Основе веб програмирања

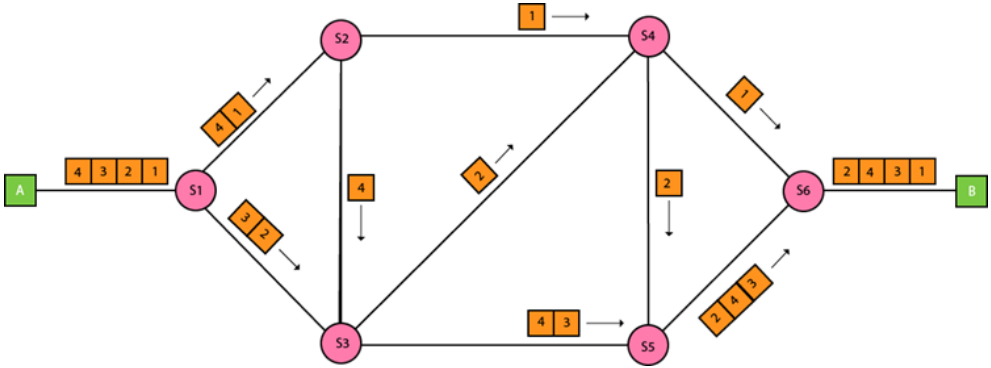
Борисав Живановић

2. фебруар 2023.

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

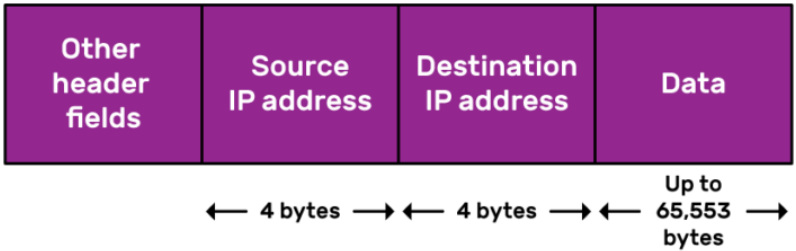


Слика: комутација пакета (packet switching)

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 **замењене**!

