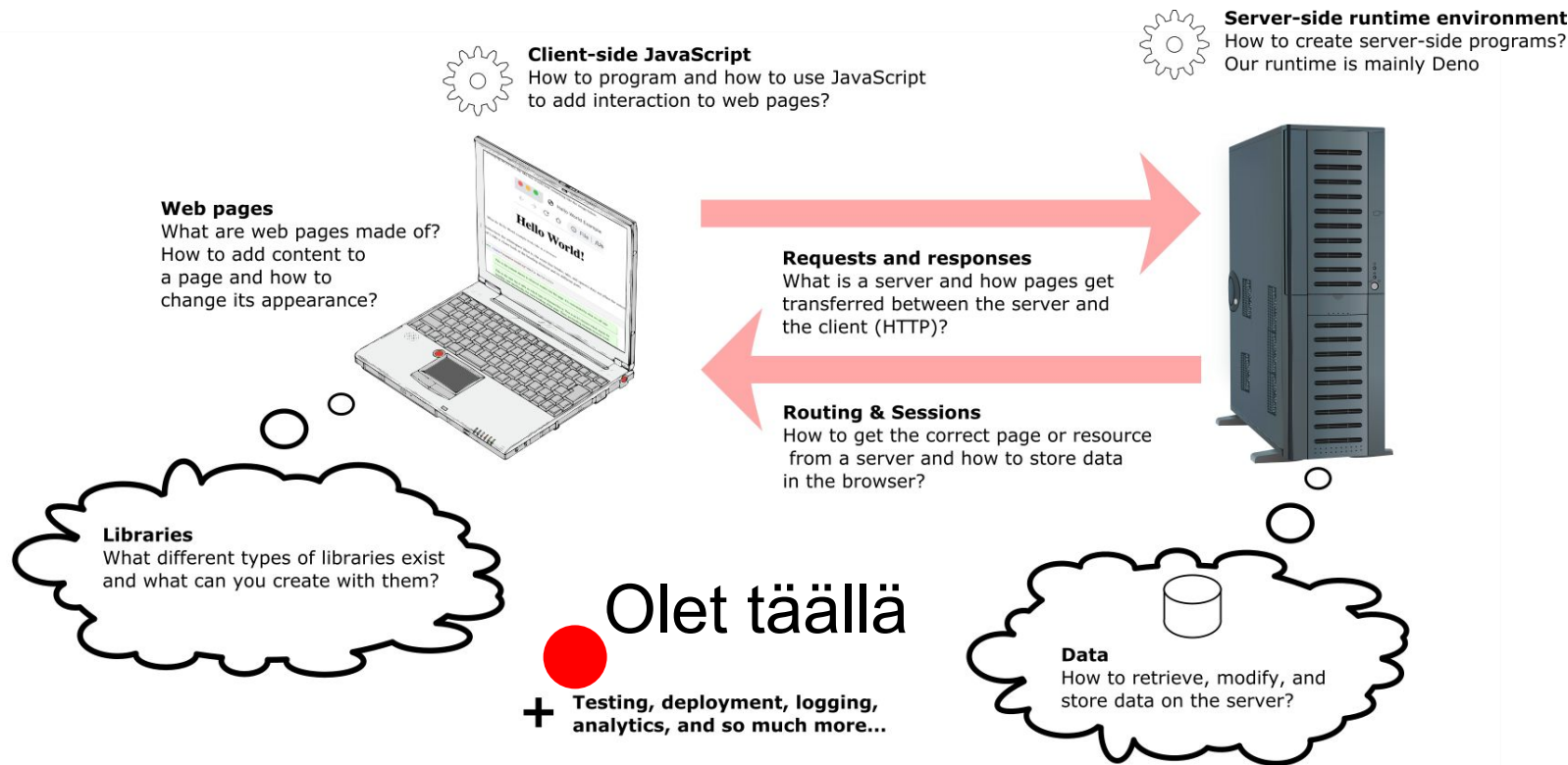


Päivä 6 - Sovellusarkkitehtuuri, sovellusten testaaminen

2021-12-02

AaltoPRO - Websovelluskehitys

Web-sovellukset korkealla tasolla

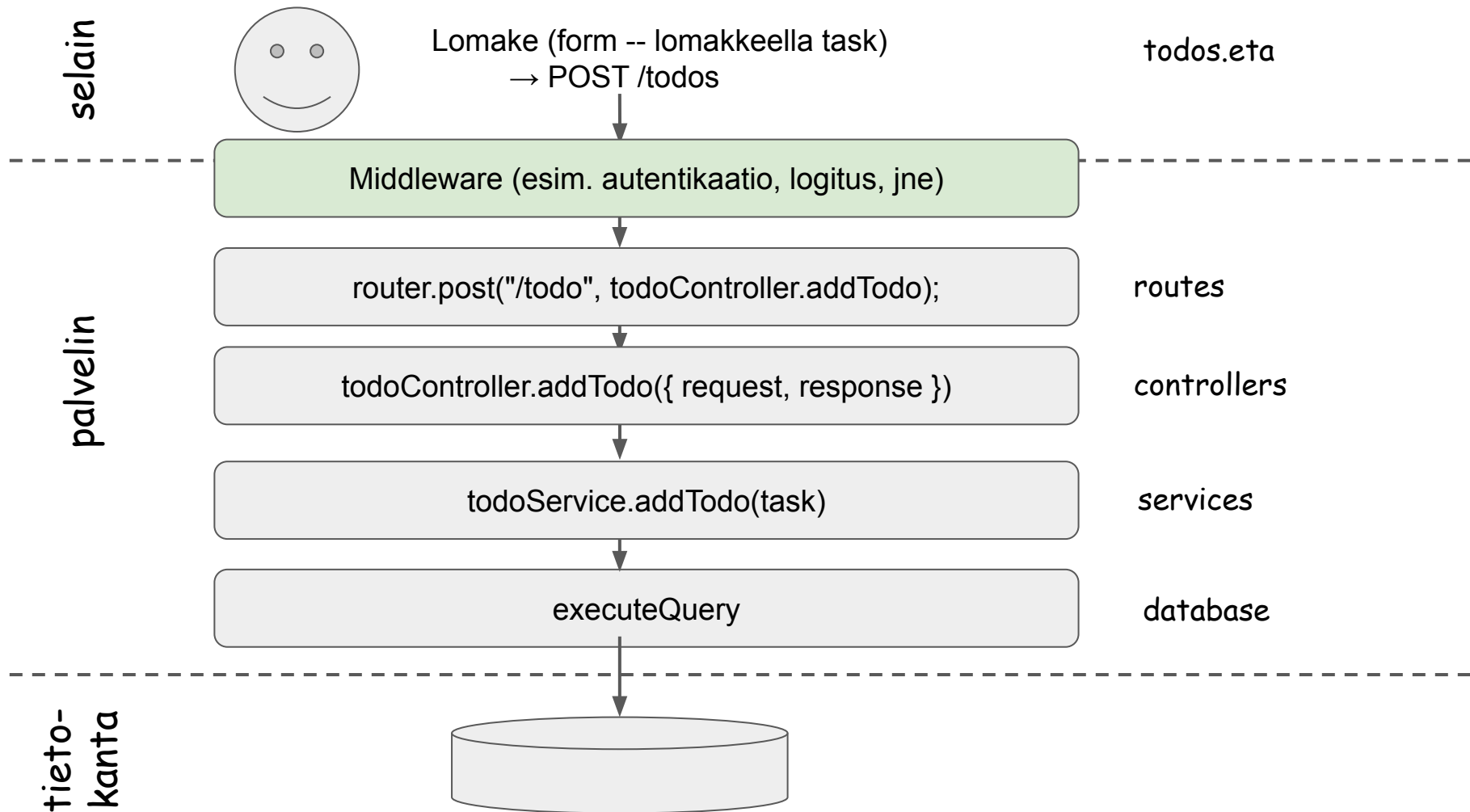


Päivä 6

- 9-12 Aamupäivä
 - Kerrosarkkitehtuurin kertaus, muutama sana sovellusarkkitehtuurista
 - Kahvitauko
 - Johdanto testaukseen, automaattinen testaaminen
 - Yhteenveto
- 12:00 - 13:00 Lounas
- 13:00 - 16:00 Iltapäivä
 - Test-driven development
 - Kahvitauko
 - Integraatiotestaus
 - Yhteenveto

Kertaus: kerrosarkkitehtuuri

- Tapa jakaa sovellus loogisiin kokonaisuuksiin
 - Auttaa sovelluksen rakenteen ymmärtämistä → tuttu, toistuva rakenne
 - Auttaa sovelluksen muokkaamista → vaikka osa muuttuu, muut palat pysyvät
 - Auttaa sovelluksen testaamista → kokonaisuuksia voi testata myös erikseen
- Tyypillisesti ainakin:
 - Jonkinlainen tapa esittää näkymä (esim luotu templateista, HTML+Javascript, ...)
 - “Controllers” → Käsittelevät pyynnöt ja palauttavat vastauksen
 - “Services” → Tarjoavat pyyntöihin toiminnallisuuksia kuten tietokannan käsittelyä
 - Jonkinlainen tietokanta-abstraktio (ORMit, “executeQuery”)



