

# Cupcake

CPH Business Lyngby

Datamatiker 2. Semester efterår 2017



<b>Indledning</b>	<b>3</b>
<b>Teknologi valg</b>	<b>3</b>
<b>Domæne model og ER diagram</b>	<b>7</b>
<b>Navigationsdiagram</b>	<b>8</b>
<b>Sekvens diagrammer</b>	<b>10</b>
Login	10
<b>Særlige forhold</b>	<b>14</b>
MVC (Model View Control) designmønster	14
Forward & Session	15
Login system	15
Registrer ny bruger	16
Shop og cart	18
Invoice	19
<b>Status på implementation</b>	<b>20</b>

## Indledning

Målet med denne opgave var at kombinere backend og frontend ved brug af Java, HTML, CSS, JSP, Javascript og MySQL til en webapplikation. Webapplikationen skulle være en webshop hvor kunder kan købe cupcakes via en bruger med indbetalte penge. Og samtidig kunne se tidligere ordre.

## Teknologi valg

Til dette projekt har vi skulle bruge en del forskellige teknologier. Når man vælger sine værktøjer vægter man dets fordele og ulemper op mod hinanden og vælger de værktøjer der bedst vil kunne realisere kundens ønsker. Eller man vælger de værktøjer man er bekendt med, hvilket vi har gjort i dette projekt.

### **Netbeans IDE 8.2**

Dette er den udviklingssuite som vi primært arbejder i. Netbeans understøtter mange forskellige programmeringssprog, men vi har i dette projekt brugt Java, da det er det vi er bekendt med.

### **JDK 1.8**

Dette er en forkortelse for "Java development kit" med dets versionsnummer. Og er grundlæggende selve Java-strukturen, med alle dets indbyggede klasser og funktioner.

### **Java EE 7 Web**

Er en indbygget funktionalitet i Netbeans IDE'en som tager gør det nemmere for udviklere at lave webapplikationer, ved at prækonfigurere mange af de aspekter man normalt ville skulle sætte op manuelt. Såsom XML og web containers.

### **JDBC**

Er en forkortelse for "Java Data-Base Connector", det er ret selvforklarende vi bruger JDBC til at forbinde vores Java webapplikation til en database.

Vores implementation af JDBC ser således ud:

```
1  package data;
2
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.SQLException;
6
7  public class DB {
8
9      private static final String DRIVER = "com.mysql.jdbc.Driver";
10     private static final String URL = "jdbc:mysql://localhost:3306/cupcake";
11     private static final String USER = "root";
12     private static final String PASSWORD = "rootmaster!42";
13     private static Connection conn = null;
14
15     public static Connection getConnection() {
16         if (conn == null) {
17             try {
18                 Class.forName(DRIVER);
19                 conn = DriverManager.getConnection(URL, USER, PASSWORD);
20             } catch (ClassNotFoundException ex) {
21                 ex.printStackTrace();
22             } catch (SQLException ex) {
23                 //se hele sekvensen til det gik galt. Dette kan skrives til logfil.
24                 ex.printStackTrace();
25             }
26         }
27         return conn;
28     }
29 }
```

Man vil skulle ændre på linje 10-12 hvis man vil deploye den på sin virtuelle computer på DigitalOcean, da den ikke arbejder via localhost og brugeren og kodeordet er nødvendigvis ikke det samme. Ud over dette vil forbindelsen som standard være sat til NULL som vi ser på linje 13, hvilket går ned i metoden fra linje 15 til og med linje 28. Som bare er en metode der tjekker om der er hul igennem.

### Apache Tomcat 8.0.27.0

Tomcat er en applikationsserver, hvilket kort fortalt er den teknologi vi bruger til at køre vores program og hjemmeside. Tomcat giver en mulighed for at "deploy" en Java webapplikations WAR-fil. Vi vil ikke uddybe hvad WAR-filer er, udover at det er en autogeneret fil som indeholder specifikke instrukser om en Java applikation.

## MySQL Workbench 6.3 CE

Denne teknologi bruger vi til at lave databaser, så alt vores data ikke forsvinder så snart vi lukker hverken Netbeans eller Browseren. Her er et eksempel fra vores SQL-script. Dette udsnit viser strukturen for hvordan vores tabeller ser ud.

```
20 -----
21 -- Table `cupcake`.`user`
22 -----
23 CREATE TABLE IF NOT EXISTS `cupcake`.`user` (
24   `username` VARCHAR(45) NOT NULL,
25   `password` VARCHAR(45) NOT NULL,
26   `admin` TINYINT(1) NULL DEFAULT NULL,
27   `userId` INT(11) NOT NULL,
28   `balance` INT(10) UNSIGNED NULL DEFAULT NULL,
29   `email` VARCHAR(45) NULL DEFAULT NULL,
30   PRIMARY KEY (`userId`),
31   UNIQUE INDEX `username` (`username` ASC))
32 ENGINE = InnoDB
33 DEFAULT CHARACTER SET = utf8;
```

Vores SQL-script er udarbejdet fra en SQL-model og har derfor nogle prædefineret linjer som man normalt ikke behøver at skrive når man laver en tabel. Linje 32 og 33 er ikke nødvendige for at scriptet vil kunne køre.

