SIT774 Web Technologies and Development

Workshop Week 9 - Node.js 2

Instructor : Ran Zhang
Unit chair : Michael Hobbs
Instructor: Ran Zhang & Ruyi Li

DEAKIN UNIVERSITY

Week 9: NodeJS (II)

Key content

Server-side programming with Node.JS (9.1-9.4): Create a server with Node.js to accept the Form data from the client and discover how to use SQL to create a database.
Wrap up (9.5): Reflect on how the Form data is processed on the server side and on the basics of database and SQL.

9.1 Welcome to week 9
Web Page

9.2 Processing Forms on the server with NodeJS
Web Page

9.3 Database and SQLite Primer
Web Page

9.4 Database access on server-side VIDEO
Web Page

9.5 Weekly wrap up
Web Page

Ontrack 9.1/9.2

Get/Post Methods

DEAKIN UNIVERSITY

9.2 Processing Forms on the server with NodeJS

This example starts the Express server and listens for incoming requests on port 3000. When the server starts, it logs a message to the console indicating that it is running.

1. Let's start off with the basic NodeJS template:

```
let express = require('express')
let app = express()

//now any files in public are routed
app.use(express.static('public'))

//Here we are configuring express to use body-parser as middle-ware.
app.use(express.urlencoded({ extended: false }));
```

This imports the Express library and initializes an Express application instance.

This tells Express to serve static files (like HTML, CSS, JavaScript, and images) from the public directory. Any files placed in this directory will be accessible via HTTP requests to the server.

This middleware parses URL-encoded data (typically from HTML forms) and makes it available under req.body in route handlers. The extended: false option means that the query string library querystring is used for parsing.

2. We need to actually start our server:

```
app.listen(3000, function () {
    console.log('Example app listening on port 3000!')
})
```

app.listen tells Node to start a server on port 3000!

app.listen tells Node to start a server on port 3000!

3. We need to be able to tell our NodeJS code that we are expecting some data to come from a form!

```
// REST endpoint for posting a new user
app.post('/users', function (req, res, next) {
    let username = req.body.name;
    let password = req.body.password;
    let comment = req.body.comment;

    console.log("Just received POST data for users endpoint!");
    console.log(`Data includes: ${username}, ${password} and ${comment}`);
});
```

This defines a route that handles POST requests to /users.

When a POST request is made to this endpoint, the server extracts the name, password, and comment fields from the request body and logs them to the console.

DEAKIN UNIVERSITY

Final code

The final code should look something like this:

```
let express = require('express')
let app = express()

//now any files in public are routed
app.use(express.static('public'))

//Here we are configuring express to use body-parser as middle-ware.
app.use(express.urlencoded({ extended: false }));

// REST endpoint for posting a new user
app.post('/users', function (req, res, next) {
    let username = req.body.name;
    let password = req.body.password;
    let comment = req.body.comment;

    console.log("Just received POST data for users endpoint!");
    console.log(`Data includes: ${username}, ${password} and ${comment}`);
});

app.listen(3000, function () {
    console.log('Example app listening on port 3000!')
})
```

This is a fully functioning server, all that is missing is a html front-end with a post form!

Don't forget to run npm install followed by node . (or equivalent commands could be node src/index.js, npm run start) after extracting the project, then you can visit localhost:3000 in your browser and test it out!

DEAKIN UNIVERSITY

9.3 Database and SQLite Primer

Database

A database is a shared, integrated electronic structure in the computer that stores a collection of data.

The commonly used or dominant way of storing data is to use tables, where data is stored as the table rows or records, and each record consists of fields or columns.

The following is a simple table structure that stores some sample student personal data (ie `Student_ID`, `Student_FName`, `Student_LName`, and `Student_Email`).

Student_ID	Student_FName	Student_LName	Student_Email
2017031001	James	Brown	jbrown@gmail.com
2016072008	Anne	Smith	as@deakin.edu.au
2015031097	Paul	Orlando	porlan@smart.com

A database that contains tables as well as other related objects is also called a **relational database**.

Tables are managed by a database engine such as SQLite or a database management system (DBMS) such as Oracle DBMS.

SQL is a standard language for accessing and manipulating databases.

What is SQL?

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

Using SQL in Your Web Site

To build a web site that shows data from a database, you will need:

- An RDBMS database program (i.e. MS Access, SQL Server, MySQL)
- To use a server-side scripting language, like PHP or ASP
- To use SQL to get the data you want
- To use HTML / CSS to style the page

Creating a table in a database

The SQL CREATE TABLE Statement

The `CREATE TABLE` statement is used to create a new table in a database.

Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

Tip: For an overview of the available data types, go to our complete Data Types Reference.

SQL CREATE TABLE Example

The following example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

Example

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

The PersonID column is of type int and will hold an integer.

The LastName, FirstName, Address, and City columns are of type varchar and will hold characters, and the maximum length for these fields is 255 characters.

The empty "Persons" table will now look like this:

PersonID	LastName	FirstName	Address	City

DEAKIN UNIVERSITY

Creating a table in a database example –unit site

For example, the following SQL command creates the above example table "Student" in a database:

```
CREATE TABLE Student (
    Student_ID VCHAR(10),
    Student_FName VCHAR(20),
    Student_LName VCHAR(20),
    Student_Email VCHAR(20),
    PRIMARY KEY (Student_ID)
);
```

Here `VCHAR` is a variable character string data type, the number after `VCHAR` (eg VCHAR(20)) is the maximum number (eg 20) of characters in the character string.

`PRIMARY KEY` indicates the primary key of the table, whose value can uniquely identify a row/record in the table. For example, `student_ID` is a primary key as its value can uniquely identify a student record in the table.

Inserting a row/record of data into a table can be implemented by using INSERT command of SQL. It's syntax is as follow:

```
INSERT INTO table_name VALUES (value1, value2, ...);
```

Here `value1` is for the first column of the table, `value2` is for the second column, and so on.

For example, the following SQL command inserts a student record into the table "Student":

```
INSERT INTO Student VALUES (
    '2017031001', 'James', 'Brown', 'jbrown@gmail.com'
);
```

where character string data should be quoted in the command.

Retrieving data or information from tables

Retrieving data or information from one or more tables can be implemented by using SELECT command of SQL.

