Spring & Spring Boot Framework

Agenda

1. Krótko o całym dziale Spring
2. Czym jest Spring Framework
3. Czym jest biblioteka, a czym framework?
4. Komponenty Spring – czym są?
5. Spring Boot vs Spring
6. Podstawy tworzenia projektu w Spring Boot
7. Omówinie pliku pom.xml
8. Szczegółowe omówienie komponentów
9. Budowanie aplikacji z komponentów Spring
10. Omówienie adnotacji Spring na podstawie prostych przykładów
11. Dostarczanie serwisu z funkcjonalnością CRUD
12. Programowanie aspektowe
13. Czym jest DTO i jak go użytwać?
14. Konfiguracja oraz adnotacje bazodanowe – JPA

Krótko o dziale i prezentacji

W tym bloku nauczymy się tworzyć aplikacje Web'owe oraz serwisy, które mają dostarczać dane. Rozwiniemy teorię dwóch modeli dostarczanych aplikacji – skupimy się na początku na grubym kliencie (gdzie zbudujemy aplikację backend'ową, oraz frontend do niej), a następnie przejdziemy do budowania tej samej aplikacji w modelu cienkiego klienta (czyli stworzenie jednej aplikacji z wbudowanymi widokami.

W bloku należy skupić się głównie na zasadach którymi należy się kierować w budowaniu aplikacji. Będziemy pisali aplikację rozwijając ją aspektowo. Nauczymy się implementować funkcje pod kątem realizacji pewnych funkcjonalności.

Pod koniec tego bloku powinniśmy być w stanie odpowiedzieć sobie na kilka pytań, które powinny się pojawić przed rozpoczęciem pisania projektu końcowego.

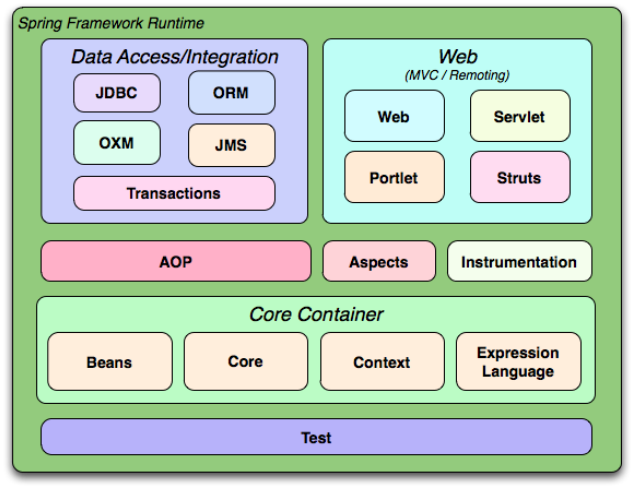
- Jaki model aplikacji wybieram? Cienki czy gruby klient?
- Czy swoją aplikację będę pisać w posługując się frameworka (Spring, Spring Boot), czy używając Servletów Java EE?
- Jakiej bazy będę używać? Jakiego rozwiązania do zarządzania nią (JPA, Hibernate)?
- W jakiej technologii chcę rozwijać swój frontend (Vaadin, Angular 2/4/5, Thymeleaf, JSP)?
- Jakie będą modele którymi będę manipulować w swojej aplikacji?
- Jaką funkcjonalność chcę dostarczyć na koniec projektu, a jaką postaram się rozwinąć jeśli starczy mi czasu?

Czym jest Spring Framework?

W świecie w którym panuje nieustanny rozwój każda chwila ma ogromne znaczenie. Development oprogramowania bywa czasochłonny i złożony, a każdemu zależy na czasie i jak najszybszym dostarczaniu nowych rozwiązań. Projektowanie aplikacji i ich dostarczanie to długi proces, zatem zaistniała potrzeba jego skrócenia.

Spring Framework to platforma napisana w Javie, której wykorzystanie zapewnia nam uniwersalną infrastrukturę dla rozwijania wszelkiego rodzaju aplikacji. Zamysłem spring'a było stworzenie infrastruktury w taki sposób, aby programista mógł zająć się rozwojem aplikacji.

Framework zapewnia nam kilka podstawowych funkcjonalności, między innymi Inversion Of Control lub Dependency Injection. Całe rozwiązanie można podzielić na moduły i komponenty.



Czym jest biblioteka, a czym framework?

Skupimy się najpierw na wyjaśnieniu słowa framework. Żeby to zrobić zaczniemy od słowa biblioteka (software'owa).

Biblioteka – to gotowe rozwiązanie które zostało stworzone w celu zapewnienia/dostarczenia pewnej funkcjonalności. Np. biblioteka ***libusb*** służy do zarządzania i kontrolowania portami usb, a biblioteka ***libconfig++*** służy do tworzenia i manipulowania plikami konfiguracyjnymi w naszej aplikacji. Tym samym – bibliteka realizuje wyłącznie jedną małą funkcjonalność którą możemy dodać do swojej aplikacji.

Framework – to gotowe rozwiązanie, które jest na tyle kompletne że możemy je postrzegać jako szablon aplikacji. Frameworki zostały stworzone w celu wyeliminowania potrzeby tworzenia infrastruktury/architektury aplikacji i aby dostarczyć rozwiązanie które w szybki sposób pozwoli stworzyć aplikację. Przykładem frameworków jest Spring framework, oraz np. Android Framework. Szablon aplikacji frameworkowej składa się z wielu komponentów i modułów które zarządzają np. czasem życia obiektów w aplikacji. Tworząc aplikację skupiamy się na implementacji niezbędnej funkcjonalności.

Framework to znacznie większe rozwiązanie i bardzo często zbudowane jest z wielu bibliotek.

Komponenty w Spring – czym są?

Komponenty w spring zapewniają podział odpowiedzialności i funkcjonalności. Tworzenie komponentów odbywa się poprzez dodanie nad definicją klasy adnotacji (jednej z):

- Component
- Service
- Controller
- RestController
- Repository

Po utworzeniu klasy z jedną z podanych adnotacji, w kontenerze aplikacji zostanie stworzona instancja tego typu. Obiekty są ogólno dostępne w aplikacji i ich podział ma charakter podkreślenia funkcji w projekcie.

Spring Boot vs Spring

Spring framework to rozbudowany projekt który posiada spore możliwości. Do stworzenia projektu w Spring'u należy po utworzeniu projektu skonfigurować wiele opcji, takich jak – odnaleźć komponenty aplikacji, wskazać klasy konfiguracyjne, utworzyć model MVC, skonfigurować klasy widoków i wskazać folder z zasobami, dodać konfigurację bazy danych, utworzyć połączenie...

Możnaby powiedzieć i tak dalej...

Spring Boot to to samo co Spring jednak bez konieczności konfigurowania wszystkiego od zera. Wszystkie konfiguracje które można nazwać podstawowymi konfiguracjami stanowią bazę projektu. Dzięki użyciu Spring Boota nie musimy poświęcać dodatkowych godzin pracy na konfigurację Spring'a, a możemy od razu przejść do implementacji funkcjonalności.

