Spring Rest API. Serialization. JSON

softserve

Agenda

- Web Services. Introduction
- RESTful Web Services
- REST Services with Spring
- Convert Java Object to / from JSON
- REST Services. Case Studies
- Security for REST Service
- Call REST API
- Case Studies

softserve

Web services.
Introduction

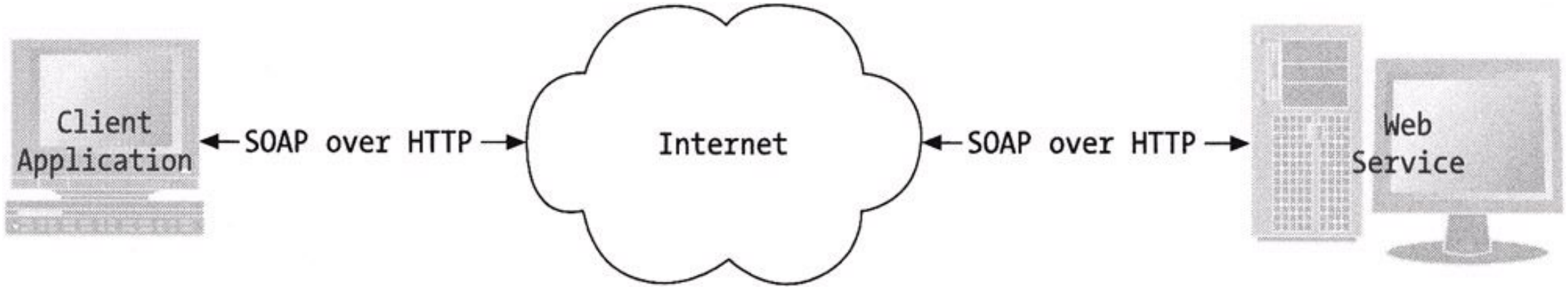
softserve

Distributed Computing

- Java RMI is a mechanism that allows one to invoke a method on an object that exists in another address space.
- RPC (remote procedure call) in distributed computing is when a computer program causes a procedure (subroutine) to execute in a different address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction.
- DCOM is the distributed extension to COM (Component Object Model) that builds an object remote procedure call (ORPC) layer on top of DCE RPC to support remote objects.
  - COM is an component based development model for Windows environment.
- CORBA is based on the Request-Response architecture.

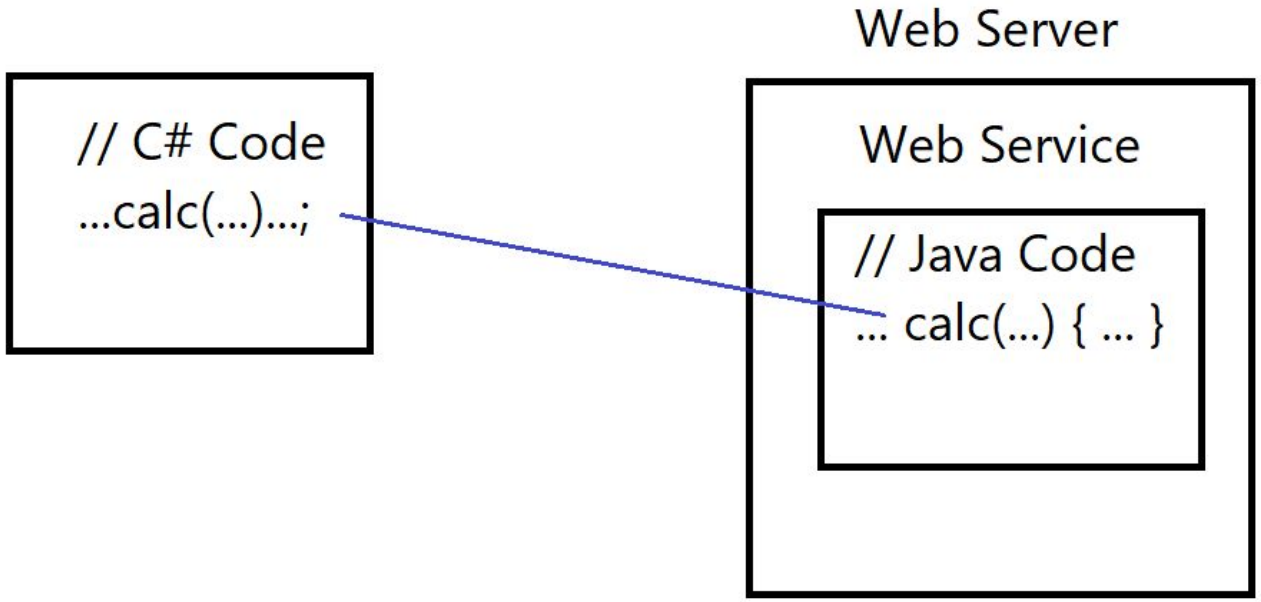
Web Services

- Web services are software components that can be accessed and executed remotely via a network by a client application using standard protocols such as Hypertext Transfer Protocol (HTTP) and Simple Object Access Protocol (SOAP)



