

# Sovelto

Spring

---

## Springin peruspiirteet

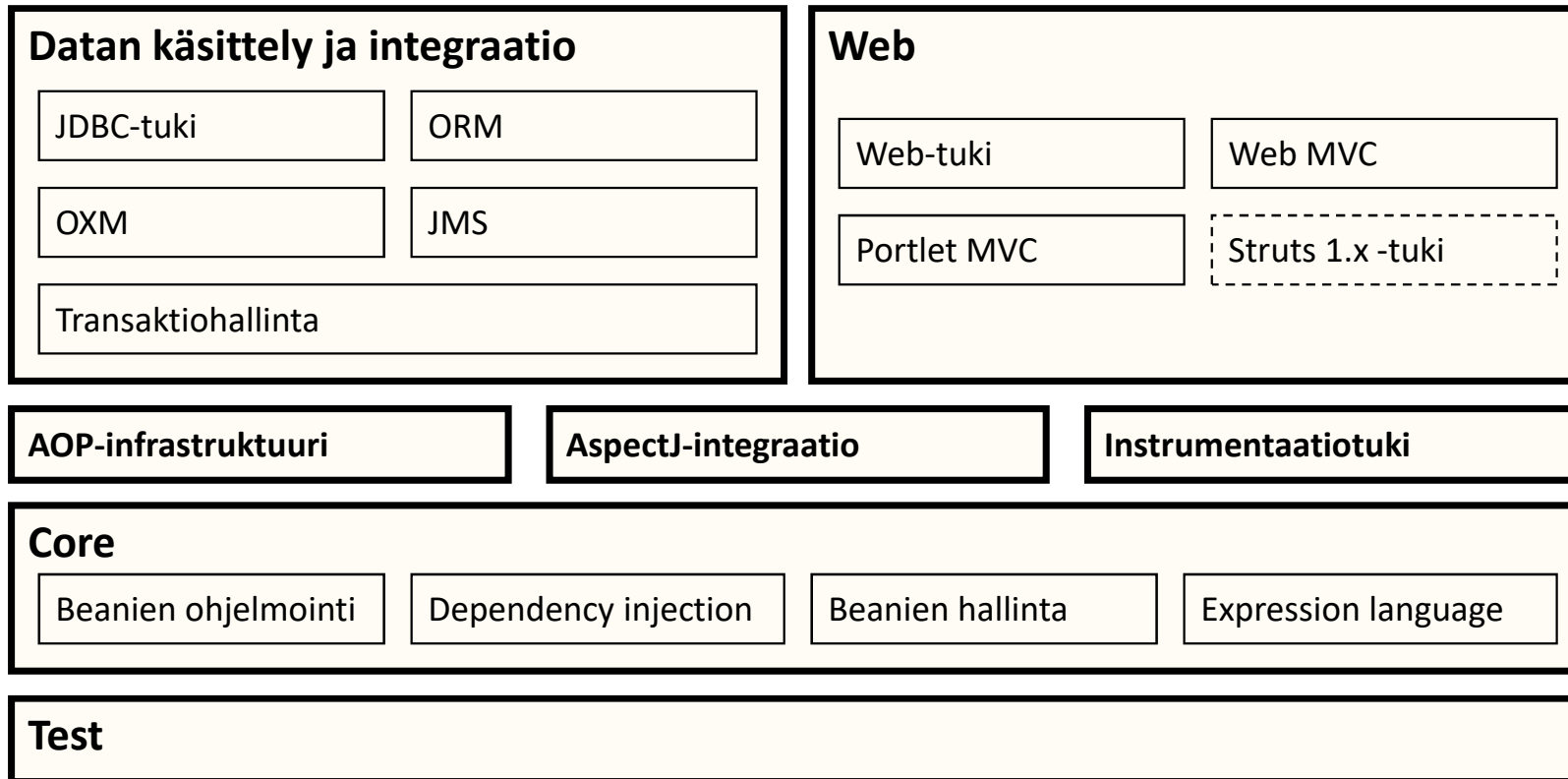
---

**Sovelto**

# Yleistä

---

- Spring on open source -periaatteella luotu modulaarinen sovelluskehys enterprise-sovellusten luomiseksi
- Osat (Spring 4.x):



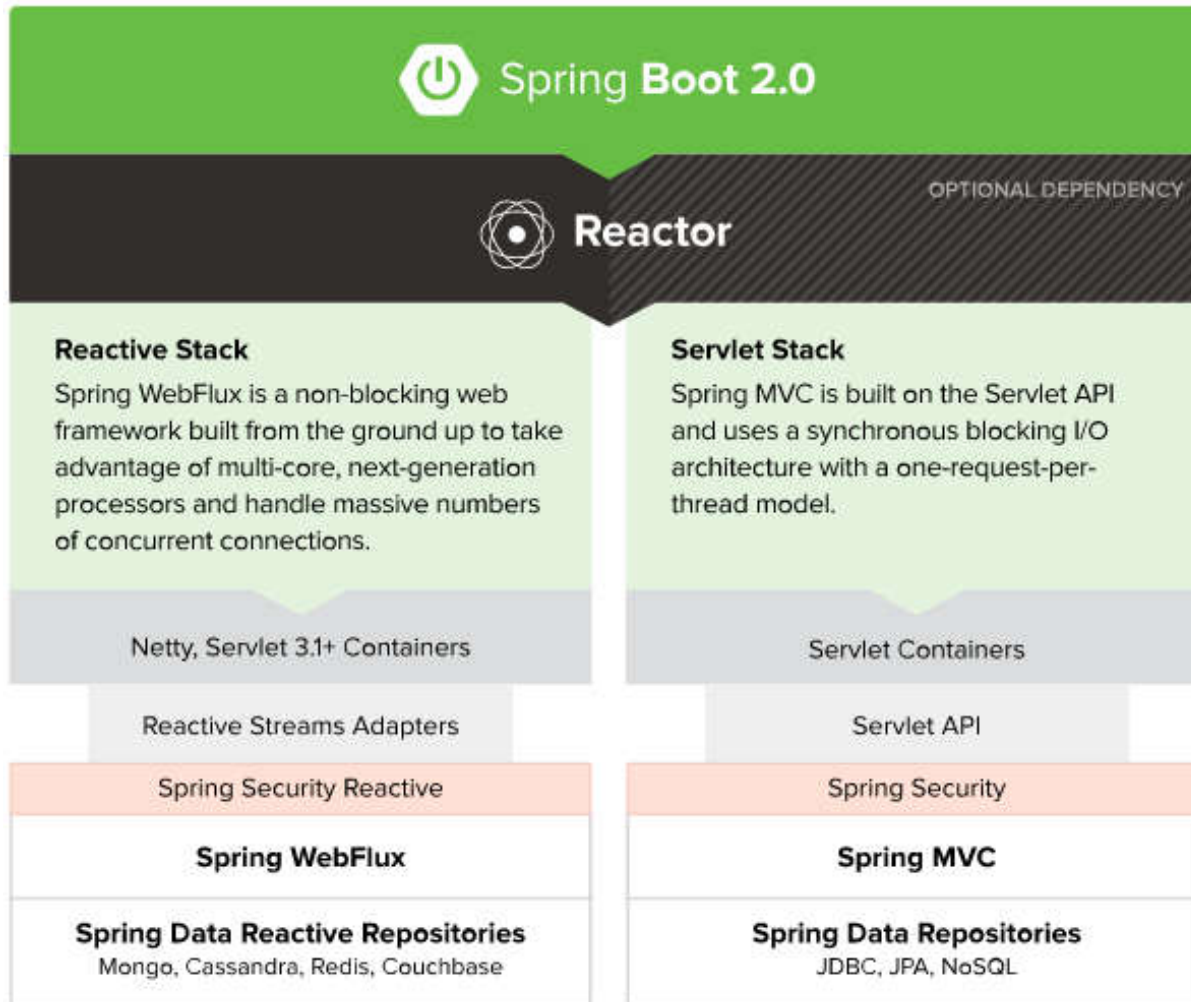
# Spring Boot

---

- Spring Boot on "kevytversio" Spring viitekehyksestä, jolla voidaan toteuttaa Spring sovellus nimenomaan Springin keskeisiä periaatteita käyttäen
  - Spring Boot is Spring's convention-over-configuration solution for creating stand-alone, production-grade Spring-based Applications that you can "just run" - Wikipedia
  - Applikaatioon kuuluu myös Java sovelluspalvelin, tyypillisesti embedded Tomcat
  - <https://projects.spring.io/spring-boot/>
- Yksinkertainen palvelu voisi koostua kolmesta osasta:
  1. Applikaatio-luokka, joka käynnistää Spring ympäristön
  2. Kontrolleri, joka käsittelee palvelupyynnöt
  3. Dataluokka jonka avulla asiakkaan ja palvelun välistä tiedonsiirtoa hoidetaan
- Kaikki luokat voivat olla POJOja, toiminnallisuuden saa käyttämällä Springin annotaatioita ja autokonfiguraatiota
- Palvelu käynnistetään ajamalla Applikaatio-luokan main metodia, se alustaa Spring ympäristön ja tekee konfiguraatiot
- Kontrolleri mappaa osoitteet luokan metodeille, ja käyttää Dataluokkaa / luokkia tiedon palauttamiseen tai käsittelyyn

POJO = Plain Old Java Object

# Spring Boot - nykyarkkitehtuuri



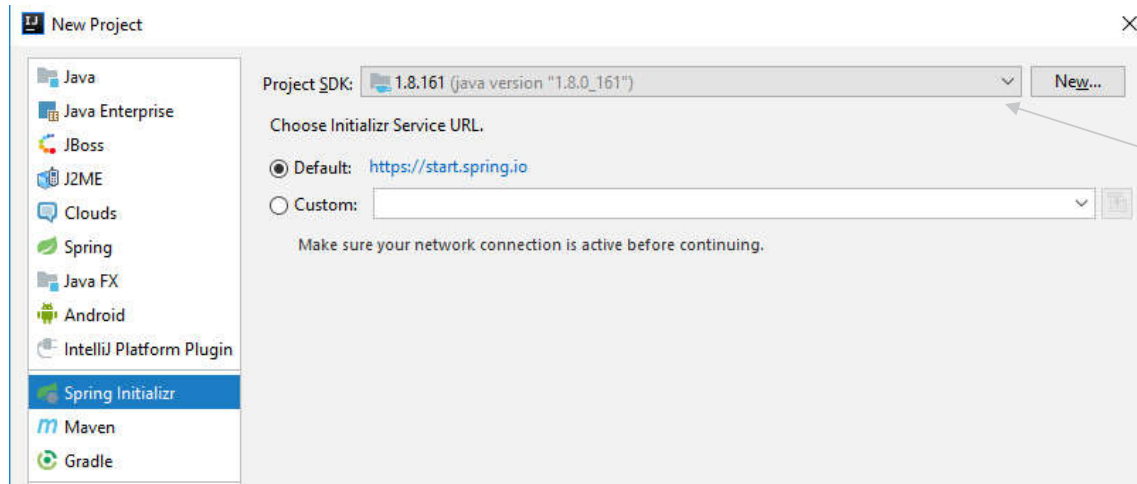
← Kuten JAX-RS

Lähde: <https://spring.io/>

# IntelliJ Spring Boot projektin luonti

1/3

- Spring Boot projektin luonnissa kannattaa käyttää Spring Initializr projektimallia