Kontakt mellem Netbeans og MySQL klares gennem nogle datamapper klasser. Her er et udsnit fra vores userMapper-klasse:

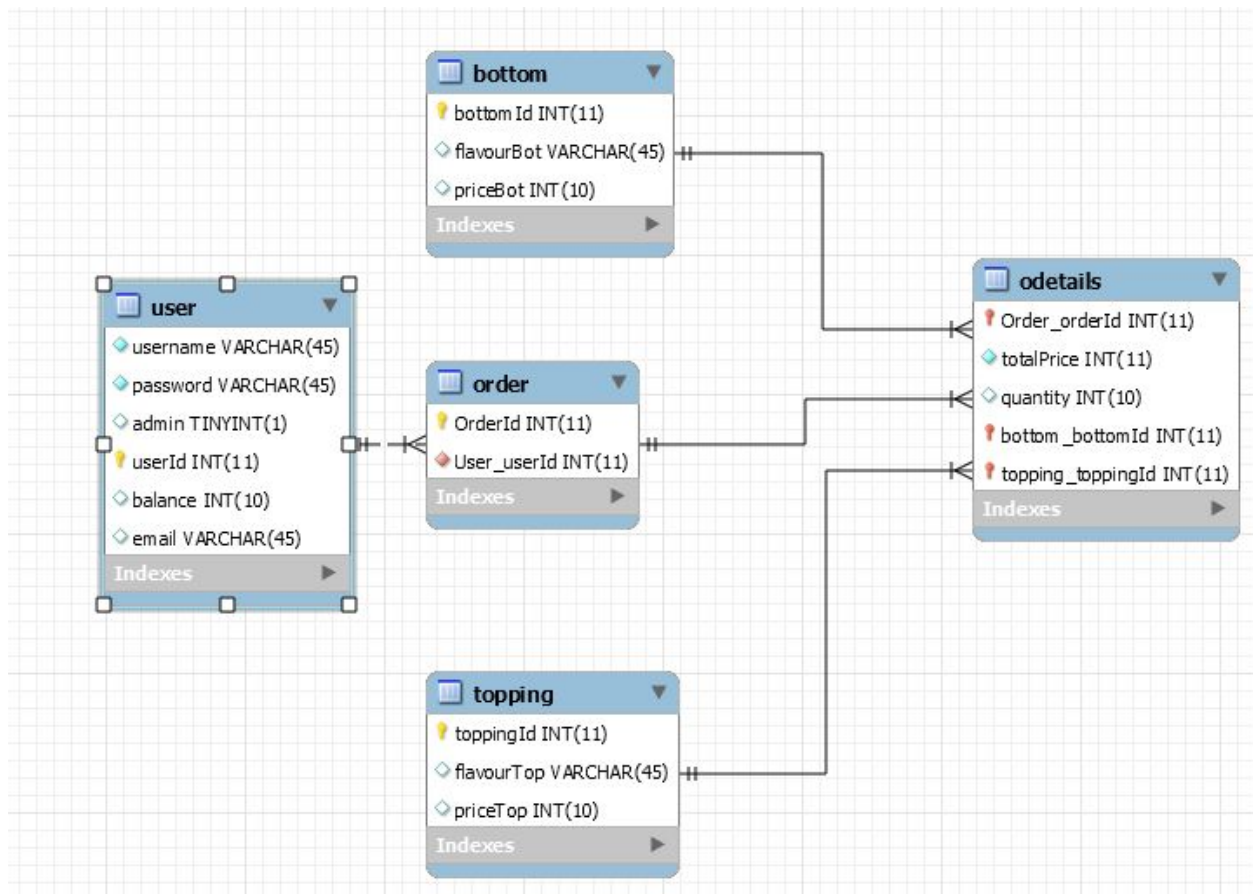
```
25 public User getUser(String u) {  
26     ResultSet rs = null;  
27     PreparedStatement stmt = null;  
28     User user = null;  
29  
30     String SQLString  
31         = "select * from user where username = ?";  
32     try {  
33         stmt = con.prepareStatement(SQLString);  
34         stmt.setString(1, u);  
35         rs = stmt.executeQuery();  
36  
37         if (rs.next()) {  
38             user = new User(u, rs.getString("password"), rs.getBoolean("admin"),  
39                             rs.getInt("userId"), rs.getInt("balance"), rs.getString("email"));  
40         }  
41     } catch (SQLException e) {  
42         System.out.println("Fail in dataMapper - getUser");  
43         e.printStackTrace();  
44     }  
45     return user;  
46 }
```

Denne metode getUser bruger et preparedStatement til at tilgå databasen.

