

**Spółeczna Akademia Nauk**  
**Wydział Studiów Międzynarodowych i Informatyki**

**Kierunek studiów:**

**INFORMATYKA**

**Kamil Cecherz**

**Numer albumu: 82739**

**RESTful WebAPI dla technologii Java.**

**Praca inżynierska napisana pod kierunkiem**

**dr inż. Konrada Grzanka**

**Termin obrony:.....**

**Recenzent: .....**

# Spis treści

|                                                                               |           |
|-------------------------------------------------------------------------------|-----------|
| <b>Wprowadzenie .....</b>                                                     | <b>3</b>  |
| Opis wzorca: Richardson Maturity Model.....                                   | 3         |
| Narzędzia.....                                                                | 4         |
| Gotowe REST API.....                                                          | 7         |
| Założenia projektu.....                                                       | 8         |
| <b>Rozdział I – Poziomy dojrzałości API.....</b>                              | <b>12</b> |
| Przykład: Aplikacja „Hello World” .....                                       | 12        |
| Level 0: SWAMP OF POX .....                                                   | 13        |
| Level 1: RESOURCES .....                                                      | 17        |
| Level 2: HTTP METHODS .....                                                   | 18        |
| Level 3: HYPERMEDIA CONTROLS .....                                            | 31        |
| <b>Rozdział II – Zapis informacji w bazie danych .....</b>                    | <b>36</b> |
| Założenia projektu a metoda utrwalania danych.....                            | 36        |
| Model danych oparty o ORM.....                                                | 38        |
| Reprezentacja relacji w REST API .....                                        | 40        |
| Obsługa wyjątków .....                                                        | 43        |
| <b>Rozdział III – Przeglądarkowa aplikacja Web, jako warstwa widoku. ....</b> | <b>47</b> |
| Wybór medium dla aplikacji .....                                              | 47        |
| Szablony HTML5 - Spring Thymeleaf.....                                        | 50        |
| Manipulacja DOM przy pomocy jQuery.....                                       | 54        |
| <b>Podsumowanie.....</b>                                                      | <b>57</b> |
| <b>Bibliografia.....</b>                                                      | <b>58</b> |
| <b>Spis ilustracji .....</b>                                                  | <b>59</b> |
| <b>Spis tabel.....</b>                                                        | <b>60</b> |

# Wprowadzenie

## Opis wzorca: Richardson Maturity Model

Pracę poświęcam zagadnieniu tworzenia aplikacji, zgodnie z konwencjami architektonicznego stylu oprogramowania REST (Representational State Transfer). Przedstawiam etapy ewolucji warstwy kontroli, przetwarzania danych w aplikacji webowej. Cztery poziomy złożoności, które możemy wyróżnić w drodze do stworzenia dojrzałego REST API. Opis techniczny opieram o fundamentalne zasady stylu wypracowane przez Roy 'a Fielding 'a w pracy doktorskiej z roku 2000 pt. „Architectural Styles and the Design of Network-based Software Architectures”<sup>1</sup>

Model Roy 'a Fielding 'a, który stanowi punkt wyjścia dla projektów programistycznych jest niezależny od rodzaju implementacji, specyfikacji różnych języków programowania. W przypadku REST-a mamy do czynienia ze wzorcem, nie jest to framework, biblioteka programistyczna ani protokół. Trzymanie się zasad REST API nie jest wspomagane przez narzędzia, które służą do jego budowy. Developer przejmuje część odpowiedzialności architekta, to on musi zadbać o utrzymanie klarowności wzorca. W niniejszym opracowaniu opisuje proces „dostrajania” kodu aplikacji, który niejednokrotnie w obciążonej błędami realnej pracy programisty, musi przejść przebudowę, aby stać się poprawną realizacją schematu wydajnej komunikacji przy maksymalnym wykorzystaniu możliwości protokołu HTTP.

Możliwe jest stworzenie aplikacji, która luźno realizuje założenia wzorca łamiąc przy tym ograniczenia. Działanie takiego programu może przypominać funkcjonalnie aplikacje z poprawnie napisanym REST API jednak od strony komunikacyjnej, niewidocznej bezpośrednio dla użytkownika, będzie kolidowało z zasadami opracowanymi przez Roy 'a Fielding 'a. Tym samym utraci zalety takie jak uniwersalność czy skalowalność<sup>2</sup>. W pracy zacznę od opisu etapu projektu znanego, jako „The Swamp of POX”, stanowiący przykład antywzorca:

---

1 [https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\\_dissertation.pdf](https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf) (odwiedzono dn. 05.11.2019)

**UNIVERSITY OF CALIFORNIA, IRVINE** Architectural Styles and the Design of Network-based Software Architectures  
**DISSERTATION** submitted in partial satisfaction of the requirements for the degree of  
**DOCTOR OF PHILOSOPHY** in Information and Computer Science by Roy Thomas Fielding  
Dissertation Committee: Professor Richard N. Taylor, Chair Professor Mark S. Ackerman Professor David S. Rosenblum  
**2000**

2 <https://patryk-nizio.pl/2018/03/18/rest-wprowadzenie/> (odwiedzono dn. 05.11.2019) Patryk Nizio - „Wprowadzenie.  
Dlaczego warto poznać REST i co to jest?”

Autor artykułu wymienia takie zalety jak: niezależność od technologii i platform, bezstanowość, łatwy dostęp.

*Usługi [...] mają jeden identyfikator URI i używają jednej metody HTTP (zazwyczaj POST). Jest to najbardziej prymitywny sposób budowania aplikacji SOA za pomocą jednej metody POST i używania XML do komunikacji między usługami.<sup>3</sup>*

Przechodząc stopniowo do kolejnych poziomów API:

- Level 1: Resources
- Level 2: HTTP Methods
- Level 3: Hypermedia Controls

Uzupełniającym etapem mojej pracy jest realizacja warstwy utrwalania danych i widoku, w oparciu o prosty projekt aplikacji do zapisu rachunków domowych. Wykorzystanie maperów obiektowo-relacyjnych i związanych z tym technik obsługi, przekazywania danych pomiędzy serwerem a API, API a klientem.

## Narzędzia

W celu stworzenia poprawnie działającej aplikacji webowej opartej o model REST potrzebne jest oprogramowanie odwzorowujące działanie uczestników komunikacji sieciowej, w oparciu o protokół HTTP:

*The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible entity-body content [...]<sup>4</sup>*

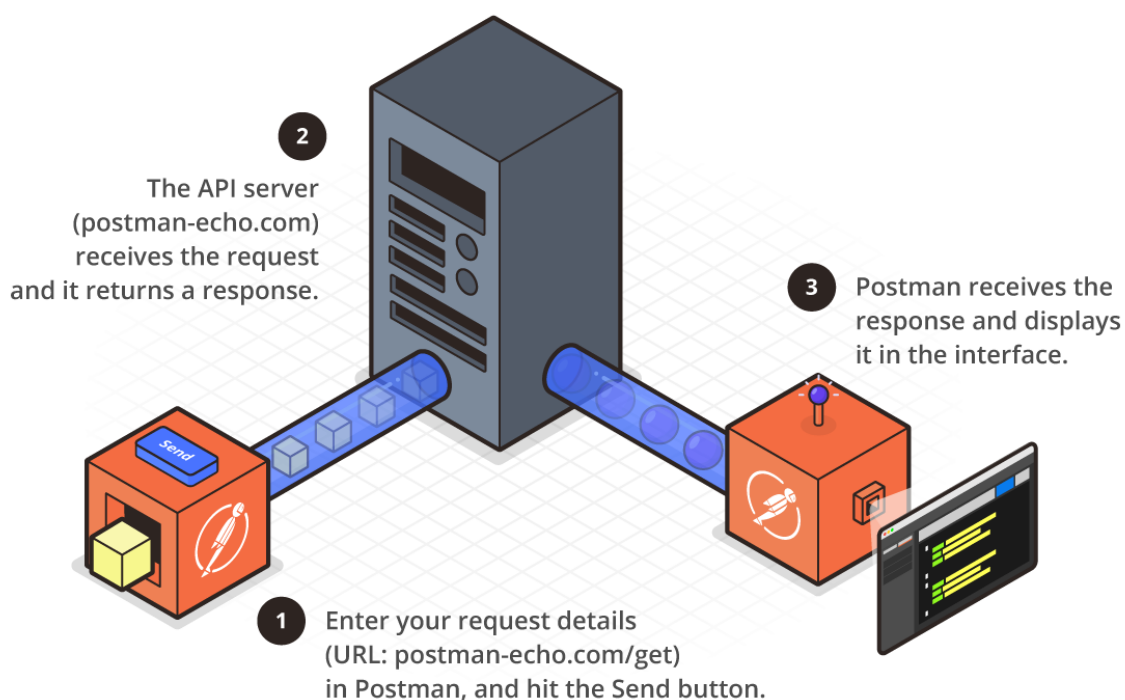
Klient, jakim jest przeglądarka WWW, stanowi dobre narzędzie do testów dla webowego API. Obecnie Chrome, Firefox czy Edge posiadają bogatą paletę wtyczek, które pozwalają deweloperom zobaczyć więcej niż konsumentom Internetu, przeciętnym ludziom, których interesuje przede wszystkim

---

3 <https://blog.restcase.com/4-maturity-levels-of-rest-api-design/> (odwiedzono 05.11.2019) „4 Maturity Levels of REST API Design” (aktualizacja 25.11.2018) - tłumaczenie fragmentu artykułu języka ang.

4 <https://tools.ietf.org/html/rfc2068> (odwiedzono 08.11.2019) „Hypertext Transfer Protocol - HTTP/1.1” Network Working Group Request for Comments: 2068 UC Irvine Category: Standards Track  
J. Gettys, J. Mogul, R. Fielding DEC H. Frystyk, T. Berners-Lee MIT/LCS January 1997

zawartość stron WWW w postaci HTML-a. API REST-owe powinno być jednak uniwersalne, format przekazywanych nie jest ważny, chcemy mieć możliwości wygodnej obsługi XML-a, JSON-a, czystego tekstu, CSV, itd. W celu stworzenia wygodnego, skalowalnego API, które będzie mogło zostać użyte przez klientów na różnych urządzeniach, potrzeba programiście pełnej kontroli przepływu danych w oparciu o HTTP. Protokół specyfikuje metody, określa zachowania i właściwości dla obiektów żądań i odpowiedzi.<sup>5</sup> Deweloper aplikacji zostaje odpowiedzialny za odpowiednie rozłożenie, zmapowanie zasobów na adresy URI. To zdecydowanie dużo, stąd pomysł stworzenia narzędzi takich jak Postman<sup>6</sup>



**Ilustracja 1.** Schemat opisujący przekazywanie odpowiedzi o oparciu o metody i zasoby URL

Postman pozwala na bezpośredni kontakt z API. Nie musimy znać rozbudowanej biblioteki Curl<sup>7</sup>. Program posiada czytelny interfejs graficzny, który pozwala wykonać serię szybkich testów bez

---

5 [https://pl.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol#Typowe\\_zapytanie\\_HTTP](https://pl.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Typowe_zapytanie_HTTP)  
[https://pl.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol#Odpowied%C5%BA\\_serwera\\_WWW](https://pl.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Odpowied%C5%BA_serwera_WWW)  
(odwiedzono 08.11.2019) „Hypertext Transfer Protocol”

6 <https://learning.getpostman.com/docs/postman/launching-postman/sending-the-first-request/>  
(odwiedzono 08.11.2019) „Sending the first request”

7 <https://curl.haxx.se/> (odwiedzono dn. 08.11.2019) - dokumentacja: CURL - command line tool and library for transferring data with URLs

zagłębiania się w dokumentację. W pewnym uproszczeniu jednak Postman jest rodzajem przeglądarki WWW (w tym wypadku warstwę graficznej prezentacji zastępuje postać tekstowa).

Niezbędnym do wystawienia własnego API jest serwer aplikacyjny, testowe narzędzie lokalne, które pozwoli na „odpytywanie” adresów URL aplikacji, gdzie zlokalizowane są jej zasoby.

W pracy rezygnujemy z dokładnego opisu konfiguracji narzędzi takich jak Tomcat czy Jetty. Stanowią one część projektu Spring Boot opartego o najpopularniejszy obecnie framework dla aplikacji webowych pisanych w Javie, Spring. Spring Boot<sup>8</sup> obok takich projektów jak Java Spark, Restlet czy Java Lite jest jednym z wielu narzędzi przystosowanych, do tworzenia aplikacji z REST-owym API<sup>9</sup>.

Rozwiązania te upraszczają tworzenie fragmentów oprogramowania w modelu żądanie-odpowiedź w oparciu zasady protokołu HTTP. Kontrolery REST stanowią pewien etap ewolucji Servletów. Java Servlet API<sup>10</sup> leży u podstaw Java Enterprise Platform<sup>11</sup>, początkowo był to jedyny sposób sterowania przepływem wymiany danych w oparciu o HTTP. Oba rozwiązania jednak różnią się od siebie, co warto podkreślić:

1. *Servlets are Java specific but RESTful web services are not.*
2. *Servlets are API but RESTful is not.*
3. *RESTful web service can use Servlets as there implementation but vice versa is not true.*
4. *Servlets can run in Servlet container only but RESTful services can run in web container as well.*
5. *RESTful talks about Resources/Entities/Verbs and gives you more Specific way to use the services i.e There is clear explanation on the usage of HTTP verbs.*
6. *Request handling in both is totally different e.g. Servlets are multi-threaded inherently but RESTful services are not.*

---

8 <https://spring.io/projects/spring-boot> (odwiedzono dn. 10.11.2019) Strona projektu Spring Boot.

9 <https://www.gajotres.net/best-available-java-restful-micro-frameworks/> (odwiedzono dn. 10.11.2019) „8 Top Java RESTful Micro Frameworks of 2019 [Pros/Cons/Stats]”

10 <https://docs.oracle.com/javaee/7/api/javax/servlet/package-summary.html>

(odwiedzono dn. 10.11.2019) Dokumentacja Oracle - fragm. The javax.servlet package contains a number of classes and interfaces that describe and define the contracts between a servlet class and the runtime environment provided for an instance of such a class by a conforming servlet container.

11 <https://www.oracle.com/java/technologies/java-ee-glance.html> (odwiedzono dn. 10.11.2019)

7. *If you want to share the RESTful services with other teams/client/public to consume there are plenty of Document tools which can generate documentation for your REST services (e.g swagger) but you won't find that kind of documentation tools with Servlets.*<sup>12</sup>

Wzorzec Roy 'a Fielding 'a jest dużo wygodniejszym rozwiązaniem, stąd przewaga technologii, które go wykorzystują. Chodzi przede wszystkim o to, by zmniejszyć ilość nadmiarowego kodu, czyniąc logikę aplikacji zrozumiałą.

Spring Boot jest zestawem gotowych narzędzi, zapewniających pełną automatyzację pracy. Dostarcza wbudowany serwer aplikacyjny, pozwala na wykorzystanie starterów, dzięki którym ułatwiona staje się konfiguracja zależności w projekcie, nie wymaga konfiguracji poprzez pliki XML. Wykorzystując Spring Boot 'a rezygnujemy z wymagającej konfiguracji, koncentrując się na aplikacji *stand-alone*.

## Gotowe REST API

Udostępnianie gotowego REST API deweloperom stało się powszechną praktyką. Uwagę zwracam tu w kierunku takich realizacji jak: Google Geolocation API, YouTube Data API czy WordPress REST API, które są dziełem internetowych gigantów<sup>13</sup>. Webowe API bardzo wspomaga pracę programisty, gdyż dostarcza mu ono gotowych komponentów. Bez znajomości detali danej platformy, możliwe jest zrealizowanie „konsumpcji” danych przez aplikację. Mając do dyspozycji gotowe endpointy rezygnujemy z warstwy logiki biznesowej aplikacji, deweloper skupia się zasobach zmapowanych na konkretne adresy WWW. To bardzo wygodne rozwiązanie.

Prowadzę stronę WWW blog oparty na CMS Wordpress 'a. Gdy przeczytałem o API dla tego bardzo popularnego silnika, postanowiłem je przetestować. W oparciu o przejrzystą dokumentację<sup>14</sup>, udało mi się wykonać zastosować kryteria do wyświetlania i sortowania danych dotyczących opublikowanych postów. Gdybym zechciał wykonać zestawienie danych dotyczących artykułów oraz ich lokalizacji na stronie w zupełnie innym miejscu, z widokiem w formie tabeli nie stanowiłoby to problemu.

---

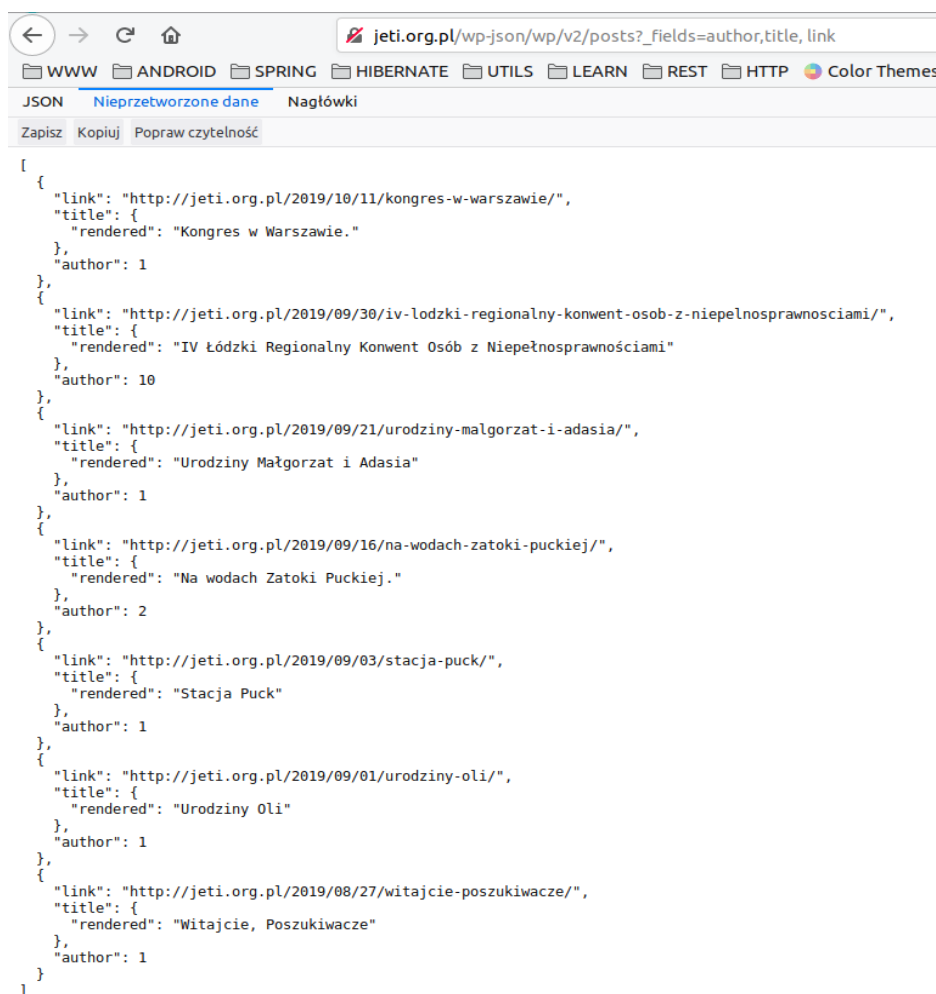
12 <https://www.kutayzorlu.com/introduction-to-java-programming-java-2/servlet-vs-restful-web-service-15046.html>

(odwiedzono dn. 10.11.2019) „You can, for example, implement REST-like services using Servlets” (aktualizacja: 20.08.2019) odniesienia: Atul Tiwari, Big data and Analytics (2016-present) Gowtham

13 <https://rapidapi.com/blog/most-popular-api/> (odwiedzono dn. 09.11.2019) „Top 50 Most Popular APIs on RapidAPI (2019)” Artykuł przedstawia realizację 50 API należą do nich m.in. API Nasa, API Chucka Norrisa, API Pokemon Go

14 <https://developer.wordpress.org/rest-api/> (odwiedzono dn. 09.11.2019) „REST API Handbook” Dokumentacja API Wordpress'a. Platformy, CMS-a napisanego w języku PHP służącego do budowy m.in blogów internetowych.

Wejście w link: [http://jeti.org.pl/wp-json/wp/v2/posts?\\_fields=author,title,link](http://jeti.org.pl/wp-json/wp/v2/posts?_fields=author,title,link) zwraca zasoby w postaci JSON-a o następującej treści:



**Ilustracja 2.** Zrzut ekranu z przeglądarki: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:70.0) Gecko/20100101 Firefox/70.0

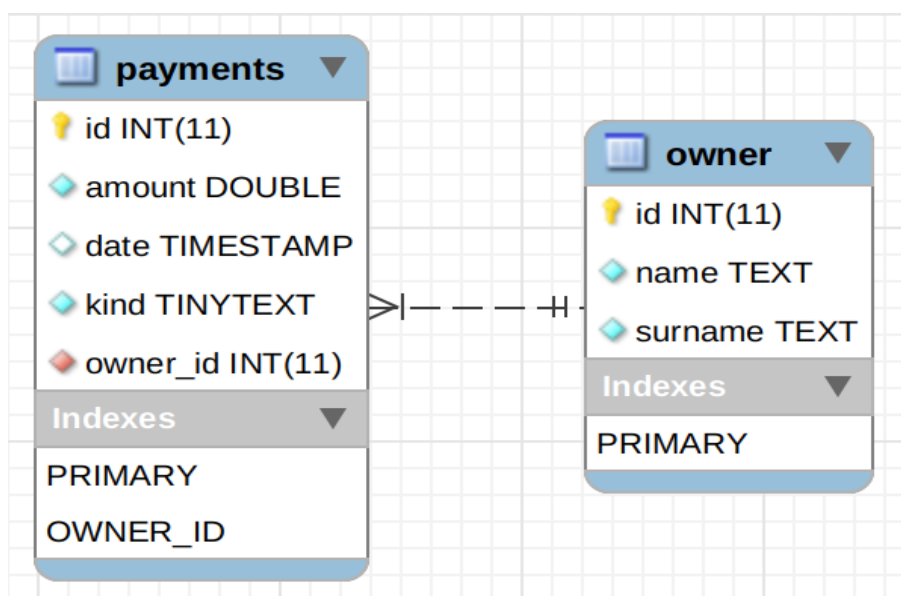
Przykład stanowi zaledwie ułamek możliwości API Wordpress 'a, które nie musi być narzędziem wyłącznie dla twórców aplikacji. Zaprezentowane dane można „obsłużyć” przykładowo w widoku na urządzenia mobilne, gdzie króluje platforma Android. REST-owe API jest rozwiązaniem uniwersalnym.

