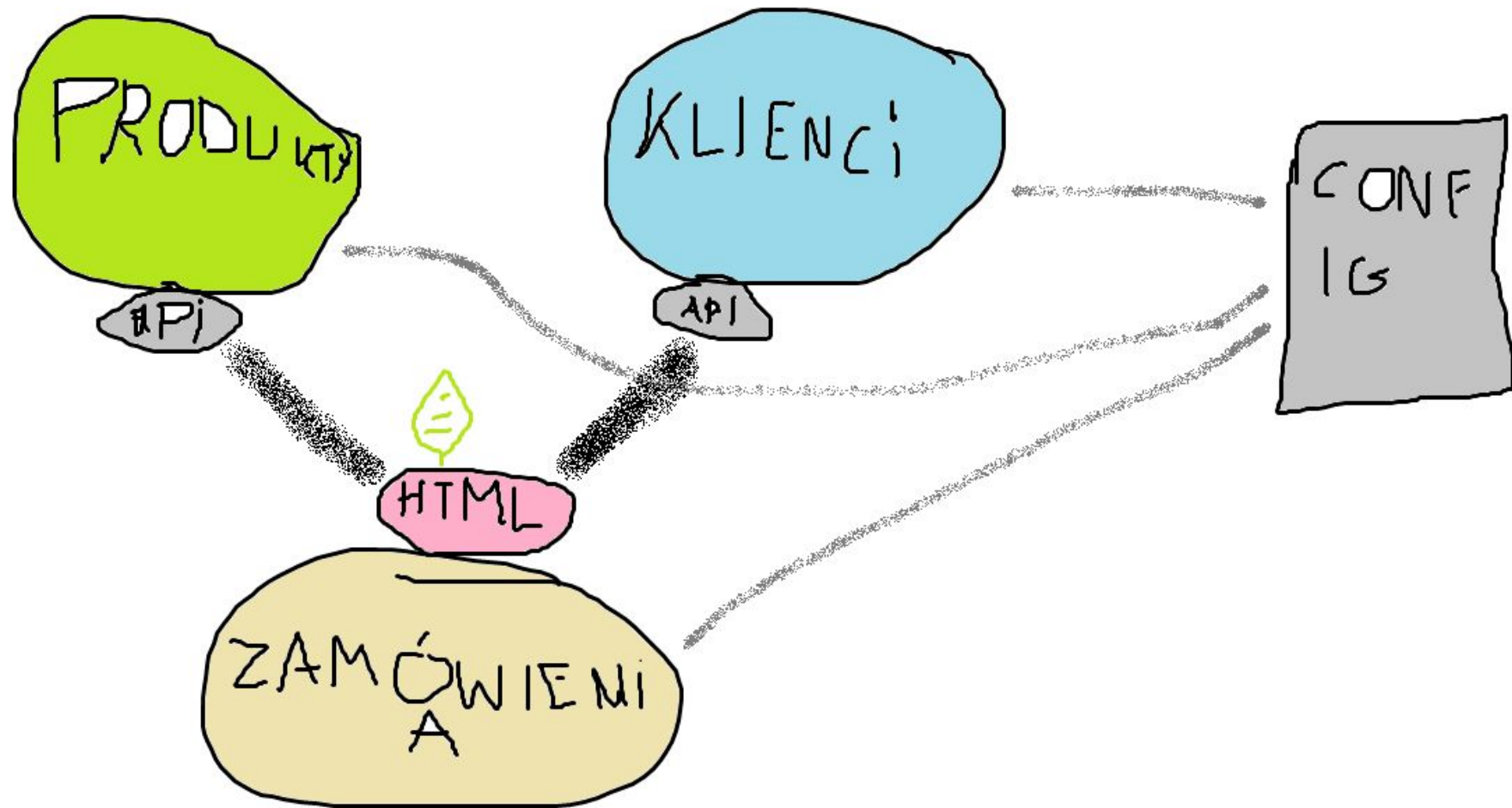


# Tworzenie aplikacji opartych o mikroserwisy w Spring Framework

---

Autor: Hubert Zabłocki



# Czym są mikroserwisy?

**Mikroserwisem nazwiemy małą aplikację, która dobrze wykonuje tylko jedno zadanie.** To niewielki komponent, który można łatwo wymienić, niezależnie rozwijać i niezależnie instalować. Mikroserwis to część większego systemu, uruchomionego i działającego razem z innymi mikroserwisami dla osiągnięcia tego, co inaczej byłoby obsługiwane przez jedną dużą, samodzielna aplikację.

Definicja na podstawie książki „Mikrouслуги” Susan J. Fowler.

# Jak utworzyć mikroserwis w Springu?

Użyjemy do tego  
**Spring Initializr** oraz **IntelliJ IDEA**  
dzięki temu w prosty sposób  
wygenerujemy projekt i  
użyjemy domyślnej konfiguracji

---

Nasz pierwszy mikroserwis  
będzie **WYŁĄCZNIE**  
prostym CRUD-em  
operującym na produktach.

# Tworzenie projektu

Na stronie [start.spring.io](https://start.spring.io)

uzupełniamy metadane,

dodajemy zależności.

W naszym przypadku będą to:

