Internet Services Architectures

Spring Platform

Michał Wójcik

Spring Platform

Spring refers to an entire family of projects built on top of a Spring Framework project:

Spring Boot,
Spring Framework,
Spring Data,
Spring Cloud,
Spring Cloud Data Flow,
Spring Security,
Spring Session,
Spring Integration,
Spring HATEOAS,
Spring REST Docs,
Spring Batch,
Spring AMQP,
Spring for Android,
Spring CredHub,
Spring Flo,
Spring for Apache Kafka,
Spring LDAP,
Spring Mobile,
Spring Roo,
Spring Shell,
Spring Statemachine,
Spring Vault,
Spring Web Flow,
Spring Web Services.

Spring Framework

Spring Framework - a framework that allows to create complex web and enterprise-class applications running on a Java virtual machine:

- supports container based dependency injection (DI),
- supports container based Inversion of Control (IoC),
- can be used to create desktop applications,
- name suggests fresh approach to business applications' development,
- considered as competition to Java EE.

Spring Framework

Spring Framework history:

- 2002 *Expert One-on-One J2EE Design and Development* book by *Rod Johnson*,
- 2003 Spring first release,
- 2004 Spring 1.0,
- 2007 Spring 2.5,
- 2012 Spring 3.2,
- 2014 Spring 4.0,
- 2017 Spring 5.0,
- 2022 Spring 6.0.

Spring Framework

Spring Framework is developed in a modular architecture:

- each module is responsible for different scope of behaviours,
- some of them can be used independently.

Some of the modules:

- `spring-core` - contains mainly core utilities and common stuff,
- `spring-context` - provides Application Context, that is Spring's DI container,
- `spring-mvc` - model-view-controller framework with requests to handlers dispatcher,
- `spring-data` - access to various data sources (SQL databases, NoSQL databases, etc.),
- `spring-security` - authorization and authentication mechanisms.

Spring Boot

Spring Boot - a mechanism facilitating the development of applications based on the Spring platform:

- simplifies projects configuration,
- matches compatible platform (and not only) elements (controls libraries versions),
- simplifies distribution package preparation:
  - web application require to be launched within web server in order to support HTTP requests,
  - allows to uses embedded Tomcat server configured on the project level.

Spring Boot Application

Instead of creating new projects from scratch and adding dependencies manually we can:

1. use Spring Initializr: http://start.spring.io/,
2. set GroupId i ArtifactId,
3. select desired modules (dependencies):
   - Web,
   - JPA,
   - Derby (database),
   - ...
4. generate projects,
5. download ZIP archive and unpack,
6. open project in favorite IDE.

Spring Boot Application

Main entry point:

```
@SpringBootApplication
public class SimpleRpgApplication {

    public static void main(String[] args) {
        SpringApplication.run(SimpleRpgApplication.class, args);
    }

}
```

Application can be started simply by:

```
java -jar simple-rpg.jar
```

It starts embedded Tomcat server and deploys the application.

Spring Context

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).

How to provide dependencies

There are three methods for dependency injection:

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

Injection with constructor

```
public class UserService {

    private UserRepository repository;

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

}
```

```
UserRepository repository = new UserRepository();
UserService service = new UserService(repository);
```

Injection with constructor:

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

Injection with setters

```
public class UserService {

    private UserRepository repository;

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

}
```

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

Injection with setters:

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

Field injection

```
public class UserService {

    private UserRepository repository;

}
```

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

Field injection:

- no additional methods,
- requires reflection mechanism.

DI in Spring

Dependency Injection in Spring Container:

- supports all three methods,
- declared with `@Autowired` annotation.

DI in Spring

```
public class UserService {

    private UserRepository repository;

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

}
```

```
public class UserService {

    private UserRepository repository;

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

}
```

```
public class UserService {

    @Autowired
    private UserRepository repository;

}
```

Controlled bean life-cycle

In case of container controlled beans there is possibility to act on creation or destroy event.

```
@Component
public class DataStoreComponent {

    @PostConstruct
    public void init() throws Exception {
        //..
    }

    @PreDestroy
    public void clean() throws Exception {
        //..
    }

}
```

Initialization:

- constructor should be used only for creating object, no logic initialization,
- `@PostConstruct` methods are called after object creation and after all dependencies injection.

Command Line Application

Command line applications in Spring Boot can be achieved with components implementing `CommandLineRunner` interface:

```
@Component
public class ApplicationCommand implements CommandLineRunner {

    private UserService service;

    @Autowired
    public CommandLine(UserService service) {
        this.service = service;
    }

    @Override
    public void run(String... args) throws Exception {
        service.findAll().forEach(System.out::println);
    }

}
```

Helpful resources

Additional information:

Guides:

- Spring Team, *Spring Quickstart Guide*, https://spring.io/quickstart.
- Spring Team, *Spring Guides*, https://spring.io/guides.
- Baeldung, *Spring Tutorial*, https://www.baeldung.com/spring-tutorial.
- Baeldung, *Learn Spring Boot*, https://www.baeldung.com/spring-boot.
- Baeldung, *Spring Tutorials*, https://www.baeldung.com/category/spring/.
- Baeldung, *Spring Boot Tutorials*, https://www.baeldung.com/category/spring/spring-boot/.

Additional information:

- https://www.baeldung.com/spring-boot-console-app

Internet Services Architectures

Spring Data JPA

Michał Wójcik

Spring Data

Spring Data is an umbrella module providing consisted assess mechanism to different storage types.

Spring Data

Main modules:

- Spring Data Commons - core concepts for every Spring Data module
- Spring Data JDBC - SQL database access using JDBC,
- Spring Data JDBC Ext - support for databases specific ex tensions,
- Spring Data JPA - SQL database access using JPA,
- Spring Data KeyValue - support for key-value stores (non-relational databases, map-reduce frameworks),
- Spring Data LDAP - support for LDAP catalog,
- Spring Data MongoDB - support for document oriented database MongoDB,
- Spring Data Redis - support for in-memory key-value database Redis,
- Spring Data REST - automatically exports repositories as rest resources,
- Spring Data for Apache Cassandra - support for wide-column store Apache Casandra,
- Spring Data for Apache Geode - support for in-memory data grid Apache Geode,
- Spring Data for Apache Solr - support for Lucene based search platform Apache Solr,
- Spring Data for Pivotal GemFire - support for distributed data management GemFire.

Java Persistence API

Jakarta Persistence API (JPA):

- formerly Java Persistence API,
- specification for object-relational mapping (ORM) libraries,
- popular in Java frameworks:
  - Java SE, Java EE, Jakarta EE, Spring Framework, Play Framework;
- does not require creating complex DAO (Data Access Object),
- supports ACID transactions (Atomicity, Consistency, Isolation, Durability),
- databases system provider independent:
  - JDBC drivers for all most popular database servers,
  - in simpler cases, it is possible to avoid playing with SQL.

JPA - implementations

JPA is only a standard, there is a number of implementations:

- Hibernate from Red Hat,
- Toplink from Oracle,
- OpenJPA from Apache Software Foundation,
- EclipseLink from Eclipse Foundation, reference implementation.

Entity classes

Entity classes:

- classes mapped to tables stored in the database,
- simple POJO (Plain Old Java Object) classes,
- class fields should not be public,
- each entity object must have unique identifying key:
  - complex keys are represented with separate classes implementing `hashCode()` i `equals()` methods;
- defined with annotations.

JPA - important annotations

Annotations on **class** level:

- `@Entity` – mark class as entity (requried),
- `@Table` – table properties, e.g.:
  - `name` – table name,
  - `indexes` – additional indexes (besides default index for primary key).

JPA - important annotations

Annotations on **field** level:

- `@Id` – primary key,
- `@GeneratedValue` – automatically generated value for primary key,
- `@Column` – column properties, e.g.:
  - `name` – column name,
  - `nullable` – if can be null or if is required,
  - `unique` – if column values are unique,
  - `updatable` – if column value can up updated after row creation;
- `@Temporal` – required for `Date` i `Calendar` types:
  - allows to user database date/time types to be used to store values (if database supports them);
- `@Transient` – fields which will be skipped during object-relational mapping.

Example entity

```java
@Getter
@Setter
@NoArgsConstructor
@Entity
@Table(name = "users")
public class User {

    @Id
    private UUID uuid;

    @Column(unique = true)
    private String login;

    @Column(name = "user_name")
    private String name;

    private String password;

    @Column(unique = true)
    private String email;

}
```

Annotations `@Getter`, `@Setter`, `@NoArgsConstructor` ale from Lombok Project.

Entity relationships

Database tables **relationships** can be mapped as **connection** between entity classes:

- directional - one class contains reference to the second one,
- bidirectional- both classes contain reference to each other.

Annotations defining **relationships**:

- `@OneToOne`,
- `@OneToMany`,
- `@ManyToOne`
- `@ManyToMany`.

Selected properties of annotations:

- `mappedBy` - marks field being owner of the relationship,
- `cascade` - cascade operations on elements being in relationship.

Example relationship

```java
@Getter
@Setter
@NoArgsConstructor
@Entity
@Table(name = "users")
public class User {

    @Id
    private UUID uuid;

    @Column(unique = true)
    private String login;

    @Column(name = "user_name")
    private String name;

    private String password;

    @Column(unique = true)
    private String email;

    @OneToMany(mappedBy = "user")
    private List<Character> characters;

}
```

```java
@Getter
@Setter
@NoArgsConstructor
@Entity
@Table(name = "characters")
public class Character {

    @Id
    private UUID uuid;

    private String name;

    @ManyToOne
    @JoinColumn(name = "user")
    private User user;

}
```

Connection configuration

There are several methods for defining connection to the database in Spring. One of the easiest is to user `application.properties` configuration file stored in project sources.

Data source settings:

```
spring.datasource.url=jdbc:h2:mem:simple-rpg
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=admin
spring.datasource.password=password
```

JPA settings:

```
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.generate-ddl=true
```

Hibernate specific settings:

```
spring.jpa.hibernate.ddl-auto=update
```

H2 specific settings:

```
spring.h2.console.enabled=true
```

Using H2 database

H2 is small easy in use in-memory database.

Required dependency:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

In order to use built-in web console http server must be enabled.

Repositories

Spring offers automatic data repositories creation based on interface implementation:

```
@Repository
public interface UserRepository extends JpaRepository<User, UUID> {

}
```

The `JpaRepository` is only one of a number of different repositories types which can be used.

Custom queries

On of a number of ways adding custom queries is by using derived query methods.

```
@Repository
public interface UserRepository extends JpaRepository<User, UUID> {

    Optional<User> findByLoginAndPassword(String login, String password);

}
```

Custom queries

In case preparing derived query method is too complicated, queries in JPQL can be used.

```
@Repository
public interface UserRepository extends JpaRepository<User, String> {

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

}
```

JPQL - basic queries

```
@Entity
public class Product {

    @Id
    UUID id;

    String name;

    String description;

    Integer price;

    Integer amount;

}
```

JPQL - basic queries

Select all products:

```
SELECT p FROM Product p
```

Select all products with specified name:

```
SELECT p FROM Product p WHERE p.name = :name
```

JPQL - basic queries

Select products with name matching regular expression - `LIKE` operator:

```
SELECT p FROM Product p WHERE p.name LIKE :name
```

Select products with name containing `VT-x` :

```
SELECT p FROM Product p WHERE p.description LIKE '% VT-x %'
```

Ignore case:

```
SELECT p FROM Product p WHERE LOWER(p.name) LIKE LOWER('%Laptop%')
```

Get products with low stock sorted by name:

```
SELECT p FROM Product p WHERE p.amount < 5 ORDER BY p.name
```

Derived methods vs JPQL

Derived query methods and JPQL syntax:

keyword	derived method	JPQL
And	findByLastNameAndFirstName	where x.lastName = ?1 and x.firstName = ?2
Or	findByLastNameOrFirstName	where x.lastName = ?1 or x.firstName = ?2
Is,Equals	findByFirstName, findByFirstNameIs, findByFirstNameEquals	where x.firstName = ?1
Between	findByStartDateBetween	where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	where x.age <= ?1
GreaterThan	findByAgeGreaterThan	where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	where x.age >= ?1
After	findByStartDateAfter	where x.startDate > ?1
Before	findByStartDateBefore	where x.startDate < ?1
IsNull	findByAgeIsNull	where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	where x.age not null

Derived methods vs JPQL

Derived query methods and JPQL syntax:

keyword	derived method	JPQL
Like	findByFirstNameLike	where x.firstName like ?1
NotLike	findByFirstNameNotLike	where x.firstName not like ?1
StartingWith	findByFirstNameStartingWith	where x.firstName like ?1 (parameter bound with appended %)
EndingWith	findByFirstNameEndingWith	where x.firstName like ?1 (parameter bound with prepended %)
Containing	findByFirstNameContaining	where x.firstName like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastNameDesc	where x.age = ?1 order by x.lastName desc
Not	findByLastNameNot	where x.lastName < > ?1
In	findByAgeIn(Collection ages)	where x.age in ?1
NotIn	findByAgeNotIn(Collection ages)	where x.age not in ?1
True	findByActiveTrue()	where x.active = true
False	findByActiveFalse()	where x.active = false
IgnoreCase	findByFirstNameIgnoreCase	where UPPER(x.firstName) = UPPER(?1)
First,Top	queryFirst10ByLastName	where x.lastName asc limit ?

Helpful resources

Guides:

- Baeldung, *JPA Tutorials*, https://www.baeldung.com/persistence-with-spring-series.
- Baeldung, *Spring Data Tutorials*, https://www.baeldung.com/category/persistence/spring-persistence/spring-data/.

Additional information:

- https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa
- https://www.baeldung.com/spring-data-derived-queries
- https://www.baeldung.com/jpa-entities
- https://www.baeldung.com/jpa-composite-primary-keys
- https://www.baeldung.com/jpa-strategies-when-set-primary-key
- https://www.baeldung.com/jpa-one-to-one
- https://www.baeldung.com/jpa-join-column
- https://www.baeldung.com/jpa-joincolumn-vs-mappedby
- https://www.baeldung.com/jpa-queries

Internet Services Architectures

Spring MVC - REST Services

Michał Wójcik

Approach to web applications

Before the era of advanced personal devices (e.g. smartphone, smartwatch, etc.), a typical server application generated HTML documents that were rendered by web browsers.

Today, the browser client is only one of many channels for accessing services on the server:

- mobile applications - devices such as smartphones, tablets, smartwatches,
- software in devices such as Smart TV, Smart Car,
- intelligent assistants: Alexa, Siri.

New client applications require programming APIs instead of HTML documents.

Approach to web applications

Since the server must provide an API implementing the business logic anyway, the browser client can also use it. Providing the same server API to all types of client applications (mobile applications, websites, etc.) makes the application much easier to maintain.

Web browser client applications (web frontend) use JavaScript-based frameworks:

- Angular (Google),
- React (Facebook),
- Vue.js,
- Backbone.js
- ...

Web services

Web services - client and server applications communicating with each other via the web using the HTTP protocol:

- interoperability between applications running on different platforms and frameworks,
- descriptions processed by applications,
- use of textural representation like XML or JSON,
- loosely coupled services can be combined to create complex operations.

RESTful Web Services

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 OpenAPI,
- 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.

Resources

Resources should be arranged in a hierarchy (nouns instead of verbs):

- `api/books` - all books,
- `api/books/1` - book with index 1,
- `api/books/1/authors` – authors of the specified book,
- `api/books/1/authors/1` – author with index 1 of the specified book.

Instead of indexes, other values that uniquely describe the item can be used:

- next index,
- domain key (e.g. isbn for books),
- generated UUID.