Web Services

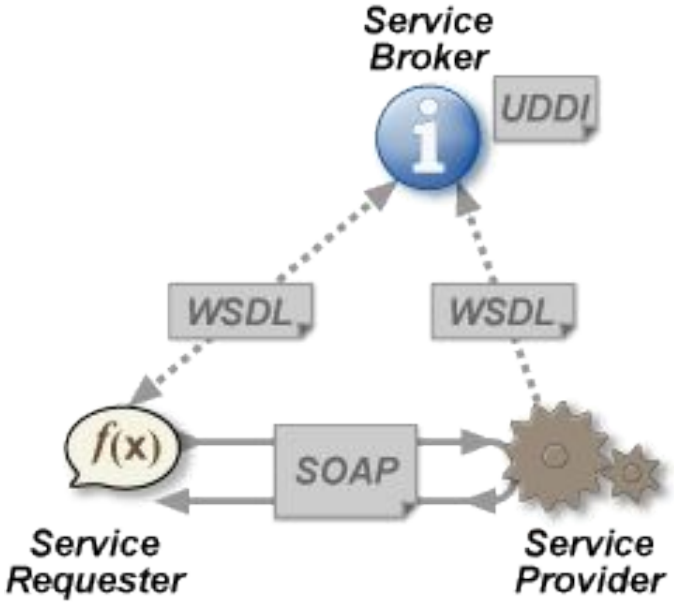
- The W3C defines a "web service" as "a software system designed to support interoperable machine-to-machine interaction over a network.
- It has an interface described in a machine processable format (specifically Web Services Description Language WSDL).
- Other systems interact with the web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."



softserve

Web Services

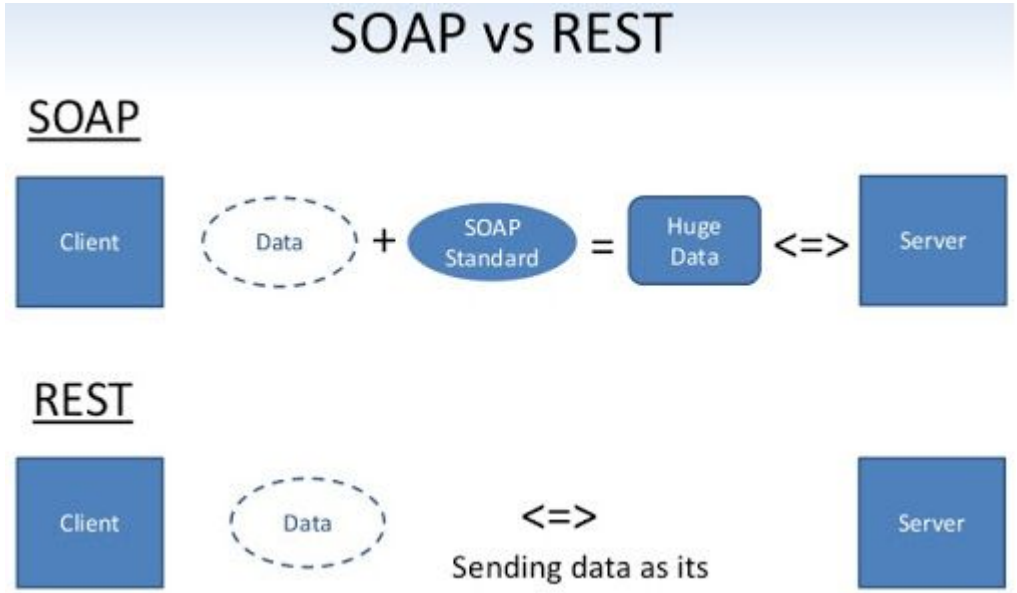
- UDDI is an evolving standards for describing, publishing, and discovering the web services that a business provides.
- Once a web service is developed and a WSDL document describing it is created, there needs to be a way to get the WSDL information into the hands of the users who want to use the web service it describes.
- When a web service is published in a **UDDI** registry, potential users have a way to look up and learn about the web service's existence.



softserve

Web Services

- REST (Representational state transfer) is a software architectural style that defines a set of constraints to be used for creating Web services.
- REST services do not require XML, SOAP, or WSDL service-API definitions.
- An architecture based on **REST** (one that is '**RESTful**') can use WSDL to describe SOAP messaging over HTTP, can be implemented as an abstraction purely on top of SOAP (e.g., WS-Transfer), or can be created without using SOAP at all.