The simplest syntax of SELECT command is as follow.

```
SELECT column1, column2, ...
FROM table_name;
```

Here, column1, column2, ... are the column names of the table you want to select data from. If you want to select all the columns available in the table, use the following syntax:

```
SELECT * FROM table_name;
```

For example, if you want to retrieve all column values for all records/rows in the table 'Student", you can use the command:

```
SELECT * FROM Student;
```

If we want to retrieve only those records that fulfill a specified condition, we can use the WHERE clause. The syntax is:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

For example, if we want only those records with the first name "John" from the table "Student", the SQL command is:

```
SELECT * FROM Student
WHERE Student_FName = 'John';
```

The SQL SELECT Statement

The `SELECT` statement is used to select data from a database.

The data returned is stored in a result table, called the result-set.

SELECT Syntax

```
SELECT column1, column2, ...
FROM table_name;
```

Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax:

```
SELECT * FROM table_name;
```

The SQL WHERE Clause

The `WHERE` clause is used to filter records.

It is used to extract only those records that fulfill a specified condition.

WHERE Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

SQlite

SQLite

SQLite is a self-contained, serverless, zero-configuration, transactional SQL database engine.

Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file.

The SQLite database file format is cross-platform, so you can freely copy a database between 32-bit and 64-bit systems. The most important thing is that SQLite supports all SQL commands. These features make SQLite a popular choice as an application file format.

Installing the SQLite3 module

Node.js has a module SQLite3 that supports SQLite database engine. This module provides almost all connection/query from SQLite3.

To use SQLite3 in Node.js, we need to install it first.

You can use `npm` to download and install the SQLite3 module in the folder where your Node.js server is located. The command is:

```
npm install sqlite3
```

DEAKIN UNIVERSITY

9.4 Database access on server-side VIDEO

Database access on server-side

Watch this video to see how to set up an SQL database and save data to that database
SQLite3 is the database used in this video for:
1.setting up a table
2.saving the data from a Form to that table
3.retrieving the data from the table

Summary:

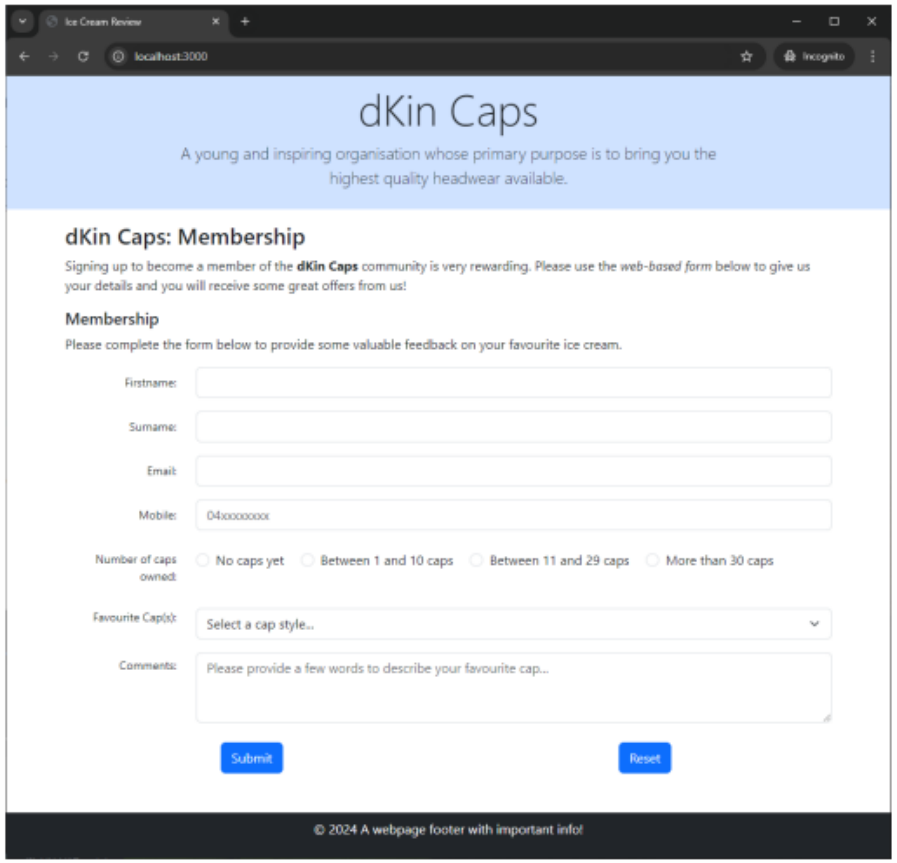
- This week we mainly focused on creating a server with Node.js to accept the Form data from the client (web browser). The programming techniques introduced via an application example are necessary to the further data processing on the server, such as saving data into a database and retrieving data from the server.
- We briefly introduced basics of database and SQL. SQL is a standard language used for data manipulations in relational databases

https://video.deakin.edu.au/media/t/0_9isi0xvg

DEAKIN UNIVERSITY

Ontrack task 9.1P: Send a Response to the Received Data

In previous weeks we have looked at how we can handle `GET` requests in an express server. Here, we will look at how to handle `POST` requests, as generated when submitting a web form. You are supplied with the code that presents the following web page form for accepting some information for a user's membership registration, including their name, an email address, phone number, their favourite cap style and a comment:



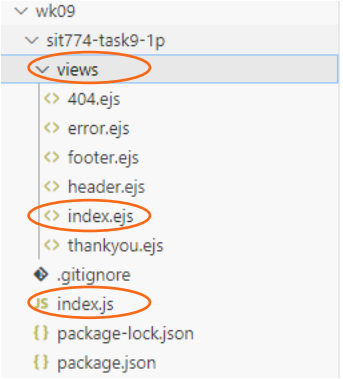
Task9.1.1 Example form layout

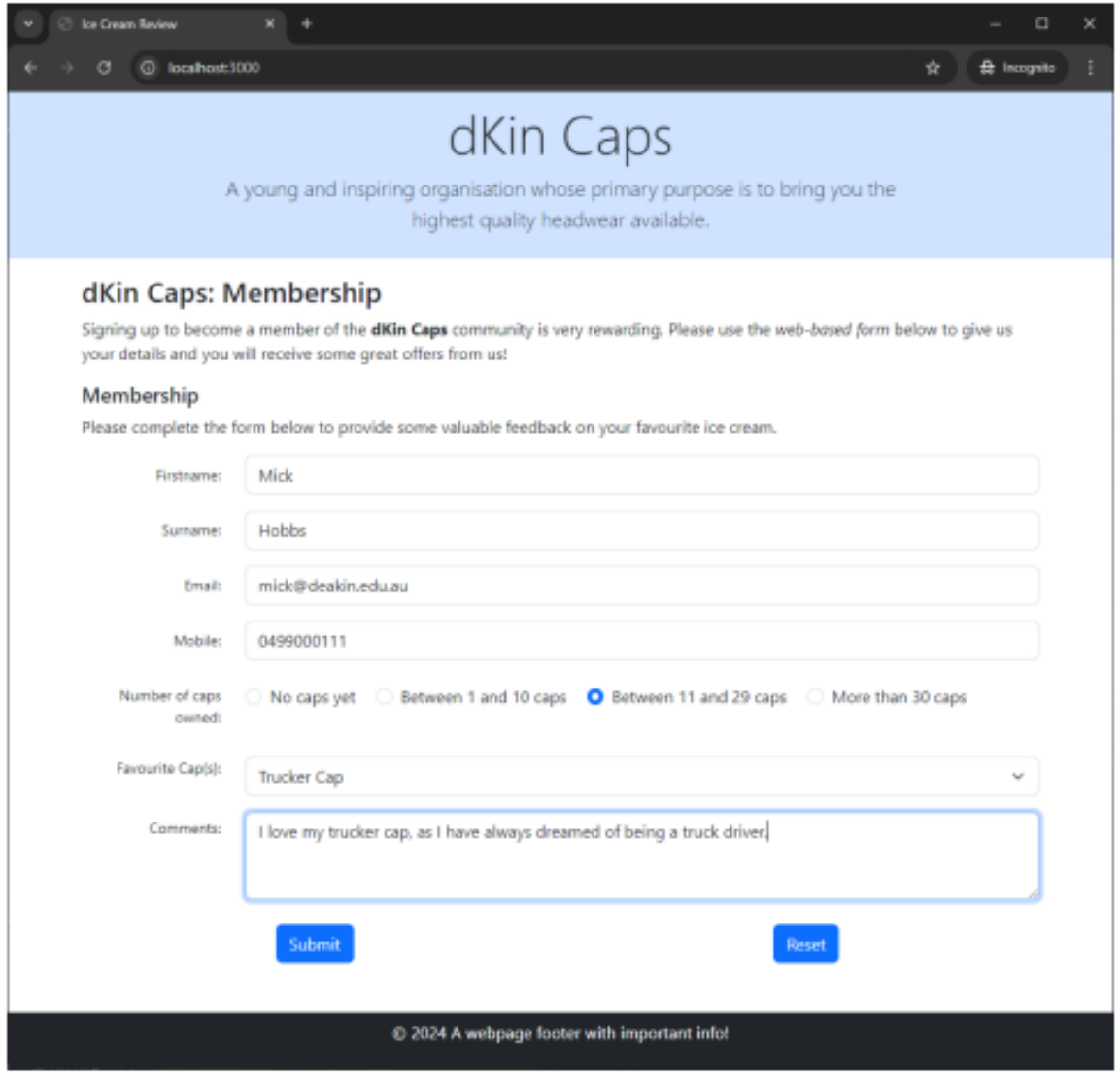
In this task, your web page should use Encapsulated JS templates for the main form page and for the thank you response page. The template files for the main form page (broken up into header.ejs , index.ejs , footer.ejs sections) is provided on the Ontrack website, in the task 9.1P page through the Resources link found in the bottom right-hand corner of the page. You are required to do the following things:

..		
404.ejs	668	326
error.ejs	555	260
footer.ejs	486	332
header.ejs	511	342
index.ejs	7,089	1,331

Requirements

1. Save the provided EJS template files into a new `views` folder in your project directory (following the example provided in the Week 08 content page 8.10 Using templates).
2. Make a *Node.js* application program (i.e., an `index.js` file) in your *Node.js* server folder. Add a `GET` handler for the default `/` route to render the `index.ejs` template with an appropriate `title` parameter.
3. Create a new `thankyou.ejs` template file (that can also *include* the `header.ejs` and `footer.ejs` templates to maintain consistency in presentation). The template should include fields that can be populated by arguments it is provided for the *name*, *favourite ice cream*, *rating* and *feedback*.
4. Add a new `POST` handler to accept data from the **form page**. The route address can be found in the form's `action` value. The request handler should:
   - extract the forms fields (from the *body* of the *request*)
   - render a `thankyou` template for the response page. The form's fields can be passed as parameters to the template to populate the *response* message.
5. Visit the form web page via the local *Node.js* server (e.g., `http://localhost:3000/`) using a web browser; which should render the template `index.ejs` file **NOT** a static file.
6. Enter some data into the form, like the following:

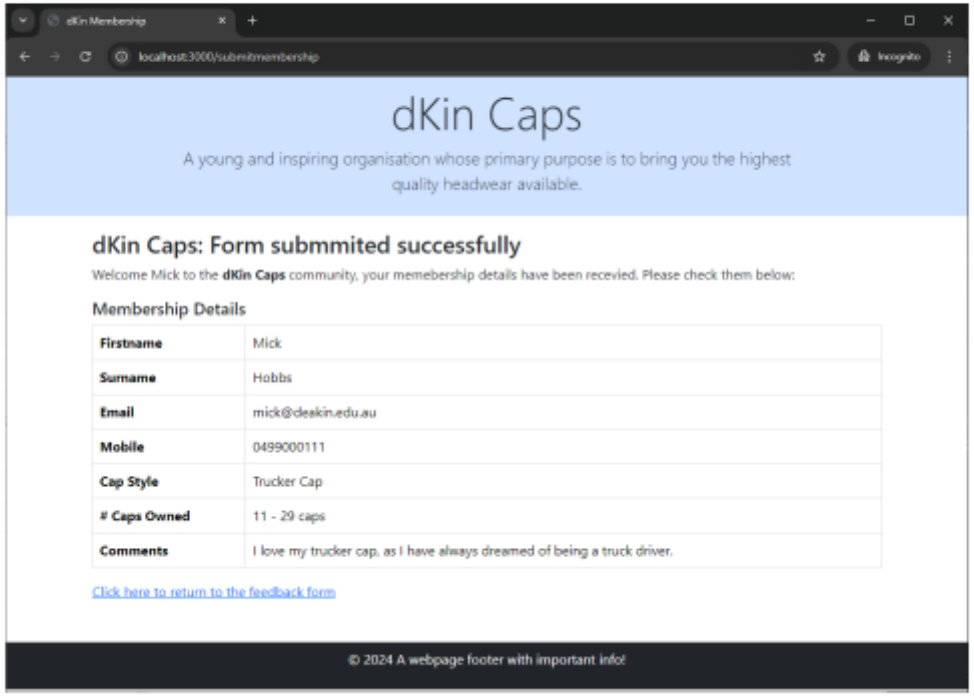




Task9.1.2 Example form with input

Submission

7. When the "**Submit**" button is clicked, then a response page (message) generated from the `thankyou.ejs` template and should be displayed in the browser, like the following:



Task9.1.3 Response from the server

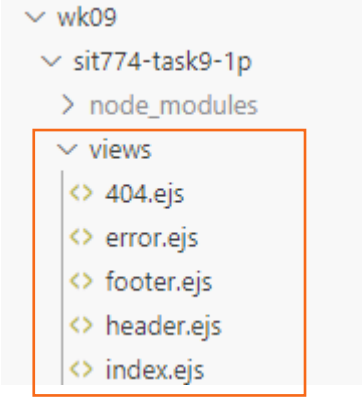
What will you submit?

You should submit:

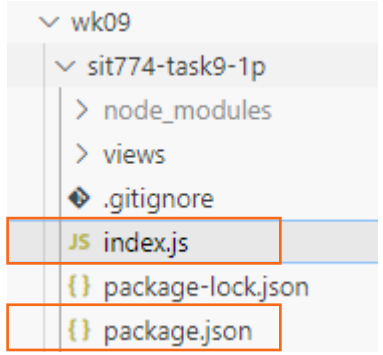
- Source code of the *Node.js* file (i.e., the `index.js` file).
- Source code of the *template* file for your **Thank You** page (i.e., the `thankyou.ejs` file).
- Screenshot of the browser window showing the form web page with entered data.
- Screenshot of the browser window showing response message after the "**Submit**" button is clicked.

Hints

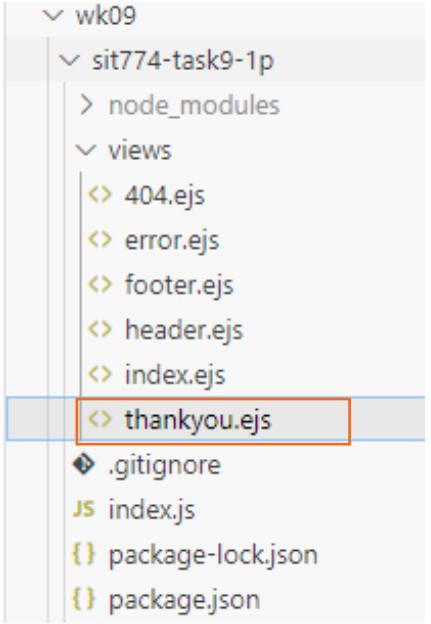
1. Save the provided EJS template files into a new `views` folder in your project directory (following the example provided in the Week 08 content page 8.10 Using templates).
2. Make a *Node.js* application program (i.e., an `index.js` file) in your *Node.js* server folder. Add a `GET` handler for the default `/` route to render the `index.ejs` template with an appropriate `title` parameter.
3. Create a new `thankyou.ejs` template file (that can also *include* the `header.ejs` and `footer.ejs` templates to maintain consistency in presentation). The template should include fields that can be populated by arguments it is provided for the *name*, *favourite ice cream*, *rating* and *feedback*.



Step1



Step2： Add a GET handler for the default / route to render the index.ejs template with an appropriate title parameter



Step3

wk09 > sit774-task9-1p > views > thankyou.ejs > ? > ?

```
<%- include('header'); -%>

<header>
  <div class="container-fluid bg-primary-subtle">
    <div class="col-sm-8 mx-auto text-center py-2">
      <h1 class="display-4">dKin Caps</h1>
      <p class="lead ">A young and inspiring organisation whose
        primary purpose is to bring you the highest quality headwear available.</p>
    </div>
  </div>
</header>

<main>
  <div class="container">
    <h3 class="mt-3">dKin Caps: Form submmited successfully</h3>

    <p>
      Welcome <%= firstname %> to the <strong>dKin Caps</strong> community, your memebership
        details have been recevied. Please check them below:
    </p>

    <h5>Membership Details</h5>

    <table class="table table-bordered">
      <tr>
        <th>Firstname</th>
        <td>
          <%=firstname %>
        </td>
      </tr>
      <tr>
        <th>Surname</th>
        <td>
```

Hints

4. Add a new `POST` handler to accept data from the **form page**. The route address can be found in the form's `action` value. The request handler should:
   - extract the forms fields (from the *body* of the *request*)
   - render a `thankyou` template for the response page. The form's fields can be passed as parameters to the template to populate the *response* message.
5. Visit the form web page via the local *Node.js* server (e.g., `http://localhost:3000/`) using a web browser; which should render the template `index.ejs` file **NOT** a static file.

index.ejs

sit774-task9-1p > views > index.ejs > ?

