

Основе веб програмирања

Борисав Живановић (borisavz)

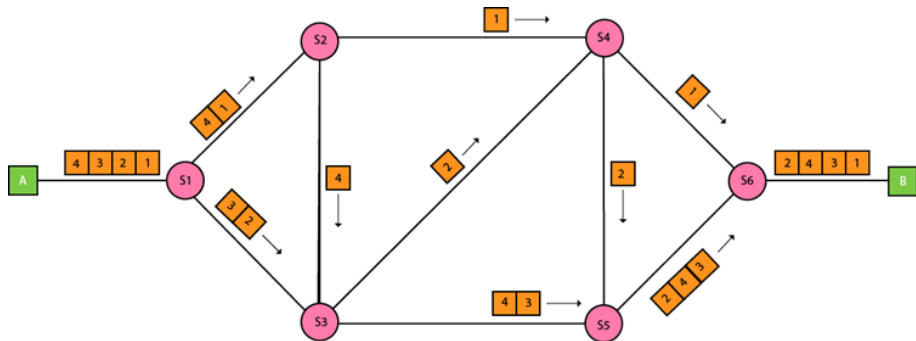
27. јануар 2023.

- 1 Основни појмови мрежног програмирања
- 2 Клијент-сервер архитектура
- 3 Еволуција веб апликација
- 4 HTTP протокол
- 5 Рад са базом података
- 6 Архитектура веб апликације
- 7 Аутентификација и ауторизација

Packet switching I

- Потребно је да поруку пошаљемо примаоцу
- Директна веза са сваким примаоцем није остварива
- Идеја: повезивање пошиљаоца/примаоца у мрежу, дељење комуникационог канала
- Решење: **комутација пакета (packet switching)**
 - Поруку изделимо на пакете
 - Пакетима додамо заглавље (header) са адресом пошиљаоца и примаоца
 - Систем зна путање до примаоца
 - Поруку шаљемо пакет по пакет
 - Само један пакет заузима комуникациони канал
 - Пакети могу да путују различитим путањама кроз мрежу, да дођу у различитом редоследу до примаоца, или да нестану

Packet switching II

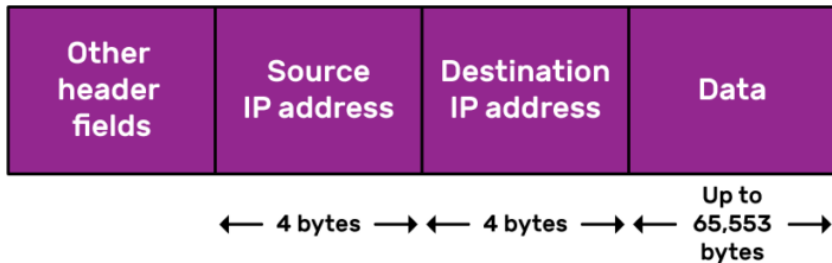


Слика: комутација пакета (packet switching)

Internet Protocol I

- Како би комуницирали у мрежи, потребно је да сваки учесник у комуникацији има додељену **јединствену** адресу
- Поруци придружујемо **заглавље (header)** које садржи:
 - Адресу пошиљаоца (source address)
 - Адресу примаоца (destination address)
 - Додатна поља (верзија IP протокола, flags, TTL, checksum, ...)
- Захваљујући овом заглављу систем зна коме да проследи поруку
- У одговори су адресе пошиљаоца и примаоца **заменењене!**