Kertaus: kerrosarkkitehtuuri

- project-05-foods läpikäynti

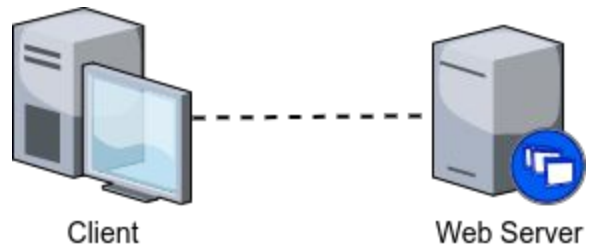
Multitier / N-tier arkkitehtuuri

- Kerrosarkkitehtuurissa tarkastellaan sovelluksen sisäistä loogista rakennetta
- N-tier arkkitehtuuri jakaa sovelluksen useampaan erotettavissa olevaan osaan: esim. käyttäjän koneella oleva selain, palvelimella oleva web-sovellus, palvelimella oleva tietokanta

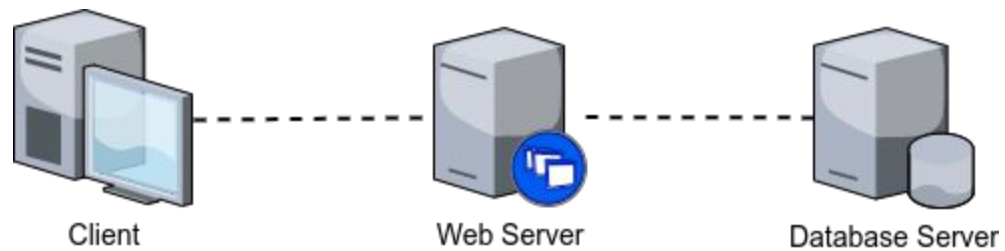
1-Tier



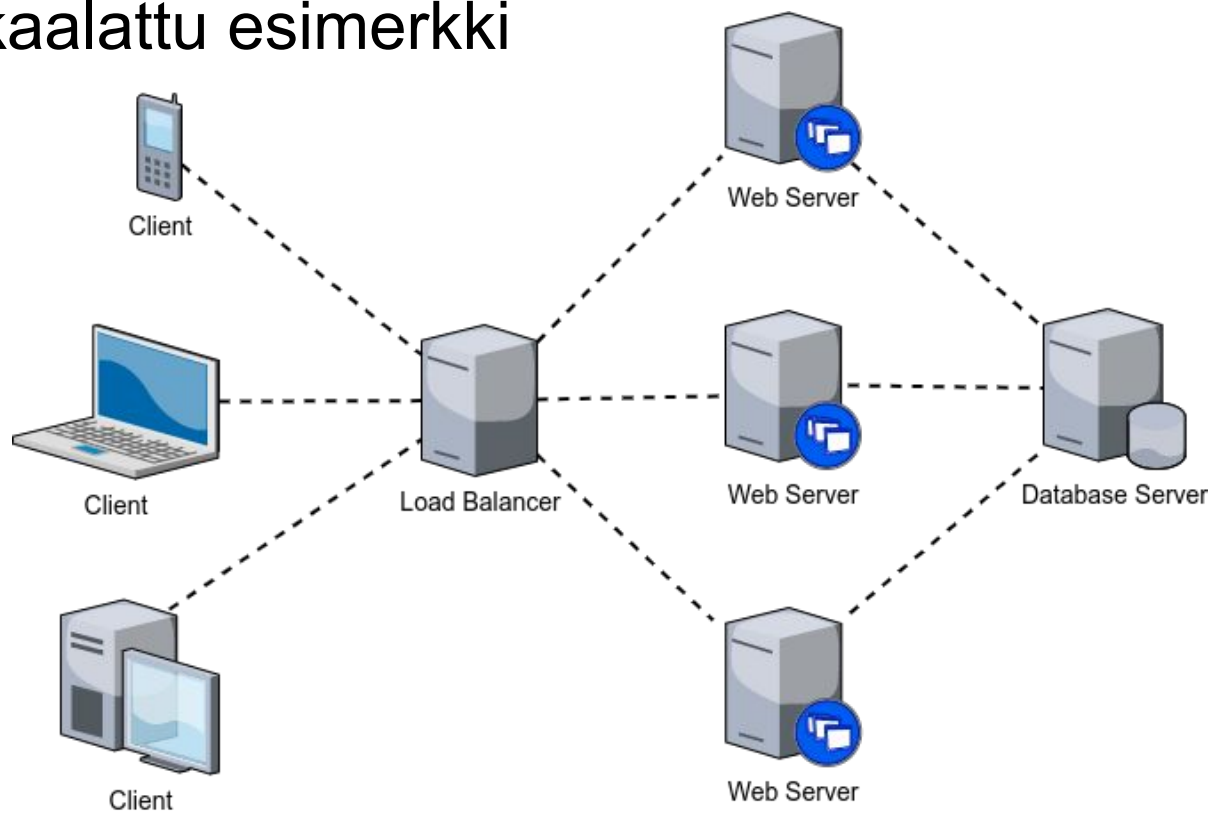
2-Tier



3-Tier



4-Tier, skaalattu esimerkki



Multitier / N-tier arkkitehtuuri

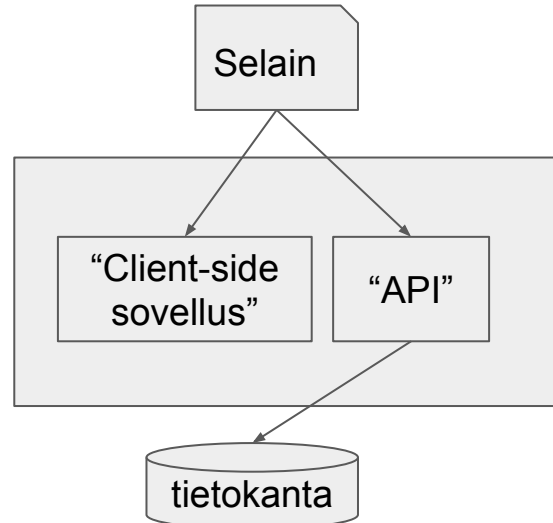
- Hyötyjä mm. skaalautuvuudessa (esimerkiksi web-palvelimia voi olla useita, tietokantapalvelimia voi olla useita, jne..)
- Huom! Netissä (ja muualla) olevissa resursseissa käsitteet N-tier arkkitehtuuri ja kerrosarkkitehtuuri usein sekoittuvat

Monoliittinen arkkitehtuuri vs. mikropalveluarkkitehtuuri

- Monoliittisessä arkkitehtuurissa sovellus rakennettu kokonaisuutena. Sovelluksella usein looginen sisäinen rakenne -- esim. edellä kerrattu kerrosarkkitehtuuri.
- Mikropalveluarkkitehtuurissa sovellus rakennetaan useammasta pienemmästä palvelusta, joista jokainen itsenäinen kokonaisuutensa
 - Omia kokonaisuuksiaan esimerkiksi sovelluksen käyttöliittymä, käyttäjänhallinta, varastonhallinta, ...
- Sovelluskehityksessä yhä tyypillisesti suositaan monoliittisella arkkitehtuurilla aloittamista, ja sovelluksen pilkkomista osiin kun tarve esiintyy