Resources

Parameter types:

- path param - parameter that is an element of the address, indicates resources,
- query param - the parameter appended to the address (after the `?` character), make the user's (client application) query more precise:
  - paging: `api/books?offset=10&limit=5` - download five books starting from 10.
  - filtering: `api/books?unread=true` - download only unread books,
  - sort: `api/books?sort=latest` - get books sorted by release date.

Resources operations

Resources management is done with HTTP methods:

- `PUT` - insert resource,
- `GET`- retrieve resource,
- `POST` - add resource to collection,
- `DELETE` - delete resource.

operation	resource	effect
GET	`api/books`	fetch all books
GET	`api/books/1`	fetch single specified book
POST	`/api/books`	add new book
PUT	`/api/books`	replace collection of books
PUT	`/api/books/1`	replace specified book
DELETE	`/api/books`	delete whole collection
DELETE	`/api/books/1`	delete specified book

Resources operations

Because POST is not idempotent (different result for subsequent calls), it should not be used in REST approach. Instead PUT request can be used. Additionally, PATCH request for partial update can be used.

operation	resource	effect
GET	`api/books`	fetch all books
GET	`api/books/1`	fetch single specified book
POST	`/api/books`	add new book
PUT	`/api/books`	replace collection of books
PUT	`/api/books/1`	replace specified book
PATCH	`/api/books/1`	book modification
DELETE	`/api/books`	delete whole collection
DELETE	`/api/books/1`	delete specified book

REST as CRUD

operacja	REST	SQL
create	PUT	INSERT
read	GET	SELECT
update	PACH	UPDATE
delete	DELETE	DELETE

Request result

Request result should contain HTTP response code.

request	resource	example response code
GET	`api/books`	200 OK
GET	`api/books/1`	200 OK
POST	`/api/books`	201 Created, (+location)
PUT	`/api/books`	201 Created, 204 No Content, (+location)
PUT	`/api/books/1`	201 Created, 204 No Content, (+location)
PATCH	`/api/books/1`	204 No Content
DELETE	`/api/books`	202 Accepted, 204 No Content, 200 OK
DELETE	`/api/books/1`	202 Accepted, 204 No Content, 200 OK

Additionally, 401 Unauthorized, 403 Forbidden or 404 Not Found.

Tools

Popular tools for testing REST API:

- Postman - standalone application (downloaded from homepage),
- various REST clients as web browser plugins,
- SoapUI - standalone,
- Visual Studio Code REST Client plugin,
- IntelliJ IDEA HTTP Client plugin,
- ...

Rest service

```java
@RestController
@RequestMapping("/shop")
public class ShopController {

    private OrderService service;

    public ShopController(OrdersService service) {
        //...
    }

    @GetMapping("/orders")
    public List<Order> listOrders(@RequestParam(required = false) String sort) {
        //...
    }

    @GetMapping("/orders/{id}")
    public ResponseEntity<Order> getOrder(@PathVariable UUID id) {
        //...
    }

    @PostMapping("/orders")
    public ResponseEntity<Void> addOrder(@Request BodyOrder order) {
        //...
    }

}
```

REST service

The above configuration allows to send the following HTTP requests (assuming the server was started on the default port 8080):

- GET on `localhost:8080/shop/orders` - returns all orders,
- GET on `localhost:8080/shop/orders?sort=asc` - returns all orders sorted ascending,
- GET on `localhost:8080/shop/orders/9eaee866-6eb3-11ea-bc55-0242ac130003` - returns order with specified id,
- POST on `localhost:8080/shop/orders` - adds new order.

REST service - annotations

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.

Binding parameters and responses

```
@GetMapping("/orders")
public List<Order> listOrders(@RequestParam(required = false) String sort) {
    List<Order> orders;
    if (sort != null) { //optional query param
         orders = service.findAllSorted(sort);
    } else {
        orders = service.findAll();
    }
    return orders; //HTTP 200 code with body automatically converted to JSON array
}
```

```
@GetMapping("/orders/{id}")
public ResponseEntity<Order> getOrder(@PathVariable UUID id) {
    Order order=service.find(id);
    if (order == null){
        return ResponseEntity.notFound().build(); //HTTP 404 error code
    } else {
        return ResponseEntity.ok(order); //HTTP 200 code with body in JSON format
    }
}
```

Response body:

- automatically built (by default to JSON format) based on the object returned by the method,
- using the `ResponseEntity` class that allows to control all aspects of the HTTP response (code, body, headers).

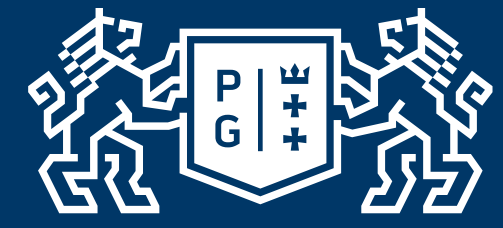
Helpful resources

Guides:

- Baeldung, *REST with Spring Tutorial*, https://www.baeldung.com/rest-with-spring-series.
- Baeldung, *Spring Data Tutorials*, https://www.baeldung.com/category/persistence/spring-persistence/spring-data/.

Additional information:

- https://www.baeldung.com/spring-controller-vs-restcontroller
- https://www.baeldung.com/spring-request-response-body
- https://www.baeldung.com/spring-pathvariable
- https://www.baeldung.com/spring-request-param
- https://www.baeldung.com/spring-requestmapping
- https://www.baeldung.com/spring-response-status
- https://www.baeldung.com/spring-request-response-body
- https://www.baeldung.com/building-a-restful-web-service-with-spring-and-java-based-configuration



Internet Services Architectures

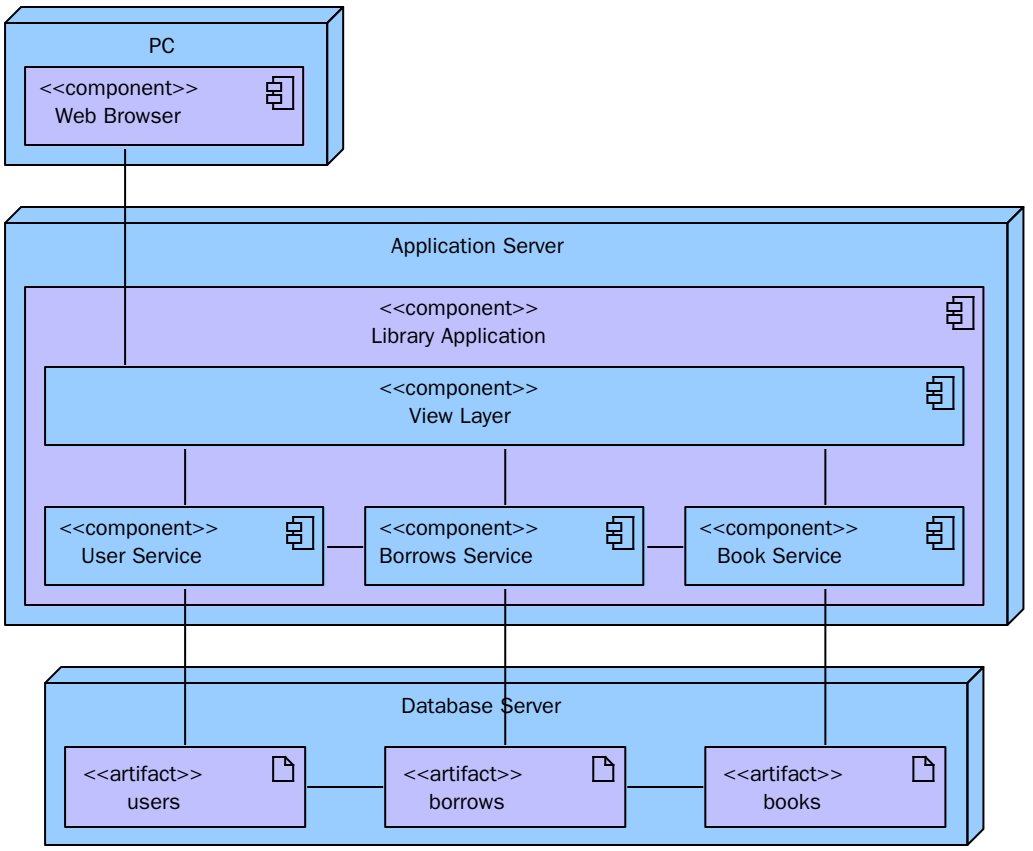
Microservices

Michał Wójcik

POLITECHNIKA GDAŃSKA

Monolithic Application

Traditional monolithic approach to building applications. Most common in old (but still maintained) systems.



Monolithic Application

Monolithic application:

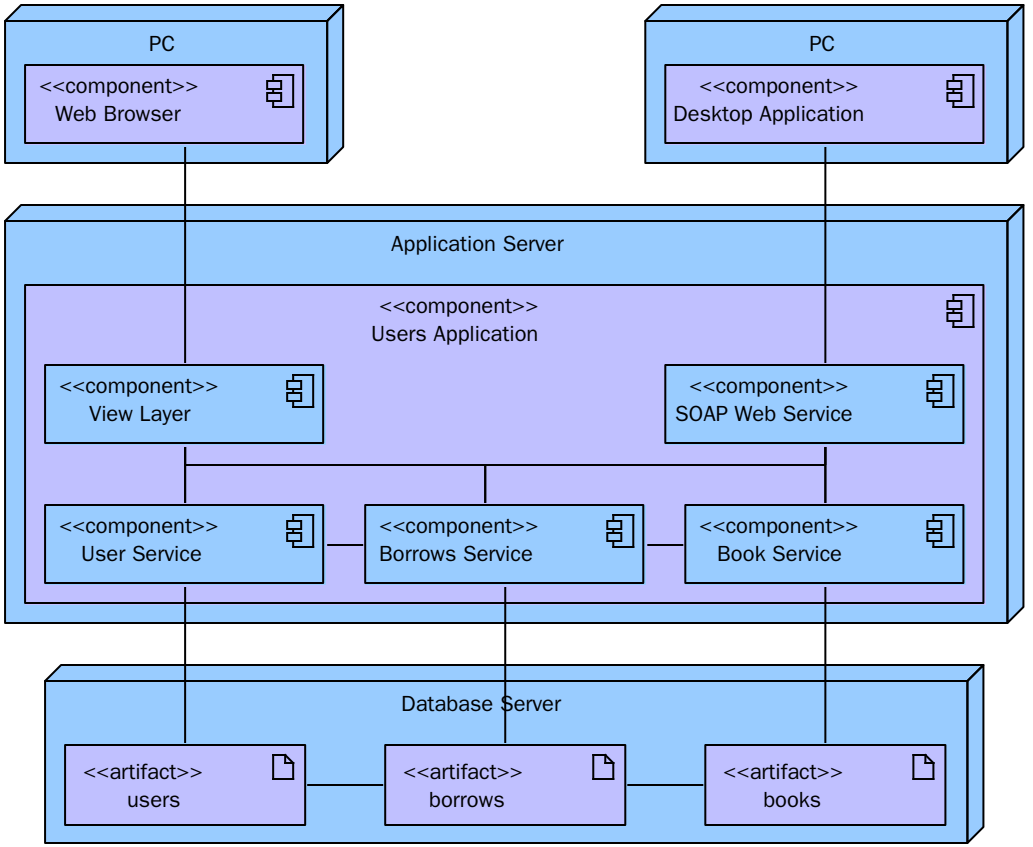
- small applications are simple to develop, everything in single place,
- all teams work on the same project (maybe different modules), know the same context,
- easy to deploy (e.g.: single war file for Java web applications),
- easy scaling by running multiple instances.

Problems start when application goes **big**:

- a lot of code in one project is difficult to understand,
- high cost of introducing new project members,
- it's difficult for teams to work independently,
- the larger code base is the slower IDE is,
- long CI/CD process,
- long-term commitment to technology stack.

Programming API

With increasing popularity of introducing interoperability with desktop applications between and external systems, there was demand for programming API instead of HTML web pages.



Programming API

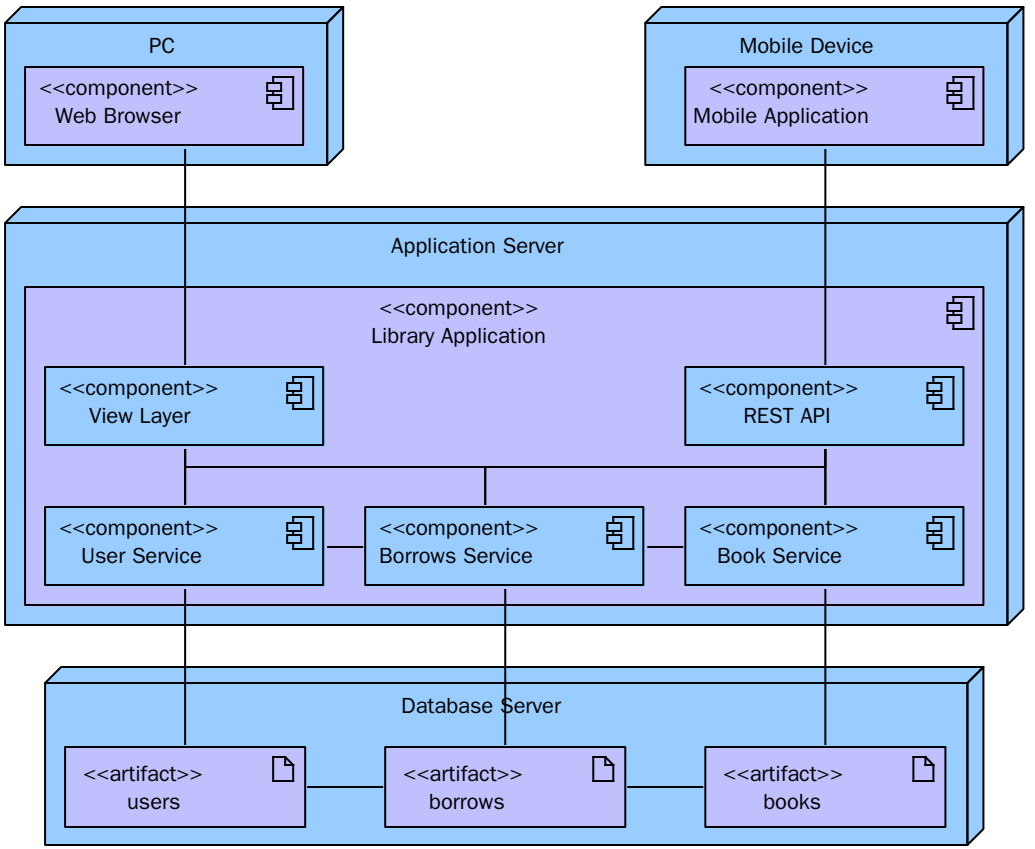
Adding **Programming API** to **monolith**:

- 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 desktop clients without knowledge of base application implementation (only documentation is required).

Popular implementations: SOAP Web Services, Remote Procedure Call (RPC), Remote Method Invocation (RMI).

Programming API - Web API

With increasing usage of devices like smartphones, tables, smart TV etc. there was demand for lightweight programming API.