Internet Protocol II

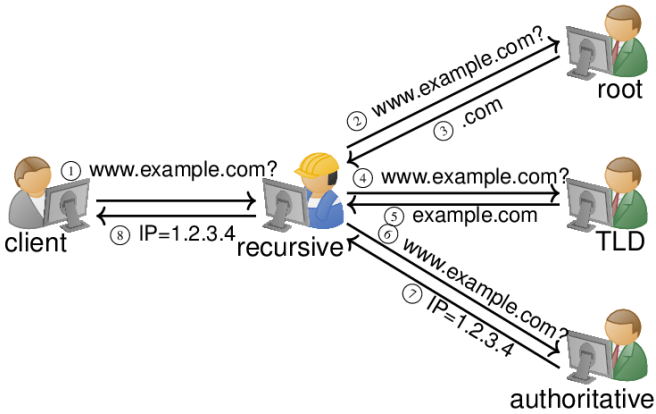


Слика: упрошћена структура IP пакета

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

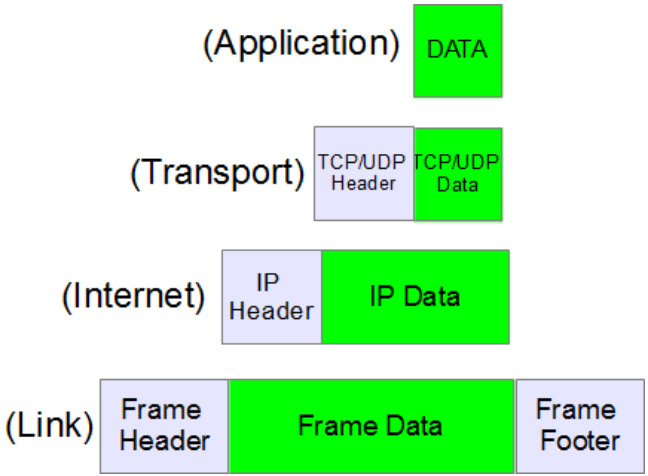
Transmission Control Protocol I

- 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

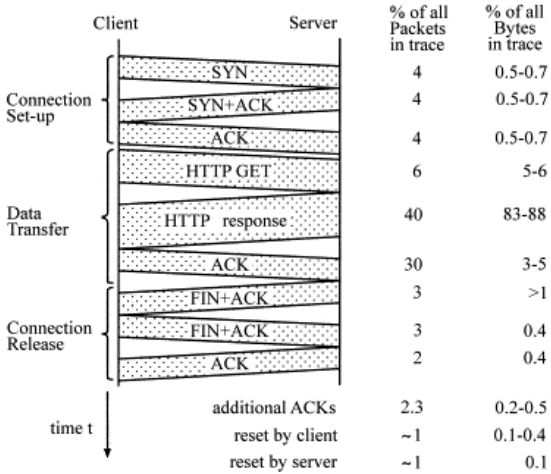
- Решење: **TCP (Transmission Control Protocol)**
  - Додајемо додатно заглавље на нашу поруку
  - Заглавље садржи source и destination port (слично адреси пошиљаоца и примаоца, али се односи на апликацију), sequence number (редослед поруке)
  - Уколико пакет нестане, шаље се поново
  - Оперативни систем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

Transmission Control Protocol III



Слика: енкапсулација пакета

Transmission Control Protocol IV

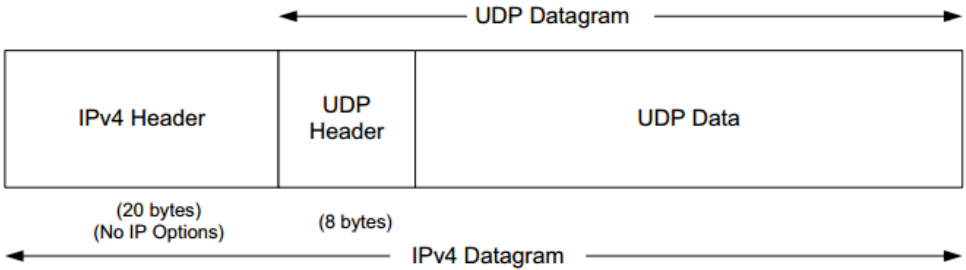


Слика: Ток TCP комуникације

User Datagram Protocol I

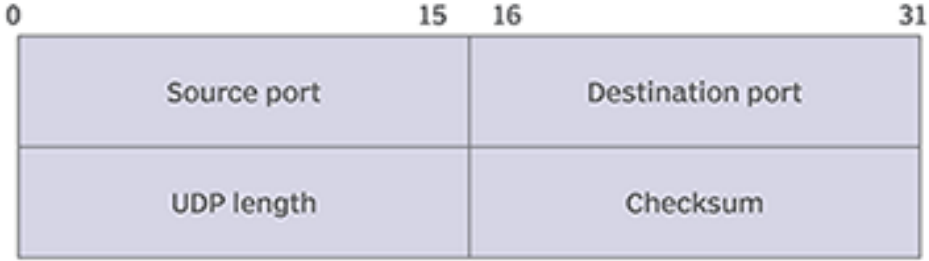
- Успостављање конекције траје одређено време
- За поруке које стају у један пакет, можемо користити једноставнији **UDP (User Datagram Protocol)**
- Задржавамо адресирање апликација, али губимо гаранцију испоруке
- DNS користи UDP
- Омогућава изградњу протокола који имају гаранције испоруке
  - пример: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

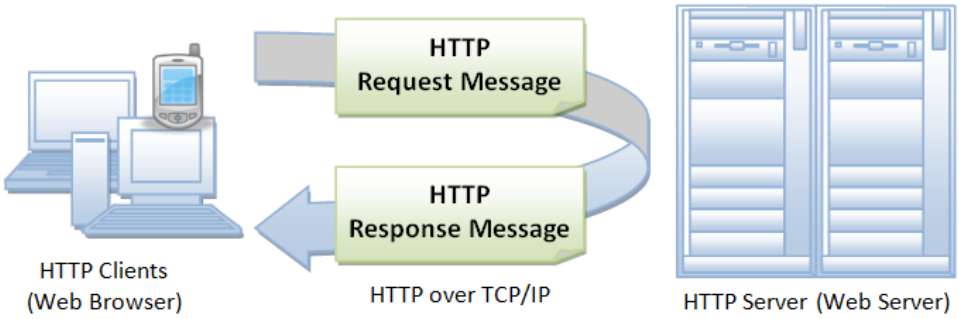
```json
{
  "string": "Hi",
  "number": 2.5,
  "boolean": true,
  "null": null,
  "object": { "name": "Kyle", "age": 24 },
  "array": ["Hello", 5, false, null, { "key": "value", "number": 6 }],
  "arrayOfObjects": [
    { "name": "Jerry", "age": 28 },
    { "name": "Sally", "age": 26 }
  ]
}

