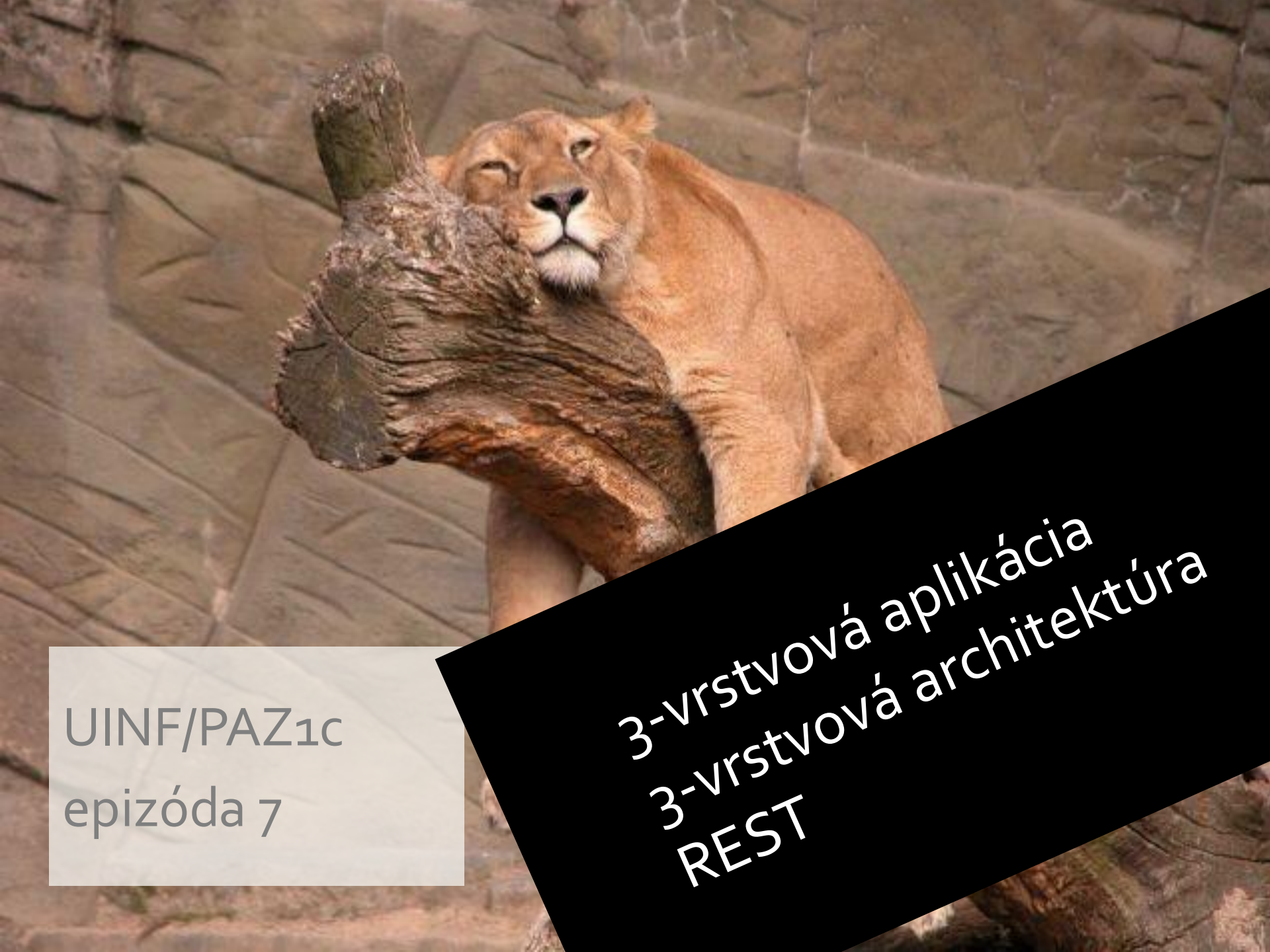




UINF/PAZ_{1c}
epizóda 7

3-vrstvová aplikácia
3-vrstvová architektúra
REST

kde sme?

dvojvrstvová aplikácia

perzistencia

Dvojvrstvová / two-tier aplikácia

Prezentačná vrstva + aplikačná logika

- používateľské rozhranie
- zobrazenie a získavanie informácií od používateľa priamo prepojená s biznis logikou

Databázová vrstva

- úložisko dát
 - typicky relačná databáza/súborový systém
- dáta bývajú často na samostatnom stroji

Ako rozpoznať dvojvrstvovú aplikáciu?