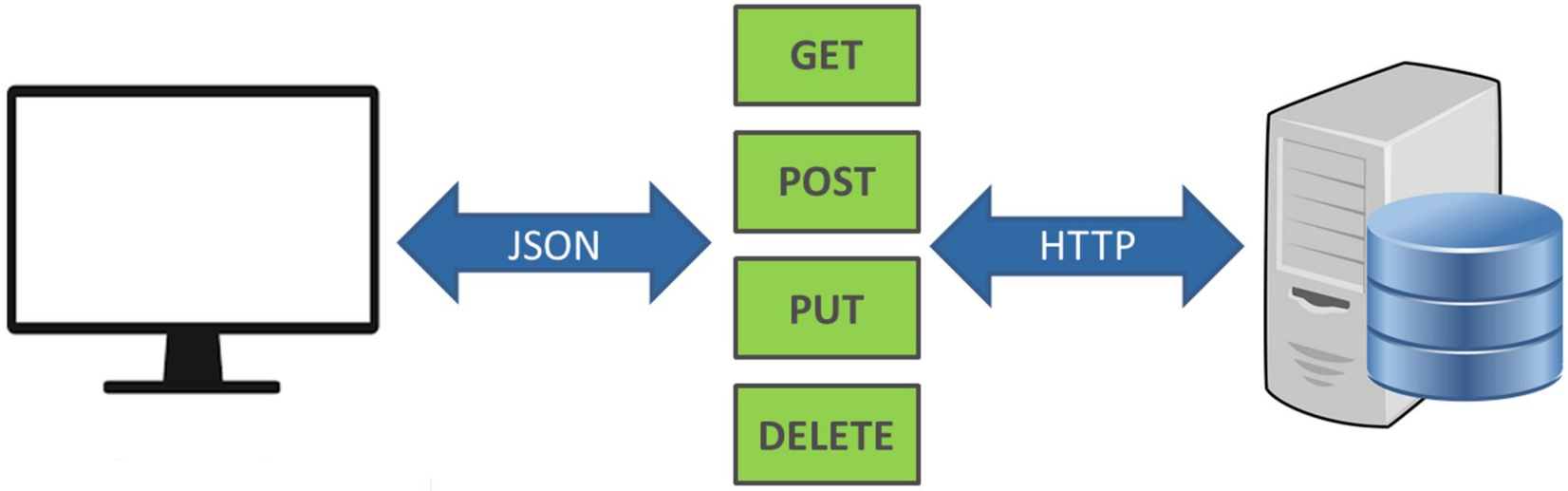
softserve

RESTful Web Services

softserve

RESTful Web Services

- REST (Representational state transfer) – a style of software architecture for distributed systems.
- The term REST was introduced in 2000 by Roy Fielding, one of the authors of HTTP-protocol.
- Web services that conform to the REST architectural style, called **RESTful** Web services, provide interoperability between computer systems on the internet.



softserve

RESTful Web Services

- REST attempts to describe architectures which use HTTP or similar protocols by constraining the interface to a set of well-known, standard operations (GET, POST, PUT, DELETE for HTTP).
- Here, the focus is on interacting with resources.
- Each unit is uniquely determined by the **information URL**.
  - This means that the **URL** is actually a primary key for the data unit.
- The third book from the bookshelf will look /book/3, and 35 pages in this book – /book/3/page/35

Resource	GET	POST	PUT	DELETE
/book/{id or name}	Get book by id or name	Create/add new book	Change the book	Remove book
/books	Get all books	Add books by list	Update books by list	Remove books by list

RESTful Web Services. Constraints

- The constraints of the REST architectural style
  - simplicity of a uniform interface;
  - interacting with resources through views. Each resource has its own unique URI;
  - scalability allowing the support of large numbers of components and interactions among components;
  - self-contained messages. Each response should **contain all the necessary information** so that it can be processed correctly without resorting to the help of other resources.
  - modifiability of components to meet changing needs (even while the application is running);
  - visibility of communication between components by service agents;
  - portability of components by moving program code with the data;

Properties of HTTP Methods

- Idempotency – the ability to perform the same call to the service several times, while the answer will be the same each time.

HTTP Method	Idempotent	Safe
OPTIONS	Yes	Yes
GET	Yes	Yes
HEAD	Yes	Yes
PUT	Yes	No
POST	No	No
DELETE	Yes	No
PATCH	Yes	No

REST Services with Spring

softserve

Spring. REST

- First, you need to give some thought to what your API will look like.
- For Example, to handle GET requests for `/hello-world`, optionally with a name query parameter.
- In response to such a request, REST service to send back JSON, representing a greeting.

```
{
 "id": 1,
 "content": "Hello, World!"
}
```

```
@AllArgsConstructor
@Data
public class Greeting {
  private final long id;
  private final String content;
}
```

softserve

Spring. REST

- In Spring, REST endpoints are **Spring MVC controllers**.
- The following Spring MVC controller handles a GET request for the `/hello-world` endpoint and returns the Greeting resource.

```
@Controller
public class HelloWorldController {
  private static final String template = "Hello, %s!";
  private final AtomicLong counter = new AtomicLong();
  @GetMapping("/hello-world")
  @ResponseBody
  public Greeting sayHello(
  @RequestParam(name="name", required=false,
      defaultValue="Stranger") String name) {
    return new Greeting(counter.incrementAndGet(),
      String.format(template, name));
  }
}
```

Convert Java Object to/from JSON

softserve

Java Gson Library

- Gson is a Java serialization/deserialization library to convert Java Objects into JSON and back.
- Gson was created by Google for internal use and later open sourced.
- Maven dependency

```
    <dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
    </dependency>
```

```
Map<Integer, String> colours = new HashMap<>();
colours.put(1, "blue");
colours.put(2, "yellow");
colours.put(3, "green");
Gson gson = new Gson();
String output = gson.toJson(colours);
System.out.println(output);
{"1":"blue","2":"yellow","3":"green"}
```