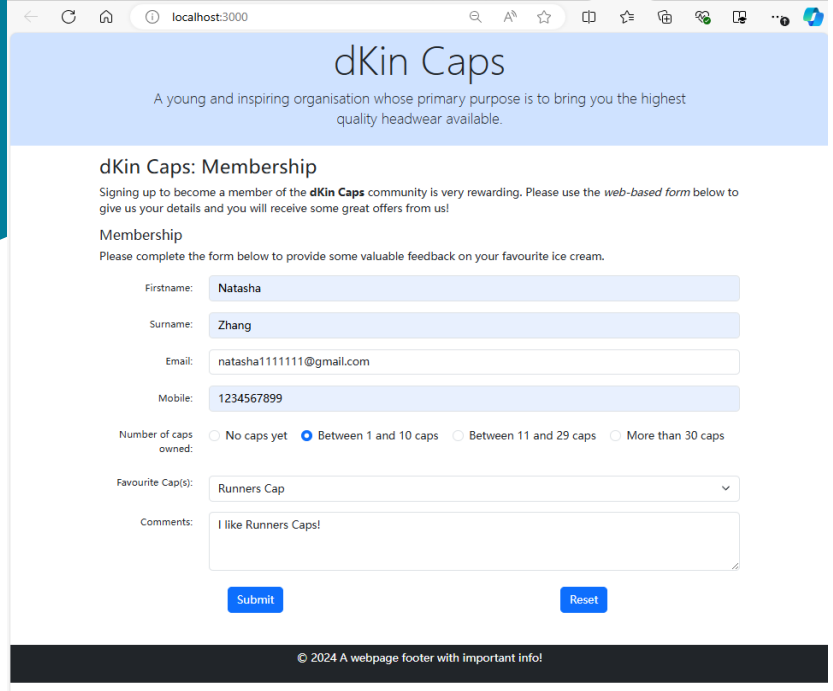
```
1   ader'); -%>
13
14  ass="container">
27
28  v id="formContainer">
29
30  <form action="/submitmembership" method="post" id="postMembershipForm">
31      <div class="row mb-3">
32          <label for="inputFirstname" class="col-sm-2 pt-2 col-form-label-sm text-sm-end">
33              Firstname:
34          </label>
35          <div class="col-sm-10">
36              <input type="text" class="form-control" id="inputFirstname"  name="firstname" required>
37          </div>
38      </div>
39      <div class="row mb-3">
40          <label for="inputSurname" class="col-sm-2 pt-2 col-form-label-sm text-sm-end">Surname:</label>
41          <div class="col-sm-10">
42              <input type="text" class="form-control" id="inputSurname" name="surname" required>
43          </div>
44      </div>
45      <div class="row mb-3">
46          <label for="inputEmail" class="col-sm-2 pt-2 col-form-label-sm text-sm-end">
47              Email:
48          </label>
49          <div class="col-sm-10">
50              <input type="text" class="form-control" id="inputEmail" name="email" required>
51          </div>
52      </div>
53      <div class="row mb-3">
54          <label for="inputMobile" class="col-sm-2 pt-2 col-form-label-sm text-sm-end">
55              Mobile:
56          </label>
57          <div class="col-sm-10">
58              <input type="tel" class="form-control" id="inputMobile" name="mobileNumber"
59                  placeholder="04xxxxxxxx" title="Invalid Number 04xxxxxxxx" pattern="[0-9]{10}" required>
60          </div>
61      </div>
```

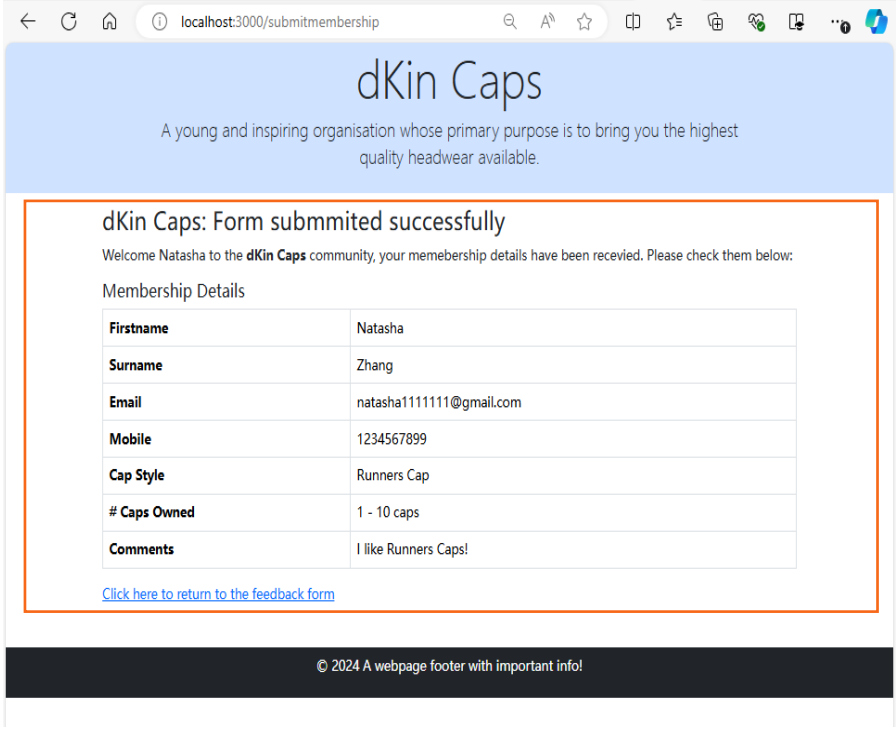
```
// The default route handler '/' uses the index.ejs temaplte
app.get('/', (req, res, next) => {
  res.render('index', { title: 'dKin Membership' });
});

app.post('/submitmembership', (req,res) => {
  let firstname = req.body.firstname;
  let surname = req.body.surname;
  let email = req.body.email;
  let mobile = req.body.mobileNumber;
  let numcaps = req.body.inputNumCaps;
  let capstyle = req.body.capstyle;
  let comment = req.body.comments;

  console.log(req.body);

  res.render('thankyou', {
    title: 'dKin Membership',
    firstname: firstname,
    surname: surname,
    email: email,
    mobile: mobile,
    numcaps: numcaps,
    capstyle: capstyle,
    comments: comment
  });
});
```