- dáta sú uložené v **vzdialenej databáze**, všetko ostatné je v rámci aplikácie
 - „windowsové účtovníctvo“
- **indikátor**: žonglovanie s dátami v obsluhu GUI
 - v metódach formulára sa zoskupujú/prepočítavajú dáta, ktoré prídu z DAO
- **indikátor**: vývoj alternatívneho UI/UX vyvoláva mdloby

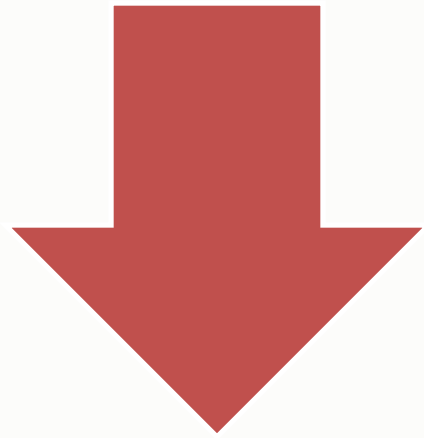
Jednovrstvová aplikácia

- **single-tier** application
- „všetko na hromade“
- klasické skripty / úlohy z PAZ1a,...
- aplikácia načíta zo súboru, použije dáta, vyplúje ich

Užitočné len v prípade jednopoužívateľských
jednouúčelových aplikácií

biznis logika

Na koho zakričať „mám otvoriť
používateľovi pri tejto čítačke“?



DAO?

Kontrolér?



komplexné algoritmy?

- autorizácia na dverách
- nájsť, ktorí ľudia majú najviac práv
-

Kód do obslužnej metódy vo formulári?

- čo ak sa UI zmení / rozšíri / vymení?
- čo ak chceme viacero rozličných UI?

Často používaný prístup, ale vedie k extrémne neprehľadnému kódu.



Kód do metódy DAO?

- čo ak chceme viacero rozličných úložísk?
- čo ak chceme viacero variantov algoritmu?

Vhodný prístup pre jednoduché dvojvrstvové aplikácie, s výhradami.



trieda biznis logiky
- 3. vrstvy

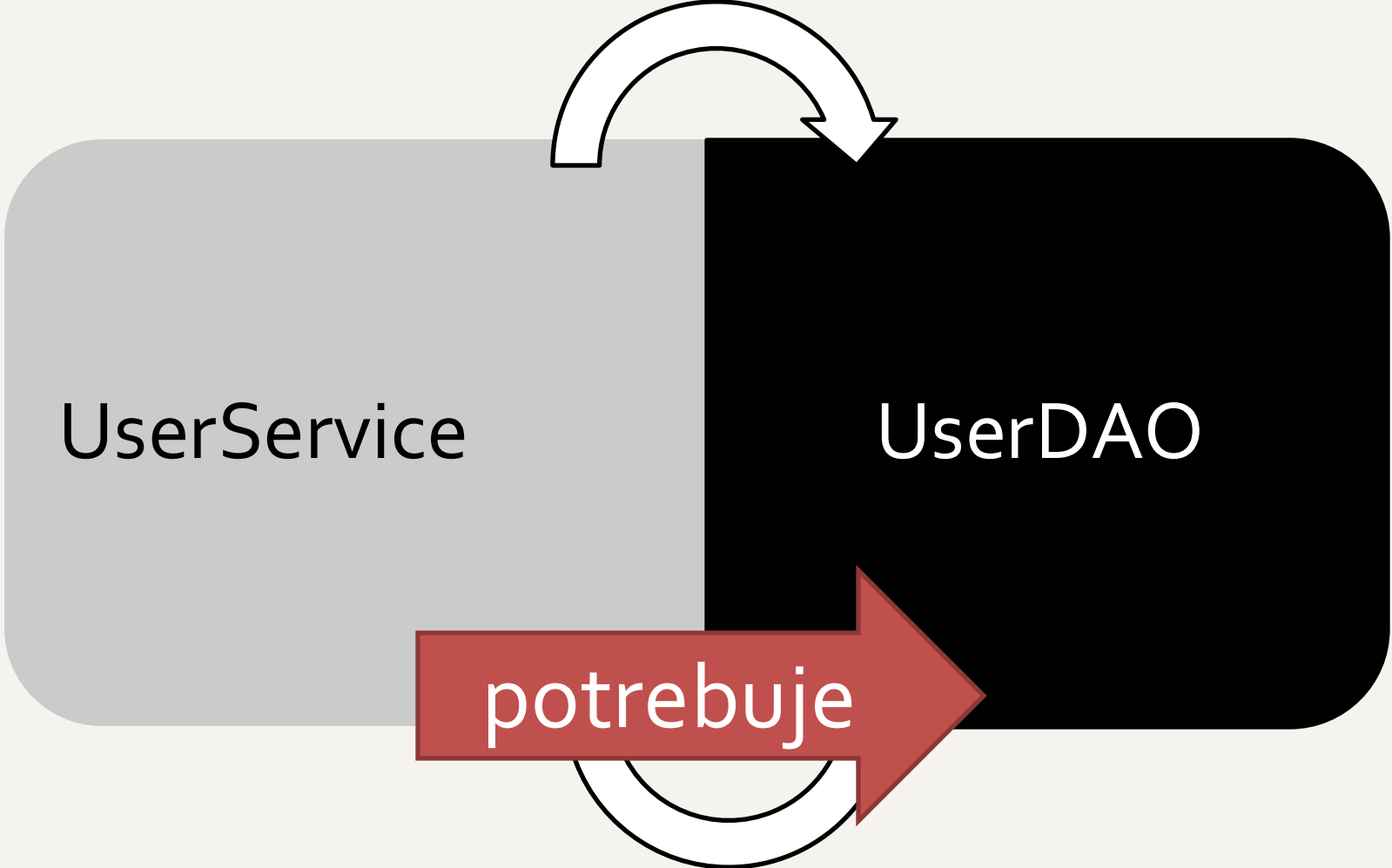
```
public interface UserService {  
    boolean validateAccess(  
        String chipId,  
        Long chipReaderId  
    );  
}
```

```
public class DefaultUserService  
    implements UserService {
```

```
...
```

```
    public boolean validateAccess(String chipId,  
        CardReader cr) {  
        User user =...;  
        if (user.active() &&  
            user.hasAccess(CardReader cr)) {  
            return true;  
        }  
        return false;  
    }  
}
```