Demo: project-05-world-explorer-frontend

- Sovelluksessa kaksi palvelinta / palvelua, toisen vastuulla selainpuolen toiminnallisuus, toinen tarjoaa ohjelmointirajapinnan
- Näiden lisäksi käytössä tietokanta
- Palveluilla yhä oma looginen rakenteensa



Lyhyt hands-on: project-05-world-explorer-frontend

- Muokkaa sovellusta siten, että:
 - sovelluksessa on enemmän paikkoja, joihin käyttäjä voi mennä
 - sovelluksessa on enemmän esineitä
- Huom! Muokkaa `world-kansion Dockerfile-tiedoston` ensimmäinen rivi muotoon:

```
FROM denoland/deno:debian-1.16.3
```

Kahvitauko?

Johdanto ohjelmistojen testaamiseen

- *Testaaminen* → selvitetään toimiiko sovellus toivotulla tavalla
- Mikäli sovellus ei toimi toivotulla tavalla, virheet korjataan
 - Virheiden korjaaminen kehitysvaiheessa edullisempaa kuin ohjelmiston ollessa tuotantokäytössä

Johdanto ohjelmistojen testaamiseen

- Virheiden mittakaava (ja kustannus) liittyy usein kontekstiin
 - Ariane 5 - avaruusraketti tuhoutui
 - Boeing 737 - lentokone tippui
 - Airbus A400M - kolme moottoria neljästä sammui, lentokone tippui
 - Zoom - kuka tahansa pystyi katsomaan palaverin keskusteluja sekä palaverin tietoja
 - Yhdysvaltojen vankijärjestelmä - noin 3000 vankia pääsi etuajassa vapaaksi
 - Tesco - Applen iPadeja myytiin ~50 punnalla
 - Reebok - lenkkareita sai vain tilausmaksulla
 - Nissan - matkustajan paikan ilmatyyny ei lauennut
 - jne..