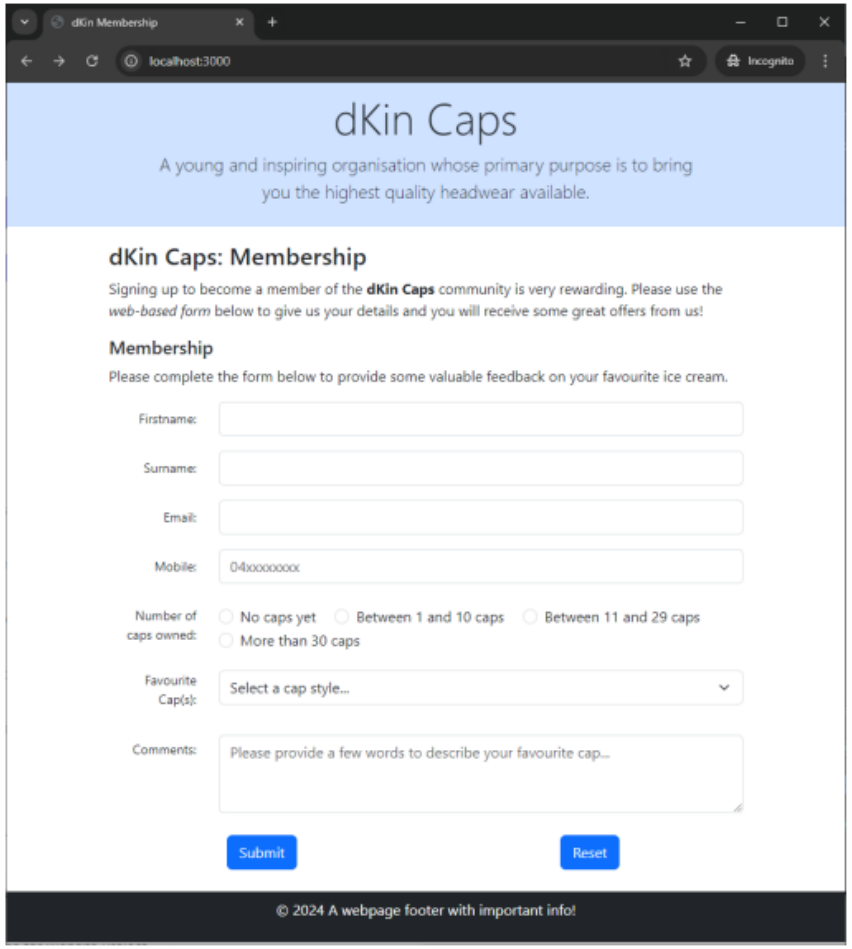




Ontrack task 9.2C: Create a Response Page

Accepting data form a client without checking it is valid first, is a very risky thing to do! We **can not** always trust what a client sends us to process!

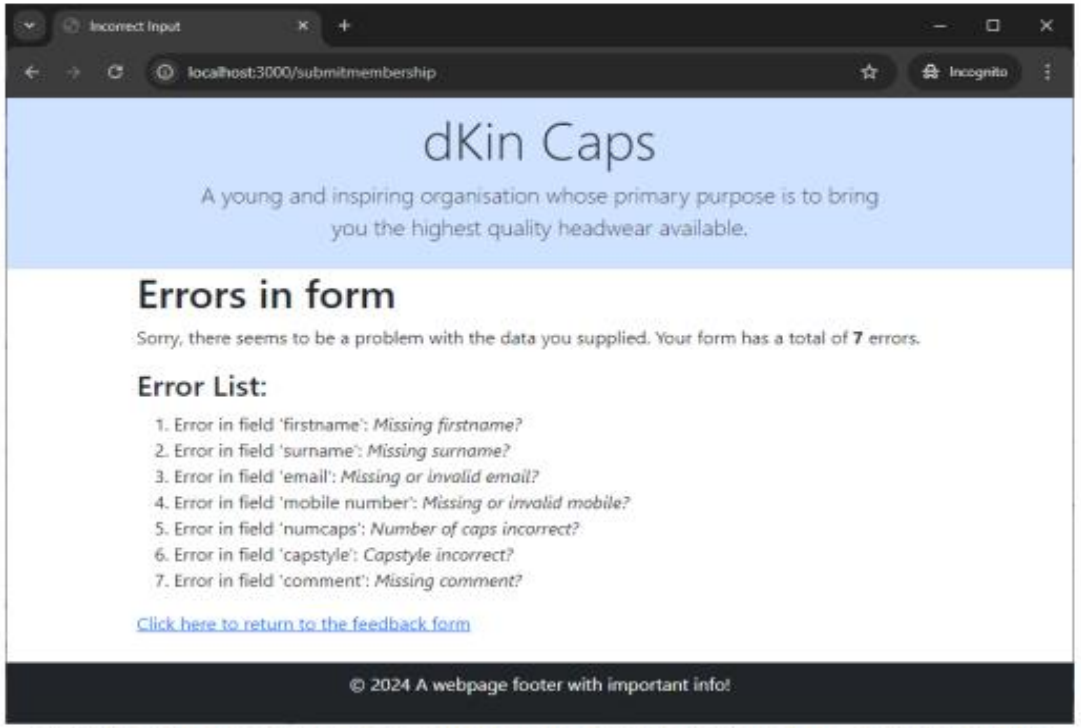
In this task you are provided with a *tweaked* input form, taken from Task 9.1P, with the `required` fields removed. This will allow for *testing* on the server side.



Task9.2.1 Modified form page

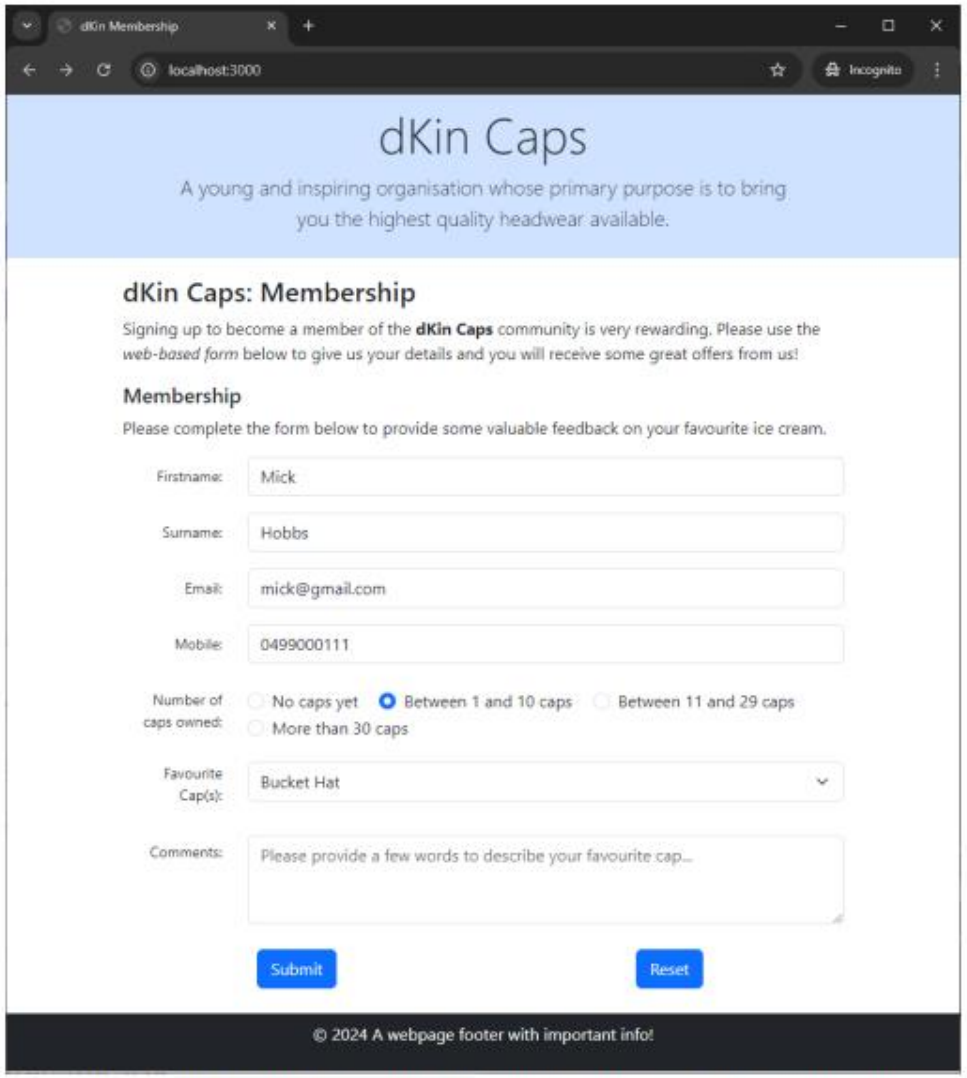
This tasks involves two activities. The first requires you to extend your `/submitmembership` POST handler from **Task 9.1P** to *validate* the form fields, checking they exist (contain a value) and that the email address ends with `@deakin.edu.au`. The `required` field has been removed from the provided `index.ejs` file… as to allow for *Server Side* checking.

