|  |  |
| --- | --- |
| **Project – External Documentation** |  |
| COMP6114001  Pattern Software Design |
| Even Semester Year 2022/2023 |

* Project Title

RAAMEN

* Introduction

**RAAMEN** is a web-based online ramen selling application. It is a great place for buyer to buy ramen and for seller to sell their shows. The CEO of **RAAMEN** wants to hire your team to build this application. Before working there, you are given an assignment to make a simple version of this application.

* Report / Documentation

// loginBtn\_Click()

…

string errorTemp = loginControl.validateBtn(username, password);

if(string.IsNullOrEmpty(errorTemp) ) { // authentication successful

// get roles from repository

string roleTemp = loginRepository.getRoles(username);

Session["user"] = roleTemp; // hold roleTemp in session

// get userId from username given and save to Session[]

int userId = loginRepository.getId(username);

Session["userId"] = userId;

// if rememberMe selected, set cookies

if(rememberMe) {

HttpCookie cookie = new HttpCookie("roleCookie");

cookie.Value = roleTemp;

cookie.Expires = DateTime.Now.AddHours(2); // set expiry

Response.Cookies.Add(cookie);

HttpCookie userIdCookie = new HttpCookie("userIdCookie");

userIdCookie.Value = userId.ToString();

userIdCookie.Expires = DateTime.Now.AddHours(2);

Response.Cookies.Add(userIdCookie);

}

In this part of login.aspx.cs on loginBtn\_Click(), I designed my code so that it virst validates the account with the db. Continued with getting the role name of the given Username as well as the userID from the db. I then use Session[“user”] to hold the role name and Session[“userID”] to hold the userID as a Session[] object, which then can be used for other purposes. I also set a cookie system to allow the ‘remember me’ functionality work properly. In this case, I set 2 cookies to hold the role name and the userID.

// redirect for each roles

if(roleTemp.Equals("Staff")) {

Response.Redirect("~/View/Staff/staffHome.aspx");

}

else if(roleTemp.Equals("Member")) {

Response.Redirect("~/View/Member/memberHome.aspx");

}

else if(roleTemp.Equals("Admin")) {

Response.Redirect("~/View/Admin/adminHome.aspx");

}

else {

Response.Redirect("login.aspx");

}

This part of the code is followed by the Session[] and Cookies setup. This will check the roleTemp for the user’s role, and redirect them to their designated role pages.

if(Session["user"] == null && Request.Cookies["roleCookie"] == null) {

Debug.WriteLine("not valid");

Response.Redirect("~/View/login.aspx");

}

else if(!"Staff".Equals(Session["user"]?.ToString()) && !"Staff".Equals(Request.Cookies["roleCookie"]?.Value)) {

// use null-conditional operator to access value safely

// reset cookie and session

HttpCookie cookie = new HttpCookie("roleCookie");

cookie.Expires = DateTime.Now.AddDays(-1);

Response.Cookies.Add(cookie);

Session["user"] = null;

HttpCookie userIdCookie = new HttpCookie("userIdCookie");

userIdCookie.Expires = DateTime.Now.AddDays(-1);

Response.Cookies.Add(userIdCookie);

Session["userId"] = null;

Response.Redirect("~/View/login.aspx");

}

This part of the code is placed inside every page’s Page\_Load(). I designed this code specifically to validate the user role from their Session[] and Cookies. This is to prevent any security breaches, should the user purposely access other roles. This can be done by checking if Session[“user”] is empty and Cookies[“roleCookie”] is empty, then it will redirect the user to the login page.

Should the Session[“user”] and Cookies[“roleCookie”] does not belong to the page, it will reset the Session[] and Cookie[], then redirect the user to the login page. Otherwise, the user will be able to navigate through their designated pages.