Internet Protocol II

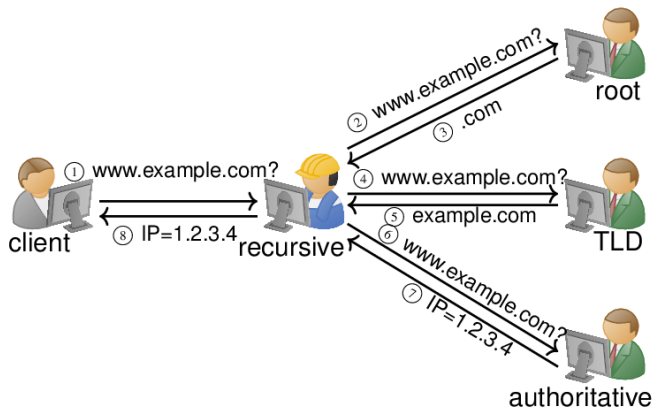


Слика: упрошћена структура IP пакета

DNS I

- Проблем: све више сервера на мрежи
- Није практично памтити сваку адресу у бројчаном облику
- Идеја: систем за придруживање имена, сличан телефонском именику
- Решење: **DNS (Domain Name System)**
 - IP адреси додељујемо симболичко име (домен)
 - Домени су хијерархијски (структура стабла)
 - DNS је одговоран за одређени део хијерархије
 - Као одговор враћа IP адресу или адресу одговорног DNS сервера
 - Морамо знати IP адресу DNS сервера!

DNS II



Слика: DNS упит

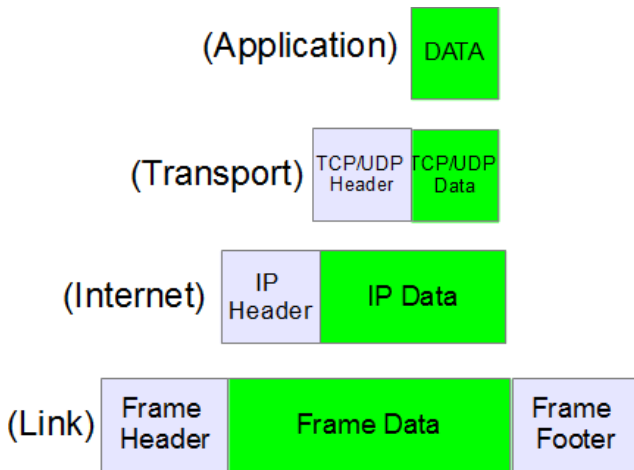
Transmission Control Protocol I

- Решили смо проблем адресирања уређаја на мрежи...
- ...али нисмо проблеме редоследа пристиглих пакета и нестајања пакета
- Додатни проблем: шта ако имамо више мрежних апликација на истом рачунару, како да проследимо поруку одговарајућој апликацији?

Transmission Control Protocol II

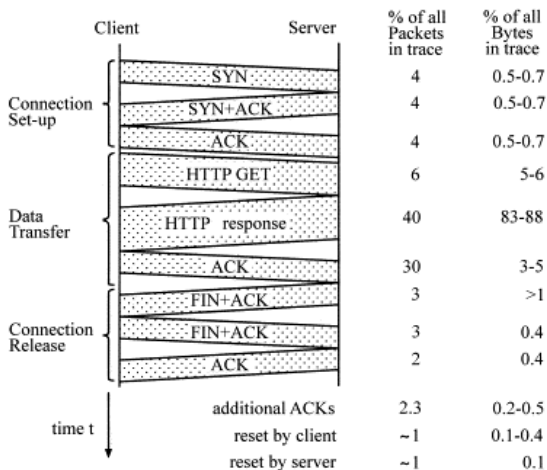
- Решење: **TCP (Transmission Control Protocol)**
 - Додајемо додатно заглавље на нашу поруку
 - Заглавље садржи source и destination port (слично адреси пошilhaоца и примаоца, али се односи на апликацију), sequence number (редослед поруке)
 - Уколико пакет нестане, шаље се поново
 - Оперативни систем осигурава да само једна апликација користи одређени порт