dáta pre službu?



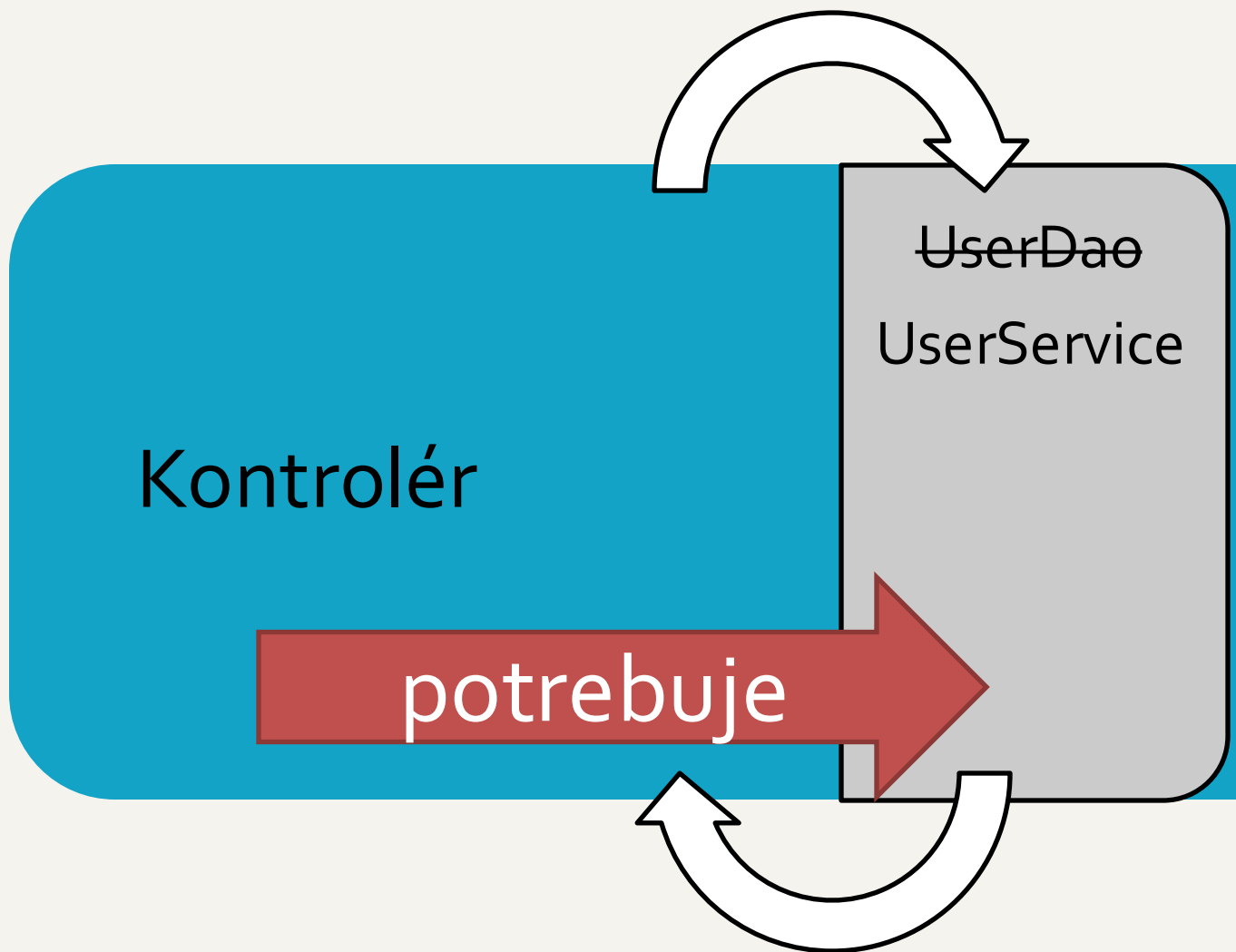
```
public class DefaultUserService  
    implements EntranceService {
```

```
    private UserDao userDao  
        = UserDaoFactory.INSTANCE.getUserDao();
```

```
    public boolean validateAccess(String chipId,  
        CardReader cr) {
```

```
        User user = userDao.getByChipId(chipId);  
        /* overíme práva */
```

```
    }  
}
```



