc najlepiej jest zachować jasność.
- Zachowaj adresy URL tak krótkie, jak to możliwe, z tak niewielu segmentów, jak to ma sens.

KRÓTKIE WSKAZÓWKI DOTYCZĄCE REST API

- Użyj kodów odpowiedzi HTTP, aby określić status
- 200 OK - Ogólny kod statusu sukcesu. Jest to najczęściej występujący kod. Używany do wskazania sukcesu.
- 201 CREATED - Udało się utworzyć zasób (poprzez POST lub PUT). Ustaw nagłówek Location tak, aby zawierał link do nowo utworzonego zasobu (w przypadku POST). Treść odpowiedzi może, ale nie musi być obecna.
- 204 NO CONTENT - Wskazuje na sukces, ale nic nie ma w treści odpowiedzi, często używane dla operacji DELETE i PUT.
- 400 BAD REQUEST - Ogólny błąd dla sytuacji, gdy spełnienie żądania spowodowałoby nieprawidłowy stan. Błędy walidacji domeny, brakujące dane, itp. to tylko niektóre przykłady.

KRÓTKIE WSKAZÓWKI DOTYCZĄCE REST API

- 401 UNAUTHORIZED - Kod błędu odpowiedzi dla brakującego lub nieprawidłowego tokena uwierzytelniającego.
- 403 FORBIDDEN - Kod błędu dla sytuacji, gdy użytkownik nie jest upoważniony do wykonania operacji lub zasób jest niedostępny z jakiegoś powodu (np. ograniczenia czasowe, itp.).
- 404 NOT FOUND - Używane, gdy żądany zasób nie został znaleziony, niezależnie od tego, czy nie istnieje, czy też wystąpił błąd 401 lub 403, który ze względów bezpieczeństwa serwis chce zamaskować.
- 405 METHOD NOT ALLOWED - Używane do wskazania, że żądany adres URL istnieje, ale żądana metoda HTTP nie ma zastosowania.

KRÓTKIE WSKAZÓWKI DOTYCZĄCE REST API

- 409 CONFLICT - Zawsze, gdy spełnienie żądania spowodowałoby konflikt zasobów. Duplikaty wpisów, takie jak próba utworzenia dwóch klientów z tymi samymi informacjami, oraz usuwanie obiektów głównych, gdy nie jest obsługiwane usuwanie kaskadowe, to tylko kilka przykładów.
- 500 INTERNAL SERVER ERROR - Nigdy nie zwracaj tego celowo. Ogólny błąd catch-all, gdy strona serwera rzuca wyjątek. Używaj go tylko w przypadku błędów, których konsument nie może rozwiązać po swojej stronie.

KRÓTKIE WSKAZÓWKI DOTYCZĄCE REST API

- **Oferuj zarówno JSON jak i XML**

- **Faworyzuj obsługę JSON, chyba że działasz w wysoce znormalizowanej i regulowanej branży, która wymaga XML, weryfikacji schematów i przestrzeni nazw, i oferuj zarówno JSON, jak i XML, chyba że koszty są zbyt wysokie. Idealnie, pozwól konsumentom przełączać się między formatami za pomocą nagłówka HTTP Accept.**

KRÓTKIE WSKAZÓWKI DOTYCZĄCE REST API

- **Twórz zasoby drobnoziarniste**

- **Na początku najlepiej jest tworzyć interfejsy API, które naśladują podstawową domenę aplikacji lub architekturę bazy danych systemu.**
- **Ostatecznie, będziesz potrzebował zagregowanych usług, które wykorzystują wiele zasobów bazowych, aby zmniejszyć częstotliwość komunikacji.**
- **Jednak o wiele łatwiej jest tworzyć większe zasoby później z pojedynczych zasobów, niż tworzyć drobnoziarniste lub indywidualne zasoby z większych agregatów.**
- **Ułatw to sobie i zaczni od małych, łatwo definiowalnych zasobów, zapewniając funkcjonalność CRUD na tych zasobach.**

KRÓTKIE WSKAZÓWKI DOTYCZĄCE REST API

- **Weź pod uwagę powiązania**
 - Jedną z zasad REST jest łączność - poprzez hipermedialne linki (szukaj HATEOAS).
 - Chociaż usługi są nadal użyteczne bez nich, interfejsy API stają się bardziej samoopisowe i łatwiejsze do odkrycia, gdy linki są zwracane w odpowiedzi. Przynajmniej odnośnik "self" informuje klientów, w jaki sposób dane zostały lub mogą zostać pobrane.
 - Dodatkowo, wykorzystaj nagłówek HTTP Location do umieszczenia linku przy tworzeniu zasobu poprzez POST (lub PUT).
 - Dla kolekcji zwracanych w odpowiedzi, które obsługują paginację, bardzo pomocne są linki 'first', 'last', 'next' i 'prev'.