- **Spring Web**
- **Spring Data JPA**
- **H2 Database**

Następnie generujemy projekt

Project

☒ Maven Project
 ☐ Gradle Project

Language

☒ Java
 ☐ Kotlin
 ☐ Groovy

Spring Boot

☐ 2.4 (SNAPSHOT)
 ☐ 2.3.1 (SNAPSHOT)
 ☒ 2.3.0
 ☐ 2.2.8 (SNAPSHOT)
 ☐ 2.2.7
 ☐ 2.1.15 (SNAPSHOT)
 ☐ 2.1.14

Project Metadata

Group

pl.hzablocki

Artifact

air-mikroserwis

Name

air-mikroserwis

Description

Pierwszy mikroserwis

Package name

pl.hzablocki.air-mikroserwis

Packaging

☒ Jar
 ☐ War

Java

☐ 14
 ☐ 11
 ☒ 8

Dependencies

ADD ... CTRL + B

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA

SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database

SQL

Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

GENERATE CTRL + ↵

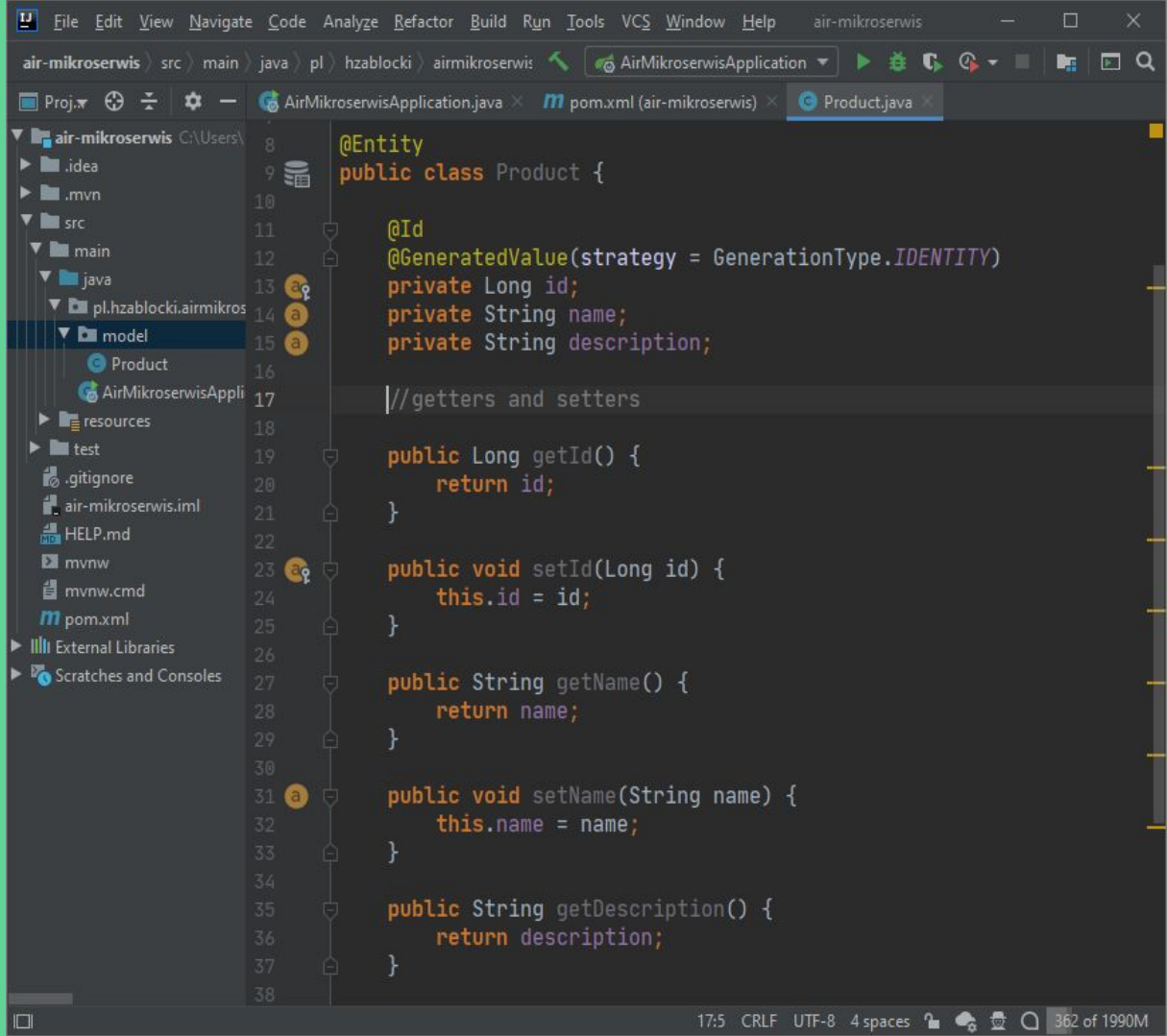
EXPLORE CTRL + SPACE

SHARE...

# Model

Teraz utworzymy reprezentację naszego modelu w postaci encji.

Dodajemy odpowiednie adnotacje ze standardu **JPA**, czyli **@Entity** **@Id** **@GeneratedValue**



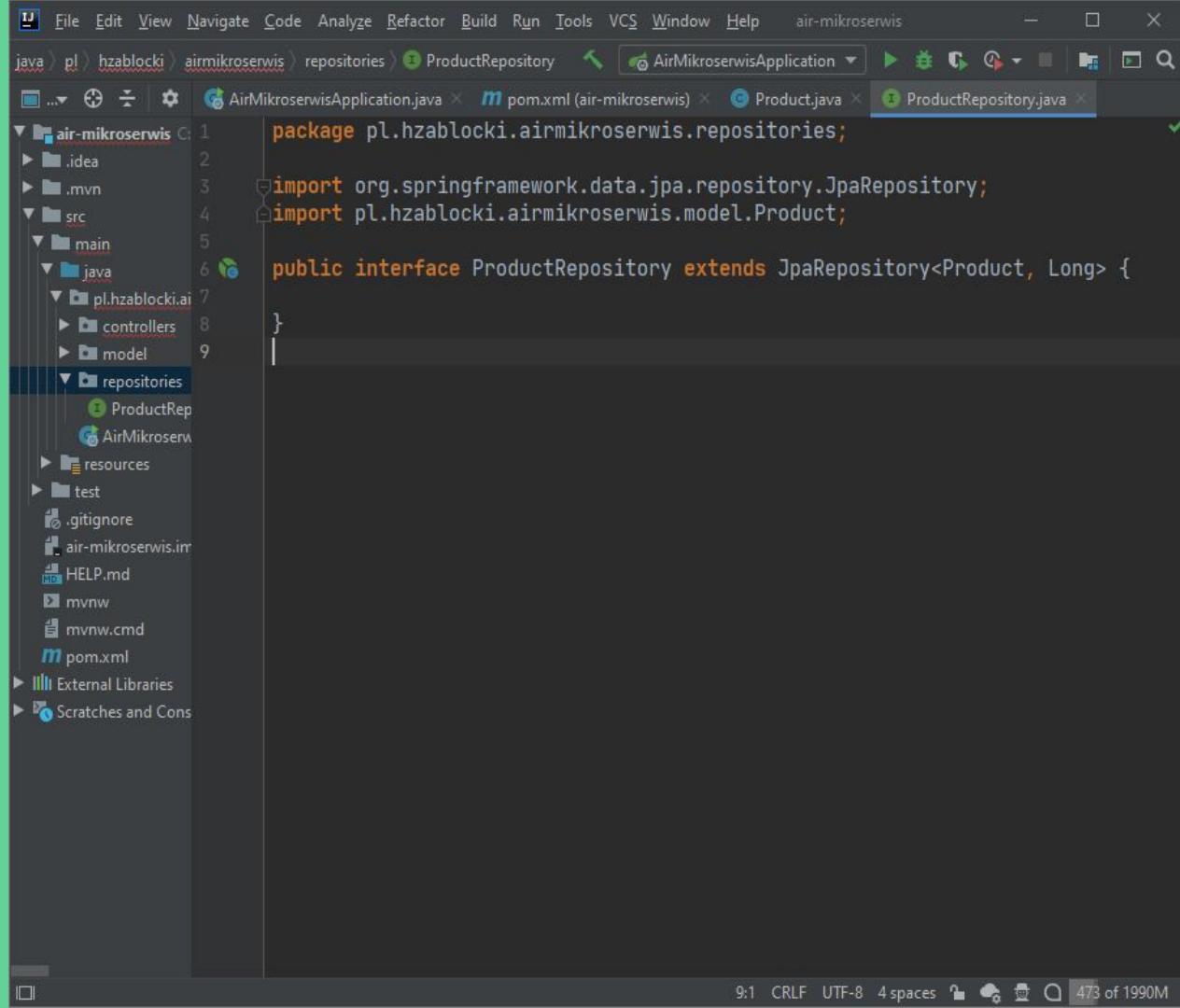
The screenshot shows an IDE with the following components:

- Project Explorer (Left):** Displays the project structure. The path `air-mikroserwis > src > main > java > pl.hzablocki.airmikros > model` is expanded, showing the `Product` class being created.
- Editor (Right):** Contains the code for `Product.java`. The code is as follows:

```
8  @Entity
9  public class Product {
10
11     @Id
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private Long id;
14     private String name;
15     private String description;
16
17     //getters and setters
18
19     public Long getId() {
20         return id;
21     }
22
23     public void setId(Long id) {
24         this.id = id;
25     }
26
27     public String getName() {
28         return name;
29     }
30
31     public void setName(String name) {
32         this.name = name;
33     }
34
35     public String getDescription() {
36         return description;
37     }
38 }
```
- Bottom Bar:** Shows the status bar with the following information: `17:5 CRLF UTF-8 4 spaces 362 of 1990M`.

# Repozytorium

Nasze repozytorium  
rozszerza **JpaRepository**  
dzięki czemu otrzymujemy  
gotowy **CRUD**.