Johdanto ohjelmistojen testaamiseen

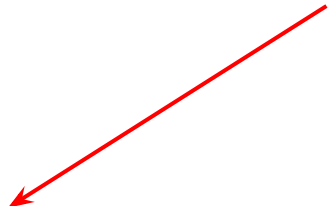
- Testaustyytit - Black-box testing vs. white-box testing
 - Black-box testing: testaaja tietää mitä sovelluksen pitäisi tehdä, mutta ei tiedä miten sovellus se sen tekee (lähdekoodi ja sisäinen logiikka piilossa)
 - White-box testing: testaaja tietää, mitä sovelluksen pitäisi tehdä, jonka lisäksi testaaja myös tietää miten sovellus tekee sen

Johdanto ohjelmistojen testaamiseen

- Hands-on: project-06-temperatures
 - Lataa projekti ja käynnistä se: `docker-compose up --build`
 - Avaa sovellus selaimessa -- <http://localhost:3000>
 - Black-box -testausta: älä katso lähdekoodia vaan etsi sovelluksesta virheitä sovellusta kokeilemalla

Automaattinen testaus

Tämä kansiossa, jossa on
docker-compose.yml



- Hands-on: project-06-temperatures
 - Mene komentorivillä kansioon “temperatures”, suorita komento
`deno test`
Tai: `docker-compose run --rm temperatures deno test`
- Tiedostossa `tests/arithmeticService.test.js` on kolme automaattisesti suoritettavaa testiä. Aiempia testejä matkien, lisää seuraavat testit:
 - Testi, joka laskee keskiarvon yhdelle mittaukselle. Testin tulee käyttää mittauksena arvoa 5 ja odottaa että keskiarvo on 5. Aja tämän jälkeen testit uudestaan yllä annetulla komennolla.
 - Testi, joka laskee keskiarvon kolmelle mittaukselle. Testin tulee käyttää arvoja 1, 0 ja -1. Testin tulee odottaa, että mittauksen keskiarvo on 0. Aja tämän jälkeen testit uudestaan yllä annetulla komennolla.
 - Testi, joka laskee keskiarvon kolmelle mittaukselle. Testin tulee käyttää arvoja 1, 1 ja 1. Testin tulee odottaa, että mittauksen keskiarvo on 1. Aja tämän jälkeen testit uudestaan yllä annetulla komennolla.
 - Viimeisen testin pitäisi näyttää virhe. Korjaa virhe sovelluksesta ja aja testit uudestaan.

Automaattinen testaus

- Automaattisessa testauksessa kirjoitetaan testejä, jotka ohjelma suorittaa
- Mm. yksikkötestaus, integraatiotestaus, järjestelmätestaus
 - Yksikkötestaus: yksittäinen osa tai funktio toimii toivotulla tasolla
 - Integraatiotestaus: osat toimivat yhdessä toivotulla tasolla
 - Järjestelmätestaus: sovellus toimii kokonaisuutena toivotulla tavalla



Tähän otettiin äsken
ensiaskeleet

Automaattisesti suoritettavan testin rakenne

```
Deno.test("Counting averages, no data", () => {  
  const data = [];  
  const average = countAverage(data);  
  assertEquals(average, "Not enough data.");  
});
```


Testin määrittely



```
Deno.test("Counting averages, no data", () => {  
  const data = [];  
  const average = countAverage(data);  
  assertEquals(average, "Not enough data.");  
});
```

Testin määrittely

Testille
annettava nimi

```
Deno.test("Counting averages, no data", () => {  
  const data = [];  
  const average = countAverage(data);  
  assertEquals(average, "Not enough data.");  
}));
```

Testin määrittely

Testille
annettava nimi

```
Deno.test("Counting averages, no data", () => {  
  const data = [];  
  const average = countAverage(data);  
  assertEquals(average, "Not enough data.");  
});
```

Testin yhteydessä
suoritettava funktio

Testin määrittely

Testille
annettava nimi

```
Deno.test("Counting averages, no data", () => {  
  const data = [];  
  const average = countAverage(data);  
  assertEquals(average, "Not enough data.");  
});
```

Tarkastelu tai
vaatimus -- "näin
tulee olla"