Transmission Control Protocol III



Слика: енкапсулација пакета

Transmission Control Protocol IV

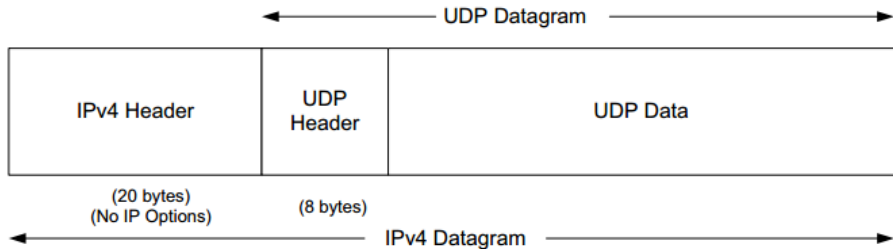


Слика: Ток TCP комуникације

User Datagram Protocol I

- Успостављање конекције траје одређено време
- За поруке које стају у један пакет, можемо користити једноставнији **UDP (User Datagram Protocol)**
- Задржавамо адресирање апликација, али губимо гаранцију испоруке
- DNS користи UDP
- Омогућава изградњу протокола који имају гаранције испоруке
 - пример: HTTP3/QUIC

User Datagram Protocol II



Слика: енкапсулација пакета

User Datagram Protocol III



Слика: Садржај заглавља

Однос између чворова

- До сада смо говорили искључиво о чворовима који учествују у комуникацији
- Видели смо да један чвор започиње комуникацију, а други даје одговор
- У раду уочавамо две врсте односа између чворова:
 - **peer-to-peer**: обе стране су подједнако важне у комуникацији
 - **client-server**: клијент се обраћа серверу за податке или обављање акције

Клијент-сервер архитектура I

- Модел настао још раних дана рачунарства
- Рачунари су били велики и скупи
- Било је потребно омогућити дељење ресурса између више корисника
- Клијенти су били далеко једноставнији, главна намена је била слање команде и испис резултата
- Данас је овај приступ познат као **thin-client**

Клијент-сервер архитектура II



Слика: PDP-7 (рачунар)

Клијент-сервер архитектура III



Слика: DEC VT100 (терминал)

Клијент-сервер архитектура IV

- Кроз године, рачунарска моћ је расла
- Ово је довело до појаве **PC (Personal Computer)**
 - користи се непосредно
 - без конукурентних корисничких сесија
- Потреба за централним сервером и даље није потпуно избачена, али је могућа далеко већа интерактивност
- Данас је овај приступ познат као **thick-client**
 - пример: Google Docs

Еволуција веб апликација I

- **World Wide Web (WWW)** је изумео Тим Бернерс-Ли у CERN-у
- Оригинална замисао је била систем за дељење докумената
- Језик докумената: **HTML (HyperText Markup Language)**
- Протокол за комуникацију: **HTTP (HyperText Transfer Protocol)**
- Иницијално садржај је био статички (могуће је прегледање искључиво предефинисаних докумената)
- Убрзо су уочени недостаци и настала је потреба за динамичким садржајем

Еволуција веб апликација II

- Идеја: чувати садржај у бази података и на основу њега динамички генерисати HTML документе
- Постоје два решења:
 - **server-side render**: HTML документ генеришемо користећи шаблон и вредности из базе података
 - **client-side render**: са сервера учитавамо основни HTML и JavaScript код, након тога размењујемо JSON објекте

Еволуција веб апликација III

```
<!DOCTYPE html>
<html>

<head>
  <title>Page Title</title>
</head>

<body>
  <h2>Heading Content</h2>
  <p>Paragraph Content</p>
</body>

</html>
```

Слика: HTML документ

Еволуција веб апликација IV

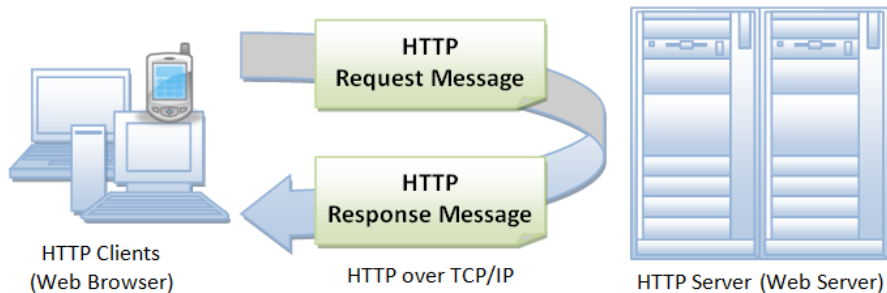
```
1  {  
2    "string": "Hi",  
3    "number": 2.5,  
4    "boolean": true,  
5    "null": null,  
6    "object": { "name": "Kyle", "age": 24 },  
7    "array": ["Hello", 5, false, null, { "key": "value", "number": 6 }],  
8    "arrayOfObjects": [  
9      { "name": "Jerry", "age": 28 },  
10     { "name": "Sally", "age": 26 }  
11  ]  
12 }  
13
```