Programming API - Web API

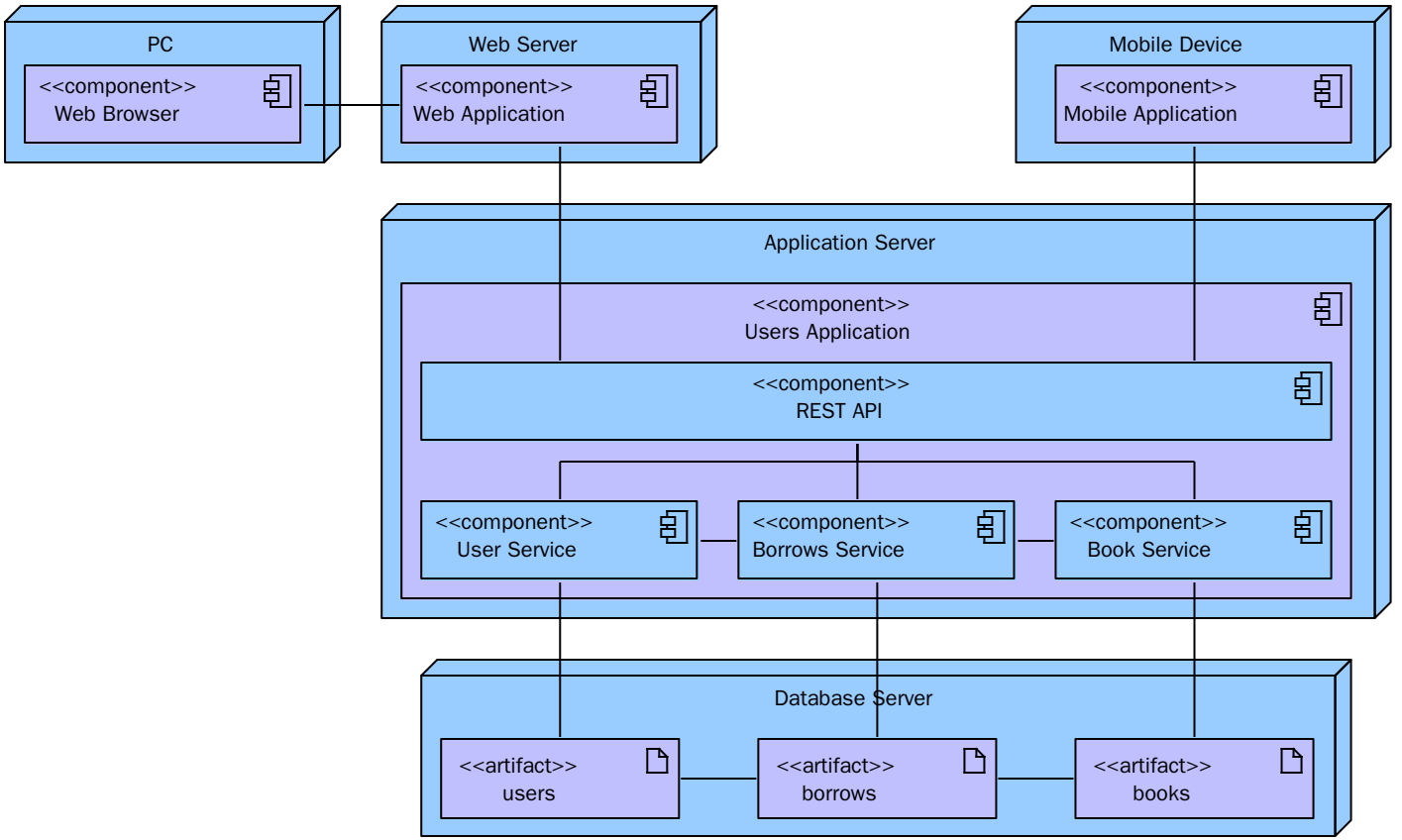
Adding **Web API** to **monolith**:

- 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 desktop clients without knowledge of base application implementation (only documentation is required).

Popular implementations: REST, GraphQL.

Single entry point and web application

If we must maintain Web API, maybe it should be the only entry point?



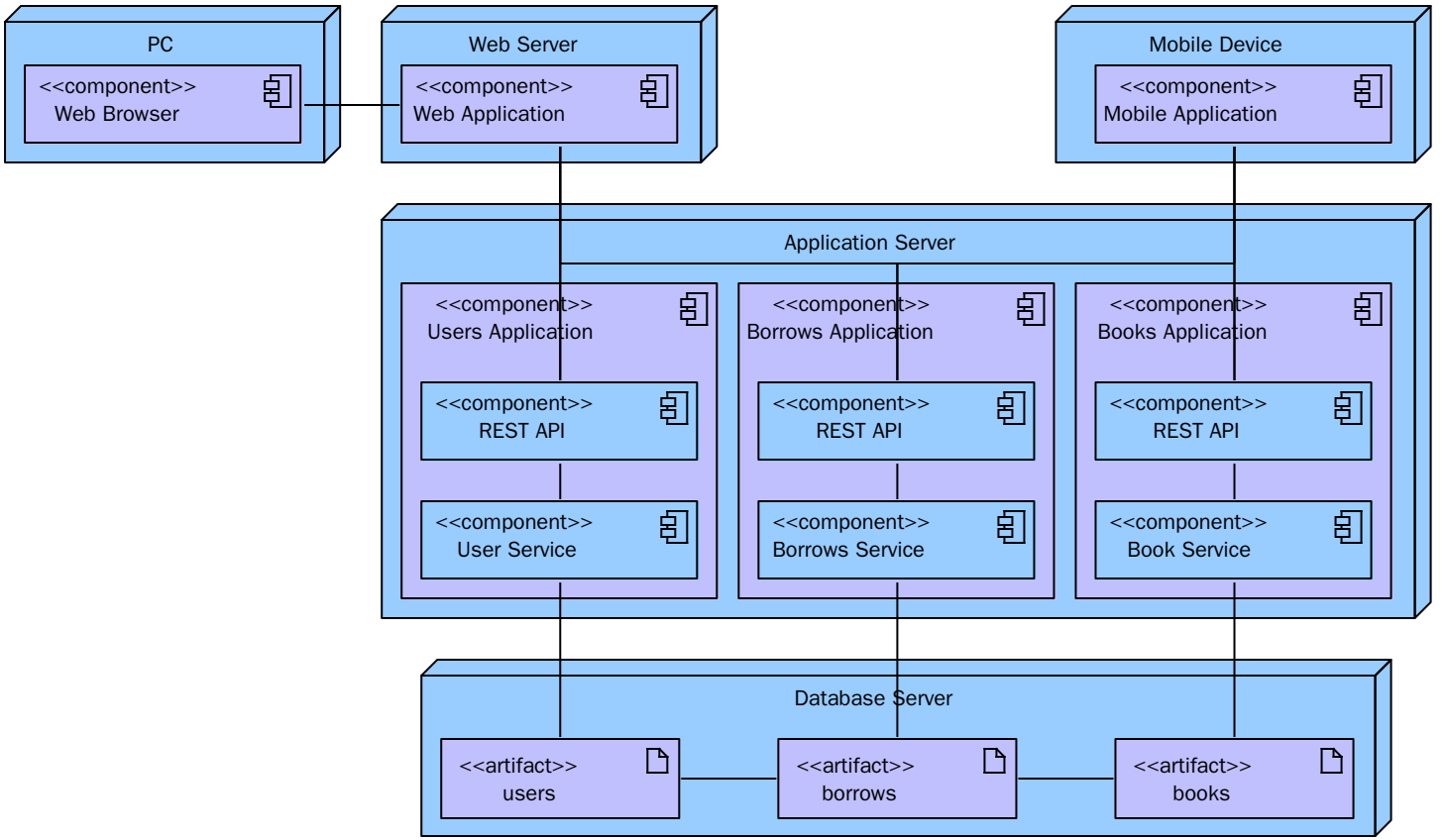
Single entry point and web application

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.

Microservices on Single Server

Can we decompose the monolith application?



Microservices on Single Server

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).

Using **single application server**:

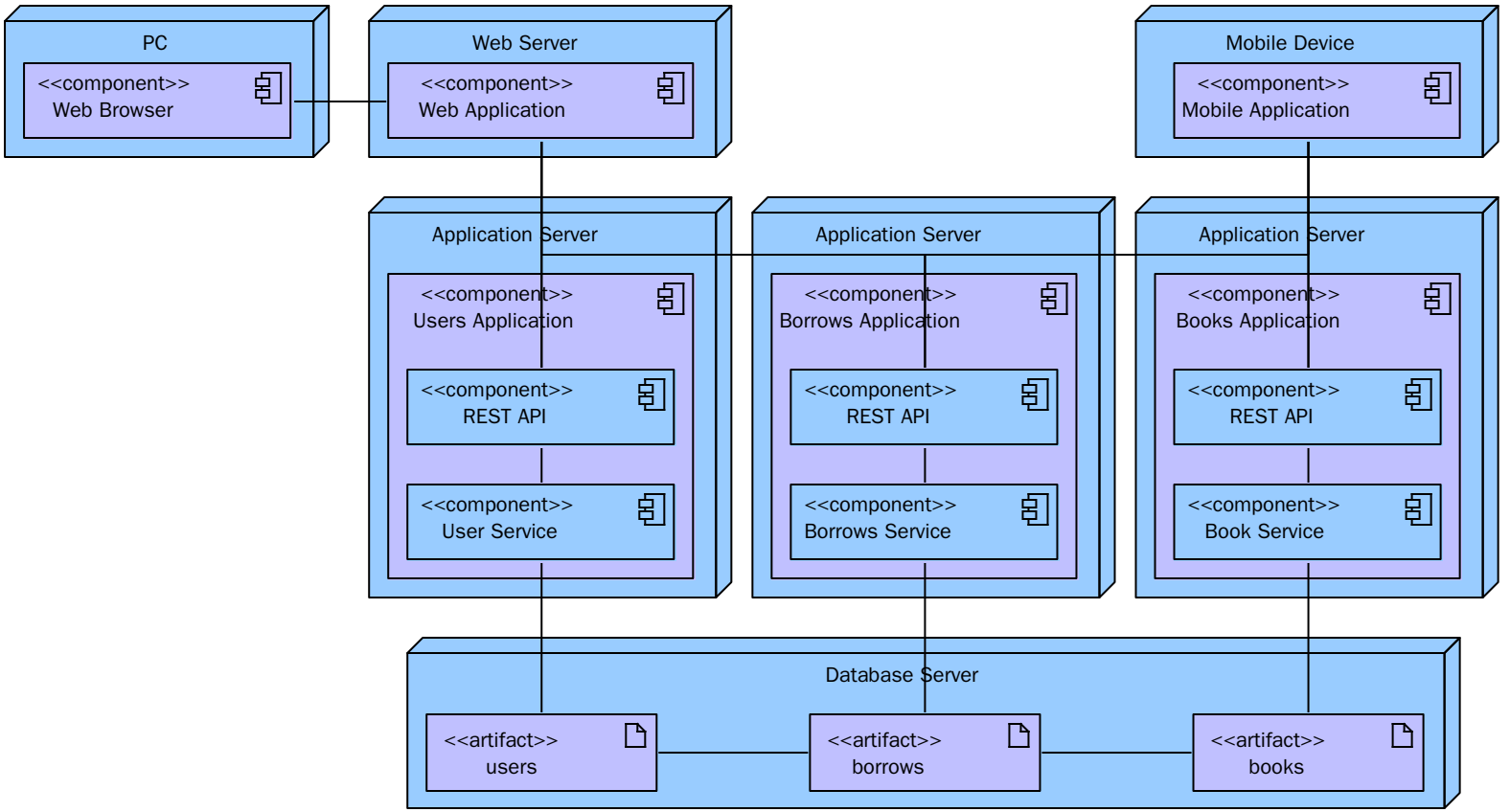
- easy configuration,
- can use communication using non-public ports,
- performance issues.

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).

Separated Application Servers

If modules are already separated we can move them to different servers.



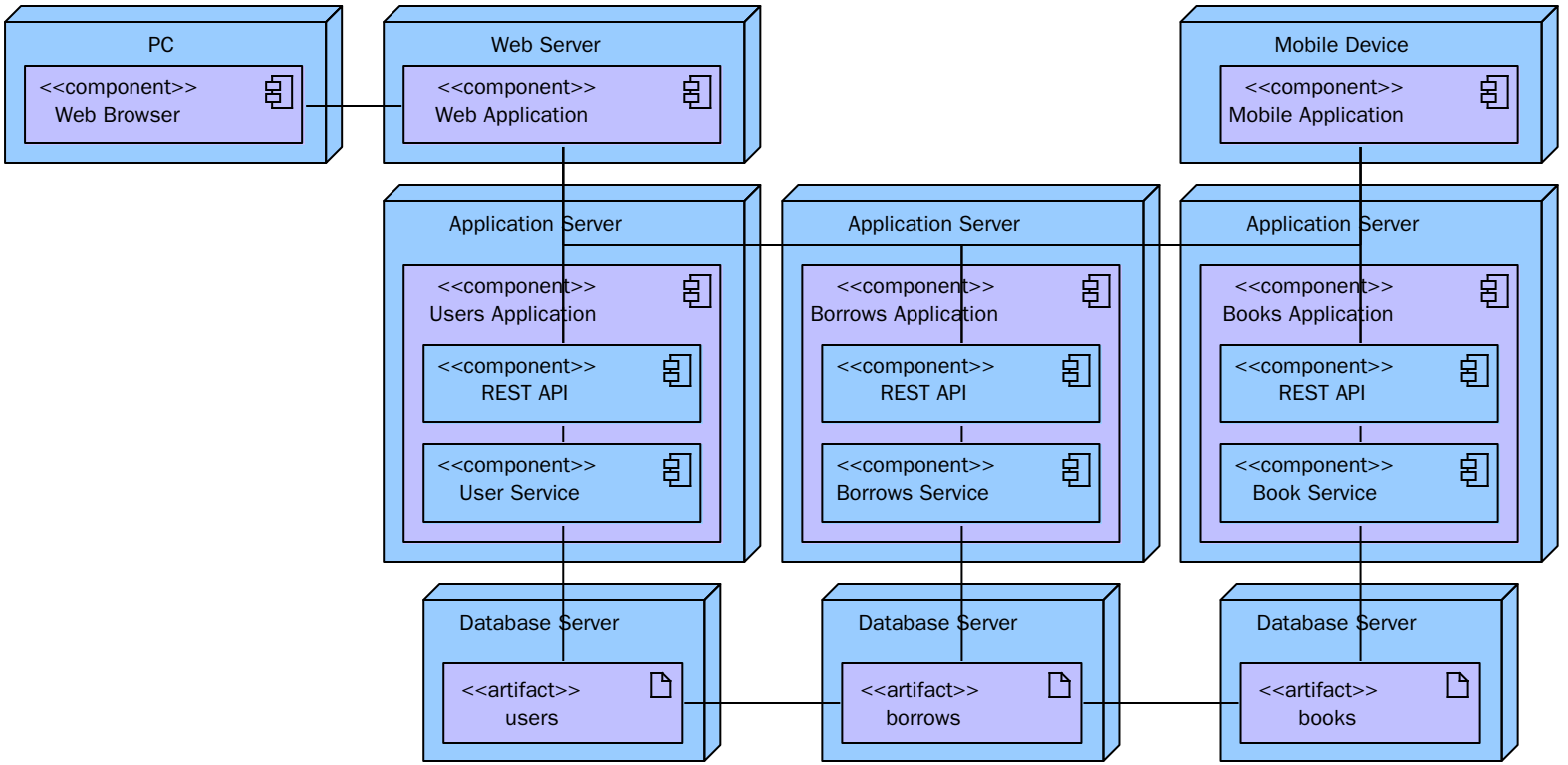
Separated Application Servers

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.

Separated Databases

If application servers are separated why not database servers?