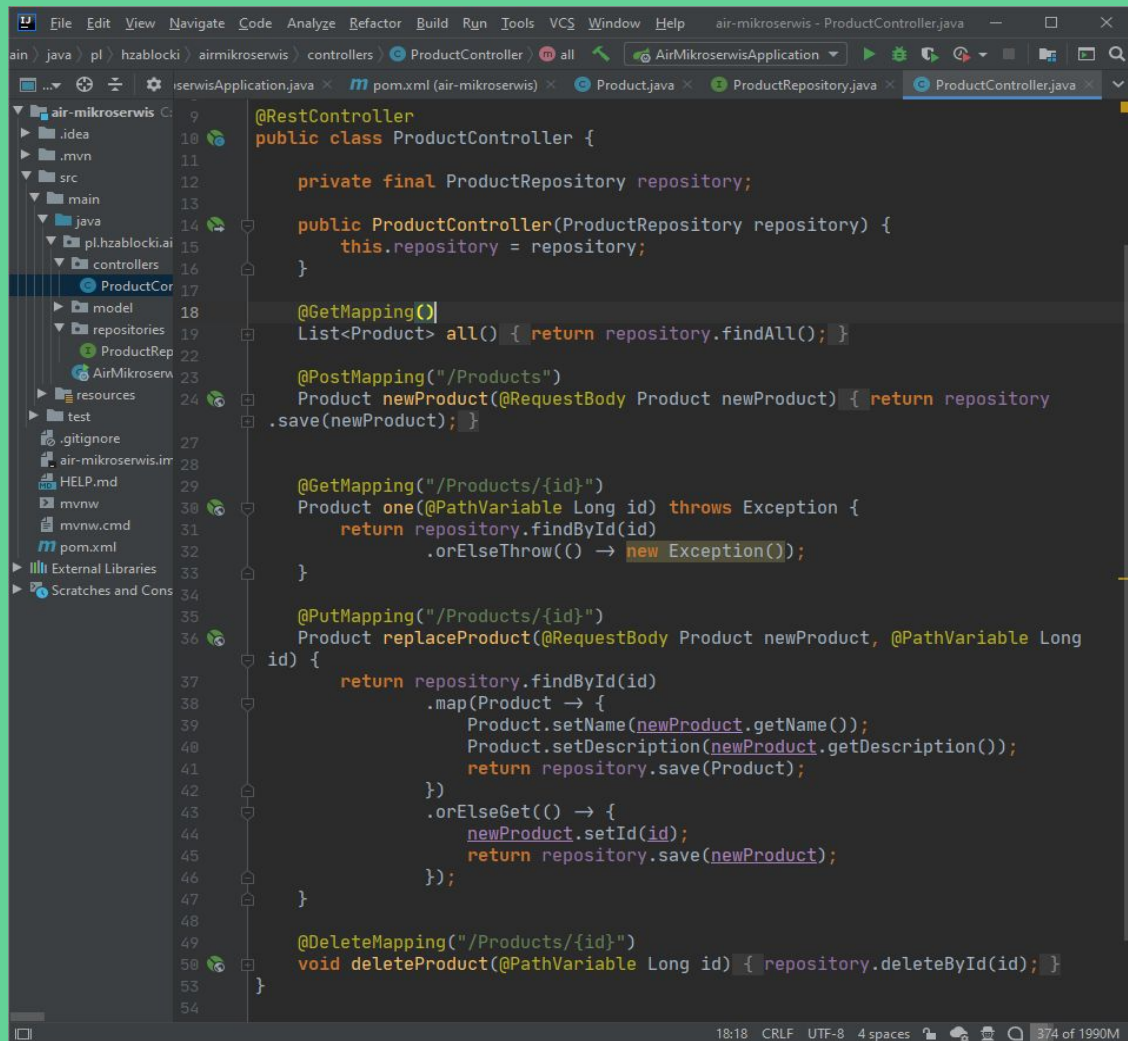




# Controller

W Spring WebServices  
żądania HTTP są  
obsługiwane przez  
kontroler.

**@RestController** oznacza  
klasę jako kontroler, w  
którym każda metoda  
zwraca obiekt domeny w  
formacie JSON zamiast  
widoku.



```
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help air-mikroserwis - ProductController.java
ain java pl hzablocki airmikroserwis controllers ProductController all AirMikroserwisApplication
iservisApplication.java pom.xml (air-mikroserwis) Product.java ProductRepository.java ProductController.java
air-mikroserwis
  .idea
  .mvn
  src
    main
      java
        pl.hzablocki.ai
          controllers
            ProductController.java
        model
          repositories
            ProductRepository.java
        resources
          AirMikroserwisApplication.java
        test
          .gitignore
          air-mikroserwis.iml
          HELP.md
          mvnw
          mvnw.cmd
          pom.xml
    External Libraries
    Scratches and Cons
9 @RestController
10 public class ProductController {
11
12     private final ProductRepository repository;
13
14     public ProductController(ProductRepository repository) {
15         this.repository = repository;
16     }
17
18     @GetMapping()
19     List<Product> all() { return repository.findAll(); }
20
21     @PostMapping("/Products")
22     Product newProduct(@RequestBody Product newProduct) { return repository
23         .save(newProduct); }
24
25
26
27
28
29
30
31     @GetMapping("/Products/{id}")
32     Product one(@PathVariable Long id) throws Exception {
33         return repository.findById(id)
34             .orElseThrow(() -> new Exception());
35     }
36
37     @PutMapping("/Products/{id}")
38     Product replaceProduct(@RequestBody Product newProduct, @PathVariable Long
39         id) {
40         return repository.findById(id)
41             .map(Product -> {
42                 Product.setName(newProduct.getName());
43                 Product.setDescription(newProduct.getDescription());
44                 return repository.save(Product);
45             })
46             .orElseGet(() -> {
47                 newProduct.setId(id);
48                 return repository.save(newProduct);
49             });
50     }
51
52     @DeleteMapping("/Products/{id}")
53     void deleteProduct(@PathVariable Long id) { repository.deleteById(id); }
54 }
```

18:18 CRLF UTF-8 4 spaces 374 of 1990M

# GOTOWE!

W prosty sposób utworzyliśmy działającą aplikację.

A teraz dodamy do niej interfejs **Swagger UI** pozwalający na wizualizację zasobów API i korzystanie z nich bez konieczności posiadania zewnętrznych aplikacji.

# Swagger UI

Aby uruchomić tą przyjemną nakładkę musimy dodać jej zależności do pliku **pom.xml**

Następnie dodajemy do naszej klasy startowej adnotacje **@EnableSwagger2**

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.9.2</version>
</dependency>

<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.9.2</version>
</dependency>
```

```
1 package pl.hzablocki.airmikroserwis;
2
3 import ...
4
5
6
7 @SpringBootApplication
8 @EnableSwagger2
9 public class AirMikroserwisApplication {
10
11     public static void main(String[] args) { SpringAppli
12         (AirMikroserwisApplication.class, args); }
13
14
15 }
```

GET	/ all
POST	/products newProduct
GET	/products/{id} one
PUT	/products/{id} replaceProduct
DELETE	/products/{id} deleteProduct

POST /products newProduct

Parameters Try it out

Name	Description
newProduct <span>required</span> (body)	newProduct Example Value   Model <pre>{   "description": "string",   "id": 0,   "name": "string" }</pre> Parameter content type application/json

Po uruchomieniu aplikacji należy przejść pod adres:

**localhost:8080/swagger-ui.html**

UI pozwala na wywołanie metod REST  
– z parametrami, ciałem, nagłówkami

200

Response body

```
{
  "id": 2,
  "name": "test",
  "description": "test"
}
```

Download

Response headers

```
connection: keep-alive
content-type: application/json
date: Mon, 01 Jun 2020 13:14:56 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```

Pora na drugi mikroservis.  
Jego odpowiedzialność to  
przechowywanie informacji o  
klientach i ich zamówionych  
produktach.

# Tworzenie projektu

W analogiczny sposób generujemy drugą aplikację. Dodatkowo dołączymy zależności **Lombok** i **Rest Repositories** o których wkrótce dowiemy się więcej.

Project

☒ Maven Project  
☐ Gradle Project

Language

☒ Java ☐ Kotlin  
☐ Groovy

Spring Boot

☐ 2.3.1 (SNAPSHOT) ☒ 2.3.0  
☐ 2.2.8 (SNAPSHOT) ☐ 2.2.7  
☐ 2.1.15 (SNAPSHOT) ☐ 2.1.14

Project Metadata

Group

pl.zablocki

Artifact

customer-webservice

Name

customer-webservice

Description

Demo project for Spring Boot

Package name

pl.zablocki.customer-webservice

Packaging

☒ Jar ☐ War

Java

☐ 14 ☐ 11 ☒ 8

Dependencies

ADD ... CTRL + B

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Rest Repositories

WEB

Exposing Spring Data repositories over REST via Spring Data REST.

H2 Database

SQL

Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Spring Data JPA

SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Lombok

DEVELOPER TOOLS

Java annotation library which helps to reduce boilerplate code.