Слика: JSON објекат

HTTP протокол

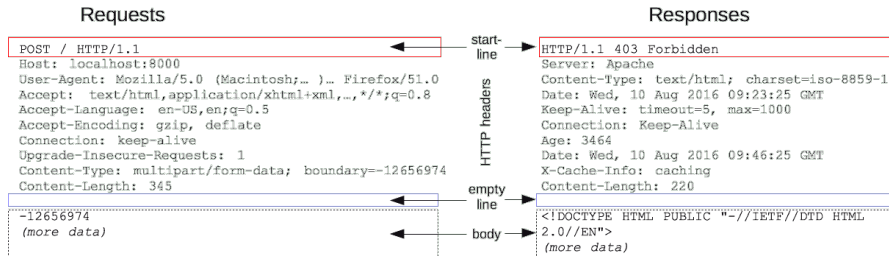
- Текстуални протокол (поруке једноставно могу да читају и људи)
- Користи TCP (гаранција испоруке је неопходна како би протокол успешно функционисао!)
- Подразумевани порт: 80 (HTTP), 443 (HTTPS)
- Stateless протокол
 - неопходно је придружити додатне информације уз сваки захтев како би пратили корисничку сесију
 - обично преко header-a
- Путања означава **ресурс** у систему
 - додатни атрибути кроз **query params**
- Метода означава **акцију** коју желимо да извршимо над ресурсом
- Статусни код означава да ли је акција успешно извршена, и ако није, разлог

Request-response I



Слика: Request-response модел

Request-response II

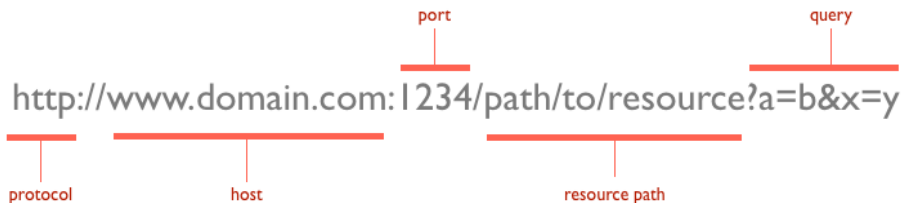


Слика: Садржај request и response порука

Request-response III

- Формирамо HTTP захтев (string)
- Извршавамо DNS упит како би добили IP адресу сервера
 - могуће је и кеширање DNS одговора на клијентској страни
- Успостављамо TCP конекцију са сервером (подразумевани или наведени порт)
- Захтев шаљемо издељен у пакете
- Чекамо одговор
 - клијенти обично постављају timeout
- Затварамо TCP конекцију
 - потенцијално уско грло уколико у кратком временском периоду шаљемо више захтева
 - исправљено у наредним верзијама протокола

Request-response IV



Слика: URL

HTTP методе

- **GET**: добављање ресурса из система
- **PUT**: измена постојећег ресурса у целости
- **POST**: додавање новог ресурса у систем
- **PATCH**: измена дела постојећег ресурса
- **DELETE**: брисање ресурса из система