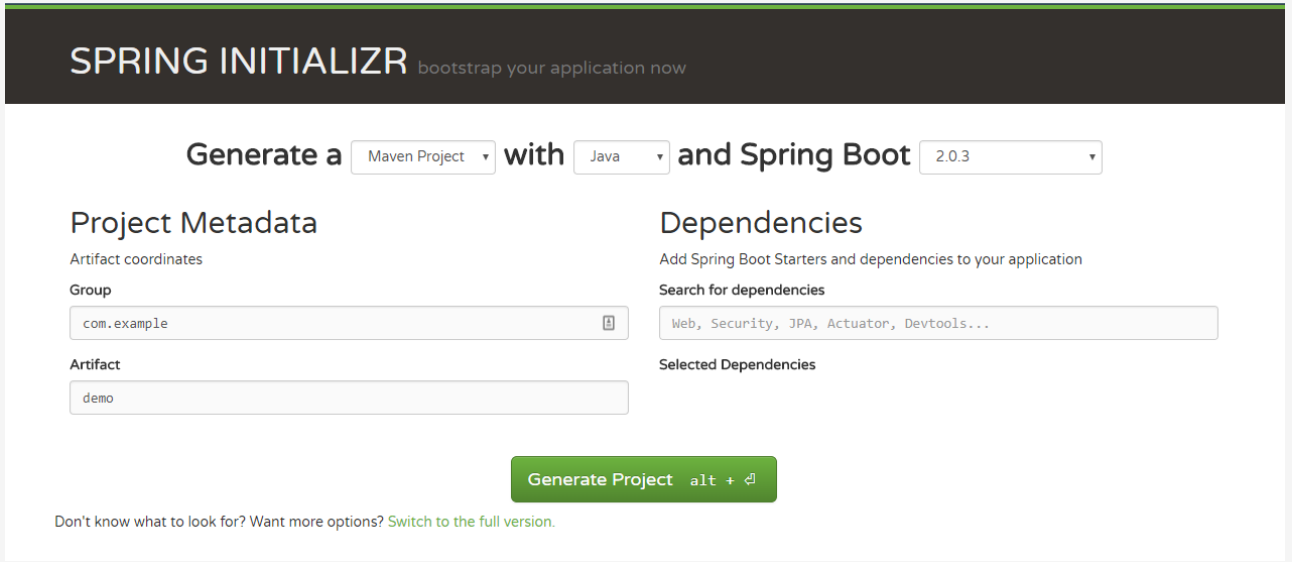
Możnaby powiedzieć, że Spring Boot zawiera bardziej kompletny szablon projektu z ustawioną domyślną konfiguracją dla projektów Web'owych i Rest'owych (i nie tylko).

Dla zainteresowanych instrukcja step-by-step jak skonfigurować Spring'a z ‚odrobiną' angular'a JS:

https://drive.google.com/drive/folders/1263NhzD_AuExQH7cFx-5WMufrr6801QX?usp=sharing

Podstawy tworzenia projektu Spring Boot

Do stworzenia projektu Spring Boot możemy się posłużyć IntelliJ'em lub wykonać to samo poprzez stronę internetową: https://start.spring.io/



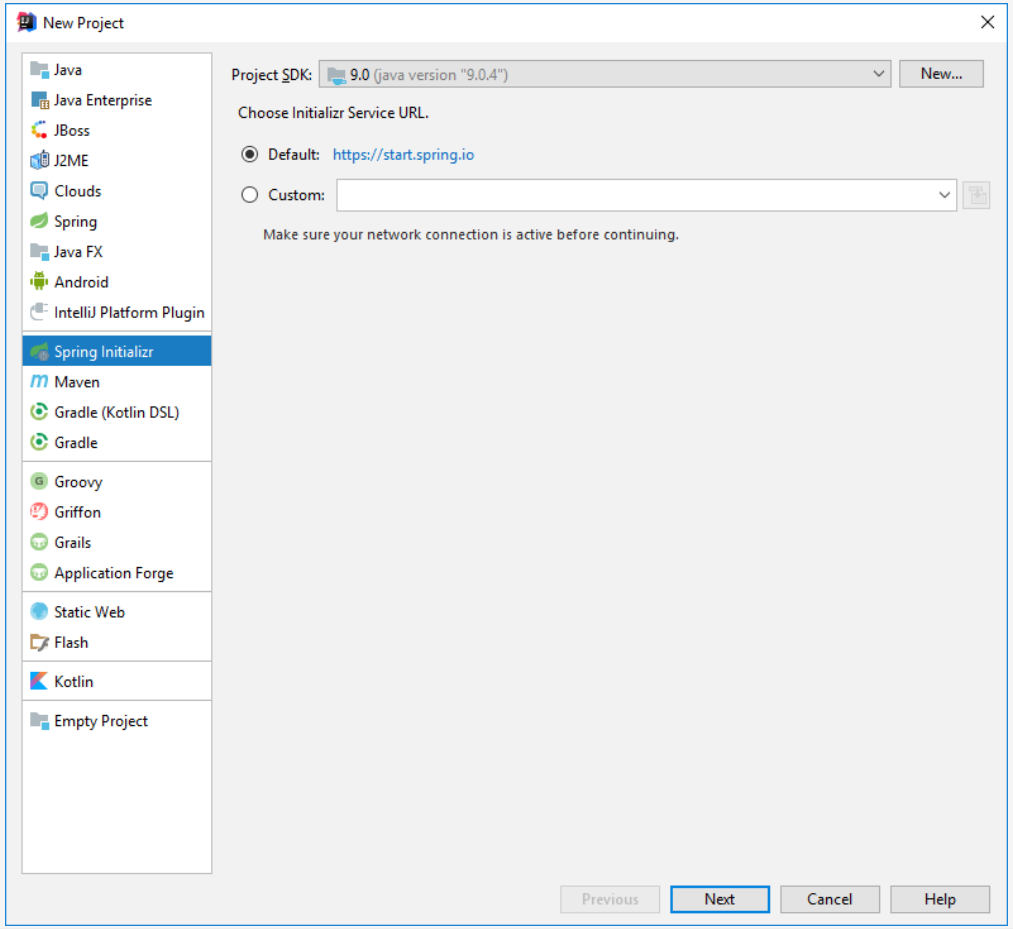
Podstawy tworzenia projektu Spring Boot

Podczas kursu zaprezentowana jest wersja generowania projektu za pomocą IntelliJ, któa jest zbliżona do generowania projektu na stronie internetowej.

Podstawy tworzenia projektu Spring Boot

Podczas kursu zaprezentowana jest wersja generowania projektu za pomocą IntelliJ, któa jest zbliżona do generowania projektu na stronie internetowej.