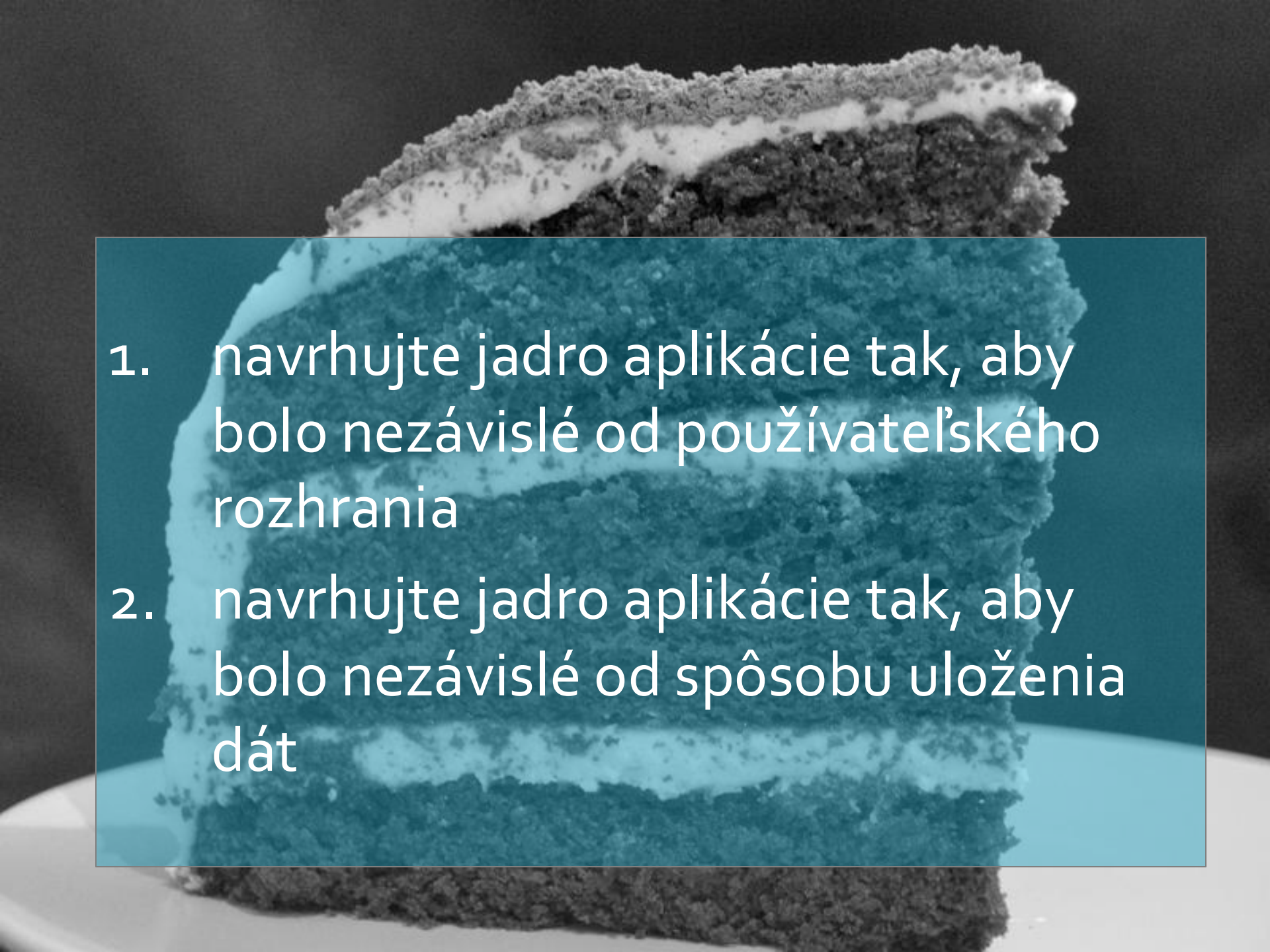
Kontrolér (UI)

potrebuje

UserService (Business)

potrebuje

UserDao (DB)

- 
1. navrhujte jadro aplikácie tak, aby bolo nezávislé od používateľského rozhrania
 2. navrhujte jadro aplikácie tak, aby bolo nezávislé od spôsobu uloženia dát

3tier

prezentácia
UI/UX

biznis

príkazy,
rozhodovanie,
spracovanie

perzistencia
uloženie dát

Kontrolér (UI)

Kurník, ako mám ja, primitívny procesor, technológia z 80's
klikať v GUI?