```

Слика: JSON објекат

HTTP протокол

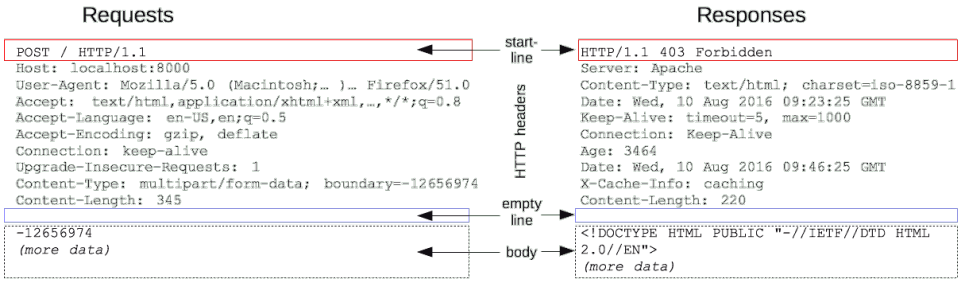
- Текстуални протокол (поруке једноставно могу да читају и људи)
- Користи TCP (гаранција испоруке је неопходна како би протокол успешно функционисао!)
- Подразумевани порт: 80 (HTTP), 443 (HTTPS)
- Stateless протокол
  - неопходно је придружити додатне информације уз сваки захтев како би пратили корисничку сесију
  - обично преко header-а
- 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 изврешна, и ако није, разлог

Request-response I



Слика: Request-response модел

Request-response II

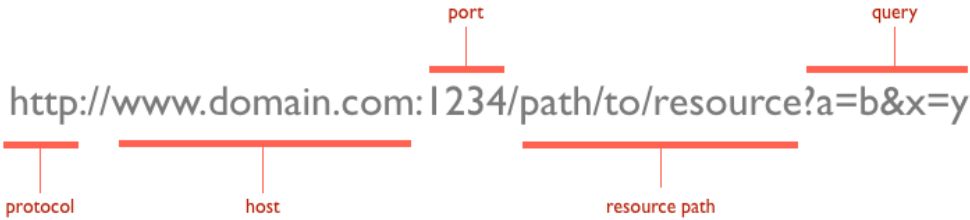


Слика: Садржај **request** и **response** порука

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 ресрса 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, ...

База података

- База података нам омогућује централизовање логике за чување, приступ и измену података, како би се ефикасније посветили развоју апликативне логике
- Грубо гледано, база података је софтвер за читање фајлова
  - ...али тај формат је комплекснији од CSV, обично варијације B-Tree или LSM Tree
- Интеракција се обавља кроз **упитни језик**
- Могућа интеракција кроз алате за администрацију (пример: DBeaver) или кроз библиотеке жељеног програмског језика
- Аутентификација и ауторизација је подржана

Модел података

- Модел података представља ентитете из реалног система и везе међу њима
- Модел је увек апроксимација реалног система!
  - због тога бирамо онај модел података који најбоље одговара нашем систему
- Данас актуелни: релациони, граф, key-value, document, time series, wide column
- У већини случајева, релациони модел је погодан
- Примери изузетака:
  - друштвене мреже је најприродније представити као граф
  - чување фајла/низа бајтова је најједноставније у key-value

Упитни језик

- Упитни језици су уско везани за математичке формализме иза модела података
  - SQL: релациона алгебра, Neo4J Cypher: обилазак графа
- Спадају у групу **језика специфичних за домен (Domain Specific Languages, DSL)**
  - замисао: језиком описујемо коју акцију желимо, док извршавање конкретних корака препуштамо систему
  - кораци потребни за извршавање упита се називају **query plan**
- **Data Definition Language**: креирање и измена шеме базе података, креирање индекса
- **Data Manipulation Language**: Create, Read, Update, Delete (CRUD) операције