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL

| HTTP Verb | CRUD | Cała kolekcja (np. /customers) | Określony element (np. /customers/{id}) |
|-----------|----------------|---|--|
| POST | Create | 201 (Created), nagłówek 'Location' z linkiem do /customers/{id} zawierający nowe ID. | 404 (Not Found), 409 (Conflict) jeśli zasób już istnieje... |
| GET | Read | 200 (OK), lista klientów. Użyj paginacji, sortowania i filtrowania, aby poruszać się po dużych listach. | 200 (OK), pojedynczy klient. 404 (Not Found), jeśli identyfikator nie został znaleziony lub jest nieprawidłowy. |
| PUT | Update/Replace | 405 (Method Not Allowed), chyba że chcesz zaktualizować/zastąpić każdy zasób w całej kolekcji. | 200 (OK) lub 204 (brak treści). 404 (Not Found), jeśli identyfikator nie został znaleziony lub jest nieprawidłowy. |

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL

| HTTP Verb | CRUD | Cała kolekcja (np. /customers) | Określony element (np. /customers/{id}) |
|-----------|---------------|--|--|
| PATCH | Update/Modify | 405 (Method Not Allowed), chyba że chcesz zmodyfikować samą kolekcję. | 200 (OK) lub 204 (brak treści). 404 (Not Found), jeśli identyfikator nie został znaleziony lub jest nieprawidłowy. |
| DELETE | Delete | 405 (Metoda niedozwolona), chyba że chcesz usunąć całą kolekcję - co nie jest często pożądane. | 200 (OK)/204 (No content). 404 (Not Found), if ID not found or invalid. |

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - POST

- Czasownik POST jest najczęściej wykorzystywany do **tworzenia** nowych zasobów. W szczególności, jest on używany do tworzenia zasobów podrzędnych. To znaczy, podrzędnych w stosunku do jakiegoś innego (np. nadrzędnego) zasobu. Innymi słowy, tworząc nowy zasób, POST do rodzica, a usługa zajmie się powiązaniem nowego zasobu z rodzicem, nadaniem mu ID (URI nowego zasobu), itp.
- Przy udanym tworzeniu zwróć status HTTP 201, zwracając nagłówek Location z linkiem do nowo utworzonego zasobu ze statusem HTTP 201.

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - POST

- POST nie jest ani bezpieczny, ani idempotentny. Dlatego jest zalecany dla nie-idempotentnych żądań zasobów. Wykonanie dwóch identycznych żądań POST najprawdopodobniej spowoduje powstanie dwóch zasobów zawierających te same informacje.
- Przykłady:
 - *POST <http://www.example.com/customers>*
 - *POST <http://www.example.com/customers/12345/orders>*

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - GET

- Metoda HTTP GET służy do ****odczytywania**** (lub pobierania) reprezentacji zasobu. W "pozytywnej" ścieżce, GET zwraca reprezentację w XML lub JSON i kod odpowiedzi HTTP 200 (OK). W przypadku błędu, najczęściej zwraca 404 (NOT FOUND) lub 400 (BAD REQUEST).
- Zgodnie z założeniami specyfikacji HTTP, żądania GET (wraz z HEAD) służą **jedynie** do odczytu danych, a nie ich zmiany. Dlatego, gdy są używane w ten sposób, są uważane za bezpieczne. Oznacza to, że mogą być wywoływanie bez ryzyka modyfikacji lub uszkodzenia danych - wywołanie ich raz ma taki sam efekt jak wywołanie ich 10 razy, lub wcale. Dodatkowo, GET (i HEAD) jest idempotentny, co oznacza, że wykonanie wielu identycznych żądań daje taki sam rezultat jak pojedyncze żądanie.

