WEBSITE AND DATABASE INTEGRATION

SFDWD401

Integrate website with database

Competence

REQF Level: 4

Credits: 6

Learning hours

60

Sector: ICT

Sub-sector: Software Development

Issue date: February, 2019

Purpose statement

This module describes the skills, knowledge, and attitudes required to connect a database to the web application, insert data, retrieve data, update data and delete data into / from a database using SQL language.

By the end of this module the learner will be able to connect web application to a database, create report to present summary information required.

Course Outline

Learning Unit1: Connect to the Database

20 Hours

Learning Outcomes:

1. Review the importance and strategies of database and website integration
2. Select connection tools and platforms according to the application specifications.
3. Connect web application to a database using server-side script languages.
4. Manage connections based on transactions.

Learning Unit2: Implement CRUD operations

25 Hours

Learning Outcomes:

1. Insert data into database using structured query language (SQL) in reference to standard queries.
2. Retrieve and display data from database in the most appropriate control according to the information requirements
3. Update data with user-supplied input according to the information changes.
4. Delete data from database according to the information requirements.

Learning Unit3: Create reports to present summary information

15 Hours

Learning Outcomes:

1. Display general data from database according to the information requirements.
2. Create specific report based on user-supplied input data.
3. Prepare customized and periodic Report according to information requirements.

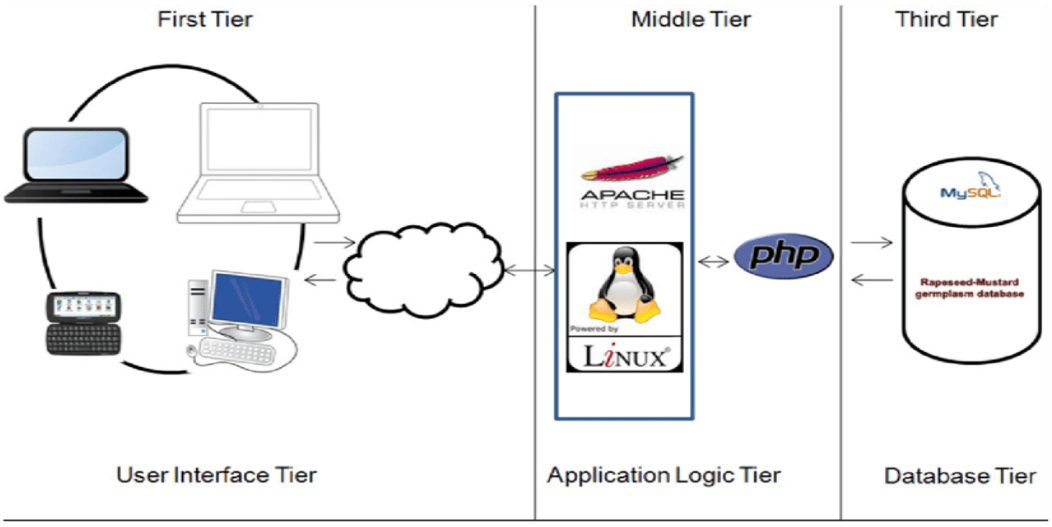
Learning Unit1: Connect to the Database

20 Hours

Learning Outcomes:

1. Review the importance and strategies of database and website integration **Importance to integrate website with database**

3-tier architecture is a client-server architecture in which the *functional process logic, data access, computer data storage and user interface* are developed and maintained as independent modules on separate platforms. A "**tier**" in this case can also be referred to as a "**layer**".



- ✓ **Presentation layer**

Presentation Tier- The **presentation tier** is the front end **layer** in the 3-**tier** system and consists of the user interface. This user interface is often a graphical one accessible through a **web** browser or **web**-based **application** and which displays content and information useful to an end user.

- ✓ **Logic layer**

The "business" logic tier plays the role of *transferring information between the website* and *the data tier*, including *integration of the required decision logic* or *transformation of transferred data* (calculations, aggregation of information from more data sources and the like).