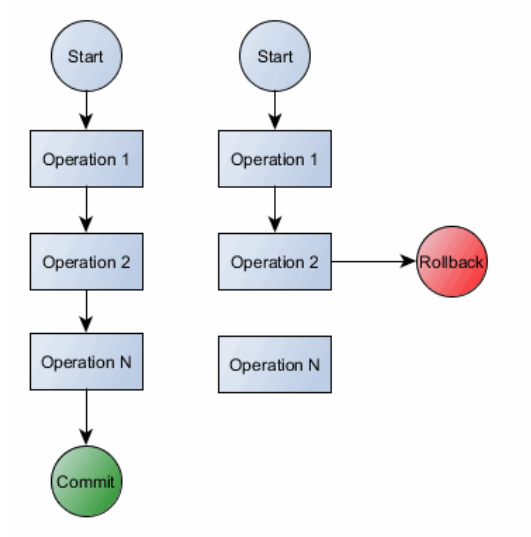
Трансакције I

- Чест случај је да извршавање једне корисничке акције може да захтева измене над више записа, који су често у различитим табелама
- Проблем: грешка у сред акције може да податке остави у неконзистентном стању
- Решење: свака измена података се извршава у трансакцији
  - **commit**: трајно сачувати измене уколико су све успешно извршене
  - **rollback**: трајно одбацити све измене први првој неуспешној
- Трансакција пресликава једно валидно стање у друго валидно стање
  - дозвољена стања се називају инваријанте
  - дефинисана су ограничењима (PRIMARY KEY, FOREGIN KEY, CHECK)

Трансакције II

- Трансакције морају да задовољавају **ACID** скуп особина:
  - **Atomicity**: извршавају се све акције, или ниједна
  - **Consistency**: измене морају да произведу валидно стање
  - **Isolation**: трансакција не мора да буде свесна других конкурентних трансакција
  - **Durability**: измене остају трајно сачуване уколико је успешан резулат потврђен
- Уочено је да комлетна изолација видно обара перформансе, те је ниво изолације могуће подесити на нивоу трансакције
  - потенцијално уноси аномалије при читању
  - нивои изолације нису адекватно стандардизовани, неопходно је консултовати се са документацијом!

Трансакције III



Нормализација и денормализација I

- Замисао нормализације је спречавање аномалија до којих долази приликом измене дуплираних података
  - кажемо да је база података нормализована уколико задовољава математичку дефиницију треће нормалне форме
- Замисао моделовања релационе базе података је да уколико добро измоделујемо објекте у систему, можемо извршити било који упит над њима
- Цена овог приступа је да поједини упити могу бити изузетно скупи (превише JOIN операција)
- У денормализованом моделу није потребно извршавање JOIN

Нормализација и денормализација II

- Жртвујемо нешто спорији упис ради далеко бржег читања
- Класичан пример нерешив у нормализованом моделу података: feed на друштвеним мрежама
  - замислите три-четири JOIN-а над гигантским скуповима
- Моделовање већине NoSQL база података захтева познавање упита унапред, што их чини непогодним за ране фазе развоја
- Добра пракса: започети са нормализованим моделом података, након уочавања уских грла и честих упита, денормализовати модел података по потреби

Нормализација и денормализација III

Слика: нормализован модел података

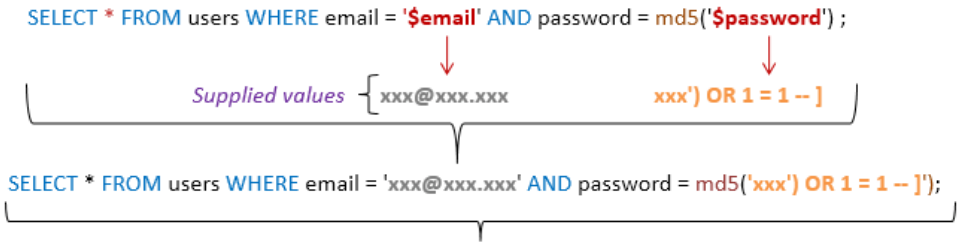
Нормализација и денормализација IV

Слика: денормализован модел података

Injection напади I

- Улаз добијен од корисника је ван наше контроле и зато потенцијална опасност!
- До овог проблема долази уколико у упит наивно додамо параметре кроз конкатенацију стрингова
- То омогућава нападачу да, уз познавање коришћене базе података, упит измени и тако изврши произвољан код
  - обрише табелу/базу података, заобиђе правила филтрирања, добави целу табелу...
- Решење: коришћење искључиво параметризованих упита, никада конкатенације стрингова