## Założenia projektu

W celu zilustrowania procesu tworzenia webowego API przechodzę przez kolejne etapy dojrzałości Roy 'a Fielding 'a. Aplikacja mojego pomysłu, opisuje przykład wykorzystania Spring Boot 'a, jest ona dobrym przykładem do nauki, nie wyczerpuje jednak możliwości frameworka. Wybrałem

projekt, którego wdrożenie nie wymaga dużej ilości kodu, zależy mi na esencji, zgłębieniu zagadnienia REST API. Warto jednak pamiętać, że prezentowane rozwiązania znajdą zastosowanie przy realizacji dużych serwisów, projektów, gdzie liczy się skalowalność i przenośność.

Prezentuje aplikację do przechowywania informacji o rachunkach użytkowników zamieszkujących wspólne gospodarstwo domowe. Celem programu jest gromadzenie danych o płatnościach, z podziałem na rodzaj z uwzględnieniem daty. Informacje o kwotach rachunków zostają z agregowane w tabeli posiadającej relację do wykazu danych ich właścicieli. Aplikacja pozwala na filtrowanie danych, wykonywanie prostych numerycznych zestawień.



**Ilustracja 3.** Diagram tabel wykonany w programie MySQL Workbench

### Warstwa utrwalania danych

W projekcie wykorzystuje mapery obiektowo-relacyjne w celu uproszczenia modelu. Zmniejszenia ilości warstw pośredniczących w dostępie do danych, o które musi dbać programista. Wykorzystuje silnik bazy danych MySQL<sup>15</sup>, skupiam się jednak odwzorowaniu tabel w postaci klas encyjnych.

---

<sup>15</sup> W przypadku Mavena wykorzystywanego przy konfiguracji Spring Boot'a wystarczy dodanie odpowiedniej zależności celem dostępu do MySQL-a:

```
<dependency><groupId>mysql</groupId><artifactId>mysql-connector-java</artifactId><version>8.0.18</version></dependency>
```

Specyfikacja Oracle'a dostępna jest pod adresem: <https://dev.mysql.com/doc/> (odwiedzono dn. 11.11.2019)

```
mysql> describe owner;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11) | NO | PRI | NULL | auto_increment |
| name  | text | NO | | NULL | |
| surname | text | NO | | NULL | |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> describe payments;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11) | NO | PRI | NULL | auto_increment |
| amount | double | NO | | NULL | |
| date  | timestamp | YES | | CURRENT_TIMESTAMP | DEFAULT_GENERATED |
| kind  | tinytext | NO | | NULL | |
| owner_id | int(11) | NO | MUL | NULL | |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

*Ilustracja 4. Widok struktury tabel w postaci tekstowej*

### Warstwa widoku

W celu stworzenia przyjemnego interfejsu webowego dla użytkownika, korzystam z dobrodziejstw Spring Boot 'a. W projekcie wykorzystuje Thymeleaf<sup>16</sup>, który jest silnikiem do szablonów HTML/XHTML. Dzięki narzędziu możemy zapewnić aplikację wygodną prezentację i wymianę danych. Chociaż komunikacja z REST API może odbywać się poprzez same linki nie stanowi to zachęcającej perspektywy dla użytkownika. W projekcie realizuje wzorzec MVC w celu zilustrowania wykorzystania działającego API. W tym aspekcie jest to luźny wybór, parsowanie danych w postaci takiej jak np. JSON, można zrealizować dowolnie. Wybór Themeleaf został podyktowany łatwością integracji z frameworkiem Spring.

### Ograniczenia

Poprawnie zrealizowana aplikacja webowa zawiera elementy, które z racji tematu rozpiętości tematu pominię. Otwartym zastanawiam temat zabezpieczenia aplikacji: poprzez mechanizm logowania, kontrolę długości sesji. Pomijam sposoby na skuteczną walidację danych dostarczanych przez użytkowników. Nie zrealizuje funkcjonalności wysyłania maili poprzez aplikację, która mogłaby

---

16 W przypadku Mavena wykorzystywanego przy konfiguracji Spring Boot'a dodajemy następującą zależność:

```
<dependency><groupId>org.thymeleaf</groupId><artifactId>thymeleaf
</artifactId><version>3.0.11.RELEASE</version></dependency>
```

Dokumentacja narzędzia: <https://www.thymeleaf.org/documentation.html> (odwiedzono dn. 11.11.2019)

stać się uzupełnieniem mechanizmu rejestracji. Zakładamy, że korzystając z ogólnodostępnego API dostarczamy użytkownikowi pewien prototyp, program obsługiwany przez przeglądarkę, ale jednak niedostępny w Internecie.

Skupię się na realizacji operacji CRUD, filtrowaniu danych oraz stworzeniu funkcji umożliwiających robienie prostych kalkulasi rachunkowych.

Bazą mojego projektu jest JDK<sup>17</sup> w wersji 12, środowisko niezbędne do programowania w języku Java. Kody źródłowe aplikacji realizuje wykorzystując funkcje języka dostępne od wersji 8, chodzi przede wszystkim o Stream API<sup>18</sup> oraz Lambda expression<sup>19</sup>. REST API zrealizowane z wykorzystaniem dobrodziejstw programowania funkcyjnego posiada bardziej zwarty i czytelny zestaw instrukcji.

### Kod źródłowy

Pliki mojej aplikacji, w postaci kodu źródłowego znajdują się w publicznym repozytorium na portalu GitHub: <https://github.com/pangeon/calc-bill>. Aby uruchomić program należy sklonować repozytorium, skompilować klasy i uruchomić przy go w przeglądarce.

---

17 <https://www.oracle.com/technetwork/java/javase/12all-relnotes-5211423.html> (odwiedzono dn. 11.01.2019)

Informacje o wersji dla wydań aktualizacji JDK 12 i JDK 12

18 <https://howtodoinjava.com/java8/java-streams-by-examples/> (odwiedzono dn. 11.01.2019) Lokesh Gupta - „Java 8 Stream API”

19 <https://howtodoinjava.com/java8/lambda-expressions/> (odwiedzono dn. 11.01.2019) Lokesh Gupta - „Lambda expressions”

## Rozdział I – Poziomy dojrzałości API

### Przykład: Aplikacja „Hello World”

```
1 /* index.html */
2 <html>
3   <head>
4     <meta http-equiv="Content-Type"
5       content="text/html; charset=UTF-8"
6     />
7   </head>
8   <body>
9     <p>Wykonaj test
10    <a href="/greeting">kliknij</a>
11  </p>
12 </body>
13 </html>
14
15 /* GreetingController.java */
16 @Controller
17 public class GreetingController {
18   @GetMapping("/greeting")
19   public String greeting(
20     @RequestParam(name="name", required=false, defaultValue="World")
21     String name, Model model)
22   {
23     model.addAttribute("name", name);
24     return "greeting";
25   }
26 }
```

*Ilustracja 5. Widok w postaci pliku HTML oraz klasa kontrolera*

REST API napisane z wybranym zestawem technologii zawiera konkretne słowa kluczowe, rodzaj dialektu, określony styl definiowania instrukcji. Charakterystyczną techniką programowania dla Spring’ a jest wykorzystanie adnotacji (inaczej meta-danych)<sup>20</sup>. Adnotacje mówią kompilatorowi, co dodatkowo ma wykonać z obiektowymi strukturami kodu Javy. Zwiększa to klarowność, zwięzłość kodu klas przeznaczonych do obsługi danych przy pomocy żądań HTTP.

Pełny obraz daje przykład, kody które zamieściłem, jako część projektu aplikacji: klasa z prostą funkcją zwracającą ciągi znaków, dzięki użyciu adnotacji staje się kontrolerem HTTP przekazującym dane, przy użyciu metody GET. Na zapytanie w postaci adresu URL: `http://localhost:8080/greeting?name=Kamil`. Przeglądarka odpowie wygenerowaniem strony o zawartości w postaci pliku HTML, którego częścią są napisy: *Hello, Kamil!*

---

<sup>20</sup> <https://kobietydokodu.pl/niezbędnik-juniora-adnotacje/> (odwiedzono dn. 14.01.2020) Anna Pietras - „Niezbędnik Juniora. Adnotacje” (aktualizacja 15.04.2015)

Adnotacje pozwalają na uproszczenie konfiguracji, rezygnację z dodatkowych plików XML. Identyczną funkcjonalność można uzyskać wykorzystując niskopoziomowe dla aplikacji webowych Servlet API jednak będzie się to wiązało z dużo większym narzutem pracy m.in ręczną obsługą obiektów żądań oraz odpowiedzi i definiowaniem typu zwracanych danych. Obecnie rekomendowaną praktyką przy budowie REST API jest korzystanie z frameworków.

## Level 0: SWAMP OF POX

Najniższy poziom dojrzałości REST API określamy, jako *The Swamp of POX*<sup>21</sup>. W praktyce chodzi o złamanie wszystkich zasad modelu Richardson 'a. Co to jednak oznacza?

Pisanie lekkich aplikacji, łatwych do wdrożenia w Sieci, zgodnie z tym wzorcem, przy użyciu frameworka takiego jak Spring, może okazać się trudne, czasochłonne i nieefektywne:

*Do poziomu 0 zalicza się przesyłanie danych przy użyciu technologii SOAP i XML-RPC jako POX [...] Używana jest tylko metoda POST. Jest to najprostszy sposób budowania aplikacji SOA z jedną metodą POST i przy użyciu formatu XML do komunikacji między usługami.*<sup>22</sup>

Celem REST-owego modelu aplikacji jest maksymalne wykorzystanie możliwości komunikacyjnych oferowanych przez protokół HTTP. Programistę posługującego się Spring Boot czy Java Spark nie interesują rozwiązania wykorzystywane przez technologię SOAP<sup>23</sup>, która znajduje zastosowanie w szerszym zakresie, gdy szczególnie zależy nam na:

- Wykorzystaniu innych protokołów komunikacyjnych, takich jak: STMP czy UDP
- Wsparciu dla Web Services Security

REST API jest dobrym rozwiązaniem, gdy liczy się zwiększenie elastyczności. API REST 'owe sprawdzi się w przypadku długo rozwijanego serwisu dla szerokiej gamy konsumentów, ze względu na:

- Obsługę wielu formatów, takich jak: JSON, HTML czy YAML.
- Lepszą wydajność i szybkość poprzez mniejsze wykorzystanie mocy obliczeniowej maszyn.
- Bezstanowość, która wpływa na skalowalność i łatwość modernizacji programów.

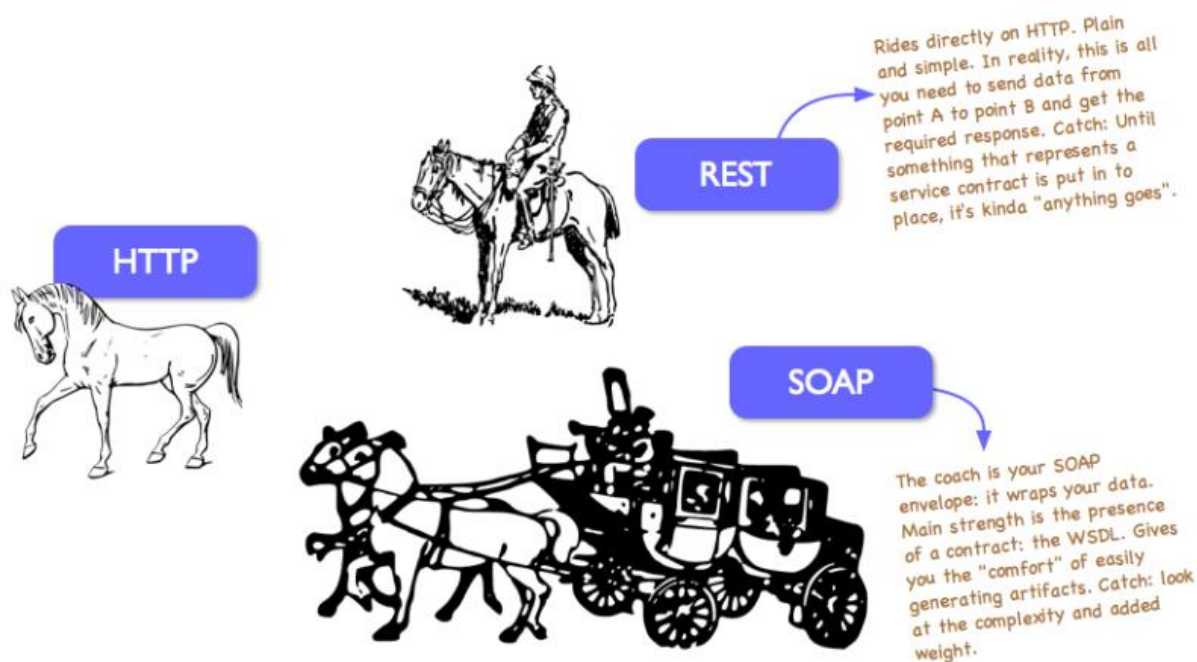
Nie należy rozważać modelu Richardson 'a w kontekście SOAP Web Services, są to dwa osobne pomysły.

---

21 W dosłownym tłum. bagno prostego starego XML.

22 Bhakti Mehta, REST – *Najlepsze praktyki i wzorce w języku Java*, tłum. Łukasz Piwko, wyd. Helion, Gliwice 2015, s. 17

23 <http://www.developer.net.pl/ogolne/rest-vs-soap/> (odwiedzono dn. 05.02.2020) „REST VS SOAP?” (aktualizacja 07.01.2020)



**Ilustracja 6.** Porównanie technologii opartych na REST oraz SOAP<sup>24</sup>

Schodkowe, ewolucyjne podejście do projektowania pomaga uporządkować etapy budowy aplikacji. W uzyskaniu pełnej klarowności wymagany jest poziom 2, maksimum funkcjonalności daje zaś wdrożenie HATEOAS<sup>25</sup>. Nie wszystkie aplikacje będą posiadały API z pełną realizacją modelu Richardson 'a, to uczciwe podejście.

Napisanie aplikacji o charakterze Swamp of POX z wykorzystaniem Spring Boot 'a jest możliwym do zrealizowania zadaniem, którego efektem jest korzyść w postaci wiedzy. Autor rozprawy „Architectural Styles and the Design of Network-based Software Architectures” daje wskazówki, zbiór zasad, odnośnie tego jak powinna wyglądać idealna realizacja jego modelu, by jednak to zrozumieć, należy je złamać.

Cechy antywzorca REST-owego API będą wyglądały następująco:

- Brak podziału serwisów ze względu na charakter obsługiwanych typów i rodzajów informacji. Duża klasa odpowiedzialna za obsługę wszystkich żądań HTTP.
- Posłużenie się tylko jedną z możliwych metod HTTP POST. Jedna funkcja z wieloma instrukcjami warunkowymi, do której argument zostaje przekazany przez żądanie.
- Cała zawartość dostępna z poziomu jednego zasobu, jednego adresu URI.

<sup>24</sup> Ibidem., źródło: <http://formacdownload.com>

<sup>25</sup> Hypermedia as the Engine of Application State - <https://geek.justjoin.it/hateoas-api-moze-byc-lepsze/> (odwiedzono dn. 27.03.2020) Mateusz Stasch - „HATEOAS. Czy Twoje API może być lepsze?” (aktualizacja 24.09.2018)

Brak fundamentalnych cech dla dobrego oprogramowania webowego, nie sprawia, że zmniejsza się funkcjonalność napisanych w ten sposób aplikacji, maleje przede wszystkim ich czytelność, a co za tym idzie możliwości skalowania. Program trudniejszy w konserwacji jest droższy.

```
@RestController
@RequestMapping("/api/maturity/l0/service")
public class AppService {
    private List<Document> documents = initDocuments();