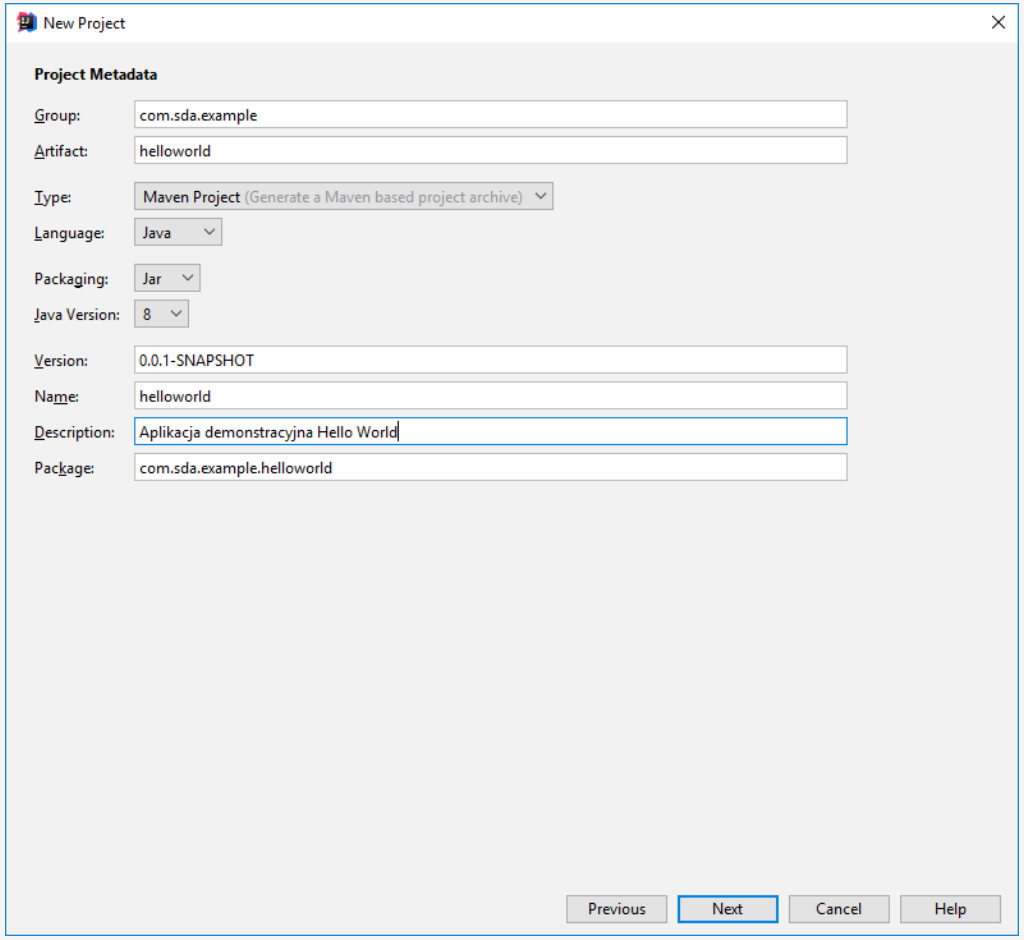
1. **File** -> **New**... -> **Project**
2. Z listy wybierz **Spring Initializr**
3. Wybieramy odpowiedniąwersję **SDK** i kilkamy **Next**



Podstawy tworzenia projektu Spring Boot

Podczas kursu zaprezentowana jest wersja generowania projektu za pomocą IntelliJ, któa jest zbliżona do generowania projektu na stronie internetowej.

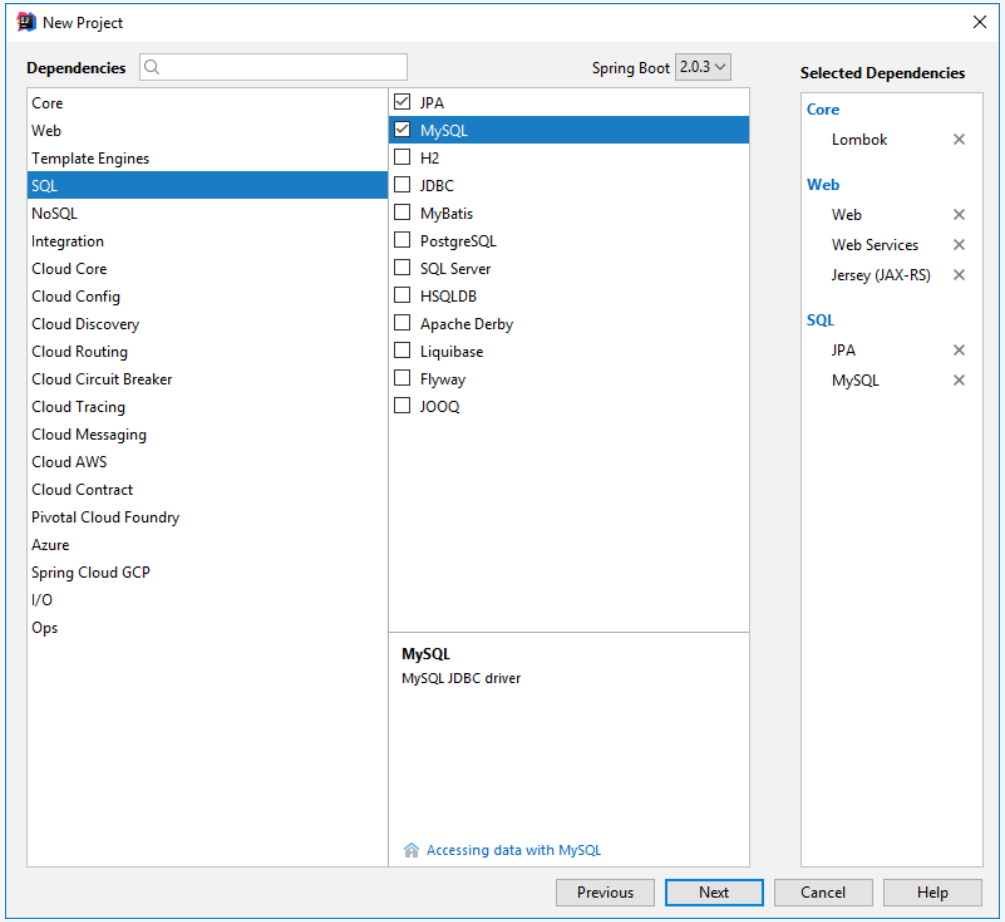
1. **File** -> **New**... -> **Project**
2. Z listy wybierz **Spring Initializr**
3. Wybieramy odpowiedniąwersję **SDK** i kilkamy **Next**
4. W formularzu **podajemy informacje** dotyczące **naszego projektu**
5. Klikamy **Next**



Podstawy tworzenia projektu Spring Boot

Podczas kursu zaprezentowana jest wersja generowania projektu za pomocą IntelliJ, któa jest zbliżona do generowania projektu na stronie internetowej.

