Kolos 1

14 November 2022 14:28

Basic component types managed by the Spring container:

- `@Component` - most basic, generic component managed by the container,
- `@Controller` - specialized component to be used as controller in MVC framework,
- `@Repository` - DAO (data access object) in persistence layer,
- `@Service` - specialized component responsible for business logic.

By default all of those are realized as singletons (only one global instance in the container).

There are three methods for dependency injection:

- using constructor arguments,
- using setters,
- directly into class field.

```
public class UserService {

    private UserRepository repository;

    public UserService(UserRepository repository) {
        this.repository = repository;
    }

}
```

```
UserRepository repository = //create notifier
UserService service = new UserService(repository);
```

Injection with constructor:

- immutable objects (no setters and final fields) can be created,
- construction and injection in single step.

```
public class UserService {

    private UserRepository repository;

    public void setRepository(UserRepository repository) {
        this.repository = repository;
    }

}
```

```
UserService service = new UserService();
UserRepository repository = //create notifier
service.setRepository(repository);
```

Injection with setters:

- the default constructor is present,
- dependency can be replaced after injection,
- setters is required (no immutable objects with final fields).

```java
public class UserService {

    private UserRepository repository;

}
```

```java
UserService service = new UserService();
Field field = service.getClass().getDeclaredField("repository");
field.setAccessible(true);
UserRepository repository = //create notifier
field.set(service, repository);
```

Field injection:

- no additional methods,
- requires reflection mechanism.

Dependency injection in Spring: all 3, @Autowired

@PreDestroy i @PostConstruct do robienia logiki po utworzeniu i przed zniszczeniem componentu.

Spring Context

- komponenty to różne klasy o różnym przeznaczeniu, opatrzone adnotacjami
- @Component → podstawowy komponent, informacja że obiektami tej klasy będzie zarządzał kontener
- @Controller → wyspecjalizowany komponent w obsłudze żądań HTTP
- @Reposiroty → komponent, który odpowiada dostęp do danych, służą do pobierania, zapisywania lub edytowania jakichś danych
- @Service → odpowiada za logikę biznesową
- aplikacja webowa obsługująca żądania HTTP
    + żądanie idzie do kontrolera, który je tłumaczy
    + kontroler przekazuje do serwisu
    + serwis wykonuje logikę biznesową
    + jeśli logika biznesowa wymaga zmian w danych, to wywołuje repozytorium
- domyślnie każdy komponent będzie singletonem (jedna instancja w kontenerze)

Wstrzykiwanie zależności

- czasami jeden komponent będzie potrzebował drugiego komponentu
- wstrzykiwanie zależności to wzorzec projektowy, w którym klasa nie tworzy sama swojej zależności, tylko dostaje w prezencie
- wstrzykiwanie przez konstruktor → serwis potrzebuje repozytorium, więc można w konstruktorze jako argument podać mu gotowe, stworzone repozytorium, tutaj zaletą jest brak setterów, można robić obiekty immutable
- wstrzykiwanie przez settery → tworzymy posty serwis i potem dostarczamy przez setter gotową implementacje repozytorium, tutaj nie można robić obiektów immutable, ale można podmieniać implementacje w trakcie
- wstrzykiwane bezpośrednio do pól → nie ma konstruktora i settera, bierzemy klasę i ustawiamy na chwilę dostęp do danego pola, tutaj kod jest czysty (bez metod)
- na poziomie springa, kontener sam robi wstrzykiwanie
- deklarujemy punkt wstrzyknięcia adnotacją @Autowired przed konstruktorem, setterem czy danym polem → spring obsługuje wszystkie mechanizmy wstrzykiwania

Tworzenie i usuwanie obiektów

- w kodzie nie ma zdefiniowanego miejsca tworzenia i usuwanie obiektu, jest tylko deklaracja komponentu
- czasami potrzebna jest jakaś akcja po utworzeniu obiektu lub przed jego usunięciem
- do tego służą adnotacje @PostContruct i @PreDestroy
- konstruktor powinien służyć tylko stworzeniu obiektu, jeśli jest jakaś logika, to nie jest on miejscem na jej wykonywanie
- @PostContruct jest wywoływany po konstruktorze i ewentualnym wstrzykiwaniu
- datastore → component, który trzyma wszystkich użytkowników i profesje w pamięci
- character/entity i user/entity → definicje klas modelowych

