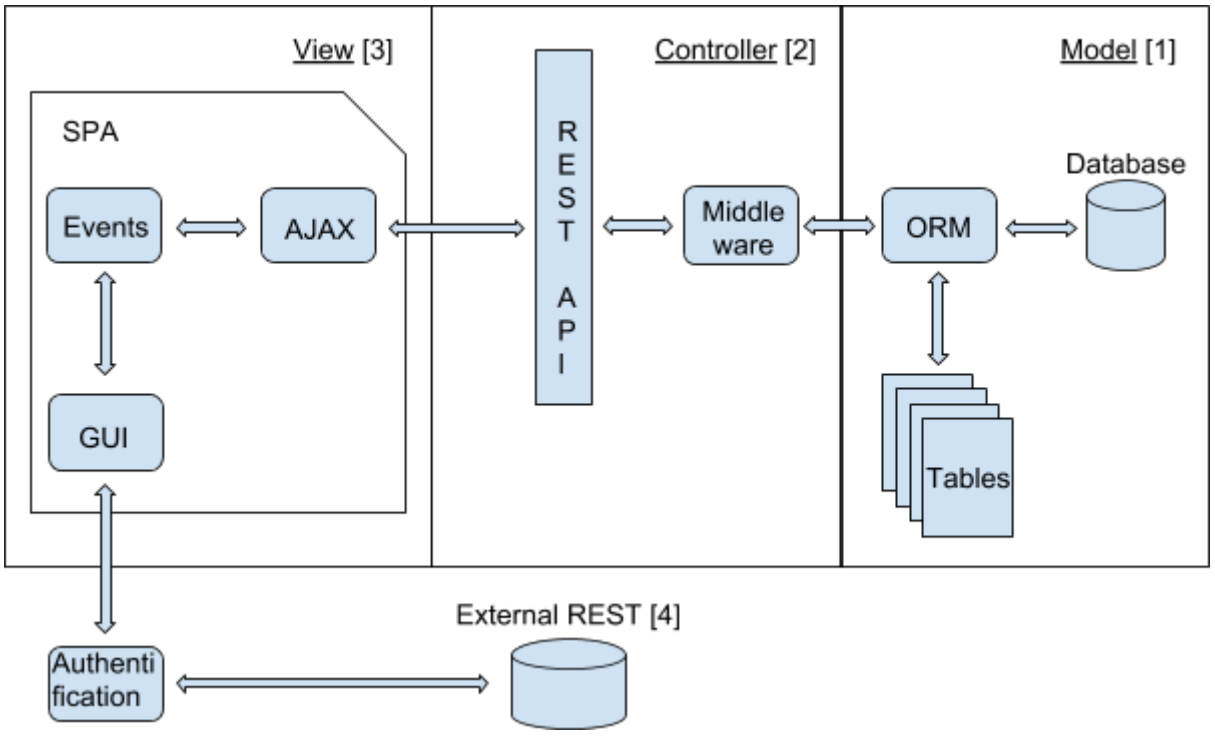
Rapport - DAT076, Grupp 1

Adrian Lindberg
Jonathan Sundkvist
Jonatan Nylund
Johannes Matsson

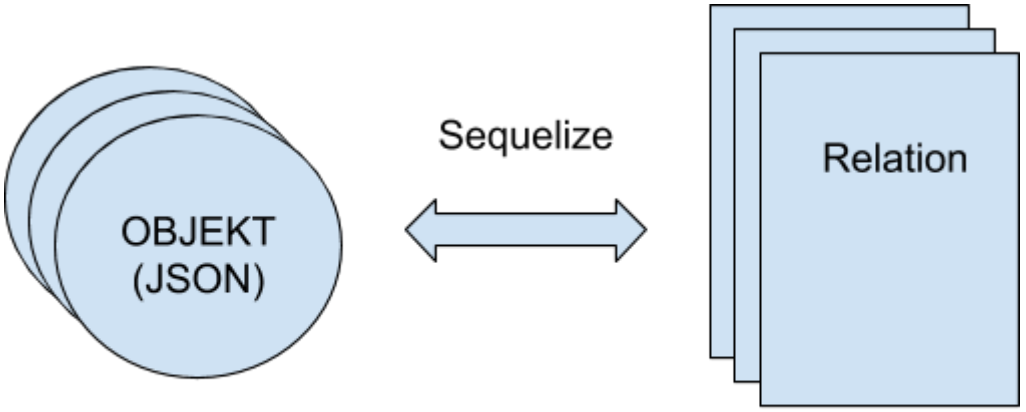
Applikationen vi konstruerat är en webbshop (se bilagor för screenshot på applikationen). Det är en fullstack-applikation med Node.js som server-ramverk. Anledningarna till att valet blev en webbshop är flera. En av dem är att vi ville arbeta med Node.js, vilket lämpar sig väl för applikationer som är mer lättviktiga (till skillnad från JEE). En webbshop är ett tydligt sådant exempel, där många anrop sker men i regel få beräkningar behöver utföras. En ytterligare motivering till varför vi valde att konstruera en webbshop är att en sådan lämpar sig väl att applicera ett MVC-mönster på samtidigt som den erbjuder bra möjligheter till expansion och/eller avgränsningar.