1. **File** -> **New**... -> **Project**
2. Z listy wybierz **Spring Initializr**
3. Wybieramy odpowiedniąwersję **SDK** i kilkamy **Next**
4. W formularzu **podajemy informacje** dotyczące **naszego projektu**
5. Klikamy **Next**
6. Z listy wybieramy **komponenty** którymi będziemy się posługiwać podczas pracy nad projektem (na liście obok – po prawej – widoczna jest lista komponentów do rozwijania projektu REST'owego i WEB'owego)
7. Klikamy **Next**
8. Klikamy **Finish**



Omówienie pliku pom.xml

Projekt który udało nam się stworzyć jest projektem Maven'owym, więc zawiera plik pom.xml. Po pobraniu wszystkich zależności plik pom.xml wygląda następująco:

Pierwsza sekcja jest taka sama jak we wcześniejszych projektach. Zawiera podstawowe informacje o projekcie.

```
<groupId>com.sda.example</groupId>
<artifactId>helloworld</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>helloworld</name>
<description>Aplikacja demonstracyjna Hello World</description>
```

Omówienie pliku pom.xml

Projekt który udało nam się stworzyć jest projektem Maven'owym, więc zawiera plik pom.xml. Po pobraniu wszystkich zależności plik pom.xml wygląda następująco:

Kolejna sekcja zawiera definicję projektu nadrzędnego – mamy tutaj deklarację projektu Spring Boot w wersji **2.0.3**. Oznaczamy tym samym, że nasz kod to fragment implementacji, któej dużą część stanowi szablon **spring boot**.

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.0.3.RELEASE</version>
   <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Omówienie pliku pom.xml

Projekt który udało nam się stworzyć jest projektem Maven'owym, więc zawiera plik pom.xml. Po pobraniu wszystkich zależności plik pom.xml wygląda następująco:

Kolejna sekcja zawiera definicję projektu nadrzędnego – mamy tutaj deklarację projektu Spring Boot w wersji **2.0.3**. Oznaczamy tym samym, że nasz kod to fragment implementacji, któej dużą część stanowi szablon **spring boot**.

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.0.3.RELEASE</version>
   <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Kolejna sekcja zawiera definicję zmiennych pliku pom:

```
<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
   <java.version>1.8</java.version>
</properties>
```

Omówienie pliku pom.xml

Projekt który udało nam się stworzyć jest projektem Maven'owym, więc zawiera plik pom.xml. Po pobraniu wszystkich zależności plik pom.xml wygląda następująco:

Ostatnia istotna częśc projektu to zależności, które zostały dobrane na podstawie wybranych przez nas komponentów Spring'a.

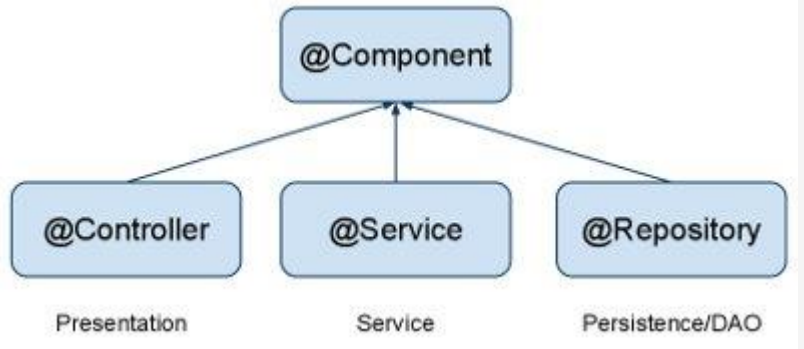
Mamy tu zależności związane z protokołem HTTP, z JSON'em, JPA (baza danych) oraz project lombok i zawsze dołączane – testy.

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-jersey</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web-services</artifactId>
   </dependency>

   <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <scope>runtime</scope>
   </dependency>
   <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
   </dependency>
</dependencies>
```

Omówienie komponentów

Aplikacje Spring'owe są projektowane z komponentów. Komponenty to **Bean'y** które po utworzeniu są konfigurowane, a następnie zarządzane przez framework. Każdy komponent aplikacji musi być oznaczony, aby później został umieszczony w kontenerze. Do oznaczania komponentów korzystamy z adnotacji (aby nie przechodzić z Java do XML).

@Configuration – oznaczenie dla klasy konfiguracyjnej. Zbliżony do adnotacji Bean. Klasy konfiguracji są ładowane podobnie jak w przypadku servlet'owych aplikacji były ładowane inne klasy i pliki konfiguracyjne (np. Xml'e). Wewnątrz klasy możemy zawrzeć Bean'y.



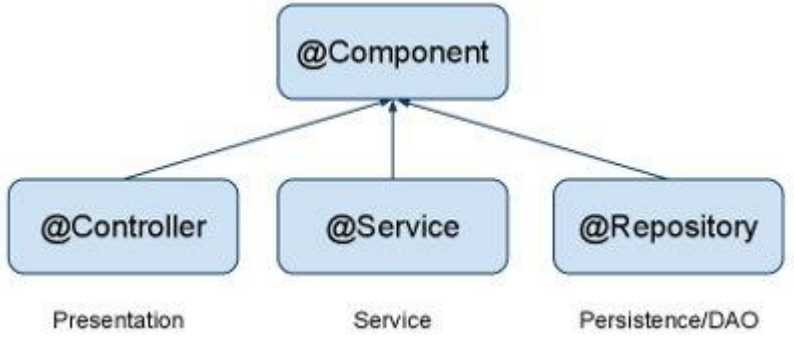
Omówienie komponentów

Aplikacje Spring'owe są projektowane z komponentów. Komponenty to **Bean'y** które po utworzeniu są konfigurowane, a następnie zarządzane przez framework. Każdy komponent aplikacji musi być oznaczony, aby później został umieszczony w kontenerze. Do oznaczania komponentów korzystamy z adnotacji (aby nie przechodzić z Java do XML).

@Configuration – oznaczenie dla klasy konfiguracyjnej. Zbliżony do adnotacji Bean. Klasy konfiguracji są ładowane podobnie jak w przypadku servlet'owych aplikacji były ładowane inne klasy i pliki konfiguracyjne (np. Xml'e). Wewnątrz klasy możemy zawrzeć Bean'y.

```java
package com.tutorialspoint;
import org.springframework.context.annotation.*;

@Configuration
public class HelloWorldConfig {
   @Bean
   public HelloWorld helloWorld(){
      return new HelloWorld();
   }
}
```