UserService (Business)

UserDao (Data)



Entrance potrebuje viacero UI

- GUI pre ľudských používateľov systému Entrance
- Nejaké iné "rozhranie" pre fyzické čítačky na dverách

Potrebujeme centrálnu DB
na serveri

2-vrstvová ~~appka~~ architektúra

- Appka sa priamo pripája k DB
 - Úložisko (u nás MySQL)
 - Užívateľské rozhranie (u nás JavaFX) s DAO a Service triedami



GUI (frontend)



2-vrstvová ~~appka~~ architektúra

- Appka sa priamo pripája k DB
- Nevhodný zastaralý prístup (pre mobil/desktop GUI)
 - Použiteľný prístup pre server-side web UI
- Ťažko portovateľné (web UI, mobily, ...)
- Obrovský risk uniknutia prístupu k DB
 - Nikdy neverte kódu (ani vlastnému), ktorý beží na stroji klienta
 - Hackeri vedia ľahko pripojiť debugger k bežiacemu procesu a čítať aj meniť premenné

Znaky web appky

- Centrálna DB spoločná pre vš. užívateľov
- Aktivity užívateľov sa navzájom ovplyvňujú
 - Napr. objednávkový systém, eshop, internet banking, soc. siete, marketplace, **prístupový systém,**
- ...

Moderná 3-vrstvová appka architektúra (2005)

- Moderná appka sa skladá z aspoň 3 programov
 - Úložisko (u nás MySQL)
 - Užívateľské rozhranie (u nás JavaFX)
 - Biznis API – prepojenie UI a DB

Moderná 3-vrstvová appka architektúra (2005)

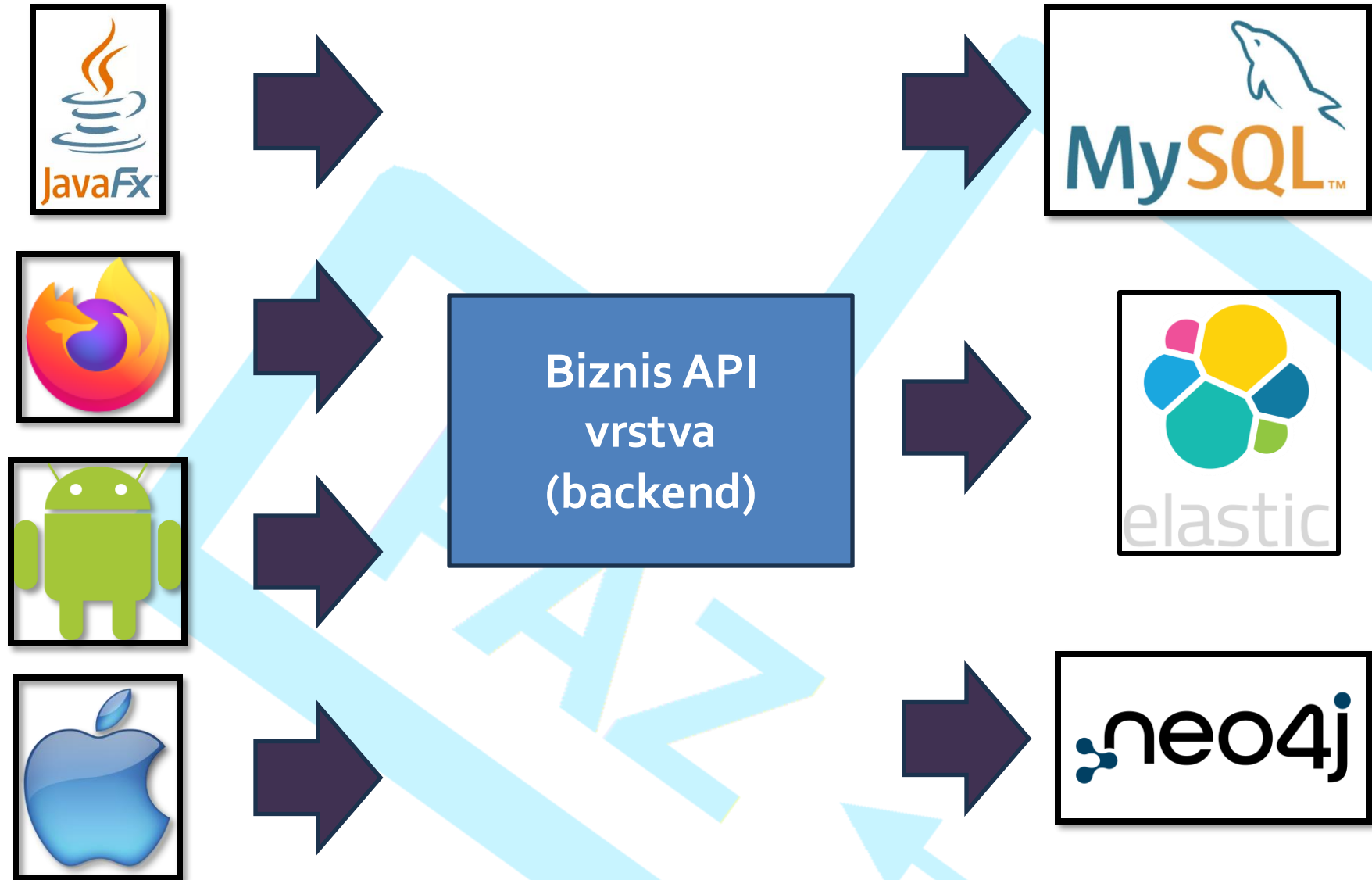
- Moderná appka sa skladá z aspoň 3 programov
 - Úložisko (u nás MySQL)
 - Užívateľské rozhranie (u nás JavaFX)
 - Biznis API – prepojenie UI a DB