## PlantUML

Dette er navnet på programmet brugt til at lave alle de diagrammer der bruges i denne rapport

## Domæne model og ER diagram



Vi har lavet et ER-diagram af vores database, som danner et meget godt overblik over vores system. I vores database har vi lavet en entity user tabel, hvor vi har alle user relaterede attributter, vores user tabel har userId som primary key, da hver user får deres egen unikke id. Fra vores user tabel har vi en, en til mange relation til ordrer, dette gør vi fordi, en user godt kan have mange ordrer, men en ordrer kan ikke have mange brugere. I vores order tabel har vi to attributter, den første attribut er OrderId

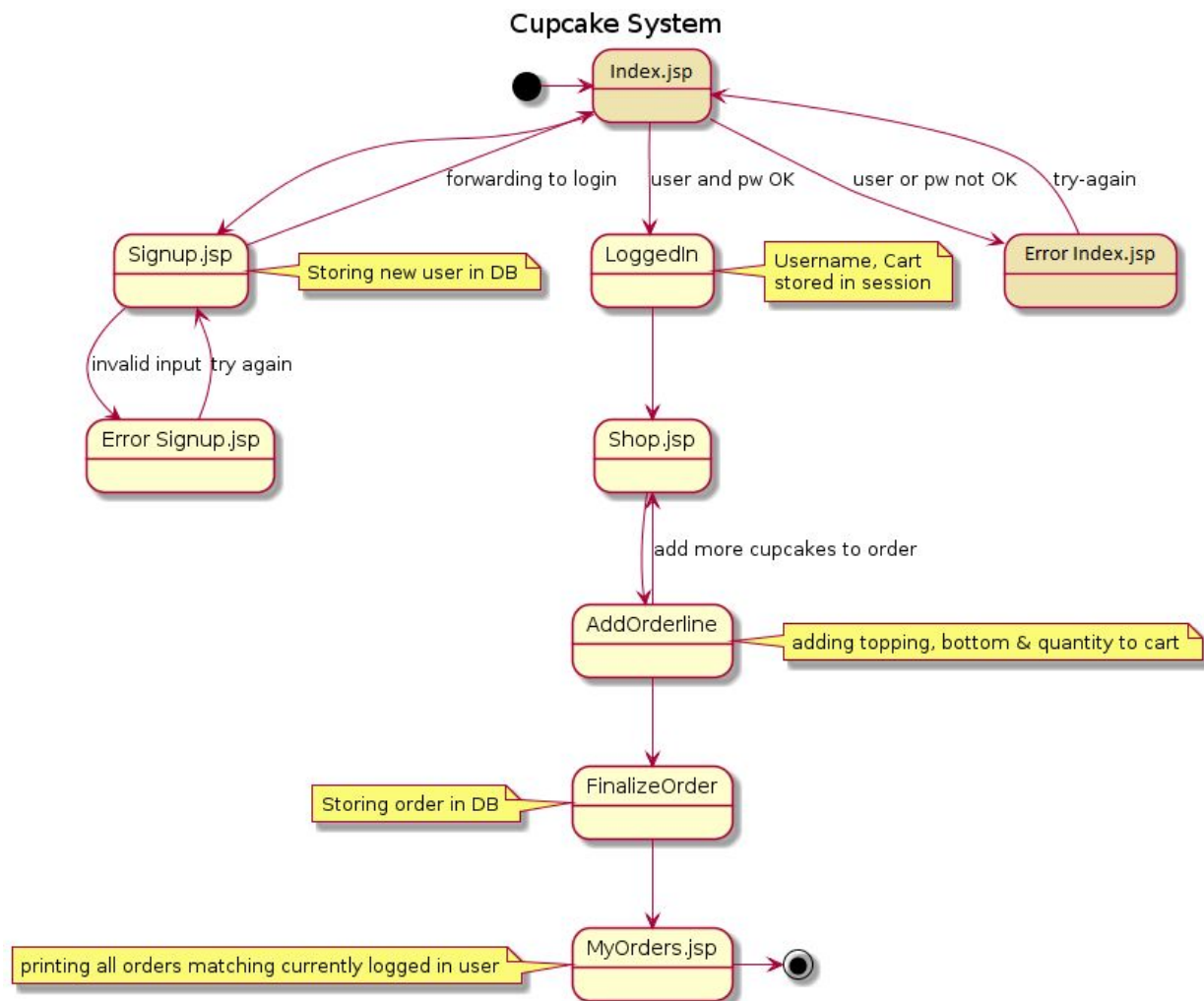


som er en unik primary key. Vi har også en foreign key på UserId, så order tabellen gemmer hvilken bruger der har lavet ordren. Fra vores order tabel har vi lavet en, en til mange relation til odetails, dette er gjort fordi, en ordre kan eksempelvis godt have mange toppingId og bottomId, men odetails kan ikke have flere forskellige ordrer. Inde i vores odetails tabel, har vi alle detaljer, om brugerens ordrer. Vi har derfor lavet de nødvendige attributter så odetails kan indeholde alle detaljer, dette er blandt andet gjort ved, at vi har 3 foreign keys, så odetails kan få fat i værdierne fra order, topping og bottom tabellerne. Derudover har vi i odetails 3 foreign keys, som danner vores primary key, altså en sammensat nøgle.