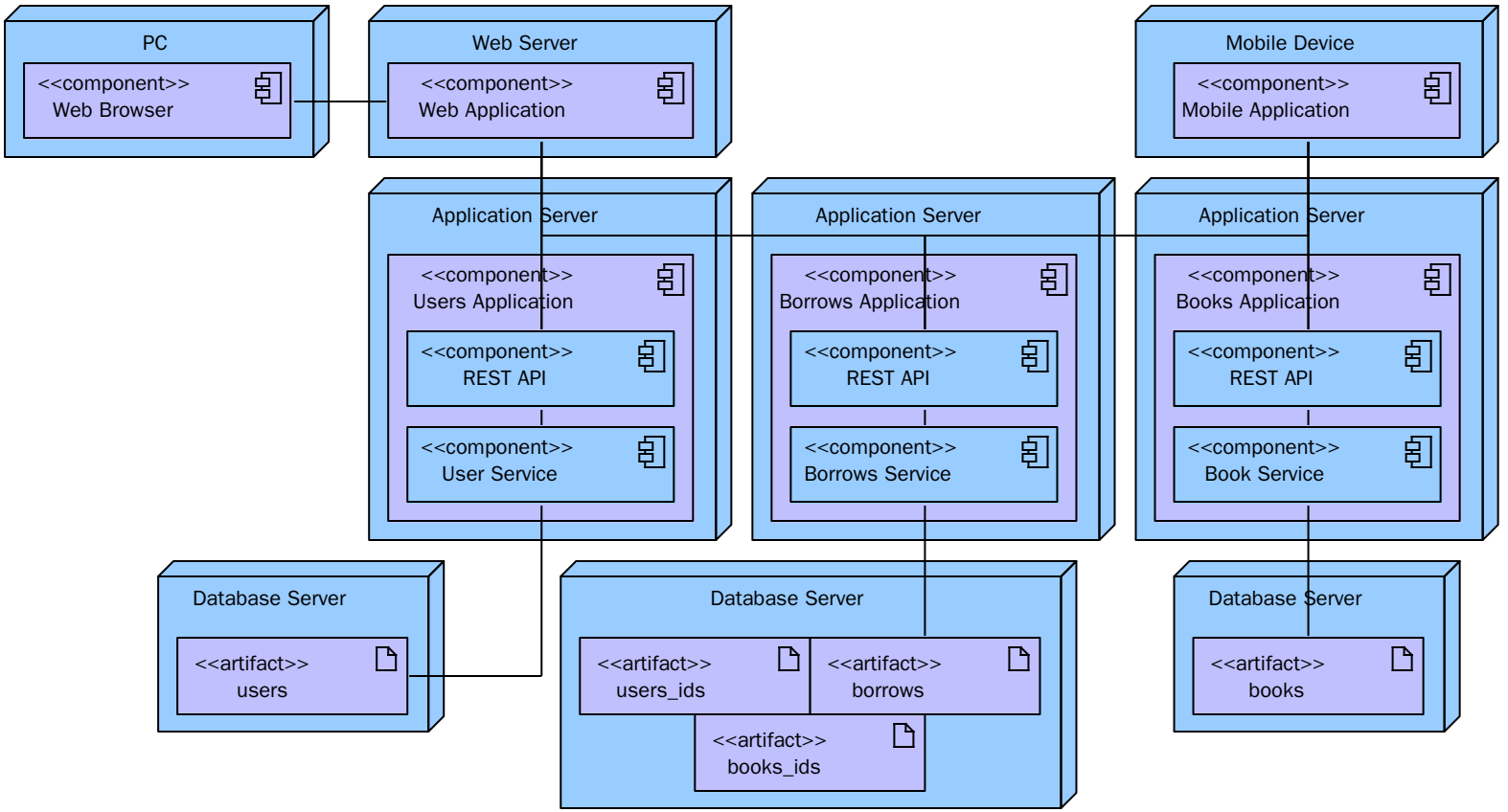
Separated Databases

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.

Duplicate Database Entries

What about relationships between data stored in different databases?



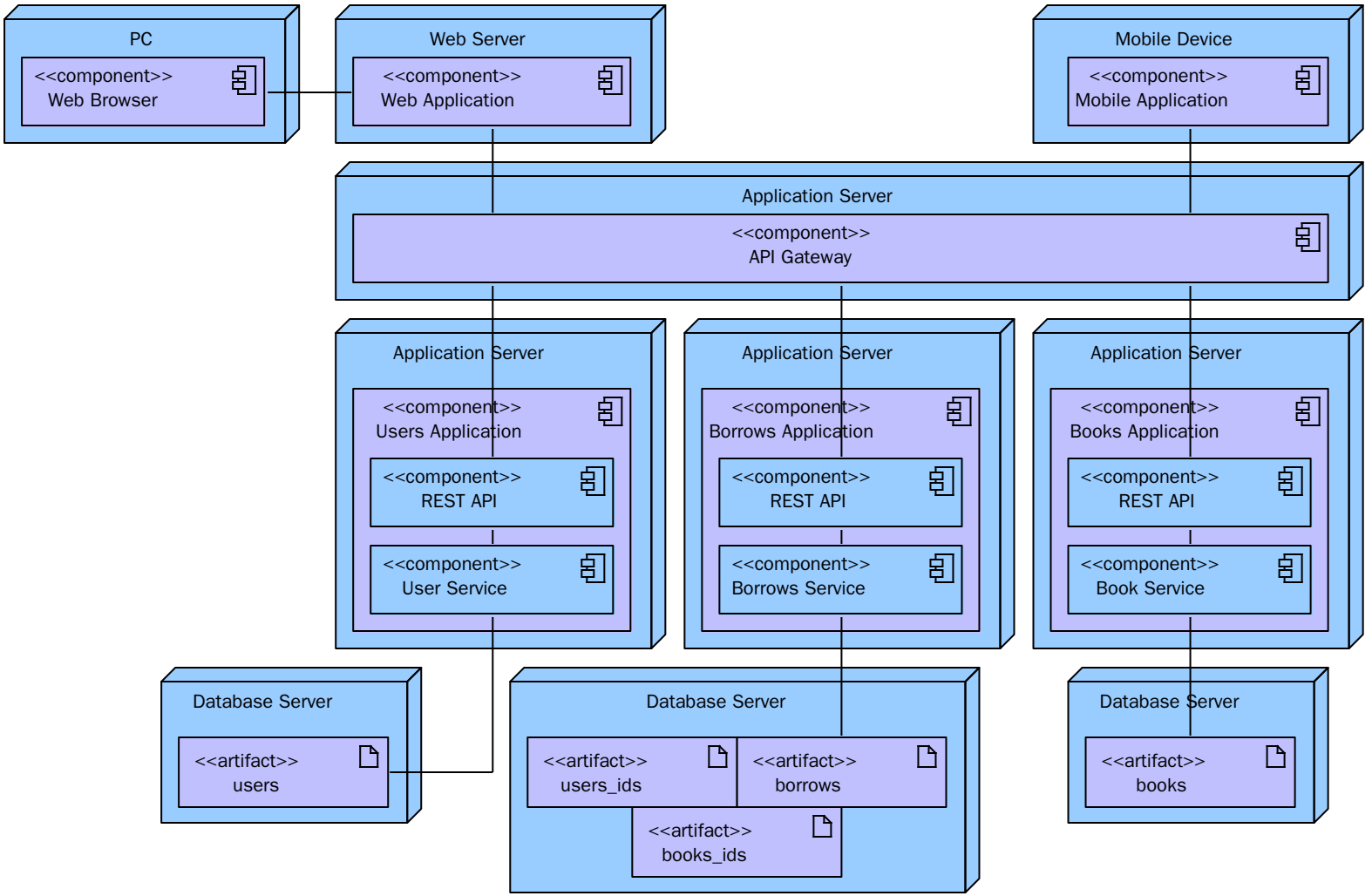
Duplicate Database Entries

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

When we have a number of web services, do we need to remember them all to use?



API Gateway

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 in Spring

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`.

Producer

Declare producer method:

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(SimpleUserRpgApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

}
```

Producer

From this moment `RestTemplate` can be injected as any other managed bean:

```
@Repository
public class RestRepository {

    @Autowired
    private RestTemplate restTemplate;

}
```

Producer

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;
```

Consuming REST

In Spring Web REST clients can be built with `RestTemplate`.

```
RestTemmplate client = new RestTemplateBuilder()
        .rootUri("http://localhost:8080/")
        .build();
```

Consuming REST

GET request:

```
Book book = restTemplate.getForObject("/api/books/{id}", Book.class, id);
```

```
Book[] books = restTemplate.getForObject("/api/books/", Book[].class);
```

PUT request:

```
restTemplate.put("/api/books/{id}", book, id);
```

POST request:

```
URI uri = restTemplate.postForLocation("/api/books/", book);
```

DELETE request:

```
restTemplate.delete("/api/books/{id}", id);
```

Gateway

Gateway:

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

Gateway

Spring Cloud provides Gateway implementation:

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

Spring Cloud

Spring Gateway comes from Spring Cloud project and in Spring Boot projects requires dependency configuration.

```
<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>
```

Spring Cloud version must be checked with Spring Boot compatiblity chart.

Gateway

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();
}
```

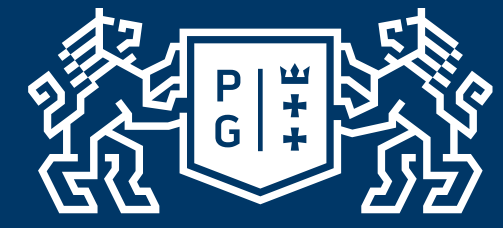
Helpful Sources

Useful articles:

- Chris Richardson, *Microservice Architecture*, https://microservices.io/.

Additional information:

- https://www.baeldung.com/spring-bean-annotations
- https://www.baeldung.com/spring-resttemplate-secure-https-service
- https://www.baeldung.com/spring-resttemplate-uri-variables-encode
- https://www.baeldung.com/spring-resttemplate-exchange-postforentity-execute
- https://www.baeldung.com/spring-rest-template-list
- https://www.baeldung.com/spring-cloud-gateway



Internet Services Architectures

JavaScript - Consuming Services

Michał Wójcik

JavaScript

JavaScript - scripting programming language:

- commonly used with web technologies,
- weak typing,
- interpreted dynamically,
- allows to manipulate DOM objects tree,
- usually executed on client side in the browser.

Script location

Different methods for adding JavaScript scripts to the HTML page:

- place it inside `<script>` tag in `<head>` or `<body>` sections,
- load external file using `<script src=""></script>`.

It's recommended to separate logic (scripts) from view (HTML document).

Adding scripts to HTML document

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

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

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

Adding scripts to HTML document

```
<html>
    <head>
        <!-- Define methods and events in external script. -->
        <script defer="defer" src="script.js"></script>
    </head>

    <body>

    </body>
</html>
```

Attribute `defer` - execute script after the document has been parsed, but before firing `DOMContentLoaded` event.

Adding scripts to HTML document

```
<html>
    <head>
        <!-- Define methods and events in external script. -->
        <script type="module" src="script.js"></script>
    </head>

    <body>

    </body>
</html>
```

Type `module` - causes the code to be treated as a JavaScript module.

A module is a Javascript file. However unlike a normal Javascript file, a module can specify which variables and functions can be accessed outside the module. Other sections of the module cannot be accessed. A module can also load other modules.

Modifying the DOM tree

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

- fetch element using:
  - `document.getElementById('id');`
  - `document.getElementsByTagName('name');`
  - `document.getElementsByClassName('name');`
  - `document.querySelectorAll('selector')`
- change element content with `element.innerHTML = 'value';`
- manage element attribute:
  - `element.attribute = 'value';`
  - `element.hasAttribute('attribute');`
  - `element.getAttribute('attribute');`
  - `element.setAttribute('attribute', 'value')`
- change CSS styles with `element.style.key = 'value';`

Modifying the DOM tree

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

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

Creating DOM elements

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.

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

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

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

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

Events handling

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

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

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

Event handling

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.

Event handling

```
<html>
    <head>
    </head>

    <body>
        <script>
            window.onload = function (e) {
                alert('woof');
            }
        </script>
    </body>
</html>
```

Event handling

```
<html>
    <head>
    </head>

    <body>
        <script>
            document.addEventListener('load', () => {

            });
        </script>
    </body>
</html>
```

Events:

- `load` - whole page has loaded, including all dependent resources such as stylesheets, scripts, iframes, and images,
- `DOMContentLoaded` - as soon as the page DOM has been loaded, without waiting for resources to finish loading.

AJAX and JavaScript

AJAX - Asynchronous JavaScript and XML:

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

AJAX and JavaScript

```
let xhttp = new XMLHttpRequest();

xhttp.onreadystatechange = function () {
    if (this.readyState == 4 && this.status == 200) {
        let object = JSON.parse(this.responseText);
    }
};

xhttp.open("GET", "http://localhost:8080/api/wolves", true);
xhttp.send();
```

AJAX and JavaScript

```
fetch('http://localhost:8080/api/wolves').then(response => {
    if (response.ok) {
        return response.json();
    }
    throw new Error("Network error.");
}).then(response => {
    console.log(response);
});
```

Sending POST with JavaScript

```
let xhttp = new XMLHttpRequest();
let wolf = {name: "green", age: 9};

xhttp.open("POST", "http://localhost:8090/api/wolves", true);
xhttp.setRequestHeader("Content-type", "application/json");
xhttp.send(JSON.stringify(wolf));
```

Sending POST with JavaScript

```
let init = {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify(wolf)
}

fetch('http://localhost:8090/api/wolves', init).then(response => {
    if (response.ok) {
        return;
    }
    throw new Error("Network error.");
}).then(() => {
    console.log('done');
});
```

Sending DELETE with JavaScript

```
let xhttp = new XMLHttpRequest();

xhttp.open("DELETE", "http://localhost:8090/api/wolves/0", true);
xhttp.send();
```

Sending DELETE with JavaScript

```
let init = {
  method: 'DELETE'
}

fetch('http://localhost:8090/api/wolves/0', init).then(response => {
    if (response.ok) {
        return;
    }
    throw new Error("Network error.");
}).then(() => {
    console.log('done');
});
```

SOP vs CORS

SOP - Same Origin Policy:

- origin (protocol, host and port) defines web application,
- different applications can not exchange content:
  - images linking,
  - sending forms to different server;
- secures against XSS (Cross-site scripting) and CSRF (Cross-site request forgery) attacks,
- default browser behavior,
- blocks AJAX requests between applications.

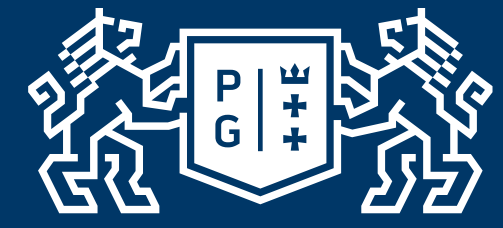
CORS - Cross-Origin Resource Sharing:

- different applications can exchange content,
- defines using headers on the server side,
- allows for:
  - JavaScript's applications to communicate with API on different servers,
  - SSO (Single Sing On) systems for a number of applications.

Helpful resources

Helpful resources:

- Mozilla, *JavaScript*, https://developer.mozilla.org/en-US/docs/Web/JavaScript.
- w3schools, *JavaScript Tutorial*, https://www.w3schools.com/js/.



Internet Services Architectures

TypeScript

Michał Wójcik

TypeScript

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, e.g. ECMAScript 5:
  - at the developer's choice: `tsc --target`,
  - the problem of the standard version and the version of browsers.

Basic use

Tool installation:

```
npm install -g typescript
```

File transpilation:

```
tsc file.ts
```

Monitoring file changes:

```
tsc --watch file.ts
```

Modern IDEs offer tool integration.

Online playground https://www.typescriptlang.org/play

Types - basic

Basic types in TypeScript.