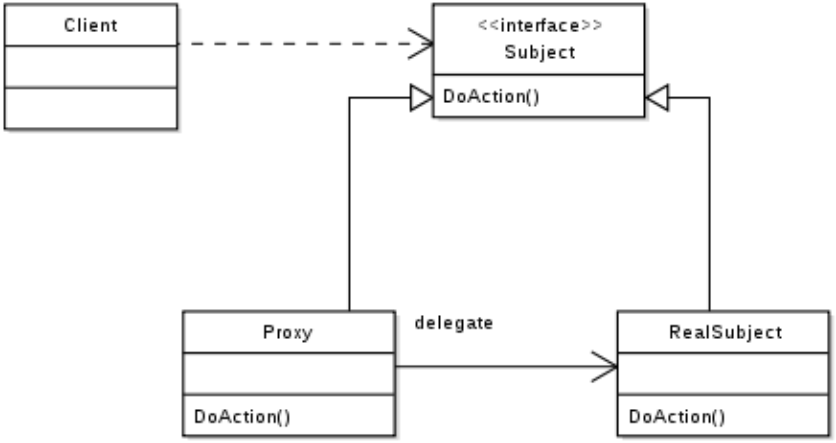
Injection напади II



Објектно-релационо мапирање I

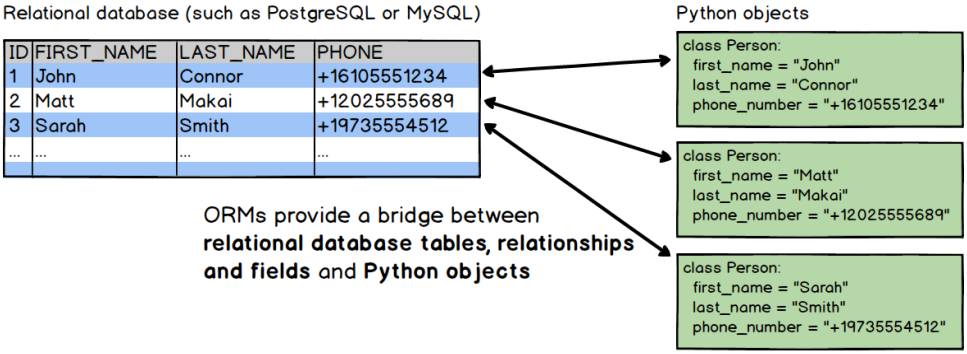
- Писање упита за релативно једноставне измене или добављање везаних ентитета може да буде напорно
- Идеја: интеракција са објектним моделом у жељеном програмском језику се у позадини конвертује у одговарајуће упите
- Имплементација помоћу **proxy pattern**-а
  - наше класе анотирамо како би их ORM библиотека препознала, добијамо динамички креиран **proxy** објекат
  - **getter** учитава везане ентитете, уколико нису учитани
- Чест шаблон: добављање преко упита, чување измена преко ORM
- Упозорење: неопрезно **eager** добаваљање везаних ентитета може да озбиљно наруши перформансе и стабилност!