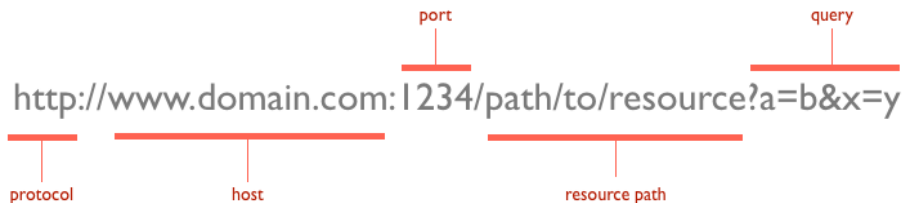
Status codes

- **1xx:** информациони одговор
 - 100 Continue, 101 Switching Protocols, 103 Early Hints, ...
- **2xx:** успешан одговор
 - 200 OK, 201 Created, 202 Accepted, ...
- **3xx:** редирекција
 - 301 Moved Permanently, 302 Found, ...
- **4xx:** грешка клијента
 - 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 405 Method Not Allowed, 415 Unsupported Media Type, 422 Unprocessable Entity, ...
- **5xx:** грешка сервера
 - 500 Internal Server Error, 501 Not Implemented, 502 Bad Gateway, 503 Service Unavailable, 505 HTTP Version Not Supported, ...

Архитектура веб апликације I

- Потребно је да омогућимо комуникацију преко HTTP
- И да комуницирамо са базом како би извршавали упите
- Једну акцију може да чини више упита ка бази
- Потребно је запис у бази представити структуром података у жељеном програмском језику
- ...и то су, у суштини, компоненте веб апликације

Архитектура веб апликације II



Слика: шематски приказ архитектуре

Controller

- Садржај HTTP захтева претвара у структуру података
- Позива методу из сервисног слоја
- Резултат добијен позивом сервисног слоја претвара у HTTP одговор
- Може да садржи логику за ауторизацију
- Упозорење: грешка коју шаљемо клијенту не сме да открива интерне детаље

Пример имплементације: Go/Gorilla Mux

```
func (c *AuthController) VerifyRegistration(w http.ResponseWriter, req *http.Request) {
    ctx, span := c.tracer.Start(req.Context(), "AuthController.VerifyRegistration")
    defer span.End()

    verificationId := mux.Vars(req)["verificationId"]

    appErr := c.authService.VerifyRegistration(ctx, verificationId)
    if appErr != nil {
        span.SetStatus(codes.Error, appErr.Error())
        http.Error(w, appErr.Message, appErr.Code)
        return
    }
}
```

Пример имплементације: Java/Spring

```
@PostMapping("/{postId}/comments")
@IsLoggedIn
public long addPostComment(
    @PathVariable long postId ,
    @RequestBody @Valid CreateCommentDTO createCommentDTO
) {
    return postService.addPostComment(postId , createCommentDTO);
}
```

Service

- Садржи пословну логику апликације
- Једна сервисна метода се састоји из позива једне или више метода из repository
- Уколико база података подржава трансакције, сервисна метода је граница трансакције
 - **commit** уколико је акција успешна
 - **rollback** уколико је акција неуспешна
- Садржи комплетне провере права приступа
 - чест шаблон је да извршимо упит који проверава да ли корисник има право приступа (рецимо, чланство на пројекту), и у зависности од резултата извршимо акцију

Пример имплементације: Go/Gorilla Mux

```
func (s *AuthService) VerifyRegistration(ctx context.Context, verificationId string) *app_errors.AppError {
    serviceCtx, span := s.tracer.Start(ctx, "AuthService.VerifyRegistration")
    defer span.End()

    username, err := s.authRepository.GetVerification(serviceCtx, verificationId)
    if err != nil {
        span.SetStatus(codes.Error, err.Error())
        return &app_errors.AppError{500, ""}
    }

    user, err := s.authRepository.GetUser(serviceCtx, username)

    user.Enabled = true

    err = s.authRepository.SaveUser(serviceCtx, user)
    if err != nil {
        span.SetStatus(codes.Error, err.Error())
        return &app_errors.AppError{500, ""}
    }

    err = s.authRepository.DeleteVerification(serviceCtx, verificationId)
    if err != nil {
        span.SetStatus(codes.Error, err.Error())
        return &app_errors.AppError{500, ""}
    }

    return nil
}
```