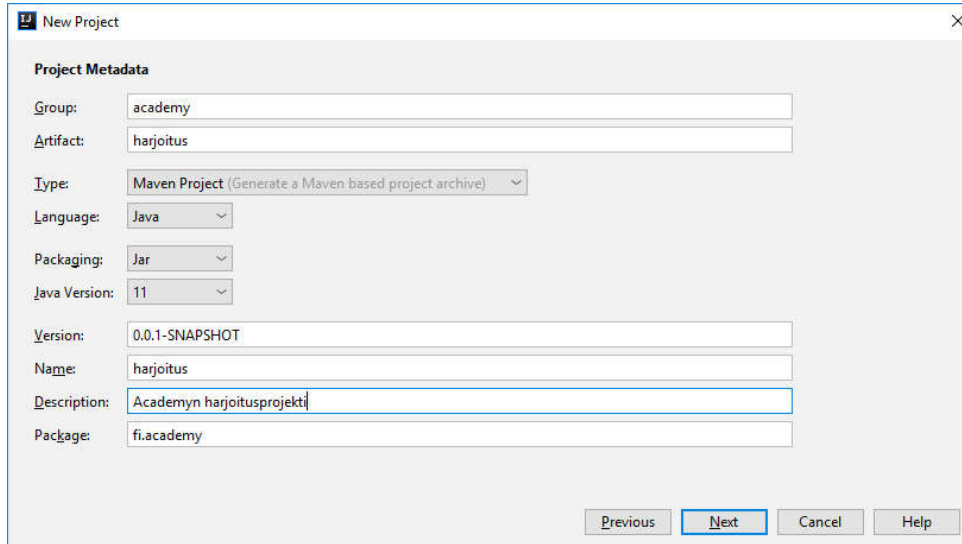
Mieluummin ehkä versio 11

- Tämä generoi spring.io sivustolla Maven-pohjaisen Spring Boot projektin, jonka IntelliJ käy automaattisesti lataamassa paikalliselle levylle

# IntelliJ Spring Boot projektin luonti

2/3

- 



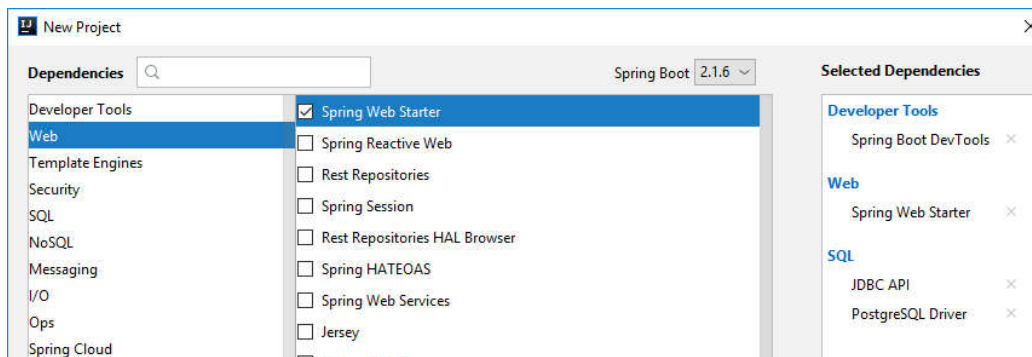
The 'New Project' dialog in IntelliJ IDEA, 'Project Metadata' tab. The fields are filled with the following values:

- Group: academy
- Artifact: harjoitus
- Type: Maven Project (Generate a Maven based project archive)
- Language: Java
- Packaging: Jar
- Java Version: 11
- Version: 0.0.1-SNAPSHOT
- Name: harjoitus
- Description: Academyn harjoitusprojekti
- Package: fi.academy

Buttons at the bottom: Previous, Next, Cancel, Help.

Maven asetukset

- Projektin riippuvuudetkin (dependencies) alustetaan Wizardin avulla, niitä voi luonnollisesti muuttaa tarvittaessa myöhemmin



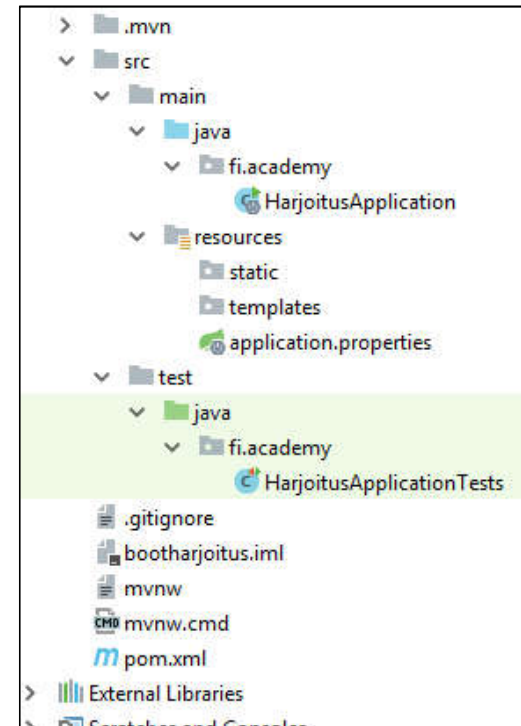
The 'New Project' dialog in IntelliJ IDEA, 'Dependencies' tab. The 'Spring Boot' version is set to 2.1.6. The 'Web' category is selected in the left sidebar. The 'Selected Dependencies' list on the right includes:

- Developer Tools
  - Spring Boot DevTools
- Web
  - Spring Web Starter
- SQL
  - JDBC API
  - PostgreSQL Driver

# IntelliJ Spring Boot projektin luonti

3/3

- Projektin rakenne: tuttu Maven + Spring Boot hakemistot
- Koodit kirjoitetaan tyypillisesti samalle tasolle kuin projektin Application-luokka
- Applikaatioluokka on siis generoitu, oman koodin kirjoittaminen aloitetaan tyypillisesti kontrollerilla



Myös muille työkaluille on vastaava (esim. Eclipse + STS) projektin luonti ja aina voi käyttää [start.spring.io](https://start.spring.io) sivua



# SpringBootApplication

---

```
package kurssi;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

- @SpringBootApplication koostuu seuraavista annotaatioista:
  - @Configuration - luokassa voidaan määritellä @Bean'eja
  - @EnableAutoConfiguration - Spring Boot etsii muut beanit
  - @EnableWebMvc jos spring-webmvc on polussa
  - @ComponentScan etsii beanit paketista kurssi, muut paketit konfiguroitava

```
@SpringBootApplication(scanBasePackages = {"kurssi", "fi.academy.harjoitukset"})
```

# Projektin määrittäminen: Maven (esimerkki)

---

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fi.academy</groupId> <artifactId>demoprojekti</artifactId>
  <version>0.1.0-SNAPSHOT</version>
  <parent><groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.5.RELEASE</version>
  </parent>
  <dependencies><dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency><groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional></dependency>
</dependencies>
<properties><java.version>1.8</java.version></properties>
<build><plugins><plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
</plugin></plugins></build>
</project>
```

# Web MVC

---

- Yksi osa Spring arkkitehtuuria
- Spring Web MVC moduuli tuo mukanaan tuen HTTP palveluiden toteuttamiseen
  - dynaamisten HTML-sivujen näyttämiseen esimerkiksi JSP- tai Thymeleaf template engineilla
  - REST palveluiden toteutukseen
- Useimmiten Spring Boot projektit ottavat MVC-moduulin mukaan
- Toteutuksena @Controller annotaatiolla varustettu luokka, joka palvelee kutsujaa (selain/REST palvelua kutsuva koodi)
  - @RestController mikäli toteutetaan nimenomaan REST palvelu

# REST-kontrollerin pohja

---

- Esimerkki:

```
package kurssi;

@RestController
public class RestKontrolleri {

    @GetMapping(value = "/restkutsu")
    public Henkilo restMetodi() {
        Henkilo henkilo = new Henkilo();
        henkilo.setEtunimi("Mikki");
        henkilo.setSukunimi("Hiiri");
        return henkilo;
    }
}
```

- @RestController yhdistää Controller ja.ResponseBody annotaatiot
  - .ResponseBody ilmoittaa, että metodin paluuarvo palautetaan sellaisenaan kutsujalle, eikä siirrytä paluuarvon osoittamaan näkymään
- @GetMapping määrittelee, että metodia voidaan kutsua vain Get-HTTP metodilla
  - Vastaavasti löytyy @PostMapping, @PutMapping yms.
  - @RequestMapping yhdistää yleisimmät

# Harjoitus: Hello Spring Boot

---

- Luo uusi Spring Boot projekti IntelliJssä
  - Riippuvuuksina Core-kategorian DevTools, sekä Web-kategorian Web
  - **Älä** ota tietokanta-kategorian riippuvuuksia mukaan
- Toteuta projektiin uusi REST controller. Sille metodi, joka palauttaa vakiotekstin (String), esimerkiksi "Terve Spring Bootista"
- Muista toteutuksessa:
  - @RestController luokalle, ja
  - @GetMapping julkiselle metodille, joka palauttaa Stringin - tee mäppäys osoitteeseen "/terve"
- Käännä projekti ja aja (Java applikaationa)
- Avaa selain (tai mieluummin Postman/curl/IntelliJ:n rest-api työkalu) osoitteeseen `http://localhost:8080/terve`

# Spring boot - ajaminen, lisätietoa

---

- Spring boot applikaatio pohjautuu public static void main(String[]) metodiin, joten sen ajaminen on samanlaista kuin ajaisi normaalia desktop sovellusta
  - Mavenilla: mvn -q spring-boot:run
- Käännös tehdään itse ajettavaksi (self extracting) jar-tiedostoksi. Tämä toimii mainiosti mikropalvelin-arkkitehtuurissa
- Mavenin spring-boot-starter-parent tekee kaikki asetukset oletuksena itse ajettavaksi Jar-tiedostoksi

<parent>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-parent</artifactId>

    <version>2.1.6.RELEASE</version>

</parent>

# Levittäminen ulkoiselle palvelimelle, extra

---

- Jos Spring Boot applikaation haluaa ajaa jo olemassa olevalla palvelimella, täytyy sen build tehdä War-paketiksi jarin sijaan - samoin Application luokkaa täytyy muuttaa palvelimen käynnistettäväksi

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
    public SpringApplicationBuilder configure(SpringApplicationBuilder app) {
        return app.sources(Application.class);
    }
}
```

```
<packaging>war</packaging>
```

- Minimissään käännöksen tuloksen muutos voidaan tehdä yhdellä muutoksella pom.xml tiedostoon
- Lisäksi resources/application.properties tiedostossa voidaan kertoa sovelluksen kontekstin juuri: **server.contextPath=/omasovellus**

# Riippuvuuksien injektointi ja Core-paketti

---

- *Dependency Injection*
- Olio ei itse aktiivisesti hae viittausta toiseen oloon, vaan viittaus välitetään esimerkiksi set-metodin tai konstruktorin kautta suoritussympäristön toimesta
- Helpottaa ohjelmointityötä ja selkeyttää koodia
  - Voi myös vähentää myös riippuvuuksia sovelluskehyskohtaisista rajapinnoista
- Springissä injektointipiirteet kuuluvat Core-pakettiin
  - Eli on siis eri injektointitoteutus kuin Java EE kehyksessä
- Injektointitavat:
  - @Autowired: Springin annotaatio
  - @Inject: JSR-330, lähes täysin sama kuin @Autowired, ainoa ero on @Autowired annotaation required attribuutti joka mahdollistaa että resurssin löytyminen ei aiheuta virhettä
  - @Resource: JSR-250, sama ero @Injectin kanssa kuin standardi Java EE:ssä, eli miten resurssia haetaan
    - resource: nimi>tyyppi>qualifierit / inject+autowired: tyyppi>qualifierit>nimi



# Injektointi: Esimerkki

---

- Yleisin annotaatio on `@Autowired`
  - Tämän avulla voidaan kertoa, että jollekin koodin osalle (set-metodi, jäsenmuuttuja, konstruktori, ...) tulee tehdä resurssin injektointi, esimerkiksi:

```
public class WebBean {  
    @Autowired private Asiakashallinta ah;  
    ...  
}
```

- tai konstruktorin parametrina:

```
public class WebBean {  
    private Asiakashallinta ah;  
    public WebBean(@Autowired Asiakashallinta ah) {  
        this.ah = ah;  
    }  
    ...  
}
```

- Kumpaa käytetään on hieman tilanneriippuvaista, tai mielipidekysymys:  
<https://stackoverflow.com/questions/7779509/setter-di-vs-constructor-di-in-spring>

# Omien tyyppien injektointi

---

- Jotta omien tyyppien olioita voi injektoida täytyy Spring containerin löytää ne
- Spring käyttää neljää annotaatiota komponenttien annotointiin, niiden pääasiallinen ero on semanttinen
- @Component - geneerinen stereotyyppi Springin hallinnoimille komponenteille
- @Service - stereotyyppi palvelukerrokselle
- @Resource - stereotyyppi talletuskerrokselle
- @Controller - stereotyyppi esityskerrokselle (Spring MVC)

# Konfigurointi

---

- Springin luokkia, joilla on Configuration annotaatio voi käyttää alustamiseen
  - Esimerkiksi @SpringBootApplication sisältää myös Configuration annotaation
  - <https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html>
- Luokalle voi lisätä @Bean annotoituja metodeita, jotka palauttavat konfiguroivan olion
- Application luokalle lisätyt Beanit, jotka palauttavat CommandLineRunner-tyyppisen olion voi käyttää esimerkiksi applikaation osien alustamiseen
  - @AutoWired ei pakollinen, ainakaan näin nimettynä. Myös useampi parametri mahdollinen

```
@Bean
public CommandLineRunner alusta(@Autowired HenkiloLista henkiloLista) {
    return (args) -> {
        henkiloLista.lisaa(new Henkilo("Aku", "Ankka", new Date(54, 2, 13)));
        henkiloLista.lisaa(new Henkilo("John", "Doe", new Date()));
        henkiloLista.lisaa(new Henkilo("Walpuri", "Ankka", new Date(100, 3, 1)));
    };
}
```

# Harjoitus

---

- Muokkaa edellistä projektia
- Lisää kontrolleriin `@Autowired` Numero numero muuttuja
- Toteuta luokka Numero, muista `@Component`
- Lisää Numero-luokkaa muuttuja `int` arvo; Sille getteri ja setteri, lisäksi `getAndIncrement()`, joka palauttaa arvon ja kasvattaa arvoa yhdellä
- Lisää kontrollerille uusi request metodi, joka palauttaa numero-muuttujan arvon ja kutsu sitä
- Koita myös toisella ohjelmalla/selaimella, onko kasvatettu arvo sielläkin? Miksi?
- Lisätehtävä, toteuta Application luokkaan `CommandLineRunner`in tyyppinen `@Bean`, ja aseta siellä Numeron alkuarvo arvoon 10

---

## **REST palveluiden toteutus**

---

# REST-pyynnön paluuarvo

---

- Springin `@Controller`-ohjelmointitapa tukee REST-kutsuja lähes automaattisesti, koska URL ohjataan aina erilliselle metodille ja pyynnön parametrit (tai polku- tms. parametrit) määritellään metodin parametreiksi
  - Yleensä `@RestController`
- Ainoa lisä on metodin merkitseminen `@ResponseBody`-annotaatiolla, jolloin kutsujalle palautetaan ainoastaan metodin paluuarvo
  - `@RestController` luokalle lisää automaattisesti `@ResponseBody` annotaation kaikille metodeille
- Usein paluuarvo on JSON-formaatissa; muunnokseen voidaan käyttää Jackson-kirjastoa
  - Muunnos tapahtuu automaattisesti, kunhan pyynnössä on oikea accept-otsikko - Spring Boot käyttää oletusarvoisesti JSONia, muissa toteutuksissa oletuksena saattaa olla XML
  - Spring Bootin starter tuo Jackson riippuvuuden automaattisesti projektiin

# REST-pyynnön pohja kertauksena (plus hieman lisää)

---

- Esimerkki:

```
@RestController
@RequestMapping("/api")
public class RestController {

    @RequestMapping(value = "/restkutsu", method = RequestMethod.GET)
    public Henkilo restMetodi(@RequestParam(name="hs", required = false) String hakusana) {
        Henkilo henkilo = new Henkilo();
        henkilo.setEtunimi("Mikki");
        henkilo.setSukunimi("Hiiri");
        return henkilo;
    }
}
```

- @RestController alussa kertoo, että kaikki metodit palauttavat dataa
  - RestController oikeastaan yhdistää Controller ja.ResponseBody annotaatiot
- Luokan @RequestMapping kertoo, että kaikille tämän luokan palvelu-osoitteille tulee /api alkuun
  - esim. <http://localhost:8080/api/restkutsu>
- Metodin @RequestMapping annotaation ja metodin määrittelyn sijaan voitaisiin käyttää @GetMapping annotaatiota

# Harjoitus

---

- Tee yksinkertainen REST sovellus, joka osaa palauttaa tietoa JSON-muodossa
- Toteuta tätä varten luokka Henkilo, jolla on etunimi, sukunimi ja ika ominaisuudet
- Palauta uusi Henkilo-olio uudesta request-metodista
  
- Lisätehtävä:
  - Toteuta Henkiloita palauttava REST luokka:
  - @GetMapping("/henkilot") - palauttaa listan henkilöitä (alusta lista konstruktorissa/request metodissa)
  - Lisää Request parametri, joka kertoo minkä nimiset henkilöt palautetaan, kaikki jos parametria ei ole - tai se on tyhjä (miksi id ei ole hyvä request parametrina?)



# Parametrit pyynnölle

---

- Parametreja voi olla eri paikoissa
  - Yhdessä pyynnössä tietoa voi tulla useammalla kuin yhdellä tavalla
1. Polkuparametri: asiakkaat/**123**
    - Identifioidaan aliresurssi
  2. Kyselyparametri: asiakkaat?**sort=asc**
    - Annetaan lisätietoa, esimerkiksi filteröintiä tai sorttausta
  3. Pyyntön bodyssa tuleva data, esim. POST tai PUT pyynnöissä

# Esimerkkikoodit

---

- Seuraavilla sivuilla olevat metodit on lisätty alla olevaan kontrolleriin

```
@RestController
@RequestMapping("/api/oppilaat") // Yhteinen alku
public class OppilasController {
    private List<Oppilas> oppilaat;

    public OppilasController() {
        oppilaat = new ArrayList<>(Arrays.asList(
            new Oppilas("Mikki", "Hiiri"),
            new Oppilas("Aku", "Ankka")));
    }

    @GetMapping("") // eli /api/oppilaat, kts. yllä
    public List<Oppilas> palautaKaikki() {
        return oppilaat;
    }
}
```

# Polkuparametrit

---

- Polkuparametreilla määritellään tyypillisesti aliresurssi
- Esimerkiksi yhden oppilaan **Listin** indeksillä palauttavan metodin voisi toteuttaa seuraavasti

```
@GetMapping("/{ind}")  
public Oppilas palautaYksi(@PathVariable(name = "ind", required = true) int ind) {  
    return oppilaat.get(ind);  
}
```

- Siinä määritellään että parametrin nimi on ind, ja että parametri on pakko olla
- Request metodille arvo tulee (annotaatiota lukuun ottamatta) aivan normaalina parametrina

# Pyyntöparametrit

---

- Pyyntöparametreilla rajoitetaan tai muokataan palautettavan resurssin ulkoasua
- Toteutetaan esimerkiksi kaikkien oppilaiden haku ja siihen filteröinti etunimen perusteella
  - Huomaa, että ei voi tehdä uutta metodia, vaan täytyy muokata aiempaa, sillä määpättävä osoite on sama

```
@GetMapping("")
public List<Oppilas> palautaKaikki(@RequestParam(name="haku", required=false) String haku) {
    if (haku==null) return oppilaat;
    List<Oppilas> palautettavat = new ArrayList<>();
    for (Oppilas o : oppilaat) {
        if (o.getEtunimi().toLowerCase().contains(haku.toLowerCase())) {
            palautettavat.add(o);
        }
    }
    return palautettavat;
}
```

## Bodyssa tulevat parametrit

---

- Aiemmin mainittujen URL:n mukana välitettävien parametrien suoran käsittelyn lisäksi on mahdollista muuntaa pyynnössä tuleva JSON-data automaattisesti olioksi `@RequestBody`-annotaatiolla:

```
@PostMapping("")
public void lisääOppilas(@RequestBody Oppilas uusi) {
    oppilaat.add(uusi);
    // Oikeasti palautettaisiin jotain, tästä myöhemmin
}
```

```
$('#painike1').on('click', function () {
    $.ajax({
        url: '/api/oppilaat',
        type: 'POST',
        contentType: 'application/json',
        data: JSON.stringify({ etunimi: 'Roope', sukunimi: 'Ankka' }),
        success: function () {
            $('#tuloste1').text('Lisätty');
        }
    });
});
```

# HTTP-statuskoodit

---

- HTTP-statuskoodin voi asettaa kontrollerin metodien annotaatiolla `@ResponseStatus`, tai mieluummin asetetaan metodin rungossa
- Tyypillisiä statuskoodeja, jotka löytyvät `HttpStatus`-luokasta:
  - 200, OK: Onnistunut GET, joka palauttaa sisältöä
  - 201, CREATED: POST tai PUT loi uutta sisältöä; `Location`-otsikon tulisi palauttaa luodun sisällön URL
  - 204, NO\_CONTENT: Onnistunut pyyntö ilman sisältöä, esimerkiksi DELETE tai PUT-päivitys
  - 400, BAD\_REQUEST: Virheellinen pyynnön formaatti
  - 401, UNAUTHORIZED: Käyttäjä ei kirjautunut
  - 403, FORBIDDEN: Ei oikeuksia
  - 404, NOT\_FOUND
  - 405, METHOD\_NOT\_ALLOWED: HTTP-pyyntö ei tuettu; `Allow`-otsikon tulisi palauttaa sallitut tyypit
  - 409, CONFLICT: Päivityskonflikti, esimerkiksi PUT ja optimistisen lukituksen virhe; vastaus voi sisältää eri versiot datasta

# Location-otsikko

---

- Otsikon täytyy olla täysi URL
- Enkoodauksessa auttaa esimerkiksi Spring:n valmis `UriTemplate`-luokka, tai `ServletUriComponentsBuilder` luokka
  - Jälkimmäisestä esimerkki seuraavalla sivulla

```
@PostMapping("/asiakas")
@ResponseStatus(HttpStatus.CREATED)
public void luoAsiakas(@RequestParam String nimi,
    HttpServletRequest pyynto, HttpServletResponse vastaus) {
    logi.info("luoAsiakas(): POST, nimi: " + nimi);
    int luotuId = 123;
    UriTemplate osoite = new UriTemplate(
        pyynto.getRequestURL() + "?id={id}");
    vastaus.addHeader("Location", osoite.expand(luotuId).toASCIIString());
}
```

# ResponseEntity

---

- Springin luokka, jonka avulla voidaan muokata palautettavaa tietoa
  - <https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/ResponseEntity.html>
- Erityisen hyödyllinen kun HttpStatus täytyy asettaa itse
- Voidaan luoda konstruktorilla, tai builder metodeilla
  - `new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED);`
  - `ResponseEntity.created(location).header("MyResponseHeader", "MyValue").body("Hello World");`