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - GET

- Nie należy eksponować niebezpiecznych operacji poprzez GET - nie powinny one nigdy modyfikować żadnych zasobów na serwerze.
- Przykłady
 - *GET <http://www.example.com/customers/12345>*
 - *GET <http://www.example.com/customers/12345/orders>*
 - *GET <http://www.example.com/buckets/sample>*

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - PUT

- PUT jest najczęściej wykorzystywany do **aktualizacji** możliwości, PUT-owanie do znanego URI zasobu z ciałem żądania zawierającym nowo zaktualizowaną reprezentację oryginalnego zasobu.
- Jednakże, PUT może być również użyty do tworzenia zasobów w przypadku, gdy identyfikator zasobu jest wybierany przez klienta, a nie przez serwer. Innymi słowy, jeśli PUT jest wysyłany do URI, który zawiera wartość nieistniejącego identyfikatora zasobu. Ponownie, ciało żądania zawiera reprezentację zasobu. Zdaniem wielu jest to zagmatwane i mylące. W związku z tym, ta metoda tworzenia powinna być używana oszczędnie, lub wcale.

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - PUT

- Przy udanej aktualizacji zwróć 200 (lub 204 jeśli nie zwrócisz żadnej zawartości w ciele) z PUT. Jeśli używasz PUT do tworzenia, zwróć status HTTP 201 przy udanym tworzeniu. Ciało w odpowiedzi jest opcjonalne - dostarczenie go zużywa więcej transferu. Nie jest konieczne zwracanie linku poprzez nagłówek Location w przypadku tworzenia, ponieważ klient już ustawił ID zasobu.
- PUT nie jest bezpieczną operacją, ponieważ modyfikuje (lub tworzy) stan na serwerze, ale jest idempotentny. Innymi słowy, jeśli utworzysz lub zaktualizujesz zasób używając PUT, a następnie wykonasz to samo wywołanie ponownie, zasób nadal tam jest i nadal ma ten sam stan, jak przy pierwszym wywołaniu.

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - PUT

- Jeśli, na przykład, wywołanie PUT na zasobie powoduje inkrementację licznika wewnętrz zasobu, wywołanie nie jest już idempotentne. Czasami tak się zdarza i może wystarczyć udokumentowanie, że wywołanie nie jest idempotentne. Jednakże, zaleca się, aby żądania PUT były idempotentne.
- Przykłady
 - `PUT http://www.example.com/customers/12345`
 - `PUT http://www.example.com/customers/12345/orders/98765`
 - `PUT http://www.example.com/buckets/secret_stuff`

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - PATCH

- PATCH jest używany dla ****modyfikacji**** możliwości. Żądanie PATCH musi zawierać tylko zmiany w zasobie, a nie cały zasób.
- Przypomina to PUT, ale ciało zawiera zestaw instrukcji opisujących, jak zasób aktualnie rezydujący na serwerze powinien zostać zmodyfikowany, aby powstała jego nowa wersja. Oznacza to, że ciało PATCH nie powinno być po prostu zmodyfikowaną częścią zasobu, ale w pewnego rodzaju językiem latek, takim jak JSON Patch lub XML Patch.

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - PATCH

- PATCH jest używany dla ****modyfikacji**** możliwości. Żądanie PATCH musi zawierać tylko zmiany w zasobie, a nie cały zasób.
- Przypomina to PUT, ale ciało zawiera zestaw instrukcji opisujących, jak zasób aktualnie rezydujący na serwerze powinien zostać zmodyfikowany, aby powstała jego nowa wersja. Oznacza to, że ciało PATCH nie powinno być po prostu zmodyfikowaną częścią zasobu, ale w pewnego rodzaju językiem latek, takim jak JSON Patch lub XML Patch.

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - PATCH

- PATCH nie jest ani bezpieczny, ani idempotentny. Jednakże, żądanie PATCH może być wydane w taki sposób, aby było idempotentne, co również pomaga zapobiec negatywnym rezultatom kolizji pomiędzy dwoma żądaniami PATCH na tym samym zasobie w podobnym czasie. Kolizje wynikające z wielu żądań PATCH mogą być bardziej niebezpieczne niż kolizje PUT, ponieważ niektóre formaty poprawek muszą być obsługiwane ze znanego punktu bazowego, w przeciwnym razie uszkodzą zasób.
- Przykłady
 - `PATCH http://www.example.com/customers/12345`
 - `PATCH http://www.example.com/customers/12345/orders/98765`
 - `PATCH http://www.example.com/buckets/secret_stuff`

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - DELETE

- DELETE jest dość łatwe do zrozumienia. Służy do ****usunięcia**** zasobu identyfikowanego przez URI.
- Przy udanym usunięciu, zwraca status HTTP 200 (OK) wraz z ciałem odpowiedzi, być może reprezentacją usuniętego elementu , lub opakowaną odpowiedź. Lub też zwróci status HTTP 204 (NO CONTENT) bez treści odpowiedzi. Innymi słowy, status 204 bez ciała, lub odpowiedź w stylu JSEND i status HTTP 200 są zalecanymi odpowiedziami.