GENERATE CTRL + ⌘

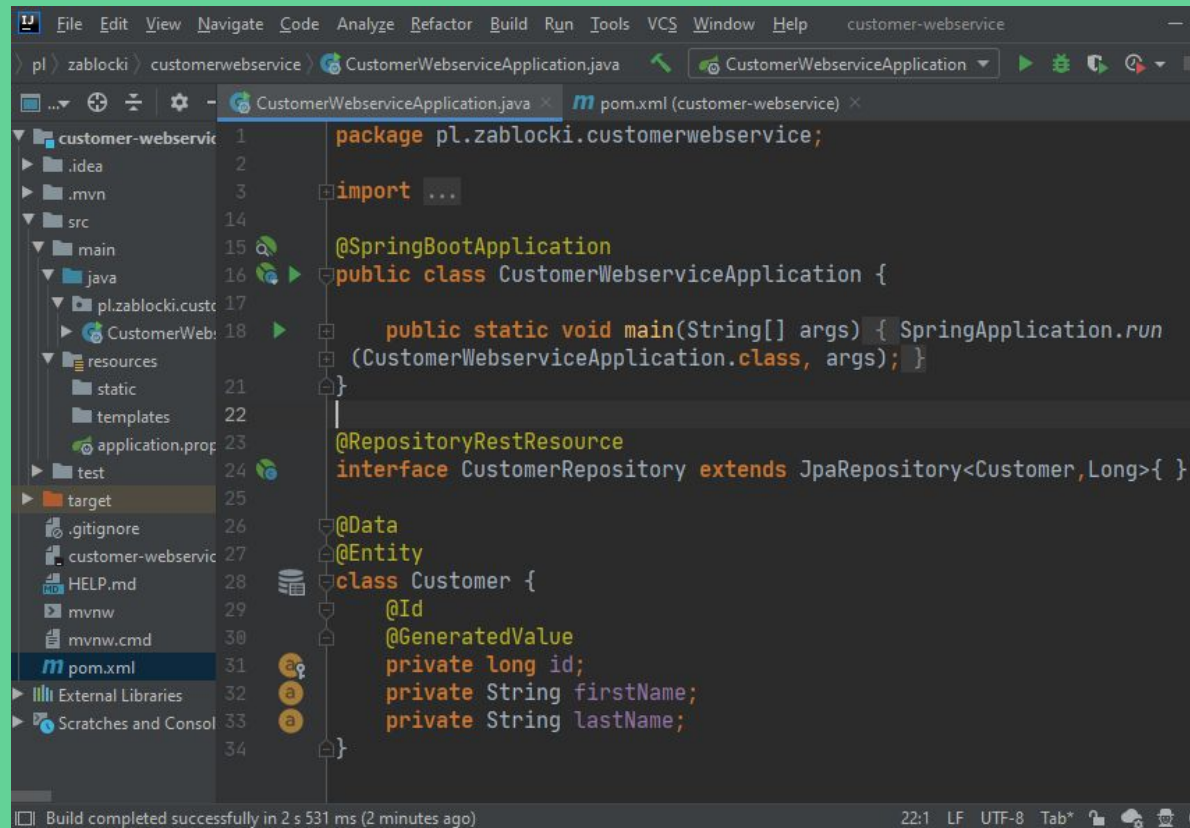
EXPLORE CTRL + SPACE

SHARE...

Utworzyliśmy model klienta z adnotacją **@Data**, dzięki której **Lombok** podczas kompilacji wygeneruje cały boilerplate kod wewnątrz naszego POJO

**Spring Data REST** utworzy za nas eksplorowalny interfejs API. Wystarczy tylko adnotacja

**@RepositoryRestResource**



```
1 package pl.zablocki.customerwebservice;
2
3 import ...
4
5 @SpringBootApplication
6 public class CustomerWebserviceApplication {
7
8     public static void main(String[] args) { SpringApplication.run
9         (CustomerWebserviceApplication.class, args); }
10
11     @RepositoryRestResource
12     interface CustomerRepository extends JpaRepository<Customer, Long>{ }
13
14     @Data
15     @Entity
16     class Customer {
17         @Id
18         @GeneratedValue
19         private long id;
20         private String firstName;
21         private String lastName;
22     }
23 }
```

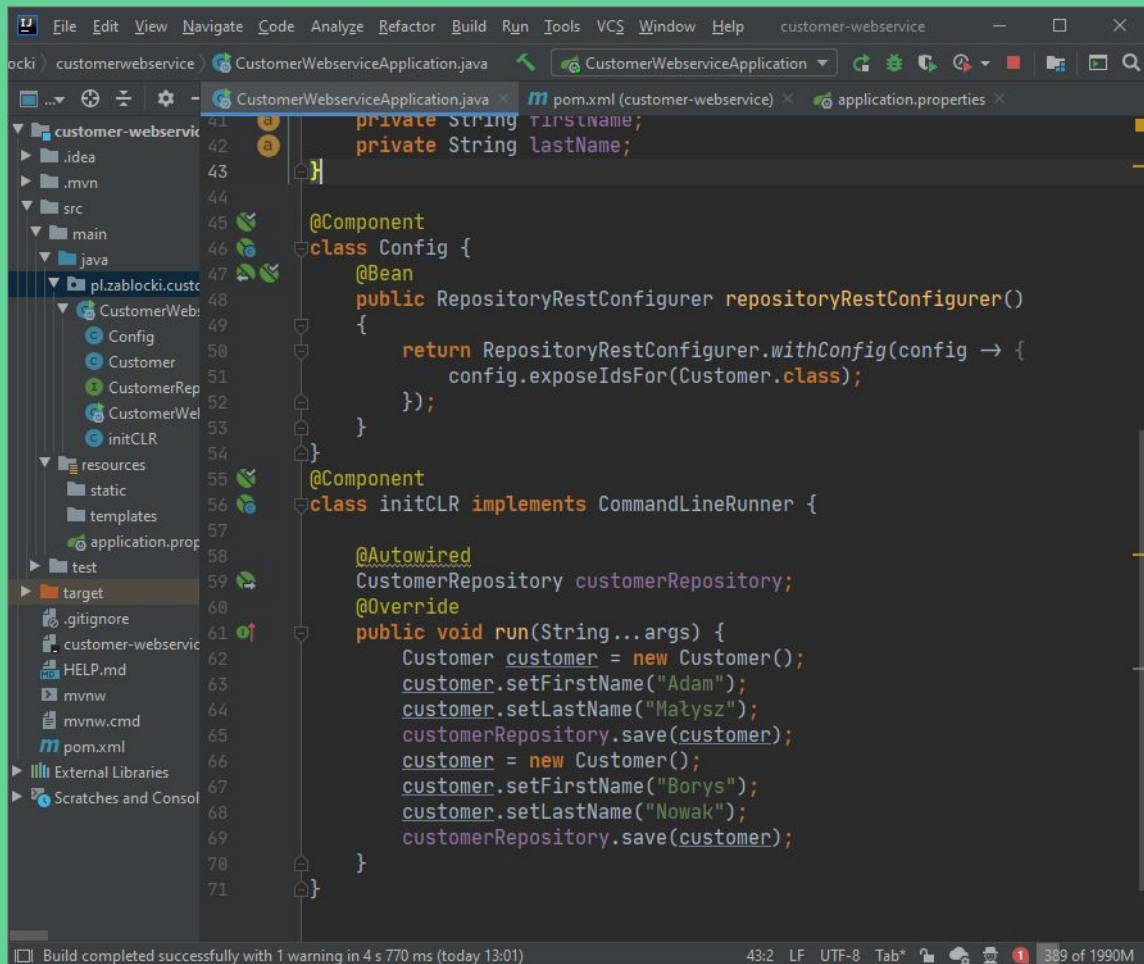
*Wszystko umieściłem w jednym pliku aby pokazać jak mało kodu potrzeba*



Zmodyfikujemy jeszcze 2 rzeczy.

Domyślnie API z **Spring Data REST** nie udostępnia ID encji. W naszym przypadku będzie ono potrzebne. Dlatego dostarczymy swój obiekt konfiguracji, żeby ID było zwracane wraz z resztą pól.

Dodatkowo przy starcie aplikacji do repozytorium dodamy w testowych Klientów.