```
@PostMapping("/aihe")
public ResponseEntity<?> uusiAihe(@RequestBody Aihe aihe) {
    aihe = palvelu.talleta(aihe);
    if (aihe==null)
        return ResponseEntity.noContent().build();
    URI location = ServletUriComponentsBuilder
        .fromCurrentRequest().path("/{id}")
        .buildAndExpand(aihe.getId()).toUri();
    return ResponseEntity.created(location).build();
}
```



# Harjoitus

---

- Toteuta REST palvelu, jolla voit käsitellä sanontoja/aforismeja - **tai** voit myös jatkaa Henkiloiden parissa - toteuta seuraavat toiminnot
  - Katsoa (GET) - kaikki, tai yksi
  - Lisätä (POST)
  - Muuttaa (PUT)
  - Poistaa (DELETE) - kaikki, tai yksi
- Tee palvelua varten uusi kontrolleri SanontaKontrolleri, joka käyttää osoitetta "/aforismit"
- Aforismi on Java-luokka, jossa on muuttujina
  - `int id;` // esimerkiksi `id = seuraavaID++` (kun `seuraavaID` on staattinen)
  - `String teksti;` // aforismin teksti
  - `String sanoja;` // keneltä aforismi on peräisin
- Toteuta myös luokka AforismiLista, joka on käytännössä wrapperi `List<Aforismi>` muuttujan ympärille. Lisää AforismiLista jäsenmuuttujaksi SanontaKontrolleriin. Luo luokasta olio kontrollerin konstruktorissa, ja lisää siihen valmiiksi pari sanontaa
  - AforismiLista: `lisaa(Aforismi)`, `etsi(int id)`, `muuta(int id, Aforismi uusi)`, `poista(int id)`
- Toteuta yllä mainittu CRUD toiminnallisuus myös SanontaKontrolleriin
- Käytä curl'illa tai Postmanilla

# Muun kuin JSONin palauttaminen - Extraa\*\*

- Tekstin palauttaminen onnistuu helposti, String paluutyypinä palauttaa plain tekstiä
  - Content-Type: text/plain;charset=UTF-8
- XML palauttaminen ei onnistu automaattisesti (*406 Not Acceptable*)

406

```
C:\>curl -i -H"Accept: application/xml" http://localhost:8080/api/henkilo
HTTP/1.1 406
Content-Length: 0
Date: Wed, 20 Dec 2017 11:27:53 GMT
```

- Nopea tapa saada XML-tuki, jos XML muodolla ei niin väliä, on lisätä riippuvuus Jacksonin XML dataformaattiin mukaan projektiin. **Huom: Java 10 käytettäessä edes tämä ei riitä..**

```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

- Sitten:

```
C:\>curl -i -H"Accept: application/xml" http://localhost:8080/api/henkilot
HTTP/1.1 200
Content-Type: application/xml;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 20 Dec 2017 11:36:56 GMT
```

```
curl -i .../api/henkilot.xml tai henkilot.json
```

# Paluutyypin konfigurointi - Extraa\*\*

---

- Voidaan myös konfiguroida esimerkiksi XML oletuspaluutyypiksi

```
@Configuration
@EnableWebMvc
public class Konfiguraatio extends WebMvcConfigurerAdapter {
    @Override
    public void configureContentNegotiation(ContentNegotiationConfigurer config) {
        config.ignoreAcceptHeader(true)
            .favorPathExtension(true)
            .defaultContentType(MediaType.APPLICATION_XML);
    }
}
```

- Huom. vaikka yllä palvelu palauttaa palvelu.json tyyppisellä kutsulla JSON dataa, niin esimerkiksi POST yhteydessä kutsujan on annettava Content-type

---

## Spring test

---

# Spring Boot sovellusten testaaminen

---

- Spring Test sisältää test-runnerin, jonka avulla on helppo luoda integraatiotestit
- Test kehys koostuu kahdesta moduulista: spring-boot-test ja spring-boot-test-autoconfigure. Helpointa on ottaa mukaan riippuvuus spring-boot-starter-test, joka tuo mukaan molemmat moduulit, sekä tuen mm. JUnit, Hamcrest Mockito ja JSONassert kirjastot

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
  <version>2.0.3.RELEASE</version>
</dependency>
```

- Testien ajaminen SpringRunner luokalla

```
@RunWith(SpringRunner.class)
public class RestKontrolleriTest {
    @Test
    public void testimetodi() { /* testit.. */ }
}
```

# Testaus

---

- Yksikkötestaaminen onnistuu normaalisti JUnit ja Hamcrest metodeilla
- Integraatiotestejä varten löytyy paljon tukea
  - REST toteutusten testaamisessa auttaa `@TestRestTemplate` luokka (esimerkki seuraavalla sivulla)
  - Jsonia palauttavat testiluokat saa autokonfiguroitua käyttäen `@JsonTest` annotaatiota
  - Mock olioiden käyttöön saa tuen esimerkiksi määrittelemällä `@MockBean`in ja tekemällä testikonfiguraation - tästä lisää tietokantojen yhteydessä
  - JPA testejä varten on `@DataJpaTest`
- Verkosta löytyy paljon apua, pari helposti ymmärrettävää dokumenttia:
  - <https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html>
  - <http://www.baeldung.com/spring-boot-testing>

# REST palvelun ja TestRestTemplaten

---

- Spring sisältää RestTemplate luokan palvelun asiakkaan tekemiseen tai testaamiseen, testeissä on kuitenkin hyvä käyttää sen varianttia TestRestTemplate jonka voi injektoida testiluokkaan mukaan

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HenkiloApiTest {
    @Autowired
    private TestRestTemplate restTemplate;
    @Test
    public void etsi_henkilo() {
        ResponseEntity<Henkilo> response = restTemplate.getForEntity("/api/henkilot/0",
                                                                    Henkilo.class);
        assertEquals(200, response.getStatusCodeValue());
        Henkilo henkilo = response.getBody();
        assertNotNull(henkilo);
        assertEquals(henkilo.getEtunimi(), "Aku");
    }
}
```

# TestRestTemplate ja Listat/taulukot

---

- RestTemplaten `getForEntity()` toimii hienosti yhdelle oliolle, mutta jos palvelu palauttaa useamman olion, niin siinä vaiheessa tilanne muuttuu hieman mutkikkaammaksi
- Tyypin määrittäminen tyypitettyillä tyypeillä ei onnistukaan `.class` kentän avulla, vaan täytyy käyttää avuksi tyyppiä `ParameterizedTypeReference`
- RestTemplaten `exchange()` metodi mahdollistaa tyypityksen, ja palauttaa esimerkiksi `ResponseEntity<List<OmaTyyppi>>` tyyppisen vastauksen
- Koko koodi seuraavalla sivulla..

```
ResponseEntity<List<Henkilo>> response =  
    restTemplate.exchange("/api/henkilot", HttpMethod.GET,  
        null,  
        new ParameterizedTypeReference<List<Henkilo>>() {}  
    );
```



```

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.test.context.junit4.SpringRunner;
import java.util.List;
import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class RestKontrolleriTest {
    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testaaAforismilistaEiOleNull() {
        ResponseEntity<List<Henkilo>> response =
            restTemplate.exchange("/api/henkilot", HttpMethod.GET, null,
                new ParameterizedTypeReference<List<Henkilo>>() {}
            );
        assertThat(response.getStatusCodeValue(), is(200));
        List<Henkilo> henkilot = response.getBody();
        assertThat(henkilot, is(notNullValue()));
    }
}

```

# POST, PUT ja DELETE

---

- Jos POST palauttaa muutetun olion, niin postForEntity() toimii kuten getForEntity()

```
ResponseEntity<Henkilo> response = restTemplate
    .postForEntity(URI.create(url), henkilo, Henkilo.class);
```

- DELETEn kanssa voi käyttää esimerkiksi exchange() metodia

```
ResponseEntity<?> resp = restTemplate.exchange(poistettavavurl,
    HttpMethod.DELETE, null, Object.class, id);
```

- PUT on näistä kolmesta yleensä haastavin. Yksi tapa on ensin luoda RequestEntity tyyppisen olio, jota sitten käytetään varsinaisessa pyynnössä exchange() metodin kanssa

```
RequestEntity<Henkilo> requestEntity =
    new RequestEntity<>(muokattavaHlo, HttpMethod.PUT, new URI(hlonOsoite));
ResponseEntity<Henkilo> response =
    restTemplate.exchange(requestEntity, Henkilo.class);
```

# Harjoitus

---

- Toteuta testit edellisen harjoituksen REST palvelulle
- Muista Maven projektin rakenne: src/test/java hakemistossa testikoodit
- Käytä samaa pakettia kuin missä Application luokka sijaitsee

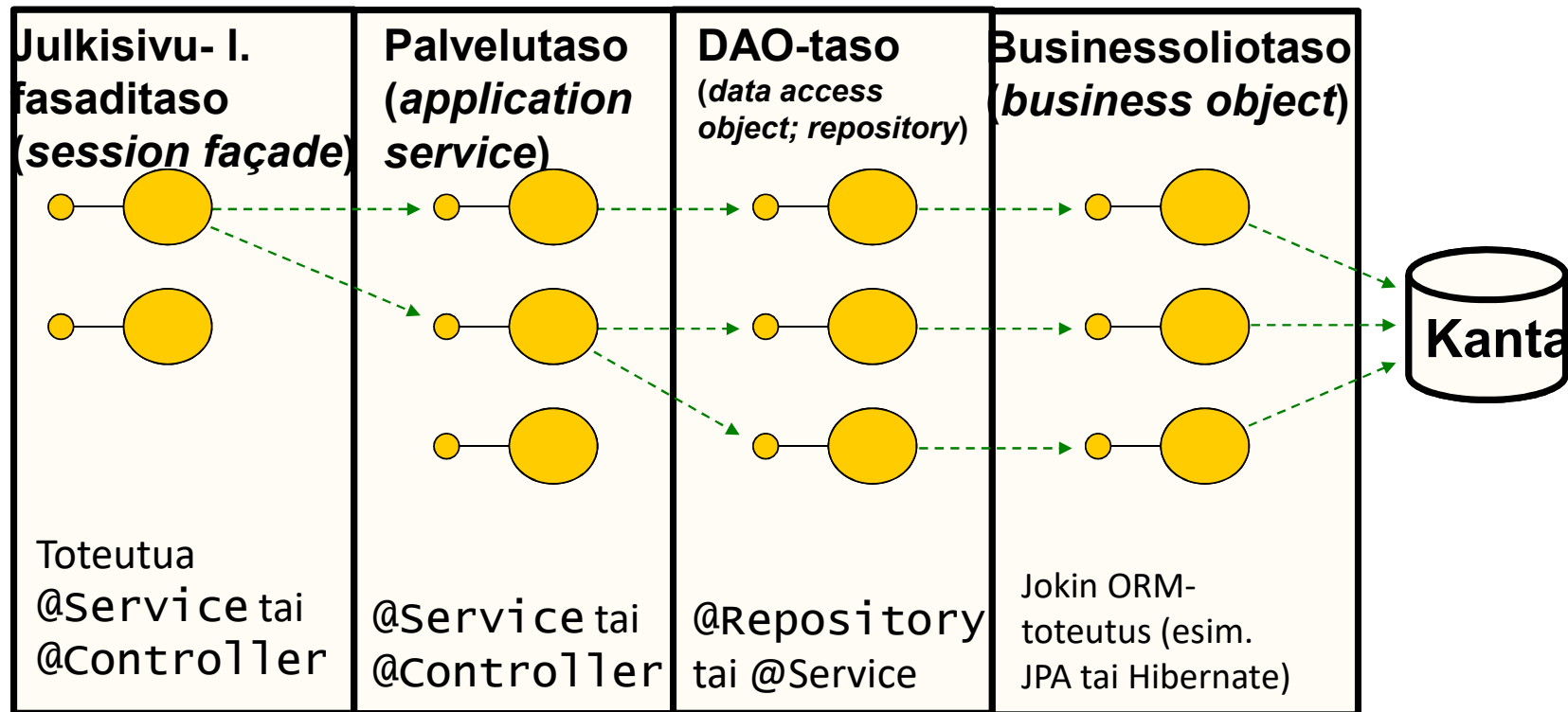
---

## Tietokantakäsittely - Spring Data

---

# Sovelluksen sisäiset tasot

- Tyypillisiä Spring-perustaisesta ratkaisusta löytyviä sovelluksen sisäisiä tasoja:



# Sovelluksen sisäiset tasot

---

- **Fasaditaso**
  - Fasaditaso on kuvan tasoista harvinaisin
  - Tarvitaan, jos yhden operaation suorittaminen (esimerkiksi uuden asiakkaan luominen) tarvitsee useita palvelutason kutsuja ja tämä monimutkaisuus halutaan piilottaa asiakassovelluksilta
- **Palvelutaso**
  - Palvelutaso on tasoista vakiintunein
  - Täällä oleva logiikka liittyy paljolti seuraavan tason kutsujen yhdistämiseen (kuten kuvassa) ja niiltä haetulla datalla operoimiseen
- **DAO/repository-taso**
  - Business-olioiden käsittelyn logiikkaan liittyvät piirteet kannattaa mahdollisesti erottaa omaksi tasokseen
    - Esimerkiksi haut voivat sisältää logiikkaa, jota pelkillä relaatioilla ei voida ilmaista (tilaukset ja niihin liittyvät tilausrivit, joilla on toimittamattomia tuotteita)
- **Businessoliotaso**
  - Business-oliotasolle löytyy paljon teknisiä toteutusvaihtoehtoja (kuva) ja joissakin arkkitehtuureissa taso voi myös puuttua
    - Jos taso puuttuu, se on usein korvattu proseduraalistyylisellä kannan käsittelyn koodilla
  - Tälle tasolle pyritään sijoittamaan mahdollisimman paljon logiikkaa; esimerkiksi laskua kuvaavan komponentin viitenumeron laskennan algoritmi tulisi tänne

# Yleistä

---

- Koska tietokantakäsittely tulee vastaan lähes kaikissa enterprise-sovelluksissa, tarjoaa Spring siihen liittyen paljon erilaisia apuluokkia
- Spring:n palveluita ovat:
  - Tietokantayhteysaltaat ja transaktiohallinta
  - Apuluokat erityisesti JDBC:n yhteydessä (resurssien sulkeminen ja tulosjoukon käsittely)
  - Poikkeusten yhtenäistäminen
- Itse kantakäsittely on mahdollista tehdä:
  - Joko JDBC:llä...
  - tai käyttäen jotakin yleistä persistenssisovelluskehystä (tuettuja ovat Java Persistence API (JPA), Hibernate, JDO, Oracle TopLink, Apache OJB ja MyBatis)

# Tietolähteen asetukset

---

- Muistikanta H2 on oletusarvoisesti käytössä. Muille tarvitsee esitellä kannan asetukset
- Tietokantayhteysallas (`javax.sql.DataSource`) on yksinkertaisinta esitellä kuvaustiedostossa; esimerkki:
  - `resources\application.properties` tiedosto:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/kurssi
spring.datasource.username=jdbckayttaja
spring.datasource.password=salasana
```

- Tarvitset myös ajurin, PostgreSQL esimerkki alla. Ajurin saa helpoiten valitsemalla sen SpringInilizr projektin luonnin yhteydessä

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.2.5</version>
  <scope>runtime</scope>
</dependency>
```



# Usean tietolähteen käyttäminen

- Toteuta kaksi määritystä application.properties tiedostoon, tee @Configuration luokka joka määrittää ne molemmat, ja kerro jos haluat käyttää muuta kuin primaaria

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/world?useSSL=false
spring.datasource.username=jdbc
spring.datasource.password=salasana

spring.kurssikanta.driver-class-name=org.postgresql.Driver
spring.kurssikanta.url=jdbc:postgresql://localhost:5432/kurssi
spring.kurssikanta.username=jdbc
spring.kurssikanta.password=salasana
```

```
@Configuration
public class DbConfiguration {
    @Bean(name = "worldkanta")
    @Primary
    @ConfigurationProperties(prefix="spring.datasource")
    public DataSource worldDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "kurssikanta")
    @ConfigurationProperties(prefix="spring.kurssikanta")
    public DataSource kurssiDataSource() {
        return DataSourceBuilder.create().build();
    }
}
```

```
private JdbcTemplate jdbcTemplate;

@Autowired
public RestHenkiloKontrolleri(DbConfiguration config) {
    jdbcTemplate = new JdbcTemplate(config.kurssiDataSource());
}
```

# Tietokannan käsittelyn apuluokkia

---

- Tavanomaisessa JDBC-ohjelmoinnissa suurin osa koodiriveistä kuluu resurssien hallintaan (yhteyden, lauseen ja tulosjoukon luonti sekä kaikkien näiden sulkeminen `finally`-lohkossa -tai `try-with resources`)
- Spring:ssä esimerkiksi luokka `org.springframework.jdbc.core.JdbcTemplate` helpottaa ohjelmointia merkittävästi
- Luokan esittely kuvaustiedostossa:

```
@Autowired  
JdbcTemplate jdbc;
```

- Maven-riippuvuus tulee helposti `spring-boot-starter-jdbc` lisäyksellä

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-jdbc</artifactId>  
</dependency>
```

# Tietokannan käsittelyn apuluokkia

---

- JdbcTemplate-luokan käyttö koodissa; perushaku:

```
List<String> asiakkaat = jdbcPohja.queryForList(  
    "SELECT nimi1 FROM Asiakas ORDER BY nimi1",  
    String.class);
```

- Parametrisoitu haku:

```
List<String> asiakkaat = jdbcPohja.queryForList(  
    "SELECT nimi1 FROM Asiakas WHERE nimi1 LIKE ? ORDER BY nimi1",  
    String.class, "Laine%");
```

# Tietokannan käsittelyn apuluokkia

---

- Jos halutaan palauttaa muita olioita kuin primitiivityyppejä tai merkkijonoja, täytyy luokan kuvaaminen ilmaista erikseen
- Esimerkki (RowMapper toteutettu anonymyminä sisäluokkana), AsiakasTO on seuraavalla sivulla:
  - Java8 yhteydessä käytetään usein lambda-lausetta nimettömän sisäluokan sijaan

```
List<AsiakasTO> asiakkaat = jdbcPohja.query(
    "SELECT nimi1, nimi2 FROM Asiakas ORDER BY nimi1",
    new RowMapper<AsiakasTO> () {
        public AsiakasTO mapRow(ResultSet tulosjoukko, int indeksi)
            throws SQLException {
            AsiakasTO asiakas = new AsiakasTO(
                tulosjoukko.getString("nimi2"),
                tulosjoukko.getString("nimi1"));
            return asiakas;
        }
    });
```

# Tietokannan käsittelyn apuluokkia\*

---

- Esimerkki-luokka:

```
public class AsiakasTO implements Serializable {  
    private static final long serialVersionUID = 1L;  
    private String etunimi;  
    private String sukunimi;  
  
    public AsiakasTO() { }  
  
    public AsiakasTO(String etunimi, String sukunimi) {  
        this.etunimi = etunimi;  
        this.sukunimi = sukunimi;  
    }  
  
    public String getEtunimi() {  
        return etunimi;  
    }  
  
    public void setEtunimi(String etunimi) {  
        this.etunimi = etunimi;  
    }  
  
    public String getSukunimi() {  
        return sukunimi;  
    }  
  
    public void setSukunimi(String sukunimi) {  
        this.sukunimi = sukunimi;  
    }  
}
```

# Harjoitus

---

- Toteuta REST rajapinta, joka listaa world-kannan maat
- Luokat
  - *ProjektiApplication*: main() metodi, käynnistää palvelimen - Spring Initializr generoi, *älä muuta*
  - *MaaController*: REST rajapinnan toteutus, @GetMapping("/api/maat"). Ruiskuta *MaaDAO*
  - *MaaDAO*: DAO-kerros, tietokannan kanssa kommunikointi. Ruiskuta *JdbcTemplate*
  - *Maa*: vastaa tietokannan country-aulua, jäsenmuuttuja vastaa saraketta. Ei tarvitse toteuttaa kaikkia sarakkeita
- Kokonaisuuden toimintalogiikka: REST client (esim. cURL/Postman) kutsuu palvelinta, joka ohjaa pyynnön *kontrollerin* palvelumetodille. Kontrolleri pyytää tiedot *DAO-luokalta*, joka hakee ne tietokannasta ja palauttaa listan *Maa-olioita*
- **Lisätehtäviä:**
  - Listaa myös muiden taulujen tietoja.
  - Ota liitokset mukaan, eli tulostuloksessa voisi olla esimerkiksi pääkaupungin nimi eikä kaupungin id, kuten oletuksena tulee

# Tietokannan käsittelyn apuluokkia

---

- Spring 2.0 toi mukanaan myös mahdollisuuden käyttää nimettyjä parametreja paketin `org.springframework.jdbc.core.namedparam` luokan `NamedParameterJdbcTemplate` avulla
- Esimerkki:

```
NamedParameterJdbcTemplate jdbcNPPohja =  
    new NamedParameterJdbcTemplate(jdbcPohja);  
Map<String, String> hakusanat =  
    new HashMap<String, String>();  
hakusanat.put("nimi1", "Laine%");  
List<String> asiakkaat = jdbcNPPohja.queryForList(  
    "SELECT nimi1 FROM Asiakas WHERE nimi1 LIKE :nimi1",  
    hakusanat, String.class);
```

# Poikkeusten käsittely

---

- Spring tarjoaa laajan luokasta `org.springframework.dao.DataAccessException` periytetyn poikkeushierarkian, jonka tarkoituksena on toimia aina samalla tavalla riippumatta tietokannan käsittelyn tekniikasta
  - Eli suora JDBC, JDO, Hibernate, JPA etc. heittävät aina samassa tilanteessa saman poikkeuksen
  - Osaa käsitellä myös joitakin kantakohtaisia poikkeuksia
- Poikkeus on tyypiltään järjestelmäpoikkeus (*unchecked exception*), joten kutsujan ei tarvitse käsitellä sitä kuin erityisesti halutessaan



# Harjoitus

---

- Lisää edelliseen harjoitukseen koodia, jolla voit listata vain halutut maat. Eli toteuta REST-palveluun filteröinti maiden hakuun
- Filteröinti tehdään kyselyparametrien avulla, eli lisää palvelumetodin parametriksi `@RequestParam` -annotoitu parametri, jota tutkimalla selviää halutaanko filteröidä vai palautetaanko kaikki
  - Esimerkki metodista, jonka "filteri" kyselyparametri on vapaaehtoinen. Jos käyttäjä antaa parametrin sillä on arvo, jos parametria ei anneta niin muuttujan hakusana arvo on null
  - ```
public List<Henkilo> haeHenkilot(@RequestParam(name="filteri", required = false) String hakusana) {  
    ...  
}
```