This might be written in C#, Java, C++, Python, Ruby, **php**, etc.

It function includes:

- performing all required calculations and validations

- managing workflow o **state management**: to keep track of application execution o **session management**: to distinguish among application instances o **user identification**
    - o **service access**: to provide application services in a consistent way
- managing all data access for the presentation layer

- **Data access layer**

The data storage tier implements persistent data storage, with a relational database (RDBMS) or another type of database (NoSQL).

A **data access layer** (DAL) in computer software, is a **layer** of a computer program which provides simplified **access** to **data** stored in persistent storage of some kind, such as an entityrelational **database**.

It's also responsible for managing updates, allowing simultaneous (concurrent) access from web servers, providing security, ensuring the integrity of data, and providing support services such as data backup. Importantly, a good database tier must allow quick and flexible access to millions upon millions of facts.

This could be MSSQL, MySQL, Oracle, or PostgreSQL, Mongo, etc.

Strategies to integrate website with database

✓ **Protocols**

❖ **Transmission Control Protocol/Internet Protocol** (**TCP/IP**)

It is the standard **communication protocol suite** used for client/server communication over a network. **TCP** is the transport protocol that manages the exchange of data between hosts.

❖ **HTTP: the Hypertext Transfer Protocol**

The three-tier architecture provides a conceptual framework for web database applications. The Web itself provides the protocols and network that connect the client and middle tiers of the application: it provides the connection between the web browser and the web server. HTTP is one component that binds together the three-tier architecture.

HTTP allows resources to be communicated and shared over the Web.

✓ **Browser requests**

The browser sends an **HTTP request** message to **the** server, asking it to **send a** copy of **the** website to **the** client (you go to **the** shop and order **your** goods). This message, and all other data **sent** between **the** client and **the** server, **is sent** across **your** internet connection using TCP/IP.

The general idea though it that your **browser** sends a **request** for a specific file, often an HTML file.

The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers. HTTP works as a **request**-response protocol between a client and server. A **web** browser may be the client, and an application on a computer that hosts a **web** site may be the server.

- ✓ **Web Server processes**

A **web server** (or **Web server**) is **server** software, or hardware dedicated to running said software, that can satisfy World Wide **Web** client requests. A **web server** can, in general, contain one or more websites. A **web server processes** incoming network requests over **HTTP** and several other related protocols.

- **Web server responses**

2. **Select connection tools and platforms according to the application specifications.**

Software specifications

- ✓ DBMSs to be used

DBMS stands for database management system; in other words, a system that manages databases.

Some of the popular **DDBMS** are **Oracle**, **SQL Server**, **MySQL**, SQLite, and **IBM DB2**.

For our case, we will use **MySQL**

- ✓ Editors

An **HTML editor** is a specialized piece of software that assists in the creation of **HTML** code. Similar to text editors such as ***Notepad, Sublime text*** and ***TextEdit***, **HTML** editors allow users to enter raw text. Most (if not all) professional web developers use an **HTML editor** to create and maintain their websites.

Software used to create and change **Web** pages (HTML-based documents). Low-level **Web** page editors are used to write HTML code directly. High-level **Web** authoring programs provide complete WYSIWYG design with the ability (in varying degrees) to switch back and forth between the page layout and the HTML code.

Web content editors are responsible for planning, creating, **editing** and publishing information on websites.

- ✓ Browsers

Browser is a computer program with a graphical user interface for displaying HTML files, used to navigate the World Wide Web.

Web **Browser**. A web **browser**, or simply "**browser**," is an application used to access and view websites. Common web **browsers** include Microsoft Internet Explorer, Google Chrome, Mozilla

Firefox, and Apple Safari. The primary function of a web **browser** is to render HTML, the code used to design or "mark up" webpages

- **Identification of connection tools and platforms**
  - ✓ **server-side script language (JavaScript)**

Javascript (JS) is a scripting languages, primarily used on the Web. It is used to enhance HTML pages and is commonly found embedded in HTML code. *JavaScript* is an interpreted language. Thus, it doesn't need to be compiled. *JavaScript* renders web pages in an interactive and dynamic fashion.