Java Gson Library

```
@AllArgsConstructor
@Data
public class User {
    private String firstName;
    private String lastName;
}
```

- JSON serialization

```
User user = new User("Peter", "Flemming");
Gson gson = new GsonBuilder()
  .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE)
  .create();
String output = gson.toJson(user);
System.out.println(output);
{"FirstName":"Peter","LastName":"Flemming"}
```

Java Gson Library

- Deserialization with Gson

```
String json_string =
  "{\"firstName\":\"Tom\", \"lastName\": \"Broody\"}";
Gson gson = new Gson();
User user = gson.fromJson(json_string, User.class);
System.out.println(user);
```

- This is the output of the example.

```
User[firstname: Tom, lastname: Broody}
```

- Notice that getters and setters are not necessary.

softserve

Java Jackson Library

- Jackson Dependency

```
<dependency>
  <groupId>org.codehaus.jackson</groupId>
  <artifactId>jackson-mapper-asl</artifactId>
</dependency>
```

- Java Object to JSON

```
ObjectMapper mapper = new ObjectMapper();
User user = new User("Peter", "Flemming");
String jsonString = mapper.writeValueAsString(user);
System.out.println(jsonString);
```

- Java JSON to Object

```
User user2 = objectMapper.readValue(jsonString, User.class);
```

Spring MVC. RestController

- Spring MVC implicitly uses Jackson to serialize an object to JSON.
- Spring Boot makes converting an object to JSON much easier.
- Classes `User` and `UserController` are enough to get user information in JSON format

```
@RestController
public class UserController {
  @GetMapping ("/")
  public User getUser() {
    User user = new User();
    user.setFirstName("Peter");
    user.setLastName("Flemming");
    return user;
  }
}
```

Spring MVC. RestController

- Change field name

```
public class User {
    private String firstName;
    @JsonProperty("surname")
    private String lastName;
}
```

- Output of the request.

```
{"firstName":"Peter","surname":"Flemming"}
```

- Sort fields alphabetically

```
@JsonPropertyOrder(alphabetic = true)
public class User {...}
```

- Remove empty fields

```
@JsonInclude(JsonInclude.Include.NON_NULL)
public class User {...}
```

Spring MVC. RestController

- Conversion to json can be done explicitly

```
@RestController
public class JsonController {
  private ObjectMapper mapper;
  @Autowired
  public JsonController(ObjectMapper mapper) {
    this.mapper = mapper;
  }
  @GetMapping("/")
  public User getUser() throws JsonProcessingException {
    User user = new User("Peter","Flemming");
    log.info("Json: ", mapper.writeValueAsString(user));
    return user;
  }
}
```

REST Services.
Case Studies

softserve

REST Service. DB

Data Output | Explain | Messages | Notifications

	id [PK] integer	name character varying (255)
1	1	ROLE_USER
2	2	ROLE_ADMIN

Data Output | Explain | Messages | Notifications

	id [PK] integer	login character varying (255)	password character varying (255)	roles_id integer
1	1	admin	$2a$10$nSRsm.NAMX21L1.q...	2
2	2	test	$2a$10$kDkHYxQJall9nKjOg...	1

softserve

REST Service. Role

```java
@AllArgsConstructor
@NoArgsConstructor
//@RequiredArgsConstructor
@Data
@Entity
@Table(name = "roles")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column
    private String name;
}
```

REST Service. User

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

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column
    private String login;

    @Column
    private String password;

    @ManyToOne
    @JoinColumn(name = "roles_id")
    private Role role;

}
```

softserve

REST Service. Repository

```java
public interface RoleRepository extends JpaRepository<Role, Integer> {

    Role findByName(String name);
}
```

```java
public interface UserRepository extends JpaRepository<User, Integer> {

    User findByLogin(String login);
}
```

REST Service. Service

```java
@Service
@Transactional
public class UserServiceImpl implements UserService {

    private UserRepository userRepository;

    private RoleRepository roleRepository;

    @Autowired
    public UserServiceImpl(UserRepository userRepository,
            RoleRepository roleRepository
            ) {
        this.userRepository = userRepository;
        this.roleRepository = roleRepository;
        initDefaultData();
    }

    private void initDefaultData() {
        if ((roleRepository.count() ==0)
                && (userRepository.count() ==0)) {
            roleRepository.save(new Role(1, RoleData.USER.toString()));
            Role adminRole = roleRepository.save(new Role(2, RoleData.ADMIN.toString()));
            User admin = new User(1, "admin", "admin", adminRole);
            userRepository.save(admin);
        }
    }
```

softserve

REST Service. Service