    @PostMapping
    public ResponseEntity<Object> handleCommand(@RequestBody Command command) {
        if ("getAllDocuments".equals(command.getAction())) {
            return ResponseEntity.ok(documents);
        }
        else if ("addDocument".equals(command.getAction())) {
            documents.add(command.getDocumentToAdd());
            return ResponseEntity.ok().build();
        }
        else if ("deleteDocument".equals(command.getAction())) {
            documents.removeIf(document -> document
                .getNumber() == command.getNumberToRemove());
            return ResponseEntity.ok().build();
        }
        else if ("getAllProfiles".equals(command.getAction())) {
            return ResponseEntity.ok(allProfiles());
        }
        else if ("downloadFile".equals(command.getAction())) {
            return ResponseEntity.ok()
                .contentType(MediaType.IMAGE_PNG)
                .body(new ClassPathResource("/images/image" +
                    ".png", getClass()));
        }
        else if ("managePermissions".equals(command.getAction())) {
            // TODO: change permissions
            return ResponseEntity.ok().build();
        }
        else {
            throw new IllegalArgumentException("Unknown action");
        }
    }
}
```

*Ilustracja 7. Kod źródłowy: przykład serwisu aplikacji typu Swamp of POX<sup>26</sup>*

Zaprezentowany antywzorzec, klasy autorstwa Michała Wojtyny dodałem, jako osobny pod-projekt w pakiecie **pl.strefakursow.swamp\_of\_pox<sup>27</sup>**. Kod jest możliwy do przetestowania, nie ma zaś

---

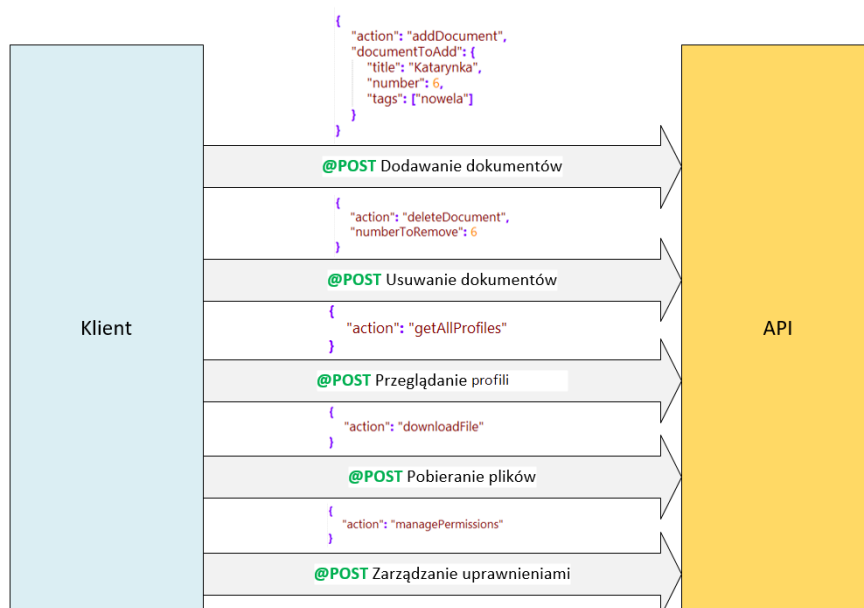
<sup>26</sup> Michał Wojtyna, Projektowanie RESTful API – dobre praktyki (videokurs), rdz.4, lekcja 1, portal Strefa Kursów: [https://strefakursow.pl/kursy/programowanie/projektowanie\\_restful\\_api\\_-\\_dobre\\_praktyki.html](https://strefakursow.pl/kursy/programowanie/projektowanie_restful_api_-_dobre_praktyki.html) (odwiedzono dn. 05.02.2020). Wykorzystano pliki źródłowe, udostępnione jako materiał do kursu.

<sup>27</sup> [https://github.com/pangeon/calc-bill/tree/master/src/main/java/pl/strefakursow/swamp\\_of\\_pox](https://github.com/pangeon/calc-bill/tree/master/src/main/java/pl/strefakursow/swamp_of_pox) (link aktywny dn. 06.02.2020) URL wskazuje na repozytorium na portalu github.com, gdzie zlokalizowane są kody źródłowe projektu calc-bill

wpływu na działanie aplikacji, która stanowi uzupełnienie części teoretycznej mojej pracy. W pakiecie **pl.cecherz.calcbill** znajduje się cały kod mojego autorskiego programu.

Serwis webowy, którego kod przedstawiłem na Ilustracji nr.7, działa prawidłowo. Spełnia zadania oczekiwane przez jego architekta, realizując w poprawny sposób zarządzanie danymi. Nie jest jednak wygodny w obsłudze ani czytelny. Zawiera instrukcje, które zostały zastosowane w zbyt ogólnym zakresie. Adnotacja *@PostMapping*, służąca do programowania zachowania metody POST protokołu HTTP, jest użyta w nieprawidłowym kontekście. POST używamy przy zapisie danych, gdy dana operacja wywołuje skutki wiążące dla użytkownika, np. rejestracja w sklepie internetowym. Metody nie da się odczytać ani zapisać poprzez po przez adres URI. W przykładzie wyłącznie ta jedna metoda odpowiada za operacje dotyczące: dodawania, usuwania, wyświetlania danych tekstowych, a także manipulacji uprawnieniami i pobieraniem plików. Uniknięto przydzielenia odpowiedzialności innym metodom HTTP m.in. GET i DELETE. Jeden serwis wykonuje wszystkie zadania, *de facto* mamy tylko jeden adres. Niewykorzystana zostaje możliwość mapowania zasobów, która stoi u podstaw modelu Richardson 'a.

Obsługa tak zaprojektowanego REST API jest trudna po stronie klienta. Podczas każdego żądania należy informować, w przesyłanych danych, o rodzaju operacji, jaką ma wykonać serwis. Znika konieczność komunikacji z aplikacją poprzez protokół HTTP, takie podejście mnoży problemy. W przypadku dużych serwisów WWW, niemożliwy jest tak permanentny brak organizacji zaplecza, w fazie rozbudowy.



**Ilustracja 8.** Schemat działania API opartego o antywzorzec Swamp of POX

## Level 1: RESOURCES

Poziom pierwszy modelu Richardson 'a kładzie nacisk rozdzielanie zasobów. Można to uzyskać dzieląc serwisy ze względu na rodzaj obsługiwanych danych. W przykładzie prezentowanym w poprzednim rozdziale jeden serwis ma zadanie obsługiwać obiekty klas dwóch różnych modeli – dokumentów oraz profili. Jest to błąd architektury oprogramowania, który w przypadku rozbudowy aplikacji sprawi trudności konserwacji kodu. Serwisy należy projektować względem modeli w architekturze pionowej, zbliżając się przy tym do fundamentalnej w programowaniu obiektowym zasady SRP<sup>28</sup>.

API, które zaprojektowałem dla aplikacji zarządzającej rachunkami posiada dwa serwisy, po jednym dla każdego modelu. W pierwszej fazie projektu, etapie, na którym tworzę kontroler bez warstwy persystencji danych, trudno będzie zauważyć sens wszystkich rozwiązań. API współpracujące z mapper 'em obiektowo-relacyjnym wygląda inaczej niż kontroler operujący na samych kolekcjach pozostających tylko w pamięci tymczasowej komputera. W przypadku wykorzystania baz danych konieczne jest odpowiednie obsłużenie relacji. Wyjaśnię to szerzej w rozdziale 3.

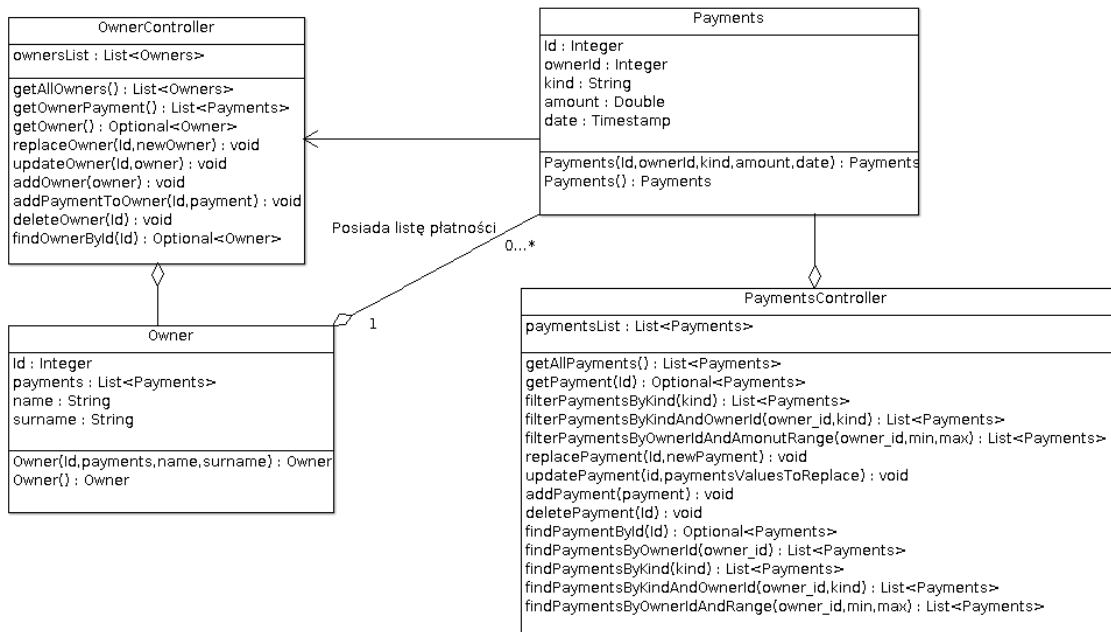
Zaczynam od utworzenia beanów<sup>29</sup>. Klasy zbudowane są na podstawie tabel umieszczonych na stronie 9. Klasa *Owner* zawiera obiekty klasy *Payments* w postaci listy. Jeden użytkownik aplikacji może posiadać wiele rachunków jemu przypisanych. Odwrotnie zaś jedna płatność nie może być wspólna dla dwóch różnych osób.

Serwisy operują na niezależnych od siebie kolekcjach danych. Wprowadzenie obiektów płatności przypisanego dla posiadacza *OwnerController* nie jest widoczne w danych obsługiwanych przez serwis *PaymentsController*. API w pakiecie **pl.cecherz.calcbill.controller.json.v1** nie jest spójne, zaprezentowanie jednak tego etapu produkcji jest niezbędne, aby wyjaśnić kluczowe pojęcia.

---

28 Single Responsibility Principle (tłum. zasada jednej odpowiedzialności) - więcej na ten temat można dowiedzieć się z książki Robert 'a C. Martin 'a pt. *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall 2002.

29 <https://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html> (odwiedzono dn. 07.02.2020)  
specyfikacja JavaBeans



**Ilustracja 9. Diagram UML - relacje klas modelu i kontrolera**

Odpytywanie, tworzenie, nadpisywanie i kasowanie danych odbywa się osobno dla płatności i posiadaczy. Chociaż pominięty został problem powiązania informacji, nie dochodzi do pomieszczenia w jednej klasie zależności dwóch różnych modeli danych bez logicznego uzasadnienia. Spełniony zostaje kontrakt poziomu pierwszego dla modelu Richardson 'a. Do konkretnych zasobów służą inne URI REST<sup>30</sup>.

## Level 2: HTTP METHODS

### Czasowniki HTTP

W mojej ocenie API REST-owe zyskuje dopiero na poziomie drugim, przy wykorzystaniu mapowania zasobów z odpowiednimi metodami HTTP<sup>31</sup>. Spring udostępnia w tym celu szereg prostych adnotacji: `@PostMapping`, `@GetMapping`, `@PutMapping`, `@PatchMapping`, `@DeleteMapping`. Z pomocą pięciu verbów, można wykonać typowe operacje na danych: create, read, replace, update i delete. Dla pozostałych metod HTTP wykorzystywanych w bardziej

30 Nadal wystarczy wykorzystanie tylko jednej metody HTTP. W przypadku pierwszego poziomu chodzi wyłącznie o separację zasobów i zmapowanie ich w postaci URI.

31 <https://sekurak.pl/protokol-http-podstawy/> (odwiedzono dn. 12.02.2020) Michał Sajdak – „Protokół HTTP – podstawy” (aktualizacja: 07.03.2019)

wyspecjalizowanym zakresie: HEAD, OPTIONS, TRACE oraz CONNECT; nie istnieją adnotacje do mapping 'u. Można je wykorzystać po odpowiednim przekonfigurowaniu aplikacji i użyciu adnotacji `@RequestMapping` z odpowiednimi parametrami.<sup>32</sup>

Przy pomocy operacji GET serwis odpytuje o konkretne elementy kolekcji, nadaje się ona do tworzenia w języku Java metod wyszukujących pewne określone zbiory danych. W projekcie mapuje kilka różnych metod, filtrujących zbiory danych na podstawie podanych parametrów. Wykorzystuje w tym celu adnotację `@RequestParam`<sup>33</sup>. Alternatywnym sposobem poruszania się po zbiorach zasobów jest wykorzystanie adnotacji `@PathVariable`<sup>34</sup>, która przekazuje do ciała metody kontrolera fragment adresu URI, by w ten sposób wyświetlić tylko określone dane.

**Tabela 1.** Opis metod do wyszukiwania elementów kolekcji przy wykorzystaniu HTTP.GET

| Metoda                                               | Kontroler          | Użycie <code>@RequestParam</code>                                                          | Użycie <code>@PathVariable</code>                                 |
|------------------------------------------------------|--------------------|--------------------------------------------------------------------------------------------|-------------------------------------------------------------------|
| <code>getOwner()</code>                              | OwnerController    | <code>.../owners?id={id}</code>                                                            | <code>.../owners/{id}</code>                                      |
| <code>getOwnerPayment()</code>                       | OwnerController    | <code>.../owners/payments?id={id}</code>                                                   | <code>.../owners/payments/{id}</code>                             |
| <code>getPayment()</code>                            | PaymentsController | <code>.../payments?id={id}</code>                                                          | <code>.../payments/{id}</code>                                    |
| <code>filterPaymentsByKind()</code>                  | PaymentsController | <code>.../payments?kind={kind}</code>                                                      | <code>.../payments/{kind}</code>                                  |
| <code>filterPaymentsByKindAndOwnerId()</code>        | PaymentsController | <code>.../payments?kind={kind}</code><br><code>&amp;owner_id={owner_id}</code>             | <code>.../payments/{owner_id}/</code><br><code>{kind}</code>      |
| <code>filterPaymentsByOwnerIdAndAmonutRange()</code> | PaymentsController | <code>.../payments?owner_id={owner_id}</code><br><code>&amp;min={min}&amp;max={max}</code> | <code>.../payments/{owner_id}/</code><br><code>{min}/{max}</code> |

Założyłem, że użytkownik będzie oczekiwał możliwości, posortowania i przefiltrowania płatności celem lepszego взгляду we własne wydatki. Zagregowane dane, będą wyświetlane tylko częściowo w zależności od parametrów przekazanych poprzez adres URI. Metody wymienione w tabeli umożliwiają:

- Zwracanie obiektu posiadacza płatności po unikatowym identyfikatorze.
- Zwracanie grupy obiektów płatności przypisanych do posiadacza.
- Zwracanie obiektu płatności po unikatowym identyfikatorze.

<sup>32</sup> <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-documentation> (odwiedzono dn. 08.02.2020) Więcej na ten temat można dowiedzieć z oficjalnej dokumentacji frameworka Spring Boot

<sup>33</sup> <https://docs.spring.io/spring/docs/3.1.x/spring-framework-reference/htmlsingle/spring-framework-reference.html#mvc-ann-requestparam> (odwiedzono dn. 08.02.2020) - Spring Framework: Reference Documentation: 16.3.3.3 Binding request parameters to method parameters with `@RequestParam`

<sup>34</sup> <https://docs.spring.io/spring/docs/3.1.x/spring-framework-reference/htmlsingle/spring-framework-reference.html#mvc-ann-requestmapping-uri-templates> (odwiedzono dn. 08.02.2020) - Spring Framework: Reference Documentation: 16.3.2.2 URI Template Patterns

- Zwracanie grupy obiektów płatności po rodzaju, będącego opisem pojedynczej płatności.
- Zwracanie grupy obiektów płatności po przekazanym obiekcie posiadacza oraz rodzaju.
- Zwracanie grupy obiektów płatności po przekazanym obiekcie posiadacza oraz określeniu granicy dolnej i górnej wartości zapisanych płatności.

Zaproponowane rozwiązanie jest jednym z wielu możliwych podejść do tematu, wszystko zależy od ilości i rodzaju danych, a także od przeznaczenia aplikacji. W przypadku dobrego REST API wymagane jest wykorzystanie w celach ekspozycyjnych danych, wielu mapowań metody GET w celu dostępu do zasobów. Takie podejście zgadza się z tym, co mówi specyfikacja protokołu HTTP<sup>35</sup>.

Napisany przeze mnie kontroler posiada pełną możliwość manipulacji nad danymi, odbywa się to ze zgodzie użytymi adnotacjami, które odpowiadają obsłudze metod HTTP. W przypadku POST, PUT, PATCH oraz DELETE korzystam z mapowania przy użyciu adnotacji *@PathVariable* oraz *@RequestBody*, która umożliwia przekazanie w ciele metody obiektu żądania. We wszystkich java 'owych metodach w kontrolerach, poza oznaczonymi adnotacją *@GetMapping*, wykorzystuje tylko pole id. Można oczywiście pokusić o bardziej wyrafinowane API do m.in. kaskadowego kasowania i nadpisywania grup danych, ale wydało mi się to zbędne. Brak edycji wielu rekordów jednocześnie stanowi rozsądne ograniczenie dla prostej aplikacji.

Zaletą kodu aplikacji, uwzględniającej podział odpowiedzialności obsługi na wiele metod protokołu HTTP, jest możliwość łatwej i elastycznej rozbudowy. Napisanie wielu wariantów metody oznaczonej adnotacją np. *@DeleteMapping* będzie wyglądało analogicznie jak stworzenie funkcji do obsługi metody GET, bez potrzeby usuwania reszty kodu. W tym kontekście ważne jest przeanalizowane zaprojektowanego modelu danych.

## Wersjonowanie

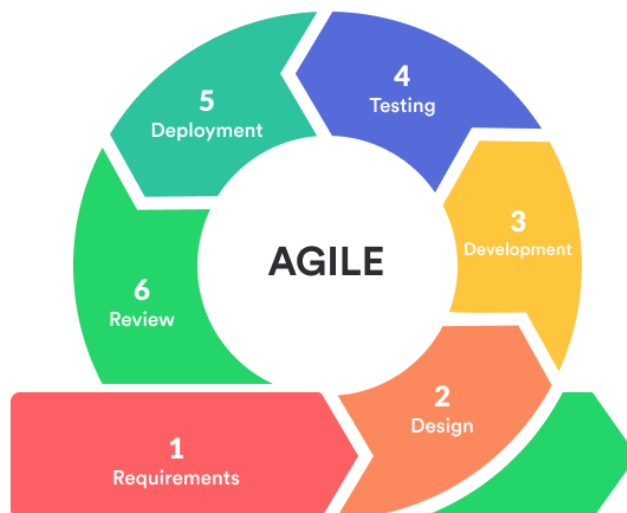
API na obecnym etapie rozwoju pakiet **pl.cecherz.calcbill.controller.json.v1**, uwzględnienia rozdzielanie zasobów, obsługuje najważniejsze metody protokołu HTTP. Pozwala na wykonywanie operacji CRUD, jest stosunkowo proste w utrzymaniu. Części programistom taki stan wystarczy. API jest wystarczająco kompletne, by zająć się tworzeniem klientów.

Cykl tworzenia oprogramowania to iteracyjno – przyrostowy proces ciągły. Aplikacja powinna dostosowywać się do zmieniających się oczekiwań użytkowników oraz wymagań stawianych przez

---

<sup>35</sup> <https://www.w3.org/Protocols/Specs.html> (odwiedzono 09.02.2020) - HTTP Specifications and Drafts.

rozwój technologiczny urządzeń. Kontroler REST w tym kontekście staje niezależną warstwą programu, stającą się sprostą dużej liczbie różnych klientów. Projektując back-end, nie trzeba zastanawiać się nad rodzajem hosta, który skonsumuje wygenerowane dane.



**Ilustracja 10.** Tworzenie oprogramowania w metodykach zwinnych.<sup>36</sup>

Podejście Richardson 'a stało się modne dzięki wysypowi różnego rodzaju urządzeń mobilnych, a co za tym idzie, powstaniu większej ilości oprogramowania. HTTP stanowi nadal uniwersalny fundament komunikacji sieciowej, co pozwala na wymianę danych pomiędzy aplikacjami napisanymi w różnych językach, działającymi na odmiennych platformach. Umożliwia to większe rozproszenie warstw oprogramowania, zapewnia spójność. Aplikacja mobilna napisana w Kotlinie<sup>37</sup>, aplikacja webowa na iOS-a<sup>38</sup> napisana przy użyciu frameworka Vapor<sup>39</sup> czy oprogramowanie na smartwatch 'u mogą wykorzystywać identyczne API.

Czym większa ilość programów wykorzystujących API tym większy narzut pracy nad jego rozwojem, gdyż zwiększa się nie tylko ilość użytkowników, ale też dochodzi zróżnicowanie konsumentów. Coś, co wydaje się zbędne w aplikacji na konsolę Xbox<sup>40</sup> okazać może się konieczne w

36 <https://softmahal.com/software-development/agile-software-development/> (odwiedzono dn. 11.02.2020) Agile Software Development, 11.12.2019

37 <https://kotlinlang.org/docs/kotlin-docs.pdf> (odwiedzono dn. 11.02.2020) Kotlin Language Documentation

38 <https://developer.apple.com/ios/> (odwiedzono dn. 11.02.2020) Dev Center at Apple Developer Connection

39 <https://docs.vapor.codes/3.0/> (odwiedzono dn. 11.02.2020) Vapor Documentation, autorzy: m.in. Tanner Nelson, Logan Wright.

40 <https://www.xbox.com/pl-PL> (odwiedzono dn. 11.02.2020) Strona oficjalna

<https://docs.microsoft.com/en-us/windows/uwp/xbox-apps/reference> (odwiedzono dn. 11.02.2020) Xbox Device Portal REST API

czytniku Kindle<sup>41</sup>. W praktyce oczywiście multi-platformowość ma swoje ograniczenia warto jednak mieć świadomość istnienia uniwersalnego rozwiązania, nawet, jeśli nie ma konkretnego klienta na daną platformę nie ma nic nie stoi na przeszkodzie by został napisany, gdy istnieje odpowiednie API.

Projektanci API, w trakcie jego rozwoju, wdrażając nowe funkcjonalności, nie mogą zapominać kompatybilności ze starymi klientami, stąd potrzeba wersjonowania. Tworząc nowe ulepszone API dla konkretnej aplikacji nie należy usuwać starego, „pod którego” zostały napisane konkretne aplikacje klienckie. Nie da się stworzyć oprogramowania odpornego na wszystkie możliwe błędy, rozproszenie warstw aplikacji wymusza wręcz ostrożność. Klient powinien mieć możliwość stopniowej migracji do nowej wersji kontrolera.

Funkcjonalności związane z nagłówkami, statusami HTTP, negocjacji zawartości wprowadzam w API wersji 2, pakiet **pl.cecherz.calcbill.controller.json.v2**. W przypadku małej aplikacji, która nie posiada jeszcze napisanego widoku, tworzenie wersji może wydać się pracą nadmiarową. Celem mojej pracy jest jednak przede wszystkim pokazanie dobrych praktyk. Nie jestem w stanie przewidzieć, kiedy użycie prostszego kontrolera okaże bardziej konieczne, nie wiedzą też tego programiści dużych aplikacji webowych.

W celu zachowania poprawnego działania aplikacji konieczne jest poinformowanie Spring 'a o wykorzystaniu różnych kontrolerów, można to zrobić na kilka sposobów. Ja w projekcie wykorzystuję adnotację *@Component*, z wartością tekstową w postaci nazwy, przed nazwą klasy. Stąd *[...]ControllerJSON-V1* oraz *[...]ControllerJSON-V2*. Nazwy klas dla następnych wersji pozostawiam niezmiennie. Dla kolejnych wcieleń API charakterystyczny będzie fragment URI, wartość w adnotacji *@RequestMapping*, informujący o jego wersji - przykładowo */api/v2/owners* dla klasy *OwnerController* w pakiecie **pl.cecherz.calcbill.controller.json.v2**.

Zmiana ścieżek kolejnych wersji nie jest uniwersalną techniką, w ten sposób przenosimy większą odpowiedzialność za obsługę API na klienta. Lepszym rozwiązaniem jest wykorzystanie HEADER 'ów HTTP. Opowiadam o tym w kolejnym podrozdziale.

## Nagłówki HTTP

Protokół HTTP określa zasady komunikacji pomiędzy klientem a serwerem. Łatwo to przedstawić za pomocą prostego przykładu, analogii z życia.

W celu nadania listu należy wrzucić go do skrzynki pocztowej, poprawnie zaadresowaną kopertę odbiera poczta. List trafia do węzła ekspedycyjno-rozdzielczego, gdzie zostaje sprawdzony i

---

41 Produkt firmy Amazon, popularny czytnik plików w formacie m.in. w formacie MOBI.

<https://rainforestapi.com/docs> (odwiedzono dn. 11.02.2020) RainForest API - Interfejs API danych produktów Amazon.

przekazany do kolejnego punktu pocztowego. Stamtąd listonosz bierze go i przynosi do punktu wskazanego na kopercie. Aby system działał sprawnie, musi istnieć ścisły podział ról i zasad komunikacyjnych, muszą istnieć także scenariusze w razie pojawienia sytuacji nieoczekiwanych. Przepływ wiadomości jest kontrolowany, źle zaadresowane listy, koperty bez znaczka nie trafiają do odbiorców, zostaną zwrócone do nadawców. Nadawca listu nie zna detali systemu logistycznego poczty, nie potrzebuje ich by nadać wiadomość. Umie jednak poprawnie napisać adres na kopercie, wie jaki znaczek kupić, zna miejsce ulokowania skrzynek. Oczywiście sam usługodawca też może się pomylić, list zaginać i nie trafić do celu.

W przypadku usługi, jaką Internet, HTTP odpowiada zbiorowi reguł komunikacyjnych poczty. List można tłumaczyć, jako obiekt żądania klienta, przeglądarki WWW. Punkty stanowiące elementy systemu logistycznego poczty zaś, jako urządzenia pośredniczące w przekazaniu wiadomości: serwery, routery i inne elementy sieciowe pozwalające na sprawne dotarcie żądania. W tym kontekście nie liczy się treść, ciało przekazywanego dokumentu, list może zawierać tekst w obcym języku, załączone zdjęcia lub po prostu być pusty. Poczta interesuje koperta i znaczek, serwer zaś będzie oczekiwał nazwy metody HTTP oraz treści zawartej w nagłówku, która pozwoli lepiej określić cechy zwracanego zasobu.

List zwykle trafi do odbiorcy, w pewnej liczbie przypadków, mogą jednak pojawić się problemy: źle zaadresowana koperta lub błąd ze strony usługodawcy. W przypadku pierwszego, jeśli nadawca nie zapomni wpisać opcjonalnego adresu zwrotnego, koperta z zawartością wróci do niego. Dowie się on, że popełnił błąd, źle zaadresował kopertę. W wypadku błędu ze strony usługodawcy, wszystko zależy od wewnętrznej polityki. Programowanie reakcji serwera na niewłaściwe żądanie: takie, którego nie rozumie, gdyż nie potrafi na nie odpowiedzieć, powinno uwzględniać generowanie czytelnych komunikatów w odpowiedzi na błędy. Użytkownik aplikacji otrzymujący poprawne sygnały, ze strony serwera, na którym znajduje się konkretne API, będzie mógł zachować lepiej w przypadku sytuacji, gdy coś pójdzie nie tak.

