**REST webszolgáltatások - GET művelet**

**Restful webszolgáltatások tulajdonságai**

modern webes alkalmazásoknál backend és frontend különböző technológiákkal van implementálva; klasszikus: backend(üzleti logika és perzisztencia=adatbáziskezelés) Javaval van megírva, a frontend, a böngészőbe futó része az alkalmazásnak, html, css és javaScript technológiákat használja valamilyen keretrendszerrel, pl. Angular vagy React.

Ahhoz, hogy ez a két **stack** kommunikálni tudjon egymással, tipikusan **valamilyen integrációs technológia szükséges**; ezek közül a legelterjettebbek a RESTFUL webszolgáltatások.

Roy Fielding nevéhez fűződik 2000ből

RESTFUL = Representational state transfer

elv: az **alkalmazásokra tekintsünk úgy, mint erőforrásoknak a gyűjteménye, és minden egyes erőforrást valamiféle egyedi azonosítóval meg lehessen címezni**; ezeken az erőforrásokon csak 4 alapvető műveletet lehessen elvégezni -> **CRUD** (create, read, update, delete)

**erősen épít a http protokollra -> erőforrások címzésére az URIt, a különböző műveleteket a http metódusokkal lehet elvégezni ->** PUT, POST, GET, DELETE; státuszkódjait használja

a restful bár nem írja elő, hogy **milyen formátumban közlekedjen az adat** a kommunikáció során, de **leggyakoribb esetben jSON a formátum**, amit könnyű böngészővel feldolgozni és emberi általi olvasásra alkalmas(string)

a restful tervezésénél figyelembe vették az egyszerűséget, skálázhatóságot és platformfüggetlenséget

Annotációk

**Springbe hogyan kell rest webszogáltatásokat implementálni?**

a **Spring MVC-t kell** használni, **kontrollereket** kell létrehozni; a különbség csak annyi, hogy amit visszaadnak a metódusok, már nem template nevek lesznek, hanem **a visszatérési értéket szerializálni kell** (= ellátás egyedi azonosítóval) és azt kell visszaküldeni az ügyfélnek; **a szerializáció tipikusan jSON formátumban** történik

Ennek eléréséhez Springben rá kell tenni a @ResponseBody-t minden metódusra; de hogy ne kelljen ezt minden metódussal eljátszani, létrehozták a **@RestControllert**, ami azzal egyenértékű, minden metódusra rá lenne téve; a @RequestMapping is továbbfejlődött, már nem paraméterként kell megadni, hogy GET vagy POST metódust szolgál-e ki az adott java-s metódus, hanem ezekre külön annotációk vannak már: **@GetMapping, @PostMapping stb.**

**Egy ilyen Controller, ami visszaadja az alkalmazottak listáját jSONben**



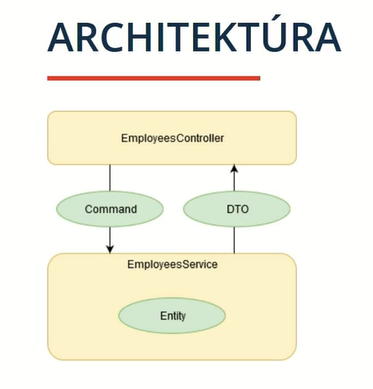
**A GET műveletek az erőforrások lekérésére szolgál.**

@RequestMappinget használunk osztályszinten, melynek megmondjuk, hogy milyen url-en lehet elérni az adott metódusokat

**Konstruktor injectionnel beállításra kerül a Service-**re való hivatkozás

**@GetMapping a metóduson;** jelentése: ez a java-s metódus a megadott urlen lesz elérhető, http GET metódussal.

A metódus amit visszaad, az továbbadja a http responseba.

****

**DTO** = Data Transfer Object(=adatátviteli objektum). A DTO objektum összegyűjti több hívás adatait és egyben küldi el a szervernek, ezáltal csökken a hívások száma és a költség a szerver-kliens kapcsolatokban.

Abban különbözik a DAO(=adathozzáférési objektum)-tól, hogy ez **nem tartalmaznak tesztelendő üzleti logikát; adatátvitelen kívül mást nem csinálnak.**

Az alkalmazásnak az **üzleti entitásai,** amiket adatbázisban is lementünk **csak serviceben elérhetőek és nem közvetítődnek ki a controller rétegbe** (olvashatóság és biztonsági okok miatt) -> Jelen esetben nem Employee osztályt, hanem DTO osztályt, ami nem tartalmazza az Employee metódusait stb.

építhetünk olyan architektúrát is, ahol a DTOkat megkülönböztetjük egymástól a következőképpen:

**befelé jövő DTO-k** – amelyek során mindenképpen történik valami **állapotváltozás**; ez történik klasszikusan, amikor **INSERT** vagy **UPDATE** művelet végződik a adatbázisban; ilyenkor a **DTO neve Command**

Az **entitások** (objektumok, amiket közvetlen adatbázisba lementünk) és a **DTO-k között** kelleni fog valamilyen **konvertálás**

**Lombok**

3rd party library, ami képes kódot generálni; arra jó, hogy a **felesleges gépeléstől megóvjuk** magunkat

a Lombokkal elég csak a megfelelő annotációt rátennünk az osztályra, és a ko**getter/setter** metódusok elérhetővé válnak

