2 – Unit test

Till denna övning ska vi bekanta oss med att skriva tester till redan befintlig fungerande kod – detta för att lättare bekanta oss med ett nytt testningsramverk!

Vi kommer att använda **Jest** som är ett testningsramverk skapat av Facebook som tagit inspiration från andra testningsramverk som Jasmine, Karma och Mocha. En stor skillnad mellan dessa ramverk är att Jest kommer färdiginstallerad med flera funktionaliteter som man tidigare fick genom att kombinera flera testningsramverk! Samt så är den lätt att integrera med front-end ramverket React och headless-browser verktyg som Puppeter.

1 – Sätta upp Jest

1. Öppna upp Visual Studio Code i en mapp som du vill arbeta i, samt se till att mappen innehåller programmet från felsökningsövningen – vi kommer att arbeta med den under större delen av den här dagen.
2. Skapa en package.json fil genom att ange kommandot **`npm init -y`** där flaggan -y ger dig automatiskt standardinställningarna.
3. Installera jest med kommandot **`npm i -D jest`** (en förkortning av **`npm install --save-dev jest`**) vilket kommer att installera jest samt sätta modulen jest i `devDependencies` i din package.json fil. Anledningen till varför vi inte installerar jest med kommandot **`npm i -S jest`** (förkortning av **`npm install --save jest`**) är för att då hade modulen hamnat i `dependencies` i package.json. Skillnaden mellan `dependencies` och `devDependencies` är att den första ska innehålla moduler som behövs i run-time i produktion, medan den andra ska innehålla de moduler som behövs i run-time när man utvecklar! Testramverk som jest är ett exempel på en modul som vi enbart behöver när vi utvecklar.

   Ta en kopp kaffe/te/etc under tiden den hämtar alla filer (glöm ej att låsa datorn!).

4. Editera din package.json fil så att under ”scripts” vill vi att värdet för ”test” sätts till ”jest”

   ```
   "scripts": {
     "test": "jest"
   },
   ```

   När vi har gjort det kan vi se om kommandot fungerar! Kör **`npm test`** (vilket är samma sak som **`npm run test`**) så bör du se att den försöker köra jest men sedan ger dig en massa felmeddelande som börjar med något i stil med ”No tests found, exiting with code 1”. Vilket är logiskt eftersom vi ännu inte skapat några testfiler!
5. I jest är testfiler vanliga javascript-filer med ändelsen js. De 2 vanligaste sätten att organisera sina testfiler med jest är följande:
   1. Skapa en mapp i roten som heter **__tests__** (två markstreck på varsin sida) och lägg alla dina testfiler i den, eller
   2. Döp testfilerna med ändelsen <valfrittNamn>**.test.js**

   Vi kommer att använda oss utav det 2:a alternativet som har den stora fördelen att vi enkelt kan se vilken testfil som hör ihop med vilken källkodsfil. Dvs **<dinKällkod>.js** och testfilen

<dinKällkod>.test.js. Skapa en sådan testfil som vi ska använda för att testa koden inuti felsökningsprogrammet.

2 – Programmera med Jest, grunderna

1. Låt oss skriva vårt första test. Syntaxen är som följer:
   test("Kort text som beskriver testet", callback);
   där ***callback*** är en funktion som utför testet. Det som ska testat påminner om assert-logiken från tidigare föreläsning men bygger på funktionskedjning. Testen börjar oftast med funktionen ***expect(…)*** som tar emot ett uttryck som ska evalueras (det vi vill testa). Den står väldigt sällan ensam utan följs av en sk matcher-metod, tex ***.toBe(…)*** som innehåller det vi tror det ska bli (vår hypotes, gör en === jämförelse).
   Skriv in följande testkod som undersöker om 2+2 är 4:
   ```
   test("Adds 2 + 2 to equal 4", () => {
       expect(2 + 2).toBe(4);
   });
   ```
   Du kan läsa mer om **toBe här** och om **expect här**. Läs också på **här för att få upp API-metodern för allt som kan kedjas på expect**.
2. Testkör genom att skriva **`npm test`** och se att testet går igenom. Du bör få upp lite statistik som visar bla Test Suites (hur många testfiler som kördes), Tests (hur många tests som kördes totalt), Snapshots (hur många avbildningar av ett state vi gjorde) och Time (hur lång tid det tog att exekvera alla test).
3. Skriv ett nytt test som testar negering, dvs motsatt. Skriv ett test som ska undersöka att 2 + 3 inte är 6. I jest kan man negera med funktionen not(). Läs på om funktionen **not här** och skriv klart testet.
4. Skriv ett nytt test som undersöker om 0.1 + 0.2 är 0.3. Gick testet igenom? Om du skulle få fel, titta på utskrifterna från jest som beskriver vad Expected och Recieved har för värden.

   Relevant serie-comic: **https://www.smbc-comics.com/comics/20130605.png**

   När vi arbetar med flyttal i jest bör vi istället för toBe(…) använda oss av **toBeCloseTo(…) som per-default avrundar till 2 decimaler**. Byt ut toBe(…) mot toBeCloseTo(…) och se om testet går igenom nu!