Nagłówek HTTP, którego zdefiniowanie jest opcjonalne dla obiektu żądania, zawiera listę właściwości, metadanych, które pozwalają z większą precyzją obsłużyć przekazane do serwera dane. Serwer odpowiadając klientowi może wykorzystać dane nagłówka do lepszej integracji aplikacji, dostosowaniu jej do wymagań użytkownika. Zawierają one m.in informacje o formacie, statusie odpowiedzi czy dacie.



**Ilustracja 11.** Obiekt odpowiedzi i żądania w HTTP<sup>42</sup>

Lista możliwych parametrów nagłówka, jest równie długa jak lista kodów odpowiedzi. Jest dostępna ona w specyfikacji Mozilla Developer Center<sup>43</sup>.

W projekcie wykorzystuje nagłówki, w celach informacyjnych dla klientów API oraz przy obsłudze wyjątków. API w wersji 2, pakiet **pl.cecherz.calcbill.controller.json.v2**, jest bardziej złożone i elastyczne.

Wszystkie funkcje klas kontrolerów *OwnerController* oraz *PaymentController* zwracają obiekty klasy *ResponseEntity*<sup>44</sup>:

[...] Obiekty *ResponseEntity*, oprócz samego obiektu, który zostanie skonwertowany i zwrócony jako reprezentacja zasobu, zawierają metadane (takie jak nagłówki i kody statusu) dotyczące odpowiedzi.<sup>45</sup>

Dzięki temu możliwe zwracanie danych kolekcji opakowanych w obiekt klasy, który realizuje założenia protokołu HTTP, wykorzystuje jego komunikacyjny potencjał. Ma to kluczowe znaczenie przy obsłudze błędów. Przesyłanie zasobów pomiędzy klientem a serwerem to tylko część funkcjonalności dobrego API. Równie istotne są metadane pod spodem, które pozwalają zapewnić aplikacji stabilność. Klient musi mieć od dyspozycji backend, który spełni jego wymagania, pozwoli reagować na różne problematyczne sytuacje.

42 [https://developer.mozilla.org/pl/docs/Web/HTTP/HTTP\\_wiadomosci\\_og%C3%B3lne](https://developer.mozilla.org/pl/docs/Web/HTTP/HTTP_wiadomosci_og%C3%B3lne) (odwiedzono dn. 17.02.2020) „HTTP - wiadomości ogólne”, 27 listopada 2019, autorzy MDN

43 <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers> (odwiedzono dn. 17.02.2020) „HTTP headers”, ostatnio zmodyfikowany: 10 lutego 2020 r, autorzy MDN

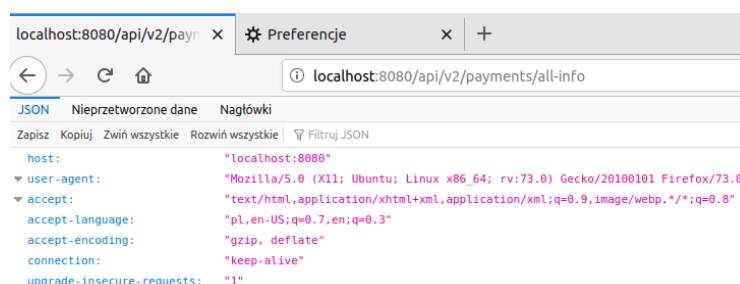
44 <https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/ResponseEntity.html> (odwiedzono dn. 17.02.2020) - Spring Framework 5.2.3.RELEASE API - API documentation for the Spring Framework.

45 Craig Walls, „Spring w Akcji”, tłum. Mirosław Gołda, Piotr Rajca, wydanie IV, wyd. Helion, Gliwice 2015 (str. 465, rdz. 16)

Zaprezentowane przeze mnie API w wersji drugiej, ze strony projektowej, może wydawać się dalekie od ideału. Spring MVC<sup>46</sup> pozwala na stworzenie kodu aplikacji z wykorzystaniem większej ilości abstrakcji. Kod w pakiecie **pl.cecherz.calcbill.controller.json.v2** nie zawiera zależności klas narzędziowych takich jak np. *RestTemplate*<sup>47</sup>, pozwalających na optymalizację pracy programisty, umożliwiających na prosty opis interakcji pomiędzy zasobami REST. Nie wykorzystuje możliwości wynikających z separacji warstwy obsługi błędów, do którym przeznaczona jest m.in. adnotacja *@ExceptionHandler*<sup>48</sup>. Temat dobrej architektury kodu REST API stanowi osobne, szerokie zagadnienie, w niniejszym opracowaniu skupiam się przede wszystkim na łatwym wykorzystaniu modelu Richardson'a w praktyce. Stąd proste rozwiązania, które z punktu widzenia narzędzia, jakim jest framework Spring, są niskopoziomowe. W przypadku dużych serwisów mało efektywne.<sup>49</sup>

Aby pokazać dane przezywane w nagłówkach, w wersji drugiej API, wprowadzam klasę o nazwie *HTTPHeaderUtils*, której znaczenie, w kontekście zwiększenia funkcjonalności całego programu do zliczania rachunków, jest niewidoczne. Klasa pozwala na odczytanie przez klienta, domyślnych wartości danych nagłówka. Metoda *getBaseUrl* zwraca zaś bazowy adres URI dla aplikacji. Pobrane przez informacje mogą być przydatne w celach diagnostycznych.

Funkcje w opisanej powyżej klasie, pobierają parametr oznaczony adnotacją *@RequestHeader*<sup>50</sup>. Instrukcja pozwala na zdefiniowanie w przekazywanym nagłówku poszczególnych informacji, takich jak: Accept, Accept-Encoding, Accept-Language, Cache-Control, Expires, Host, User-Agent, w zależności od potrzeb.



**Ilustracja 12.** Efekt działania metody *showHeaders*

46 <https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html> (odwiedzono dn. 18.02.2020) - Część projektu Spring Framework.

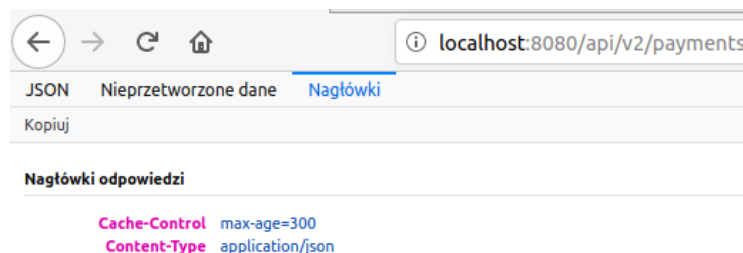
47 [https://www.tutorialspoint.com/spring/spring\\_web\\_mvc\\_framework.htm](https://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm) (odwiedzono dn. 18.02.2020)

48 <https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc> (odwiedzono dn. 18.02.2020) Paul Chapman - „Exception Handling in Spring MVC”, (data utworzenia: 1.11.2013, aktualizacja: 04.2018)

49 Publikacja Craig'a Walls, „Spring w Akcji”, tłum. Mirosław Gołda, Piotr Rajca, wydanie IV, wyd. Helion 2015, Gliwice, rdz. „Tworzenie API modelu REST przy użyciu Spring MVC” opisuje wyczerpująco możliwości Spring MVC w kontekście REST API.

50 <https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestHeader.html> (odwiedzono dn. 18.02.2020)

Nagłówki w moim projekcie podniosły funkcjonalność metod odpowiedzialnych na przekazywanie wszystkich obiektów kolekcji. Dzięki użyciu Cache-Control w metodach *getAllOwners* i *getAllPayments* udało mi się pozbyć oczekiwania, na pobranie wszystkich danych. Za każdym razem, gdy odpytane zostanie API pierw zostanie sprawdzona pamięć podręczna. Opisane rozwiązanie znacznie zwiększa prędkość działania aplikacji webowych, pozwala ograniczyć obciążenie łącza internetowego, a co za tym idzie serwera.



**Ilustracja 13.** Efekt działania metod *getAllOwners* oraz *getAllPayments*

Wykorzystanie nagłówków, znacznie zwiększa narzut pracy przy pisaniu aplikacji. W wielu przypadkach prostych zastosowań API można zrezygnować z wykorzystania obiektów *ResponseEntity* i całej gamy abstrakcji oferowanej przez framework. Wykorzystanie zaproponowanego przeze mnie API w wersji pierwszej jest wystarczającym fundamentem do stworzenia trójwarstwowej aplikacji z widokiem w HTML-u i zapisem do relacyjnej bazy danych. To jednak zdecydowanie za mało dla dużego komercyjnego projektu. Wraz ze wzrostem skali serwisu webowego rośnie ilość potencjalnych problemów, pojawiają się ograniczenia związane z infrastrukturą i mocą urządzeń.

## Content Negotiation

Negocjacja treści jest możliwa do zrealizowania dzięki wykorzystaniu nagłówków HTTP. Stanowi to kolejne uzupełnienie drugiego poziomu dojrzałości według modelu Richardson 'a. Głównym powodem popularności modelu REST jest jego elastyczność. Projektując API warto wziąć pod uwagę różnorodność klientów, a co za tym idzie, wymagania dostarczenia wielu reprezentacji zasobu.

Nie trzeba z góry implementować obsługi wszystkich możliwych formatów, w odpowiedzi na żądania. API powinno ewoluować, kształtować się poprzez zmieniające się potrzeby w kolejnych cyklach rozwoju. Spring umożliwia zwrócenie poprzez ten sam URI oraz metodę HTTP różnych formatów (m.in. text/plain, application/xml, text/html, application/json, image/gif czy image/jpeg) tego samego zasobu, w tym celu potrzebna jest zmiana tylko jednego parametru *produces* dla

*@GetMapping*. Takie działanie jest proste, pozwala na szybkie wdrożenie nowych funkcjonalności API. Klient wysyła żądanie do serwera oczekując konkretnej reprezentacji zasobu, jej rodzaj definiuje przy pomocy właściwości *Accept* nagłówka HTTP. Zwracany jest zasób o wartości nagłówka określonej w parametrem *Content-Type*<sup>51</sup>.

Umieściłem w API tylko jedną funkcję, która zwraca inny wariant tego samego zasobu. W klasie *OwnerController* dwie metody korzystają z czasownika GET i mają taki sam adres URI, są to:

- *getAllOwners*, która zwraca w formacie JSON elementy kolekcji typu *Owner* (id, płeć, imię, nazwisko)
- *getHTMLOwnersNamesandSurname*, która zwraca w formie strony HTML-a opakowane elementy kolekcji typu *Owner* (imię i nazwisko)

Metody wywoływane są w zależności od informacji zdefiniowanych w nagłówku, kontroler domyślnie zwraca JSON-a, po przesłaniu zaś w nagłówku: *Accept: text/html*, pełną stronę HTML. W projekcie wykorzystałem możliwość stworzenia atrakcyjnego efektu, poprzez przekazanie w parametrów pozwalam na zmianę wyglądu danych wyświetlanych przez przeglądarkę. Dodałem możliwość określania poprzez pasek adresu, koloru i wielkości pisma.

Metoda *getHTMLOwnersNamesandSurname* zwraca cały gotowy widok. Wykonałem ją by pokazać elastyczność wykorzystania możliwości frameworka. Chociaż rozwiązanie wydaje się pomysłowe, narusza zasady REST API. Metoda zwraca więcej niż reprezentację zasobu, chociaż daje oryginalny szybki efekt jest nieużyteczna. Klientom będzie chodziło wyłącznie o czyste dane. Style, dodatkowe znaczniki strukturalne są w tym kontekście elementami nadmiarowymi. Według wzorca MVC, część kodu aplikacji odpowiadającej za wygląd musi znajdować się osobno. Takie działanie umożliwia skalowanie kontrolera w oderwaniu od frontend 'u. Programista powinien dążyć do stanu, w którym jedna funkcja wykonuje jedną operację.<sup>52</sup> Jest to jedna z najpopularniejszych heurystyk, która w przypadku REST API zyskuje na znaczeniu. Chodzi o stworzenie mapy zasobów, po której przy pomocy ścieżek URL możemy poruszać się w sposób jednoznaczny, zwracanie elementów, które nie należą ściśle do profilu danych jest błędem.

---

<sup>51</sup> Lista dostępnych typów zdefiniowanych poprzez standard MIME jest naprawdę obszerna: <https://www.iana.org/assignments/media-types/media-types.xhtml> (odwiedzono 19.02.2020)

2020-02-18, Expert Review for Vendor and Personal Trees - Ned Freed, Murray Kucherawy

<sup>52</sup> Robert C. Martin „Czysty Kod - podręcznik dobrego programisty”, tłum. Paweł Gonera, wyd. Helion, Gliwice 2014 (G.30, s.317)

Możliwość odpowiedzi serwera na żądania klienta w wielu formatach rozwiązuje problemy niezgodności. Sprawia, że API staje się zunifikowane; backend ma możliwość integracji z wieloma klientami na różnych urządzeniach. API jednak nie może przejmować odpowiedzialności prezentacji danych. Jest to domena klienta.

## HTTP Headers

Dobrze napisane API posiada obsługę statusów HTTP. Zwraca klientom łatwe w zweryfikowaniu informacje o przebiegu zapytań przesłanych serwerowi. Nie ma obowiązku dbania o kody odpowiedzi są one podobnie jak część nagłówkowa, opcjonalnymi metadanymi. Ignorowanie jednej detali komunikacji opartej o protokół http może nieść poważne konsekwencje.

Chociaż użytkownik podczas korzystania z aplikacji webowej nie będzie śledził, komunikacji serwera i klienta w sposób jawny, nie oznacza to, że nigdy nie zobaczy strony z informacją o błędzie. Jeśli po stronie serwera wystąpi błąd lub błąd spowoduje klient, a nikt nie zadba o strony informacyjne dla konkretnych statusów, użytkownik otrzyma informację niejasną, nie będzie umiał sam poradzić sobie z sytuacją nieprzewidzianą. Popularny przykład stanowi tu błąd 404, można na niego natrafić w przypadku, gdy pod konkretną domeną odwołamy się do podstrony, której już nie ma lub nigdy nie było. Dobrą praktyką stosowaną przez popularne serwisy jest tworzenie ładnych wizualnie stron HTML dla konkretnych błędów.



**Ilustracja 14.**

*a) Przykład serwisu bez spersonalizowanej strony błędu dla statusu 404;*

*b) przykład serwisu, który ją posiada.*

Ilustracja nr. 14 uwidacznia możliwe różnice w czytelności. Serwis webd.pl informuje o przyczynie niepowodzenia bez korzystania ze słownictwa technicznego, pozwala użytkownikowi wykorzystać linki.

Statusy HTTP stanowią integralny element komunikacji internetowej. Choć są ukryte dla użytkowników programu, nie mogą zostać zupełnie pominięte. Działania REST API można doświadczyć poprzez program Postman, w którym można prześledzić, z czego dokładnie składa się odpowiedź HTTP ze strony aplikacji obsługiwanej przez serwer lokalny. Aplikacja bez dodatkowej konfiguracji: użycia odpowiednich adnotacji lub zmiany zwracanego typu, zareaguje w sposób domyślny. W przypadku metod z czasownikami POST, PUT, PATCH serwer zwróci zawsze kod odpowiedzi 200 (OK). Tak dzieje się w API w wersji 1.

Poprawnym kodem odpowiedzi wydaje się status 201 (CREATED), informujący o utworzeniu nowej porcji danych. W przypadku metod służących do częściowego i całościowego nadpisywania danych może przydać się kod odpowiedzi 404 (NOT FOUND) informujący o braku wyszukiwanego zasobu. Logicznym jest, że nie można zaktualizować rekordu, którego nie ma. API w wersji drugiej klient otrzyma status HTTP w zależności od finalnego efektu operacji wykonanych przez metody zapisu. Utworzenie obiektów zarówno posiadacza jak i płatności, zamiast 200 (OK) zwróci 201 (CREATED). W przypadku odwołania się do elementu listy, którego id nie jest znane program zwróci 404 (NOT FOUND).

Metody poprzedzone adnotacją *@DeleteMapping* zwracać będą status 200 (OK) w przypadku poprawnego usunięcia obiektu z pamięci lub 404 (NOT FOUND) w przypadku, gdy rekordu nie da się usunąć, gdyż nie znajduje się już w indeksie.

W klasie *OwnerController* znajduje się jedna metoda, której obsługę zrealizowałem nietypowo. Mając na uwadze fakt, iż zwrócić może ona pustą kolekcję. Funkcja *getOwnerPayment* służy do odnajdywania konkretnych posiadaczy i zwracania listy przypisanych im płatności. Dopuszczałem istnienie rekordu, który może posiadać właściwość płatności w postaci pustego zbioru. Lista bez dodanych obiektów, będzie jednak istniała w pamięci, stąd odpowiedź serwera aplikacji w tym wypadku 204 (NO CONTENT).

Pozostałe funkcję obu kontrolerów obsługiwane przez HTTP GET zwracać będą kody odpowiedzi 200 (OK) lub 404 (NOT FOUND).

Wykonane przeze mnie API w projekcie nie uwzględnia wszystkich możliwych scenariuszy błędów i stanowi jedynie wstęp. Zupełnie inaczej będzie wyglądać API, jeśli dodamy kontrolę dostępu

z wykorzystaniem Spring Security<sup>53</sup>. Moduł odpowiedzialny za autoryzację użytkowników aplikacji, stwarza konieczność obsługi dodatkowych statusów, przede wszystkim 401 (UNAUTHORISED).

Nie zająłem się też problem weryfikacji poprawności formatu danych wysyłanych przez klienta, co w przypadku dobrego REST API może mieć duże znaczenie. W tym wypadku konieczna jest obsługa kodów odpowiedzi 406 (NOT ACCEPTABLE) oraz 415 (UNSUPPORTED MEDIA TYPE).

Ilość kodów odpowiedzi jest ogromna, nie istnieje jednak konieczność, nawet przy komercyjnych aplikacjach, obsługi przez aplikację wszystkich.<sup>54</sup> Listę najważniejszych przedstawiam poniżej w tabelce.

**Tabela 2.** Najważniejsze kody odpowiedzi HTTP

| Kod odpowiedzi | Opis tekstowy         | Typ          |
|----------------|-----------------------|--------------|
| 200            | Ok                    | Success      |
| 201            | Created               | Success      |
| 204            | No content            | Success      |
| 300            | Multiple Choices      | Redirection  |
| 301            | Moved Permanently     | Redirection  |
| 302            | Found                 | Redirection  |
| 304            | Not Modified          | Redirection  |
| 307            | Temporary Redirect    | Redirection  |
| 400            | Bad Request           | Client Error |
| 401            | Unauthorized          | Client Error |
| 403            | Forbidden             | Client Error |
| 404            | Not Found             | Client Error |
| 410            | Gone                  | Client Error |
| 500            | Internal Server Error | Server Error |
| 502            | Not Implemented       | Server Error |
| 503            | Service Unavailable   | Server Error |
| 504            | Gateway Timeout       | Server Error |
| 510            | Not Extended          | Server Error |

53 <https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html> (odwiedzono dn. 18.02.2020) - Część projektu Spring Framework. Służy do zapewnienia bezpieczeństwa aplikacji webowej.

54 [https://pl.wikipedia.org/wiki/Kod\\_odpowiedzi\\_HTTP#cite\\_note-2](https://pl.wikipedia.org/wiki/Kod_odpowiedzi_HTTP#cite_note-2) (odwiedzono dn. 20.02.2020) Wikipedia „Kod odpowiedzi HTTP” - Ciekawy przykład stanowi Status 418 - I’m a teapot „Jestem czajnikiem” – tzw. easter egg. Zdefiniowany w 1998. Obecnie nie jest implementowany do serwerów HTTP, ale znane są takie przypadki.

## Level 3: HYPERMEDIA CONTROLS

### HETEOAS

Dobry interfejs API prowadzi klienta przez całą aplikację. Serwer powinien dostarczać informacji, jakie operacje może wykonać on w ramach backend 'u. HATEOAS<sup>55</sup> (Hypermedia as the Engine of Application State) można uznać za rodzaj drogowskazu dla klienta aplikacji.

W pełni dojrzałe API, poziom trzeci według modelu Richardson 'a, potrafi poinformować klienta o lokalizacji powiązanych ze sobą zasobach oraz metodach HTTP udostępnionych przez konkretny endpoint. API bazujące na rozwiązaniu HATEOAS zwraca klientom linki umożliwiające eksplorację zbiorów kolekcji. Chodzi o powiązanie danych biznesowych oraz hiperłączy w taki sposób, by programista, a dalszej kolejności klient aplikacji nie musiał znać szczegółów (lokalizacji zasobów URI), by mógł wywołać konkretną operację po stronie serwera.

Zmniejsza to odpowiedzialność oprogramowania hostów wykorzystujących API, pozwala na rezygnację z części logiki biznesowej, czyniąc aplikacje klientom bardziej odporne na zmiany po stronie serwera. Wygodnie jest korzystać z interfejsu programistycznego, który podpowiada deweloperowi, jakich metod ma używać dla określonych zbiorów kolekcji. API poziomu trzeciego dostarcza mapę zdefiniowanych relacji pomiędzy różnymi obszarami interfejsu. Pozwala to na łatwiejszy dostęp do dużej ilości rozproszonych danych. Rozwiązanie sprawdza się w przypadku serwisów mających dużą liczbę usług.

W przypadku każdej odpowiedzi klient otrzymuje zbiór linków, dzięki którym może decydować, co zrobić dalej. Przykładowo. W przypadku operacji zwracającej całą kolekcję danych określonego typu, przekazany w odpowiedzi zostanie JSON+HAL<sup>56</sup>, z linkami w postaci self-referencji, a także szablonami sparametryzowanych URI, umożliwiającymi filtrowanie grupowe czy odwoływanie się konkretnych do jednostek, za pomocą identyfikatorów. Kolekcja może również posiadać linki do zasobów zewnętrznych, łącząc się w ten sposób z całym Internetem.

W przypadku napisania klienta, który będzie potrafił poruszać po odpowiednio ponazywanych relacjach, zgodnie ze specyfikacją HAL znika problem kompatybilności w przypadku relokacji zasobów. Wystarczy znajomość tylko jednego endpointa by trafić wszystkich miejsc w API. Dobry przykład, który daje obraz poprawnie napisanego API z wykorzystaniem HATEOAS znajduje się pod adresem:

---

55 <https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven> (odwiedzono dn. 27.02.2020)  
„REST APIs must be hypertext-driven” - posted by Roy T. Fielding under software architecture, web architecture  
(aktualizacja: 20.10.2008)

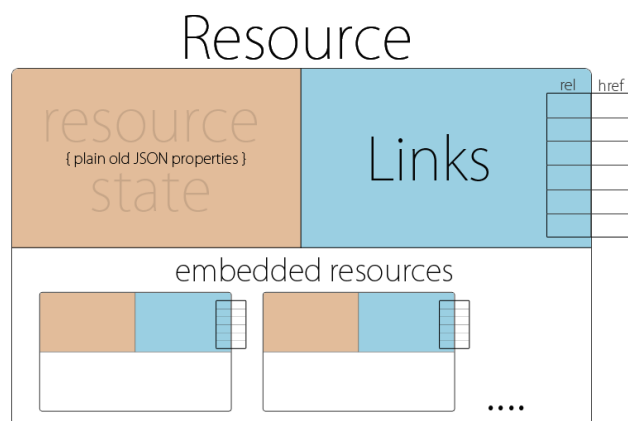
56 [http://stateless.co/hal\\_specification.html](http://stateless.co/hal_specification.html) (odwiedzono dn. 27.02.2020) „HAL - Hypertext Application Language”  
Mike Kelly (utworzenie 2011-06-13, aktualizacja 2013-09-18)

<http://jeti.org.pl/wp-json> (przykład bloga postawionego z użyciem CMS Wordpress). Po wpisaniu adresu zwracany jest cały szereg informacji, które tylko w niewielkiej części dotyczą konkretnej strony. W przeważającej części jest to zbiór linków i dodatkowych informacji pożytecznych ze względu na całość API. Wchodząc głębiej możliwe jest uzyskanie informacji i hiperłączy, znaczących dla użytkowników bloga. Przedstawiłem to na początku pracy - ilustracja nr.2 (strona 8). Widać, że interfejs programistyczny Wordpress 'a spełnia wszystkie poziomy dojrzałości Richardson 'a.

Co warto uwagi mamy tu do czynienia z backend 'em napisanym w języku skryptowym PHP. W przypadku modelu REST technologia do stworzenia API jest wyborem dewelopera, który swobodnie realizuje określone założenia architektury. Udało mi się przeczytać na jednym z portali o korzyściach wynikających z użycia API Wordpressa:

*Główną zaletą jest brak konieczności znajomości języka PHP. Nie korzystamy tu z żadnych funkcji, pętli etc. Jedyne, co musimy podać to odpowiedni endpoint, który zwróci nam żądane dane.<sup>57</sup>*

Rest API jest czymś niezależnym, narzędzie w jakim zostało stworzone w żaden sposób nie określa sposobu działania po stronie klienta. Wspólnym językiem są założenia protokołu komunikacyjnego HTTP.



**Ilustracja 15.** Ilustracja struktury HAL<sup>58</sup>

## Linki w Springu

<sup>57</sup> <https://devszczepaniak.pl/rzut-okiem-na-wordpress-rest-api/> (odwiedzono 27.02.2020) „Rzut okiem na WordPress REST API”

<sup>58</sup> [http://stateless.co/hal\\_specification.html](http://stateless.co/hal_specification.html) (odwiedzono 27.02.2020)

W projekcie pokazuje jak w praktyczny sposób wykorzystać mechanizmy HATEOAS by zwiększyć funkcjonalność aplikacji. API w wersji trzeciej: **pl.cecherz.calcbill.controller.json.v3** wykorzystuje bibliotekę Spring HATEOAS, która jest pobierana dzięki dodaniu dodatkowych zależności w postaci startera do pliku konfiguracyjnego Maven 'a'<sup>59</sup>. Aby móc realizować założenia trzeciego poziomu REST API Richardson 'a wystarczy dodanie:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
    <version>2.1.4.RELEASE</version>
</dependency>
```