Java Persistence API

- Spring Data pozwala na połączenie się z wszelakimi źródłami baz danych
- JPA to framework, który pozwala nam mapować obiekty na tabele bazy danych
- entity classes → zwykłe klasy, które mogą być mapowane na kolumny lub tabele
- klasy encyjne nie mogą mieć publicznych pól i muszą mieć jednoznacznie identyfikujący klucz główny
- wykorzystywane adnotacje na poziomie klasy:
  + @Entity → oznaczenie klasy jako encyjnej
  + @Table → własności tabeli bazie danych (name, indexes)
- adnotacje na poziomie pól
  + @Id → klucz główny
  + @GeneratedValue → automatyczne generowanie wartości klucza
  + @Column → własności kolumny (name, nullable, unique, updatable)
  + @Temporal → wymagane dla typów Data i Calendar
  + @Transient → pola, które mają być pominięte przy mapowaniu
- związki między klasami mogą być jedno lub dwukierunkowe, wykorzystuje się do tego adnotacje → @OneToOne, @OneToMany, @ManyToOne, @ManyToMany
- w relacyjnych bazach danych nie ma związków dwukierunkowych, dlatego korzysta się z atrybutu mappedby
- park narodowy ma adnotacje @OneToMany(mappedby='park') a szlak ma adnotacje @ManyToOne oraz @JoinColumn(name='park')

JPA Repozytoria

- repozytoria tworzymy przez dziedziczenie z interfejsu JpaRepository, który ma dwa argumenty → klasę encyjną oraz primary key
- takie repozytorium musi mieć adnotację @Repository i używa Spring Data
- pozwala to na generowanie metod CRUD podczas runtime
- tworzenie dodatkowych zapytań w repozytorium wymaga odpowiednich nazw np. findByLoginAndPassword → dzięki temu JPA samo wygeneruje zapytanie
- jeśli zapytanie jest zbyt skomplikowane, można skorzystać z JPQL
- JPQL nie korzysta z kolumn w bazie danych, ale pól w klasach encyjnych
- przykład zapytania w JPQL, które szuka usera o danym haśle i loginie

```
@Query("select u from User where u.login = :login and u.password = :pasword")
Optional<User> find(  @Param("login") String login,
                      @Param("password") String password);
```

REST

Basic assumptions:

- stateless, interaction should be immune to server restart,
- application server caching services and other elements can be used to improve performance, as long as the elements returned by the service are not dynamically generated and can be cached,
- possible description with WADL or Swagger,
- the producer and the consumer must handle the same context and the content sent,
- low data overhead, ideal for devices with limited resources,
- often used in conjunction with AJAX technology.

REST Service

- aplikacja użytkownika sama renderuje dane, które otrzymuje od serwera
- dzięki temu serwer przygotowuje tylko dane (JSON, XML), a nie ich cały plik html
- web service → aplikacja typu klient serwer, która korzysta do komunikacji między nimi z protokołu HTTP
- restful powinien być niezależny od restartu serwera
- wszystkie requesty są od siebie całkowicie niezależne
- zasoby powinny być zhierarchizowane np. api/parks/id/routes
- słówko api oddziela strony biznesowe od stron z zasobami
- zapytania do serwisu mają dwa rodzaje parametrów:
  + path params → dynamiczne elementy ścieżki np. dla api/parks/1/routes/4 będą to numery
  + query params → standardowe parametry HTTP, znajdują się w adresie po znaku zapytania i są oddzielone ampersandem np. api/books/unread=true
- path params odpowiada za udostępnienie zasobu, a w query params dajemy dodatkowe meta informacje o zapytaniu (ale nie wskazują konkretnego zasobu)

HTTP Protokół

- GET → odczytywanie zasobu
- POST → tworzenie nowego zasobu
- DELETE → usuwanie zasobu
- PUT → aktualizowanie zasobu
- 200 (zakończono pomyślnie) 201 (dodano nowy zasób)
- 400 (zabroniono) 401 (brak autoryzacji) 404 (brak zasobu)
- za 500 odpowiada programista, za 400 użytkownik

Metody w web serwisach

- metody w web serwisach można tworzyć na dwa sposoby
- pierwszy to metoda, która zwraca obiekt lub listę obiektów → robimy to poprzez dodanie adnotacji np. GetMapping("adres") i ustalenie atrybutu, przy czym atrybutami metody są query params (to co jest po pytajniku)
- drugim sposobem jest zwracanie obiekty klasy ResponseEntity<myClass> i tutaj argumentem jest path variable, a zwracamy ResponseEntity.notFound().build() lub ResponseEntity.ok(myClass)

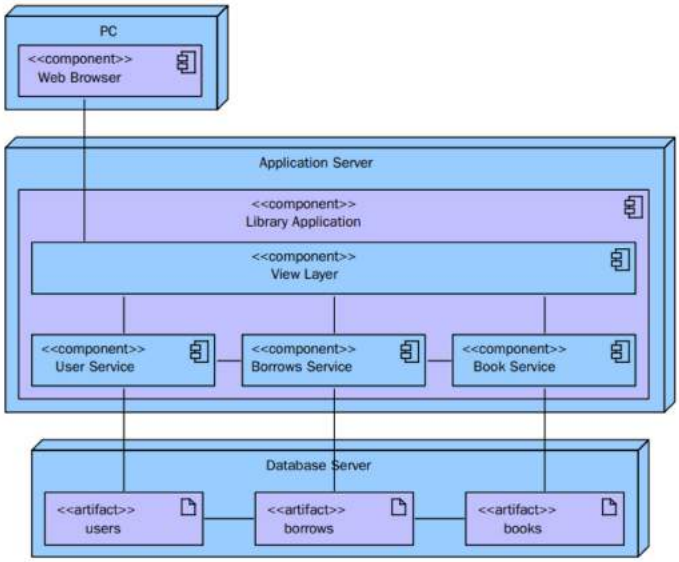
Used annotations:

- `@RestController` - registering a class as a controller for REST services, an instance of the class will be created automatically,
- `@RequestMapping` - register based address for all class methods,
- `@GetMapping` - handling GET HTTP requests,
- `@PostMapping` - handling POST HTTP requests,
- `@PathVariable` - mapping path parameter to method argument,
- `@RequestParam` - mapping query parameter to method argument,
- `@RequestBody` - mapping request body (default JSON) to an object.

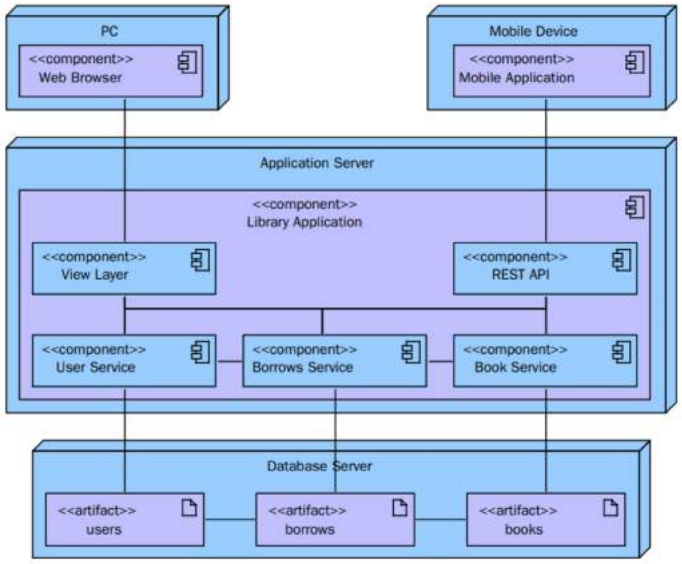
Microservices

Aplikacje monolityczne

- aplikacje monolityczne → aplikacja w ramach serwera aplikacji
- serwis w aplikacji monolitycznej to po prostu logika biznesowa
- warstwa widoku jest budowana po stronie serwera
- użytkownik przez przeglądarkę łączy się warstwą widoku, ona przekazuje zadania przez serwis i łączy się z serwerem baz danych
- jak aplikacja się rozwija pojawiają się problemy - długo trwa zapoznanie się z kodem, ciężko podmieniać kod na inne technologie, testy wykonują bardzo długo
- jak pojawiły się urządzenia mobilne, to chciały one otrzymywać np. tylko XML → do aplikacji monolitycznej obok warstwy widoku pojawiło się REST API
- w pewnym momencie REST API całkowicie zastąpiło warstwę widoku
- kolejnym pomysłem jest podział aplikacji monolitycznej na aplikacje konkretnych serwisów, które mogą się ze sobą komunikować przez REST endpoint



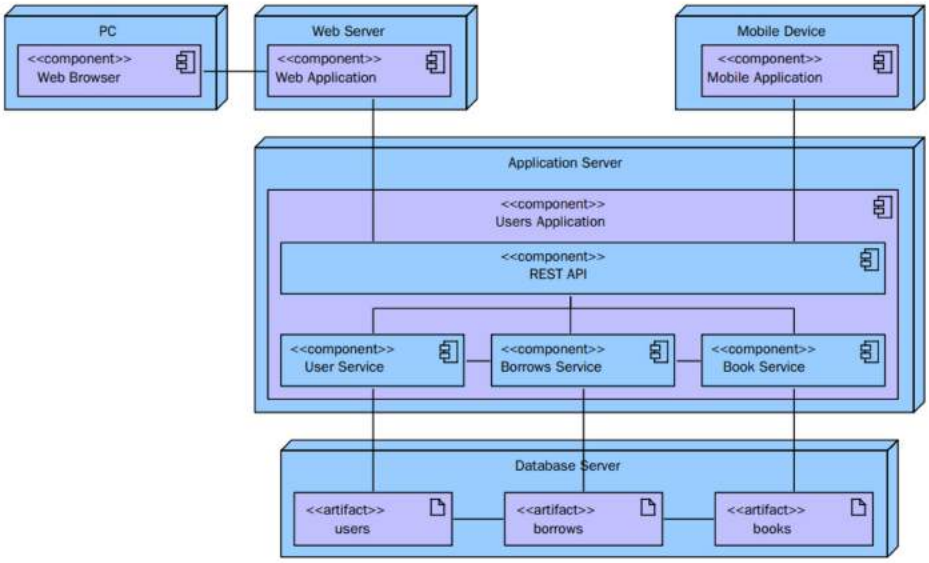
Z REST API:



Adding **REST endpoint** to **monolith**:

- two access modules in one application must be maintained,
- even more responsibility for project team which must have knowledge about next module,
- separate teams can develop different mobile clients without knowledge of base application implementation (only REST API documentation is required).