1. a

protected void regisBtn\_Click(object sender, EventArgs e) {

string username = userIn.Text;

string email = emailIn.Text;

string gender = radioGender.SelectedValue;

string password = passIn.Text;

string passConfirm = passInConfirm.Text;

string selectedRole = radioRole.SelectedValue;

switch(regisControl.validateBtn(username, email, gender, password, passConfirm, selectedRole)) { // call controller for validation

case 0:

errorLbl.Text = "Username cannot be empty";

break;

case 1:

errorLbl.Text = "Username must be between 5 and 15 characters with only alphabets";

break;

case 2:

errorLbl.Text = "Email cannot be empty";

break;

case 3:

errorLbl.Text = "Email must end with .com";

break;

case 4:

errorLbl.Text = "Gender cannot be empty";

break;

case 5:

errorLbl.Text = "Password cannot be empty";

break;

case 6:

errorLbl.Text = "Password Confirmantion cannot be empty";

break;

case 7:

errorLbl.Text = "Password does not match with confirmed password";

break;

case 8:

errorLbl.Text = "Role cannot be empty";

break;

case 9:

errorLbl.Text = "Successfulyl registered";

break;

default:

errorLbl.Text = "Something went wrong";

break;

}

This is a basic validation algorithm. In this case, this is a registration validation. I use a switch case that takes the returned value from a controller.

public static int validateBtn(string username, string email, string gender, string password, string passConfirm, string selectedRole) {

if(username.Equals("")) // username empty

return 0;

if(username.Length < 5 || username.Length > 15 || !Regex.IsMatch(username, "^[a-zA-Z]+$")) // username between 5 - 15, only alphabets

return 1;

if(email.Equals(""))

return 2;

if(!email.EndsWith(".com")) // email ends with .com

return 3;

if(gender.Equals("")) // gender empty

return 4;

if(password.Equals("")) //password empty

return 5;

if(passConfirm.Equals("")) // passConfirm empty

return 6;

if(!(password.Equals(passConfirm))) // passwords does not match

return 7;

if(selectedRole.Equals("")) // radio empty

return 8;

else {

// open access from repository to handler to add to db

registRepository.addUser(username, email, gender, password, selectedRole);

return 9;

}

}

The controller will take in the required values to be validated, in this case, registration will validate username, email, gender, password, passConfirm, and selectedRole. Should there be an error, the method will then return numbers, and this number will then be used as a sort of ‘code’. This is followd with calling the repository and do further processing. I implement this type of scenarios for all input-validate scenarios, such as updateRamen, updateProfile, etc.

public class updateRamenRepository {

public static localDBEntities db = new localDBEntities();

public static List<RamenDetail> getById(int id) {

return (from x in db.RamenDetails where x.ID == id select x).ToList();

}

public static void updateRamen(string name, string meat, string broth, string price, int id) {

// call handler to update, parse ramenDetail

updateRamenHandler.updateRamen(name, meat, broth, price, db, id);

}

}

This is a repository file. Throughout the project, I designed the data flow so that only a repository have exclusive access to the db. From the snippet above, I designed it so that the class opens an access to the db, get a list of RamenDetail by id, and update the ramen data. For the update ramen method, I call a handler to specifically undergo the business logic such as insert, update, and delete. In this case, I parse through the localDBEntities db, so that the handler is given access to the db.

public class updateRamenHandler {

public static void updateRamen(string name, string meat, string broth, string price, localDBEntities db, int id) {

RamenDetail ramenDetail = (from x in db.RamenDetails where x.ID == id select x).FirstOrDefault();

ramenDetail.Name = name;

// check selected meat

if(meat.Equals("Pork"))

ramenDetail.meatID = 1;

else if(meat.Equals("Beef"))

ramenDetail.meatID = 2;

else if(meat.Equals("Chicken"))

ramenDetail.meatID = 3;

ramenDetail.Broth = broth;

ramenDetail.Price = price;

db.SaveChanges();

}

}

This is a handler for updating ramen. the updateRamen() method will require the variables to be updated, along with a db access.