Wpis pozwala na pobranie odpowiednio skonfigurowanych narzędzi, które integrują się z całym frameworkiem.<sup>60</sup> W celu zwiększenia klarowności klasy kontrolera przenieśliśmy obsługę linków, do klasy narzędziowej *HateoasBuilder*, pakiet: **pl.cecherz.calcbill.controller.json.rest\_utils**. Program działa poprawnie realizując powierzone mu zadania, jest to jednak prosty przykład. W wersji produkcyjnej aplikacji należałoby pomyśleć o nieco lepszej organizacji kodu, wykorzystaniu dodatkowych warstw abstrakcji oferowanych przez Spring 'a.

Kontroler w wersji 3 rozszerza funkcjonalności wersji 2. Zmiany występują tylko po stronie funkcji opatrzonej adnotacją *@GetMapping*, gdyż chodzi tu o rozszerzenie formy zwracanych danych. Linki generowane są w sposób automatyczny z wykorzystaniem mechanizmu refleksji. Nie trzeba przysyłać informacji o chęci ich utworzenia, nie są nigdzie zapisywane.

Aby nie stracić możliwości obsługi header 'ów, nie zmieniłem typu *ResponseEntity* zwracanego przez metody. Przekształciłem natomiast klasy kolekcji, które ten typ opakowuje:

**Tabela 3.** Opis typów w kolejnych wersjach API

| API wersja 2                                             | API wersja 3                                                                          |
|----------------------------------------------------------|---------------------------------------------------------------------------------------|
| <code>ResponseEntity&lt;List&lt;Owner&gt;&gt;</code>     | <code>ResponseEntity&lt;CollectionModel&lt;EntityModel&lt;Owner&gt;&gt;&gt;</code>    |
| <code>ResponseEntity&lt;Optional&lt;Owner&gt;&gt;</code> | <code>ResponseEntity&lt;EntityModel&lt;Owner&gt;&gt;</code>                           |
| <code>ResponseEntity&lt;List&lt;Payments&gt;&gt;</code>  | <code>ResponseEntity&lt;CollectionModel&lt;EntityModel&lt;Payments&gt;&gt;&gt;</code> |

59 <https://docs.spring.io/spring-hateoas/docs/1.1.0.BUILD-SNAPSHOT/reference/html/> (odwiedzono dn. 28.02.2020)  
„Spring HATEOAS - Reference Documentation”, Oliver Gierke, Greg Turnquist, Jay Bryant  
Version 1.1.0.BUILD-SNAPSHOT, 2020-02-11

60 <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter> (odwiedzono dn. 28.02.2020)  
Spring Boot Reference Documentation, Phillip Webb, Dave Syer, Josh Long, Stéphane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, Sébastien Deleuze, Michael Simons, Vedran Pavić, Jay Bryant, Madhura Bhavne - 3.1.5.  
Starters

|                                    |                                       |
|------------------------------------|---------------------------------------|
| ResponseEntity<Optional<Payments>> | ResponseEntity<EntityModel<Payments>> |
|------------------------------------|---------------------------------------|

*CollectionModel* jak łatwo się domyśleć będzie użyteczne przy próbie zwracania wielu elementów kolekcji. *EntityModel*, gdy chodzi o zwrócenie reprezentacji jednego obiektu.<sup>61</sup> Dzięki wykorzystaniu metod *linkTo* oraz *methodOn* klasy *WebMvcLinkBuilder*<sup>62</sup> udało mi się zaprogramować API by dostarczało klientowi, różne typy linków:

- **self-referencje:** link do zasobu, który aktualnie został zwrócony;
- **odwołania do zbiorów:** link do zasobu, który zwraca całą kolekcję tą, w której znajduje się element, który stał się celem zapytania;
- **templates:** linki oddzielające odpowiedzi na pytania jak tworzyć sparametryzowane zapytania, posiadają miejsca w nawiasach, które należy uzupełnić konkretnymi danymi.

Pozwalają one swobodne poruszanie się po API, pozwalają na szybkie testowanie zapytań wyszukujących elementy kolekcji przy określeniu kryteriów. Nie trzeba uczyć się jak formułować adresy by otrzymać konkretny efekt. Bardzo ułatwia to rozbudowę.

---

61 <https://docs.spring.io/spring-hateoas/docs/1.1.0.BUILD-SNAPSHOT/reference/html/#migrate-to-1.0.changes>

(odwiedzono dn. 28.02.2020) W nowej wersji, zostało zmienione nazewnictwo dla representation models. Łatwo o pomyłkę, gdyż stara wersja bibliotek HATEOAS dla Spring 'a nadal funkcjonuje. Porównaj: ResourceSupport oraz RepresentationModel, Resource oraz EntityModel; Resources oraz CollectionModel, PagedResources oraz PagedModel

62 <https://docs.spring.io/spring-hateoas/docs/1.0.0.BUILD-SNAPSHOT/api/org/springframework/hateoas/server/mvc/WebMvcLinkBuilder.html> (odwiedzono dn. 28.02.2020)

```

http://localhost:8080/api/v3/payments?kind=medicine
{
  "_embedded": {
    "paymentsList": [
      {
        "id": 1,
        "kind": "medicine",
      },
      {
        "id": 3,
        "kind": "medicine",
      },
      {
        "id": 11,
        "ownerId": 5,
        "kind": "medicine",
        "_links": {
          "self": {
            "href": "http://localhost:8080/api/v3/payments/11"
          }
        }
      }
    ]
  },
  "_links": {
    "allPayments": {
      "href": "http://localhost:8080/api/v3/payments"
    },
    "self": {
      "href": "http://localhost:8080/api/v3/payments?kind=medicine"
    }
  }
}

http://localhost:8080/api/v3/payments
{
  "_embedded": {
    "paymentsList": [
      {
        "id": 1,
        "ownerId": 1,
        "kind": "medicine",
        "amount": 101.61,
        "date": "2020-02-27T17:45:10.030+0000",
        "_links": {
          "self": {
            "href": "http://localhost:8080/api/v3/payments/1"
          }
        }
      },
      {
        "id": 3,
        "ownerId": 1,
        "kind": "medicine",
        "amount": 101.61,
        "date": "2020-02-27T17:45:10.030+0000",
        "_links": {
          "self": {
            "href": "http://localhost:8080/api/v3/payments/3"
          }
        }
      },
      {
        "id": 11,
        "ownerId": 5,
        "kind": "medicine",
        "amount": 101.61,
        "date": "2020-02-27T17:45:10.030+0000",
        "_links": {
          "self": {
            "href": "http://localhost:8080/api/v3/payments/11"
          }
        }
      }
    ]
  },
  "_links": {
    "self": {
      "href": "http://localhost:8080/api/v3/payments"
    },
    "PaymentById": {
      "href": "http://localhost:8080/api/v3/payments/{id}",
      "templated": true
    },
    "PaymentByKind": {
      "href": "http://localhost:8080/api/v3/payments?kind={kind}",
      "templated": true
    },
    "PaymentsByOwnerIdAndKind": {
      "href": "http://localhost:8080/api/v3/payments/owner_id={owner_id}&kind={kind}",
      "templated": true
    },
    "PaymentsByOwnerIdAndAmountRange": {
      "href": "http://localhost:8080/api/v3/payments/owner_id={owner_id}&min={min}&max={max}",
      "templated": true
    }
  }
}

```

**Ilustracja 16.** Uprozczone wyniki przykładowych zapytań HTTP dla API w wersji 3

Wszystkie informacje potrzebne trafić na konkretne elementy kolekcji zwracane są w odpowiedzi. Dzięki linkom typu templated, programista klienta dowie się, w jaki sposób API pozwala na filtrowanie płatności. Nie będzie próbował pisać podobnej funkcjonalności, skorzysta z gotowego rozwiązania zaoferowanego po stronie serwera.

Przedstawiłem ewolucję API, które przeszło wszystkie poziomy dojrzałości według modelu Richardson 'a. Zrobiłem przekrój przez mechanizmy dostarczane przez Spring' a. Udało mi zaprezentować wiele przykładów i omówić szereg technik, które służą do ulepszania kontrolera REST. W przykładach celowo pominąłem wykorzystanie trwałego źródła do zapisu informacji, gdyż wykorzystanie baz danych stwarza konieczność zmiany podejścia do aplikacji. W kolejnym rozdziale umówię jak spiąć relacyjną bazę MySQL z REST-owym API wykorzystując w tym celu mapery obiektowo-relacyjne JPA.

## Rozdział II – Zapis informacji w bazie danych

### Założenia projektu a metoda utrwalania danych.

#### Magazynowanie danych

Głównym celem internetowych aplikacji wykorzystujących REST-owe API jest zarządzanie zagregowanymi danymi. W pełni rozwinięty CMS<sup>63</sup> posiada możliwość zapisu danych zebranych przez klientów. Użytkownicy korzystający z przeglądarek internetowych, wykorzystując komunikacyjny protokół HTTP, wymieniają bezustannie porcje danych. API w tym sensie pozostaje pośrednikiem, warstwą aplikacji konsumowaną przez klienta, której dane w postaci graficzno-tekstowej trafiają do użytkowników. Mamy trzy rodzaje interfejsów: interfejs graficzny dla osób korzystających z aplikacji webowych, interfejs programistyczny API za pomocą, którego można pisać funkcjonalności dla klientów oraz interfejs dla źródła utrwalania informacji, podzbiór instrukcji danego języka, umożliwiający programowanie modelu danych dla kontrolera aplikacji.

Dysk twardy serwera, może przechowywać różne dane: grafikę, audio, filmy itp. Dla systemu ważna jest tylko reprezentacja zasobu w postaci nazwy, wskaźnika, referencji. Mając dużą ilość informacji należy skupić na organizacji: łącząc pojedyncze jednostki w zbiory. Zbiory w dużych systemach będą wchodziły w relacje, rzutując na kształt CMS-a. REST-owe API powinno realizować założenia modelu danych dla konkretnej aplikacji.

Mechanizm, narzędzie za pomocą, którego serwer organizuje, przechowuje dane może być dowolny. Decydują o tym zwykle względy ekonomiczne. W przypadku komercyjnych rozwiązań coraz częściej spotykane są rozwiązania typu NoSQL<sup>64</sup>. Można, więc dostarczyć programiście klienta, REST-owe API, pod którym znajduje się dowolne źródło utrwalania: relacyjna baza danych, ale też baza danych w postaci XML plików czy DSMS<sup>65</sup>. Sposób zapisu, agregacji danych po stronie serwera ma wpływ na podejście twórcy API. Musi on w kodzie aplikacji odwzorować model danych, zbudować czytelne abstrakcje na poziomie warstwy kontrolera.

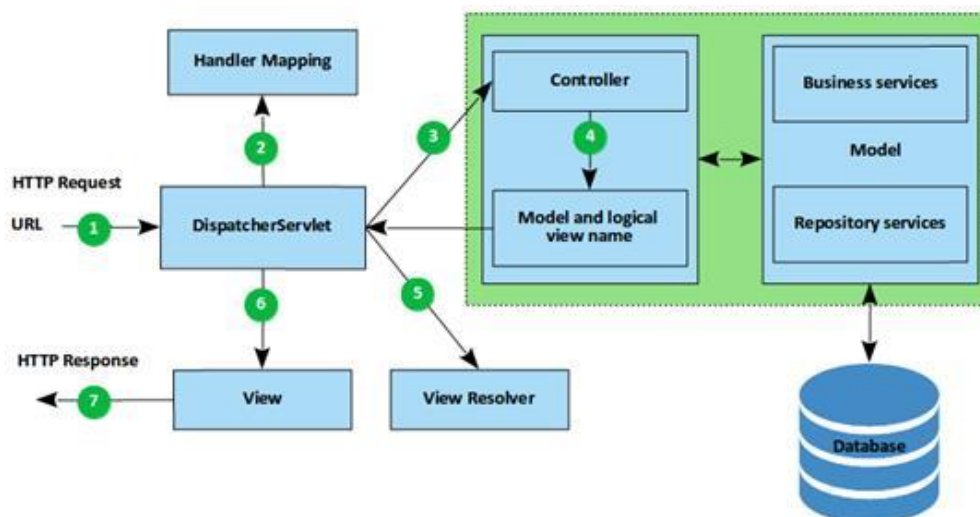
---

63 Lista CMS-ów: [https://pl.wikipedia.org/wiki/Por%C3%B3wnanie\\_system%C3%B3w\\_CMS](https://pl.wikipedia.org/wiki/Por%C3%B3wnanie_system%C3%B3w_CMS) (odwiedzono dn. 06.03.2020)  
[https://pl.wikipedia.org/wiki/System\\_zarz%C4%85dzania\\_tre%C5%9Bci%C4%85](https://pl.wikipedia.org/wiki/System_zarz%C4%85dzania_tre%C5%9Bci%C4%85) (odwiedzono dn. 06.03.2020)

Content Management System, CMS - oprogramowanie pozwalające na łatwe utworzenie i prowadzenie serwisu WWW, a także jego późniejszą aktualizację i rozbudowę, również przez redakcyjny personel nietechniczny.

64 <https://www.widzialni.pl/blog/nosql-alternatywa-dla-relacyjnych-baz-danych/> (odwiedzono dn. 06.03.2020) Bartłomiej Słomiński - „NoSQL – alternatywa dla relacyjnych baz danych” (aktualizacja: 14.05.2018)

65 <https://www.unity.pl/blog/dlaczego-twojej-firmie-potrzebne-jest-strumieniowe-przetwarzanie-danych/> (odwiedzono dn. 06.03.2020) - Unity Group, „Dlaczego Twojej firmie potrzebne jest strumieniowe przetwarzanie danych?”



**Ilustracja 17.** Opis działania REST-API napisanego Spring Framework<sup>66</sup>

### Relacyjna baza danych

Decydując się na wykorzystanie w projekcie relacyjnych baz danych uwzględniłem, szereg ograniczeń z tym związanych. Wykorzystane przeze mnie rozwiązanie stanowi jeden z przykładów gdzie REST-owe API przydaje się praktyce. Trudno wyobrazić sobie program do zarządzania budżetem domowym bez możliwości odwołania się do wcześniej zapisanych płatności. W tym celu wykorzystuje relacyjną bazę danych, nie stanowi to jednak jedynego rozwiązania. Co więcej, w przypadku małych zbiorów informacji, wykorzystywanie DBMS<sup>67</sup> jest niepotrzebne ze względów ekonomicznych, wiąże się ze zbyt dużym narzutem pracy. Wszystkie informacje dotyczące rachunków można zmieścić w prostym pliku tekstowym.

W szerszym kontekście praca z bazami danych stanowi temat na osobne zagadnienie, stąd podjąłem się realizacji przykładu z wykorzystaniem najbardziej kanonicznego sposobu na agregację informacji, jakim jest użycie relacyjnej bazy danych. Nie da się stworzyć dobrego API bez wiedzy o sposobie przechowywania dostarczanych przez klienta, a wyprowadzanych przez program na serwerze, danych. W dalszych rozdziałach pracy rozwijam wątpliwości związane z dodaniem do aplikacji źródła utrwalania danych, opisuje potencjalne problemy i sposoby ich rozwiązania na poziomie frameworka Javy, jakim jest Spring.

<sup>66</sup> <https://www.developer.com/java/data/exploring-rest-apis-with-spring-mvc.html> (odwiedzono dn. 06.03.2020) „Exploring REST APIs with Spring MVC” (aktualizacja: 26.02.2018)

<sup>67</sup> <https://www.techopedia.com/definition/24361/database-management-systems-dbms> (odwiedzono dn. 06.03.2020) „Database Management System (DBMS)”, Techopedia - The IT Education Site, (aktualizacja: 30.08.2019)

## Model danych oparty o ORM

Java pozwala na napisanie aplikacji łączącej się bezpośrednio z bazą danych. Język posiada odpowiedni interfejs programistyczny JDBC<sup>68</sup> w postaci bibliotek pozwalających na wykonywanie poleceń SQL z poziomu klas. W zależności od wyboru silnika potrzebny jest dodatkowy sterownik (provider), umożliwiający użycie konkretnego serwera bazodanowego. Pakiet **java.sql.\*** pozwala na niskopoziomowe, w kontekście aplikacji webowych, programowanie komponentów aplikacji, które wchodzi w interakcje z bazą danych. Java umożliwia tworzenie funkcjonalnych programów, które wykorzystują stałe źródło utrwalania informacji, jednak dopiero wykorzystanie możliwości frameworków pozwala w pełni zautomatyzować i uprościć proces integracji warstw oprogramowania. Zrezygnować z wielu rutynowych czynności stojących po stronie dewelopera.

W projekcie wykorzystuje mapowanie obiektowo-relacyjne, realizuje podejście, którego założeniem jest odwzorowanie w architekturze aplikacji modelu bazodanowego. Mapery pozwalają na rezygnację z konieczności wstępnego zapisu schematu tabel w bazie. W tym podejściu model bazy odpowiada modelowi klas języka. Pola zawierające specyficzne adnotacje, odpowiadają kolumnom w tabeli. Teoretycznie programista nie musi znać języka SQL, aby wygenerować za pomocą odpowiednio zastosowanych adnotacji i ich właściwości, konkretnych tabel, które aplikacja będzie wykorzystywała do zapisu informacji. W praktyce często jednak oznacza to konieczność stworzenia osobnych plików SQL, w celu uniknięcia niejednoznaczności. Przetłumaczenie jednego modelu danych na inny wymaga dobrej znajomości tego, co dzieje pod spodem. Automatyzacja pracy z wykorzystaniem ORM bez wiedzy na temat mechanizmów związanych z bazami danych, będzie powodować straty czasowe. Gotowe rozwiązania mają zawsze ograniczenia.