```
    public boolean saveUser(UserRequest userRequest) {
        User user = new User();
        user.setRole(roleRepository.findByName(RoleData.USER.toString()));
        user.setLogin(userRequest.getLogin());
        //user.setPassword(passwordEncoder.encode(userRequest.getPassword()));
        user.setPassword(userRequest.getPassword());
        return (userRepository.save(user) != null);
    }

    public User findByLogin(String login) {
        return userRepository.findByLogin(login);
    }

    public UserResponce findByLoginAndPassword(UserRequest userRequest) {
        UserResponce result = null;
        User user = userRepository.findByLogin(userRequest.getLogin());
        if ((user != null) ) {
            result = new UserResponce();
            result.setLogin(userRequest.getLogin());
            result.setRolename(user.getRole().getName());
        }
        return result;
    }
}
```

softserve

REST Service. DTO

```java
@AllArgsConstructor
@Data
public class OperationResponce {

    private boolean status;

}
```

```java
@AllArgsConstructor
@Data
public class UserRequest {

    @NotEmpty
    private String login;

    @NotEmpty
    private String password;

}
```

```java
@AllArgsConstructor
@Data
public class TokenResponse {

    private String token;

}
```

```java
@Data
public class UserResponce {

    private String login;
    private String rolename;
}
```

softserve

REST Service. Controller

```
@RestController
@Slf4j
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping(value = { "/", "/index" }, method = RequestMethod.GET)
    public OperationResponce signUp() {
        return new OperationResponce(true);
    }

    @PostMapping("/signup")
    public OperationResponce signUp(
            @RequestParam(value = "login", required = true)
            String login,
            @RequestParam(value = "password", required = true)
            String password) {
        log.info("**/signup userLogin = " + login);
        UserRequest userRequest = new UserRequest(login, password);
        return new OperationResponce(userService.saveUser(userRequest));
    }
```

softserve

REST Service. Controller

- A token is just a string that is generated at the user's request.
- The user registers in the system, makes a request to generate a token.
- Then user can make authorized requests to the server using the token.

```
    @PostMapping("/signin")
    public TokenResponse signIn(
            @RequestParam(value = "login", required = true)
            String login,
            @RequestParam(value = "password", required = true)
            String password) {
        log.info("**/signin userLogin = " + login);
        UserRequest userRequest = new UserRequest(login, password);
        UserResponce userResponce = userService.findByLoginAndPassword(userRequest);
        return new TokenResponse("123456789" + userResponce.getLogin());
//          return new TokenResponse(jwtProvider.generateToken(userResponce.getLogin()));
    }