Odetails har en, mange til en relation til topping og bottom, da odetails kan have mange cupcake toppe og bunde. Vores topping og bottom tabeller, er nærmest ens, den ene indeholder attributter for cupcake toppe, og den anden for cupcake bunde, de har begge tre attributter, en unik primary key, et navn(flavour) og en pris.



## Navigationsdiagram



Vi har lavet et navigationsdiagram over vores cupcake projekt. Vi starter på vores **Index.jsp** eller **Signup.jsp**, hvis brugeren ikke har en bruger at logge ind med, kan de vælge at starte på **Signup.jsp** og lave en ny bruger, brugerens inputs bliver så valideret i vores **RegisterServlet**. Hvis inputtene er forkerte, bliver brugeren redirected til **Signup.jsp**, hvor de igen kan få lov til, at prøve at lave en ny bruger. Hvis inputtene er korrekte bliver brugeren redirected til **Index.jsp**.

På **Index.jsp** kan brugeren indtaste username og password, hvor **LoginServlet** tjekker om username password er indtastet forkert eller rigtigt, hvis brugeren indtaster forkert bliver de redirected tilbage til login, hvilket er vist med **Error Index.jsp**. Når brugeren har

indtastet username og password korrekt, vil de komme ind på shop.jsp.

På shop.jsp som vores ProductServlet styrer, kan brugeren vælge en cupcake bund, top og antal. For at tilføje deres inputs, skal kunden trykke på addOrderline, som tilføjer deres valg til indkøbsvognen, brugeren kan med addOrderline, blive ved at tilføje cupcakes til deres indkøbsvogn. Når brugeren er færdig med at vælge cupcakes, kan kunden trykke på en knap på Shop.jsp der hedder FinalizeOrder, som tilføjer indkøbsvognen som brugeren har tilføjet til databasen, og FinalizeOrder redirecter derefter til en side der hedder, MyOrders.jsp, som printer en ArrayList ud med alle brugerens ordrer.