External Web Application:



Using **external web application**:

- front-end developers as separated team,
- knowledge about whole application implementation is not required (REST API contract is important),
- back-end developers can focus only on business logic implementation not considering interactions with users.

Mikroserwisy:

Mikroserwisy

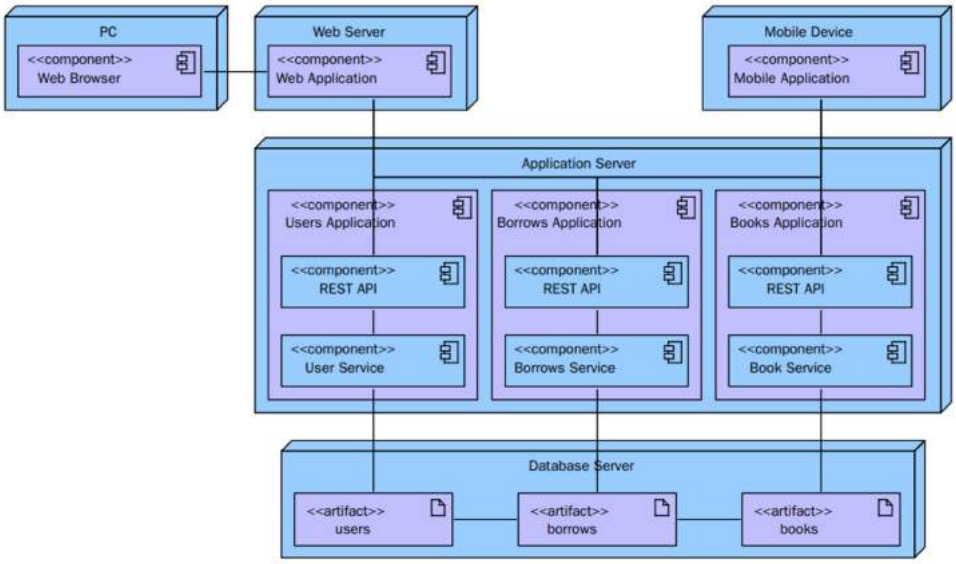
- komunikacja między serwisami może odbywać się między prywatnymi portami, więc nie są potrzebne dodatkowe zabezpieczenia
- jedna baza danych zapewnia nam transakcje ACID, klucze obce
- ACID → atomicity, consistency, isolation, durability
- takie mikroserwisy można podzielić na kilka różnych aplikacji, ale trzeba im zapewnić bezpieczne sposób komunikacji między sobą
- można też podzielić bazę danych na części odpowiadające danym serwisom, wówczas jeśli któraś z tabel korzysta z kluczy obcych warto dodać do tej "podbazy" tabele z kluczami obcymi

Using **microservices**:

- separated modules (different projects),
- different modules can be developed using different technologies,
- modules can communicate using e.g.: REST endpoints (but not only).

Można użyć jednych lub kilka serwerów, jeden lub kilka baz danych.

Jeden serwer:



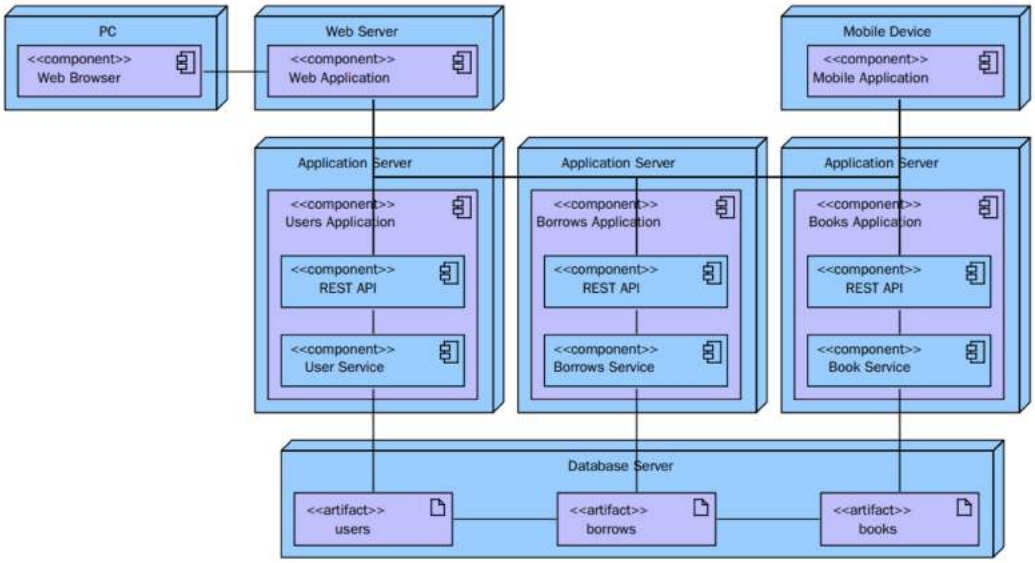
Using **single application server**:

- easy configuration,
- can use communication using non public ports,
- performance issues.

Using **single database**:

- simple in use ACID (atomicity, consistency, isolation, durability) transactions,
- one database is easy to maintain,
- schema changes must be coordinated between development teams,
- number of modules using the same database can lead to performance problems (e.g.: during long running transaction locks).

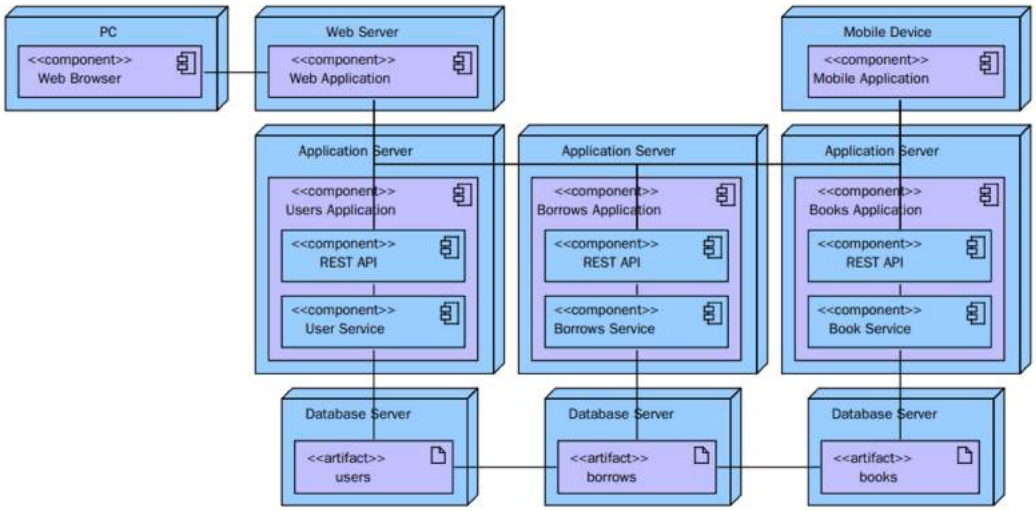
Odseparowane serwery aplikacji:



Using **separated application servers**:

- performance by using multiple machines,
- inside private network non public ports can be used,
- outside private network additional security mechanism can be required.

Kilka serwerów i kilka baz danych:



Using **separated database**:

- each service can use different architecture which is best for its purposes (e.g.: SQL, NoSQL, file system storage etc.),
- no ACID transactions or relationships unless we use distributed database with distributed transactions:
  - not always supported by NoSQL databases.

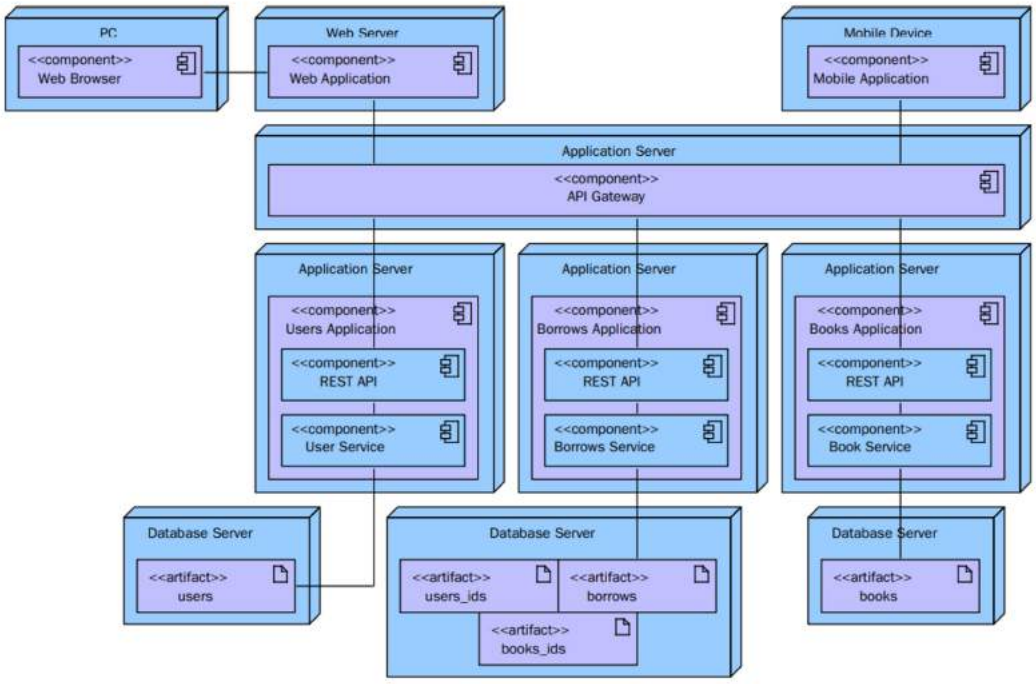
Some data may need to be **duplicated**:

- relationships between objects,
- store only identifiers, not whole data,
- synchronization between modules is required:
  - it can be done by sending synchronization events between modules.

API Gateway:

API Gateway

- API Gateway to aplikacja, która jest jedynym miejsce, gdzie zapamiętujemy lokacje potrzebnych modułów, pewien request dispatcher
- odpowiada za request dispatcher → rozdziela zadania na poszczególne serwisy
- może zapewnić load balancing → jak mamy kilka takich samych instancji serwisów, to może podzielić wykonywane zadania na te serwisy, tak żeby nie obciążać jednego
- zapewnia kompleksową synchronizację serwisów, jeśli muszą ze sobą współpracować, by wyprodukować jedną odpowiedź



Using **API Gateway**:

- clients do not need to locate all the services,
- there can be different instances providing API for different clients,
- data from number of services can be joined by the gateway,
- another module to maintain,
- additional communication.

Producer:

- some beans cannot be automatically produced by Spring Context,
- some beans require complex creation,
- calling constructors inside beans constructors is against dependency injection pattern,
- using `@Bean` annotation beans can be registered in Spring Context,
- `@Bean` annotation can be used in classes annotated with `@SpringBootApplication` or `@Configuration`.

Different beans configurations (different creation parameters) can be distinguished with qualifiers.

```
@Bean @Qualifier("library")
public RestTemplate restTemplate() {
    return new RestTemplateBuilder()
            .rootUri("http://localhost:8080/library")
            .build();
}
```

```
@Autowired @Qualifier("library")
private RestTemplate restTemplate;
```

Gateway:

- clients (mobile, web) need to communicate with different services,
- services decomposition should be transparent for clients,
- clients should not need to known location of all distributed services,
- there should be single gateway endpoint routing requests to particular services.

Gateway routing configuration is done by providing `RouteLocator` to Spring Context:

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder
            .routes()
            .route("library", r -> r
                    .host("localhost:8080")
                    .and()
                    .path("/api/books", "/api/books/**")
                    .uri("http://localhost:8081"))
            .build();
}
```

Discovery:

- services can be deployed on different addresses,
- services should not need to known exactly where other services are,
- there should be single (or distributed) catalog service,
- all services need to known only address of the catalog service.

Load balancer:

- there can be multiple instances of the same service,
- client can choose which service will be called,
- in order to balance the load different instances should be used,
- *hits* can be counted,
- round robin can be used,
- load balancing based on data ranges.

Local (client side) load balancer is integrated with Spring Cloud discovery service:

```
@Repository
public class RestRepository {

    @Autowired
    private LoadBalancerClient loadBalancerClient;

    @Autowired
    private RestTemplate restTemplate;

    public void delete() {
            URI uri = loadBalancerClient.choose("library")
                .getUri();
            restTemplate.delete(uri + "/" + id);

}
```

Load balancer can be also used automatically in Gateway:

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder
            .routes()
            .route("library", r -> r
                    .host("localhost:8080")
                    .and()
                    .path("/api/books", "/api/books/**")
                    .uri("lb://library"))
            .build();
}
```

Health check:

- there can be multiple instances of the same service,
- load balancer selects appropriate one to call,
- some of them can be down because of some errors,
- discovery service should be aware which are down,
- monitor tool for administrators would be helpful,
- can be called by service registry, load balancer or monitoring tool.

Most of health check expose `/health` endpoint with JSON response:

```
{
    "status": "UP"
}
```

and `200` HTTP response code.

Database migration:

- database schema needs to be created before application is stared,
- automatic generation mechanisms (eg. JPA) should not be used on production,
- schema can change with application development,
- on production with existing data appropriate migrations must be performed.

Some used Java libraries for migrations:

- Flyway (popular with Java EE applications),
- Liquibase (popular with Spring applications).

Centralized configuration:

- default configuration stored in `application.properties`,
- default configuration can be overwritten with environment variables,
- changing shared configuration requires modification in every module,
- configuration could be stored in centralized application.