```xml
<beans>
   <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" />
</beans>
```

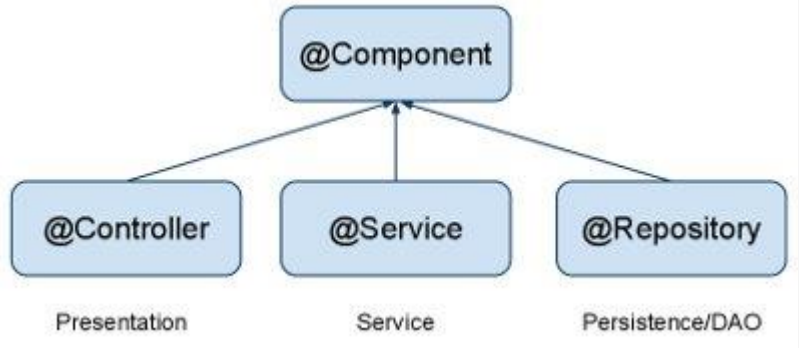


Omówienie komponentów

Aplikacje Spring'owe są projektowane z komponentów. Komponenty to **Bean'y** które po utworzeniu są konfigurowane, a następnie zarządzane przez framework. Każdy komponent aplikacji musi być oznaczony, aby później został umieszczony w kontenerze. Do oznaczania komponentów korzystamy z adnotacji (aby nie przechodzić z Java do XML).

@Configuration – oznaczenie dla klasy konfiguracyjnej. Zbliżony do adnotacji Bean. Klasy konfiguracji są ładowane podobnie jak w przypadku servlet'owych aplikacji były ładowane inne klasy i pliki konfiguracyjne (np. Xml'e). Wewnątrz klasy możemy zawrzeć Bean'y.

@Bean – oznaczenie dla obiektu który po utworzeniu jest zarządzany przez kontener IoC. Po stworzeniu obiektu możemy dodać do niego adnotację Bean, dzięki czemu obiekt zostanie umieszczony w kontenerze, do którego mamy dostęp w dowolnej części naszej aplikacji. Dopisanie adnotacji **Inject'ującej** spowoduje wstrzyknięcie obiektu w zmienną. Zamiast posługiwać się dalej tworzeniem obiektów przez **new** możemy otworzyć Bean'y i je **Autowire'ować.**



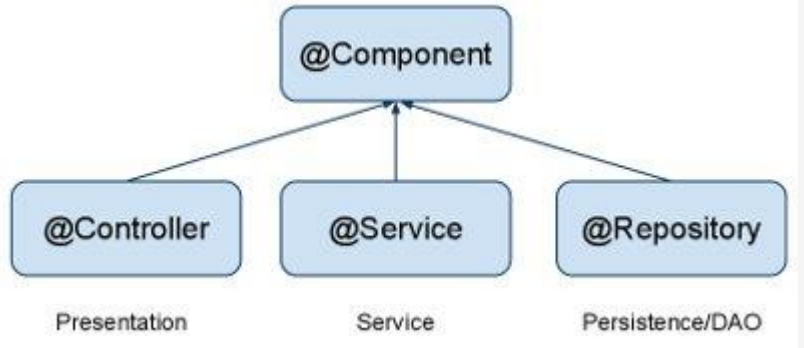
Omówienie komponentów

Aplikacje Spring'owe są projektowane z komponentów. Komponenty to **Bean'y** które po utworzeniu są konfigurowane, a następnie zarządzane przez framework. Każdy komponent aplikacji musi być oznaczony, aby później został umieszczony w kontenerze. Do oznaczania komponentów korzystamy z adnotacji (aby nie przechodzić z Java do XML).

@Configuration – oznaczenie dla klasy konfiguracyjnej. Zbliżony do adnotacji Bean. Klasy konfiguracji są ładowane podobnie jak w przypadku servlet'owych aplikacji były ładowane inne klasy i pliki konfiguracyjne (np. Xml'e). Wewnątrz klasy możemy zawrzeć Bean'y.

@Bean – oznaczenie dla obiektu który po utworzeniu jest zarządzany przez kontener IoC. Po stworzeniu obiektu możemy dodać do niego adnotację Bean, dzięki czemu obiekt zostanie umieszczony w kontenerze, do którego mamy dostęp w dowolnej części naszej aplikacji. Dopisanie adnotacji **Inject'ującej** spowoduje wstrzyknięcie obiektu w zmienną. Zamiast posługiwać się dalej tworzeniem obiektów przez **new** możemy otworzyć Bean'y i je **Autowire'ować.**

@Repository – oznaczenie dla klasy która reprezentuje dla nas klasę DAO – Data Access Object. To obiekt który będzie zapewniał dostęp do bazy danych. W nim będą zawarte wszystkie metody którymi będziemy dodawać, usuwać, edytować czy szukać obiektów w bazie danych poprzez zapytania **SQL** lub inne.

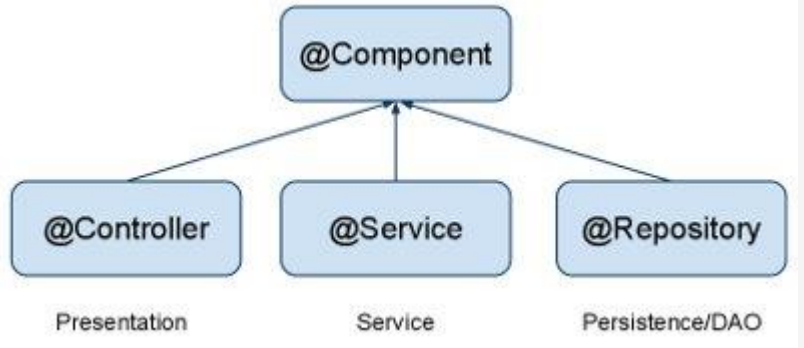


Omówienie komponentów

@Controller – oznaczenie dla klas które reprezentują kontrolery w modelu **MVC.** Kontrolery to klasy które odpowiadają za obsługę wszystkich akcji realizowanych przez klienta/inne aplikacje. Kontrolery zawierają endpoint'y do ktorych dopisujemy ścieżki pod którymi możemy je adresować. Kontrolery pełnią funkcję podobną do servlet'ów. Wyróżniamy adnotację @Controller oraz @RestController. Różnica pomiędzy nimi polega na treści które dostarczają – klasy z adnotacją **Controller** dostarczają widoki, natomiast **RestController** dostarcza usługi w REST'owej.

@Service – komponent reprezentujący obiekt odpowiadający za procesowanie i obsługę danych. Bardzo często zdarza się pomijać ten komponent, co nie jest dobrą praktyką. Logika obsługi powinna być możliwie jak najbardziej przeniesiona na serwis w celu zapewnienia mu oraz kontrolerowi pojedynczej odpowiedzialności.

@Component – adnotacja nadrzędna dla komponentów. Możemy nią oznaczyć pozostałe komponenty które nie wpasowują się w jeden z wymienionych (klasyczny podział).



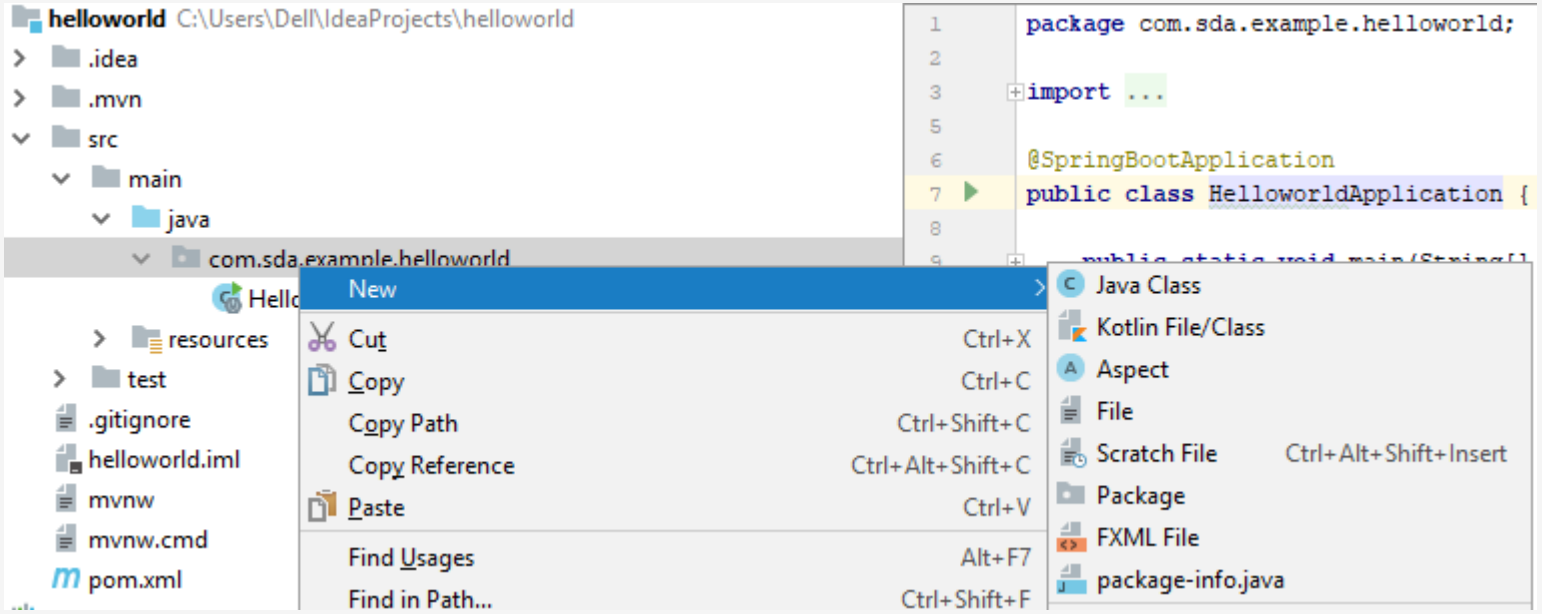
Omówienie komponentów – na przykładzie.

Prosty przykład – aplikacja do dodawania użytkowników.

Rozwój projektu można śledzić na repozytorium: https://bitbucket.org/saintamen/spring-hello/
W odpowiednich momentach prezentacji były wykonywane commity, w ten sposób prościej jest dostrzec przyrostowe różnice w projekcie.

W drzewie projektu stwóz package. Pamiętaj **WSZYSTKIE PACZKI PROJEKTU POWINNY ZNAJDOWAĆ SIĘ W PACZCE PODRZĘDNEJ OD GŁÓWNEJ PACZKI PROJEKTU**. W moim projekcie paczka główna to: com.sda.example.helloworld

Stwórz w ten sposób paczkę:
- configuration
- controller
- model
- service
- repository



Omówienie komponentów – na przykładzie.

W paczce model, dodaj klasę **AppUser** (staraj się nie używać klasy User, ponieważ może się on kłócić z tabelą w mysql).

Stworzona klasa posiada minimum pole **ID** – jest to ważne z punktu widzenia bazy danych. Aby klasa miała odniesienie do bazy danych konieczne jest dopisanie adnotacji @**Entity.** Pozostałe adnotacje pochodzą z biblioteki lombok.

Stworzony został również dodatkowy konstruktor z dwoma parametrami – został stworzony, ponieważ będę wymagał od rejestrującego wyłącznie email'a i hasła do rejestracji, a pozostałe dane pozwolę mu dodać w późniejszym czasie.