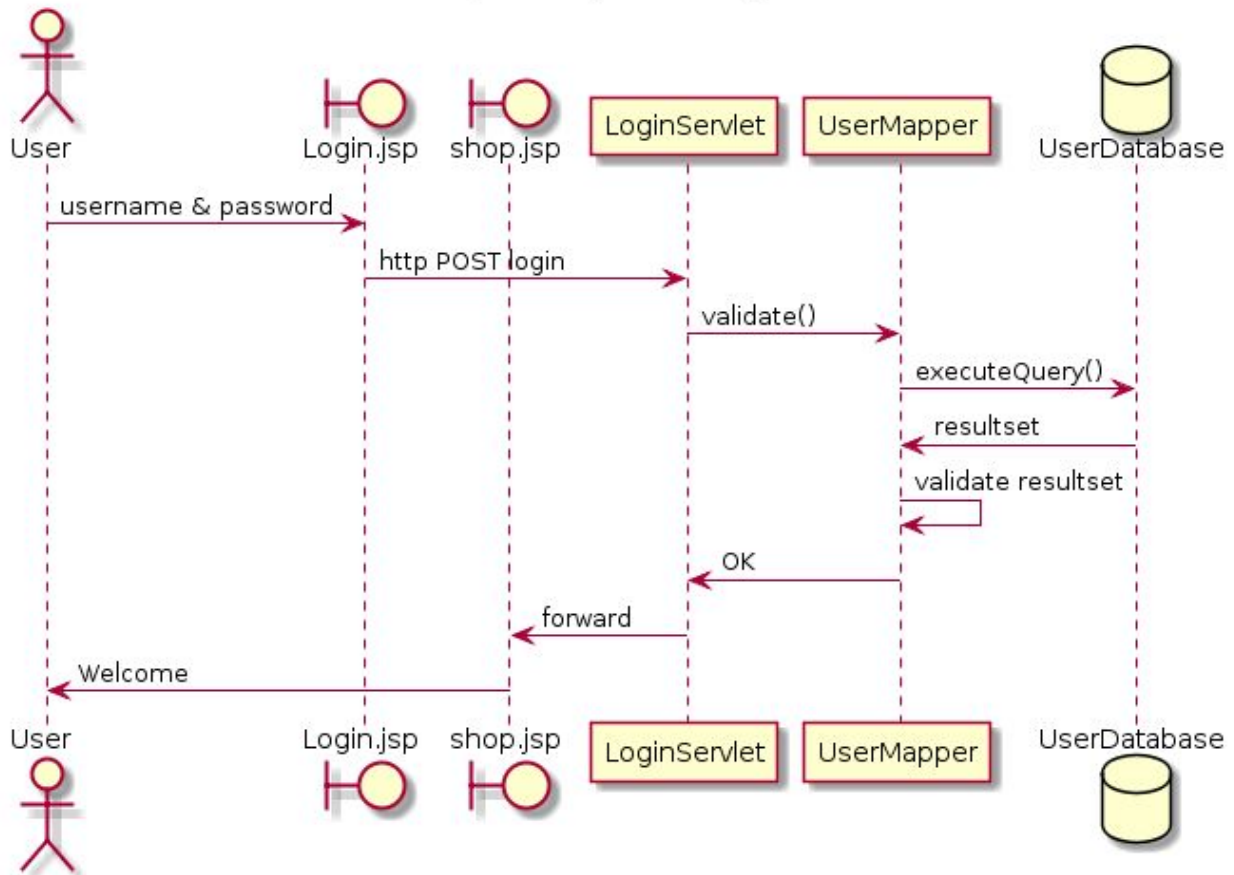
## Sekvens diagrammer

I dette afsnit vil vi gennemgå 2 af vores systemer, navnlig vores login system og vores indkøbskurv på vores hjemmeside.

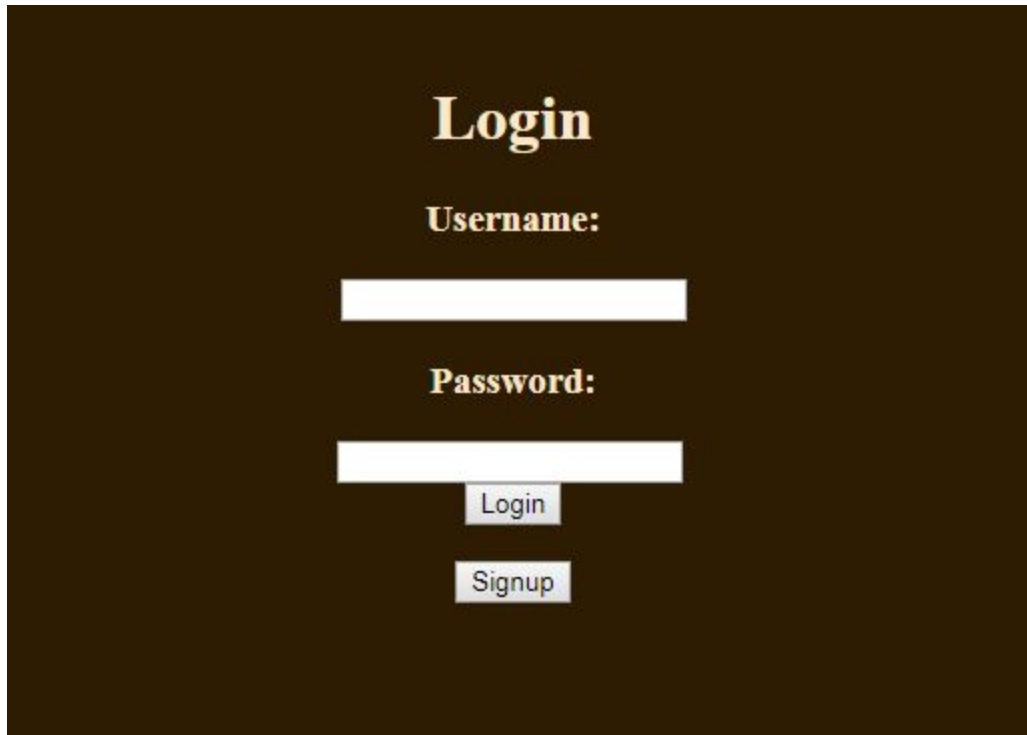
### Login

Vi starter med at kigge på vores login system da det alligevel er det første brugeren vil støde på. Nedenfor ses sekvens diagrammet for dette system.

## "Login - Sequence Diagram"



Brugeren vil først blive mødt af en login side(index.jsp) i sin browser som ser således ud:



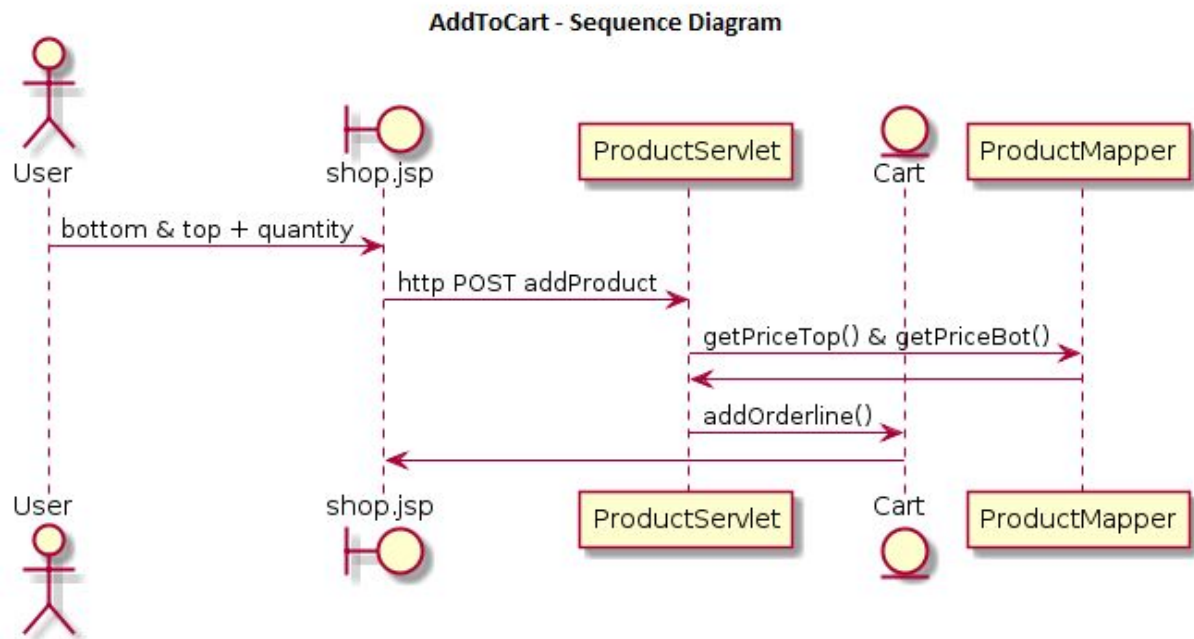
Her bliver brugeren så præsenteret for 2 tekstfelter som skal udfyldes hvis man ønsker at logge ind. Der er også mulighed for at oprette sig som bruger på hjemmesiden via “signup” knappen nederst på siden. Disse valg bliver behandlet af 2 servlets kaldt “LoginServlet” og “RegisterServlet”.

Brugeren skal så udfylde felterne “Username” og “Password”-feltet og derefter trykke på “Login”-knappen. LoginServleten tager så de informationer som brugeren har indtastet og tjekker om der findes en bruger med det brugernavn og kodeord i databasen via en validate metode. Hvis der findes en bruger med de indtastede informationer vil brugeren blive ført videre ind på hjemmesiden. Hvis ikke brugeren eksisterer i databasen vil han/hun blive sendt til signup-siden(signup.jsp) hvor brugeren så vil kunne oprette sig.

## Cart

Nu er brugeren så logget ind og vil gerne oprette en ordre så han/hun kan nyde nogle lækre cupcakes.

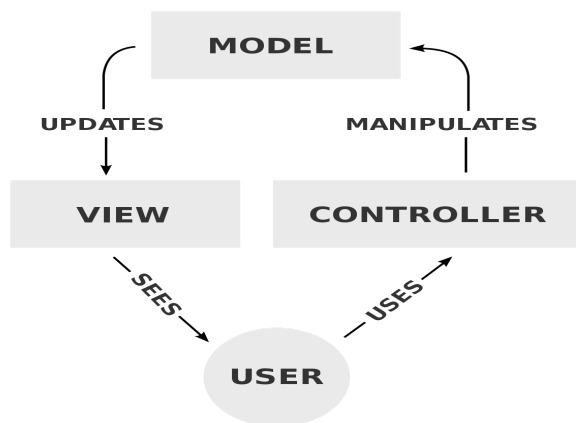
Nedenfor ses sekvens diagrammet for vores indkøbskurv når en bruger tilføjer én slags cupcake til sin ordre:



Brugeren vil blive bedt om at udfylde nogle felter på shopsiden(shop.jsp), navnlig 2-dropdown menuer(bottom og top) og 1 inputfelt(quantity), og til sidst trykke på addOrder-knappen. ProductServleten modtager så disse input og henter de tilsvarende produkter, samt deres respektive priser fra databasen, der vil i samme instans blive lavet en ordrelinje, som først kigger på indkøbskurvens(cart) indhold, hvorefter den sender og fremviser disse informationer via ProductServlet til siden(shop.jsp). Brugeren vil så kunne tilføje endnu et produkt eller afslutte sin ordre ved at trykke på "Finalize" på samme side.

## Særlige forhold

MVC (Model View Control) designmønster



Projektet er bygget på MVC designmønstret.

MVC designmønstret er delt op i tre lag, model-laget som står for data og logic for eksempel entity klasser og de forskellige datamappers i projektet som enten inserter data i vores database eller hiver den ud. Model laget får instrukser fra Control-laget, hvor efter data'en bliver gemt eller behandlet af model-laget. View-laget består af vores JSP-sider, som altså er vores UI. JSP-siderne er ansvarlige for, at genererer og vise outputtet til vores kunde. Control-laget står for kommunikationen mellem de 2 andre lag og er derfor limen i projektet der får det hele til at fungerer sammen.

Eksempel:

Brugeren indtaster sine login-oplysninger og trykker på login-knappen, og sender et kald til LoginServlet. LoginServlet som er vores controller, kalder metoden `getUser()` i

vores UserMapper som derefter returnerer et User-objekt, som bliver valideret med metoden validateLogin(). Hvis valideringen er succesfuld vil brugeren blive sat fast på sessionen og blive sendt videre til et af vores views, Shop.jsp.