Testin yhteydessä
suoritettava funktio

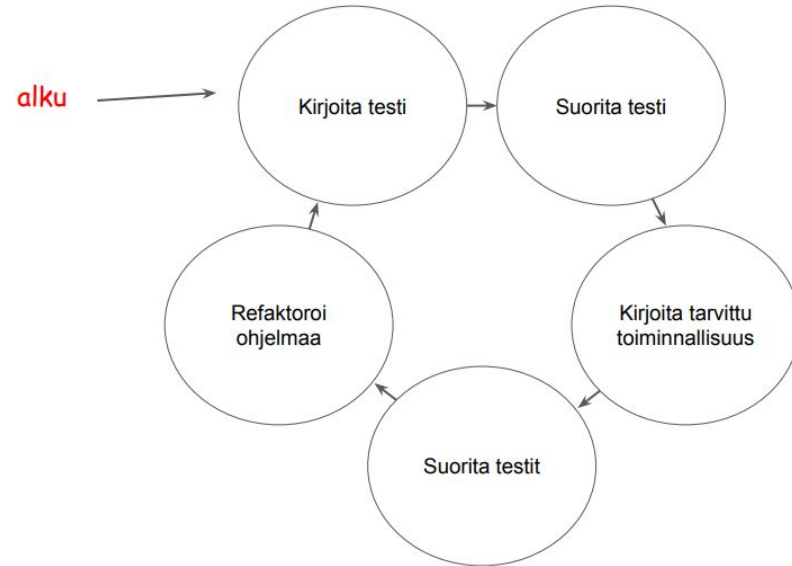
Tarkastelut (varmistukset) - *assert**

- Automaattisesti suoritettavan testin yhteydessä tehdään yksi tai useampi tarkastelu, joilla varmistetaan että syötteillä saadaan toivottu arvo tai tuloste
- Useita erilaisia tarkastelumahdollisuuksia, mm.
 - `assertEquals(arvo, odotettuArvo)` - arvot ovat täsmälleen samat
 - `assertExists(arvo)` - arvo on olemassa
 - `assertStringIncludes(merkkijono, odotettu)` - merkkijonossa esiintyy odotettu
 - `assert(lauseke)` - lauseke totta (esim. `luku > 5`)
 - ...
- Kts. lisää <https://deno.land/manual/testing/assertions>

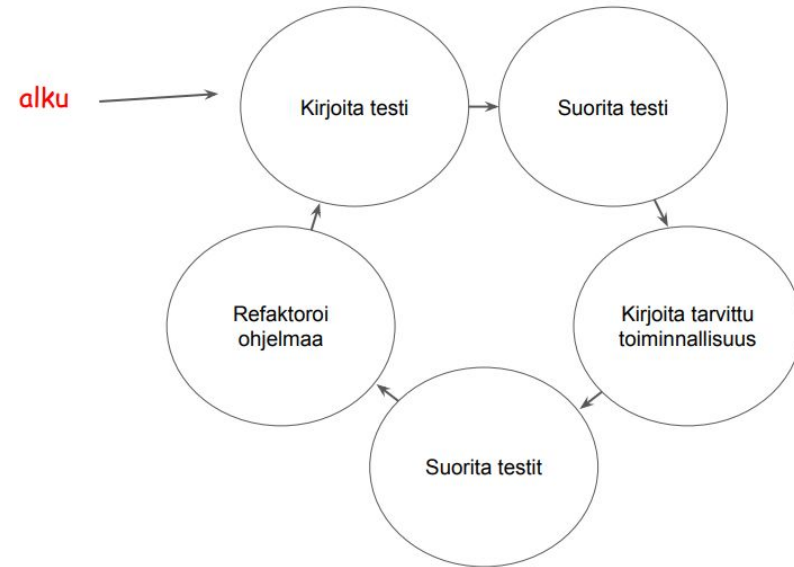
Lounas

Test-driven Development (TDD)

- Ohjelma luodaan pienin askelin siten, että jokaiselle uudelle toiminnallisuudelle tai toiminnan osalle luodaan ensin testi
- Kun testi on luotu, testit ajetaan → uusi testi ei mene läpi
- Tämän jälkeen luodaan toiminnallisuus, joka täyttää testin
- Kun toiminnallisuus on luotu, testit ajetaan uudestaan

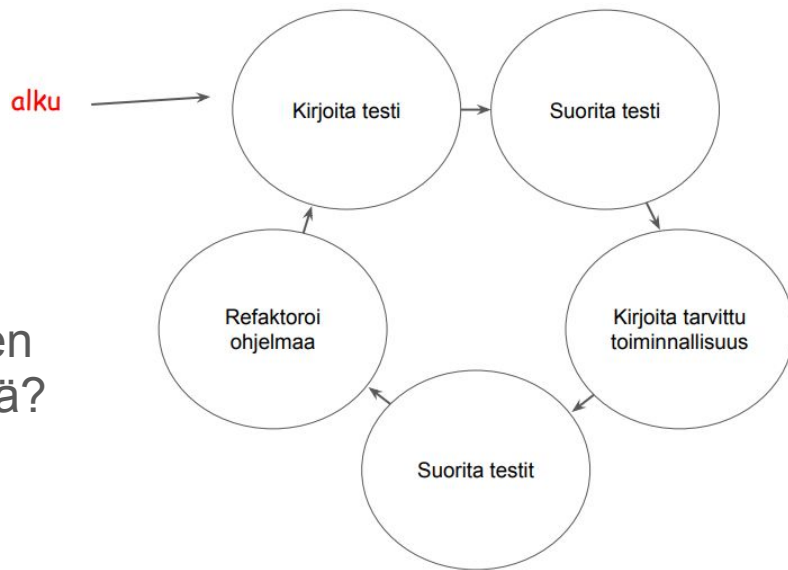


Test-driven Development (TDD) - Demo



Hands-on: project-06-temperatures

- Pienimmän arvon selvittäminen
 - Lisää tiedostoon `arithmeticService` tyhjä funktio `findMinimum`
 - Toteuta testi, joilla selvitetään annetusta datasta pienin lämpötila
 - Toteuta toiminnallisuus
- Pienimmän arvon selvittäminen
 - Lisää tiedostoon `arithmeticService` tyhjä funktio `findMaximum`
 - Toteuta testi, joilla selvitetään annetusta datasta suurin lämpötila
 - Toteuta toiminnallisuus
- Kannattaa tehdä useampi testi -- esim. miten ohjelman tulee toimia kun annettu data tyhjä?