```
@Entity
@Data
@AllArgsConstructor
public class AppUser {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    private String email;
    private String password;
    private String name;
    private String surname;

    public AppUser(String email, String password) {
        this.email = email;
        this.password = password;
    }
}
```

Podana zmiana została zatwierdzona commit'em:
https://bitbucket.org/saintamen/spring-hello/commits/41c041ef9873e8742f209119d1cf4acdf3d83cd0

Omówienie komponentów – na przykładzie.

Dodamy teraz konfigurację bazy danych. Aby to zrobić wystarczy otworzyć plik **application.properties** w zasobach projektu.

Parametry konfiguracji (minimum niezbędne) to login, hasło i adres bazy. Dodatkowa opcja ddl-auto powoduje wygenerowanie nowej bazy danych – a w zasadzie jej tabel. **KONIECZNE JEST WCZEŚNIEJSZE STWORZENIE BAZY hello_db ABY TABELE MOGŁY BYĆ WYGENEROWANE.**

```
spring.jpa.hibernate.ddl-auto=create
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.url=jdbc:mysql://localhost:3306/hello_db
```

Podana zmiana została zatwierdzona commit'em:
https://bitbucket.org/saintamen/spring-hello/commits/c00773727e2db0313f6b69a1878ecfe3d6f2b7a6

Omówienie komponentów – na przykładzie.

Dodamy teraz klasę do zarządzania bazą danych. Za te czynności odpowiadają klasy **Repository**. Jedyne co należy stworzyć dla podstawowych operacji bazodanowych (CRUD) to interfejs. Spring Framework stworzy w miejsce interfejsu **Bean** z wymaganą funkcjonalnością, z wygenerowanymi zapytaniami.

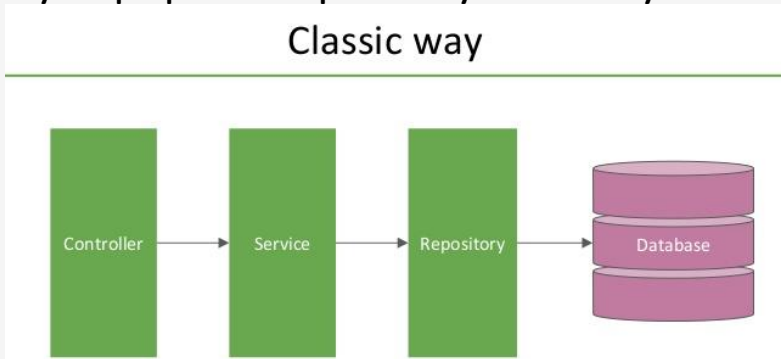
```
@Repository
public interface AppUserRepository extends JpaRepository<AppUser, Long>{
}
```

Podana zmiana została zatwierdzona commit'em:
https://bitbucket.org/saintamen/spring-hello/commits/ce0a862806fd02a38bb07ebe9bbf983ef7fdb3a9

Omówienie komponentów – na przykładzie.

Następna zmiana jest duża. Stworzyliśmy serwis który ma powiązane pole bazy danych. Adnotacja Autowired w Serwisie pozwala wybrać Bean Repository z IoC i przypisać jego instancję do pola klasy.

Podobnie dzieje się w AppUserController – gdzie kontroler powiązany jest z Serwisem. Ten przepływ informacji zapewnia, że kontroler odwoła się zawsze ze swoim zapytaniem do serwisu, a ten ma dostęp do bazy danych poprzez repository. Dzielimy odpowiedzialność i odciążamy kontroler.



Załóżmy że chcę stworzyć instancję użytkownika. Przy dodawaniu go przez postmana musimy użytkownikowi dać możliwość podpięcia się do naszego api – innymi słowy musimy mu powiedzieć jak ma wyglądać obiekt który ma wysyłać – czyli nasz AppUser. (Adnotacja RequestBody – przez którą obiekt podanego typu musi znaleźć się w sekcji body ramki)

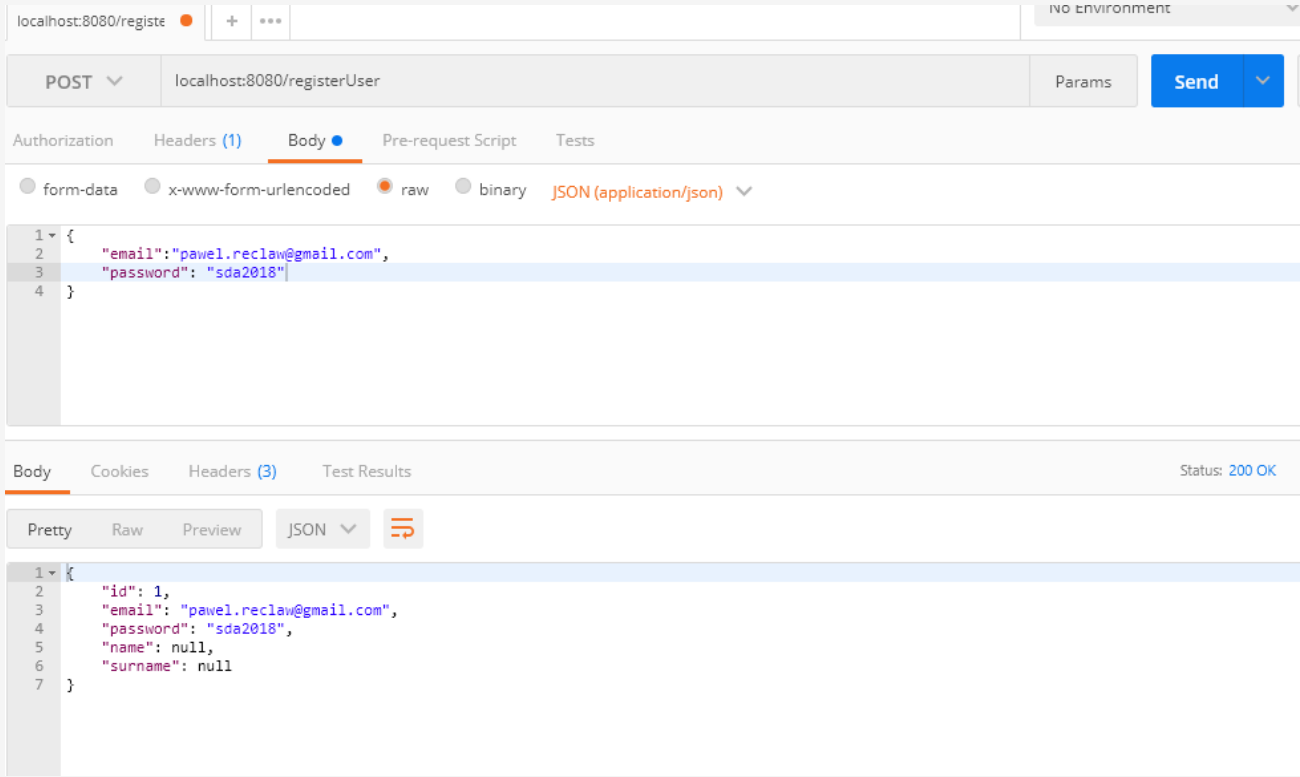
```
@RestController
public class AppUserController {