## Forward & Session

Vi sætter for det meste objekter eller data fast på sessionen som tit skal være tilgængelige, det er blandt andet User-objektet, det aktuelle Cart-objekt for den bruger som er logget ind og vores lister af toppings og bottoms. Dette gør vi for at kunne benytte os af request.getParameter() så vi kan sende informationerne videre med en bestemt 'action' eller 'origin'.

## Login system

I det brugeren taster sit username, password og trykker på knappen sender den en action som er lig med login til vores LoginServlet, så vi kommer ind i nedenstående if-statement.

```
if ("login".equals(action)) {  
    String username = request.getParameter("username");  
    String password = request.getParameter("password");  
    User user = dm.validateLogin(username, password);  
    if (user == null) {  
        sendLoginForm(request, response);  
        return;  
    }  
    request.getSession().setAttribute("cart", new Cart());  
    request.getSession().setAttribute("bottoms", pm.getBottoms());  
    request.getSession().setAttribute("toppings", pm.getToppings());  
  
    sendFrontPage(request, response);  
    return;  
}
```



Her spørger vores login-controller om det tastede username og password, for at vi kan benytte os af vores metode `validateLogin()`.

Det der sker i vores `validateLogin`-metode er, at brugeren bliver hentet ud fra databasen ved brug af parametren 'username', hvis brugeren det som bliver hentet fra databasen er baseret på det username man har tastet er null, får vi blot null tilbage. Men hvis der rent faktisk er noget i det user-objekt, sammenlignes det tastede password med det som står i databasen og passer de bliver der returneret et User-objekt.

Hvis brugeren succesfuldt bliver valideret, sætte vi hermed et nyt Cart-objekt fast på session samt en ArrayList af toppings og bottoms.

Derimod, hvis man taster noget som ikke passer til det som står i vores databasen vil man blive redirected tilbage til samme `index.jsp` som fungerer som vores view for login, og ikke videresendt til `Shop.jsp`

### Registrer ny bruger

Hvis kunden ikke allerede har en bruger i vores system opretter de en bruger med et brugernavn, e-mail og password på `Signup.jsp`.

```
<input type="hidden" name="action" value="register">  
<input type="submit" value="Sign up">
```

Derefter laver vi en action på register til vores `RegisterServlet`:

```
if ("register".equals(action)) {
    String username = request.getParameter("registerUsername");
    String password = request.getParameter("registerPassword");
    boolean admin = Boolean.parseBoolean(request.getParameter("admin"));
    int userId = Integer.parseInt(request.getParameter("registeruserId"));
    int balance = Integer.parseInt(request.getParameter("balance"));
    String email = request.getParameter("registerEmail");

    dm.registerUser(username, password, admin, userId, balance, email);
    user = dm.getUser(username);
    request.getSession().setAttribute("user", user);
    sendFrontPage(request, response);
    if (user == null) {
        //registration fail
        sendRegisterForm(request, response);
    } else {
        sendFrontPage(request, response);
    }
}
```

Når register bliver kaldet, sætter vi vores textbox værdier ind fra signup.jsp med `request.getParmeter`, derefter bruger vi værdierne i metoden `registerUser`, som er inde i vores `dm(UserMapper)`.

```
public void registerUser(String username, String password, boolean admin, int userId, int balance, String email) {
    PreparedStatement newUser = null;
    String SQLString = "insert into cupcake.user (username, password, userId, balance, email) values (?, ?, ?, ?, ?)";
    try {
        con.setAutoCommit(false);
        newUser = con.prepareStatement(SQLString);
        newUser.setString(1, username);
        newUser.setString(2, password);
        newUser.setInt(3, userId);
        newUser.setInt(4, balance);
        newUser.setString(5, email);
        newUser.executeUpdate();

        con.setAutoCommit(true);
    } catch (SQLException e) {
        System.out.println("Fail in DataMapper - registerUser");
        System.out.println(e.getMessage());
    }
}
```

Inde i vores `registerUser` i `UserMapper`, tager vi de inputs fra `signup.jsp` og sætter dem ind i databasen.

## Shop og cart

Efter at have være logget ind bliver kunden sendt videre til shop siden, hvor der nu skal tastes og vælges en kombination af cupcakebund og topping samt hvor mange af denne kombination der ønskes.

Efter kunden har tastet den ønskede kombination af top, bund og quantity klikkes der på 'Add to order'-knappen.

Den sender en action "addProduct" til ProductServlet, som står for at styre alt

```
56     cart.addOrderline(user.getUserId(), bottomId, bottomPrice, toppingId, toppingPrice, quantity, totalPrice);  
57     request.getSession().setAttribute("cart", cart);  
58     RequestDispatcher rd = request.getRequestDispatcher("shop.jsp");  
59     rd.forward(request, response);  
60  
61     break;
```

shop-relaterede kald.

Den vil så køre ovenstående kode, som vil tilføje en ny Orderline til vores Cart-objekt ved brug af addOrderline-metoden.