# CRUD

---

- Pelkän kyselyn lisäksi on hyvä olla hallussa ainakin perusteet myös muille toiminnoille tietokannan kanssa
- Alla yksinkertaiset esimerkit DELETE, INSERT ja UPDATE -komentojen käyttöön

```
public int deleteById(int id) {
    return jdbcTemplate.update("delete from henkilo where id=?", new Object[] { id });
}

// Tästä parempi versio hieman myöhemmin..
public int insert(Henkilo henkilo) {
    return jdbcTemplate.update("insert into henkilo (nimi, ika) " + "values(?, ?)",
        new Object[] { henkilo.getNimi(), henkilo.getIka() });
}

public int update(Henkilo henkilo) {
    return jdbcTemplate.update("update henkilo set nimi = ?, ika = ? where id = ?",
        new Object[] { henkilo.getNimi(), henkilo.getIka(), henkilo.getId() });
}
```

# Kontrolleri ja CRUD

---

- Kontrollerin perustoteutus CRUD toiminnoissa on jo käyty läpi
  - GET - helppo, palauta List<Tyyppe>, tai Tyyppe tai ResponseEntity<Tyyppe>
  - DELETE - helppo
  - POST - tyypillisin: luo ResponseEntity olio, jonka Headers osassa on asetettu Location-määre. **Vaatii JdbcTemplaten käyttäjältä Generoitujen avainten kaivamisen**
  - PUT - helppo, ota id polkuparametrina, päivitä sen avulla löydettävä resurssi pyynnön Bodyssa olevien tietojen mukaisesti
- Generoidut avaimet JdbcTemplaten kanssa saa selville, mutta hieman eri tavalla kuin normaalin JDBC:n kanssa
- Avaimet saa, kun käyttää esimerkiksi PreparedStatementCreator luokkaa, jota sitten käytetään JdbcTemplate.update metodin kanssa - tämän toinen parametri on KeyHolder jonka arvo (eli avain) muokkaantuu kutsun aikana
  - Esimerkki seuraavalla sivulla

# INSERT ja generoitu avain

---

```
public int insert(Henkilo henkilo) {
    KeyHolder keyHolder = new GeneratedKeyHolder();

    PreparedStatementCreator psc = connection -> {
        PreparedStatement ps = connection
            .prepareStatement("insert into henkilo (nimi, ika) values(?, ?)",
                Statement.RETURN_GENERATED_KEYS);
        ps.setString(1, henkilo.getNimi());
        ps.setInt(2, henkilo.getIka());
        return ps;
    };

    jdbcTemplate.update(psc, keyHolder);
    int id = keyHolder.getKey().intValue();
    henkilo.setId(id);
    return id;
}
```

# Harjoitus

---

- Toteuta REST projektiin muutakin toiminnallisuutta kuin vain tietojen hakua
- Tee siis CRUD toteutus, eli tarjoa kaikki neljä Luo, lue, päivitä ja poista -toiminnot
  - Muista että world-kannan voi palauttaa alkuperäiseen muotoon luomalla tietokannan haetun skriptin perusteella - silti saa olla hieman varovainen siinä mitä kantaan tekee
- Aiemmat harjoitukset käsittelivät maita, niiden parissa on hyvä jatkaa
- Muokkaa esimerkiksi Suomen valtionpäämies ajan tasalle, lisää uusi maa listalle
- **Lisätehtävät:**
  - ota myös city ja country\_language taulut käsittelyyn ja tarjoa niillekin samaa toiminnallisuutta. Muista myös viite-eheyden säilyttäminen

---

## H2

---

**Sovelto**

# H2 - muistipohjainen tietokanta

---

- H2 muistipohjaisuus tarkoittaa sitä, että koko tietokanta on tietokoneen muistissa - eli kun applikaatio sammutetaan, niin kaikki tietokannan tiedot katoavat
  - On mahdollista käyttää myös tiedostopohjaisena, eli tällöin tietokannan sisältö talletetaan yhteen tiedostoon jolloin muutokset saadaan pysyvämmiksi
- Spring Bootin, tai tarkemmin Spring Datan, oletusarvoinen tietokanta on juuri H2 ja Springin application.properties sisältääkin seuraavat oletusarvot

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

# H2 mukaan projektiin

---

- H2 kirjastot, ja ajurin, saa käyttöön normaalilla dependency-määrittelyksellä

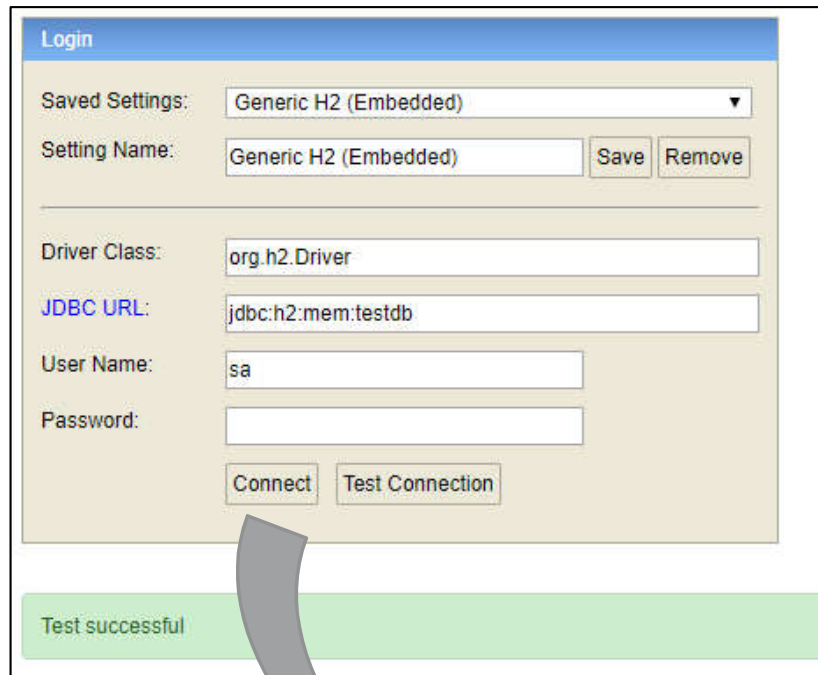
```
<dependency>  
  <groupId>com.h2database</groupId>  
  <artifactId>h2</artifactId>  
  <scope>runtime</scope>  
</dependency>
```

- Vaikka tietokanta on muistissa, niin sen sisältöön pääsee käsiksi H2-konsolilla
- Lisää application.properties tiedostoon `spring.h2.console.enabled=true`
- Kun Spring Boot applikaatio on käynnissä, niin konsoli löytyy selaimella
  - <http://localhost:8080/h2-console>



# Konsolin käyttö

<http://localhost:8080/h2-console>



Login

Saved Settings: Generic H2 (Embedded) ▼

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

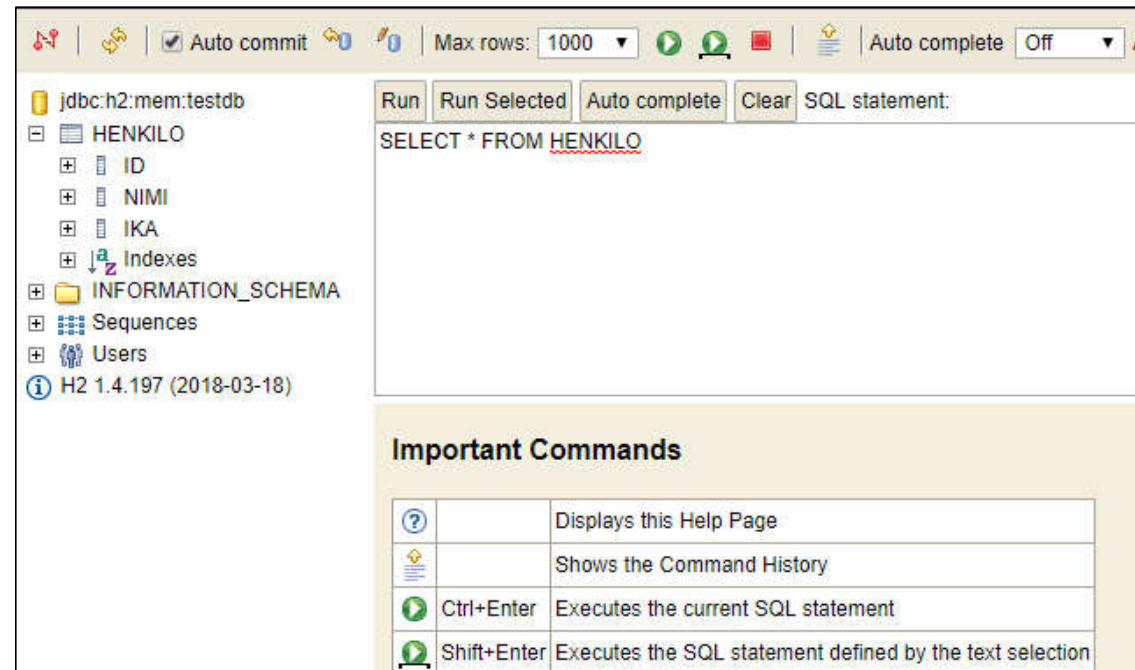
JDBC URL: jdbc:h2:mem:testdb

User Name: sa

Password:

Connect Test Connection

Test successful



Auto commit Max rows: 1000 Auto complete Off

jdbc:h2:mem:testdb

HENKILO

- ID
- NIMI
- IKA
- Indexes

INFORMATION\_SCHEMA

Sequences

Users

H2 1.4.197 (2018-03-18)

Run Run Selected Auto complete Clear SQL statement:

SELECT \* FROM HENKILO

Important Commands

?		Displays this Help Page
		Shows the Command History
▶	Ctrl+Enter	Executes the current SQL statement
▶	Shift+Enter	Executes the SQL statement defined by the text selection

# Kannan alustaminen

---

- H2 kannan voi alustaa kirjoittamalla JDBC:n avulla koodia esimerkiksi Configuration Beaniin, tai lisäämällä projektiin erityisiä SQL-tiedostoja
- Jos `src/main/resources` hakemistossa löytyy `schema.sql` ja/tai `data.sql` nimiset tiedostot, niin niissä sijaitseva SQL-koodi suoritetaan automaattisesti projektin käynnistyessä (murre eri kuin PostgreSQL:n kanssa)
- Schema tarkoittaa rakennetta, siellä määritelläänkin taulujen rakenne

```
create table henkilo (  
    id integer auto_increment not null, nimi varchar(255) not null,  
    ika integer not null, primary key(id)  
);
```

- Data taas on tietoa, ja data.sql käsittää tyypillisesti joukon INSERT-komentoja

```
insert into henkilo(nimi, ika) values('Tara', 17);  
insert into henkilo(nimi, ika) values('Robin', 16);
```

## H2 kanta testauksessa

---

- H2 toimii mainiosti myös testejä tehdessä, sillä se on kevyt ja nopea ja sitä voi näin käyttää helposti varsinaisen kannan sijaan
- Testejä varten voidaankin määritellä omat tietokanta-asetukset helposti: luo testien resources hakemistoon oma application.properties tiedosto, jossa määritellään testeissä käytettävä tietolähde