    @Autowired
    private AppUserService appUserService;

    @PostMapping("/registerUser")
    public ResponseEntity<AppUser> registerUser(@RequestBody AppUser registerUser) {
        AppUser registeredUser = appUserService.registerAppUser(registerUser);

        return ResponseEntity.ok(registeredUser);
    }
}
```

Podana zmiana została zatwierdzona commit'em:
https://bitbucket.org/saintamen/spring-hello/commits/dd34a0992dc8e258f4326d8355326c0b373a0639

Omówienie komponentów – na przykładzie.

Podane wcześniej rozwiązanie nie jest w pełni bezpieczne. Podając użytkownikowi model App User dzielimy się informacją jak wygląda struktura naszej bazy danych. Im więcej tego typu informacji wydostanie się poza projekt, tym bardziej jest on narażony na ataki. Oczywiście – możemy manipulować modelami z bazy danych, jednak popularne jest stosowanie modeli DTO – czyli modeli Data Transfer Object.

Po prawej – wysłanie ramki spowoduje zwrócenie obiektu ze wszystkimi polami i wartościami.



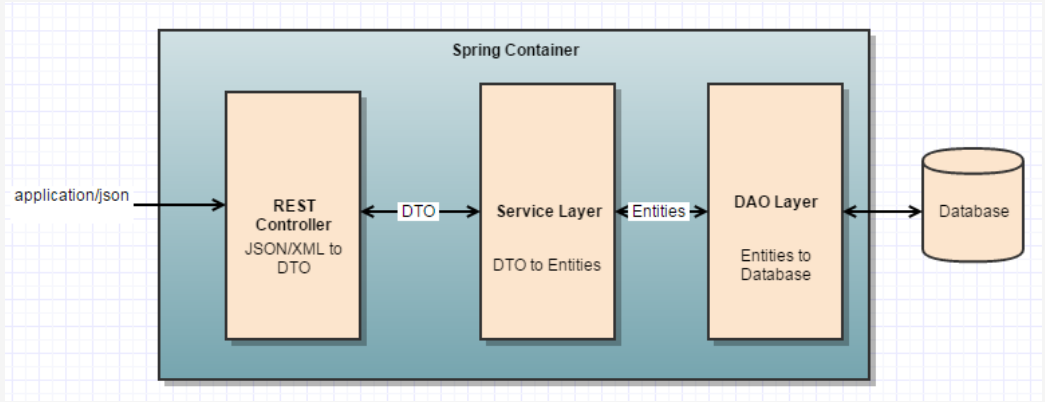
Podana zmiana została zatwierdzona commit'em:
https://bitbucket.org/saintamen/spring-hello/commits/dd34a0992dc8e258f4326d8355326c0b373a0639

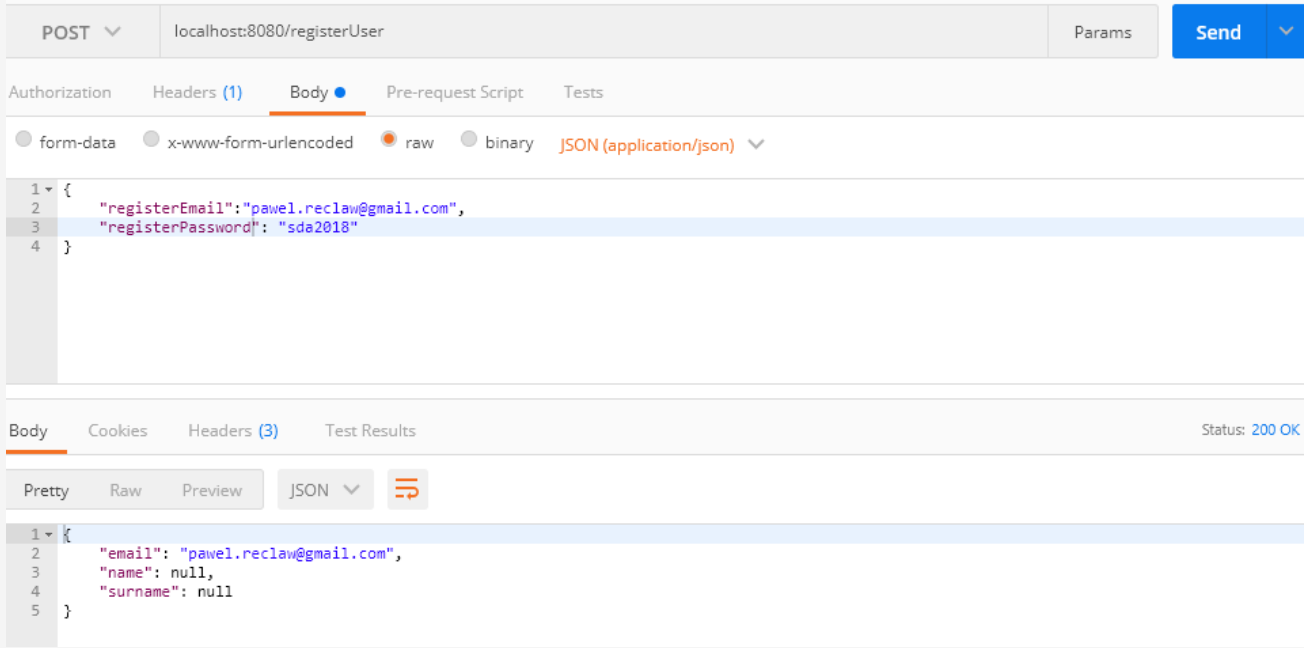
Omówienie komponentów – na przykładzie.

W nowym podejściu z DTO stworzyliśmy dwa modele. Jeden obiekt jest wysyłany przez rejestrującego, a drugi jest wysyłany w odpowiedzi. Częsć nazw pól została zmieniona aby ukryć ich nazwy.