En av anledningarna till att vi ville arbeta med Node.js är dess ökade popularitet och att det därigenom finns många bra och tillgängliga ramverk och bibliotek att ta användning av. I detta projekt har ett flertal sådana används, där några av de mest utmärkande varit React för frontend (löser AJAX, eventhantering mm), Express för backends controller-skikt och Sequelize, ett ORM-bibliotek, för databas-lagret. Alla dessa ramverk och bibliotek, deras betydelse för vårt projekt och hur vi kom att använda dem kommer att fördjupa mer senare i rapporten.

Som nämnt är applikationen designad efter MVC-mönstret och resonemang och argumentation kring övergripande arkitektur, ramverk, bibliotek och designval utgår med fördel från följande figur:



Model [1]
Databasen som valts att användas är MySQL, men i och med användandet av biblioteket Sequelize för vår ORM-hantering, abstraheras koden på ett sådant sätt att MySQL skulle kunna bytas ut till en annan relationsdatabas Sequelize har stöd för (PostGres, SQLite etc). Alltså, Sequelize hjälper oss med två saker samtidigt. Dels hjälper det till att abstrahera bort ett specifikt databasspråksberoende (detta bestäms i konfigurationsfilen för Sequelize och sequelize transpilerar därefter koden till rätt databasspråk beroende vad som angivits i filen) och dels löser den mappningen från javascript-objekt (JSON) till relationerna i tabellerna och vice versa. Detta medför att programmeraren inte behöver kunna databasens skriptspråk (i detta fall SQL) för att kunna kommunicera med databasen.



ORM (**O**bject-**R**elation-**M**apping)

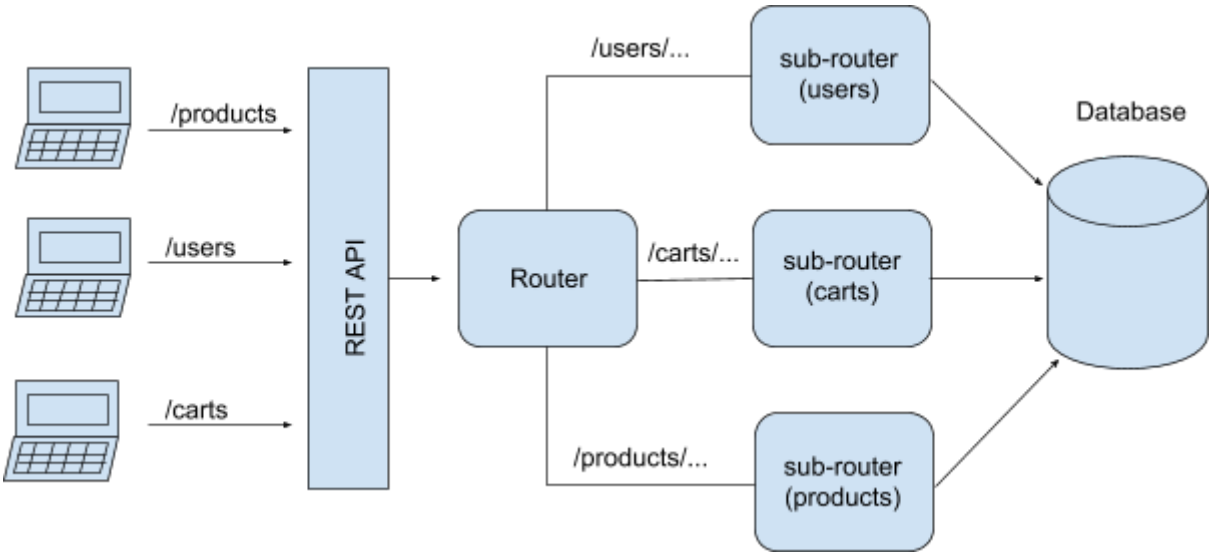
Utöver att Sequelize möjliggör lösa kopplingar till databasen har också möda lagts ner på att designa databasen modulärt. Alla modeller har extraherats ut till egna js-filer och importeras senare i önskad ordning till index.js (ibland kallad db.js). Samma angreppssätt har applicerats på skapandet av dummydata och hur populerandet av data sker. Fördelen med detta upplägg är att projektstrukturen blir mer lättnavigerad, koden mindre plottrig, men framförallt att assocaitonera mellan tabellerna mer lätthanterliga. Alla tabeller i applikationen har tilldelats nödvändiga associationer - vilket exempelvis medför att om en produkt som har ett antal reviews kopplat till sig (review = separat entitet som har product_id som FK) tas bort, tas också alla reviews som syftar på den produkten bort osv. Att använda associationer i databasen gör den mer robust och tillförlitlig.