GUI (frontend)



Moderná 3-vrstvová appka architektúra (2026)

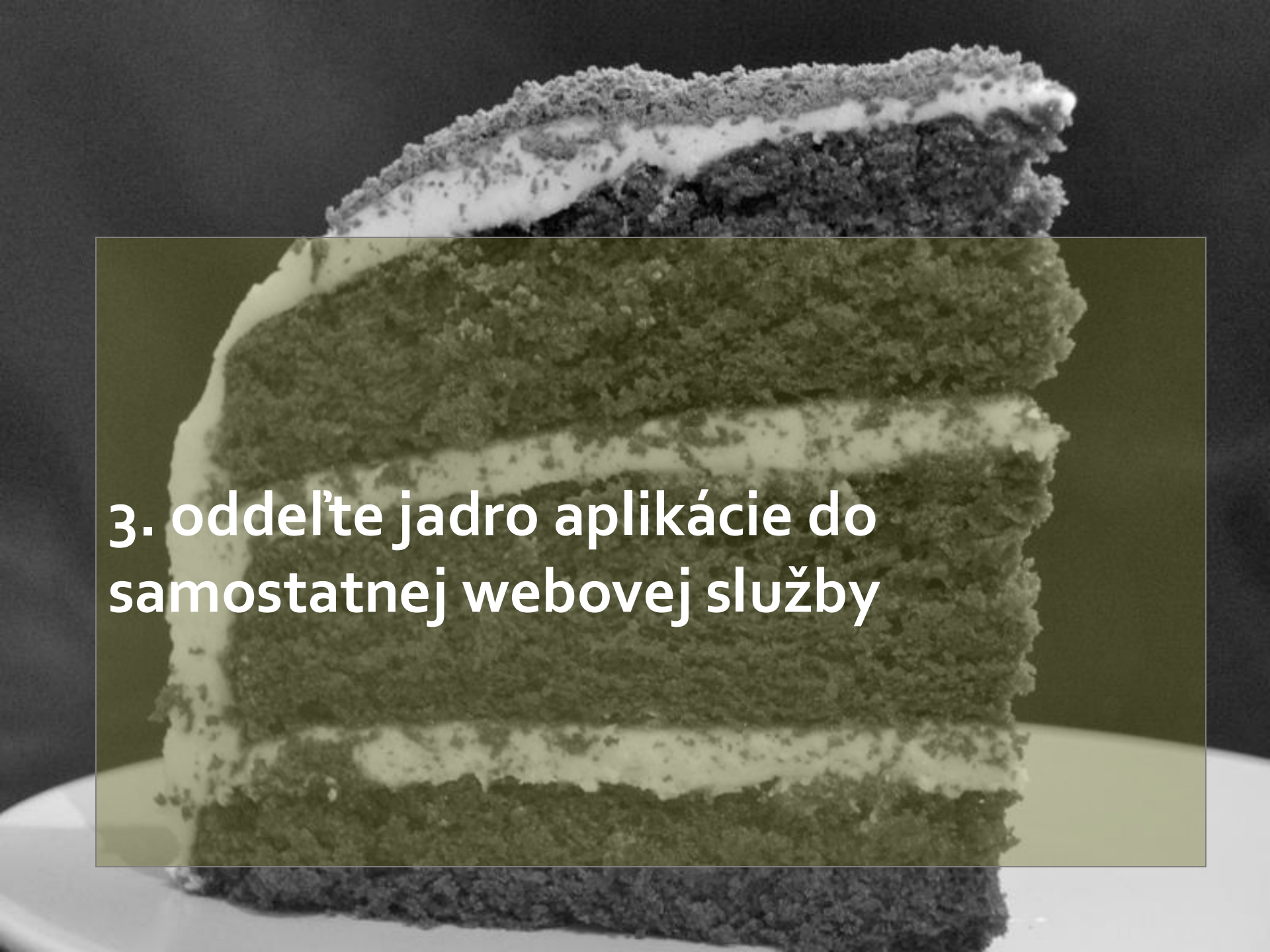


3-vrstvová ~~appka~~ architektúra

- GUI appka komunikuje s vaším business API
 - Business API validuje užívateľa a verifikuje všetky jeho akcie
- Prakticky každá viac-užívateľská appka používa túto architektúru
- Ľahšie portovateľné (web UI, mobily, ...)

3-vrstvová ~~appka~~ architektúra

- Ťažšie hacknuteľná (ak ju robíte správne)
 - Všetky užívateľské akcie verifikuje a reálne vykoná business API na vašom serveri
 - GUI appku robíte "iba aby vyzerala pekne"
 - Všetko vaše know-how a dáta sú bezpečne na vašom serveri



**3. oddel'te jadro aplikácie do
samostatnej webovej služby**

Komunikácia medzi programami po sieti

- GUI appka musí komunikovať s business API
- Potrebujeme nejaké User**NetworkDao** v GUI a nejaký User**NetworkController** pre business API

**Klientsky
počítač**

**Centrálny
server**

*Network*Dao

Network
Controller

potrebuje

```
graph LR; subgraph Client [Klientsky počítač]; ND[NetworkDao]; end; subgraph Server [Centrálny server]; NC[Network Controller]; end; ND -- potrebuje --> NC;
```

The diagram illustrates a dependency between two components. On the left, under the heading 'Klientsky počítač' (Client computer), is a blue box labeled 'NetworkDao'. On the right, under the heading 'Centrálny server' (Central server), is a blue box labeled 'Network Controller'. A red arrow points from the 'NetworkDao' box to the 'Network Controller' box, with the word 'potrebuje' (needs) written inside the arrow, indicating that the client-side component depends on the server-side component.