Your solution should check for 1 or more errors, and display them accordingly. Samples of the possible error messages are shown below:

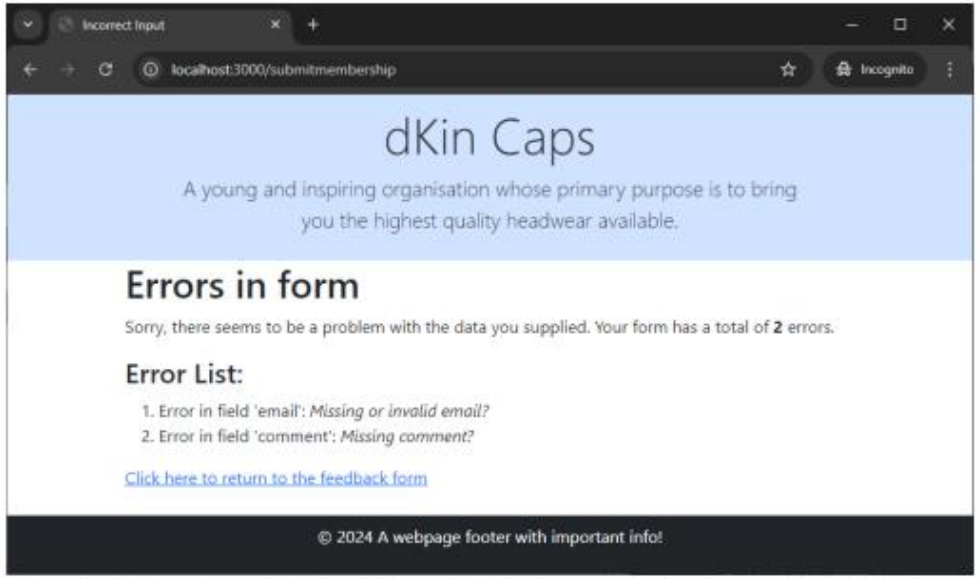


Task9.2.2a Example of error response for missing **ALL** 7 form fields

Requirements



Task9.2.2b Example a form with an incorrect email and missing comment

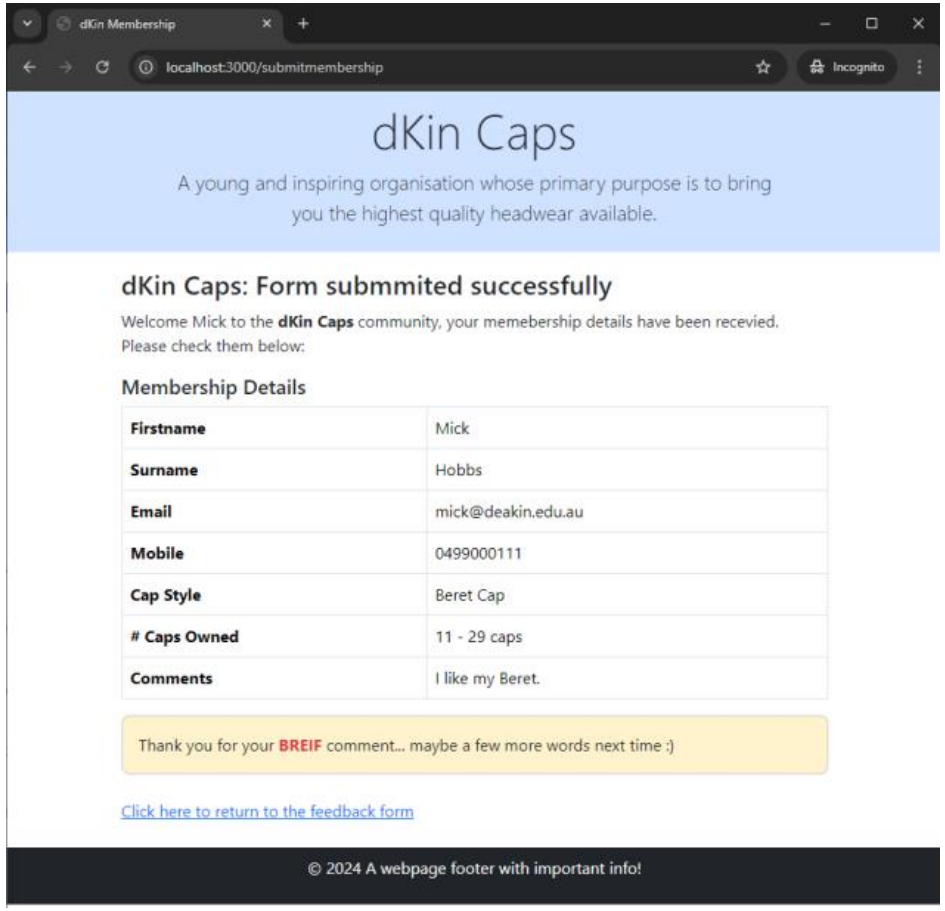


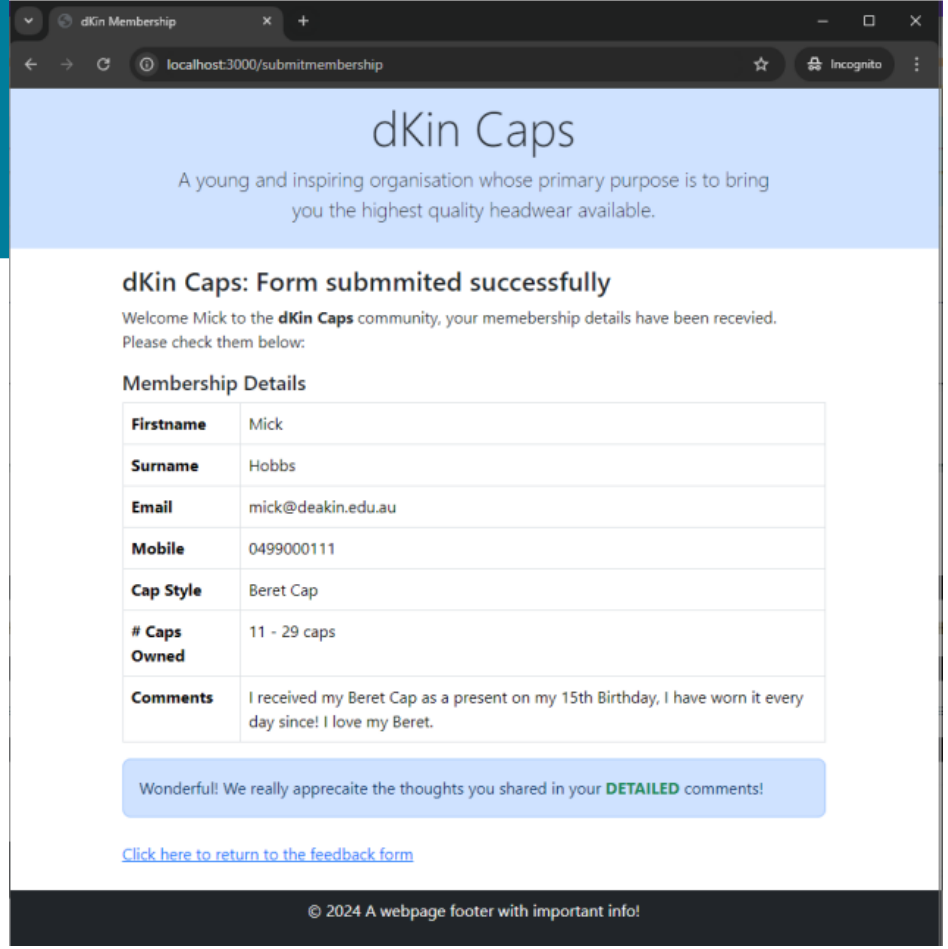
Task9.2.2c Example error response for an incorrect email and missing comment

DEAKIN UNIVERSITY

Hints and submission

The second is to extend the functionality of your `thankyou.ejs` template to dynamically present different content based on the **number of characters** (length) of the comment provided. If the comment was short (< 20 characters) then display a relevant message, otherwise a different message if the comment was longer. Examples of the possible output for the **BREIF** and **DETAILED** responses are shown below:





What will you submit?

You should submit:

- Source code of the modified *Node.js* file (i.e., the `index.js` file).
- Source code of the modified *thankyou* template file (i.e., the `thankyou.ejs` file).
- Screenshots (five) in a single PDF document, of the browser window showing:
  - the response from invalid (missing) **ALL** fields.
  - the response from invalid (missing) *SOME* fields.
  - the response from an invalid *EMAIL* field.
  - a response to a *BREIF* comment provided.
  - a response to a *DETAILED* comment provided.

DEAKIN UNIVERSITY

Hints

- sit774-task9-2c
  - views
    - 404.ejs
    - error.ejs
    - footer.ejs
    - header.ejs
    - index.ejs
    - invalid-form.ejs
    - thankyou.ejs
  - .gitignore
  - index.js
  - index.zip
  - package-lock.json
  - package.json

wk09 > sit774-task9-2c > views > invalid-form.ejs > ?

```
<%- include('header'); -%>

    <header>
        <div class="container-fluid bg-primary-subtle">
            <div class="col-sm-8 mx-auto text-center py-2">
                <h1 class="display-4">dKin Caps</h1>
                <p class="lead ">A young and inspiring organisation whose
                    primary purpose is to bring you the highest quality headwear available.</p>
            </div>
        </div>
    </header>

    <main>
        <div class="container">
            <h1>Errors in form</h1>
            <p>
                Sorry, there seems to be a problem with the data you supplied.