   För den som vill läsa på om hur nummer representeras i javascript kan man läsa denna matiga artikel: **https://modernweb.com/what-every-javascript-developer-should-know-about-floating-points/**
5. Skapa ett nytt test som testar följande:
   ```
   const obj1 = {name:"test", data:[1,2,3]};
   const obj2 = obj1;
   expect(obj1).toBe(obj2);
   ```
6. Gör ett nytt test som testar följande:
   ```
   const obj1 = {name:"test", data:[1,2,3]};
   const obj2 = {name:"test", data:[1,2,3]};
   expect(obj1).toBe(obj2);
   ```
   Vad är skillnaden på dessa två test?
   När du tror dig veta svaret, eller gett upp, gå vidare till nästa deluppgift.

7. Modifiera testet så att det fungerar – byt ut toBe(…) mot **toEqual(…) som ni kan läsa på om här**. När ni vill jämföra referenstyper (objekt, arrays, funktioner) så använder ni toEqual. För den som vill läsa på mer om primitiva,- vs referens typer **rekommenderas den här bloggen**.

3 – Testa våra egna funktioner med Jest, förberedelse

1. I node kan man få javascript-filer att använda varandras funktioner om filerna väljer att exportera dem. Så för att vår testfil ska kunna känna av våra funktioner från vår felsökningsövning måste vi exportera dem. Låt oss göra det först!

   Först behöver vi modifiera vår js-fil från felsökningsuppgiften så att den inte gör några globala funktionsanrop. I vårt fall är det **`main()`** längst ner som vi tar bort. Anledning till att vi vill ta bort alla globala anrop är för att de kommer att köras när vi testar funktionerna med jest.

   I js-filen skapar vi högst upp eller längst ner (smaksak) ett objekt som vi döper till **functions** och under den lägger vi till att det är det objektet vi vill exportera till module. Inuti **functions** lägger vi till de funktioner som vi vill exportera som properties till objektet. Vi börjar med dessa 4 funktioner:

   ```
   const functions = {
       main: main,
       addProduct: addProduct,
       totalProductCost: totalProductCost,
       emptyList: emptyList
   }
   ```

   Vi kan även skriva detta på en kortare form på detta vis:

   ```
   const functions = {
       main,
       addProduct,
       totalProductCost,
       emptyList
   }
   ```

   Längst ner i filen (oftast bäst att ha detta längst ner när allt har läst in) gör vi själva exporteringen genom att skriva:

   ```
   if (typeof module === "object"){
       module.exports = functions;
   }
   ```

   If-satsen har vi med enbart så att programmet inte ska ge varningar om vi vill köra programmet i en webläsare där ***module*** inte finns!
2. För att fortfarande kunna använda vårt program så kan vi skapa en ny fil som blir ingångsfilen för vårt program. Skapa en ny fil som heter ***main.js*** som ser ut så här:

   ```
   const functions = require("<sökvägen-på-filen-med-alla-funktioner>");
   functions.main();
   ```

   Där objektet ***functions*** kan heta vad som helst. Om du kör node main.js nu bör du kunna köra det ursprungliga programmet samtidigt som vi har en js-fil med bara funktioner som vi kan testa i jest.

4 – Testa våra egna funktioner med Jest, grunderna

I denna övning gör jag antagandet att filen som alla våra funktioner ligger i heter **functions.js** samt att testfilen heter **functions.test.js**. Om du vill döpa om filerna till det här, kom ihåg att döpa om sökvägarna i html-filerna om ni fortfarande vill köra programmet i en webläsare!

Högst upp i vår test-fil importerar vi alla funktionerna på följande vis:

```
const functions = require("./functions");
```

1. När vi testar samma funktion eller samma funktionalitet kan man gruppera in de testen i en **describe-funktion**. Den fungerar som ett block för tester som är relevanta med varandra. Tex:

```
describe('my beverage', () => {

    const myBeverage = {
       delicious: true,
        sour: false,
    };

    test('is delicious', () => {
        expect(myBeverage.delicious).toBeTruthy();
    });

    test('is not sour', () => {
        expect(myBeverage.sour).toBeFalsy();
    });
});
```

   Man försöker uttrycka sig i hela meningar från ***describe*** till ***test*** på det man testar som exemplet över visar.
2. Skapa ett describe block med 3 tester där vi testar funktionen emptyList:
   - ska returnera en tom lista
   - ska returnera en lista vars längd är 0 (använd **toHaveLength(number)**)
   - ska inte returnera en tom sträng
3. Skapa ett describe block med 3 tester där vi testar funktionen totalProductCost:
   - ska returnera 0 på en tom lista
   - ska returnera 15 på en lista med ett föremål som kostar 15
   - ska returnera 55 på en lista med 3 föremål som kostar 10, 15 och 30
4. Skapa ett describe block med 2 tester där vi testar funktionen addProduct:
   - lägger till en produkt korrekt på en tom lista – är listans längd 1 och är innehållet korrekt?
   - lägger till en produkt korrekt på en lista med 2 produkter – är listans längd 3 och är innehållet korrekt?

   Det är helt okej att göra flera expect-anrop i ett test! Testa gärna match-funktionerna **toContain(item)** och **toContainEqual(item)** för arrays!