# Harjoitus

---

- Tee uusi projekti, joka käyttää H2 tietokantaa ja tekee kannalle REST rajapinnan
  - Spring Boot tukee useaa eri tietolähdettä samassa projektissa, sen toteutus olkoon lisätehtävä (eli tee edellisten harjoitusten projektiin uusi kontrolleri ja tee tietolähdeasetukset uusiksi)
- Tietokanta voi olla hyvin yksinkertainen, esimerkiksi Henkilo-taulu tai Sanontoja tai muuta vastaavaa

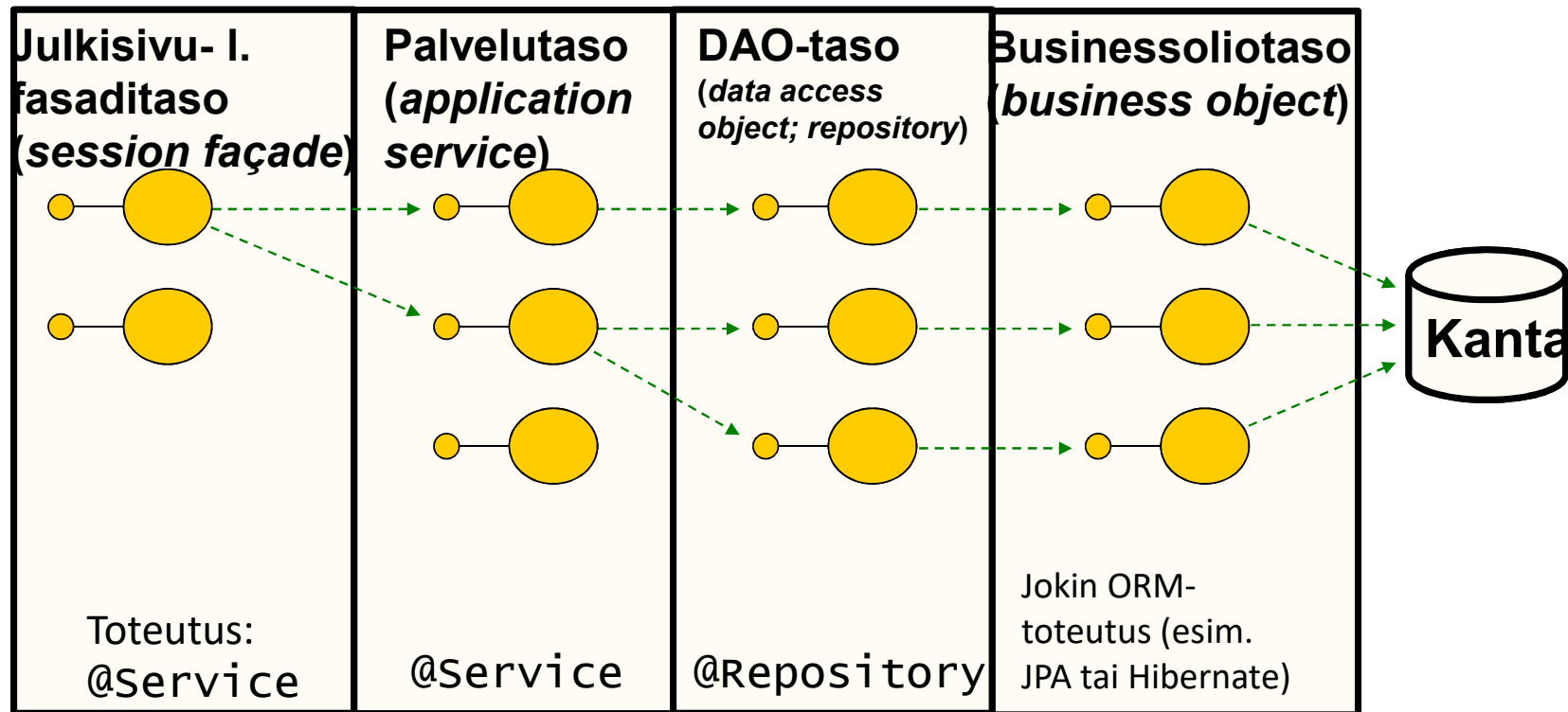
---

## Spring Data ja JPA

---

# Kertaus: Sovelluksen sisäiset tasot

- Tyypillisiä Spring-perustaisesta ratkaisusta löytyviä sovelluksen sisäisiä tasoja:



# JPA: Autogeneroitu DAO-kerros - Repository

---

- Spring Datan JPA-toteutus tarjoaa kerroksen normaalin JPA:n päälle
- JPA jo itsessään helpottaa tietokannan käsittelyä EntityManagerin avulla
- Spring Datan repositoryt mahdollistavat DAO-kerroksen generoinnin Springin avulla - yksinkertaisen esittelyn perusteella
  - Interface'in perusteella Spring luo uuden Beanin, jonka avulla kantaa voi käyttää Entity-olioiden avulla
- `T; public interface HenkiloRepository extends CrudRepository<Henkilo, Long> { }`
- CrudRepository mahdollistaa nimensä mukaisesti olioiden luonnin, haun, päivityksen ja tuhoamisen kannasta
  - <https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html>
- Repository-tyyppejä on CrudRepositoryn lisäksi mm. sen aliluokat: JpaRepository ja PagingAndSortingRepository

# Entity-luokat

---

- Entity luokka vastaa tyypillisesti taulua tietokannassa
- Vaatimuksia luokalle:
  - Luokassa on oltava julkinen tai `protected`-näkyvyydelle määritelty parametraton konstruktori (oletuskonstruktori luonnollisesti käy)
  - Luokan on oltava tavallinen luokka (ei esimerkiksi enum)
  - Luokka ei voi olla `final`, kuten eivät myöskään sen metodit tai persistoitavat jäsenmuuttujat
  - Mikäli entiteettiä on tarkoitus välittää etäkutsutekniikoilla, tulee sen toteuttaa `Serializable`-rajapinta
  - Luokka voi olla abstrakti
  - Instanssimuuttujiin saa viitata vain luokan sisältä, joten niiden tulee olla näkyvyydeltään `private`, `protected` tai oletus
    - Asiakkaiden tulee siis käyttää joko `get-/set`-metodeja tai muita business-metodeja datan lukemiseen
  - Entiteetillä on oltava primääriavain
    - Suositus: Entiteetillä on erillinen identiteettikenttä, jota ei käytetä toiminnallisuuteen vaan se varataan täysin JPA:n ja tietokannan käyttöön



# Entiteetin identiteetti

---

- Voidaan määritellä yhdeksi kentäksi annotaatiolla @Id
  - Id-kenttä pakollinen jos ei ole @Embedded
- Tietokanta tulisi suunnitella siten, että kaikkiin tauluihin luodaan erillinen PRIMARY KEY -sarake:

```
@Id  
private long id;
```

- Nykyään sarake on käytännöllisesti katsoen aina auto-id-tyyppinen (eli kanta luo arvon):

```
@Id @GeneratedValue  
private long id;
```

- Jos taulun avain on koosteinen, käytetään yleensä @Embedded tyyppistä avainta

```
@EmbeddedId  
private MyTablePK mytablePK;
```

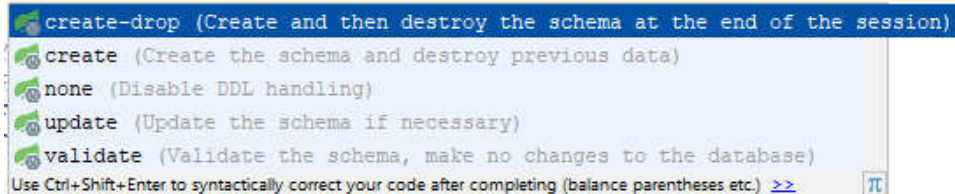
# Yhteyden määrittäminen

- Tiedostossa `resources\application.properties` määritellään tietolähteen (DataSource) asetukset. Lähes kuten "normaalin" Spring datan kanssa, mutta muutamalla JPA-lisäyksellä

```
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost:5432/kurssi
spring.datasource.username=postgres
spring.datasource.password=Salasana

# Seuraava kertoo miten JPA käsittelee taulurakennetta, kts. kuva alla
spring.jpa.hibernate.ddl-auto=create-drop

# Lob-tuki kun Postgres käytössä ja kannassa isoja teksti/binaarisarakkeita
# Ilman näitä tulee virheitä alustuksessa eikä toimi
spring.jpa.properties.hibernate.temp.use_jdbc_metadata_defaults = false
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL9Dialect
```



# Harjoitus - alkuun, eli Hello World sample kanta

---

- Toteuta projekti, joka käyttää seuraavia tekniikoita:
  - Spring Boot
  - PostgreSQL
  - Spring Data
  - JPA
- Luo uusi projekti, pom.xml voi olla kuten aiemmissa harjoituksissa, mutta lisää riippuvuudet PostgreSQL ajurille ja spring-starter-data-jpa
  - Eli Core > DevTools / Web > Web / SQL > JPA ja PostgreSQL
- Lisää kontrolleri ***MaailmaController***
- Muokkaa konfiguraatiotiedostoa ***application.properties*** ja määrittele siellä tietokantayhteys
  - Esimerkki materiaalissa, url: jdbc:postgres://localhost:5432/world
  - Käyttäjä ja salasana
  - **Huom.** `spring.jpa.hibernate.ddl-auto=none`, tai jätä pois: oletus on none

# Harjoitus - jatkuu

---

- Lisää projektiin **Country Entity** luokka, siihen valitut Country-taulun kentät
  - Esimerkiksi code (id), name, localname, population, indepyear ja headOfState
  - Käytä Integer tyyppiä, äläkä int - osa kentistä on nullable ja int tyyppinä aiheuttaa poikkeuksen
- Lisää projektiin repository interface **CountryRepository**, peri se CrudRepository interface'ista
- Ruiskuta repository kontrollerille jäsenmuuttujaksi
- Lisää kontrolleriin request metodi joka palauttaa kaikki maat GET kutsulla
- Testaa että kaikki toimii

# Repository tarkemmin

---

- Repositoryja on monenlaisia, mutta niitä voi määritellä itse - tai olemassa olevia voi laajentaa helposti
  - <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories>
- CrudRepository tarjoaa perustoiminnot, mutta niitä voi itse laajentaa *esittelemällä* uusia metodeita omaan repository-interface'iin
- Uudet metodit voi esitellä nimeämiskäytännön mukaan, eli niiden täytyy olla muotoa find...By, read...By, query...By, count...By, and get...By - esimerkiksi List<Henkilo> findByNimi(String nimi)
- Lisäksi metodeille voi antaa useita tarkennuksia
  - <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.query-creation>

```
public interface HenkiloRepository extends CrudRepository<Henkilo, Long> {  
    List<Henkilo> findByNimi(String nimi);  
    List<Henkilo> findByNimiIgnoreCase(String nimi);  
    List<Henkilo> findByNimiAndIka(String nimi, int ika);  
    Henkilo findFirstOrderByIkaAsc();  
}
```