```
let done: boolean = false;

let age: number = 42;

let color: number = 0xf00dcc;

let username: string = 'ookami';

let list1: number[] = [42, 36, 28];

let list2: Array<number> = [27, 45, 19];
```

Types - enums and tuples

TypeScript allows for enumerations and tuples.

```
let x: [string, number] = ['ookami', 24];

enum Color {Red, Green, Blue};

let c: Color = Color.Green;
```

Types - special

Variable can be defined with `any` type.

```
let variable: any = 42;
variable = 'ookami';
variable = false;
```

Special types:

- `null` - conveys that variable is empty,
- `undefined` - conveys that variable does not exist.

In non-strict mode every variable can be `null` or `undefined`. In strict mode `null` and `undefined` are treated as separate types.

```
let variable: string = '';
variable = undefined;       //error
variable = null;            //error
```

```
let variable: string|undefined = '';
variable = undefined;
variable = null;            //error
```

```
let variable: string|undefined|null = '';
variable = undefined;
variable = null;
```

Function parameters

Strong typed function parameters.

```
function greeter(username: string): string {
    return 'Hello, ' + username;
}
```

Optional function parameters (always after required parameters).

```
function greeter(username?: string): string {
    if (username) {
        return 'Hello, ' + username;
    } else {
        return 'Hello!';
    }
}
```

Default function parameters, anywhere (can be skipped with `undefined`).

```
function greeter(user: string = 'world'): string {
    return 'Hello, ' + user;
}
```

Variable number of arguments

```
function greeter(firstName: string, ...otherNames: string[]): string {
    return 'Hello, ' + username + ' ' + otherNames.join(', ');
}

greeter('world', 'ookami', 'kitsune');
```

Classes

```
class Greeter {

    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    greet(): string {
        return 'Hello, ' + this.greeting;
    }

}

let greeter = new Greeter('world');
let greetings = greeter.greet();
```

TypeScript - classes

```
class Animal {

    name: string;

    constructor(name: string) {
        this.name = name;
    }

    move(distanceInMeters: number) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }

}
```

```
class Snake extends Animal {

    constructor(name: string) {
        super(name);
    }

    move(distanceInMeters = 5) {
        console.log('Slithering...');
        super.move(distanceInMeters);
    }

}
```

TypeScript - visibility levels

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;
    }

}
```

TypeScript - abstract classes

```
abstract class Animal {

    abstract makeSound(): void;

    move(): void {
        console.log("move");
    }

}
```

TypeScript - interfaces

```
interface Moveable {

    move(distanceInMeters: number): void;

}
```

```
class Snake implements Moveable {

    move(distanceInMeters = 5) {
        console.log("Slithering...");
    }

}
```

TypeScript - interfaces

```
interface Person {

    firstName: string;

    lastName: string;

    get email(): string;

}
```

```
class Student implements Person {

    firstName: string;

    lastName: string;

    constructor(firstName: string, lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    get email(): string {
        return this.firstName
            + "." + this.lastName
            + "@student.pg.edu.pl";
    }

}
```

```
let p1: Person = new Student('Wolf', 'Bad');
```

Helpful resources

Helpful resources:

- Microsoft, *TypeScript Documentation*, https://www.typescriptlang.org/docs/.
- w3schools, *TypeSript Tutorial*, https://www.w3schools.com/typescript/.

Internet Services Architectures

Angular

Michał Wójcik

Web frameworks

There is a large number of different solutions for web frameworks executed on client side in web browser:

- Angular,
- React,
- Vue.js,
- Ember.js,
- Meteor,
- Mithril,
- Node.js,
- Polymer,
- Aurelia,
- Backbone.js.

Angular Framework

Angular framework versions:

- Angular 1.x - 2009,
- Angular 2.x - 2016,
  - rewritten from scratch,
  - TypeScript instead of JavaScript,
  - version 2.x is closer to React than to Angular 1.x:
  - migration 1.x → 2.x requires a lot of code changes;
- Angular 4, 5, 6, 7, 8, 9, ..., 14, 15, 16:
  - semantic versioning,
  - version 3.x skipped due to discrepancy in the versions of individual components (router package),
  - Google aims to release new version every half year,
  - 18 months of support.

Semantic versioning

major.minor.patch

2.7.3

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.

Framework versions and application

If we want access to **new features and security patches**, we need to **migrate** our application to newer versions of the framework:

- patch, minor - backward compatible changes:
  - application code does not need to be changed in order to work,
- major - changes in code are required in order to work:
  - requires working hours,
  - doesn't mean that changes will be painful for developers,
  - Angular's authors do not plan to rewrite the framework from scratch again;
- each subsequent release introduces new changes:
  - if we give up migration later it will only be more difficult.

Angular - basic usage

Tool installation:

```
npm install -g @angular/cli
```

Creating new project:

```
ng new project-name --routing --style=css
```

Starting application (open in default browser):

```
ng serve --open
```

Project structure

Sources:

```
src\                (source files for the root-level application project)
    app\            (component files which application logic and data)
    assets\         (images and other static assets)
    favicon.ico     (icon for bookmark)
    index.html      (main page)
    main.ts         (application entry point)
    styles.css      (global styles)
```

Other files:

```
.editorconfig       (config for editors)
.gitignore          (git ignore configuration)
angular.json        (CLI config)
package.json        (list of npm packages dependencies)
package-lock.json   (list of resolved npm packages versions)
README.md           (just a readme)
tsconfig.app.json   (project specific TypeScript config)
tsconfig.json       (default TypeScript config)
tsconfig.spec.json  (TypeScript config for tests)
```

Modules

Modules:

- Angular applications are divided into modules corresponding to particular functionalities,
- each application has a main module called `AppModule` by default,
- 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.

NgModule - metadata

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 components 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 in main module.

NgModule - example

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
    declarations: [
        AppComponent
    ],
    imports: [
        BrowserModule
    ],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule {
}
```

Components

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.

```
<!doctype html>
<html lang="en">
    <head>
    </head>
    <body>
        <app-root></app-root>
    </body>
</html>
```

Components

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 component.

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

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

Component - example

```
import { Component, Input, OnInit } from '@angular/core';

@Component({
    selector: 'app-hello',
    templateUrl: './hello.component.html',
    styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

    @Input()
    name: string;

    value: string;

    constructor() {
    }

    ngOnInit() {
        this.value = 'Hello ' + this.name + '!';
    }

}
```

```
<app-hello [name]="'world'"></app-hello>
```

Component - declaration

Component must be declared in module.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { HelloComponent } from './hello/hello.component';

@NgModule({
    declarations: [
        AppComponent,
        HelloComponent // !
    ],
    imports: [
        BrowserModule
    ],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule {
}
```

Templates

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.

```
<div *ngIf="value">
  {{value}}
</div>
```

Data binding

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

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

Interpolation

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

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

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

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

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>
```

Event binding

```
import { Component, Input, OnInit } from '@angular/core';

@Component({
    selector: 'app-hello',
    templateUrl: './hello.component.html',
    styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

    constructor() {
    }

    ngOnInit() {
    }

    onSave(): void {

    }

}
```

Bidirectional binding

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">
```

Component - example

```
import { Component, Input, OnInit } from '@angular/core';

@Component({
    selector: 'app-hello',
    templateUrl: './hello.component.html',
    styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

    @Output()
    helloEvent: EventEmitter<string> = new EventEmitter<string>();

    constructor() {
    }

    ngOnInit() {
    }

    emit(): void {
        this.helloEvent.emit('hello');
    }

}
```

Component - example

```
<app-hello (helloEvent)="onHelloAction($event)"></app-hello>
```

Directives

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

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.

Service - example

```
import { Injectable } from '@angular/core';

@Injectable()
export class HelloService {

    constructor() {
    }

}
```

Services can be injected into other services or components.

```
import { Component, Inject, Input, OnInit } from '@angular/core';
import { HelloService } from '../hello.service';

@Component({
    selector: 'app-hello',
    templateUrl: './hello.component.html',
    styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

    constructor(private service: HelloService) {
    }

    ngOnInit() {
    }

}
```

Service - example

Service provider must be declared in module.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { HelloComponent } from './hello/hello.component';
import { HelloService } from './hello.service';

@NgModule({
    declarations: [
        AppComponent,
        HelloComponent
    ],
    imports: [
        BrowserModule
    ],
    providers: [HelloService], // !
    bootstrap: [AppComponent]
})
export class AppModule {
}
```

Routing

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.

Routing - example

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HelloComponent } from './hello/hello.component';

const routes: Routes = [
    {path: 'hello', component: HelloComponent},
    {path: 'hello/:name', component: HelloComponent}
];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
})
export class AppRoutingModule {
}
```

```
<body>
    <section>
        <router-outlet></router-outlet>
    </section>
</body>
```

The component defined in routing will be displayed below: `<router-outlet></router-outlet>`.

Routing - example

Routing module must be imported into our module.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { HelloComponent } from './hello/hello.component';
import { HelloService } from './hello.service';
import { AppRoutingModule } from './app-routing.module'; // !

@NgModule({
    declarations: [
        AppComponent,
        HelloComponent
    ],
    imports: [
        BrowserModule,
        AppRoutingModule // !
    ],
    providers: [HelloService],
    bootstrap: [AppComponent]
})
export class AppModule {
}
```

Navigation

```
<nav class="navbar">
    <ul>
        <li>
            <a routerLink="/hello">Hello</a>
        </li>
    </ul>
</nav>
<section>
<router-outlet></router-outlet>
</section>
```

Navigation - programmatic

```
import { Component, Input, OnInit } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';

@Component({
    selector: 'app-hello',
    templateUrl: './hello.component.html',
    styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

    value: string;

    constructor(private route: ActivatedRoute, private router: Router) { // !
    }

    back() {
        this.router.navigateByUrl('/hello'); // !
    }
}
```

Navigation with parameter

```
<nav class="navbar">
    <ul>
        <li>
            <a routerLink="/hello">Hello</a>
        </li>
        <li>
            <a
            [routerLink]="['/hello', 'ookami']">Hello ookami
        </a>
    </li>
</ul>
        </nav>
<section>
<router-outlet></router-outlet>
</section>
```

Navigation with parameter - programmatic

```
import { Component, Input, OnInit } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';

@Component({
    selector: 'app-hello',
    templateUrl: './hello.component.html',
    styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

    value: string;

    constructor(private route: ActivatedRoute, private router: Router) { // !
    }

    back() {
        this.router.navigateByUrl('/hello');
    }

    refresh() {
        this.router.navigate(['hello', 'ookami']); // !
    }
}
```

Handling navigation parameters

```
import { Component, Input, OnInit } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';

@Component({
    selector: 'app-hello',
    templateUrl: './hello.component.html',
    styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

    value: string;

    constructor(private route: ActivatedRoute, private router: Router) { // !
    }

    ngOnInit() {
        const id = this.route.snapshot.paramMap.get('name'); // !
        if (id == null) {
            this.value = 'Hello!';
        } else {
            this.value = 'Hello ' + id + '!';
        }
    }

}
```

Calling web services

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.

Calling web service - example

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { HelloComponent } from './hello/hello.component';
import { HelloService } from './hello.service';
import { AppRoutingModule } from './app-routing.module';
import { HttpClientModule } from '@angular/common/http'; // !

@NgModule({
    declarations: [
        AppComponent,
        HelloComponent
    ],
    imports: [
        BrowserModule,
        AppRoutingModule,
        HttpClientModule // !
    ],
    providers: [HelloService],
    bootstrap: [AppComponent]
})
export class AppModule {
}
```

Calling web service - example

To avoid setting the service address permanently, it is worth using the `proxy.conf.json` file:

```
{
  "/api": {
    "target": "http://localhost:8080/library/",
    "secure": false
  }
}
```

Calling web service - example

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Book } from '../model/book';
import { Author } from '../model/author';

@Injectable()
export class BookService {

    constructor(private http: HttpClient) {
    }

    findAllBooks(): Observable<Book[]> {
        return this.http.get<Book[]>('api/books');
    }

}
```

ng commands

Instead of creating individual elements manually, you can use the appropriate commands.

Create a regular class (model):

```
ng generate class model/book
```

Creation of the enum type:

```
ng generate enum model/cover
```

Component creation:

```
ng generate component component/BookList
```

Service creation:

```
ng generate service service/book
```

ng commands

Creation of a routing module:

```
ng generate module app-routing --flat --module=app
```

Starting the server:

```
ng serve
```

Starting the server with the use of a proxy file:

```
ng serve --proxy-config proxy.conf.json
```

Download dependencies in a project without `node_modules`:

```
npm install
```

ng commands

It is a good idea to put the proxy configuration in the `package.json` file, which is used by e.g. IntelliJ:

```
{
  "name": "angular",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve --proxy-config proxy.conf.json",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  }
}
```

Helpful resources

Helpful resources:

- Google, *Angular Documentation*, https://angular.io/docs.
- Google, *Angular Tutorials*, https://angular.io/tutorial.
- w3schools, *TypeSript Tutorial*, https://www.w3schools.com/typescript/.

Internet Services Architectures

Docker

Michał Wójcik

Motivation

Why deploying applications is troublesome:

- application needs to be configured (e.g. database connection, file storage, ports, etc.),
- application requires runtime environment:
  - Java Virtual Machine for Java based applications,
  - Application Server for web applications,
  - HTTP server for static web applications;
- runtime environment must be configured,
- additional system libraries can be required.

Why developing applications is troublesome:

- different database severs or version for different projects,
- projects build with multiple modules required to be running for development,
- projects using legacy tools unable to be installed in developer's operating system.

Containers

Virtual Machine - virtualize an entire machine down to the hardware layers and containers only virtualize software layers above the operating system level.

Containers:

- in some way similar to virtual machines,
- lightweight software packages that contain all the dependencies required to execute the contained software application:
  - system libraries,
  - external third-party code packages,
  - other operating system level applications.

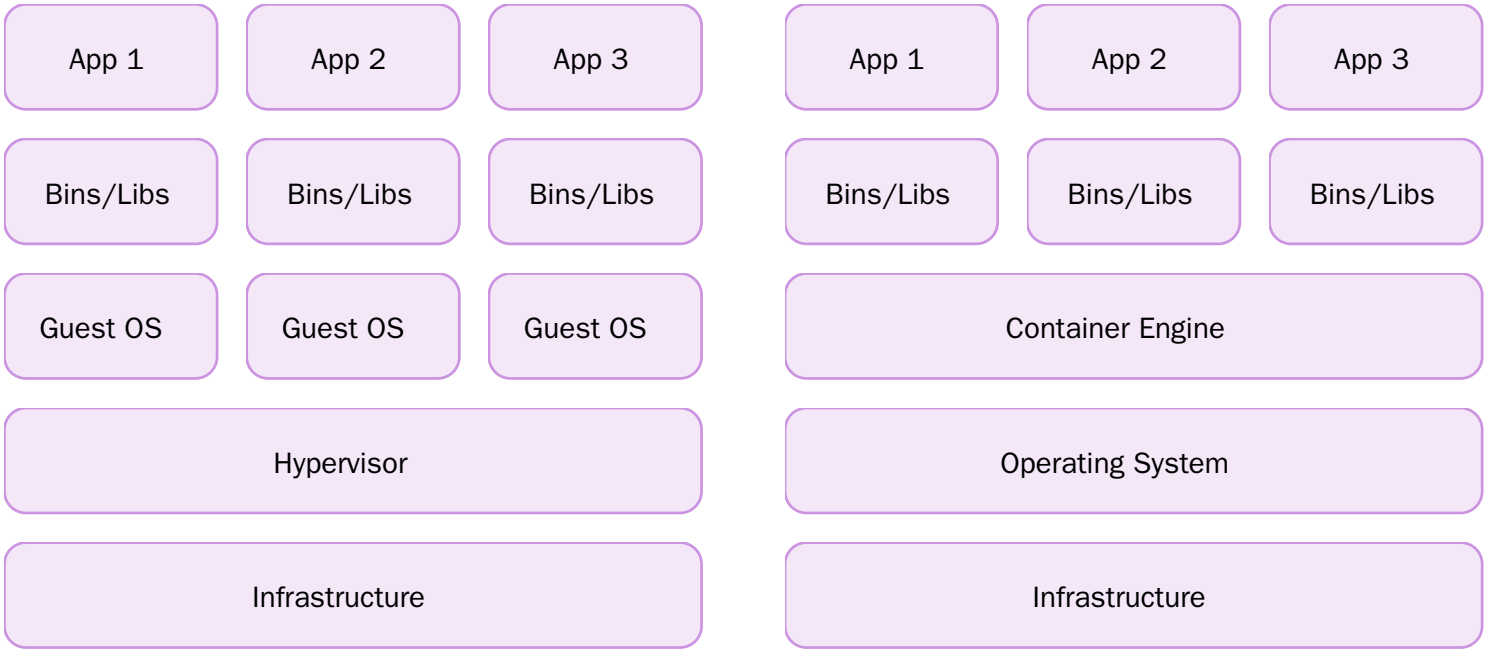
Containers vs Virtual Machines

Containers vs Virtual machines:

- **Containers:**
  - kernel shared with the host,
  - no hardware isolation,
  - small sizes,
  - containers based on the same image share its content,
  - very small performance overhead,
  - can run within minutes;
- **Virtual Machines:**
  - separate system,
  - hi level hardware separation (Intel VT-x, AMD-V),
  - big size,
  - performance problems,
  - can take some time to start.

POLITECHNIKA GDAŃSKA

Containers vs Virtual Machines



Containers vs Virtual Machines

Containers:

- **pros:**
  - fast to modify and iterate on,
  - pre-made containers in public repositories;
- **cons:**
  - shared host exploits.

Virtual Machines:

- **pros:**
  - full isolation security,
  - interactive development;
- **cons:**
  - iteration speed,
  - storage size cost.

Containers - usage

Usage for **containers**:

- production environment,
- development environment,
- microservices,
- applications abstraction from physical infrastructure,
- solution for incompatible requirements and dependencies.

Docker

Docker - example for container platform:

- allows to create images,
- allows to run containers based on created images,
- containers run applications,
- allows to create networks, so containers can communicate.

Container

Container:

- encapsulated environment that runs applications,
- lightweight alternative for virtual machine,
- doesn't have own kernel – uses hosts kernel,
- containers can communicate with each others via network,
- has its own filesystem,
- has its own set of ports which can be published in host.

Image

Image:

- contains application and all dependencies/resources required,
- ready to run application independently of operating system, filesystem, etc.,

Image:

- **pros**:
  - portable,
  - light in comparison to virtualization,
  - extendable,
  - easy to configure and run,
  - readonly;
- **cons**:
  - size in comparison to native application,
  - more resources use in comparison to native applications.

Docker CLI

Docker CLI provides hierarchical commands.

`docker` - main command:

```
docker
```

`pull` - pull image from repository:

```
docker pull hello-world
```

`run` - runs container:

```
docker run hello-world
```

```
docker run nginx
```

Docker CLI - images

`image` - images management:

```
docker image
```

`pull` - pull image from repository:

```
docker image pull hello-world
```

`ls` - list images:

```
docker image ls
```

`rm` - remove image:

```
docker image rm hello-world
```

Docker CLI - volumes

`volume` - volumes management:

```
docker volume
```

`ls` - list volumes:

```
docker volume ls
```

`create` - create a volume:

```
docker volume create woof
```

`rm` - remove volume:

```
docker volume rm woof
```

`prune` - remove unused volumes:

```
docker volume prune
```

Docker CLI - containers

`container` - containers management:

```
docker container
```

`run` - runs container:

```
docker container run hello-world
```

`ls` - list containers:

```
docker container ls
```

`attach` - attach input, output and error streams:

```
docker container attach laughing_wright
```

Docker CLI - containers

`stop` - stop container:

```
docker container stop laughing_wright
```

`start` - start container:

```
docker container start laughing_wright
```

`rm` - remove container:

```
docker container rm laughing_wright
```

`prune` - remove all stopped containers:

```
docker container prune
```

Running a container

Running container:

- can be removed when finished,
- can be started in background,
- can be named,
- it's good to provide exact version, `lates` can surprise.

```
docker container run --rm -d --name nginx-isa nginx:1.23.3
```

Running a container

Running command in container:

```
docker container exec nginx-isa ls /
```

Running interactive command in container

```
docker container exec -it nginx-isa bash
```

Docker CLI - networking

`netowork` - network management:

```
docker network
```

`ls` - list networks:

```
docker network ls
```

`create` - create network:

```
docker network create isa-network
```

`rm` - remove network:

```
docker network rm isa-network
```

Networking

```
docker run --rm -dit --name alpine1 alpine ash
```

```
docker run --rm -dit --name alpine2 alpine ash
```

```
docker exec alpine1 ping alpine2
```

```
docker network create alpine-net
```

```
docker run --rm -dit --name alpine3 --network alpine-net alpine ash
```

```
docker run --rm -dit --name alpine4 --network alpine-net alpine ash
```

```
docker exec alpine3 ping alpine4
```

```
docker stop alpine1 alpine2 alpine3 alpine4
```

```
docker network rm alpine-net
```

Dockerfile

`Dockefile` - is a recipe how to build image:

- define base image,
- define exposed ports,
- define volumes,
- execute commands (e.g. install dependencies),
- copy files (e.g. application).

Dockerfile

`Dockerfile` is made with a sequence of commands:

- `FROM` - Specify the base image for your Docker image. It is the starting point for your image.
- `WORKDIR` - Set the working directory within the container for subsequent commands.
- `COPY` - Copy files or directories from your host machine into the container.
- `ADD` - Similar to `COPY`, but can also fetch files from remote URLs and unpack compressed files.
- `RUN` - Execute commands within the container during the build process. Used for installing software or configuring the environment.
- `CMD` - Define the default command to run when a container is started from the image. This can be overridden at runtime.
- `ENTRYPOINT` - Specify the default executable when the container is run. It is more commonly used for defining the primary command, which cannot be overridden.
- `EXPOSE` - Document which network ports the container listens on at runtime, but it doesn't actually publish the ports.
- `ENV` - Set environment variables within the container, which can be used by the processes running in the container.

Dockerfile

`Dockerfile` is made with a sequence of commands:

- `ARG` - Define build-time arguments that can be passed to the Dockerfile during the `docker build` command.
- `LABEL` - Add metadata to the image, such as version information or maintainer details.
- `VOLUME` - Create a mount point for externally defined volumes to be used by the container.
- `USER` - Set the user or UID that the container process should run as, enhancing security.
- `HEALTHCHECK` - Define a command to periodically check the health of the container.

Standard description

To provide unified manner for describing containers, The OpenContainers Image Spec can be used: https://specs.opencontainers.org/image-spec/?v=v1.0.1

Some of the label values:

- org.opencontainers.image.title,
- org.opencontainers.image.authors,
- org.opencontainers.image.source,
- org.opencontainers.image.url,
- org.opencontainers.image.vendor,
- org.opencontainers.image.version,
- org.opencontainers.image.description,
- org.opencontainers.image.licenses.

Dockerfile

```
FROM nginx:1.23.3

EXPOSE 80

VOLUME /usr/share/nginx/html/custom

RUN echo '<html>' > /usr/share/nginx/html/index.html
RUN echo '  <head>' >> /usr/share/nginx/html/index.html
RUN echo '    <link rel="stylesheet" type="text/css" href="/custom/custom.css">' \
>> /usr/share/nginx/html/index.html
RUN echo '  </head>' >> /usr/share/nginx/html/index.html
RUN echo '  <body>' >> /usr/share/nginx/html/index.html
RUN echo '    <h1>Welcome to My Custom Nginx Page</h1>' >> /usr/share/nginx/html/index.html
RUN echo '    <p>This is a custom Nginx web page.</p>' >> /usr/share/nginx/html/index.html
RUN echo '  </body>' >> /usr/share/nginx/html/index.html
RUN echo '</html>' >> /usr/share/nginx/html/index.html
```

```
docker build -t isa-nginx:1.0.0 .
```

```
docker run --rm -p 8080:80 -v $(pwd)/custom:/usr/share/nginx/html/custom isa-nginx:1.0.0
```

Docker Compose

Docker Compose - tool for configuring and starting multiple container applications:

- configuration in `yaml` format,
- automatic network connection,
- simple commands to start/stop everything,
- wrapper for docker commands.

Docker Compose

```
version: '3'

services:
  nginx:
    image: custom-nginx:1.0.0
    ports:
      - "8080:80"
    volumes:
      - /tmp/custom:/usr/share/nginx/html/custom
```

Container storage

Dynamic data should not be stored inside container (those need to be easily replaceable):

- **bind mounts**:
  - allow to mount a file or directory from the host machine directly into a container,
  - be careful about rights,
  - changes can be done from container and host;
- **named volumes**:
  - provide a named reference to a directory on the host system where the data is stored,
  - Docker-managed way of creating persistent storage,
  - data in named volumes persists even if the associated container is removed;
- **anonymous volumes**:
  - automatically created by Docker and associated with a container,
  - don't have a user-defined name and are often used for temporary or cache storage,
  - automatically deleted when the container is removed.

Docker Compose

```yaml
version: '3'

services:
  mysql:
    image: mysql:latest
    environment:
      MYSQL_ROOT_PASSWORD: root_password
      MYSQL_DATABASE: database_name
      MYSQL_USER: database_user
      MYSQL_PASSWORD: database_password
    volumes:
      - mysql-data:/var/lib/mysql
    ports:
      - "3306:3306"

volumes:
  mysql-data:
```

Docker Compose CLI

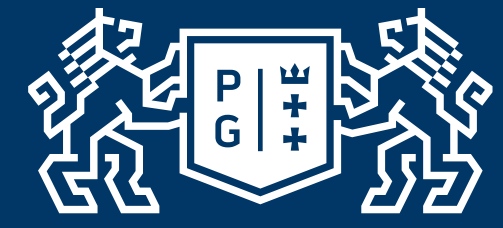
Selected commands:

- `docker-compose up` - creates and starts services defined in a `docker-compose.yml` file,
- `docker-compose down` - stop and remove containers and networks,
- `docker-compose ps` - list the status of services defined in the `docker-compose.yml` file,
- `docker-compose logs` - display logs for services defined in the compose file.

Helpful resources

Helpful resources:

- Docker Inc., *Docker Docs*, https://docs.docker.com/.
- Baeldung, *Docker Guide*, https://www.baeldung.com/ops/docker-guide.



Internet Services Architectures

Microservices - Discovery

Michał Wójcik

Services location

Service provider:

- same service can be deployed at different localizations,
- localization can change dynamically (e.g.: cluster nodes migration),
- there can be multiple instance of the same service.

Service consumer:

- needs to know service provider localization,
- needs to be aware of localization changes,
- needs to choose which instance use.

Discovery

Discovery:

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

Discovery - solutions

There are multiple solutions for discovery services:

- **Consul** by HashiCorp - standalone catalog with service discovery and health checking,
- **Eureka** by Netflix - discovery server integrated into Spring Boot applications,
- **Zookeeper** by Apache - distributed key value store which can be used for service discovery,
- **etcd** - distributed key value store which can be used for service discovery,
- **Kubernetes** - container orchestration solution with built-in service discovery.

Consul

Consul can be started with Docker command:

```
docker run --rm --name consul -p 8500:8500 consul:1.10.3
```

Consul

In client module appropriate dependency is required:

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

and configuration in `application.properties`:

```
spring.cloud.consul.host=localhost
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.instance-id=1
spring.cloud.consul.discovery.service-name=library
```

Eureka

Eureka can be started as Spring Boot application:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }

}
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Eureka

In client module appropriate dependency is required:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

and configuration in `application.properties`:

```
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
eureka.instance.appname=Library
eureka.instance.instance-id=1
eureka.instance.lease-expiration-duration-in-seconds=5
eureka.instance.lease-renewal-interval-in-seconds=2
```

Discovery client

Spring discovery client is independent of implementation (Consul, Eureka).

Enable client on main class (not required in newer versions):

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

Discovery client

Inject discovery client into component:

```
@Repository
public class RestRepository {

    @Autowired
    private DiscoveryClient discoveryClient;

    @Autowired
    private RestTemplate restTemplate;

    public void delete() {
            URI uri = discoveryClient.getInstances("library")
                .stream()
                .findAny()
                .orElseThrow()
                .getUri();
            restTemplate.delete(uri + "/" + id);
    }
}
```

POLITECHNIKA GDAŃSKA

Helpful resources

Helpful resources:

- https://www.baeldung.com/cs/service-discovery-microservices
- https://www.baeldung.com/spring-cloud-consul
- https://www.baeldung.com/spring-cloud-netflix-eureka
- https://www.baeldung.com/eureka-self-preservation-renewal
- https://www.baeldung.com/spring-cloud-custom-gateway-filters#bd-1-checking-and-modifying-the-request

Internet Services Architectures

Microservices - Load Balancer

Michał Wójcik

Load balancer

Why use **load balancer**:

- services can be overwhelmed with requests,
- there can be multiple instances of the same service,
- there can be service discovery,
- client selecting always first instance from list is not a sulution.

Web server example

Web server example:

- load balancer in gateway proxy,
- all client requests go through gateway,
- multiple instance of web server serving the same content,
- gateway forwards client request to selected server,
- gateway returns response to client,
- gateway with load balancer is the only public endpoint to access web servers.

Routing algorithms

Routing algorithms:

- **round-robin** - sends subsequent requests in a cyclic loop to all servers,
- **least connections** - load balancer keeps track of the number of active connections to each server, sends new requests to servers with the fewest active connections,
- **weighted** round-robin/least connections - different weight for servers (e.g.: different computational power),
- **random** - simple but does not guarantee even,
- **chained failover** - predefined chain of servers, all requests are sent to a server until it can't serve mor requests,
- **data ranges** - different servers are responsible for processing different data ranges (requires domain context).

Load balancer types

Load balancer:

- **Layer 4** - operate at the transport layer of the OSI model and are responsible for distributing incoming traffic based on the source and destination IP addresses and port numbers,
- **Layer 7** - operate at the application layer of the OSI model. They are responsible for distributing incoming traffic based on the content of the request.

Load balancer

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

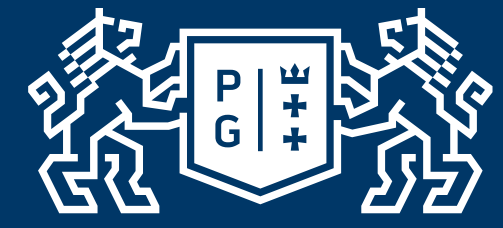
Load balancer can be also used automatically in Gateway:

```
@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();
}
```

Helpful resources

Helpful resources:

- https://www.baeldung.com/cs/load-balancer
- https://www.baeldung.com/spring-cloud-load-balancer
- https://www.baeldung.com/spring-cloud-gateway



Internet Services Architectures

Database Migration

Michał Wójcik

Database structure

From where comes **database structure**:

- it is already present (new project for existing database),
- developed together with the project,
- developed independently of the project.

Database structure - create

How to create database structure:

- write SQL scripts:
  - can be executed independently,
  - can be integrated into container configuration;
- use JPA auto generation feature:
  - can be created automatically on application startup.

Database structure - develop

Database structures are not static:

- can change (quite often) during project rapid development before first release:
  - no need to maintain production data;
- can change (probably rarely) during project maintenance:
  - need to maintain production data;
- can change during project development (introducing new features, etc.) after relases:
  - need to maintain production data.

Database structure - develop

What to do with existing data:

- let JPA make migration:
  - executed automatically on application startup,
  - will work for simple changes like adding new column;
- write migration SQL:
  - can deal with complex changes,
  - executed independently;
- use migration framework:
  - executed automatically on application startup,
  - can be executed independently using CLI,
  - can deal with complex changes,
  - can be written in SQL or framework notation.

Migration tools

Database migration tools:

- Flyway - supports version control for database changes using plain SQL scripts or Java code using JDBC,
- Liquibase - uses XML, YAML, or JSON to define database changes and supports various databases,
- Migrate - a Go based tool based on SQL scripts,
- Alembic - a Python SQL toolkit with script based migrations,
- db-migrate - Node.js applications migration tool with script based migrations.

Liquibase

Dependency for Liquibase:

```
<dependency>
    <groupId>org.liquibase</groupId>
    <artifactId>liquibase-core</artifactId>
</dependency>
```

Changelog (migrations) can be provided in XML and need to be configured in `application.properties`:

```
spring.liquibase.change-log=classpath:/db/changelog.xml
```

Liquibase - changelog

Main changelog can include number of changelogs:

```
<databaseChangeLog
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
        xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
         http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.1.xsd">

    <include file="changelog/01-create-scheme.xml"
             relativeToChangelogFile="true"/>

</databaseChangeLog>
```

Liquibase - migration

Migration described in XML format:

```
<databaseChangeLog
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
        xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
         http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.1.xsd">

    <changeSet id="01" author="psysiu">

        <createTable tableName="books">
            <column name="isbn" type="varchar(255)">
                <constraints primaryKey="true"/>
            </column>
            <column name="title" type="varchar(255)"/>
        </createTable>
    </changeSet>

</databaseChangeLog>
```

Liquibase - migrations

Changeset entries:

- **Create Table** (`createTable`): Defines a new table schema in the database.
- **Drop Table** (`dropTable`): Removes an existing table from the database.
- **Add Column** (`addColumn`): Adds a new column to an existing table.
- **Drop Column** (`dropColumn`): Removes a column from an existing table.
- **Rename Column** (`renameColumn`): Renames a column in an existing table.
- **Add Unique Constraint** (`addUniqueConstraint`): Adds a unique constraint to one or more columns in a table.
- **Drop Unique Constraint** (`dropUniqueConstraint`): Removes a unique constraint from one or more columns in a table.
- **Add Foreign Key Constraint** (`addForeignKeyConstraint`): Creates a foreign key constraint between tables.
- **Drop Foreign Key Constraint** (`dropForeignKeyConstraint`): Removes a foreign key constraint between tables.
- **SQL** (`sql`): Executes arbitrary SQL statements.
- **Include** (`include`): References and includes an external changelog file.
- **Custom Change** (`customChange`): Executes custom Java logic as a change.

Liquibase - different DB

What if developer is using different DB server in tests (e.g. H2) and different DB server on production (e.g. MySQL or PostgreSQL).

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
        xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
         http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.1.xsd">

    <changeSet id="5" author="psysiu" dbms="postgresql">
        <addColumn tableName="files" >
            <column name="content" type="BYTEA"/>
        </addColumn>
    </changeSet>

    <changeSet id="5" author="psysiu" dbms="h2">
        <addColumn tableName="files" >
            <column name="content" type="BLOB"/>
        </addColumn>
    </changeSet>

</databaseChangeLog>
```

Helpful resources

Helpful resources:

- Liquibase Inc. *Liquibase Documentation* https://docs.liquibase.com/home.html
- https://www.baeldung.com/liquibase-refactor-schema-of-java-app
- https://www.baeldung.com/liquibase-rollback
- https://www.baeldung.com/liquibase-vs-flyway
- https://www.baeldung.com/database-migrations-with-flyway

Internet Services Architectures

Microservices - configuration

Michał Wójcik

Standard configuration

Standard configuration:

- default configuration stored in `application.properties` (for Spring Boot),
- default configuration can be overwritten with environment variables,
- environment variables can be set in containerization environment,
- changing shared configuration requires modification in every module.

Centralized configuration

Centralized configuration:

- define configuration for each module in central config server,
- each module connects to config server on start,
- modules do not need to know discovery service address, database address, messaging system address etc.,
- central configuration can be versioned,
- central configuration can be updated during runtime (requires notification system).

Centralized configuration

Server dependency:

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

Client dependency:

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

Centralized configuration

Server configuration:

```
@SpringBootApplication
@EnableConfigServer
public class ConfigApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigApplication.class, args);
    }

}
```

Centralized configuration

By default, configuration is stored in git.

```
spring.cloud.config.server.git.uri=ssh://localhost/config-repo
spring.cloud.config.server.git.clone-on-start=true
spring.security.user.name=root
spring.security.user.password=s3cr3t
```

It can be configured to use local (or classpath) files.

```
spring.profiles.active=native
spring.cloud.config.server.native.search-locations=classpath:/configuration
```

Defining shared configuration in `/configuration/application.properties` or defining particular service configuration in `/configuration/library.properties`.

Centralized configuration

Configuring client in `bootstrap.properties` (in newer versions in `application.properties`):

```
spring.cloud.config.uri=http://localhost:8084
spring.cloud.config.fail-fast=true
```

```
spring.config.import=optional:configserver:http://localhost:8084
```

Requires service (application) name in `application.properties:

```
spring.application.name=library
```

Helpful resources

Helpful resources:

- https://www.baeldung.com/spring-cloud-configuration
- https://www.baeldung.com/spring-cloud-config-remote-properties-override
- https://www.baeldung.com/spring-cloud-config-without-git
- https://www.baeldung.com/spring-cloud-bootstrap-properties

Internet Services Architectures

Messaging

Michał Wójcik

Messaging

Messaging:

- loosely coupled communication:
  - no direct connection between sender and receiver,
  - senders and receivers can connect at any time,
  - everyone knows localization of messaging server;
- communication based on messages,
- asynchronous communication.

Messages destinations

When talking about messaging, there is usually common approach for message destinations:

- Queue:
  - p2p model,
  - only one consume for message,
  - no time dependencies,
  - confirmation of receipt required.
- Topic:
  - publish/subscribe model,
  - the possibility of having multiple consumers per message,
  - time dependencies,
  - confirmation of receipt not required.

Messaging systems

In Java, there are two popular used messaging approaches:

- JMS:
  - Java Messaging Service,
  - use serialization mechanisms,
  - part of Jakarta EE specification,
  - designed for Java frameworks.
- AMQP:
  - The Advanced Message Queuing Protocol,
  - platform-neutral,
  - binary application level protocol.

Messaging systems

Each of approaches have multiple implementations:

- JMS:
  - RabbitMQ (client plugin).
  - Apache ActiveMQ,
  - Apache ActiveMQ Artemis,
  - OpenMQ.
- AMQP:
  - RabbitMQ,
  - Apache ActiveMQ,
  - Apache ActiveMQ Artemis,
  - Apache Qpid,
  - Microsoft Azure Service Bus.

Other messaging systems

Other messaging systems:

- Apache Kafka,
- Apache Pulsar,
- RSocket,
- ...

RabbitMQ

RabbitMQ can be started as Docker container.

```
services:
  rabbitmq:
    image: rabbitmq:3.12.9-management-alpine
    restart: always
    ports:
      - "5672:5672"
      - "15672:15672"
```

Dependency

For dependency, general AMQP dependency can be used.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

This project provides support for using Spring and Java with AMQP 0.9.1, and in particular RabbitMQ.

Defining message goals

Message goal can be configured in the broker server, but if application has admin rights, those can be automatically created on application startup.

```
@Bean
public Queue professionQueue() {
    return new Queue("libraryQueue", true);
}
```

```
@Bean
public TopicExchange libraryTopic() {
    return new TopicExchange("libraryTopic");
}
```

Sending message

```java
@Repository
public class MessageRepository {

    @Autowired
    private final RabbitTemplate rabbitTemplate;

    public void notifyOne(Book book) {
        rabbitTemplate.convertAndSend("libraryQueue", book);
    }

    public void NotifyAll(Book book) {
        rabbitTemplate.convertAndSend("libraryTopic", book);
    }

}
```

Sending messages requires broker implementation object, like `RabbitTemplate`.

Receiving messages

```
@Component
public class ProfessionRabbitListener {

    @RabbitListener(queues = "bookQueue")
    public void onMessage(Book book) {

    }

}
```

Receiving messages requires broker implementation annotation, like `@RabbitListener`.

Different message types

```
@Component
@RabbitListener(queues = "bookQueue")
public class ProfessionRabbitListener {

    @RabbitHandler
    public void onDeleteMessage(DeleteBook book) {

    }

    @RabbitHandler
    public void onCreateMessage(CreateBook book) {

    }

}
```

Helpful resources

Helpful resources:

- https://www.rabbitmq.com/documentation.html
- https://www.baeldung.com/rabbitmq
- https://www.baeldung.com/spring-amqp
- https://docs.spring.io/spring-boot/docs/current/reference/html/messaging.html

Internet Services Architectures

Open Authorization

Michał Wójcik

Authentication vs authorization

Authentication - process of verifying the identity of a user or system attempting to access a resource. It involves confirming that the credentials provided (such as username/password, tokens, biometrics, etc.) match the expected credentials stored in a system.

Authorization - process of determining what actions or resources a verified and authenticated user or system is allowed to access or perform within the application or system. It involves checking the permissions, roles, or privileges associated with the authenticated identity and granting or denying access based on those permissions.

User authentication

User authentication:

- Basic Authentication - setting username and password encoded with Base64 sent in each request using `Authorization` header,
- Form Authentication - user credentials collected using form and appropriate session object is created.
- API key - user is authorized on a basic of API key added to each request (in header or query param).

Those are still widely used but quite legacy methods.

Keeping session

How to keep session in subsequent requests:

- Cookie-based Authentication - authenticated user's session is identified by cookie with session ID sent with each request, cookie is set in response (e.g. after form authentication),
- Query param Authentication - authenticated user's session is identified by session ID passed with each.
- Token - authorized user is identified by token created during authentication process.

Those are still widely used. For example JSF by default now uses cookie but in past used query param by default.

OAuth

OAuth (Open Authorization) - authorization protocol allowing third-party applications to access user data without exposing credentials. It works by providing tokens (access tokens) to the third-party app after user consent, allowing restricted access to specific resources.

OAuth2 - updated version of OAuth with multiple flows designed for different scenarios.

OAuth roles

Roles in OAuth:

- **Resource Owner:** The entity that can grant access to a protected resource. Typically, this is the end-user.
- **Client:** An application requesting access to a protected resource on behalf of the resource owner. This could be a web or mobile application.
- **Resource Server:** The server hosting the protected resources that can only be accessed with proper authorization.
- **Authorization Server:** The server responsible for authenticating the resource owner and providing access tokens to the client after successful authentication.

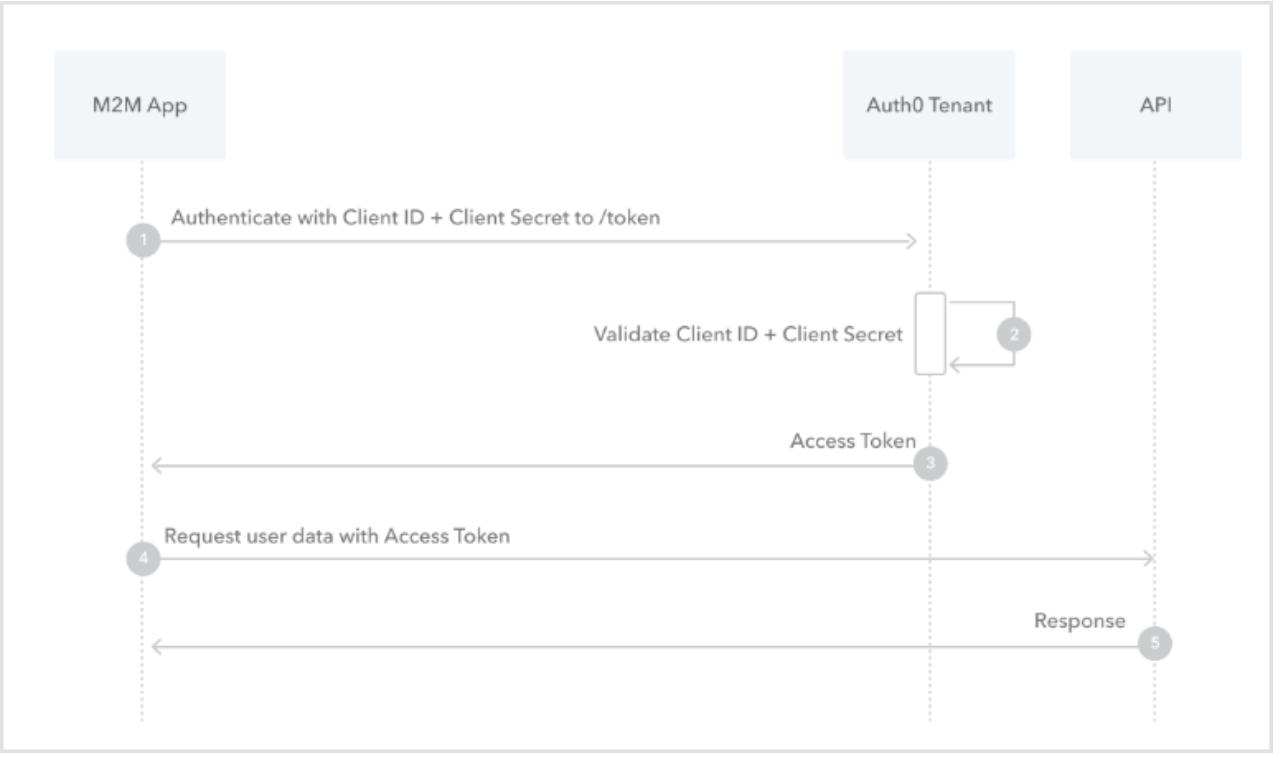
OAuth2 flows

OAuth2 flows:

- **Client Credentials Flow** - Utilized by confidential clients (such as servers) to authenticate and obtain access tokens based on client credentials (not user credentials).
- **Resource Owner Password Credentials Flow** - Allows the client to directly obtain user credentials (username/password) and exchange them for an access token. Less secure and should be used cautiously.
- **Authorization Code Flow** - A flow used by web and mobile apps where the client requests authorization from the resource owner through a redirection process. It exchanges an authorization code for an access token.
- **Implicit Flow** - Primarily used by single-page web apps (SPAs) where access tokens are issued directly without exchanging an authorization code. It's less secure due to tokens being exposed in the browser.
- **Device Authorization Flow** - Suitable for devices with limited input capabilities (e.g., smart TVs) where users can authenticate via a secondary device, obtaining an access token.