Пример имплементације: Java/Spring

```
@Transactional
public void downvotePost(long postId) {
    Post post = postRepository.getById(postId);
    User user = userRepository.getById(authUser().getId());

    if (post.getCommunity().isUserBanned(user))
        throw new NotAllowedToParticipateException();

    reactionRepository.deletePostReactionByUser(authUser().getId(), postId);

    Reaction reaction = new Reaction();

    reaction.setMadeBy(user);
    reaction.setPost(post);
    reaction.setType(ReactionType.DOWNVOTE);

    reactionRepository.save(reaction);
}
```

Repository

- Једна метода представља један упит над базом података
- Параметре прослеђује у упит
 - подсетник: потребно је да се одбранимо од injection напада!
- Резултат упита претвара у одговарајуће структуре података
 - **entity** уколико враћамо записе из базе неизмењене
 - **DTO** уколико упит садржи комплекснија пресликавања (пример: генерисање извештаја)
- У зависности од коришћене базе података/библиотеке, логику за конверзију резултата упита морамо ручно да имплементирамо, или библиотека то чини аутоматски

Пример имплементације: Go/Gorilla Mux

```
func (r *ConsulAuthRepository) DeleteUser(ctx context.Context, username string) error {  
    _, span := r.tracer.Start(ctx, "ConsulAuthRepository.DeleteUser")  
    defer span.End()  
  
    kv := r.cli.KV()  
  
    userKey, err := r.constructKey("user/%s/", username)  
    if err != nil {  
        span.SetStatus(codes.Error, err.Error())  
        return err  
    }  
  
    err = kv.Delete(userKey, nil)  
    if err != nil {  
        span.SetStatus(codes.Error, err.Error())  
        return err  
    }  
  
    return nil  
}
```

Пример имплементације: Java/Spring

```
@Modifying
@Query(value = "" +
        "␣DELETE␣FROM␣reaction" +
        "␣WHERE␣made_by_id=␣?1" +
        "␣AND␣post_id=␣?2",
        nativeQuery = true)
void deletePostReactionByUser(long userId, long postId);
```

Entity

- Представља записе у бази података
 - додатно: везе ка другим ентитетима
- Омогућује објектно-релационо мапирање
- Може да садржи бизнис логику
 - тема активне дебате

Пример имплементације: Go/Gorilla Mux

```
type User struct {  
    Username    string    'json:"username" '  
    PasswordHash string    'json:"passwordHash" '  
    Email       string    'json:"email" '  
    Role        string    'json:"role" '  
    Enabled     bool      'json:"enabled" '  
}
```

Пример имплементације: Java/Spring

```
@Getter
@Setter
@Entity
@EqualsAndHashCode(of = "id")
@SQLDelete(sql = "UPDATE post SET deleted=true WHERE id=?")
@Where(clause = "deleted=false")
public class Post {

    @Id
    @GeneratedValue
    private long id;

    private String title;
    private String text;
    private LocalDate creationDate;

    private long imageId;

    @ManyToOne(fetch = FetchType.EAGER)
    private User postedBy;

    @ManyToOne(fetch = FetchType.EAGER)
    private Community community;

    @ManyToOne(fetch = FetchType.EAGER)
    private Flair flair;

    private boolean deleted;
}
```

Data Transfer Object (DTO)

- Проблем: ентитети потенцијално нису погодни за слање клијенту
- Идеја: применити принцип енкапсулације, трансформација одговора у погодан формат
- Ова компонента је опциона, и често није неопходна
- Могуће је и комбиновање уз entity

Пример имплементације: Go/Gorilla Mux

```
type RegisterUser struct {  
    Username    string 'json:"username" validate:"required" '  
    Password    string 'json:"password" validate:"required,password" '  
    Email       string 'json:"email" validate:"required,email" '  
    FirstName   string 'json:"firstName" validate:"required" '  
    LastName    string 'json:"lastName" validate:"required" '  
    Town        string 'json:"town" validate:"required" '  
    Gender      string 'json:"gender" validate:"required" '  
    CaptchaToken string 'json:"captchaToken" validate:"required" '  
}
```