                Your form has a total of <strong>
                    <%= errors.length %>
                </strong> errors.
            </p>

            <h3>Error List:</h3>
              ......
```

- Add an unordered list <ol> … to display every occured error in the form.
- Use Loop to collect an error as a list item:

```
<% errors.forEach( (error)=> { %>
    <li>......</li>
<% }) %>
```

- thankyou.ejs

```
<% if( comments.length < 20 ) { %>
  <div class="p-3 bg-warning-subtle border border-secondary-subtle rounded-3">
    Thank you for your <strong class="text-danger">BREIF</strong> comment...
    maybe a few more words next time :)
  </div>
<% } else { %>
  <div class="p-3 text-primary-emphasis bg-primary-subtle border border-primary-subtle rounded-3">
    Wonderful! We really apprecaite the thoughts you shared in your
  <strong class="text-success">DETAILED</strong> comments!
  </div>
<% } %>
```

DEAKIN UNIVERSITY

Hints

JS index.js

```
// The default route handler '/' uses the index.ejs temaplte
app.get('/', (req, res, next) => {
  res.render('index', { title: 'dKin Membership' });
});


app.post('/submitmembership', (req,res) => {
  let firstname = req.body.firstname;
  let surname = req.body.surname;
  let email = req.body.email;
  let mobile = req.body.mobileNumber;
  let numcaps = req.body.inputNumCaps;
  let capstyle = req.body.capstyle;
  let comment = req.body.comments;

  let errorList = [];


  // Check for errors in the input (form) data -- Very _simple_ checks
  if( firstname === undefined || firstname.length === 0 ){
    errorList.push( { message: 'Missing firstname?', field: 'firstname' } );
    //return res.render('invalid-form', { title: 'Incorrect Input', message: 'Missing firstname?', field: 'firstname' });
  }

    ......
```

Summary

Before exploring how to make use of databases on the server side next week, we briefly introduced basics of database and SQL.

SQL is a standard language used for data manipulations in relational databases. More details and examples of SQL can be found from many resources, such as w3schools.com.

Next week we will explore how to create a database on the server, and use the database to store form data sent from the client. With the database, we can develop various applications to meet customer requirements.

Good luck with your Week 9 Ontrack tasks!

DEAKIN UNIVERSITY