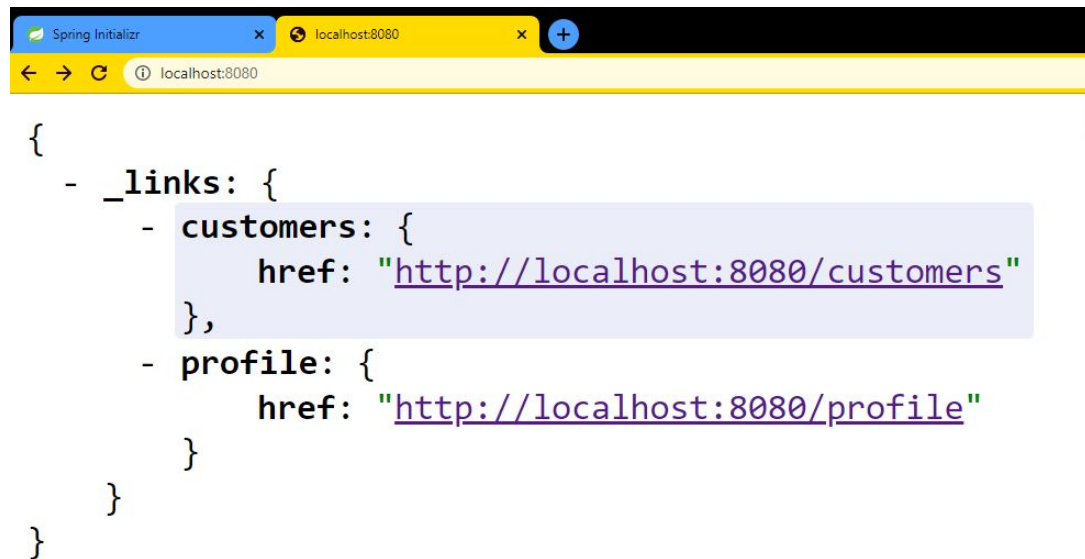


```
41 private String firstName;  
42 private String lastName;  
43  
44  
45 @Component  
46 class Config {  
47     @Bean  
48     public RepositoryRestConfigurer repositoryRestConfigurer()  
49     {  
50         return RepositoryRestConfigurer.withConfig(config -> {  
51             config.exposeIdsFor(Customer.class);  
52         });  
53     }  
54  
55 @Component  
56 class initCLR implements CommandLineRunner {  
57  
58     @Autowired  
59     CustomerRepository customerRepository;  
60  
61     @Override  
62     public void run(String... args) {  
63         Customer customer = new Customer();  
64         customer.setFirstName("Adam");  
65         customer.setLastName("Małysz");  
66         customerRepository.save(customer);  
67         customer = new Customer();  
68         customer.setFirstName("Borys");  
69         customer.setLastName("Nowak");  
70         customerRepository.save(customer);  
71     }  
72 }
```



Nasza aplikacja posiada już  
“przeklikiwalne” API.  
Domyślnie znajduje się ono  
pod adresem localhost:8080

Dodajmy do projektu **HAL  
Browser**, czyli przeglądarkę  
naszego API. Wystarczy  
tylko wprowadzić nową  
zależność do projektu



```
{
  - _links: {
    - customers: {
      href: "http://localhost:8080/customers"
    },
    - profile: {
      href: "http://localhost:8080/profile"
    }
  }
}
```

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-rest-hal-browser</artifactId>
</dependency>
```

Po ponownym skompilowaniu i uruchomieniu projektu zostaliśmy przekierowani na główną stronę przeglądarki **HAL**, z poziomu której możemy testować nasze wygenerowane API.

The screenshot displays the 'The HAL Browser (for Spring Data REST)' application running in a web browser at localhost:8080. The interface is divided into several sections:

- Explorer:** Contains a path input field with '/' and a 'Go!' button.
- Custom Request Headers:** An empty text area for defining request headers.
- Properties:** A text area showing an empty JSON object: 

```
{ }
```
- Links:** A table listing available API endpoints. The 'customers' link is highlighted with a green arrow.
- Inspector:** Displays the response details for the selected 'customers' endpoint.
  - Response Headers:** Shows a 200 success status and various headers including 'connection: keep-alive', 'content-type: application/hal+json', 'date', 'keep-alive: timeout=60', 'transfer-encoding: chunked', and 'vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers'.
  - Response Body:** Shows the JSON response: 

```
{  "_links": {    "customers": {      "href": "http://localhost:8080/customers"    }  },  }
```
  - Form Fields:** Below the response body, there are input fields for 'First name' and 'Last name', and a section for 'Action:' with a dropdown set to 'POST' and a text field containing the URL 'http://localhost:8080/customers'.

Następnym mikrosерwisem  
będzie aplikacja łącząca  
klientów i produkty w  
zamówienia.

# Tworzenie projektu

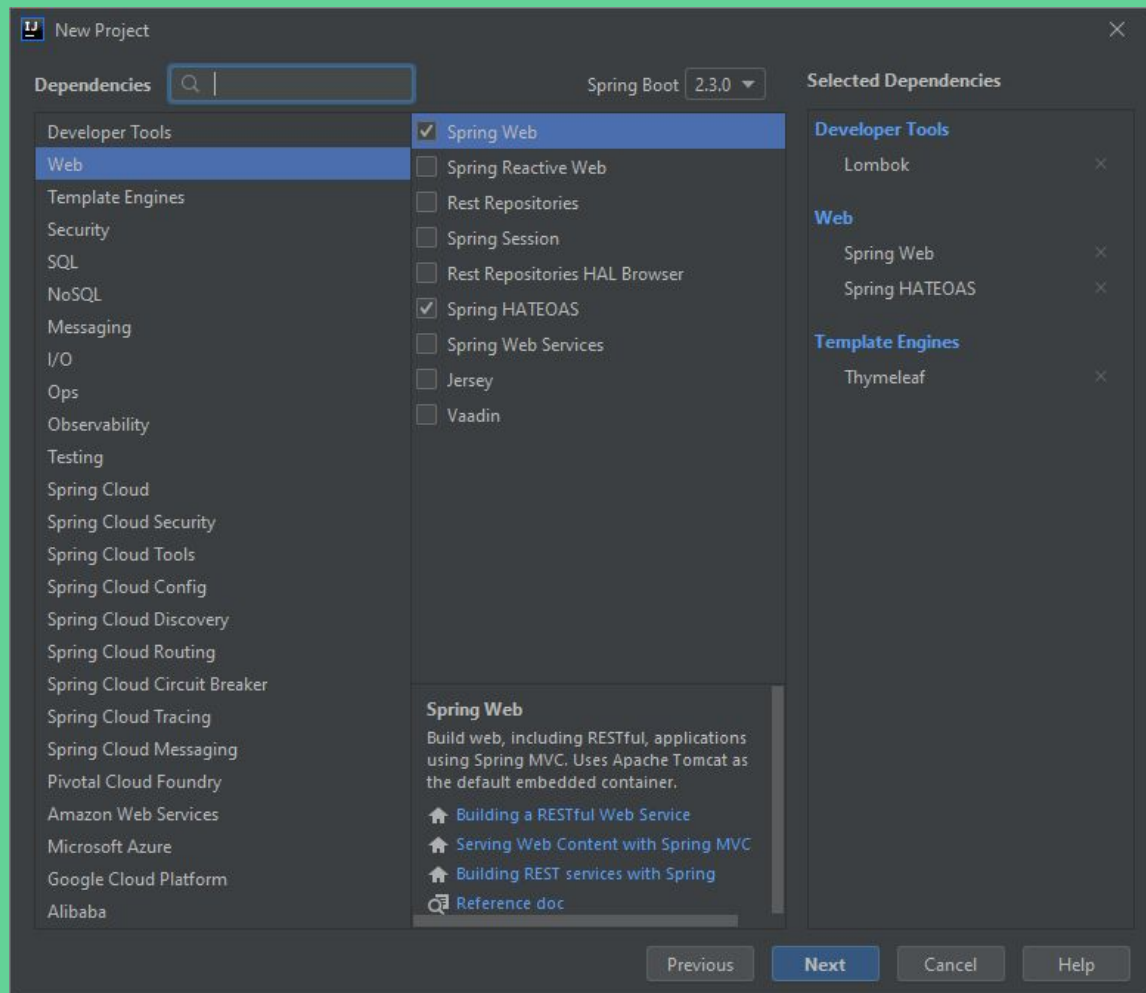
Tym razem wygenerujemy projekt przez IntelliJ IDEA.

Do zależności dołączymy dodatkowo

**Thymeleaf** - Silnik szablonów

**Spring HATEOAS** -

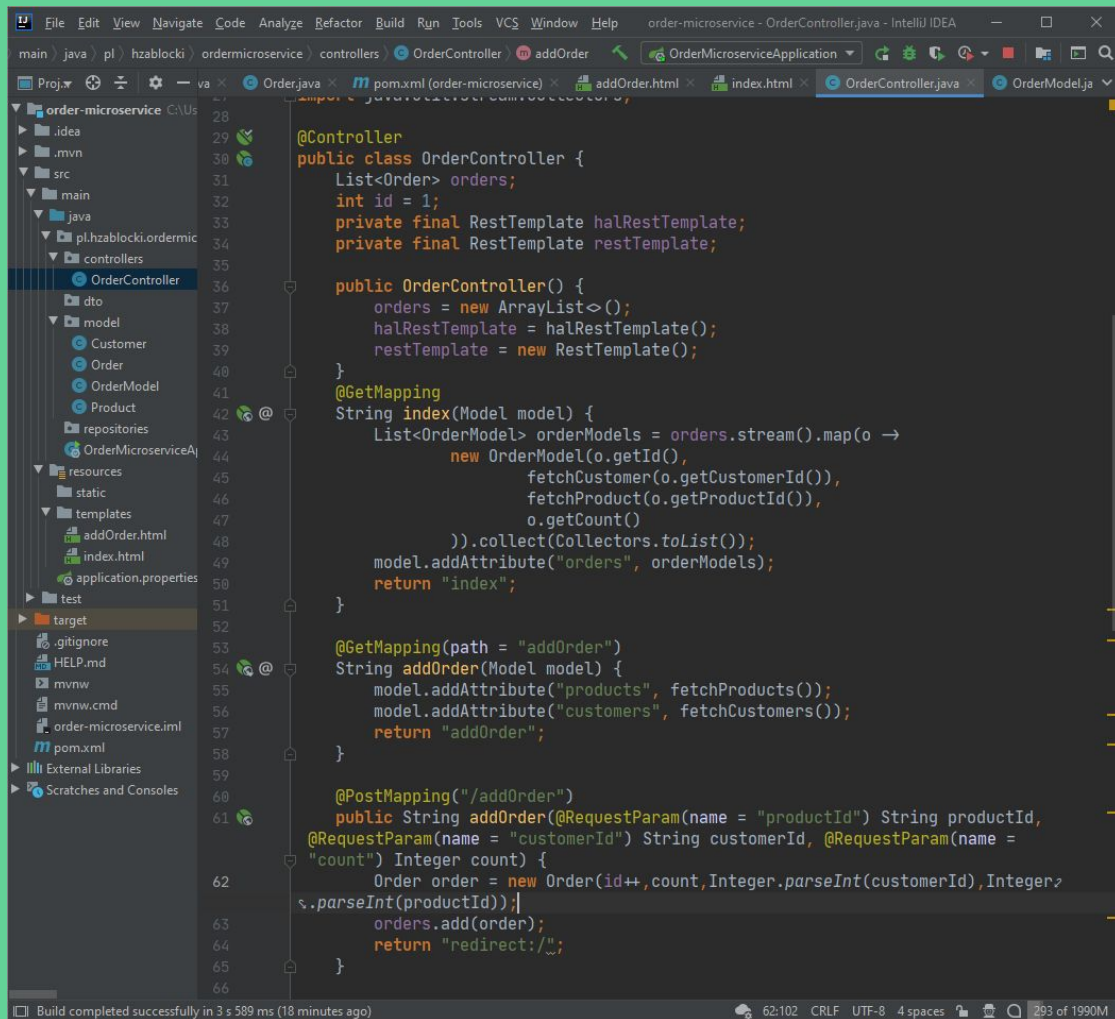
obsługuję odpowiedzi w postaci **hal+json**



# Controller

Tworzymy proste API pozwalające na dodawanie i wyświetlanie aktualnych zamówień oraz generuje widoki.

Nasza aplikacja będzie przechowywała zamówienia w postaci listy. Informacje o dostępnych produktach i klientach pozyska z naszych mikroservisów