}
```

Security for REST Service

softserve

Security for the Rest. Ways

- **Basic Authentication** – the rest client specifies his username and password to gain access to the rest service.
  - Login and password are transmitted over the network as plain text encoded with simple Base64.
  - When using this method, https must be used.
- **Digest authentication** is the same as the first method, only the login and password are transmitted in encrypted form, and not as plain text.
  - Login and password are encrypted with MD5 algorithm and it is difficult to decrypt it.
  - With this approach, you can use an unsecured http connection.
- **Token Authentication** – user using credentials, logs into the application and receives a token to access the rest service.
  - Access to the service that issues tokens must be done via an https connection.
  - The token must contain a username, password, it can also contain expiration time and user roles, as well as any information necessary for your application.

Security for the Rest. Ways

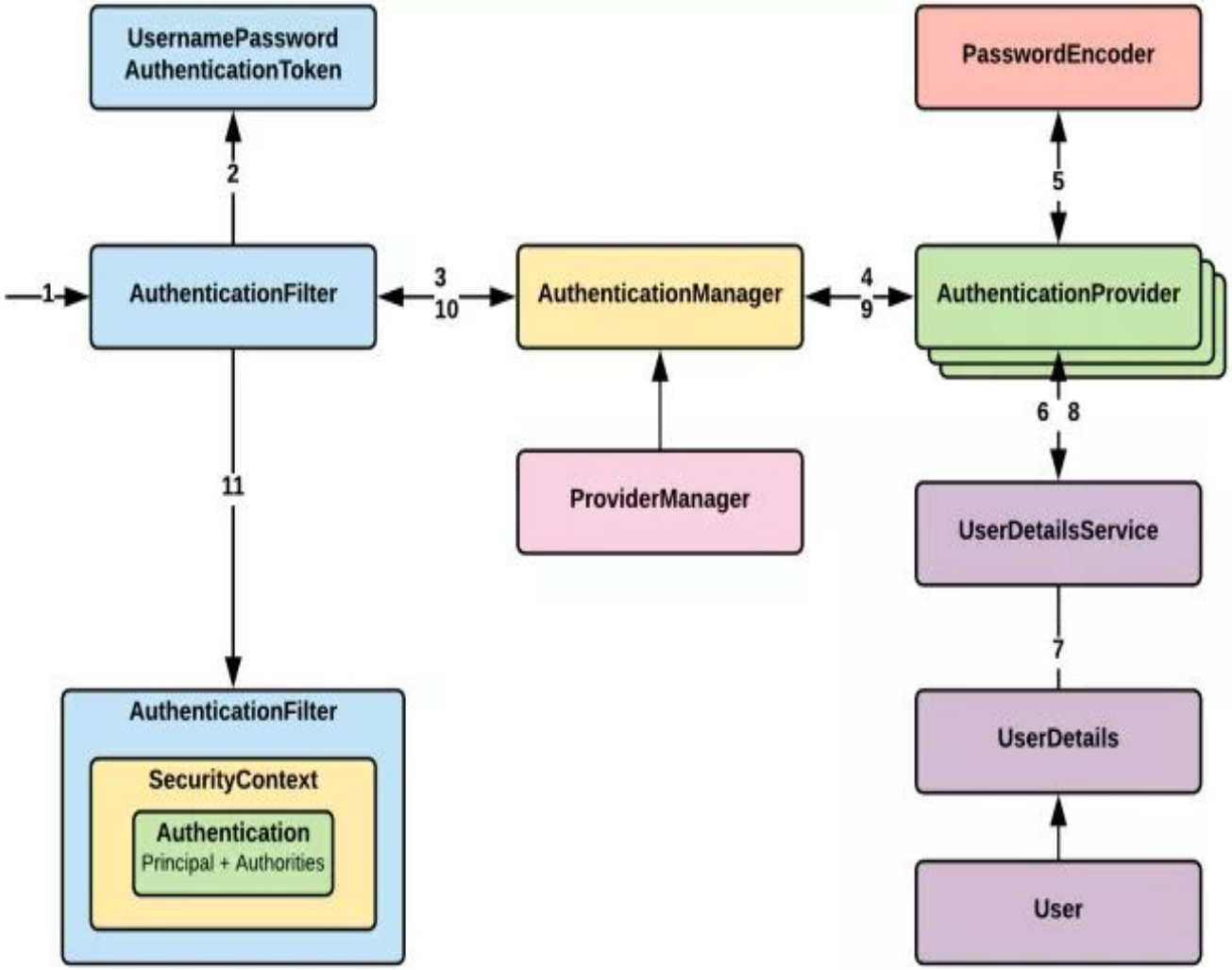
- **Digital Signature** (public / private key pair) using a public key cryptosystem.
  - To implement this approach, you need to write two filters: one on the server side, the other on the client side.
- **Certificate Authentication**, if the client does not provide the required certificate when requesting, he will not receive a response from the server, or rather receive a response that the certificate is missing.
  - There are two types of certificates
  - Trusted – those that everyone can check and they are registered in a single certification center;
  - Self signed – those that you generate yourself and they need to be added to your rest service in exceptions so that it knows about their existence and that they can be trusted.
- **OAuth2 authorization** works well for large portals.
  - Spring security provides us with the `OAuthTemplate` class.

Spring Security with JWT

- **JWT** (JSON Web Token) is one of the most common request authentication methods.
- Token is a string that is generated at the user's request
- The user registers in the system, then makes a request to generate a token.
  - When a token is generated on the server, data about the user's session or other necessary information is placed in it, if necessary, and encrypted using encryption algorithms.
- Then user can make authorized requests to the server using the token.
  - In all data extraction methods, the JWT token is also checked to see if it has expired and whether the signature is valid.
- Typically, the token is placed in the request header for ease of transmission and reading.

Spring Security with JWT

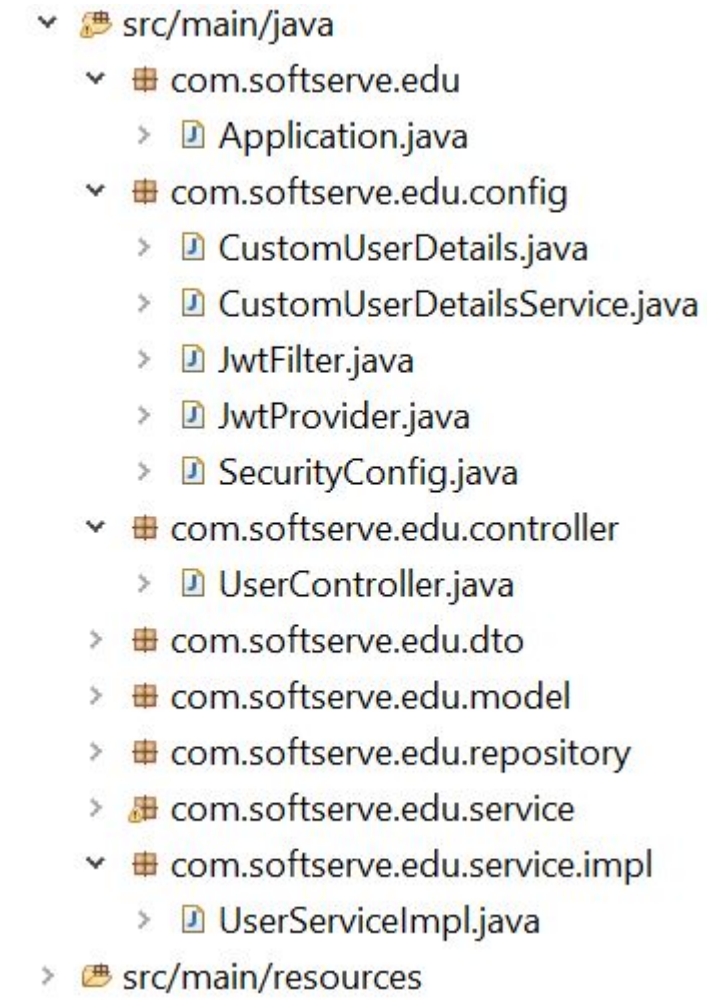
- Add the `/sigin` endpoint to the application for authentication.
- For first we must develop classes `CustomUserDetails`, `CustomUserDetailsService`, `JwtProvider` and `JwtFilter`.
- Next, we must to configure credentials in `SecurityConfig` class.
- In all requests, user send a JWT token in the header.
- Application verifies the authenticity of the token in the `JwtFilter` and, if it is correct, passes the request further.



softserve

Spring Security with JWT

- For each request, we take the JWT token from the Authorization header (it starts with the "**Bearer** " prefix).
- We extract the login from it (which was recorded when the token was created).
- Checking the signature (`validateToken()`)
- If everything is ok, set the `Authencation` object in the SecurityContext (and `UserDetails` in `Authencation`).
- If not everything is ok with the token, then the filter will **not allow** the request to the controller to the protected `/url`.



softserve

CustomUserDetails class

- `CustomUserDetails` implements `UserDetails`.
- `GrantedAuthority` is the interface for user accesses.
  - One of its implementations is `SimpleGrantedAuthority` to which you can add only the **role name** and Spring will give access if the role name matches the role name in the `hasRole` method.
  - Spring adds `ROLE_` **prefix** to the role name.
- Add a method to the `CustomUserDetails` class to converts a user from the database into a `CustomUserDetails` object.

CustomUserDetails class