Kahvitauko?

Integraatiotestaus

- Tarkastellaan toimivatko sovelluksen osat yhdessä
 - Esimerkiksi toimiiko tietokantaa käyttävä `temperatureService` oikein?
 - Sovelluksen osien tulee olla käynnissä testien ajamiseksi
- Yksittäisen komennon suorittaminen Docker-ympäristössä suoraviivaista:
`docker-compose run --rm temperatures deno test`

Integraatiotestaus

- Tarkastellaan toimivatko sovelluksen osat yhdessä
 - Esimerkiksi toimiiko tietokantaa käyttävä `temperatureService` oikein?
 - Sovelluksen osien tulee olla käynnissä testien ajamiseksi
- Yksittäisen komennon suorittaminen Docker-ympäristössä suoraviivaista:
`docker-compose run --rm temperatures deno test`


Käynnistä uusi
"kontti"

Integraatiotestaus

- Tarkastellaan toimivatko sovelluksen osat yhdessä
 - Esimerkiksi toimiiko tietokantaa käyttävä `temperatureService` oikein?
 - Sovelluksen osien tulee olla käynnissä testien ajamiseksi
- Yksittäisen komennon suorittaminen Docker-ympäristössä suoraviivaista:

`docker-compose run` `--rm` `temperatures deno test`

Käynnistä uusi
"kontti"

...joka poistetaan
lopuksi

Integraatiotestaus

- Tarkastellaan toimivatko sovelluksen osat yhdessä
 - Esimerkiksi toimiiko tietokantaa käyttävä `temperatureService` oikein?
 - Sovelluksen osien tulee olla käynnissä testien ajamiseksi
- Yksittäisen komennon suorittaminen Docker-ympäristössä suoraviivaista:

`docker-compose run` `--rm` `temperatures` `deno test`

Käynnistä uusi
"kontti"

...joka poistetaan
lopuksi

Konttiin palvelu
"temperatures" (myös
riippuvuudet käynnistetään)

Integraatiotestaus

- Tarkastellaan toimivatko sovelluksen osat yhdessä
 - Esimerkiksi toimiiko tietokantaa käyttävä `temperatureService` oikein?
 - Sovelluksen osien tulee olla käynnissä testien ajamiseksi
- Yksittäisen komennon suorittaminen Docker-ympäristössä suoraviivaista:

`docker-compose run --rm temperatures deno test`

Käynnistä uusi
"kontti"

...joka poistetaan
lopuksi

Konttiin palvelu
"temperatures" (myös
riippuvuudet käynnistetään)

Aja palvelussa
tämä komento

Integraatiotestaus

- Tarkastellaan toimivatko sovelluksen osat yhdessä
 - Esimerkiksi toimiiko tietokantaa käyttävä `temperatureService` oikein?
 - Sovelluksen osien tulee olla käynnissä testien ajamiseksi
- Yksittäisen komennon suorittaminen Docker-ympäristössä suoraviivaista:

`docker-compose run --rm temperatures deno test`

Käynnistä uusi
"kontti"

...joka poistetaan
lopuksi

Konttiin palvelu
"temperatures" (myös
riippuvuudet käynnistetään)

Aja palvelussa
tämä komento

Integraatiotestaus

- Luodaan temperatureService:n lisäämis- ja listaustoiminnallisuutta testaava testi
 - Luo kansioon tests tiedosto temperatureService.test.js
 - Lisää tiedostoon seuraava testi (myös importit!):

```
Deno.test("Adding a measurement increases database rows" , async () => {  
  const preData = await listTemperatures();  
  await addMeasurement(0);  
  const postData = await listTemperatures();  
  assertEquals(preData.length, postData.length - 1);  
});
```

Integraatiotestaus


- Testien ajaminen:

```
docker-compose run --rm temperatures deno test --allow-all
```

```
Deno.test("Adding a measurement increases database rows" , async () => {  
  const preData = await listTemperatures();  
  await addMeasurement(0);  
  const postData = await listTemperatures();  
  assertEquals(preData.length, postData.length - 1);  
});
```