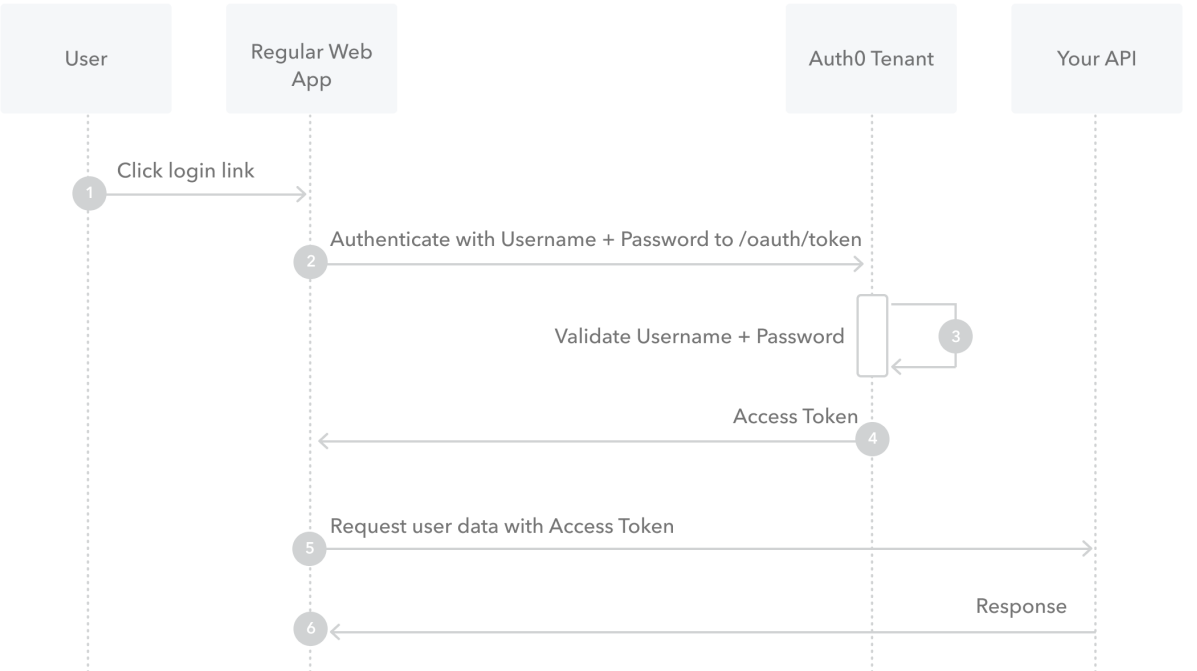
POLITECHNIKA GDAŃSKA

Client Credentials Flow



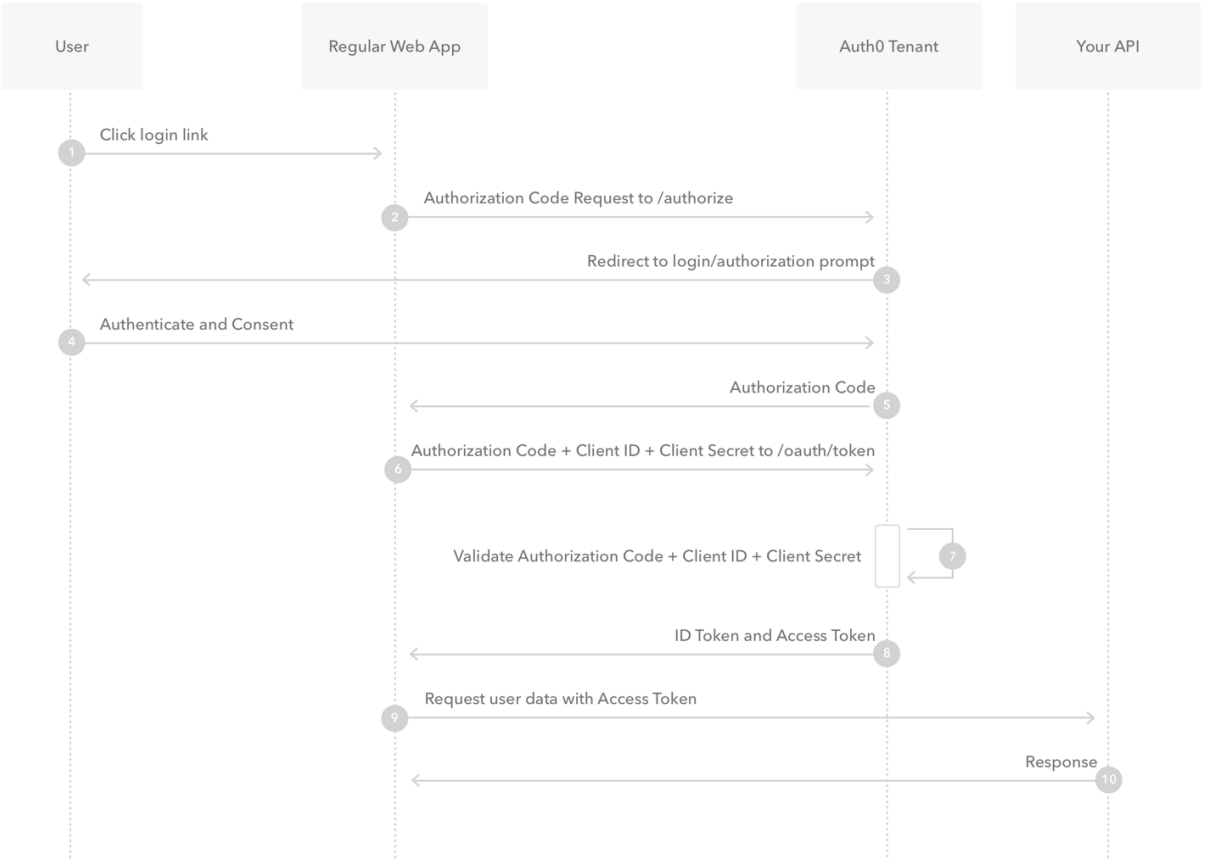
POLITECHNIKA GDAŃSKA

Resource Owner Password Credentials



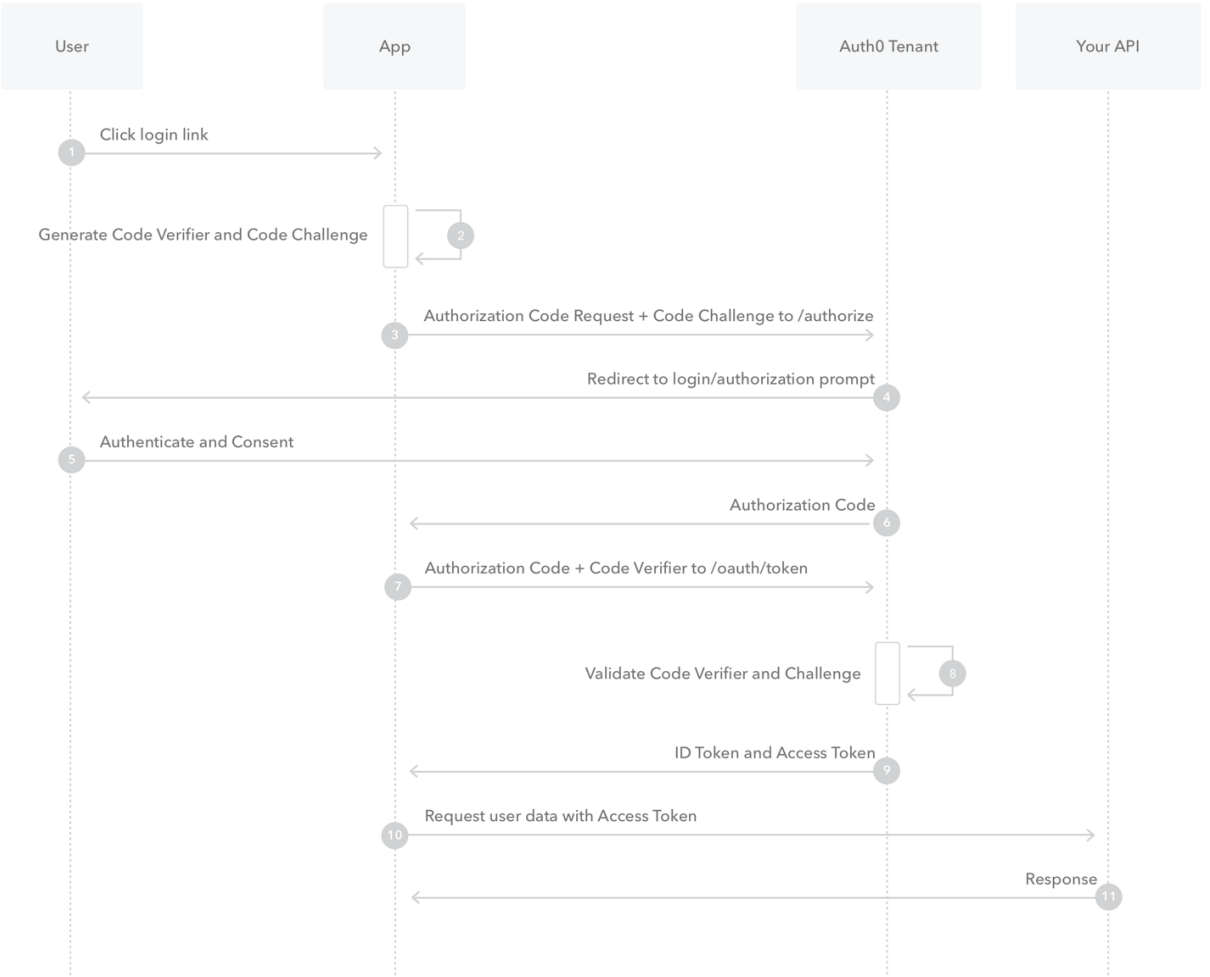
POLITECHNIKA GDAŃSKA

Authorization Code Flow



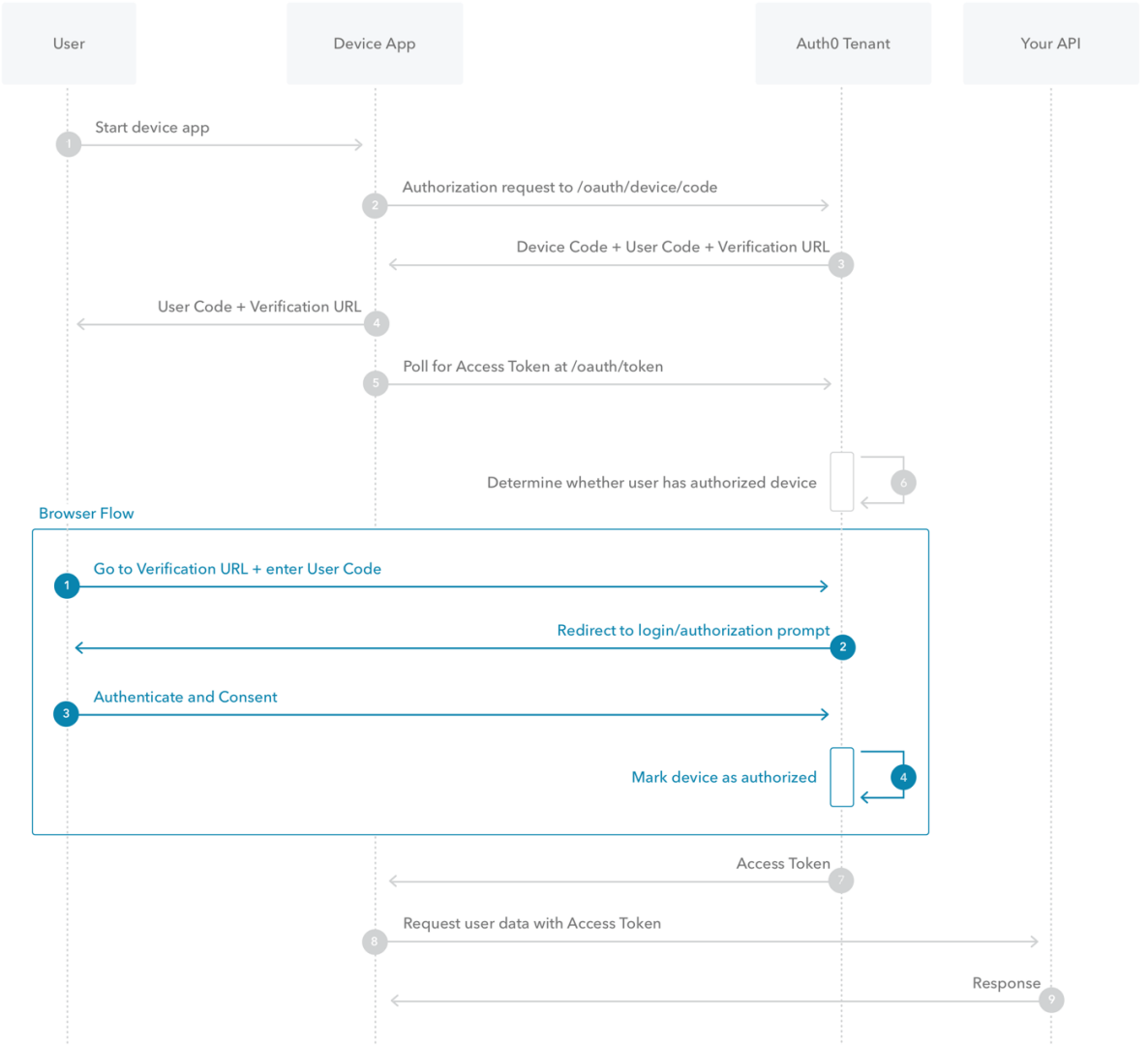
POLITECHNIKA GDAŃSKA

Authorization Code Flow with PKCE



POLITECHNIKA GDAŃSKA

Device Authorization Flow



Authorization server

So well known authorization servers with OAuth2 support:

- Keycloak,
- Auth0,
- Spring Authorization Server,
- Okta,
- Azure Active Directory,
- Google Identity Platform,
- ...

Spring authorization server

Spring Authorization Server - ready to use authorization server with OAuth2 support.

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-authorization-server</artifactId>
</dependency>
```

Spring authorization server - clients

Whole configuration can be made with properties with almost no code using `spring.security.oauth2.authorizationserver` properties defining clients.

```
m2m.registration.client-id=m2m-client
m2m.registration.client-secret={noop}m2m-secret
m2m.registration.authorization-grant-types[0]=client_credentials
m2m.registration.client-authentication-methods[0]=client_secret_basic
m2m.registration.client-authentication-methods[1]=client_secret_post
```

Spring authorization server - users

Users can be defined with `UserDetailsService`. For easy development and testing it can be done in-memory.

```
@Bean
public UserDetailsService users() {
    return new InMemoryUserDetailsManager(
            User.withUsername("admin")
                    .password("{noop}adminadmin")
                    .build()
    );
}
```

Spring resource server

Resource server in Spring can be configured with **Spring Security** and **OAuth2 Resource Server**.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
    </dependency>
</dependencies>
```

For simple configuration it is enoguh to present `jww.issuer-uri` property.

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:8090
```

Helpful resources

OAuth2 flows:

- https://auth0.com/docs/get-started/authentication-and-authorization-flow/which-oauth-2-0-flow-should-i-use
- https://auth0.com/docs/get-started/authentication-and-authorization-flow/client-credentials-flow
- https://auth0.com/docs/get-started/authentication-and-authorization-flow/resource-owner-password-flow
- https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow
- https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-proof-key-for-code-exchange-pkce
- https://auth0.com/docs/get-started/authentication-and-authorization-flow/device-authorization-flow

Helpful resources:

- https://www.baeldung.com/spring-security-5-default-password-encoder
- https://www.baeldung.com/spring-security-enable-logging
- https://docs.spring.io/spring-authorization-server/reference/getting-started.html
- https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html

POLITECHNIKA GDAŃSKA

Helpful resources

Standards:

- https://datatracker.ietf.org/doc/html/rfc6749
- https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-09

Utils:

- https://jwt.io/
- https://www.base64encode.org/
- https://tonyxu-io.github.io/pkce-generator/