JavaScript:

Javascript

- język interpretowany, wykonywany u klienta, słabe typowanie
- pozwala na manipulowania obiektami DOM
- różnica między HTML i DOM → HTML to kod źródłowy wysyłany przez serwer,a DOM to obiekty tworzone podczas skanowania pliku HTML
- obiekty DOM to np. przyciski, labelki, okna itp.
- JS dołączamy to plików HTML poprzez tag <script> lub poprzez dołączenie zewnętrznego pliku <script src=""> → oba sposoby kończymy </script>w

Drzewa obiektów DOM

- fetching elementów z wykorzystanie getElementBy → ID, TagName, ClassName
- modyfikowanie elementów (innerHTML) i atrybutów (attribute) czy stylów (style.key)
- przy dodawaniu (createElement) potem należy je uwzględnić w kontenerze za pomocą metody appendChild()

Skrypty .js w htmlu:

```html
<html>
    <head>
        <script src="script.js"></script>

        <script>
            <!-- Methods declaration. -->
        </script>
    </head>

    <body>
        <script>
            <!-- Script to be executed. -->
        </script>
    </body>
</html>
```

Manipulowanie obiektami DOM:

JavaScript allows to **modify** DOM tree elements:

- fetch element using:
  - `document.getElementById("id");`
  - `document.getElementsByTagName("name");`
  - `document.getElementsByClassName("name");`
- change element content with `element.innetHTML = "value";`
- change element attribute with `element.attribute = "value";`
- change CSS styles with `element.style.key = "value";`

```html
<p id="test"/><!-- empty paragraph -->

<script>
    let test = document.getElementById("test");
    test.innerHTML = "Hello World!";
    test.style.color = "red";
    test.align = "right";
</script>
```

Using the `innerHTML` can be used for Cross-site Scripting (XSS).

JavaScript allows for dynamic modification and creation of DOM elements:

- `document.createElement("tag name")` - create new element,
- `element.appendChild(el)` - add element to another one.

```html
<div id="container"></div>
```

```javascript
let container = document.getElementById("container");

let span = document.createElement("span");
let text = document.createTextNode("woof");

span.appendChild(text);
container.appendChild(span);
```

Events in js:

JavaScript allows executing specified functions as reaction to DOM tree **events**:

- loading particular element,
- change of element content,
- clicking on element (e.g.: buttons),
- ...

```html
<element event="JS code"/>
```

Selected events:

- `onchange` - element change,
- `onclick` - clicking element,
- `onmouseover` - cursor moved over the element,
- `onmouseout` - cursor moved out the element,
- `onkeydown` - pressing keyboard button,
- `onload` - loading document.

AJAX - Asynchronous JavaScript and XML:

- updating page without reloading whole content,
- downloading data from server,
- sending data to server.

Polityka wywołania REST API

- istnieją dwie główne polityki → Same Origin Policy (SOP) oraz Cross-Origin Resource Sharing (CORS)
- SOP polega, że tylko strony posiadające to samo pochodzenie (origin) mogą komunikować między sobą i wymieniać dane między sobą
- przeglądarka blokuje AJAX calle z innych serwerów
- do określenia wspólnego pochodzenia służą nam 3 elementy protokół, host i port
- jeśli sprawdzenie nie przejdzie (czyli przynajmniej jeden z elementów będzie różny) to przeglądarka zablokuje taką komunikację i klient nie dostanie danych o jakie prosi
- CORS jest mechanizmem, który przy pomocy zwykłych nagłówków w zapytaniu HTTP informuje przeglądarkę, że klient, który jest na konkretnym hoście, protokole i porcie może odpytywać serwer o dane.

SOP: Do określenia pochodzenia trzy rzeczy:

- protokół
- host
- port

Angular

Version elements:

- patch - bugs fixes compatible with previous version, eg. 2.7.4,
- minor - new functionalities compatible with previous versions, eg. 2.8.0,
- major - changes not backward compatible, eg. 3.0.0.

TypeScript:

- statically typed language transpiled into JavaScript:
  - the developer works in TypeScript, the browser receives understandable JavaScript;
- offers compatibility with the latest versions of JavaScript (ECMAScript 2020),
- possible transpilation to an older version, eg. ECMAScript 5:
  - at the developer's choice: `tsc --target`,
  - the problem of the standard version and the version of browsers.

TypeScript:

Visibility levels:

- public - default, fields and methods visible in other classes,
- protected- visible only in inheritance hierarchy,
- private - visible only in class.

```
class Animal {

    private _name: string;

    get name(): string {
        return this._name;
    }

    set name(name: string) {
        this._name = name;
    }

}
```

Angular framework:

Modules:

- Angular applications are divided into modules corresponding to particular functionalities,
- each application has a main module called `AppModule`,
- small applications can have only one module, large applications - hundreds of modules,
- modules defined as classes with the `@NgModule` decorator,
- decorators (functions) allow to attach metadata to a class.