```java
@PostMapping("/registerUser")
public ResponseEntity<AppUserDto> registerUser(@RequestBody RegisterAppUserDto registerUser) {
    AppUserDto registeredUser = appUserService.registerAppUser(registerUser);

    return ResponseEntity.ok(registeredUser);
}
```

Dto to obiekty wysyłane do Controller'a, a dalej przekazywane do serwisu gdzie są przetwarzane. Serwis operuje na bazie używając modeli bazodanowych (entity).





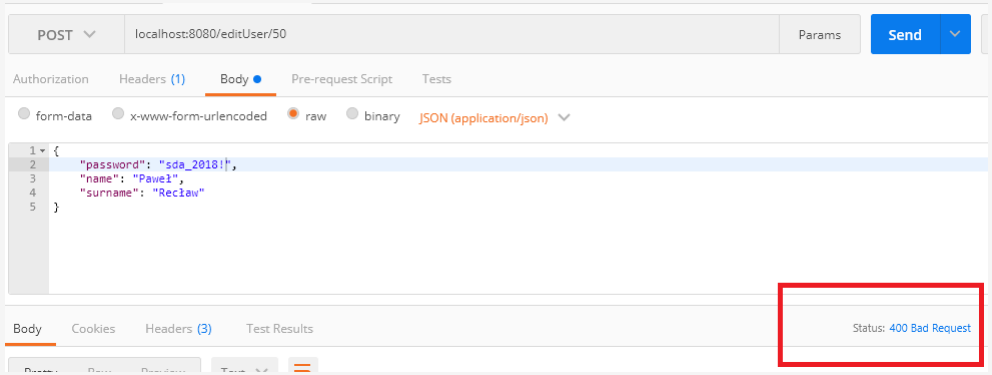
Podana zmiana została zatwierdzona commit'em:
https://bitbucket.org/saintamen/spring-hello/commits/a8ae02d4cd63eb51ce7e583b92e4de33559f8108

Omówienie komponentów – na przykładzie.

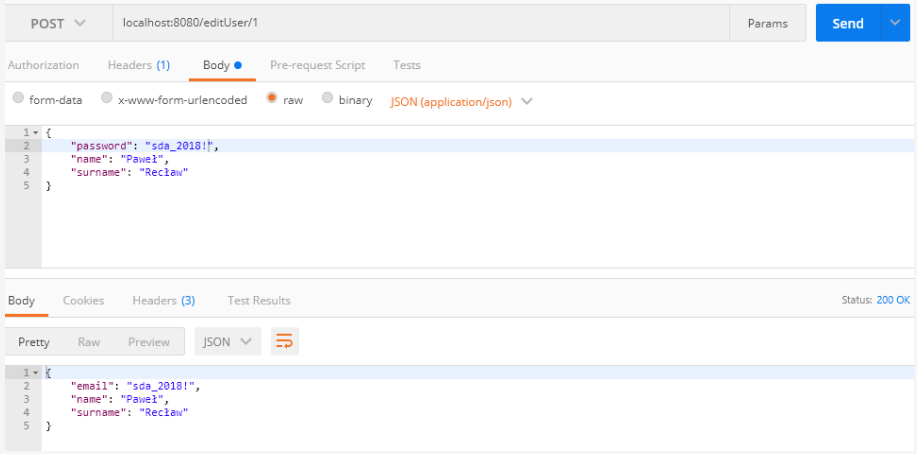
Kolejną częścią rozwoju funkcjonalności jest edycja wpisu. W podanym przykładzie edytujemy wpis podając identyfikator w adresie URL – tag {identifier} jest użyty przy adnotacji **PostMapping** oraz przy **PathVariable** i musi być podana przy wywołaniu adresu. Ciało zawierające dane edytujące podajemy w ciele metody (stąd adnotacja **RequestBody**).

Metoda edytuje użytkownika i zwraca dane edytowanego użytkownika. Jeśli spróbujemy edytować nieistniejącego użytkownika otrzymamy błąd (400).

Błąd:



OK:




```
@PostMapping("/editUser/{identifier}")
public ResponseEntity<AppUserDto> editAppUser(@PathVariable(name = "identifier") Long id,
                                              @RequestBody EditAppUserDto editAppUserDto) {
    Optional<AppUser> appUser = appUserService.editUser(id, editAppUserDto);

    if (appUser.isPresent()) {
        return ResponseEntity.ok(AppUserDto.create(appUser.get()));
    }

    return ResponseEntity.badRequest().build();
}
```

Podana zmiana została zatwierdzona commit'em:
https://bitbucket.org/saintamen/spring-hello/commits/83873cce65ef41222290020be39641757e6079cb

Programowanie aspektowe

Podczas rozwoju aplikacji zaimplementowałem funkcjonalność której będę chciał dostarczyć użytkownikowi.

Tworząc projekt powinniśmy skupiać się na funkcjonalności którą chcemy zaimplementować i powinniśmy separować zagadnienia. Jeśli chcę adresować funkcjonalność związaną z zarządzaniem użytkownikami, to będę je kierować do kontrolera użytkowników. Stworzenie nowego modelu i powiązanych z nimi kontrolera i serwisu daje nam separację tych modeli i pozwala nam rozwijać jako oddzielny fragment aplikacji.

Bardzo często budowa i funkcje kontrolerów uzależnione są od funkcjonalności którą implementujemy. Przykładowo: ponieważ potrzebuję możliwości listowania użytkowników, to stworzę do dla użytkowników kontroler a w nim odpowiednią metodę, która zapewni mi tą funkcjonalność.