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - DELETE

- Zgodnie ze specyfikacją HTTP, operacje DELETE są idempotentne. Jeśli usuniesz zasób, to zostanie on usunięty. Wielokrotne wywołanie DELETE na tym zasobie kończy się tak samo: zasób znika. Jeśli wywołanie DELETE, powiedzmy, dekrementuje licznik (wewnętrz zasobu), wywołanie DELETE nie jest już idempotentne. Jak wspomniano wcześniej, statystyki użytkowania i pomiary mogą być aktualizowane, jednocześnie nadal uważając usługę za idempotentną, o ile żadne dane zasobu nie są zmieniane.
- Zalecane jest używanie POST dla żądań zasobów nie idempotentnych.

UŻYWANIE METOD HTTP DLA USŁUG RESTFUL - DELETE

- Istnieje jednak pewne zastrzeżenie dotyczące idempotencji DELETE. Wywołanie DELETE na zasobie po raz drugi często zwróci 404 (NOT FOUND), ponieważ zasób został już usunięty i dlatego nie można go już znaleźć. To, według niektórych opinii, sprawia, że operacje DELETE nie są już idempotentne, jednak stan końcowy zasobu jest taki sam. Zwracanie 404 jest akceptowalne i dokładnie komunikuje status wywołania.
- Przykłady
 - *DELETE http://www.example.com/customers/12345*
 - *DELETE http://www.example.com/customers/12345/orders*
 - *DELETE http://www.example.com/bucket/sample*

REST API

JAK TO ROBIĄ NAJLEPSI?

PRZYKŁADY PIĘKNEGO API

- GitHub - <https://docs.github.com/en/rest>
- Twitter - <https://developer.twitter.com/en/docs/twitter-api>
- Facebook - <https://developers.facebook.com/docs/graph-api/>
 - Coś ciekawego: <https://developers.facebook.com/blog/post/616/>
- LinkedIn - <https://docs.microsoft.com/en-us/linkedin>