Објектно-релационо мапирање II



Слика: Proxy pattern

Објектно-релационо мапирање III

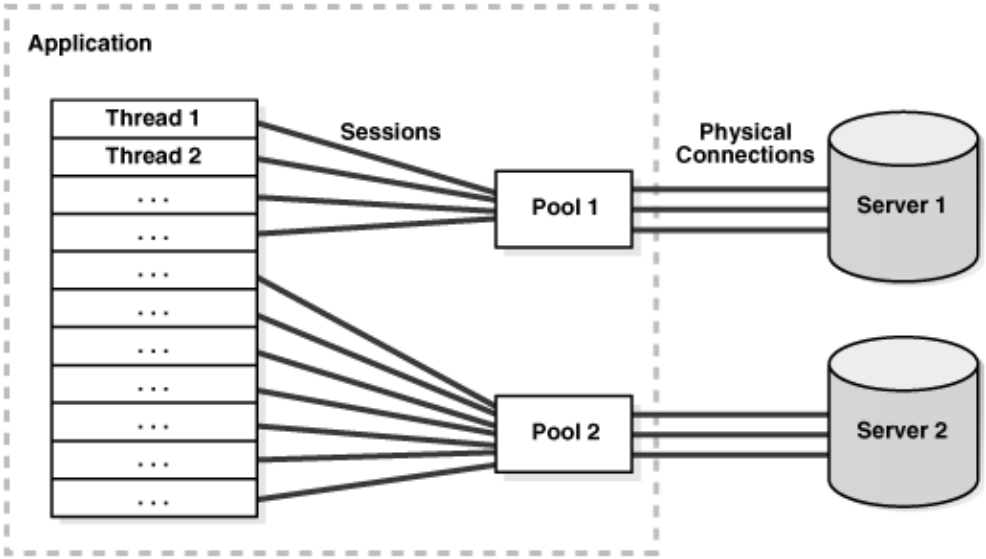


Connection pooling I

- За извршавање упита над базом, неопходна је комуникација кроз успостављену конекцију
- Једноставан приступ би био да успоставимо концекцију, извршимо упит и затим затворимо конекцију
- Како упостављање конекције траје одређено време, а често извршавамо више упита у релативно блиском временском периоду, ово видно обара перформансе
- Решење: кеширање концекција
  - конекцију никада не креирамо директно, већ је добављамо из pool-а
  - конекцију остављамо отворену одређени временски период
  - pool чува више конекција због паралелног опслуживања клијената

Connection pooling II

Connection pooling III



Connection pooling IV

Миграција базе података I

- Кроз време, наша апликација доживљава измене, а оне узрокују промену модела података
  - додавање и уклањање табела и поља, нормализација/денормализација модела података
- Једини начин да ово извршимо је покрентањем **скрипти за миграцију**
- Ово је могуће аутоматизовати тако да се скрипте изврше аутоматски при покретању нове верзије апликације
  - додатно: верзионирање измена, upgrade/downgrade модела података
- Алати: Java/Liquibase, Go/migrate

Миграција базе података II

- Математички посматрано, миграција представља пресликавање између алгебарских структура
  - старе и нове верзије базе података
- Ово пресликавање не мора да буде инјективно!
  - односно, може доћи до губитка података, због чега не постоји инверзно пресликавање које би вратило претходну верзију
- Пример: бришемо табеле, поља, или записе
- Због тога је неопходно са великом пажњом писати и тестирати скрипте за миграцију
  - потенцијално: прављење резервних копија уколико су измене ризичне

Миграција базе података III

```
--liquibase formatted sql

--changeset d_shilko:add_usertable
--labels: version1
create table liqui_schema.user(
    id int generated by default as identity primary key,
    firstname varchar(255),
    lastname varchar(255)
    );

--rollback drop table liqui_schema.user;

--changeset d_shilko:insert_users
--labels: version2
insert into liqui_schema.user (firstname, lastname) VALUES ( firstname: 'John',  lastname: 'Petrov');
insert into liqui_schema.user (firstname, lastname) VALUES ( firstname: 'Pamella',  lastname: 'Anderson');

--rollback delete from liqui_schema.user where lastname = 'Petrov';
--rollback delete from liqui_schema.user where lastname = 'Anderson';

--changeset d_shilko:plus_one_user
--labels: version2
insert into liqui_schema.user (firstname, lastname) VALUES ( firstname: 'Google',  lastname: 'Google');
--rollback delete from liqui_schema.user where lastname = 'Google';
```

3 rows | Tx: Auto | DDL | DML | CSV | data

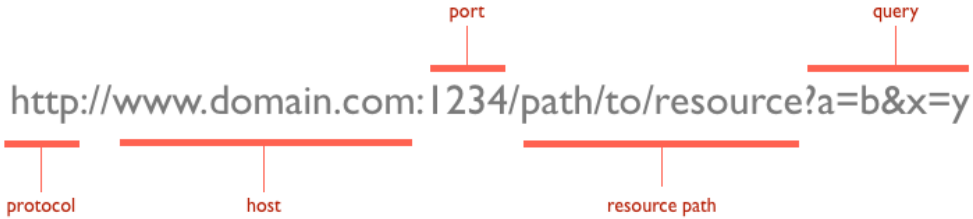
<Filter Criteria>

	id	labels	author	filename	dateexecuted	orderexecuted
1	add_usertable	<null>	d_shilko	changelog_one.sql	2021-11-17 00:51:04.722600	1
2	insert_users	<null>	d_shilko	changelog_one.sql	2021-11-17 00:51:04.766889	2
3	plus_one_user	<null>	d_shilko	changelog_one.sql	2021-11-17 00:51:04.771482	3