Modellen och databasen länkas samman med controllern i figuren med hjälp av Middleware och ramverket Express.js.

Controller [2]

Genom användandet av ramverket Express.js i controller-skiktet underlättas konstruerandet av REST API:et avsevärt. Den främsta vinsten med Express i denna applikation är Express.router som gör det enklare att hantera endpoints. Dels förenklar det hanteringen av http-anropen men också hur man kan använda det för att konstruera mer modulär och

elegant kod (med hjälp av sub-routrar). Express.router tillåter alltså en att kunna separera och skilja endpoints åt. I vår applikation manifesteras detta genom att det finns en users.js-subrouter som hanterar alla endpoints som syftar på att kommunicera med users i databasens, en carts.js för carts (kundvagn) osv. Genom att utnyttja denna funktionalitet som erbjuds i Express.js blir såväl projektstrukturen som koden mindre klottrig, mer lättläst och lättare att utöka (samma argument som vid konstruktion av databasskiktet). I ett större projekt blir det än tydligare då antalet endpoints växer i takt med applikationens ökade krav på funktionalitet och antalet entiteter i databasen man behöver kommunicera med, vilket belyser vinsten med att använda middleware som exempelvis Express.js.

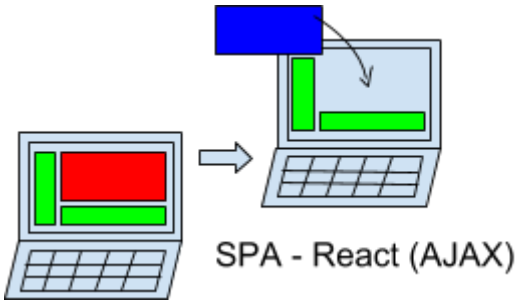


All kommunikation med databasen i applikationen sker via det REST API vi konstruerat med hjälp av dessa routrar (en fullständig lista på vilka dessa endpoints är finns här: https://github.com/lindbergan/dat076project/tree/master/react-backend/routes).

Genom att bistå frontend med ett REST API konstruerar man en tydlig ansvarsuppdelning mellan vad som utgör backend respektive frontend. Detta är inte bara positivt i termer av design (man kan byta ut frontend eller backend och behålla den andra delen) utan underlättar också arbetet då det möjliggör en tydligare ansvarsfördelning.

View [3]
I frontend används ramverket React.js. Motiveringarna till detta är flera. En av de stora fördelarna är att React.js gör det möjligt att skriva jsx-kod (javascript-liknande kod) och därefter transpilera om detta till html och css. En annan är att det har ett bra inbyggt stöd för olika AJAX-bibliotek, vilket gör det enklare att konstruera en single page application (SPA).

Genom att konstruera applikationen efter SPA minskas mängden datatrafik som flödar mellan klient och server, vilket minskar belastningen för båda parter samtidigt som det gör responstiden kortare och att applikation upplevs som mer responsiv. Webshopen är byggd efter iden om SPA med hjälp av React router, där enbart berörda komponenter renderas om vid användarinteraktion istället för att hela sidan renderas om.



Detta öppnar upp för en mer modulär design i form av komponenter, vilket likt resonemangen i backend bidrar till en tydligare projektstruktur och utbyggbarhet. En annan fördel med att använda React.js är att det tillåter - likt biblioteket Mustasch för html - att man dynamiskt kan ändra värdena på taggarna i de olika komponenterna. Detta, i kombination med att React utnyttjar jsx, gör att Mustasch inte har behövts användas i projektet.