# Harjoitus

---

- Tee vastaava listaus kuin edellisessä harjoituksessa, mutta anna käyttäjän hakea nimellä maat - tähän siis RESTmaisesti kyselyparametri
- Lisää myös kyselyparametri, jonka avulla käyttäjä voi hakea maat, jonka asukasluku on suurempi tai yhtä suuri kuin annettu luku
- Repositoryn muut metodit:
- Käytä myös Repositoryn save metodia: Toteuta palvelumetodi, jolla voit muokata maan asukaslukua
  - Toki voit tehdä koko maalle PUT toteutuksen, mutta asukasluku on oikeastikin muuttuva, ja sen toteutus on helpompi kuin kaikkien tietojen, puhumattakaan tarkemmin sovituista tiedoista
  - Lisätehtävä: kun kyseessä on päivitys vain osalle tiedoista, niin tässä voisi käyttää PATCH metodia PUTin sijaan - tee se
- Lisätehtävä:
  - Toteuta myös uuden maan lisääminen ja maan poistaminen

---

**JPQL**

---

**Sovelto**

# JPQL - Java Persistence Query Language

---

- Hakukieli, jolla voidaan suorittaa hakuja tietokantaan
- Hakutulokset ovat entiteettejä
- Muistuttaa SQL-kieltä
- Oliosuuntautunut
- Kyselyt muutetaan JPA-implementaation sisällä SQL-kyselyiksi
  - Erittäin siirrettävä, koska SQL-murteitten erot on piilotettu implementaatioiden sisään
  - Esim. Hibernate tuntee yli 30 SQL-murretta



# JPQL käyttö

---

- Spring Boot ja Spring Data yhteydessä yleisin JPQL käyttötilanne on repositoryn kyselyitä tehtäessä
  - Kun nimikäytäntö ei enää riitä
  - Nimettyjen repository-metodien tarkennettu kuvaus käyttäen JPQL kyselyitä:  
<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.query-creation>
- Repositoryyn voi siis lisätä omia metodeita, ja antaa niille Query-annotaatiolla JPQL kyselyn, jota käytetään metodikutsun yhteydessä

```
@Query("SELECT u FROM User u WHERE u.name NOT LIKE '% %'")  
List<User> haeKayttajatYhdellaNimella();
```

- Kyselyn parametrit täytyy määritellä metodissa @Param annotaatiolla

```
@Query("SELECT c FROM Country c WHERE c.capital.name = :kaupunki")  
List<Country> haePaakaupunginNimella(@Param("kaupunki") String capital);
```

# Harjoitus

---

- Jatka edellisen harjoituksen parissa, ja rajapintaan toiminnallisuutta
- Lisää ainakin seuraavat toiminnot
  - Yhden maan hakeminen id-kentän (eli maakoodi) perusteella
  - Maiden hakeminen maan nimen perusteella, välitä hakuehto request parametrin mukana
- Jos aikaa jää, niin toteuta
  - Tietyn maan tietojen muokkaaminen (PUT)
  - Uuden maan lisääminen (POST)
  - Maan poistaminen (DELETE)

# Viittaukset

---

- JPA tukee viiteavainten käyttöä varsin helposti
- Viittaukset voidaan tehdä OneToOne, ManyToOne ja ManyToMany tavalla
- Kehittäjän on pidettävä omistava osapuoli ja käänteinen osapuoli konsistentteina ohjelmointivaiheessa eli JPA ei automaattisesti päivitä toista osapuolta (!)
  - Kannasta olioita lukiessa viittaukset päivittyvät oikein
- Viittauksia sisältävissä relaatioissa voidaan attribuutilla `cascade` määrittää, mitkä operaatiot kohdistuvat myös toiseen osapuoleen
  - Arvona on taulukko enum-tyypin `CascadeType` vakioita
  - Arvot:
    - `ALL`, joka vastaa määritystä `{ MERGE, PERSIST, REFRESH, REMOVE }`
    - `MERGE, PERSIST, REFRESH, REMOVE`
- "Laiskuuden" voi määrittää mikäli oletus ei ole hyvä - usein mieluummin `JOIN FETCH` tarvittaessa
  - `fetch = FetchType.LAZY / EAGER`

# Sivutus

---

- Sivutus (paging) on käytössä usealla Web-sivulla, Spring Datan `PagingAndSortingRepository` helpottaa sen käyttöönottoa
  - Tietokantakyselyissä sivutus tehdään usein kannan käyttämällä SQL-murteella
- `PagingAndSortingRepository` lisää `Page<T> findAll(Pageable)` metodin (myös sorttaamiseen oma metodinsa)
- Tällä metodilla voidaan hakea tietoa sivu kerrallaan, myös määrittää sivun koko
- `Pageable` sisältää attribuutit `page` ja `size`, näillä voidaan hakea esimerkiksi sivu numero 3 kun yhden sivun koko on 20
- Paluuarvona tulee `Page`-tyyppinen olio sisältää tiedot palautetusta datasta, sekä yleistä tietoa sivutuksesta
  - `List<T> getContent()`, `int getTotalPages()`, `long getTotalElements()`, `int getNumber()`, `int getNumberOfElements()`, jne.

# Pageable

---

- Yksinkertaisimmin sivutuksen saa käyttöön määrittelemällä request metodille Pageable parametrin
  - Oletuksena page on 0 ja size on 20, vaikka käyttäjä ei antaisi mitään arvoja
- Pageable muuttujan saa myös määriteltyä PageRequest luokan avulla
  - Esimerkiksi

```
Pageable pageable = PageRequest.of(1, 20);  
// tai  
pageable = PageRequest.of(0, 10, Sort.Direction.ASC, "population");  
// tai  
pageable = PageRequest.of(1, 20, Sort.by("name", "population", "district"));
```

# Harjoitus - sivutetut maat

---

- Edellinen harjoitus toimii hyvänä pohjana tälle harjoitukselle, eli tässä palautetaan Country taulun sisältämät maat, mutta sivu kerrallaan
- Vaihda repository(t) perimään PagingAndSortingRepository
- Toteuta kontrolleriin uusi request-metodi sivutuksen tulostamiseen, lisää sille ensimmäiseksi parametriksi Pageable tyyppinen parametri
  - Voit kutsussa antaa arvot request-parametreina esim: `http://localhost:8080/maasivut?page=3&size=40`
- Toteuta nyt maiden haku käyttäen `findAll(Pageable)` metodia
- Lisätehtävä: Toteuta REST-palvelun sivutusta käyttävä HTML+JS sivu, jolla voit vaihtaa näytettäviä maita
  - Vaihtoehtoisesti voit toteuttaa tämän Thymeleaf-svivuna, jolloin näytettävä sivu valitaan Thymeleafin kontrollerissa

---

**Tietokanta vai koodi ensin?**

---

# Koodi ensin

---

- Usein on tilanne, jolloin tietokannan rakenne on valmis (esim. world kanta). Tällöin koodi kirjoitetaan, tai generoidaan, valmiin rakenteen mukaisesti
- Mikäli kantaa ei ole, voidaan valita vaihtoehdoksi myös että kirjoitetaan ensin koodi, ja koodin perusteella generoidaan tietokannan rakenne ja taulut
- Yhteysmäärittelyksissä (application.properties) voidaan kertoa miten koodin ja tietokannan muutosten suhde halutaan tehdä
- Asetus spring.jpa.hibernate.ddl-auto arvon käsittely
  - create-drop - ensin tämä käytössä, tietokannan rakenne tyypillisesti muuttuu hieman kehityksen alkuvaiheessa
  - create - kantaan käydään lisäämässä uudet taulut Entity luokkien perusteella, vanhat tyhjentyvät
  - update - kun schema on suurin piirtein oikea, mutta saattaa vielä muuttua. Kannassa on kuitenkin jo dataa, jonka ei haluta poistuvan aina uuden käynnistytksen yhteydessä
  - validate - kun rakenne on valmis, mutta halutaan varmistaa ettei kukaan muukaan ole käynyt schemaa muuttamassa
  - none (oletus) - kun schema on valmis ja tietokannalle ei tapahdu mitään vaikka koodin rakenne muuttuu, kannassa on myös dataa jonka halutaan siellä pysyvän



# Tietokanta ensin

---

- Tyypillisesti vain silloin kun tietokanta on jo valmiiksi toteutettu (kuten World)
- Tällöin Entity-luokat kannattaa usein generoida työkalun avulla. Esimerkiksi IntelliJ osaa tämän mainiosti
  - Kunhan IntelliJn projektiasetuksiin on otettu mukaan JPA. Hiiren 2-näppäin projektille >> Add Framework Support.. ja valitse JPA. Mahdollisesti myös persistence.xml konfigurointi
  - <https://www.jetbrains.com/help/idea/enabling-jpa-support.html#existing>

# Harjoitus

---

- UUSI PROJEKTI - Spring boot initializer
- Yhteys uuteen tietokantaan, luo **kanta** Postgresql kautta
- application.properties:
  - yhteys, käyttäjä salasana kuten ennen
  - spring.jpa.hibernate.ddl-auto=create-drop
- Entity luokka: Oppilas
  - Myöhemmin: Koulu
- Kontrolleri
- Tärkeämpi kuin REST-palvelun toiminnallisuus: tietokannan rakenne ja sisältö
- Toiminnallisuus: luo uusi oppilas, muuta tietoja katso lista, katso yhden detaljit, ...

---

**Testausta: Mockit**

---

# Testaus Mockien avulla

---

- H2 sijaan voidaan yksikkötesteissä määritellä tietokannan antama tieto Mockien eli valeolioiden avulla
- Springin testiriippuvuudet ottavat mukaan myös Mockiton ja sen Springiin toteutetut apuluokat
- Mock voidaan konfiguroida siten, että `@MockBean` annotaatiolla varustetulla valeoliolla korvataan varsinainen Dao luokan instanssi. Tämä valeolio sitten toimii varsinaisen Dao-olion sijalla kun testejä ajetaan, ja vastaa halutuilla vastauksilla Dao-oliolle tehtyihin metodikutsuihin
  - Yksi syy siihen, miksi tietokantakäsittely kannattaa eriyttää omaan luokkaansa, jota REST kontrolleri käyttää, eikä suoraan käyttää esimerkiksi JdbcTemplatea kontrollerissa
- Mockien avulla ei siis käytetä mitään tietokantaa, vaan koodissa määritellään tietokantaa käyttävän luokan vastaukset
  - Voi olla työlästä monimutkaisten kantakäsittelyiden kanssa

# Esimerkki: Mock alustus

---

- Esimerkki, joka määrittelee yksinkertaisen Dao-olion Mockin ja mitä se vastaa joihinkin metodikutsuihin

```
@AutoConfigureMockMvc
public class RestControllerMockTest {

    @MockBean
    private HenkiloDao dao;

    private Henkilo h1 = new Henkilo("A", "B", 1);
    private Henkilo h2 = new Henkilo("C", "D", 11);
    @Before
    public void setupMock() {
        given(this.dao.haeIdlla(0))
            .willReturn(Optional.empty());
        given(this.dao.haeIdlla(1))
            .willReturn(Optional.of(h1));
        given(this.dao.getHenkilot())
            .willReturn(Arrays.asList(h1, h2));
    }
}
```

# Esimerkki: Mock-testit

---

```
@Autowired
private MockMvc mvc;

@Test
public void testaaEttäAforisminHakuOlemattomallaIdlläPalauttaaNotFound() throws Exception {
    this.mvc.perform(get("/api/henkilot/0").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isNotFound());
}

@Test
public void testaaEttäAforisminHakuOnnistuu() throws Exception {
    this.mvc.perform(get("/api/henkilot/1").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("etunimi", is(h1.getEtunimi())));
}

@Test
public void testaaKaikkiTuleeHaettua() throws Exception {
    this.mvc.perform(get("/api/henkilot").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$", hasSize(2)))
        .andExpect(jsonPath("$[0].sukunimi", is(h1.getSukunimi())));
}
```