Java Persistence API oraz Spring Data<sup>69</sup> dostarczają biblioteki, które zwiększają możliwości programisty optymalizując wykonywanie rutynowych zadań, jakimi są m.in. tworzenie pośrednich warstw aplikacji potrzebnych do stworzenia połączenia z bazą danych czy zapisywanie standardowych zapytań SQL celem wyszukania odpowiednich rekordów. Spring Data rozszerza możliwości JPA umożliwiając tworzenie specjalnych klas repozytoriów. Pozwalają one na tworzenie funkcji do wyszukiwania obiektów, które programuje się przy użyciu odpowiednich nazw:

---

68 <https://javastart.pl/baza-wiedzy/java-ee/jdbc-podstawy-pracy-z-baza-danych> (odwiedzono 07.03.2020) „JDBC - Podstawy pracy z bazą danych”, Java DataBase Connectivity - jest to interfejs pozwalający na ustanowienie połączenia do bazy danych z poziomu aplikacji napisanej w Javie. Biblioteki, których potrzebujemy, zawierają pakiety `java.sql.*` i są w taki sam sposób dostępne jak kolekcje w pakietach `java.util.*`.

69 <https://techrocking.com/what-is-jpa-and-spring-data-jpa/> (odwiedzono 07.03.2020) „What is JPA and Spring Data JPA”, 29.12.2019

[...] Spring Data definiuje pewnego rodzaju miniatury język DSL (domain-specific language), w którym szczegóły utrwalania wyrażone są w sygnaturze metody [...] W repozytorium składają się one z czasownika, opcjonalnego tematu, słowa By oraz predykatu [...] umożliwia użycie w nazwie metody czterech czasowników: *get*, *read*, *find* oraz *count*.<sup>70</sup>

Repozytoria bardzo upraszczają pośredniczenie pomiędzy warstwą utrwalania danych a warstwą kontroli. Dają swobodę przy implementacji gotowych, przemyślanych rozwiązań. W przypadku bardziej złożonych kryteriów, rozbudowanych zapytań wymagają od programisty posłużenia się dodatkowymi obiektami i adnotacjami wchodzącymi w skład Spring 'a. Zaawansowane operacje wymagają znajomości specyfikacji frameworka oraz języka SQL. W tym kontekście warto zastanowić się czy użycie dodatkowych warstw aplikacji jest zawsze niezbędne.

Dla API w pakiecie **pl.cecherz.calcbill.controller.db** wykorzystuje dwie klasy rozszerzające interfejs *CrudRepository*<sup>71</sup>. Po jednej dla każdej klasy w modelu. Klasę *OwnerRepository* przeznaczyłem dla metod wyszukujących dotyczących obiektów typu *Owner*, klasa *PaymentsRepository* działa analogicznie dla bean 'a *Payments*.

**Tabela 4.** Metody wyszukujące w klasach rozszerzających *CrudRepository*

| Metoda                                                         | Klasa                     | Odpowiednik SQL                              |
|----------------------------------------------------------------|---------------------------|----------------------------------------------|
| <i>findOwnerById(Integer id)</i>                               | <i>OwnerRepository</i>    | SELECT * FROM Owner WHERE id=?               |
| <i>findPaymentById(Integer id)</i>                             | <i>PaymentsRepository</i> | SELECT * FROM Payments WHERE id=?            |
| <i>findPaymentsByKind</i><br>(String kindOfPayments)           | <i>PaymentsRepository</i> | SELECT * FROM Payments WHERE kind=?          |
| <i>findPaymentsByAmountBetween</i><br>(Double min, Double max) | <i>PaymentsRepository</i> | SELECT * FROM Payments WHERE min>? AND max<? |

W klasie *OwnerRepository* zawarta jest tylko jedna metoda, na tej podstawie nie należy jednak wnioskować funkcjonalności API. Kontroler wykorzystujący tę klasę wyszukuje danych w oparciu o relacje na poziomie bazy danych. Rozwiązanie to opisze w kolejnym podrozdziale.

70 Publikacja Craig'a Walls, „Spring w Akcji”, tłum. Mirosław Gołda, Piotr Rajca, wydanie IV, wyd. Helion 2015, Gliwice, rdz. 11 „Zapisywanie danych z użyciem mechanizmów ORM”, s.349

71 <https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html> (odwiedzono 07.03.2020) „Working with Spring Data Repositories”.

Repozytoria zawierają również zestaw gotowych metod służących do operacji bazodanowych dostępnych z poziomu kodu aplikacji, bez pisania dodatkowych linii kodu<sup>72</sup> (fragment dokumentacji poniżej):

- **count()** Returns the number of entities available.
- **delete(T entity)** Deletes a given entity.
- **deleteAll()** Deletes all entities managed by the repository.
- **deleteAll(Iterable<? extends T> entities)** Deletes the given entities.
- **deleteById(ID id)** Deletes the entity with the given id.
- **existsById(ID id)** Returns whether an entity with the given id exists.
- **findAll()** Returns all instances of the type.
- **findAllById(Iterable<ID> ids)** Returns all instances of the type T with the given IDs.
- **findById(ID id)** Retrieves an entity by its id.
- **save(S entity)** Saves a given entity.
- **saveAll(Iterable<S> entities)** Saves all given entities.

Część wymienionych metod wykorzystuje przy projektowaniu kontrolera API. Mając dostęp do tak dobrze zdefiniowanego szablonu bardzo łatwo jest wdrożyć i udostępnić klientowi operacje CRUD.

## Reprezentacja relacji w REST API

API, które opisywałem w rozdziale I nie wykorzystuje adnotacji charakterystycznych dla klas encyjnych. Nie potrzebuje ich gdyż nie korzysta z warstwy utrwalania danych (dane zapisuje w pamięci tymczasowej, jako obiekty kolekcji). Dodanie do aplikacji bazy danych zmusza do przebudowy modelu. Zwykle czyste klasy Javy w postaci beanów, nie dają możliwości integracji ze strukturą tabel. ORM pozwala opisać klasy adnotacjami, które zostaną zinterpretowane przez kompilator generując odpowiedni schemat MySQL<sup>73</sup>.

Należy założyć, że operujemy na bazie danych tylko z poziomu IDE odpowiednio w tym celu dostosowując konfigurację. Framework pozwala na wysokopoziomowe podejście ograniczając

---

72

<https://docs.spring.io/springdata/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html> (odwiedzono 07.03.2020)

73 W projekcie wykorzystuje bazy danych MySQL jednak wybór silnika, leży po stronie programisty w przypadku relacyjnej bazy danych pozostaje to bez wpływu na funkcjonalność frameworka.

wstępne ustawienia do absolutnego minimum. W podstawowych zastosowaniach można ograniczyć się do pliku *application.properties*, pozwala on określić m.in. sposób połączenia z bazą (hasło, host, użytkownik), strategię inicjalizacji tabel<sup>74</sup>. Ustawienie *spring.jpa.hibernate.ddl-auto* zmienia sposób, w jaki tworzone są struktury w bazie:

- **create** - stworzy nowe tabele, gdy nie istnieją
- **create-drop** - stworzy nowe tabele, gdy schemat nie odpowiada temu, co jest w klasach encyjnych nadpisze dane
- **update** - aktualizuje dane wewnątrz tabel, nie wpływa na zmianę właściwości kolumn.
- **validate** - sprawdza schemat

Brak znajomości działania powyższych ustawień, spowoduje problemy programiście, który chce przestawić się z poziomu czystego SQL-a na to, co oferuje framework. Schemat musi być obsługiwany z jednego miejsca inaczej dochodzi do konfliktu, mapper skasuje zmiany wprowadzone boczną furtką z poziomu programu zewnętrznego typu MySQL Workbench<sup>75</sup>. W przypadku innego scenariusza, jeśli pomyłone zostaną ustawienia, nie weźmie pod uwagę modelu zbudowanego z wykorzystaniem frameworka. Baza ma tworzyć się w sposób automatyczny, odpowiada za to kod programu Javy. Trudność polega tu na odszukaniu dla każdej operacji SQL DDL odpowiedniej alternatywy w postaci adnotacji, właściwości. Użycie ORM nie zwalnia z dobrej znajomości języka SQL. Łatwiej jest stworzyć schemat tabeli przy użyciu kreatora typu phpMyAdmin<sup>76</sup> niż odszukać zestaw adnotacji dla konkretnych funkcji, odpowiedzialnych np. za tworzenie relacji.

Z pomocą Spring Data udało mi się odwzorować schemat bazy, który przedstawiłem na początku niniejszej pracy (ilustracja nr. 3 oraz nr. 4) Model uwzględnia istnienie relacji jeden-do-wielu i przekłada to na zawartość danych zwracanych przez API. Przebudowałem kontroler przestawiony w poprzednim rozdziale inaczej realizując odpowiedzialność klas. Podejście, które zastosowałem jest odpowiedzią na problem zagnieżdżonych danych, a co za tym idzie ciągu nieskończonych rekurencyjnych wywołań, które trafiają do klienta w przypadku błędnej realizacji modelu.

---

74 <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html> (odwiedzono dn.10.03.2020), „Common Application properties”, aktualizacja: 02.27.2020

75 <https://www.mysql.com/products/workbench/> (odwiedzono dn.10.03.2020) Popularny menadżer graficzny do baz danych MySQL.

76 <https://www.phpmyadmin.net/> (odwiedzono dn.10.03.2020) Popularny webowy administrator baz danych typu MySQL. Program został napisany w językach skryptowym PHP oraz JavaScript, co sprawia, iż jest dostępny w większości usług hostingowych.

Kontroler wykorzystujący klasę *Payments*, do której dodana została funkcjonalność związana z ORM, pakiet **pl.cecherz.calcbill.model.db**, zawiera metody odpowiedzialne za definiowanie kryteriów wyszukiwania obiektów określonego typu, uwzględniając ich cechy tylko w obrębie tabeli *payments*. Dodałem możliwości wyszukania, grupowania płatności ze względu na ich rodzaj. Także możliwość ich filtrowania ze względu na wysokość z określeniem wartości granicznych. API zwraca obiekty, które nie wiedzą nic o posiadaczu. Kolumna klucza obcego nie jest widoczna dla klienta, choć dane te są przechowywane na poziomie bazy. Można powiedzieć, że klasa *PaymentsController* udostępnia klientowi pod-API, używając go nie można się dowiedzieć, do kogo należy konkretna płatność. Tylko metoda z adnotacją *@PostMapping* operuje na danych jednocześnie w dwóch tabelach. Funkcja *addPayment* dodaje płatności oczekując od klienta w adresie URI przekazania numeru id posiadacza. W sposób niejawny dla API wiąże dane w bazie, rozwiązując problem rekurencyjnych wywołań.

API w klasie *OwnerController* wykorzystuje model *Owner*. Posiada on pole, zdefiniowane jako listę zawierającą zbiór obiektów typu *Payment*. Posiadacz płatności (*Owner*) ma dostęp do zagnieżdżonej w JSON-ie struktury, która zawiera wiele obiektów poszczególnych płatności. API niczego nie zgaduje, wystarczy, że zna klucze obce w tabeli *payments*. Z poziomu *OwnerController* możliwe jest filtrowanie danych płatności, z uwzględnieniem ich właściciela. Kontroler zwraca dane posiadacza (id, imię, nazwisko) jak i dane płatności, jako obiekty zagnieżdżone.

Powyżej opisane podejście może wydawać się skomplikowane, być może wszystko udałoby się zastąpić jednym kontrolerem, skoro API w klasie *OwnerController* zwraca wszystko? Można zrezygnować z API do obsługi płatności, dane bez wskazania posiadacza nadają się tylko do ogólnych zestawień. Nie można jednak wyrzucić klasy modelu, uczestniczy ona w tworzeniu schematu bazy. Przyjęty przeze mnie podział odpowiedzialności kontrolerów odpowiada na problemy z modelem, skoro *Owner* zawiera *List<Payments>* a *Payments* obiekt *owner\_id* typu *Owner* program będzie spodziewał się w referencji dostępu do listy, a w liście odwołania do obiektu, który tę listę posiada. Koło się zamyka. Kłopoty rozwiązują dodatkowe adnotacje: *@JsonManagedReference*, *@JsonBackReference*<sup>77</sup>, które pozwalają na uniknięcie konfliktów związanych z biblioteką Jackson<sup>78</sup>.

Nie będę zajmował się szerzej opisem całego zestawu adnotacji dla JPA, więcej na ten temat można dowiedzieć się z dokumentacji.<sup>79</sup> Wykonałem aplikację, która zawiera dużo zależności, korzysta z zewnętrznych bibliotek, używa ORM, a wszystko po to wykonać zapis dwóch prostych tabelach. Nie

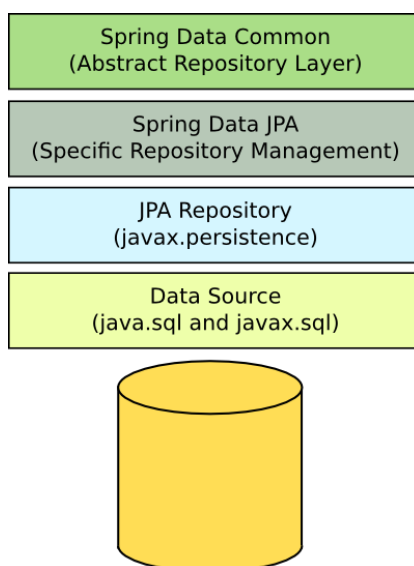
---

77 <https://www.baeldung.com/jackson-bidirectional-relationships-and-infinite-recursion> (odwiedzono dn.10.03.2020), „Jackson – Bidirectional Relationships”, aktualizacja: 12.02.2020

78 <https://github.com/FasterXML/jackson> (odwiedzono dn.10.03.2020) Jackson na portalu github.com

79 <https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html> (odwiedzono dn.10.03.2020) Package javax.persistence

wpływa to na wydajność ani nie przyspieszyło pracy. Program jest zbyt prosty by móc zobaczyć zalety wykorzystania poszczególnych narzędzi. Zyskał na pewno większą elastyczność, obejmujemy aspekt tabel i klas jednocześnie; wszystkim zarządza kod programu. To pozwala trudności uniknąć związanych z integracją, gdy warstwę utrwalania i kontroler trzeba programować osobno. Gdy wyobrazimy sobie bazę złożoną z setki tabel, zdamy sobie sprawę z ułatwienia, jakie dają mapery.



*Ilustracja 18. Architektura Spring Data, podział warstw<sup>80</sup>*

## Obsługa wyjątków

W rozdziale I umówiłem webowe API, które wykorzystuje metadane pojawiające na poziomie komunikacji z wykorzystaniem protokołu HTTP. Stworzyłem kontroler, który realizuje założenia REST zdefiniowane w modelu Richardson 'a. API w pakiecie **pl.cecherz.calcbill.controller.json**, w wersji 2 oraz 3 korzysta intensywnie z obiektów typu *ResponseEntity* w celu zdefiniowania danych zwracanych w HTTP header: kodu odpowiedzi czy czasu przechowywania danych w cache. Opisuje użycie obiektów *ResponseEntity* głównie do obsługi błędów. W rzeczywistości korzystanie z tej klasy opakowującej zwracane kolekcje nie jest wymagane, aby napisać obsługę wyjątków dla aplikacji.

W rozdziale II pokazuje zmianę podejścia. Rezygnuje z obsługi HATEOAS oraz z przekazywania w parametrach nagłówek dodatkowych właściwości, takich jak Cache-Control. Pozwala to na pozbycie się klas narzędziowych z kontrolera i daje możliwość łatwego odseparowania części aplikacji odpowiedzialnej za obsługę wyjątków. Narzędzia takie jak **org.springframework.hateoas** czy

---

<sup>80</sup> <https://www.tatvasoft.com.au/blog/spring-data-jpa-for-abstraction-of-queries/> (odwiedzono 10.03.2020) „Spring Data JPA for Abstraction of Queries”, 02.2018

**org.springframework.http.ResponseEntity** nie stoją na przeszkodzie tworzenia klarowniejszych klas do obsługi błędów, jednak zależało mi na uproszczeniu kodu aplikacji celem lepszego przedstawienia pewnych mechanizmów.

Klasy w pakiecie **pl.cecherz.calcbill.controller.db** są mniejsze, znacznie czytelniejsze. Bardziej oczywiste dla deweloperów nie znających wszystkich narzędzi, jakie oferuje Spring Framework. Przy zwracanych typach ograniczyłem się do obiektów modelu *Owner* i *Payments* opakowanych w klasy kolekcji Javy. Klasa *OwnerController* przy funkcjach typu GET, zwracających kolekcje, wykorzystuje ścieżki, adnotacja *@PatchMapping*; *PaymentsController* zaś parametry, adnotacja *@RequestParam*. Taki podział w zachowaniu kontrolerów API wydał mi się logiczny, klasa *OwnerController* wykorzystuje, pod-zasoby; zwraca wszystko to, co zawierają dwie tabele w bazie. W przypadku kontrolera do obsługi płatności, używana jest jedna tabela, w celu zwrócenia danych klientowi API.

Kontrolery z zaimplementowaną obsługą błędów rozszerza klasa *RestExceptionHandler* zawiera ona adnotacje charakterystyczne dla frameworka, ułatwiające napisanie klas obsługi wyjątków. Działanie komponentu jest bardzo proste:

1. Adnotacja *@ExceptionHandler* opisująca metodę, wskazuje na klasę stworzoną konkretnie do wybranego wyjątku.
2. Adnotacja *@ResponseStatus* definiuje kod odpowiedzi, jaki zostanie wygenerowany po przechwyceniu wyjątku.
3. Metoda zwraca obiekt typu *HTTPError*, który zawiera informacje o kodzie HTTP oraz informacje dla użytkownika w formie komunikatu.

Napisałem cztery klasy charakterystyczne dla wybranych przeze mnie sytuacji wyjątkowych. Zaproponowane przeze mnie rozwiązanie dotyczy konkretnej aplikacji, niewykluczone, że pewnych sytuacji nie przewidziałem, a niektóre rzeczy da się zrobić lepiej. Zastosowałem jeden z wielu możliwych sposobów na obsługę błędów przy pomocy Spring Framework<sup>81</sup>.

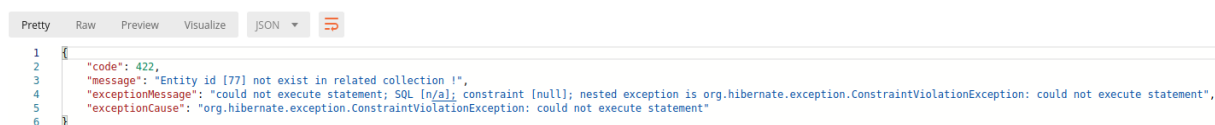
---

81 <https://www.baeldung.com/exception-handling-for-rest-with-spring> (odwiedzono dn. 11.03.2020), „Error Handling for REST with Spring” (aktualizacja 07.03.2020). Artykuł opisuje kilka możliwych rozwiązań, uwzględnia też rozwiązania, gdy aplikacja korzysta z kontroli dostępu.

**Tabela 5.** Typy wyjątków rzucanych przez API pakiet *pl.cecherz.calcbill.controller.db*

| Typ wyjątku                 | Przeznaczenie                                                | Status |
|-----------------------------|--------------------------------------------------------------|--------|
| EntityNotFoundException     | Element o podanych kryteriach nie istnieje w kolekcji.       | 404    |
| EntityEmptyContentException | Element o podanych kryteriach jest pusty.                    | 200    |
| EmptyFindResultException    | Żaden element w kolekcji nie spełnia kryterium. Zbiór pusty. | 200    |
| DataIntegrityException      | Nie można stworzyć relacji z elementem, który nie istnieje.  | 422    |

Metody typu POST dla obu kontrolerów, posiadają adnotacje `@ResponseStatus(HttpStatus.CREATED)`, co oznacza, że po utworzeniu zasobu zwracany jest kod odpowiedzi 201. Funkcja `addPayment` w klasie `PaymentController` ze względu na nietypowe działanie, zwraca wyjątek `DataIntegrityException`, który tak naprawdę jest opakowanym wyjątkiem zwracanych przez Spring 'a ze względu na pojawiającą się niespójność w bazie danych<sup>82</sup>. Nie da się stworzyć relacji w przypadku, gdy element, do którego chcemy odwołać nie istnieje. Przechwytywanie wyjątku ogólnego i rzucanie na tej podstawie własnego nie jest być może dobrą praktyką, nie udało mi wyszukać jednak lepszego rozwiązania w zestawieniu z wybraną metodą obsługi błędów.



```

1 {
2   "code": 422,
3   "message": "Entity id [77] not exist in related collection !",
4   "exceptionMessage": "could not execute statement; SQL [n/a]; constraint [null]; nested exception is org.hibernate.exception.ConstraintViolationException: could not execute statement",
5   "exceptionCause": "org.hibernate.exception.ConstraintViolationException: could not execute statement"
6 }

```

**Ilustracja 19.** Informacje, które zwraca API w momencie rzucenia wyjątku `DataIntegrityException`

Kontrolery rzucają wyjątki, które po obsłużeniu zwracają komunikaty w formie JSON-a klientowi. Taki sposób działania webowego REST API jest bardziej pożądany, w wyniku wystąpienia sytuacji nieprzewidzianej będzie ono w stanie samo wyjaśnić, co zaszło nie tak. Wszystko to nie oznacza jednak, że można zrezygnować z walidacji danych na poziomie front-end 'u. Klienci API powinny się starać by dane, które trafiają do kontrolera były pozbawione błędów. Należy uniemożliwić użytkownikom aplikacji wygenerowanie błędu ze strony serwera umiejętnie projektując wygląd strony, zawartość formularzy. Dobrą praktyką jest dodanie w warstwie widoku aplikacji stron błędu związanych, ze zwracanym kodem odpowiedzi.