Komunikácia medzi programami po sieti

- Mnohé prístupy
 - od bezstavového volania vzdialených procedúr
 - cez implementáciu zložitejších aplikačných sieťových protokolov
 - až po výpočtové frameworky manažujúce distribúciu dát a výpočov
- Čím jednoduchšie, tým
 - ľahšie použiteľné
 - ľahšie implementovateľné
 - ľahšia multiplatformnosť

Cesta k jednoduchosti

- služba ~ sieťovo prístupné API
 - z mobilnej aplikácie
 - z webovej aplikácie
 - z aplikácie operačného systému
 - z inej služby
- **REST** ~ webová služba
 - „Representational State Transfer“
 - Roy Fielding, 2000, kapitola v PhD práci
 - využíva existujúci protokol HTTP po svojom
 - jednoduché správy

Filozofia RESTu

- Realizácia CRUD operácií pomocou HTTP príkazov nad kolekciou resource-ov
 - Create ~ POST, PUT
 - Read ~ GET
 - Update ~ PUT, PATCH
 - Delete ~ DELETE
- Resource
 - objekt sveta, najčastejšie entita
 - má identifikátor v tvare URI
 - <http://paz1c.ics.upjs.sk/ucitelia/2>

Filozofia RESTu

- Resource-y po sieti cestujú v tvare JSON

```
{“meno”: “Jano Pokojný”, “vek”: 32, “deti”: [“Janko”, “Marienka”]}
```

- legálny JavaScriptový kód
 - parsery/writery v každom jazyku
- stavy ok/chyby/výnimky prichádzajú zo servera cez stavové kódy

200= OK, 404 = Not Found, 201 = Created,...