När det gäller den grafiska designen har ramverket Bootstrap använts för att få en mer elegant och responsiv design. Även en del andra bibliotek har använts för att få designen mer uttrycksfull och samtidigt undvika att “konstruera hjulet på nytt”. Ett sådant exempel är biblioteket Material UI.

Utöver ramverk och bibliotek är ett vanligt och utmärkande inslag i webbutveckling användandet av externa REST API:er, vilket vi också använt oss av i denna applikation.

External REST [4]
Vi har använt oss av externa API:er för att hantera autentisering/inloggning till applikationen. Genom att delegera denna funktionalitet till ett externt REST API där användarens uppgifter (med undantag av lösenord) returneras, medför det en ökad säkerhet och bekvämlighet för användaren (ett mindre lösenord och användarnamn att hålla reda på samt att det troligtvis är större risk att vår sida äventyras än googles). Idealt hade man dock också erbjudit användaren att skapa ett inlogg på vår sida (för de som inte har ett konto hos de externa API-leverantörerna). Detta prioriterades dock inte till förmån för annan - viktigare - funktionalitet.

Att använda och kunna hantera befintliga ramverk, bibliotek och REST API:er utgör en viktig inom webbapplikationer. Mycket av webbens framgång bygger på dess öppenhet och utbyggbarhet (snabba förändringar). Vi har försökt att i så stor mån som möjligt försökt applicera denna ide i allt från val av server- till stylingramverk. Det tar inte bort betydelsen av att fortfarande kunna konstruera genomtänkt och modulär kod, men möjliggör att man kan bygga smarta, responsiva, utbyggbara och robusta webbapplikationer inom en rimlig tidsram.