Пример имплементације: Java/Spring

@Getter

@Setter

```
public class CommentDTO {
```

```
    private long id;
```

```
    private String text;
```

```
    private LocalDate timestamp;
```

```
    private long postId;
```

```
    private List<CommentDTO> replies;
```

```
    private UserDTO writtenBy;
```

```
    private ReactionType reaction;
```

```
    private int karma;
```

```
}
```


Middleware

- Често желимо да централизујемо логику која је потребна пре/после извршавања (већине или свих) метода из контролера
 - валидација токена за ауторизацију
 - праћење информација за logging/tracing
- Математички посматрано, одговара композицији функције
- У програмским језицима који имају first-class функције (пример: JavaScript, Go) се имплементира као композиција функција
- Уколико то није подржано, имплементира се механизмом који то опонаша (пример: Java/Aspect Oriented Programming)
- Пресрећемо захтев, прослеђујемо га даље или прекидамо ланац

Пример имплементације: Go/Gorilla Mux

```
func ExtractJWTUserMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if authHeader, ok := r.Header["Authorization"]; ok {
            tokenString := authHeader[0]

            token, err := jwt.Parse(tokenString, func(token *jwt.Token) (interface{}, error) {
                return []byte(os.Getenv("SECRET_KEY")), nil
            })

            if claims, ok := token.Claims.(jwt.MapClaims); ok && token.Valid {
                authUser := model.AuthUser{
                    Username: claims["username"].(string),
                    Role:     claims["role"].(string),
                    Exp:      time.UnixMilli(int64(claims["exp"].(float64))),
                }

                authCtx := context.WithValue(r.Context(), "authUser", authUser)

                next.ServeHTTP(w, r.WithContext(authCtx))
            } else {
                http.Error(w, "Invalid token", 401)
            }
        } else {
            next.ServeHTTP(w, r.WithContext(newCtx))
        }
    })
}
```

Пример имплементације: Java/Spring

@Override

```
protected void doFilterInternal(HttpServletRequest request ,
                                HttpServletResponse response ,
                                FilterChain chain)
    throws ServletException , IOException {
    final String token = request.getHeader(HttpHeaders.AUTHORIZATION);
    if (isEmpty(token)) {
        chain.doFilter(request , response);
        return;
    }

    if (!jwtTokenUtil.validate(token)) {
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        return;
    }

    User user = userRepository.findByUsername(jwtTokenUtil.getUsername(token)).orElse(null);
    UserDetails userDetails = user == null ? null : new AuthUserDetails(user);

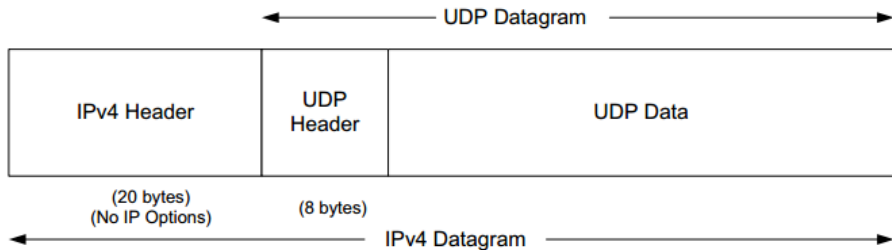
    UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(
        userDetails ,
        null ,
        userDetails == null ? List.of() : userDetails.getAuthorities()
    );

    SecurityContextHolder.getContext().setAuthentication(authentication);
    chain.doFilter(request , response);
}
```

Индирекција I

- Било који проблем у рачунарству може бити решен још једним нивоом индирекције, осим наравно проблема превише индирекција (David J. Wheeler)
- Индирекција омогућава имплементацију контроле приступа
- Извршавање **акције** мора да одобри **посредник** који дефинише правила приступа
- Механизам присутан на свим нивоима апстракције
 - енкапсулација у ООП, x86 protection rings, системски позиви, изолација процеса, **бизнис логика**