- filtrovania/triedenia cez request parametre

<http://paz1c.ics.upjs.sk/ucitelia?minimalnyVek=35>

REST v praxi

- REST API možno jednoducho zverejniť
- nejedna služba má REST API
 - Facebook, Twitter, Instagram, Claude, ChatGPT,
- štandardný spôsob

Podme REST

- Doplňme si do pom.xml

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>VERZIA</version>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

main metóda pre Spring Boot

- Pohľadá v triedach, čo má zverejniť
- Spustí webový Servlet kontajner Tomcat

```
@SpringBootApplication
```

```
public class RestApplication {
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(RestApplication.class, args);
```

```
    }
```

```
}
```

@RestController

- Controller je trieda, ktorá obsahuje zverejnené metódy
- Je oannotovaná s **@RestController**
- Zverejnené metódy sú oannotované s **@GetMapping**(časť_url_za_doménou)

```
@GetMapping("/users")  
public List<User> getAllUsers() {  
    ...  
}
```

Získanie hodnoty z URL

http://localhost:8080/users/id/1

- Všetko medzi dvoma lomkami, alebo od poslednej lomky do konca môžem vytiahnuť do premennej

```
@GetMapping("/users/id/{id}")  
public User getUserById(@PathVariable long id) {  
    ...  
}
```

Vytvorenie usera

Dobré REST API pre
POST vracia status
201 namiesto
štandardného 200

```
@PostMapping(value = "/users")  
@ResponseStatus(HttpStatus.CREATED)  
public void create(@RequestBody User user) {  
    userDao.create(user);  
}
```

Posielame dáta do Restu

- HTTP príkaz sa zmení z GET na POST
- V tele požiadavky posielame dáta vo formáte Json, ktoré chceme spracovať
- Použijeme @PostMapping

```
@PostMapping(value = "/users",  
              method = RequestMethod.POST)  
public void add(@RequestBody User user) {  
    /*spracujeme objekt user*/  
}
```

O preklad z JSONu sa
postará knižnica Jackson

Analogicky spravíme update a delete

```
@PutMapping("/users")  
public void update(@RequestBody User user) { .. }
```

```
@DeleteMapping("/users/id/{id}")  
public void delete(@PathVariable Long id) { .. }
```

Nepovinné parametre

- v URL za ? napr.
 - <http://localhost:8080/users?count=10&start=5>

```
@GetMapping("/users")
public ResponseEntity<List<User>> getUsers(
    @RequestParam(value = "count", required = false) Optional<Long> count,
    @RequestParam(value = "start", required = false) Optional<Long> start) {
    ...
    if (count.isPresent()) {
        resultCount = count.get();
    }
    ...
}
```

Vyhadzujeme vlastné výnimky

```
@GetMapping("/users/{id}")
public User getUserById(@PathVariable int id) {
    User user = userDao.findById(id);
    if (user == null) {
        throw new UserNotFoundException();
    }
    return user;
}
```

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class NotFoundException extends Exception {}
```

ResponseEntity = dáta + HTTP status

- Alternatíva k vyhadzovaniu výnimiek v controlleri
- Vieme takto riešiť ľubovoľnú výnimku
- Dobré REST API správne používa HTTP statusy
 - 200 – OK
 - 400 – BAD REQUEST
 - 404 – NOT FOUND
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

```
@GetMapping("/users/{id}")
public ResponseEntity<User> getUserById(
    @PathVariable int id) {

    User user = userDao.findById(id);
    if (user == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    return ResponseEntity<>(user, HttpStatus.OK);
}
```

Mapovač výnimiek na zmysluplné odpovede

- Anotácia `@ControllerAdvice` anotuje triedu, v ktorej budú spoločné odchytačače výnimiek (+iné mapovačky) pre všetky kontroléry
- Vieme jednotným spôsobom riešiť naše výnimky (`NotFoundException`) aj existujúce výnimky (`IllegalArgumentException`)
- Metóda, ktorá mapuje výnimku na HTTP odpoveď je anotovaná cez `@ExceptionHandler(TriedaVýnimky.class)`
- Exception handler okrem čísla HTTP stavu môže posilať aj telo odpovede

Trieda pre telo chybovej odpovede

- Môžeme si sami štandardizovať všetky chybové hlášky
- Klienti dostanú JSON z ktorého ľahko (vždy rovnako) extrahujú čo je problém

```
public record ApiError(int status, String errorMessage) {  
}
```

Príklad mapovača výnimiek

```
@ControllerAdvice
public class RestControllerAdvice {

    @ExceptionHandler(BadRequestException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ResponseBody
    public ApiError
        handleBadRequest(BadRequestException e) {
        return new
            ApiError(HttpStatus.BAD_REQUEST.value(),
                e.getMessage());
    }
    ...
}
```

Ako teda mapovať výnimky na HTTP status?

- Ukázali sme si 3 spôsoby
 - `@ResponseStatus`
 - najjednoduchšie
 - nad vlastnou výnimkou
 - nevieme takto riešiť cudzie výnimky
 - Nič nehovoriaca chybová hláška
 - `ResponseEntity<T>`
 - priamo v `RestController-i`
 - najlepšia kontrola
 - vieme riešiť naše aj cudzie výnimky
 - `@ControllerAdvice`
 - odpraceme mapovanie bokom
 - vieme riešiť naše aj cudzie výnimky
- V reálnych projektoch uvidíte všetky 3 spôsoby naraz a ešte viac

Otázky?