Highlights **metadata** of the `NgModule` decorator:

- `declarations` - a list of components used to build application views,
- `exports` - list of components that should be available for use by other modules,
- `imports` - list of modules whose exported classes are used in the current module,
- `providers` - a list of providers enabling the building of service instances to be used in the entire application (in all modules),
- `bootstrap` - component representing the main view of the application (all other views are loaded into it), used only for `AppModule`.

Components:

- represent fragments of views that make up the application interface,
- defined as classes with the `@Component` decorator,
- are represented by additional tags placed in the HTML code, e.g.:

```
<body>
  <app-root></app-root>
</body>
```

Component defines:

- template used to build a view fragment (HTML tags also including tags for other components),
- data for presentation (model),
- behaviors (event handling functions).

`@Component()` - the most important metadata:

- `selector` - a CSS selector that specifies which tags on the page are to be filled with the component's content,
- `templateUrl` - path to the `.html` file with the template,
- `styleUrls` - a list of CSS style files for this components.

`@Input()` - allows to pass attributes to a component from parent.

`@Output()` - allows to pass events to parent from component.

Templates:

- used by components to generate content in the browser window,
- the syntax is based on the syntax of the HTML language,
- contain additional elements:
  - tags for attaching other components to the view,
  - directives to control the process of generating the resulting HTML code.

There are 4 **forms of data binding** available:

- value interpolation: `{{...}}`,
- DOM element property binding: `[property]`,
- binding event handler: `(event)`,
- bidirectional data binding: `[(...)]`.

Interpolation:

- allows to determine the value used in the view based on an expression,
- the expression most often refers to fields / properties of a component class,
- expression in parentheses `{{...}}` is converted to a string before being put in the view.

```
<h3>{{imgTitle}}</h3>
<img src="{{imgUrl}}">
```

Expressions (template expressions):

- can carry out additional operations,
- should not cause side effects,
- should be quick to make,
- should be as short and simple as possible,
- complex logic should be placed in the component method and called in the expression,
- expression should be idempotent.

```
<p>The threshold has been exceeded {{score - threshold}} points.</p>
```

Property binding:

- expression values can also be associated with the properties of DOM tree elements and component properties,
- this bond is unidirectional:
  - changing the value of an expression changes the value of the property, but not vice versa;
- bindings refer to the properties of the DOM tree elements and not to HTML tag attributes.

```
<button [disabled]="unchanged">Cancel</button>
<img [src]="imgUrl">
<app-book-detail [book]="selectedBook">
```

CSS properties binding

- the binding object can be CSS classes,
- or individual CSS properties.

```
<div [class.special]="special">...</div>
```

```
<button [style.color]="special ? 'red': 'green'">
<button [style.background-color]="canSave ? 'cyan':'grey'">
```

Event binding - enables calling of event handling functions defined in the component class in response to user actions, e.g .:

```
<button (click)="onSave()">Save</button>
```

Bidirectional binding:

- changing the value of the associated field changes the value of the property,
- changing the value of a property changes the value of the field,
- especially useful when working with forms,
- built by hand or with `ngModel`.

```
<input [value]="name" (input)="name=$event.target.value" >
```

```
<input [(ngModel)]="name">
```

Directives:

- HTML documents have a static structure,
- Angular view templates are dynamic,
- the resulting HTML is the result of processing the template in accordance with the directives placed in it,
- example directives:
  - `\*ngFor` - adding elements in a loop,
  - `\*ngIf` - displaying the item conditionally.

Services:

- application logic should not be implemented in component classes:
- the component works in the context of a specific view template - it is difficult to reuse the logic embedded in the component class elsewhere in the application,
- the component should define the fields and methods for data binding, and delegate the logic to the services,
- services implement the application logic in a way that is independent of the user interface,
- easy to use in many different contexts,
- services are delivered to components by dependency injection.

Routing:

- typical web applications consist of many views between which the user navigates,
- `RouterModule` allows to define addresses that will display selected components (views) of the application,
- The `<base href =" / ">` tag in the `<head>` section of the `index.html` file specifies the base path for addresses within the application.

Use of web services:

- data presented in the front-end application is typically downloaded from the server (from the back-end),
- user input is saved on the server
  - data from the forms on the website, the contents of the basket / order, etc.;
- the back-end application provides its functions in the form of web services:
  - e.g. in REST architecture;
- the `HttpClient` service allows sending HTTP requests to the back-end.

Inversion of Control is a **principle in software engineering that transfers the control of objects or portions of a program to a container or framework**. We most often use it in the context of object-oriented programming.

In contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code. To enable this, frameworks use abstractions with additional behavior built-in. **If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.**

The advantages of this architecture are:

- decoupling the execution of a task from its implementation
- making it easier to switch between different implementations
- greater modularity of a program
- greater ease in testing a program by isolating a component or mocking its dependencies, and allowing components to communicate through contracts