W rozdziale drugim pokazałem jak połączyć REST API z relacyjną bazą danych. Wyjaśniłem krótko architekturę Spring 'a. Przybliżyłem tematykę mapek obiektowo-relacyjnych, które

przyczyniają do łatwiejszej integracji warstw aplikacji. Opisałem rozwiązania, które pozwalają zbudować model w postaci klas Javy, który dzięki zastosowaniu bibliotek narzędziowych Spring 'a, generuje schemat bazy danych. Przedstawiłem przykład przebudowanego webowego REST API z odseparowanym na poziomie kodu aplikacji, bloku do obsługi wyjątków. W rozdziale trzecim zajmę warstwą widoku, przedstawię rozwiązania mające na celu stworzenie strony w postaci strony HTML przyjaznej użytkownikom aplikacji.

## Rozdział III – Przeglądarkowa aplikacja Web, jako warstwa widoku.

### Wybór medium dla aplikacji

Mając zapewnioną obsługę danych z możliwością zapisu w relacyjnej bazie danych, można zastanowić na sposobami wykorzystania RESTful Web API. Interfejs, który dostarcza funkcjonalności w postaci zmapowanych metod dostępnych z poziomu paska adresu przeglądarki WWW, może zostać użyty w dowolny sposób, ze wsparciem technologii, która ma zastosowanie w budowaniu aplikacji webowych. Protokół HTTP stanowi uniwersalny standard komunikacyjny dla całej Sieci, której budowa opiera się o model TCP/IP<sup>83</sup>. Stąd powstaje sposobność na połączenie funkcjonalności dwóch różnych technologii, odmiennych światów programowania w jednym projekcie. Wspólny pozostaje format wymiany danych, ja wykorzystuje JSON, ale tak naprawdę klient i API mogą komunikować się dowolnie w zależności od wybranych rozwiązań dla danego systemu.

W poprzednich rozdziałach pokazałem jak eksplorować API przy pomocy klientów takich jak Mozilla Firefox czy Postman. Wykorzystując Postmana można wykonać wszystkie operacje zdefiniowane w kodzie Javy kontrolerów, co ma wpływ na zmianę stanu bazy danych. Nic nie stoi nie przeszkodzie by aplikację w Springu, umieścić na chmurze obliczeniowej Heroku<sup>84</sup>, bazę danych zaś w innym miejscu takim jak np. hosting home.pl. Aplikacja będzie działać w rozproszeniu, bez znaczenia jest architektura sprzętowa. Oczywiście sens podziału warstw oprogramowania nabiera sensu w przypadku dużych systemów a rozproszenie w momencie, gdy wdrożona zostanie w aplikacji kontrola dostępu oraz obsługa transakcyjności (w pracy nie podejmuje problemów związanych z wielodostępnością, synchronizacją i wątkami). Zadaniem aplikacji webowej jest trafienie do wielu użytkowników, model REST i zastosowanie wzorca MVC bardzo temu sprzyjają.

Postman jest klientem, który pozwala na pełne wykorzystanie możliwości stworzonej przeze mnie aplikacji. Nie nadaje się on jednak dla przeciętnego użytkownika, który oczekiwał będzie standardowego GUI<sup>85</sup>. Strony HTML-a: formularzy, linków i kontrolek w postaci przycisków, które

---

83 WWW (World Wide Web) jest usługą internetową, jednak ze względu na popularność bywa utożsamiana z całym Internetem. Model TCP/IP opisuje podział na cztery warstwy: aplikacji, transportowa, Internetu, dostępu do sieci. Warstwa aplikacji obejmuje użyteczne dla użytkowników programy takie jak serwer WWW czy przeglądarka oraz protokoły m.in. HTTP, FTP.

84 <https://www.heroku.com/> (odwiedzono dn. 17.03.2020) Popularna platforma chmurowa w modelu PaaS.

85 Rozwinięcie skrótu z tłum. graficzny interfejs użytkownika.

pozwolą mu zarządzać danymi. Istnieją trzy drogi by rozwiązać ten problem i dostarczyć przeglądarce widok w postaci znaczników:

- Budowa kolejnego kontrolera HTTP, który stanie się rodzajem fasady dla klas *OwnerController* oraz *PaymentsController*. Celem opakowania kolekcji zwracanych danych w obiekty współpracujące z biblioteką Thymeleaf<sup>86</sup>.
- Stworzenie dodatkowej warstwy aplikacji w Springu, komponentu, który zajmie się konsumowaniem API celem dostarczenia go we właściwej formie widoku.
- Napisanie obsługi danych dla widoku w języku innym niż Java, wspierającym dynamiczne generowanie i manipulację znacznikami dokumentów HTML-a. Należą do nich JavaScript i PHP.

Pierwsze, najprostsze rozwiązanie, chociaż nie wprowadza dodatkowych logicznych warstw pośrednich pomiędzy kontrolerem a widokiem, burzy klarowność aplikacji. Pomysł wstrzyknięcia komponentu w komponent nie należy do najbardziej eleganckich. Mimo pewnej użyteczności jest kontrowersyjny. Korzyścią opisanej strategii jest zachowanie klas, które zwracają czyste dane. Wykorzystanie ich w momencie, gdy wzrosną wymagania odnośnie projektu. Klienci mobilni będą oczekiwali łatwych do parsowania danych w postaci JSON-a.

Rozwiązanie drugie jest przydatne w momencie komunikacji dwóch aplikacji, w modelu dostawca-odbiorcą. Aplikacja napisana w Springu może korzystać z logiki biznesowej programu, serwisu znajdującego się w innym miejscu w Internecie. Przykładowo, aby wykorzystać API Wordpress 'a w Java 'owej aplikacji w Springu korzystającej z Thymeleaf należy pierw udostępnione przez endpointy webowe dane odpowiednio przetworzyć. Chociaż framework dostarcza abstrakcji do konsumowania API nie ma możliwości bezpośredniego generowania widoku<sup>87</sup>.

Trzecia droga. Klasy do tłumaczenia danych oraz przekazywania obiektów do szablonów widoków nie są potrzebne w sytuacji, gdy zdecydujemy się wykorzystać języki skryptowe. Biblioteki zbudowane na bazie JavaScript: takie jak jQuery<sup>88</sup>, AngularJS<sup>89</sup> czy rest.js<sup>90</sup> pozwalają na dynamicznie

---

86 <https://www.thymeleaf.org/> (odwiedzono dn. 21.03.2020) Dokumentacja fragm.

Thymeleaf is a modern server-side Java template engine for both web and standalone environments.

87 <https://spring.io/guides/gs/consuming-rest/> (odwiedzono dn. 21.03.2020) Consuming a RESTful Web Service - opis wykorzystanie abstrakcji Spring RestTemplate

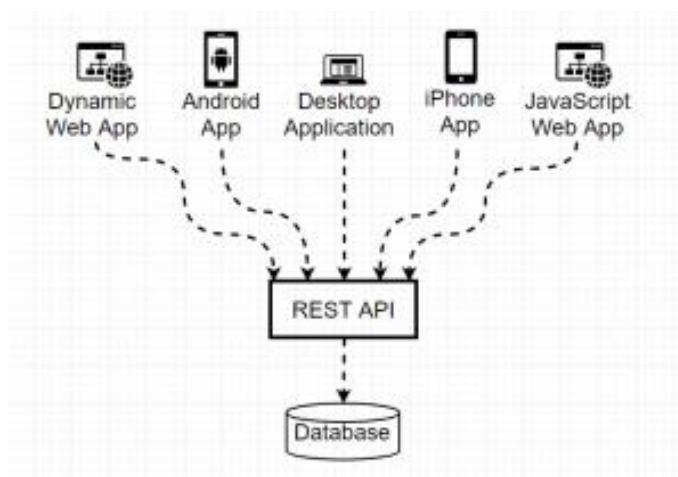
88 <https://spring.io/guides/gs/consuming-rest-jquery/> (odwiedzono dn. 21.03.2020) Consuming a RESTful Web Service with jQuery

89 <https://spring.io/guides/gs/consuming-rest-angularjs/> (odwiedzono dn. 21.03.2020) Consuming a RESTful Web Service with AngularJS

generowanie drzewa dokumentu DOM dla HTML-a w oparciu o dane w postaci JSON-a. Java jest technologią, która zapewnia większą stabilność, skalowalność i bezpieczeństwo aplikacji po stronie backend 'u. Do tworzenia efektów wizualnych lepiej wykorzystać języki umożliwiające pełną interakcję z przeglądarkowym widokiem.

Zdecydowałem, że klientem mojej aplikacji będzie przeglądarka WWW. Stąd skupienie na bibliotece do tworzenia szablonów. Warto jednak mieć świadomość szerokiego spektrum innych urządzeń, które mimo tego, że wykorzystują Internet nie korzystają z widoków w postaci dokumentów HTML. Architektura REST uwalnia od konieczności koncentrowania się na formacie danych. Wcześniej (Rozdział I) pokazałem przykład kontrolera, który zwraca bezpośrednio całą stronę internetową. Nie dowodzi to jednak, że na tym można poprzestać. Pisząc aplikację warto myśleć o docelowych użytkownikach, jak i o osobach rozwijających oprogramowanie. Kontroler API, który zwraca śmieciowe dane: HTML, style CSS nie pozwala na rozwój aplikacji klienckich, to front 'endowy programista decyduje o sposobie przedstawienia odebranych danych.

W ostatnim rozdziale mojej pracy przedstawiam sposoby integracji różnych warstw reprezentacji danych. Jednak nawet w przypadku aplikacji webowych nie ma jednego słusznego rozwiązania; pisanie aplikacji klienckich obarczone jest większymi błędami. Kwestia, co udostępnić klientowi, nie rozstrzyga pytania: jak to pokazać.



**Ilustracja 20.** REST API na różnych urządzeniach<sup>91</sup>

90 <https://spring.io/guides/gs/consuming-rest-restjs/> (odwiedzono dn. 21.03.2020) Consuming a RESTful Web Service with rest.js

91 <https://www.parallels.com/blogs/ras/rest-api/> (odwiedzono dn. 21.03.2020) Vissarion Yfantis - „REST API | What Does It Mean When an Application Has a REST API?” (aktualizacja: 07.06.2019)

## Szablony HTML5 - Spring Thymeleaf

### Kontroler dla widoków

W poprzednim podrozdziale przedstawiałem trzy strategie działania w celu wygenerowania widoków, gdy program posiada własną logikę biznesową lub korzysta z API innego projektu. W projekcie aplikacji do zarządzania rachunkami skupiam się na rozwiązaniu polegającym na rozbudowie warstwy kontrolerów. Stworzenie drugiego kontrolera HTTP uzasadnia wykorzystanie biblioteki Thymeleaf:

*[...] silnik szablonów, [...] przypomina JSP+JSTL+EL, [...] szablony tworzone są w HTML5 z dodatkowymi atrybutami, a nie poprzez niestandardowe znaczniki, [...] szablony nie są tłumaczone do postaci serwetów, a zamiast tego silnik Thymeleaf przetwarza szablon i podstawia do niego odpowiednie atrybuty z modelu.<sup>92</sup>*

Szablony Spring 'a są kompilowane po stronie serwera, generując kod HTML. Odpowiedzi API, dane JSON są generowane po stronie klienta. Nie mam możliwości bezpośredniego przekazania tych informacji pomiędzy widokami z wykorzystaniem znaczników a kontrolerem aplikacji. Należy wykorzystać klasy narzędziowe, które umożliwiają określenie dla odpowiedzi HTTP odpowiednich parametrów.

Nie dokonywałem edycji klas kontrolerów z pakietu **pl.cecherz.calcbill.controller.db**, gdyż to zniszczyłoby założenia koncepcji modelu REST. Kontroler zwracający treść danych, a także informacje o sposobie ich reprezentacji przestaje być użyteczny, jako uniwersalna część aplikacji. Webowe REST API może zwracać tylko to, co jest użyteczne dla komunikacji opartej o protokół HTTP.<sup>93</sup> Rozwiązanie, jakie zastosowałem nieco łamie zasadę odnośnie architektury aplikacji, zmiana klas kontrolerów wpłynie na powiązany z nimi komponent. Nie jest to jednak niezgodne z dobrymi praktykami konserwacji oprogramowania, zastosowanie API dla web 'u nie wyklucza jednoczesnego użycia go w urządzeniach mobilnych.

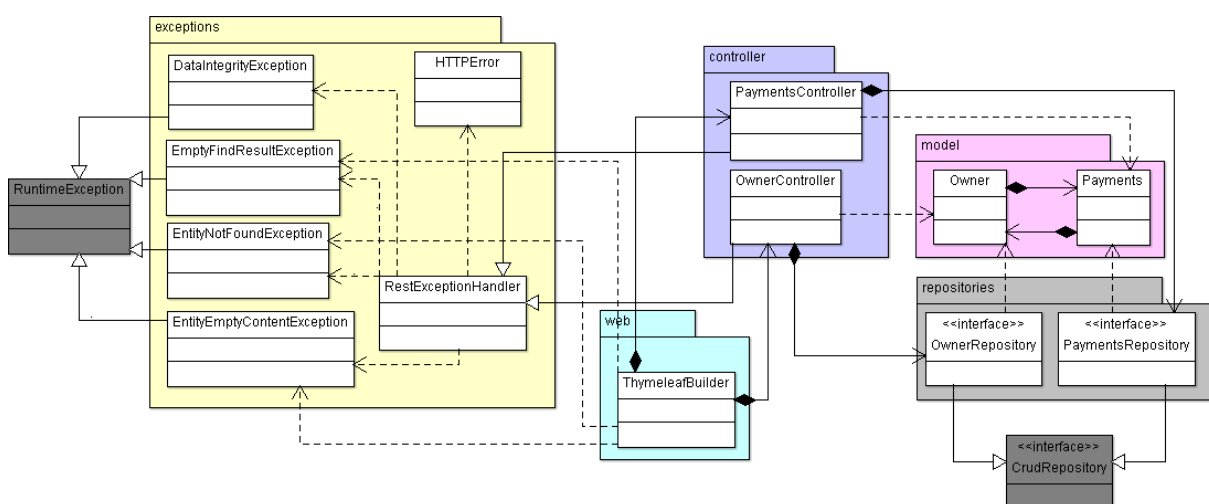
Rozbudowując kod aplikacji, dzieląc go na klasy ze względu na odpowiedzialność doprowadziłem wzrostu skomplikowania struktury zależności. W przypadku programowania z wykorzystaniem paradygmatu obiektowego stopniowy wzrost złożoności jest nie do uniknięcia. Rdzeń

---

92 <https://javastart.pl/kurs/spring/spring-spring-mvc/lekcja/spring-mvc-%2B-thymeleaf> (odwiedzono dn. 22.03.2020) - „Kurs Spring Framework”, własność i autorstwo JavaStart Sławomir Ludwiczak, Marcin Kunert spółka cywilna

93 Wykorzystujemy: request method (Metoda HTTP); header nagłówek; resource path ścieżka dostępu URI; body.

aplikacji klasy z adnotacją `@RestController` nie uległy jednak zmianom, które utrudniłyby ich czytelność. Wykorzystanie strumieni, filtrów oraz szeregu adnotacji bardzo przyspiesza i ułatwia pisanie webowych API w Javie. Dodatkowe komponenty, które stopniowo wprowadziłem do aplikacji nie zaburzyły działania kontrolera. Cały czas opieramy się endpointach, które są obsługiwane przez konkretne metody HTTP.



**Ilustracja 21.** Diagram klas aplikacji w wersji finalnej z widokiem

Klasa *ThymeleafBuilder*, którą wprowadziłem w niezależnym pakiecie stanowi rodzaj mostu między widokiem a kontrolerem API, sama jednak nie wnosi do funkcjonalności programu niczego nowego. Opakowuje wyniki wybranych funkcji klas z pakietu `pl.cecherz.calcbill.controller.db` w abstrakcje, które pozwalają Springowi na obsłużenie widoków.

Aplikacja CALC-BILL<sup>94</sup>, w wersji finalnej udostępnia użytkownikom:

- Możliwość dodawania/edycji użytkowników aplikacji.
- Możliwość dodawania/edycji płatności i przypisywania jej użytkownikom.
- Funkcję usuwania danych.
- Podgląd zsumowanych wartości wszystkich płatności dla konkretnego ich posiadacza.
- Filtrowanie zestawienia płatności wszystkich użytkowników

Nie wykorzystałem całego wcześniej napisanego przeze mnie API. Obsługa widoków dla części endpointów okazała się na tyle złożona, by pozostawić resztę funkcjonalności do rozpatrzenia przy następnych wydaniach aplikacji. Nie ma konieczności obsługiwanego przez GUI całego interfejsu.

94 Nazwa aplikacji do zarządzania rachunkami w wersji 1.0 (22.03.2020), link do repozytorium: <https://github.com/pangeon/calcbill>

Warto dodać, że projektując aplikację, która wykorzystywała będzie dynamiczne szablony HTML i biblioteki takie jak Thymeleaf nie tworzy się REST-owego API dla ogólnych zastosowań. Rozwiązanie, które przedstawiłem w projekcie ma charakter prezentacyjno-edukacyjny.

### Współpraca ze Spring MVC

Istnieją trzy klasy narzędziowe, dzięki którym framework obsługuje widoki: Model, ModelMap oraz ModelAndView<sup>95</sup>. Za każdym razem musimy dostarczyć informacje wymagane przez Spring MVC, czyli: zwracany obiekt lub ich grupa; nazwę atrybutu, nazwę strony oraz lokalizację zasobu dla serwera.

Projektowanie widoku, z użyciem Thymeleaf, sprowadza się do umiejętnego przekazywania danych kolekcji pomiędzy kolejnymi żądaniami. Aby odczytać wartości zmiennych, na jednej stronie muszą one zostać tam przekazane przy uruchomieniu linku w innym widoku, który do niej prowadzi. W moim odczuciu praca polegająca sterowaniu żądaniami jest trudna i żmudna, programista zostaje zmuszony do przeklinania się przez całe webowe GUI by mieć pewno, że całość funkcjonuje poprawnie. Kompilowanie programu za każdym razem, gdy zostanie dokonana jakkolwiek zmiana bardzo wydłuża pracę. Z tej perspektywy warto zwrócić uwagę na alternatywne frameworki dla Javy pojawiające się na rynku, m.in. Quarkus<sup>96</sup>. Być może istnieją techniki pracy z widokami mniej uciążliwe dla programisty jednak nie udało mi odszukać informacji na ten temat.

Całą funkcjonalność klasy *ThymeleafBuilder* można opisać fragmentem kodu:

```
ModelAndView view = new ModelAndView(„<nazwa strony>”);
view.addObject(„<nazwa atrybutu>”, „<obiekt lub lista do zwrócenia>”);
...
return view;
```

Klasa *ModelAndView* pozwala na najbardziej skondensowany z wymienionych zapis obsługi widoku. W konstruktorze definiujemy nazwę strony, gdzie przedstawimy zawartość obiektów, które zostaną przekazane w postaci atrybutów. Wygląd i działanie samych szablonów daje pewną przewagę bibliotece Thymeleaf nad wcześniejszymi tego typu rozwiązaniami jak JSP. Wszystko jednak bazuje na

---

95 <https://www.baeldung.com/spring-mvc-model-model-map-model-view> (odwiedzono dn. 23.03.2020) „Model, ModelMap i ModelAndView w Spring MVC” (aktualizacja 21.03.2020).

96 <https://quarkus.io/> (odwiedzono dn. 23.03.2020) Framework posiada możliwość hot-reload (kompilacji Javy z wyprzedzeniem AOT). Chodzi o przekształcenie kodu bajtowego maszyny wirtualnej (VM) na kod maszynowy. Optymalizacje oferowane przed środowisko uruchomieniowe, skracają czas oczekiwania na przetadowanie programu, do tego stopnia, że programista nie odczuwa wpływu czasu.

podobnej koncepcji. Strategii wzorca MVC, gdzie postawiono nacisk na rozdzielenie warstwy logiki biznesowej aplikacji od widoku. Zrozumienie działania Thymelaf nie stanowi wyzwania, trudność pojawia się w momencie konieczności zaprojektowania dużej liczby podstron z widokami. Transferując działanie logiki aplikacji do widoków łatwo się pomylić. Mimo względnej prostoty technika tworzenia interfejsu webowego z użyciem szablonów nie przypadła mi do gustu.

```
1 <table>
2   <tr th:each="owners : ${owners}">
3     <td th:text="${owners.id}"></td>
4     <td th:text="${owners.name}"></td>
5     <td th:text="${owners.surname}"></td>
6     <td>
7       <a th:href="@{/web/redirect-to-add-payment/{id}(id=${owners.id})}">
8         <button type="button">Dodaj płatność</button>
9       </a>
10    </td>
11    <td>
12      <a th:href="@{/web/owner-payments/{id}(id=${owners.id})}">
13        <button type="button">Lista płatności</button>
14      </a>
15    </td>
16  </tr>
17 </table>
18
19 <form action="#" th:action="@{/web/owners}" th:object="${owner}" method="post">
20   <input class="form-control" th:field="*{name}" id="name" />
21   <input class="form-control" th:field="*{surname}" id="surname" />
22   <button type="submit">Wyślij</button>
23 </form>
24
25 <form action="#" th:action="@{/web/edit-payment/{id}(id=${payment.id})}" th:object="${payment}" method="post">
26   <input class="form-control" th:value="${id}" th:field="*{id}" readonly id="id" />
27   <input class="form-control" th:value="${kind}" th:field="*{kind}" id="kind" />
28   <button type="submit" class="btn btn-primary">Wyślij</button>
29 </form>
```

**Ilustracja 22.** Fragmenty szablonów z projektu bez instrukcji służących do zmiany wyglądu HTML-a.

W celu zdefiniowania klarownego wyglądu webowego GUI skorzystałem z biblioteki Bootstrap' a<sup>97</sup> w postaci gotowych klas stylów CSS, co znacznie ułatwiło i przyspieszyło pracę. Spring nie daje wsparcia od strony czysto artystyczno-projektowej, niemniej by wykorzystać style CSS czy dodatkowe biblioteki w postaci JavaScript' u w sposób niestandardowy, potrzebna jest dodatkowa konfiguracja.<sup>98</sup>

97 <https://getbootstrap.com/docs/4.4/getting-started/introduction/> (odwiedzono dn. 23.03.2020) Bootstrap 4.4 - dokumentacja.