```
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help order-microservice - OrderController.java - IntelliJ IDEA
main > java > pl > hzablocki > ordermicroservice > controllers > OrderController > addOrder OrderMicroserviceApplication
Proj: order-microservice C:\Us...
  .idea
  .mvn
  src
    main
      java
        pl.hzablocki.ordermic...
          controllers
            OrderController
            dto
            model
              Customer
              Order
              OrderModel
              Product
            repositories
              OrderMicroserviceA...
            resources
            static
            templates
              addOrder.html
              index.html
            application.properties
            test
            target
              .gitignore
              HELP.md
              mvnw
              mvnw.cmd
              order-microservice.iml
              pom.xml
            External Libraries
            Scratches and Consoles

28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

@Controller
public class OrderController {
    List<Order> orders;
    int id = 1;
    private final RestTemplate halRestTemplate;
    private final RestTemplate restTemplate;

    public OrderController() {
        orders = new ArrayList<>();
        halRestTemplate = halRestTemplate();
        restTemplate = new RestTemplate();
    }

    @GetMapping
    String index(Model model) {
        List<OrderModel> orderModels = orders.stream().map(o ->
            new OrderModel(o.getId(),
                fetchCustomer(o.getCustomerId()),
                fetchProduct(o.getProductId()),
                o.getCount())
        ).collect(Collectors.toList());
        model.addAttribute("orders", orderModels);
        return "index";
    }

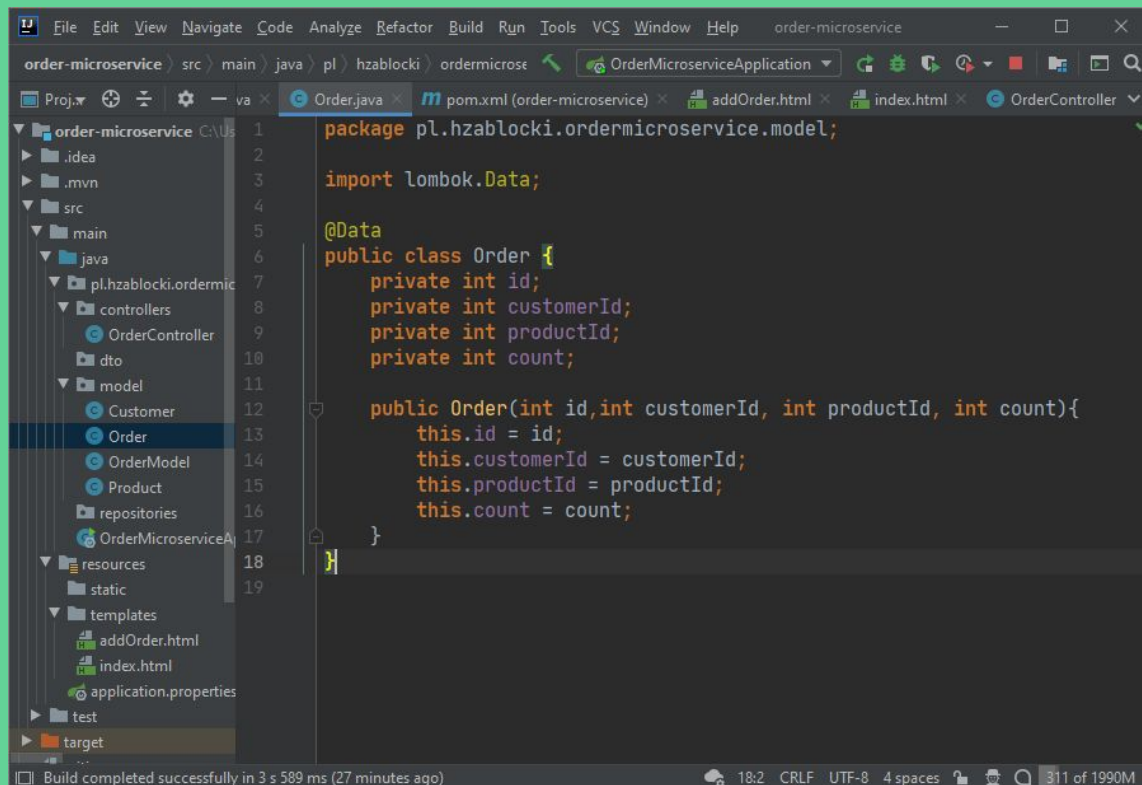
    @GetMapping(path = "addOrder")
    String addOrder(Model model) {
        model.addAttribute("products", fetchProducts());
        model.addAttribute("customers", fetchCustomers());
        return "addOrder";
    }

    @PostMapping("/addOrder")
    public String addOrder(@RequestParam(name = "productId") String productId,
        @RequestParam(name = "customerId") String customerId, @RequestParam(name =
        "count") Integer count) {
        Order order = new Order(id++, count, Integer.parseInt(customerId), Integer.
        parseInt(productId));
        orders.add(order);
        return "redirect:/";
    }
}
```

Build completed successfully in 3 s 589 ms (18 minutes ago) 62:102 CRLF UTF-8 4 spaces 293 of 1990M

# Model

W naszym modelu przechowamy tylko ID powiązanego klienta i produktu.