This scenario can be implemented to our project for other methods such as updateProfile(), insertRamen(), manageRamen(), etc.

With this way, we can ensure that only the repository have exclusive access for our db.

public class regisFactory {

public static User newUser(string username, string email, string gender, string password, int role) {

User newUser = new User();

newUser.Username = username;

newUser.Email = email;

newUser.Gender = gender;

newUser.Password = password;

newUser.roleID = role;

return newUser;

}

}

This is a factory class. A factory is used for constructing an object. In this case, this is a register factory class. The newUser() method returns a User data type. This method is called from registration handler.

public static void addUser(string username, string email, string gender, string password, string selectedRole, localDBEntities db) {

// check role selected

if(selectedRole.Equals("Staff")) {

User newUser = regisFactory.newUser(username, email, gender, password, 1); // call factory, set as Staff role

db.Users.Add(newUser);

db.SaveChanges();

}

else if(selectedRole.Equals("Member")) {

User newUser = regisFactory.newUser(username, email, gender, password, 2); // call factory, set as Member role

db.Users.Add(newUser);

db.SaveChanges();

}

}

The handler sets up a User data type variable, and calls our factory method. We then add the newUser variable to the db.

protected void addToCartBtn\_Click(object sender, EventArgs e) {

Button addToCartBtn = (Button)sender;

GridViewRow row = (GridViewRow)addToCartBtn.NamingContainer; // find the row containing the button

string id = addToCartBtn.CommandArgument; // hold the button command argument value

string qty = ((TextBox)row.FindControl("qtyIn")).Text; // get textbox value from the row

addToCart(id, qty);

// after add to cart, data bind again

gvCart.DataSource = Session["cart"];

gvCart.DataBind();

// set buttons to be visible after doing logic

orderBtn.Visible = true;

clearBtn.Visible = true;

}

This is a button inside orderRamen page. This addToCartBtn is responsible to add ramens into a cart system. It first retrieve the button object, find the row that contains the clicked button, gets the id from CommandArgument of the button, and gets the qty from the TextBox.Text. This is then followed with addToCart() method.

protected void addToCart(string id, string qtyIn) {

// use session for cart system, call cartFactory class

List<cartViewModel> cart = Session["cart"] as List<cartViewModel>;

// check if cart empty, make new List

if(cart == null)

cart = new List<cartViewModel>();

cart.Add(cartFactory.newItem(id, qtyIn));

Session["cart"] = cart;

}

This method is used to create a cart. We first create a cart variable with List<cartViewModel> data type as a Session[“cart”]. We first check if the cart is empty, we create a new empty List<>. Otherwise, we add the ramen that the user wants to order. In this case, it will pass through the ramenID and the quantity. We then set our Session[“cart”] with our updated cart List. With this way, we can ensure that our cart system stays solid throughout the session and stored in a secure place. We also use a view model class, which contains attributes that we want inside the cart List.

protected void orderBtn\_Click(object sender, EventArgs e) {

List<cartViewModel> cart = Session["cart"] as List<cartViewModel>;

int userId = getUserId();

orderRamenRepository.addRamen(cart, userId);

errorLbl.Text = "Successfuly created order";

clearCart();

}

This is a button that is responsible for ordering the ramen from the cart. We first hold the cart variable from the Session[“cart”]. We then call another method that retrieves the userID from either the Cookies[“userIdCookie”] or Session[“userId”]. With this, we can call the repository and add the ramen from the cart list.

public static void addRamen(List<cartViewModel> cart, int userId) {

// extract each values, then call handler

// first, add into header so that we can hold the headerID

// call handler to insert into Header table and storing the most recent id

int headerId = orderRamenHandler.insertHeader(db, userId);

// insert new cart using headerID, so that cart is only created once, not repetetive

orderRamenHandler.insertCart(db, headerId);

foreach (cartViewModel cartItem in cart) {

int ramenId = cartItem.ramenID;

int quantity = cartItem.Quantity;