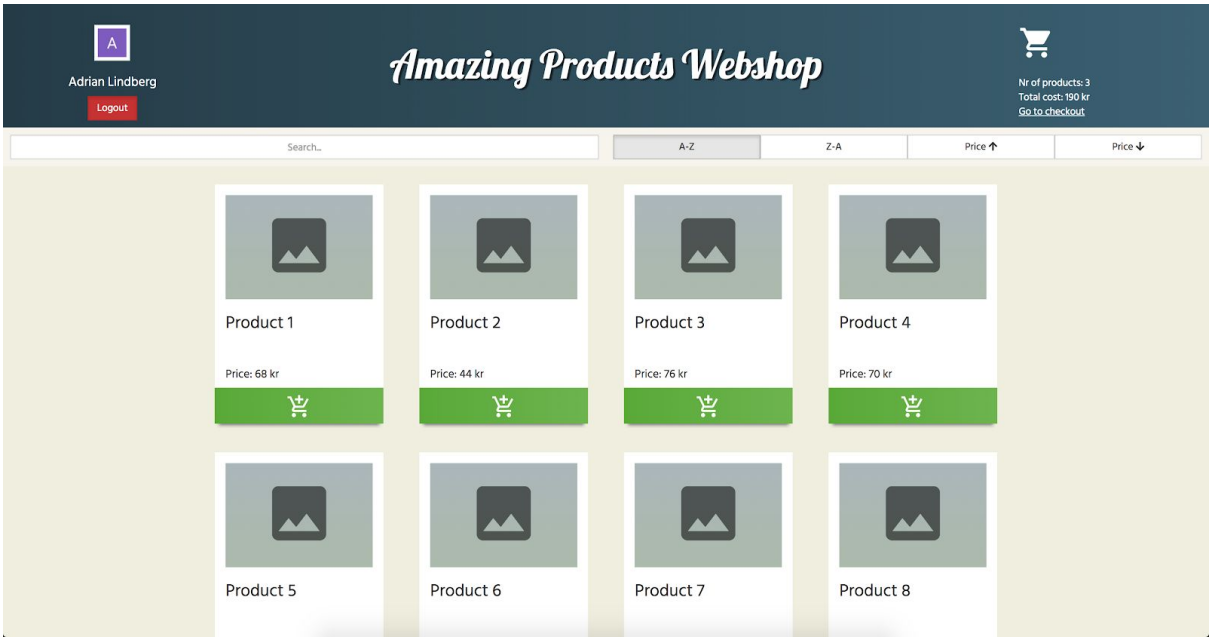
BILAGOR

Use Cases

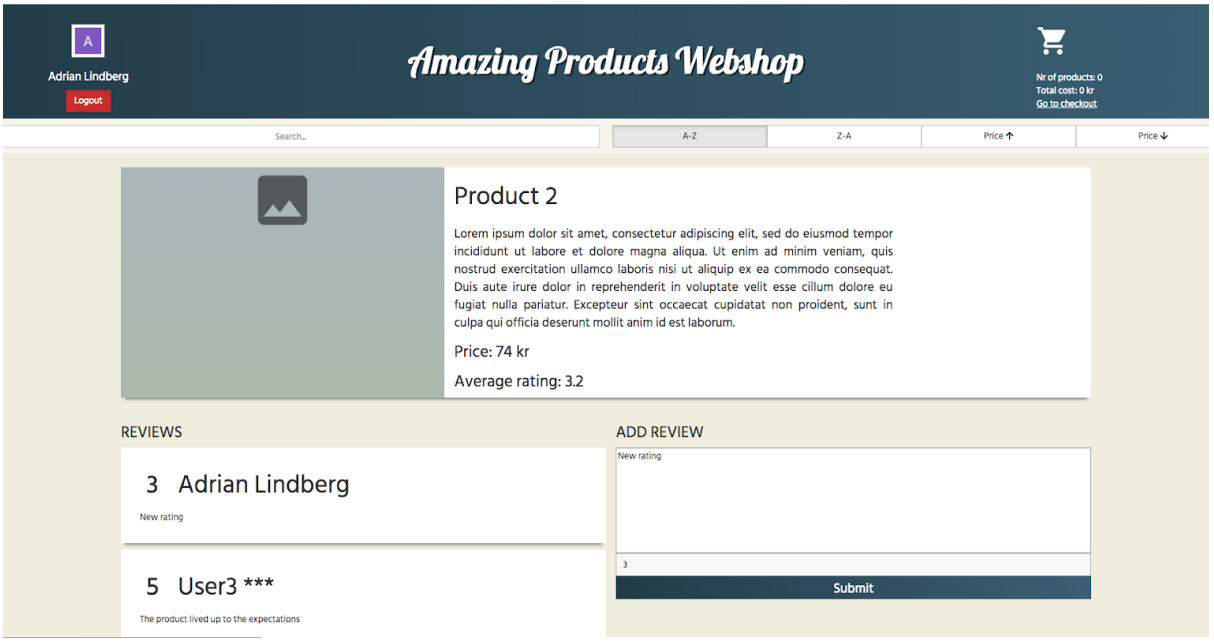
Nedan följer de UC som implementerats och i legat till grund för byggandet:

1. En kund skall kunna interagera med kundvagnen (lägga till, ta bort, justera antal varor i kundvagnen)
2. En användare skall kunna klicka på en produkt och få upp en produktvy med information om produkten (detailed view)
3. En användare skall kunna söka på sidan
4. En användare skall kunna bli meddelad om informationen hen har skrivit in i kunduppgiftsfälten (betalvy) är korrekt ifyllda eller ej (regexp, säkerhetsfråga)
5. En användare skall kunna sortera produkterna efter diverse attribut (pris stigande, pris fallande, namn etc)
6. En kund skall kunna lämna omdöme på en produkt, vilket innefattar:
    6.1. ratings
    6.2. comment (comments är ett attribut i Review-tabellen)
7. En kund skall vid checkout kunna få en bekräftelse med kvitto.
8. En användare skall kunna logga in (helst genom extern tjänst, GoogleAuth, FB etc - vi slipper bygga in säkerhet och bra att träna på)
9. Hemsidan ska vara responsiv och anpassa sig till alla vanliga storlekar.
10. Associations mellan tabellerna. Tabellerna skall ha fungerande dependencies.