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);
}
```

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)

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

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

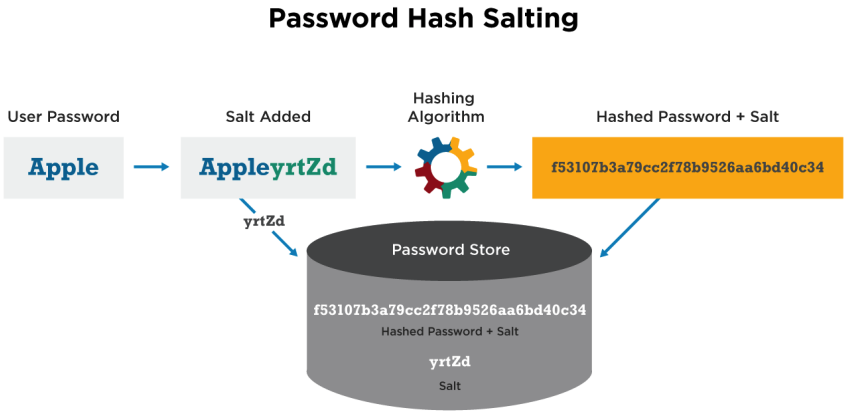
Складиштење лозинки I

- 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 компоромитовати исте лозинке

Складиштење лозинки II

- Најбоље је складиштење *salted hash*-а лозинке $salted\_hash = HashFunc(pass + salt)$
  - $salt$ 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
- Напомена: лозинке не смеју да се шаљу уколико веза није безбедна (HTTPS), јер у супротном могу да буду украдене без обзира на безбедно складиштење!

Складиштење лозинки III



Слика: шематски приказ *salted hash*-а

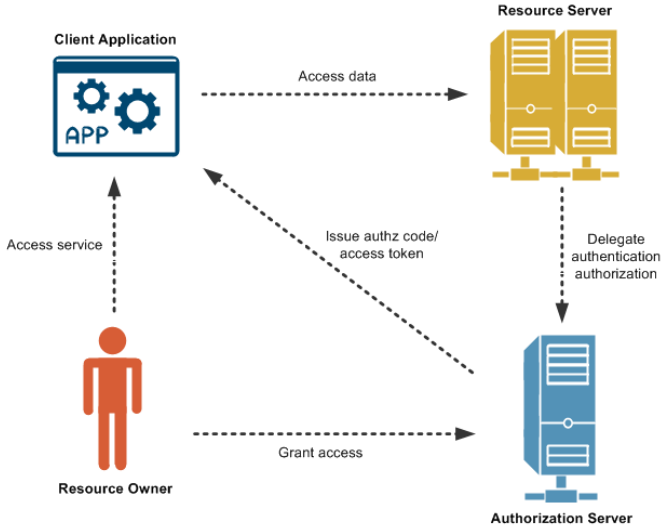
Основни ток

- Креира се кориснички налог
  - у зависности од врсте апликације, корисник се самостално региструје или добија готов налог
- Корисник се пријављује у апликацију својим креденцијалима (корисничко име и лозинка) и добија access token
  - access token садржи ID корисника као и његову улогу
- Уз сваки захтев, корисник шаље свој access token
  - уколико access token истекне, потребно је да се корисник поново пријави

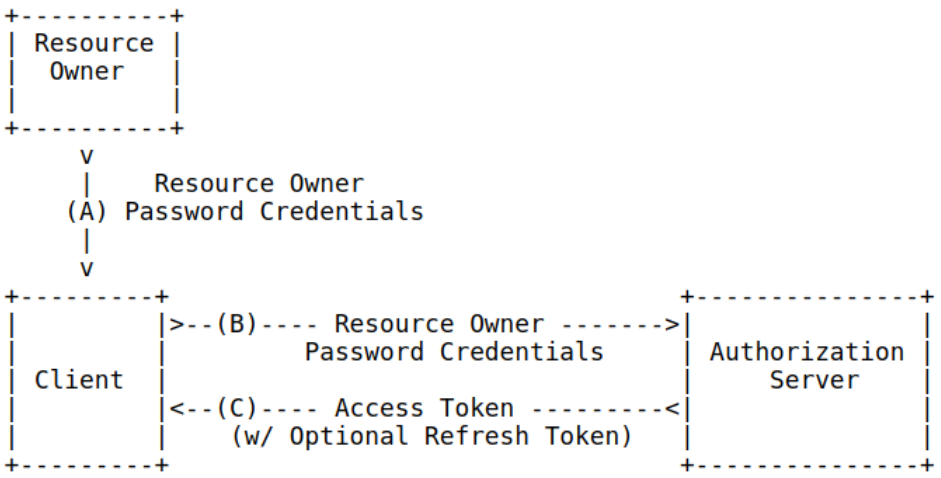
OAuth 2.0 I

- Проблем: како да омогућимо да друга апликација буде клијент који извршава акције у име корисника?
- Једноставно решење: апликацији дајемо креденцијале
  - дељење креденцијала никада није добра идеја
  - апликација би имала сва корисничка права
- Боље решење: апликацији дајемо *access token*