CONTENT NEGOTIATION

JAK WYBRAĆ PREZENTACJĘ WIDOKU ZASOBU

CONTENT NEGOTIATION

- Klient wybiera pożądaną prezentację za pomocą nagłówka "Accept"
- W aplikacji dodaj dependency
 - <dependency>

```
<groupId>com.fasterxml.jackson.dataformat</groupId>
<artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```
- W przypadku niemożności zrealizowania żądania serwer odpowiada kodem "406" Not Acceptable
- Przykład content-negotiation
 - Sprawdź działanie przy pomocy Postmana oraz przeglądarki



DŁUGI CZAS PRZETWARZANIA

STRATEGIA ROZWIAZANIA PROBLEMU

ASYNCHRONICZNOŚĆ I PROCESY O DŁUGIM CZASIE WYKONYWANIA

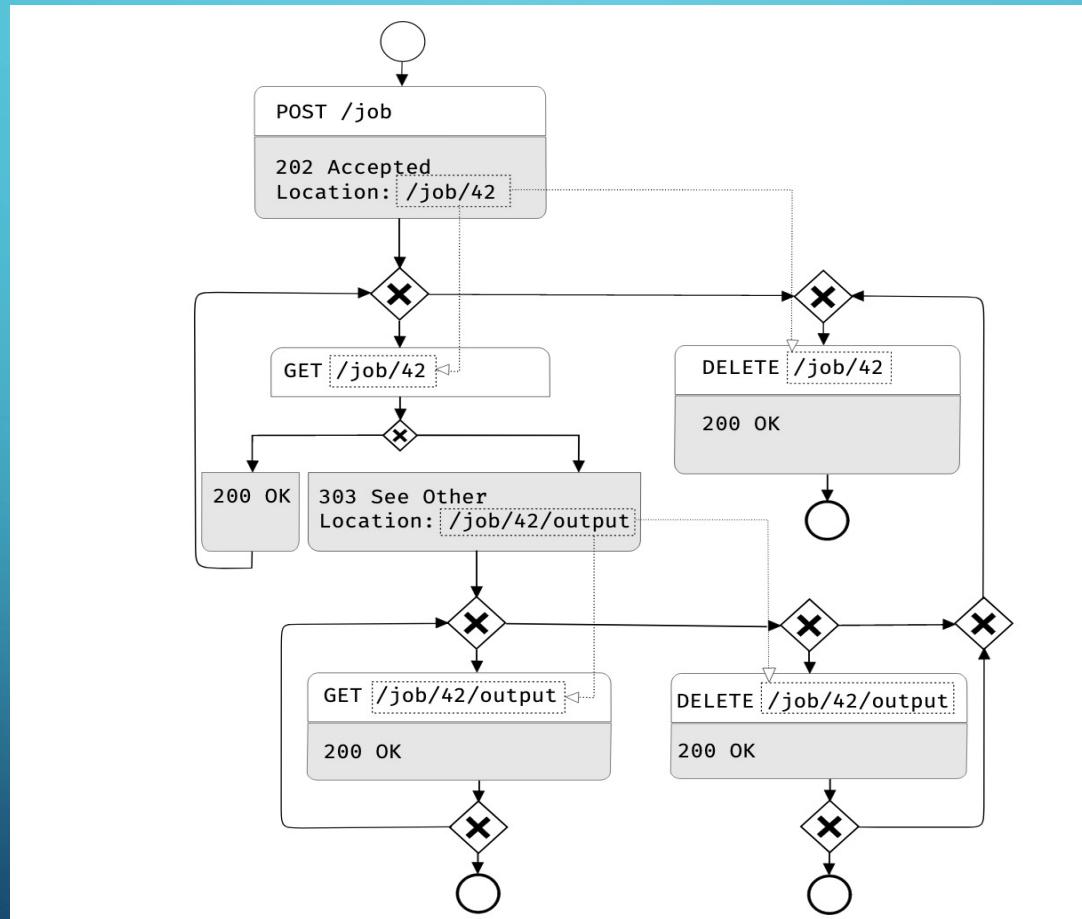
- Kontekst
 - Przetwarzanie złożonych lub wymagających dużej ilości danych operacji (np. przetwarzanie big data, zadania tworzenia kopii zapasowych) może wymagać dużo czasu.
- Problem
 - Jak klient może pobrać wynik takiej operacji bez utrzymywania otwartego połączenia HTTP przez zbyt długi czas? Zwłaszcza, że dla połączeń HTTP zwykle występuje timeout, ponieważ każde otwarte połączenie alokuje pewną ilość pamięci na serwerze i kliencie. Jak uniknąć marnowania zasobów na otwarte połączenia oraz na obliczenia, których wynik nie zostanie odebrany przez klienta w przypadku timeoutu?

ASYNCHRONICZNOŚĆ I PROCESY O DŁUGIM CZASIE WYKONYWANIA

- Rozwiążanie

- Trwająca operacja jest przekształcana w zasób, tworzony na podstawie oryginalnego żądania z odpowiedzią informującą klienta, gdzie może znaleźć wyniki.
- Klient może sprawdzać zasób, aby uzyskać jego aktualny postęp, i ostatecznie zostanie przekierowany do innego zasobu reprezentującego wynik, gdy dugo działająca operacja zostanie zakończona.
- Ponieważ dane wyjściowe mają swój własny URI, możliwe jest wielokrotne pobieranie ich, o ile nie zostały usunięte.
- Dodatkowo, dugo trwającą operację można anulować żądaniem DELETE, tym samym implicitely przerywając operację na serwerze lub usuwając jej wynik, jeśli w międzyczasie została już zakończona.