```
public class CustomUserDetails implements UserDetails {

    private static final long serialVersionUID = 1L;

    private String login;
    private String password;
    private Collection<? extends GrantedAuthority> grantedAuthorities;
    private Date expirationDate;

    public static CustomUserDetails fromUserEntityToCustomUserDetails(User user) {
        CustomUserDetails customUserDetails = new CustomUserDetails();
        customUserDetails.login = user.getLogin();
        customUserDetails.password = user.getPassword();
        customUserDetails.grantedAuthorities = Collections
                .singletonList(new SimpleGrantedAuthority(user.getRole().getName()));
        return customUserDetails;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return grantedAuthorities;
    }

    @Override
    public String getPassword() {
        return password;
    }
```

CustomUserDetails class

```
@Override
public String getUsername() {
    return login;
}

@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}

public void setExpirationDate(Date expirationDate) {
    this.expirationDate = expirationDate;
}

public Date getExpirationDate() {
    return expirationDate;
}
```

CustomUserDetailsService class

- The `CustomUserDetailsService` class will implement the `UserDetailsService` interface.
- This interface has only one `loadUserByUsername()` method.
- In this method, we get user from the database by login, convert it to CustomUser.

```
@Component
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private UserService userService;

    @Override
    public CustomUserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = userService.findByLogin(username);
        return CustomUserDetails.fromUserEntityToCustomUserDetails(user);
    }

}
```

JwtProvider class

```java
@Component
//@Log
@Slf4j
public class JwtProvider {

    @Value("$(jwt.secret)")
    private String jwtSecret;

    public String generateToken(String login) {
        Date date = Date.from(LocalDate.now().plusDays(15).atStartOfDay(ZoneId.systemDefault()).toInstant());
        return Jwts.builder()
                .setSubject(login)
                .setExpiration(date)
                .signWith(SignatureAlgorithm.HS512, jwtSecret)
                .compact();
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parser()
                .setSigningKey(jwtSecret)
                .parseClaimsJws(token);
            return true;
        } catch (Exception e) {
            log.error("invalid token");
        }
        return false;
    }
```

JwtProvider class

```
    public String getLoginFromToken(String token) {
        Claims claims = Jwts
                .parser()
                .setSigningKey(jwtSecret)
                .parseClaimsJws(token)
                .getBody();
        return claims.getSubject();
    }

    public Date getExpirationDate(String token) {
        Claims claims = Jwts
                .parser()
                .setSigningKey(jwtSecret)
                .parseClaimsJws(token)
                .getBody();
        return claims.getExpiration();
    }

}
```

JwtFilter class

- `JwtFilter` class inherits from `GenericFilterBean`.
- `GenericFilterBean` plain base `javax.servlet.Filter` implementation
  - After inheritance, we have access to one method for the definition: `doFilter()`.
  - This method will work when the filter is running.
- We need to get the token from the request.
  - The token will be received in the header with the "`Authorization`" key.
- After retrieving the token, I need to verify that it starts with a "`Bearer `" word.
  - Such a RFC6750 standard for tokens.
- Create a `UsernamePasswordAuthenticationToken` object from the spring security library and write this object to the `SecurityContextHolder`.

JwtFilter class

```java
@Component
@Slf4j
public class JwtFilter extends GenericFilterBean {

    public static final String AUTHORIZATION = "Authorization";
    public static final String BEARER = "Bearer ";