// call handler to insert into detail and cart table

orderRamenHandler.insertRamen(db, headerId, quantity, ramenId);

}

}

This is an addRamen() method inside a repository. This method will retrieve the cart List and the userID. We first call the handler to insert the Header table inside the db to make sure that the userID that ordered is saved and get the headerID at the same time. Followed with inserting the Cart table inside the db. Then we open a foreach loop that iterates every item inside the cart List. Inside this loop, we get the ramenID and the quantity from each of the cart List for every iteration and pass it to the handler to insert ramen.

This is the insertRamen() method inside the repository. This method will require the db access, headerId, quantity, and the ramenId. We call a factory to construct our object, and add the addDetail object inside the db. Through this, we have successfully order a ramen and saved the order to the db.

public static void insertRamen(localDBEntities db, int headerId, int quantity, int ramenId) {

// call factory

Detail addDetail = orderRamenFactory.newDetail(headerId, quantity, ramenId);

db.Details.Add(addDetail);

// mainly for debugging

try {

// Code to add and save the new entry

db.SaveChanges();

}

catch(DbUpdateException ex) {

Exception innerException = ex.InnerException;

string errorMessage = innerException.Message;

// Handle or log the error message as needed

Debug.WriteLine(errorMessage);

}

}

// inside Page\_Load()

...

gvOrders.DataSource = orderQueueRepository.getData();

gvOrders.DataBind();

This is the lines that is responsible for pages that contains a gridview. In this case, this lines is responsible to set the datasource and bind the data for a gridview inside the orderQueue page. A gridview will require a List type data. In this case, we call a repository to retrieve the data.

This is the getData() method inside the repository. This method will return a List type data from the db. In this case, we are using LINQ to access the db tables. Here, we are joining Header table, Detail table, and Cart table on Header.ID = Header.headerID and Header.ID = Cart.headerID. However, we are returning a headerDetailCartViewModel object.

public class headerDetailCartViewModel {

// from Header

public int ID { get; set; }

public int customerID { get; set; }

public Nullable<int> staffID { get; set; }

public System.DateTime Date { get; set; }

// from Detail

public int headerID { get; set; }

public int ramenID { get; set; }

public int Quantity { get; set; }

public virtual Header Header { get; set; }

public virtual RamenDetail RamenDetail { get; set; }

//cart's headerID not necessary because already exist from Detail

public string orderStatus { get; set; }

}

public static List<headerDetailCartViewModel> getData() {

return (from x in db.Headers join y in db.Details on x.ID equals y.headerID join z in db.Carts on x.ID equals z.headerID select new headerDetailCartViewModel {

ID = x.ID,

customerID = x.customerID,

staffID = x.staffID,

Date = x.Date,

headerID = y.headerID,

ramenID = y.ramenID,

Quantity = y.Quantity,

orderStatus = z.orderStatus

}).ToList();

}

We set each values of the headerDetailCartViewModel attributes accordingly to the data from the db. We then make sure that the data that we are returning is set as a List data type by using .ToList(). This way, we can retrieve datas from the db to use it on a gridview.

// inside Page\_Load()

...

// call repository to connect to db

localDBEntities db = reportRepository.getDB();

crystalReport cr = new crystalReport();

List<Header> headers = db.Headers.ToList();

List<User> users = db.Users.ToList();

List<Detail> details = db.Details.ToList();

List<RamenDetail> ramens = db.RamenDetails.ToList();

dataSet ds = createDataset(headers, details, users, ramens);

decimal grandTotal = reportRepository.calculateTotal();

Debug.WriteLine(grandTotal);

TextObject textField = cr.ReportDefinition.ReportObjects["textField"] as TextObject; // access textField text object from crystal report

textField.Text = "Grand Total = " + grandTotal.ToString(); // set grand total

cr.SetDataSource(ds);

CrystalReportViewer.ReportSource = cr;

This are some lines that is responsible for our crystal report in the report page. We first call our repository to gain access to the db. We then retrieve the Header, User, Detail, and RamenDetail table and keep it as List data. Afterwards, we set up our dataset by parsing our Header, Detail, User, and RamenDetail variables. After setting up our dataset, we can calculate our grand total and set it inside a TextObject from the crystal report designer. We then set our crystal report data source as our generated dataset and bind the data together. With this, we have created our crystal report the reflects our db and all the transaction done.