ASYNCHRONICZNOŚĆ I PROCESY O DŁUGIM CZASIE WYKONYWANIA



ASYNCHRONICZNOŚĆ I PROCESY O DŁUGIM CZASIE WYKONYWANIA

- Korzyści
 - Skalowalność: Klient nie musi utrzymywać otwartego połączenia z serwerem przez cały czas trwania żądania. Ma to pozytywny wpływ na liczbę klientów, których serwer może obsłużyć jednocześnie.
 - Współdzielenie wyników: Link do wyniku może być współdzielony pomiędzy wielu klientów, którzy mogą go pobrać bez potrzeby ponownego obliczania go przez serwer dla każdego klienta.
 - Anulowanie żądania: Zapewniony jest jawnny mechanizm zgodny z jednolitym interfejsem REST umożliwiający anulowanie żądań, a tym samym uniknięcie marnowania zasobów serwera na wykonywanie obliczeń, których wynikami klient nie jest już zainteresowany.

ASYNCHRONICZNOŚĆ I PROCESY O DŁUGIM CZASIE WYKONYWANIA

- Przykład:

- long-running-tasks

- Ćwiczenie

ASYNCHRONICZNOŚĆ I PROCESY O DŁUGIM CZASIE WYKONYWANIA

- Potencjalne problemy:

- Polling: Klient musi zaimplementować polling, który, jeśli jest wykonywany zbyt często, może dodatkowo obciążać serwer i zużywać niepotrzebnie pasmo. Aby zminimalizować ten problem, możliwe jest, że serwer może dostarczać klientowi informacje o postępie podczas pollingu, dzięki czemu liczba żądań GET może zostać zredukowana.
- Zużycie pamięci serwera: W zależności od typu i rozmiaru wyniku, przestrzeń dyskowa będzie zużywana, jeśli klienci zapomną usunąć wyniki zadań, a te nie są usuwane automatycznie po określonym czasie.

OBSŁUGA BŁĘDÓW



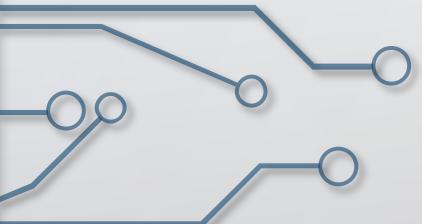
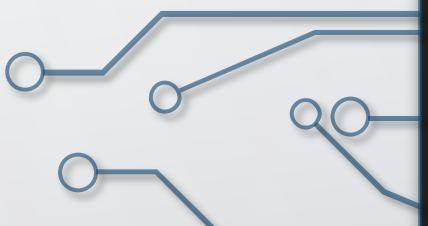
OBSŁUGA BŁĘDÓW

- Stosuj komponenty typu `RestControllerAdvice`
- Staraj się obsłużyć błędy biznesowe i nie dopuszczaj do wycieku stack trace'a wraz z kodem „500”
- Przykład: exception-handling
- Zadanie



WALIDACJA

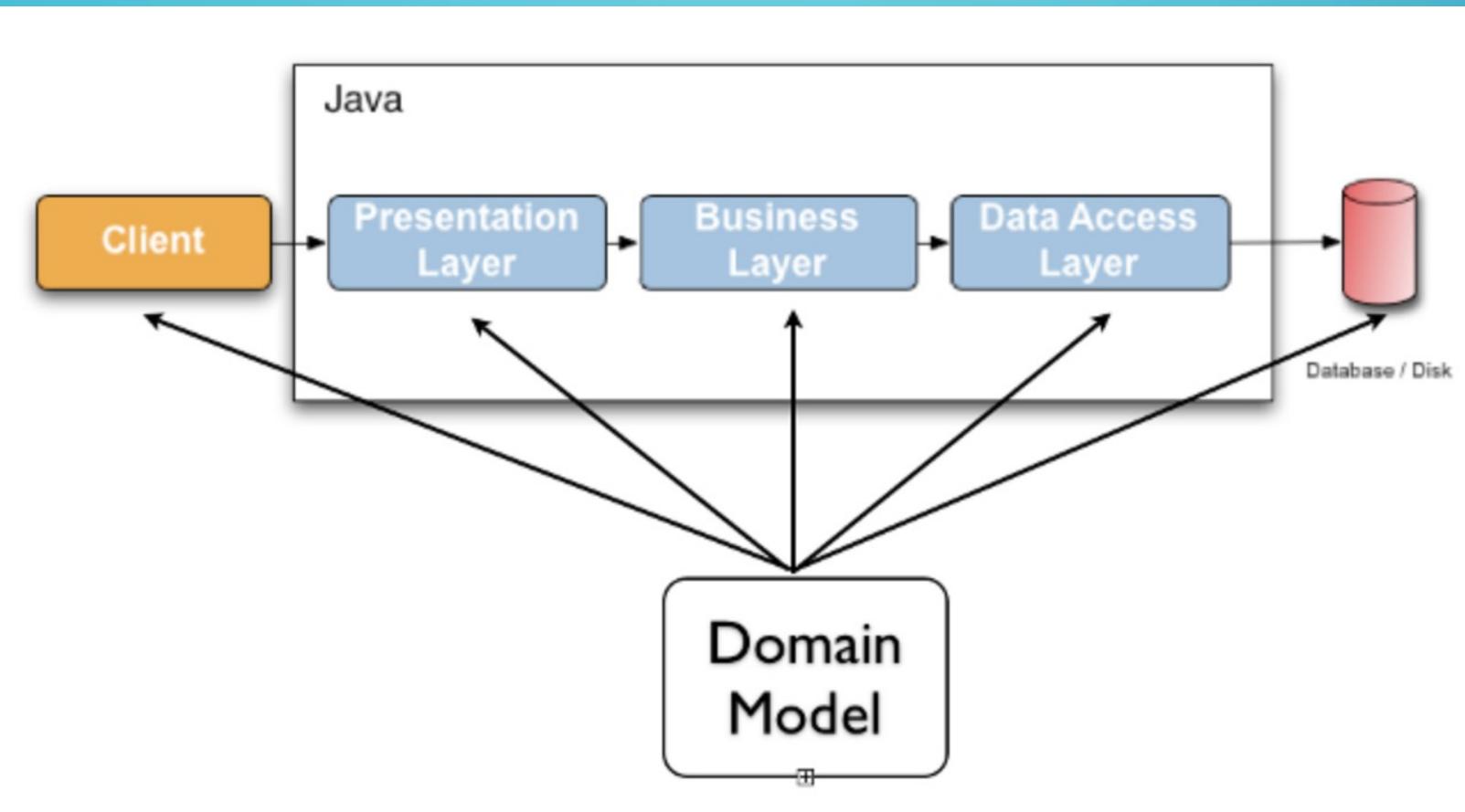
BEAN VALIDATIONS



WALIDACJA DANYCH