Integraatiotestaus

Sallitaan pääsy mm.
ympäristömuuttujiin




- Testien ajaminen:

```
docker-compose run --rm temperatures deno test --allow-all
```

```
Deno.test("Adding a measurement increases database rows" , async () => {  
  const preData = await listTemperatures();  
  await addMeasurement(0);  
  const postData = await listTemperatures();  
  assertEquals(preData.length, postData.length - 1);  
});
```

Integraatiotestaus

Sallitaan pääsy mm.
ympäristömuuttujiin



- Testien ajaminen:

Testien ajamisesta tulee
virhe! Mitä ihmettä!

```
docker-compose run --rm temperatures deno test --allow-all
```

```
Deno.test("Adding a measurement increases database rows" , async () => {  
  const preData = await listTemperatures();  
  await addMeasurement(0);  
  const postData = await listTemperatures();  
  assertEquals(preData.length, postData.length - 1);  
});
```

Integraatiotestaus

- Virhe ilmoittaa siitä, että resursseja on varattuna yhä testin jälkeen
 - Taustalla oletus siitä, että testien pitäisi olla itsenäisiä ja käytössä olevien resurssien tulisi olla vapautettuja testien ajamisen jälkeen
 - Tietokanta on käytössä testien jälkeenkin - mutta! resurssien tarkastaminen voidaan poistaa

Eksplisiittinen muoto testien kirjoittamiseen: nimi, funktio, "lisätiedot"

Integraatiotestaus

```
Deno.test({  
  name: "Adding a measurement increases database rows",  
  fn: async () => {  
    const preData = await listTemperatures();  
    await addMeasurement(0);  
    const postData = await listTemperatures();  
    assertEquals(preData.length, postData.length - 1);  
  },  
  sanitizeOps: false,  
  sanitizeResources: false,  
});
```

Nyt testit toimivat!

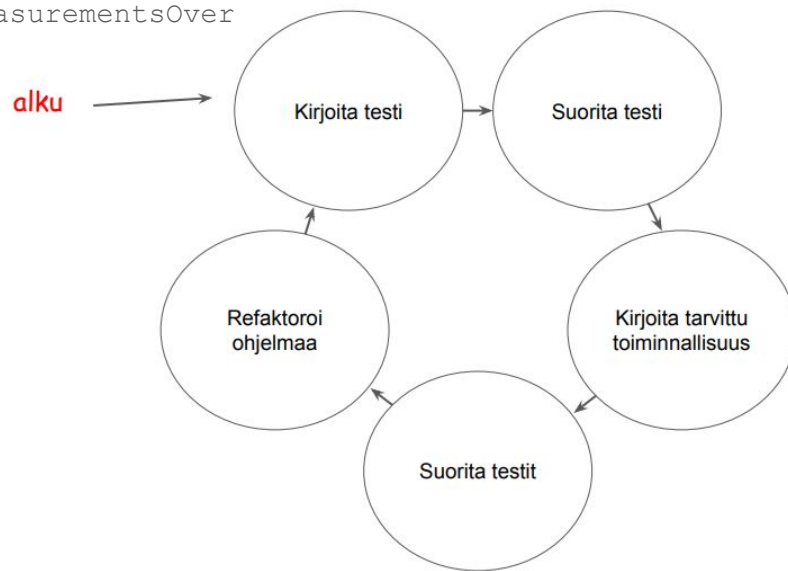
Eksplisiittinen muoto testien kirjoittamiseen: nimi, funktio, "lisätiedot"

Integraatiotestaus

```
Deno.test({  
  name: "Adding a measurement increases database rows",  
  fn: async () => {  
    const preData = await listTemperatures();  
    await addMeasurement(0);  
    const postData = await listTemperatures();  
    assertEquals(preData.length, postData.length - 1);  
  },  
  sanitizeOps: false,  
  sanitizeResources: false,  
});
```

Hands-on: project-06-temperatures

- Kaikkien mittausten poistaminen
 - Lisää tiedostoon `temperatureService` tyhjä funktio `clearMeasurements`
 - Toteuta testi, joilla selvitetään onko taulu tyhjä
 - Toteuta toiminnallisuus
- Yli tietyn arvon olevien mittausten poistaminen
 - Lisää tiedostoon `temperatureService` tyhjä funktio `clearMeasurementsOver` (funktio saa parametrina luvun)
 - Toteuta testi, jolla selvitetään poistetaanko yli tietyn luvun menevät arvot
 - Toteuta toiminnallisuus
- Ali tietyn arvon olevien mittausten poistaminen
 - Lisää tiedostoon `temperatureService` tyhjä funktio `clearMeasurementsUnder` (funktio saa parametrina luvun)
 - Toteuta testi, jolla selvitetään poistetaanko alle tietyn luvun menevät arvot
 - Toteuta toiminnallisuus



Yhteenveto iltapäivästä

+ *huominen lyhyesti*