```
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help order-microservice
order-microservice src main java pl hzablocki ordermicrose OrderMicroserviceApplication
Proj. x va x Order.java pom.xml (order-microservice) addOrder.html index.html OrderController
order-microservice
  .idea
  .mvn
  src
    main
      java
        pl.hzablocki.ordermicroservice
          controllers
            OrderController
          dto
          model
            Customer
            Order
            OrderModel
            Product
          repositories
            OrderMicroserviceApplication
          resources
            static
            templates
              addOrder.html
              index.html
            application.properties
          test
          target
```

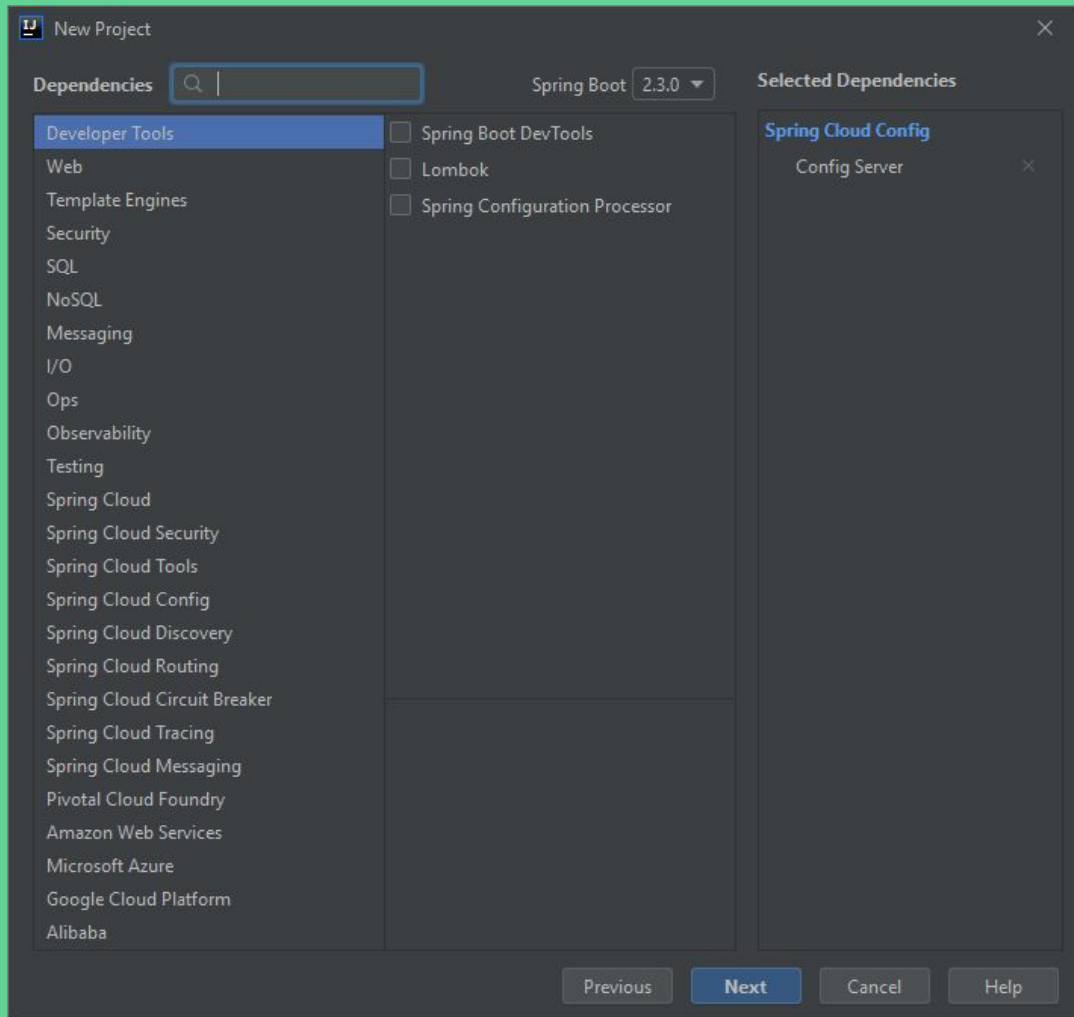
```
1 package pl.hzablocki.ordermicroservice.model;
2
3 import lombok.Data;
4
5 @Data
6 public class Order {
7     private int id;
8     private int customerId;
9     private int productId;
10    private int count;
11
12    public Order(int id,int customerId, int productId, int count){
13        this.id = id;
14        this.customerId = customerId;
15        this.productId = productId;
16        this.count = count;
17    }
18
19 }
```

Build completed successfully in 3 s 589 ms (27 minutes ago) 18:2 CRLF UTF-8 4 spaces 311 of 1990M

Na koniec za pomocą  
**Spring Cloud** utworzymy  
serwer konfiguracyjny,  
który będzie kolejnym  
mikroserwisem

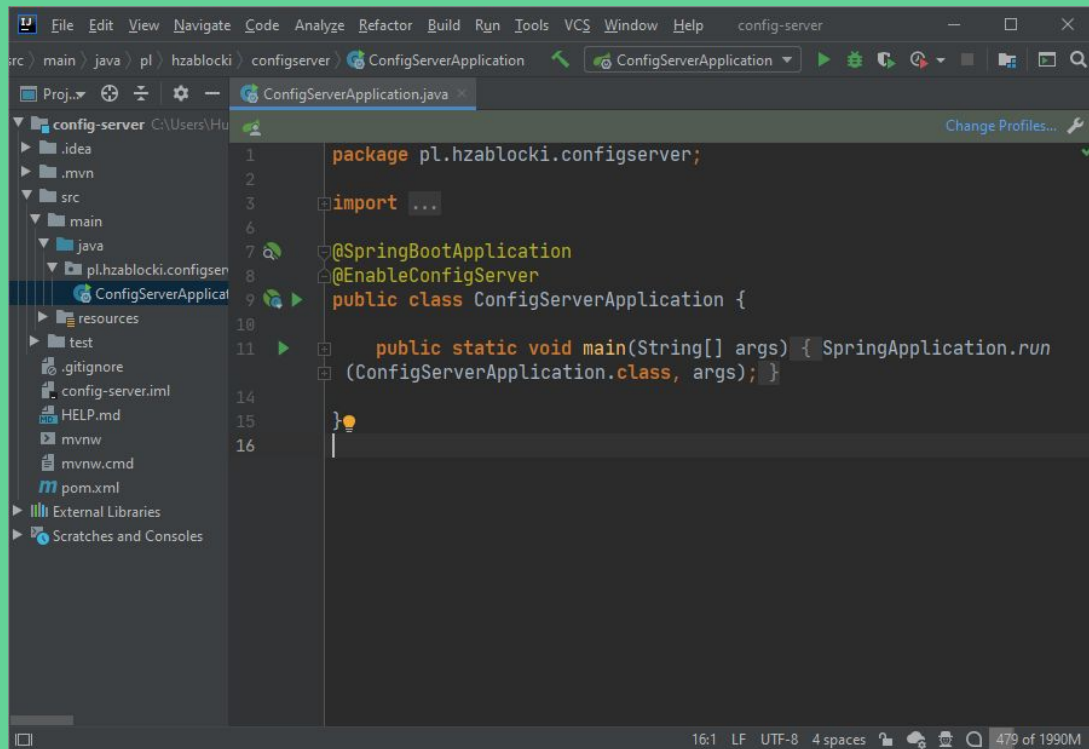
# Tworzenie projektu

Jedyną zależnością jakiej potrzebujemy jest **Config Server**





Klasę startową oznaczamy  
adnotacją  
**@EnableConfigServer**