Индирекција II



Слика: шематски приказ индирекције

Основни појмови

- **Идентификација:** процес приписивања идентитета човеку или другом рачунару
 - регистрација корисничког налога
- **Аутентификација:** процес провере идентитета
 - пријављивање на кориснички налог
- **Ауторизација:** утврђивање права која корисник има над ресурсима у систему
 - провере права приступа у апликацији (middleware/controller/service)

Role Based Access Control: концепт

- Корисник има улогу, улога има дозволе
 - улога одговара радном месту у фирми или типу налога (обичан/администраторски)
 - дозвола одговара акцији у систему
- Улога додељена кориснику се (релативно) ретко мења
 - промена радног места
- Кроз време, могућа је промена дозвола додељених улогама

Role Based Access Control: имплементација

- Уз корисника, у бази података чувамо његову улогу
- Дозволе се најчешће не чувају, већ се провере имплементирају ручно у middleware/controller
- Улога се чува у **access token**

Attribute Based Access Control: концепт

- RBAC је погодан за статичке дозволе, али је веома непогодан за динамичке
 - пример: само члан сме да приступи пројекту, преузимање видео игре је дозвољено старијима од 16 година
- Функција $f(Attr) \rightarrow Bool$ одређује да ли корисник има дозволу да обави акцију
- $Attr$ се састоји од тренутног стања система
 - што значи да $f(Attr)$ није детерминистичка функција!

Attribute Based Access Control: имплементација

- Уз записе у бази чувамо атрибуте који су потребни за одређивање права приступа
- Атрибути могу да представљају везу између корисника и заштићеног ресурса (пример: листа чланова пројекта) или да буду везани директо за заштићени ресурс (пример: старост потребна за преузимање игре)
- Провере се обављају у сервисном слоју
 - уколико се $f(Attr)$ евалуира у *False*, враћамо грешку **403 Forbidden**
- Обично захтева додатни упит над базом података

Основни ток



Складиштење лозинки I

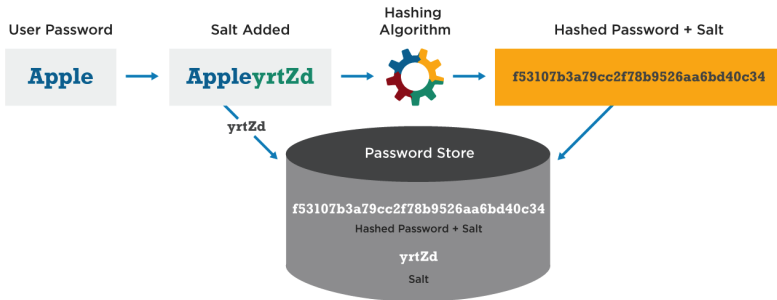
- Најједноставнији начин је складиштење лозинке у отвореном тексту
 - уколико нападач дође у посед лозинки, може да се несметано пријави у нашу, а вероватно и остале апликације
- Нешто боље је складиштење *hash*-а лозинке
$$hash = HashFunc(pass)$$
 - исте лозинке имају исти *hash* ($HashFunc(pass)$) је детерминистичка функција)
 - могуће је извести dictionary/brute force напад и тиме компромитовати исте лозинке

Складиштење лозинки II

- Најбоље је складиштење *salted hash*-а лозинке
$$\text{salted_hash} = \text{HashFunc}(\text{pass} + \text{salt})$$
 - *salt* је насумична вредност која се складишти уз лозинку
 - две исте лозинке ће због тога имати различиту *salted_hash* вредност, па је dictionary/brute force напад потребно извести одвојено за сваку лозинку
- На жалост, и даље има доста апликација које лозинке складиште у отвореном тексту, што нас чини рањивим

Складиштење лозинки III

Password Hash Salting



Слика: шематски приказ *salted hash-a*