- ✓ **Middleware.**

Software that acts as a bridge between an operating system or database and applications, especially on a network.

Middleware acts as a glue between different parts of an application made up of pure functions. To better understand middlewares let's examine the functionalities of an Express.**js** (minimalist web framework for Node.**js**) **middleware**. Infact Express.**js** is a framework based on middleware and routes.

Common **middleware examples** include database **middleware**, application server **middleware**, message-oriented **middleware**, web **middleware** and transaction-processing monitors.

3. **Connect web application to a database using server-side script languages.**
   - **Make a database structure**

     Database structure is the collection of record type and field type definitions that comprise your **database.**

Database design process consists of the following steps:

1. Determine the purpose of your database.
2. Find and organize the information required.
3. Divide the information into tables.
4. Turn information items into columns.
5. Specify primary keys.
6. Set up the table relationships.

7. Refine your design.
8. Apply the normalization rules.

- **tables**

A **database** table consists of **rows** and **columns**. A **database** table is also called a twodimensional array.

A **table** has a specified number of columns, but can have any number of rows.

In database terminology, each row is called a **record**

- ✓ **attributes and their data types**

In general, an **attribute** is a characteristic. In a **database** management system (DBMS), an **attribute** refers to a **database** component, such as a table. It also may refer to a **database** field. **Attributes** describe the instances in the row of a **database**.

primary key attributes

Flight Number	Airline	Arrival Time
34D34	Delta	4:25p
54T78	Alaska Air	5:45p
23R23	Southwest	2:01a
13G90	Southwest	12:30p

Types of **Attribute Data**. **Attribute data** can be store as one of five different field **types** in a table or database: character, integer, floating, date, and BLOB. The character property (or string) is for text based values such as the name of a street or descriptive values such as the condition of a street.

- ✓ **attributes' constraints**

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

NOT NULL - Ensures that a column cannot have a NULL value

UNIQUE - Ensures that all values in a column are different

PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table

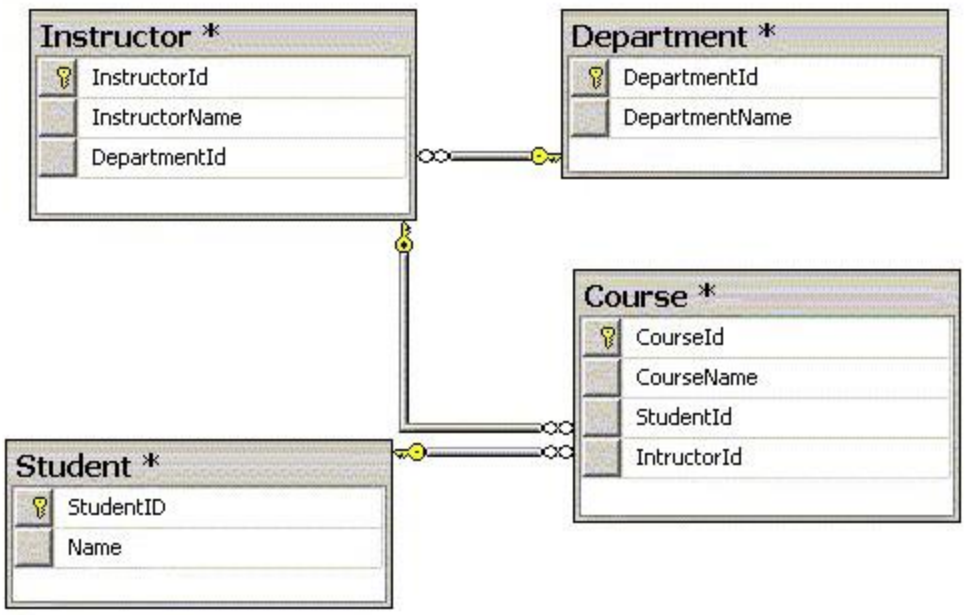
FOREIGN KEY - Uniquely identifies a row/record in another table

CHECK - Ensures that all values in a column satisfies a specific condition

DEFAULT - Sets a default value for a column when no value is specified

Tables' relationship

A table **relationship** is represented by a **relationship** line drawn between **tables** in the **Relationships** window. A **relationship** that does not enforce referential integrity appears as a thin line between the common fields supporting the **relationship**.