```
1 package pl.hzablocki.configserver;
2
3 import ...
4
5 @SpringBootApplication
6 @EnableConfigServer
7 public class ConfigServerApplication {
8
9     public static void main(String[] args) { SpringApplication.run
10         (ConfigServerApplication.class, args); }
11
12 }
13
14
15
16
```

Nasz serwer konfiguracyjny potrzebuje repozytorium GIT w którym znajdą się wszystkie pliki konfiguracyjne

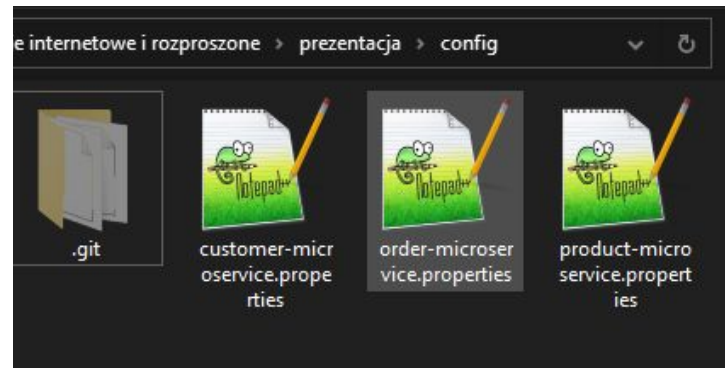
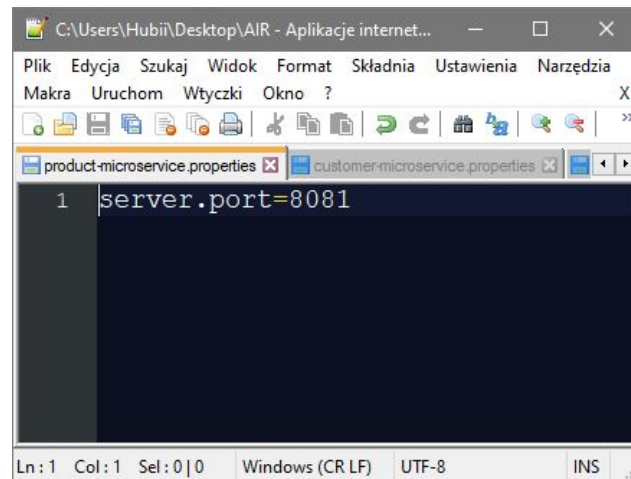
Zatem tworzymy pusty folder, następnie dla każdego z mikroserwisów tworzymy plik **{nazwa}.properties** i tam umieścimy jego konfiguracje

Na koniec tworzymy repozytorium i commitujemy:

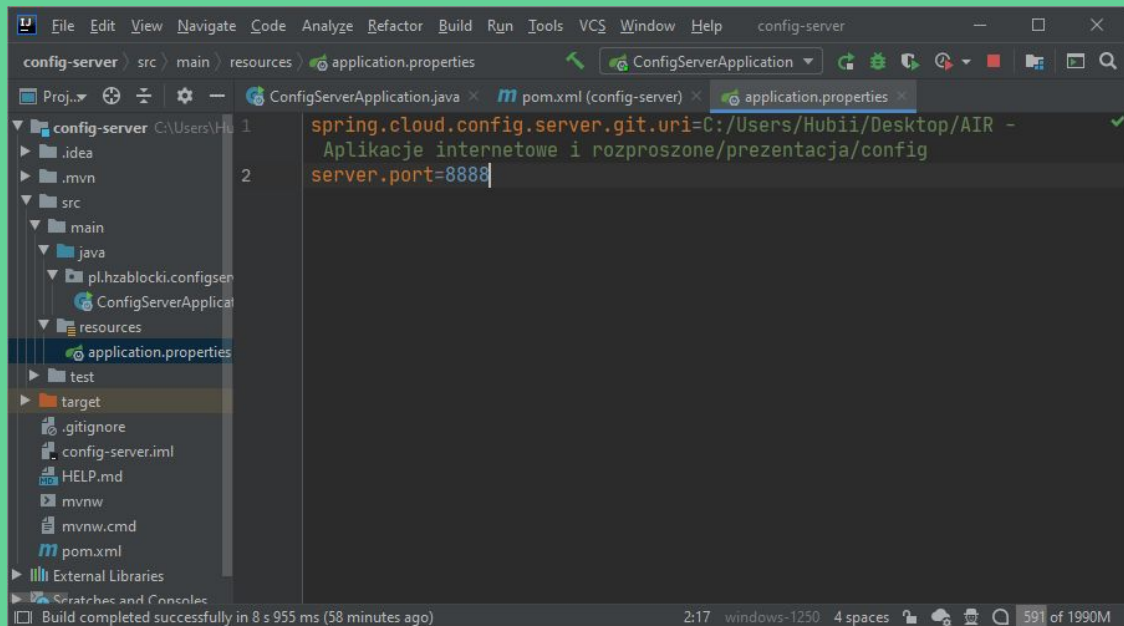
```
$ git init
```

```
$ git add *
```

```
$ git commit -m "kremowki"
```



W pliku  
**application.properties**  
servera ustawiamy jego  
port oraz ścieżkę do  
naszego git repo.



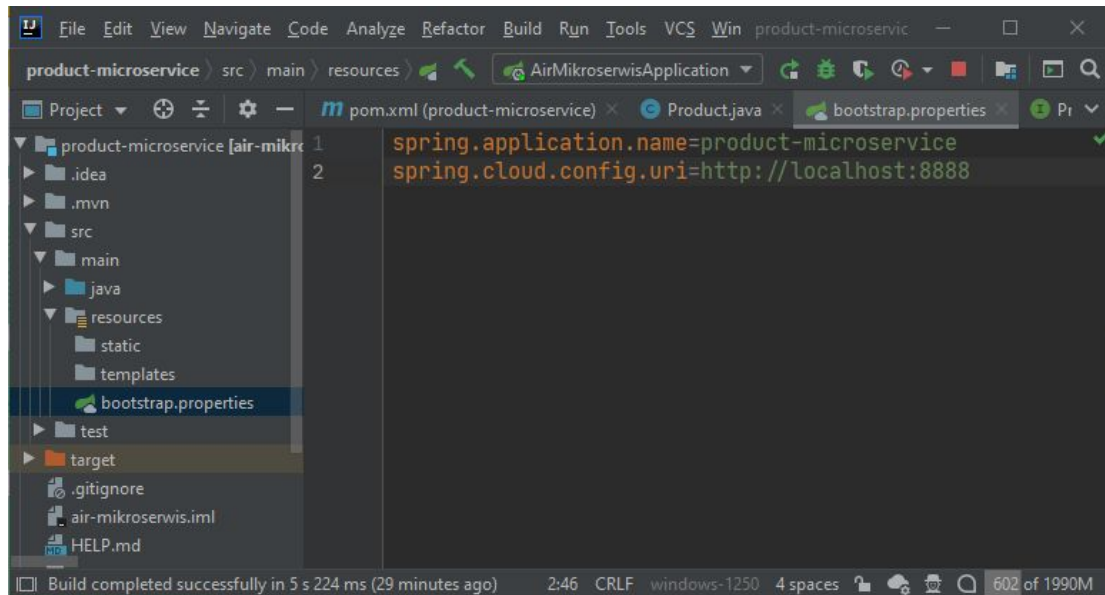
Aby móc korzystać z konfiguracji po stronie serwera, do każdego z mikroservisów dodajemy zależność **Config Client**

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

```
<properties>
  <java.version>1.8</java.version>
  <spring-cloud.version>Hoxton.SR5</spring-cloud.version>
</properties>
```

Na koniec zamieniamy nazwę pliku **application.properties** na **bootstrap.properties** i dodajemy w nim nazwę naszej aplikacji (musi się zgadzać z nazwą pliku w repo) oraz adres **Config Servera**



Swagger UI

localhost:8081/swagger-ui.html#/product-controller

### basic-error-controller

Basic Error Controller

### product-controller

Product Controller

**product-microservice**

- GET / all
- POST /product
- GET /products/{id} one
- PUT /products/{id} replaceProduct
- DELETE /products/{id} deleteProduct

Zamówienia

ID zamówienia	ID klienta	ID produktu	Ilość
1	Borys Nowak	iphone	131

[Utwórz zamówienie](#)

**order-microservice**

The HAL Browser (for Spring Data)

localhost:8082/browser/index.html#/

### Properties

```
{}
```

### Links

rel	title	name / index	docs	GET	NON-GET
customers				➔	!
profile				➔	!

```
{
  "_links": {
    "customers": {
      "href": "http://localhost:8082/customers"
    },
    "profile": {
      "href": "http://localhost:8082/profile"
    }
  }
}
```

**customer-microservice**

localhost:8888/product-microservice

localhost:8888/product-microservice/default

```
{
  name: "product-microservice",
  profiles: [
    "default"
  ],
  label: null,
  version: "27a776160dfb038d5dc07f3b816488ffc1d4d318",
  state: null,
  propertySources: [
    {
      name: "C:/Users/Hu...
      microservice.prope...
      source: {
        server.port: "8081"
      }
    }
  ]
}
```

**config-server**