  - нема дељења креденцијала
  - токен има ограничена права приступа на неопходан подскуп $token\_rights \subseteq user\_rights$
- Ми ћемо да имплементирамо упрошћену верзију која не подржава *3rd party* клијенте
  - **Resource Owner Password Credentials Grant** без *refresh token*-а

OAuth 2.0 II


Client Application
APP
Access data
Resource Server
Access service
Issue authz code/
access token
Delegate
authentication
authorization
Resource Owner
Grant access
Authorization Server

OAuth 2.0 III



Слика: Resource Owner Password Credentials Grant

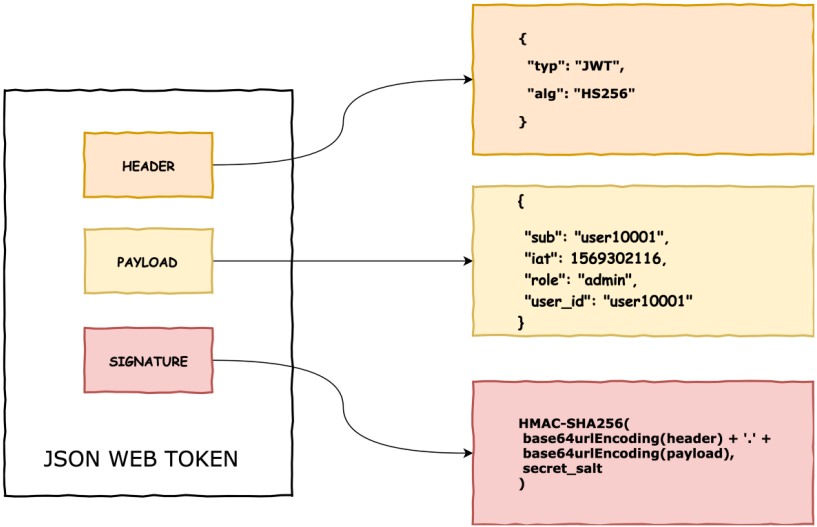
Access token

- Издаје га **Authorization Server**, шаљемо га у сваком захтеву ка **Resource Server**
- Уколико је токен истекао, или је из другог разлога невалидан, **Resource Server** одбија наш захтев
- Уколико је токен валидан, даља права приступа одређује логика апликације (подсетник: RBAC, ABAC)
- Напомена: **Authorization Server** и **Resource Server** не морају да буду одвојене апликације, већ одвојени *endpoint*-и у једној апликацији

JSON Web Token I

- Формат за представљање *access token*-а
- **Header**: Тип токена и алгоритам коришћен за дигитални потпис
- **Payload**: ID корисника, улога, датум док којег важи токен, додатна поља
- **Signature**: Дигитални потпис који апликација проверава како би утврдила да ли је она издала токен
- Напомена: Base64 је алгоритам за кодирање, а не енкрипцију!
  - односно, свако може да прочита наш токен, те он не би требало да садржи тајне информације

JSON Web Token II



JSON Web Token III

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

JSON Web Token IV