- Używaj do walidacji Beans Validation
- Zapewniony dostęp do wielu standardowych adnotacji
 - `@NotNull` waliduje, że adnotowana wartość właściwości nie jest zerowa.
 - `@AssertTrue` waliduje, że wartość właściwości opatrzonej adnotacją jest prawdziwa.
 - `@Size` waliduje, że adnotowana wartość właściwości ma rozmiar pomiędzy atrybutami min i max; może być zastosowany do właściwości String, Collection, Map i array.
 - `@Min` waliduje, że adnotowana właściwość ma wartość nie mniejszą niż atrybut value.
 - `@Max` waliduje, że adnotowana właściwość ma wartość nie większą niż atrybut value.
 - `@Email` waliduje, że adnotowana właściwość jest poprawnym adresem e-mail.

WALIDACJA DANYCH



WALIDACJA DANYCH

- `@NotEmpty` waliduje, że właściwość nie jest pusta lub null; może być zastosowany do wartości String, Collection, Map lub Array.
- `@NotBlank` może być zastosowany tylko do wartości tekstowych i waliduje, że właściwość nie jest pusta lub nie zawiera białych znaków.
- `@Positive` oraz `@PositiveOrZero` mają zastosowanie do wartości numerycznych i sprawdzają czy są one ściśle dodatnie lub dodatnie włączając 0.
- `@Negative` i `@NegativeOrZero` odnoszą się do wartości numerycznych i sprawdzają, czy są one ściśle ujemne, lub ujemne włączając 0.
- `@Past` i `@PastOrPresent` walidują, że wartość daty jest w przeszłości lub przeszłości włącznie z teraźniejszością; mogą być stosowane do typów daty włącznie z tymi dodanymi w Javie 8.
- `@Future` i `@FutureOrPresent` określają, że wartość daty jest w przyszłości lub w przyszłości łącznie z teraźniejszością.

WALIDACJA DANYCH

- Przykład: `data-validation`
- Zadanie

SPRING DATA REST

CZY TEGO NA PEWNO CHCE?

SPRING DATA REST

- Automatycznie wystawia wszystkie repozytoria jako endpointy restowe
- `spring.data.rest.base-path` – customowa ścieżka startowa
- `@RepositoryRestResource(collectionResourceRel = "auta", path = "samochody")`
- Przykład: `rest-data`