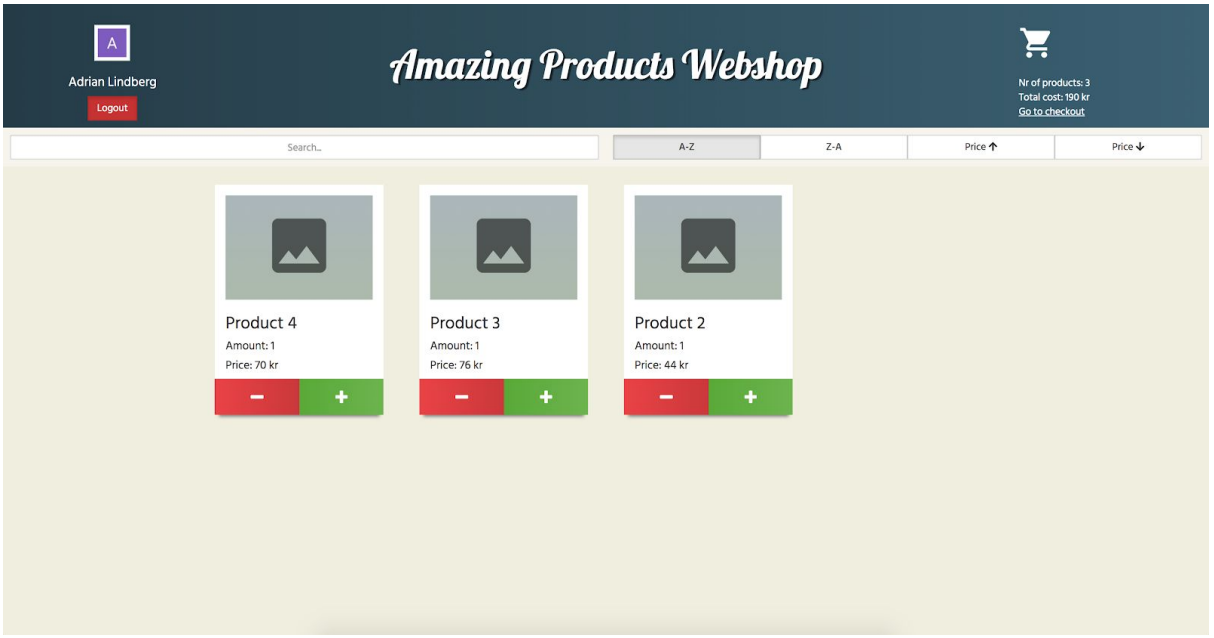
Bilder på den färdiga applikationen



Startvyn



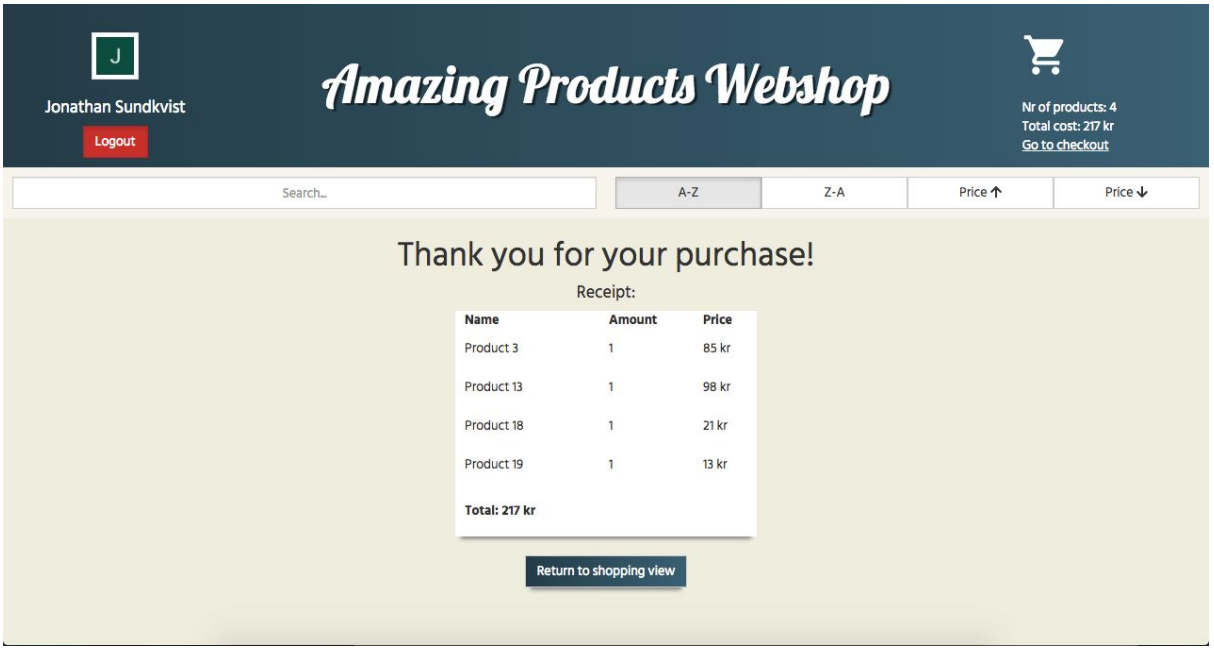
Detaljvyn



Kassavyn

Kundvagnen

Betalvy



Konfirmation/kvitto