***Hoofdstuk 11: asynchronous programming***

Het zou heel onhandig zijn als ik een koffie zou bestellen, deze vervolgens eerst helemaal gemaakt en geserveerd zou worden en vervolgens pas de volgende klant geholpen kan worden 🡪 **synchronous programming.** Met **asynchronous programming** kan dit opgesplitst worden en kunnen er andere taken worden uitgevoerd terwijl iets anders wacht:



Het wordt wel een stuk moeilijker om een asynchronous program te begrijpen.

Dit kan je doen door het gebruik van **callback functions.** Een voorbeeld:

*import {defineRequestType} from "./crow-tech";*

*defineRequestType("note", (nest, content, source, done) => {*

*console.log(`${nest.name} received note: ${content}`);*

*done();*

*});*

Asynchronous programming is hierin besmettelijk: een functie die een asynchronous functie aanroept moet dat zelf ook zijn 🡪 niet ideaal. Met een **promise** kan dit handiger:

*function storage(nest, name) {*

*return new Promise(resolve => {*

*nest.readStorage(name, result => resolve(result));*

*});*

*}*

*storage(bigOak, "enemies")*

*.then(value => console.log("Got", value));*

Deze promise kan **resolved** (klaar)of **rejected** worden en verwacht een functie als argumenten. Een promised value is een waarde die er misschien al is of misschien in de toekomst verschijnt en dan gebruikt kan worden. Wanneer een promise resolved wordt, dan wordt **then** gecalled (kan een totaal andere functie zijn). Als het gereject wordt, dan wordt **catch** gebruikt (kan ook een totaal andere fucntie zijn).

Het moeilijke aan callback functies is dat het erg guidelijk moet zijn waar failures vandaan komen en dit niet wordt gebruikt in de callback. 🡪 vaak wordt dit gedaan door het eerste argument van de callback of een actie heeft gefaald en de tweede de waarde die wordt geproduceerd bij een succes. Met een promise gaat dit nog makkelijker, omdat deze een resolved of rejected teruggeeft. Meestal geeft de rejection dan ook een reason mee (waarom kon het niet?). De **catch** method wordt daarvoor gebruikt.

**Flooding** = het netwerk wordt geflood met een stuk informatie tot alles het in bezit heeft.

Door **async funtie () =>** te schrijven kun je aangeven dat het async moet gaan worden. Het geeft dan een promise terug. Vervolgens kun je binnen zo’n **await** neerzetten voor elke expression. Deze kan functie kan bevroren worden bij elke await en later weer doorgaan. 🡪 geeft altijd promises terug.

Met **function\*** werkt het hetzelfde qua pauzeren, alleen heb je geen promises nodig. Dit heet een **generator**:

*function\* powers(n) {*

*for (let current = n;; current \*= n) {*

*yield current;*

*}*

*}*

*for (let power of powers(3)) {*

*if (power > 50) break;*

*console.log(power);*

*}*

*// → 3*

*// → 9*

*// → 27*