Miket képes még generálni? konstruktor, **toString(), equals/hashcode; logger**

Mi kell hozzá? Be kell kapcsolni az annotaiton processzort; a kódírás fordítási időben fog annotáció történni

IDEA támogatja; IDEA-s integrálásához ’Enable annotation processing’ és lombok plugin feltelepítése

a **@Data** annotáció egy metaannotáció; alkalmazásával megírásra kerül az összes getter, @setter, @tostring, @equalshashcode és @RequiredArgsConstructor (az ezekre vonatkozó sima annotációkat tartalmazza)

**@NoArgsConstructor** - legenerálja a paraméter nélküli konstruktor

**@AllArgsConstructor** - legenerálja a konstruktort amiben szerepel az összes paraméter

**Példa**

****

**Modelmapper - a DTO-k és entitások közötti konvertálás**

sok eszköz létezik rá; egyik ezek közül

eszközök két csoportja: **reflection**- és kód generálás alapú; a modelmapper reflectionös

végigmegy reflection-el a java entitáson**, kiolvassa az attribútum értékeit és behív a DTO osztályba és meghívja a settereit ezekkel az értékekkel**

a modelmapper, ha az atrik megegyeznek, minden konfiguráció nélkül átmásolja; ha nem, akkor valamilyen konvertálás kell -> az ilyen speciális esetek kezelésére fluent API-t alkalmaz

**Modelmapper példa**



a // load helyén valami példányosítás áll

a modelMapper.map() metódusával az employeet egy EmployeeDto-vá konvertálja; majd ugyanez streamesen listával

**REST webszolgáltatások - GET művelet – gyakorlat**

Egy olyan REST végpont létrehozása, amin keresztül le lehet kérdezni az alkalmazottak listáját

*employeees apllication továbbfejlesztése:*

1. Lombok függőség felvétele pom.ba; (új dependency és artifact lombok) mjd frissítés
2. új java osztály -> Employee; long id és String name + @Data, @NoargsConstructor és neves konstruktor (explicit), és @AllArgsconstructor
3. EmployeesService, @Service, List<Employees>Collections.synchronizedList! benne new Arrayslist, két alkalmazottal
4. listEmpployees() metódus;
5. Controller osztály, @RestController (azonnal írja be a metódusok visszatérési értékét a http response bodyba; a java objektumainkat először jsonbe fogja konvertálni
6. EmployeesService atri és Controllerre konstruktor
7. listEmployees() a kérés delegálásal a service felé
8. @RequestMapping(”/api/employees”) – milyen címen hallgatnak a metódusok?
9. @GetMapping – ez egy olyan metódus, ami http requesteket szolgál ki GET http metódussal
10. indítás; elindítja a Tomcatet, telepíti bele az alkalmazást; az alkalmazás figyel a gépünknek megfelelő ipn/urlen (amire az olvasható megfelelő a localhost), a megadott végponton(/api/employees) a 8080as porton(webböngészővel való kommunikáció) és várja a kéréseket
11. az alkalmazottka listája jSonként jelenik meg;

**Architektúrális szempontból nem szerencsés az entitásokat visszaadni a service rétegből, ezért létrehozunk egy DTO-t! ->**

1. EmployeeDto osztály ugyanazokkal az atrikkal; konvertáláshoz a modelMapper keretredszert használjuk ->
2. felvesszük pomba dependencyként; artifact: modelmapper
3. kell egy modelmapper példány az AC-be, amit majd felhasználunk -> Beanes példányosító metódus a appplications osztályba
4. **listemployees() metódus módosítása: adjon vissza List<EmployeeDto-t>; megadunk egy új TypeTokent: Type targetlistType = new TypeToken<List><EmployeeDto>{}{}.getType(); -> ezzel tudjuk megmondani, hogy milyen típusú listává konveráljon majd a modelmapper; végére return modelmapper.map(employees, targetlistype)**
5. Controllerbe is átírjuk a metódus visszatérési értékét(= return value)
6. 16. újraindítás, csekkolás

**Kérdések**

*Milyen klasszikus technológiákkal vannak általában implmenetálva a mai alkalmazások?*

*Mi kell a két stack kommunikációjához?*

*Mi a RESTFUL és mit jelent?*

*Alapelve szerint hogy tekint az alkalmazásokra?*

*Az erőforrásokat hogy éri el és milyen műveletke elvégzését engedi rajtuk?*

*Ezek alapján hogy épít a http protokollra?*

*Mit ír elő arra vonatkozóan, hogy milyen formátumban közlekedjen az adat a kommunikáció során?*

*Springbe hogyan kell rest webszogáltatásokat implementálni?*

*A metódusoknak mit kell tennie a visszatérési értékkel és tipikusan milyen formátumban történik ez?*

*Az előbbiek impkementációjához milyen annotácikat kell használni?*

*A @RequestMapping továbbfejlődésével milyen annotációkat teszünk a controlleres metódusokra? Mi a jelentse ezeknek az annoátcióknak?*

*Hogy néz ki egy Controller?*

*Mire jó a GET művelet?*

*Mi az a DTO és mi benne a különbség a DAO-val szemben?*

*Miért használunk DTO-t?*

*Mik azok a bejövő DTOk? Mi történik a használatukkal?*

*Mire lesz szükség az entitások és a DTOk között? Milyen third party libraryt használunk ehhez?*

*Mi az a reflection és hogyan hashnálja a Modelmapper?*

*Mi az a Lombok és milyen annotációit használjunk?*