    @Autowired
    private JwtProvider jwtProvider;

    @Autowired
    private CustomUserDetailsService customUserDetailsService;

    private String getTokenFromRequest(HttpServletRequest request) {
        String bearer = request.getHeader(AUTHORIZATION);
        if (hasText(bearer) && bearer.startsWith(BEARER)) {
            return bearer.substring(BEARER.length());
        }
        return null;
    }
```

JwtFilter class

```
@Override
public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
        throws IOException, ServletException {
    log.info("**doFilter Start");
    String token = getTokenFromRequest((HttpServletRequest) servletRequest);
    if ((token != null) && jwtProvider.validateToken(token)) {
        String login = jwtProvider.getLoginFromToken(token);
        CustomUserDetails customUserDetails = customUserDetailsService.loadUserByUsername(login);
        customUserDetails.setExpirationDate(jwtProvider.getExpirationDate(token));
        UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(customUserDetails,
                null, customUserDetails.getAuthorities());
        SecurityContextHolder.getContext().setAuthentication(auth);
    }
    filterChain.doFilter(servletRequest, servletResponse);
    log.info("**doFilter done");
}
```

SecurityConfig class

- Add the `@EnableWebSecurity` annotation to the `SecurityConfig` class, which indicates that this class is a Spring Security settings class.
- Inherited this class from `WebSecurityConfigurerAdapter`.
  - This class allows you to customize the entire security and authorization system to suit your needs.
  - Override only the configure method (HttpSecurity http).
- Disable `csrf` and `httpBasic` because they are enabled by default if you inherit from `WebSecurityConfigurerAdapter`.
- We will authorize the user by token, we need to create and store a session for him (specify STATELESS)
  - sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)

SecurityConfig class

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtFilter jwtFilter;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .httpBasic().disable()
                .csrf().disable()
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and()
                .authorizeRequests()
                .antMatchers("/admin/*").hasRole("ADMIN")
                .antMatchers("/user/*").hasAnyRole("ADMIN", "USER")
                .antMatchers("/signup", "/signin").permitAll()
                .and()
                .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

}
```

UserController class. New Methods

```
@PostMapping("/signin")
public TokenResponse signIn(
        @RequestParam(value = "login", required = true)
        String login,
        @RequestParam(value = "password", required = true)
        String password) {
    log.info("**/signin userLogin = " + login);
    UserRequest userRequest = new UserRequest(login, password);
    UserResponce userResponce = userService.findByLoginAndPassword(userRequest);
    return new TokenResponse(jwtProvider.generateToken(userResponce.getLogin()));
}

@GetMapping("/user/date")
public OperationResponce expirationDate() {
    log.info("**/user/date");
    return new OperationResponce("expiration date is " + userService.getExpirationLocalDate());
}

@GetMapping("/admin/roles")
public List<RoleResponce> listRoles() {
    log.info("**/admin/roles");
    return userService.getAllRoles();
}
```

Call REST API

softserve

Curl

- curl

  `https://curl.haxx.se/`
- curl for Windows (download)

  `https://curl.haxx.se/windows/`
- Using curl to automate HTTP jobs

  `https://curl.haxx.se/docs/httpscripting.html`
- How To Use curl

  `https://curl.haxx.se/docs/manpage.html`
- The curl guide to HTTP requests

  `https://flaviocopes.com/http-curl/`

softserve

Curl

- Perform an HTTP GET request

```
curl http://localhost:8080/
```

- Get the HTTP response headers

```
curl -i http://localhost:8080/
```

- Using HTTP authentication

```
curl -X GET http://localhost:8080/user/date
-H "accept: */*" -H "Authorization: Bearer eyJh...3DA "
```

```
curl -X GET http://localhost:8080/admin/roles
-H "accept: */*" -H "Authorization: Bearer eyJ...3DA"
```

Curl

- Perform an HTTP POST request passing data URL

```
curl -i -X POST --data "login=test&password=test“
http://localhost:8080/signup
curl -i -X POST --data "login=test&password=test“
http://localhost:8080/signin
curl -i -X POST --data "login=admin&password=admin“
http://localhost:8080/signin
```

- Perform an HTTP PUT request

```
curl -i -X PUT http://localhost:5000/api/tasks/101
```

- Perform an HTTP DELETE request

```
curl -i -X Delete http://localhost:5000/api/tasks/101
```

Call REST API Tools

- **Browser** (Get and Post methods only);
- **Postman**.
  - Launched initially as Chrome plugin, Postman has evolved to become a top-tier API testing tool. It is ideal for those who want to test APIs without coding in an integrated development environment using the same language as developers.
- **SoapUI**.
  - SoapUI is an API testing tool that is ideal for complicated test scenarios as it allows developers to test REST, SOAP, and Web Services without any hassles. It gives the user a full source framework as it is wholly dedicated to API testing.
- **Swagger**.
  - Swagger is an open-source software set of tools to design, build, document, and use RESTful web services.
- **Curl**.
  - Curl is a computer software project providing a library (`libcurl`) and command-line tool (`curl`) for transferring data using various network protocols.

THANKS

softserve