This method is responsible for generating our dataset. We first retrieve all the parameters and set them aside as a var. We then set up our User row by using a foreach loop. We then set our Header row the same way as our User row, however we are also nesting another foreach loop that sets the detail row. This is because for every Header, there may be more than 1 Detail, which is why we need to check the Detail for each Header. The same goes to the RamenDetail row, for every Detail, there may be more than 1 RamenDetail. We also set a filter detailList that ensures for each Header, the Details selected are corresponding to the Header.ID (Detail.headerID == Header.ID). The same goes to the RamenDetail, where RamenDetail.ID is corresponding to the Detail.ramenID.

private dataSet createDataset(List<Header> headers, List<Detail> details, List<User> users, List<RamenDetail> ramenDetails) {

dataSet ds = new dataSet();

var header = ds.Header;

var detail = ds.Detail;

var user = ds.User;

user.PrimaryKey = new DataColumn[] { user.Columns["ID"] }; // set so that ID is seen as primary key, dont know why

var ramen = ds.Ramen;

foreach(User u in users) {

var urow = user.NewRow();

urow["ID"] = u.ID;

urow["roleID"] = u.roleID;

urow["Username"] = u.Username;

urow["Email"] = u.Email;

urow["Gender"] = u.Gender;

urow["Password"] = u.Password;

user.Rows.Add(urow);

}

foreach(Header th in headers) {

var hrow = header.NewRow();

hrow["ID"] = th.ID;

hrow["customerID"] = th.customerID;

hrow["staffID"] = th.staffID;

hrow["Date"] = th.Date;

header.Rows.Add(hrow);

var detailList = details.Where(td => td.headerID == th.ID).ToList();

foreach(Detail td in detailList) {

var drow = detail.NewRow();

drow["headerID"] = td.headerID;

drow["ramenID"] = td.ramenID;

drow["Quantity"] = td.Quantity;

detail.Rows.Add(drow);

var rd = ramenDetails.FirstOrDefault(r => r.ID == td.ramenID);

if(rd != null) {

var rrow = ramen.NewRow();

rrow["ID"] = rd.ID;

rrow["meatID"] = rd.meatID;

rrow["Name"] = rd.Name;

rrow["Broth"] = rd.Broth;

rrow["Price"] = rd.Price;

ramen.Rows.Add(rrow);

}

}

}

return ds;

}

// inside manageRamenHandler.deleteRamen()

...

var detailRemove = (from x in db.Details where x.ramenID == ramenTemp.ID select x).FirstOrDefault();

if(detailRemove != null)

.Details.Remove(detailRemove);

db.RamenDetails.Remove(ramenTemp);

db.SaveChanges();

return "Delete Successful";

Because RamenDetail table is referenced from Detail table, when deleting a ramen will cause foreign key problems. This can be done by removing the associated ramenId from the Detail table, then we can delete the ramen.

* Reference
* PSD Lab Session 3
* PSD Lab Session 5
* PSD Lab Session 6
* PSD Lab Session 7
* PSD Lab Session 8
* PSD Lab Session 10
* PSD Lab Session 11
* <https://help.sap.com/doc/37c8938b12384f399940e22b7315979f/2020/en-US/cr20_usergde_en.pdf>
* <https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-7.0>
* Group Member
* 2540124122 – Justin Thejasukmana