Når en ny orderline er tilføjet til vores Cart-objekt i sessionen redirecter vi kunden tilbage til shop.jsp, hvor den nu har listet indholdet af Cart-objektet ud i vores view. Dette sker indtil, at kunden vil gå videre til næste step som er at afslutte ordren, og dermed få sin invoice.

Denne metode skal have en række af parametre med, og dette kunne have været gjort simplere og mindre redundant i databasen når vi derefter ligger ned i vores odetails-table. Vi kunne have undgået, at have listet alle elementerne ud fra bottom ved blot, at gøre så metoden addOrderline tog et Topping-objekt, et Bottom-objekt, en quantity og en totalPrice. Det havde gjort, så der ville være mindre redundant data som bliver stored i vores database senere hen.

Grunden til, at vi gik på kompromis med denne løsning var for at bedre kunne forstå hvilke elementer som metoden skulle have. Dog, hvis vi skulle lave et lignende system en anden gang ville vi benytte den forklarede fremgangsmåde oven over.

## Invoice

Når kunden er færdig med at tilføje til ordren, skal de derefter trykke på knappen 'Finalize order', Den sender en action "Finalize" til ProductServlet:

```
case "Finalize":  
  
    Order order = new Order(user.getUserId(), cart.getLines());  
    try {  
        om.addOrder(order);
```

Når ProductServlet bliver kaldet laver den et nyt Order object, som indeholder vores userId, og det cart som brugeren har oprettet med AddOrderline(cart.getLines()).

```
public class Order {  
    private int userId;  
    private int orderId;  
    private List<Orderline> orderlines = new ArrayList();  
  
    public Order(int userId, List<Orderline> orderlines) {  
        this.orderlines = orderlines;  
        this.userId = userId;  
    }  
}
```

Derefter i vores ProductServlet kalder vi "om", der er vores OrderMapper, som har en metode der hedder addOrder. addOrder henter vores userId fra Order klassen, som bliver sat på sessionen.

```
public int addOrder(Order order) throws SQLException {
    PreparedStatement addUserID = null;
    String sql = "insert into orderl (User_userId) values (?);";
    try {
        PreparedStatement statement = con.prepareStatement(sql,
            Statement.RETURN_GENERATED_KEYS);
        con.setAutoCommit(false);
        statement.setInt(1, order.getUserId());
        int affectedRows = statement.executeUpdate();
        // con.setAutoCommit(true);
    }
```

Derefter autogenerere addOrder orderId.

```
43         try (ResultSet generatedKeys = statement.getGeneratedKeys()) {
44             if (generatedKeys.next()) {
45                 order.setOrderId(generatedKeys.getInt(1));
46                 addOdetails(order);
            }
```

Efterfølgende i linje 45 kalder addOrder, addOdetails, som sætter vores Odetails ned i databasen.

```
public void addOdetails(Order order) {
    PreparedStatement orderl = null;
    String SQLString = "insert into odetails (Order_orderId, totalPrice, quantity, "
        + "bottom_bottomId, topping_toppingId) values (?, ?, ?, ?, ?)";
    try {
        // con.setAutoCommit(false);
        orderl = con.prepareStatement(SQLString);
        orderl.setInt(1, order.getOrderId());
        for (int i = 0; i < order.getOrderlines().size(); i++) {
            orderl.setInt(2, order.getOrderlines().get(i).getTotalPrice());
            orderl.setInt(3, order.getOrderlines().get(i).getQuantity());
            orderl.setInt(4, order.getOrderlines().get(i).getBottomId());
            orderl.setInt(5, order.getOrderlines().get(i).getToppingId());
            orderl.executeUpdate();
        }
    }
```

## Status på implementation

Overall har det været et lærerigt projekt, hvor implementationen af vores CRUD-metoder og funktioner på hjemmeside er gået udemærket. Der er nogle enkelte metoder, hvor vi har givet den for mange parametre i stedet for et objekt som indeholder de samme informationer.

Derudover har vi nået implementationen af alle de CRUD-metoder, som er nødvendige for vores projekt på nuværende plan.

Vi har i vores navigationsdiagram planlagt, at lave en JSP-side som hedder MyOrders.jsp, hvor planen med denne side var at få printet alle kundens ordrer ud på side og gøre det muligt for dem at se deres købshistorik. Måden man ville implementere denne feature på var ved at lave en CRUD-metode i vores OrderMapper, som gør det muligt at hente en ordre ud af databasen baseret på et userid.

Vores JSP-sider er blevet stilet efter nogle meget simple krav, da vi prioriterede vores backend over frontend-delen af vores projekt. Dog har vi både formået, at benytte os af et external stylesheet og meget simpelt bootstrap, for at få den helt rå HTML til at se lidt bedre ud.

I forhold til test har vi på nuværende ingen JUnit tests, men derimod har vi testet hver enkelte metode i nogle main-metoder i de enkelte klasser for at se om de kunne få fat i det efterspurgt fra databasen. Det er dog primært vores CRUD-metoder, som er blevet testet sådan.