98 <https://memorynotfound.com/adding-static-resources-css-javascript-images-thymeleaf/> (odwiedzono dn. 23.03.2020) „Adding Static Resources (css, JavaScript, Images) to Thymeleaf” (aktualizacja 02.11.2017)

## Manipulacja DOM przy pomocy jQuery

### Po stronie serwera, po stronie klienta

W poprzednim podrozdziale opisałem sposoby tworzenia webowego GUI. Zarówno interfejs programistyczny jak i obsługę widoku w postaci stron HTML napisałem Javie. Zrealizowałem kolejne funkcjonalności przy użyciu bibliotek oferowanych przez Spring Framework.

REST-owe API może zostać stworzone także w innych językach takich jak: C# (ASP.Net), Python czy PHP. Faworytem jest JavaScript. Język ten, podobnie jak CSS i HTML, działa po stronie przeglądarki. Pozwala na szybką i łatwą manipulację strukturą dokumentów opartych o znaczniki. Dodatkowo, istnieją obecnie rozwiązania, które pozwalają wykorzystać JavaScript, jako technologię po stronie serwera. Ujednolicenie tworzenia aplikacji wokół jednego języka ułatwia integrację warstw - widoku oraz logiki biznesowej<sup>99</sup>.

REST-owe Web API można obsłużyć przez aplikacje kliencką napisaną w dowolnym języku, który pozwala tworzyć programy z dostępem do Sieci<sup>100</sup>, jednak tylko JavaScript jest interpretowany przez mechanizmy przeglądarki internetowej.<sup>101</sup> Technologia ta leży u podstaw usługi WWW została stworzona nadać dynamizmu statycznym elementom strony. Nadaje się nie tylko do tworzenia animacji, zapewnienie obsługi zdarzeń, sprawdza się przy walidacji formularzy czy budowaniu elementów nawigacyjnych.

Chociaż JavaScript znacznie upraszcza pewne procesy nie można uznać tej technologii za panaceum na wszystkie problemy. Niezależnie od wyboru wsparcia dla języka: biblioteki, projektu, frameworka, nie zmieni to faktu, iż jest on jednowątkowy. Stąd pojawienie się rozwiązań takich jak AJAX, pomysł wprowadzenia asynchroniczności. Warto w tym miejscu przypomnieć, że model REST opiera się na komunikacji bezstanowej i korzysta z możliwości cache'owania danych efektem, czego są duże korzyści optymalizacyjne. Wzrost wydajności obsługi żądań przez klientów.<sup>102</sup> Technologia sprawdza przy wykorzystaniu architektury REST. Przy tworzeniu dużych projektów liczy się jednak bezpieczeństwo, stabilność i łatwość konserwacji, czego wsparcie zdecydowanie lepiej zapewnia technologia taka jak Java. JavaScript wygrywa na polu zastosowań front-end'owych.

---

99 <https://www.codementor.io/@olatundegaruba/nodejs-restful-apis-in-10-minutes-q0sgsfhbd> (odwiedzono 23.03.2020) - Olatunde Michael Garuba, „Build Node.js RESTful APIs in 10 Minutes” (aktualizacja 18.09.2017)

100 Obecnie większość języków wysokiego poziomu posiada frameworki do tworzenia aplikacji webowych. Aplikację internetową można napisać także w czystym C - w tym wypadku rośnie jednak stopień trudności i narzut pracy. Więcej informacji: <https://medium.com/@lucperkins/web-development-in-c-crazy-or-crazy-like-a-fox-ff723209f8f5> (odwiedzono 23.03.2020) - Luc Perkins, „Web development in C: crazy? Or crazy like a fox?” (aktualizacja 17.07.2013)

101 <https://sekurak.pl/jak-dzialaja-wspolczesne-przegladarki-internetowe/> (odwiedzono 24.03.2020) Rafał Janicki - „Jak działają współczesne przeglądarki internetowe” (aktualizacja 09.07.2013)

102 <http://sebastianczech.com/2017/02/20/cechy-dobrego-restful-api/> (odwiedzono 24.03.2020) Sebastian Czech - „Cechy dobrego Restful Api” (20.02.2017)

## Konsumowanie API

W projekcie wykorzystałem możliwość stworzenia widoku przy pomocy jQuery bez wykorzystania dodatkowego pliku kontrolera HTTP oraz biblioteki Thymeleaf. Repozytorium posiada dodatkową gałąź rozwoju, która pozwala zobaczyć efekt pracy<sup>103</sup>. Wersje projektu z obsługą logiki Spring' a przy pomocy JavaScript' u potraktowałem marginalnie. Widok obsługuje tylko niewielką część API napisanego w Springu, nie wykorzystuje Bootstrap 'a do uatrakcyjnienia webowego GUI.

Funkcje napisane przy pomocy jQuery, biblioteki która znacznie upraszcza korzystanie z funkcjonalności JavaScript 'u, konsumują w sposób bezpośredni dane z endpointów. Informacje w postaci JSON zostają po stronie klienta zmienione na reprezentacje w formie znaczników HTML. JavaScript operuje na strukturze drzewa DOM, identyfikuje elementy po klasie lub identyfikatorze i przebudowuje dokument zgodnie z tym, co zostało określone w ciele funkcji. Działanie języka opiera o występujące zdarzenia np. ruch myszką, kliknięcie. Opiera się o wywołania anonimowych funkcji, które następują na różnych etapach cyklu działania strony.

W skryptach konsumujących API wykorzystałem funkcję `ready`, której celem sprawdzenie czy drzewo dokumentu zostało załadowane. W pliku `owner_payments_list` użyłem prostej obsługi zdarzenia, celem przekazania do endpoint' a: **`http://localhost:8080/api/db/owners/{id}/payment`** konkretnej wartości. Numeru id użytkownika, którego płatności mają zostać pokazane.

Rolę nadrzędną przy przetwarzaniu danych w formacie JSON-a, odgrywa funkcja `jQuery.ajax()`<sup>104</sup>, która umożliwia na rozbudowaną obsługę komunikacji serwera z klientem. Mamy możliwość wyboru m.in typu połączenia (metody HTTP), typu danych w odpowiedzi dla klienta (Content-Type), adresu URI endpoint 'a API, typu danych wysyłanych do serwera. Wystarczy opanowanie fragmentu dokumentacji by z powodzeniem realizować obsługę widoków<sup>105</sup>

Widok zrobiony w JavaScript nie musi znajdować w tym samym miejscu, co cała logika aplikacji napisana w Springu. Funkcje w skryptach nie wiedzą nic o istniejących obiektach Javy, potrzebują tylko informacji o lokalizacji zasobów. Warstwa aplikacji z webowym GUI działa niezależnie może zostać przeniesiona na dowolne urządzenie. Pozwala to na pełne rozproszenie oprogramowania, niezależne skalowanie bez wchodzenia w kolizję z logiką aplikacji. Rozwiązanie, które opisałem w poprzednim podrozdziale narzucało konieczność dopasowania kontrolera, a model

---

103 Polecenie systemu kontroli GIT – `git checkout widok-jquery` przełącza na gałąź, który dotyczy widoku zrobionego przy wykorzystaniu JavaScript'u.

104 <http://kursjs.pl/kurs/jquery/jquery-ajax.php> (odwiedzono 24.03.2020), „jQuery - Ajax” (aktualizacja 08.02.2020)

105 <https://api.jquery.com/jQuery.ajax/> (odwiedzono 24.03.2020), jQuery.ajax – dokumentacja.

REST jest wykorzystywany, by w pełni odseparować poziomy aplikacji umożliwiając tym samym stworzenie wielu widoków na różne urządzenia.

### Lista Użytkowników

|   |           |            |
|---|-----------|------------|
| 1 | Adam      | Cecherz    |
| 2 | Agnieszka | Lasota     |
| 5 | Adam      | Kamisz     |
| 6 | Marcin    | Kos        |
| 8 | Marek     | Konieczny  |
| 9 | Adam      | Wolniewicz |

### Lista wszystkich płatności

|    |       |                   |                              |
|----|-------|-------------------|------------------------------|
| 7  | 7     | sex               | 2020-03-19T17:14:40.000+0000 |
| 8  | 188   | toys              | 2020-03-21T10:24:16.000+0000 |
| 9  | 15    | holidays          | 2020-03-19T17:16:47.000+0000 |
| 10 | 67.22 | batonik           | 2020-03-19T18:59:44.000+0000 |
| 13 | 67.9  | home              | 2020-03-20T14:25:37.000+0000 |
| 15 | 33    | home              | 2020-03-19T20:44:08.000+0000 |
| 18 | 67.33 | food              | 2020-03-21T16:22:19.000+0000 |
| 19 | 7     | food              | 2020-03-20T12:27:14.000+0000 |
| 20 | 7.56  | kielbasa          | 2020-03-20T13:52:16.000+0000 |
| 21 | 10.45 | proszek do prania | 2020-03-20T13:53:50.000+0000 |
| 22 | 33    | food              | 2020-03-20T14:55:15.000+0000 |

### Lista płatności

|    |       |                   |                              |
|----|-------|-------------------|------------------------------|
| 7  | 7     | sex               | 2020-03-19T17:14:40.000+0000 |
| 8  | 188   | toys              | 2020-03-21T10:24:16.000+0000 |
| 9  | 15    | holidays          | 2020-03-19T17:16:47.000+0000 |
| 10 | 67.22 | batonik           | 2020-03-19T18:59:44.000+0000 |
| 15 | 33    | home              | 2020-03-19T20:44:08.000+0000 |
| 20 | 7.56  | kielbasa          | 2020-03-20T13:52:16.000+0000 |
| 21 | 10.45 | proszek do prania | 2020-03-20T13:53:50.000+0000 |

### Lista płatności dla użytkownika

Podaj numer użytkownika

2

Pokaż

*Ilustracja 23. Widok aplikacji napisany z wykorzystaniem JavaScript*

Dodaj Użytkownika

Lista użytkowników

Lista wszystkich płatności

| Id | Rodzaj | Wartość           | Data                  |
|----|--------|-------------------|-----------------------|
| 7  | 7.0    | sex               | 2020-03-19 18:14:40.0 |
| 8  | 188.0  | toys              | 2020-03-21 11:24:16.0 |
| 9  | 15.0   | holidays          | 2020-03-19 18:16:47.0 |
| 10 | 67.22  | batonik           | 2020-03-19 19:59:44.0 |
| 13 | 67.9   | home              | 2020-03-20 15:25:37.0 |
| 15 | 33.0   | home              | 2020-03-19 21:44:08.0 |
| 18 | 67.33  | food              | 2020-03-21 17:22:19.0 |
| 19 | 7.0    | food              | 2020-03-20 13:27:14.0 |
| 20 | 7.56   | kielbasa          | 2020-03-20 14:52:16.0 |
| 21 | 10.45  | proszek do prania | 2020-03-20 14:53:50.0 |
| 22 | 33.0   | food              | 2020-03-20 15:55:15.0 |

Filtrowanie płatności po rodzaju

Rodzaj płatności

kind

Filtruj

Filtrowanie płatności po wartościach granicznych

Wysokość płatności

min

max

Filtruj

Odśwież

Dodaj Użytkownika

Lista użytkowników

Lista wszystkich płatności

| Id | Imię      | Nazwisko   | #                         | #                          | #                 | #                           |
|----|-----------|------------|---------------------------|----------------------------|-------------------|-----------------------------|
| 1  | Adam      | Cecherz    | <div>Dodaj płatność</div> | <div>Lista płatności</div> | <div>Edytnj</div> | <div>Usuń użytkownika</div> |
| 2  | Agnieszka | Lasota     | <div>Dodaj płatność</div> | <div>Lista płatności</div> | <div>Edytnj</div> | <div>Usuń użytkownika</div> |
| 5  | Adam      | Kamisz     | <div>Dodaj płatność</div> | <div>Lista płatności</div> | <div>Edytnj</div> | <div>Usuń użytkownika</div> |
| 6  | Marcin    | Kos        | <div>Dodaj płatność</div> | <div>Lista płatności</div> | <div>Edytnj</div> | <div>Usuń użytkownika</div> |
| 8  | Marek     | Konieczny  | <div>Dodaj płatność</div> | <div>Lista płatności</div> | <div>Edytnj</div> | <div>Usuń użytkownika</div> |
| 9  | Adam      | Wolniewicz | <div>Dodaj płatność</div> | <div>Lista płatności</div> | <div>Edytnj</div> | <div>Usuń użytkownika</div> |

*Ilustracja 24. Widok aplikacji napisany z wykorzystaniem Thymeleaf*

## Podsumowanie

W pracy scharakteryzowałem cechy modelu Richardson 'a. Opisałem kolejne etapy dojrzałości modelu REST. Przedstawiłem w tym celu przykłady kontrolerów napisanych z wykorzystaniem frameworka Spring. Udało mi się opisać wady i zalety rozwiązań w tworzeniu oprogramowania. Pokazałem stopniową ewolucję webowego API.

Opisałem procesy związane z dodaniem do aplikacji warstwy utrwalania danych. Wyjaśniłem działanie mapper 'ów obiektowo-relacyjnych. W oparciu o narzędzia oferowane przez JPA przebudowałem model klas (beanów), zintegrowałem go z istniejącą bazą danych. Wykorzystując abstrakcje oferowane przez Spring 'a rozbudowałem funkcjonalność kontrolerów. Wykonałem analizę problemów związanych z relacyjną bazą danych w kontekście API.

Umówiłem etap związany z implementacją obsługi błędów w aplikacji. Przedstawiłem narzędzia Spring 'a, które pozwalają kontrolerom HTTP przechwytywać i rzucać wyjątki.

W finalnym rozdziale przedstawiłem propozycję stworzenia webowego GUI dla użytkowników. Opisałem problemy związane z nieprzemyślaną architekturą oprogramowania. Przedstawiłem możliwość wykorzystania dynamicznych szablonów jak i technologii po stronie serwera bazujących na języku JavaScript.

Podczas opisywania zagadnień teoretycznych, prezentowałem przykłady zaczerpnięte z aplikacji własnego autorstwa. Nie udało mi się stworzyć idealnego oprogramowania, ale drobniaczko przedstawiłem liczne obszary, które należy uwzględnić pisząc aplikacje webowe. Praca może być wykorzystywana, jako poradnik, zawiera wskazówki jak unikać złych praktyk.

Aplikacja CALC-BILL stanowi przykład prostego, ale użytecznego programu. Brak wdrożenia takich mechanizmów jak kontrola dostępu czy obsługa transakcji nie odbiera aplikacji możliwości praktycznych zastosowań w celach zarządzania budżetem. Kod programu dzięki zastosowaniu wzorca MVC, stwarza możliwość rozbudowy. Wykorzystanie wzorca REST stwarza możliwość zaprojektowania aplikacji klientów, m.in. na urządzenia mobilne.

Popularność frameworka Spring 'a sprawia, że aplikacja jest łatwa do wdrożenia; uruchomienia w chmurze obliczeniowej. Separacja poziomów daje możliwość niezależnej przebudowy, skalowania warstw. Wykorzystanie Java Persistence API uwalnia od konieczności dbania o integrację bazy danych z klasami języka Javy.

Zakładam otwarcie kolejnego cyklu tworzenia dla mojego projektu, rozbudowę kodu w repozytorium. Dodanie nowych funkcjonalności i usprawnień. Stworzenie aplikacji klienckiej na platformę Android.

## Bibliografia

1. Bhakti Mehta, *REST - Najlepsze praktyki i wzorce w języku Java*, tłum. Łukasz Piwko, wyd. Helion, Gliwice 2015
2. Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, wyd. Prentice Hall, 2002
3. Craig Walls, *Spring w Akcji*, tłum. Mirosław Gołda, Piotr Rajca, wydanie IV, wyd. Helion, Gliwice 2015
4. Robert C. Martin, *Czysty Kod - podręcznik dobrego programisty*, tłum. Paweł Gonera, wyd. Helion, Gliwice 2014
5. Cay S. Horstmann, Gary Cornell, *Java Podstawy*, tłum. Łukasz Piwko, wyd. Helion, Gliwice 2014
6. Arun Gupta, *Java EE 7 Essentials*, wydanie I, wyd. O'Reilly, 2013
7. Krzysztof Barteczko, *Java - Programowanie praktyczne od podstaw*, wydanie I, wyd. PWN, Warszawa 2014
8. Ludovic Dewailly, Raja CSP Raman, *Building RESTful Web Services with Spring 5 - Second Edition*, wyd. Packt Publishing, 2018
9. Christian Bauer, Gavin King, Gary Gregory, *Java Persistence*, tłum. Radosław Meryk *Programowanie aplikacji bazodanowych w Hibernate*, wydanie II, wyd. Helion, Gliwice 2016
10. Geoffroy Warin, *Spring MVC 4. Projektowanie zaawansowanych aplikacji WWW*, tłum. Andrzej Watrak, wyd. Helion, Gliwice 2016
11. Jon Duckett, *JavaScript i jQuery. Interaktywne strony WWW dla każdego. Podręcznik Front-End Developera*, tłum. Robert Górczyński, wyd. Helion, Gliwice 2015
12. Krzysztof Rychlicki-Kicior, *Java EE6 Programowanie aplikacji WWW*, wyd. Helion, Gliwice 2010
13. Włodzimierz Gajda, *PHP Praktyczne Projekty*, wyd. Helion, Gliwice 2009

## Spis ilustracji

|                                                                                                                                            |    |
|--------------------------------------------------------------------------------------------------------------------------------------------|----|
| <b>Ilustracja 1.</b> Schemat opisujący przekazywanie odpowiedzi o oparciu o metody i zasoby URL                                            | 5  |
| <b>Ilustracja 2.</b> Zrzut ekranu z przeglądarki: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:70.0)<br>Gecko/20100101 Firefox/70.0          | 8  |
| <b>Ilustracja 3.</b> Diagram tabel wykonany w programie MySQL Workbench                                                                    | 9  |
| <b>Ilustracja 4.</b> Widok struktury tabel w postaci tekstowej                                                                             | 10 |
| <b>Ilustracja 5.</b> Widok w postaci pliku HTML oraz klasa kontrolera                                                                      | 12 |
| <b>Ilustracja 6.</b> Porównanie technologii opartych na REST oraz SOAP                                                                     | 14 |
| <b>Ilustracja 7.</b> Kod źródłowy: przykład serwisu aplikacji typu Swamp of POX                                                            | 15 |
| <b>Ilustracja 8.</b> Schemat działania API opartego o antywzorzec Swamp of POX                                                             | 16 |
| <b>Ilustracja 9.</b> Diagram UML - relacje klas modelu i kontrolera                                                                        | 18 |
| <b>Ilustracja 10.</b> Tworzenie oprogramowania w metodykach zwinnych.                                                                      | 21 |
| <b>Ilustracja 11.</b> Obiekt odpowiedzi i żądania w HTTP                                                                                   | 24 |
| <b>Ilustracja 12.</b> Efekt działania metody showHeaders                                                                                   | 25 |
| <b>Ilustracja 13.</b> Efekt działania metod getAllOwners oraz getAllPayments                                                               | 26 |
| <b>Ilustracja 14.</b><br>a) Przykład serwisu bez spersonalizowanej strony błędu dla statusu 404;<br>b) przykład serwisu, który ją posiada. | 28 |
| <b>Ilustracja 15.</b> Ilustracja struktury HAL                                                                                             | 32 |
| <b>Ilustracja 16.</b> Uprozczone wyniki przykładowych zapytań HTTP dla API w wersji 3                                                      | 35 |
| <b>Ilustracja 17.</b> Opis działania REST-API napisanego Spring Framework                                                                  | 37 |
| <b>Ilustracja 18.</b> Architektura Spring Data, podział warstw                                                                             | 43 |
| <b>Ilustracja 19.</b> Informacje, które zwraca API w momencie rzucenia wyjątku DataIntegrityException                                      | 45 |
| <b>Ilustracja 20.</b> REST API na różnych urządzeniach                                                                                     | 49 |
| <b>Ilustracja 21.</b> Diagram klas aplikacji w wersji finalnej z widokiem                                                                  | 51 |
| <b>Ilustracja 22.</b> Fragmenty szablonów z projektu bez instrukcji służących do zmiany wyglądu HTML-a.                                    | 53 |
| <b>Ilustracja 23.</b> Widok aplikacji napisany z wykorzystaniem JavaScript                                                                 | 56 |
| <b>Ilustracja 24.</b> Widok aplikacji napisany z wykorzystaniem Thymeleaf                                                                  | 56 |

## Spis tabel

|                                                                                             |    |
|---------------------------------------------------------------------------------------------|----|
| <b>Tabela 1.</b> Opis metod do wyszukiwania elementów kolekcji przy wykorzystaniu HTTP.GET  | 19 |
| <b>Tabela 2.</b> Najważniejsze kody odpowiedzi HTTP                                         | 30 |
| <b>Tabela 3.</b> Opis typów w kolejnych wersjach API                                        | 33 |
| <b>Tabela 4.</b> Metody wyszukiujące w klasach rozszerzających CrudRepository               | 39 |
| <b>Tabela 5.</b> Typy wyjątków rzuconych przez API pakiet pl.cecherz.calcbill.controller.db | 45 |