Environment setup

- ✓ Web browser

Web Browser. A **web browser**, or simply "**browser**," is an application used to access and view websites. Common **web browsers** include Microsoft **Internet** Explorer, Google Chrome, Mozilla Firefox, and Apple Safari. The primary function of a **web browser** is to render HTML, the code used to design or "mark up" webpages.

- ✓ Database

A database is an organized collection of data, generally stored and accessed electronically from a computer system. Where databases are more complex they are often developed using formal design and modeling techniques

- ✓ Middleware

Why do we use middleware?

Because the middleware is now the communication channel between the systems, all monitoring of your integration takes place in one system. One of the most crucial reasons that we use middleware is that it allows both systems to operate independently.

Description of connection string with authentication parameters

- ✓ Server

Server indicate which server to which database is hosted. It required because we are running server-side scripting language. For our case it will be ***Localhost.***

- ✓ User ID

When connecting to the server, you should indicate who want to connect (actually the username.) USERNAME that tipically is ***root***.

- ✓ Password

When connecting to the server, the user should have user name as well as the password for authentication. For our case the password will be ***empty***.

- ✓ Selecting Database Name

After connecting to the server, you need to specify the database name to which you need to access.

```
// Get the mysql service
var mysql = require('mysql');

// Add the credentials to access your database
var connection = mysql.createConnection({
   host     : 'localhost',
```

```
   user     : '<USERNAME that tipically is root>',
   password : '<PASSWORD or just use null if youre working lcocally>',    database
: '<DATABASE-NAME>'
});

// connect to mysql
connection.connect(function(err) {
   // in case of error
   if(err){
console.log(err.code);
console.log(err.fatal);
   }
});

// Perform a query
$query = 'SELECT * from MyTable LIMIT 10';

connection.query($query,    function(err,    rows,    fields)    {
if(err){
      console.log("An error ocurred performing the query.");
      return;
   }

   console.log("Query succesfully executed: ", rows);
});

//       Close       the       connection
connection.end (function(){     // The
connection has been closed
});
```

4. Manage connections based on transactions. States of a connection.

- ✓ Closed state connection

```
connection.end (function(){     // The
connection has been closed });
```

The above code can be used to terminate an opened connection.

- ✓ opened state connection

To create a connection, an object is create and initialize to mysql.createConnection; and later that object is used to establish connection “con.connection()”, where con is the object created before.

```
var con = mysql.createConnection({
  host: "localhost",       user:
"yourusername",   password:
"yourpassword",    database:
"mydb"
});

con.connect(function(err)    {
if      (err)      throw      err;
console.log("Connected!");
```

- **Connection error Handling**

  if (err) throw err; is used to handle error once it occur.

Learning Unit2: Implement CRUD (create, read, update, and delete) operations

25 Hours

Learning Outcomes:

1. **Insert data into database using structured query language (SQL) in reference to standard queries.**

- **Preparing a query string**
  - **Specify values**

Insert Into Table

To fill a table in MySQL, use the "INSERT INTO" statement.

Example

Insert a record in the "customers" table:

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",       user:
"yourusername",   password:
"yourpassword",    database:
"mydb"
});
```

```
con.connect(function(err)    {
if      (err)      throw      err;
console.log("Connected!");
 var sql = "INSERT INTO customers (name, address) VALUES ('Company Inc',
'Highway 37')";
 con.query(sql,  function  (err,  result)  {
if (err) throw err;
   console.log("1 record inserted");
 });
});
```

Insert Multiple Records

To insert more than one record, make an array containing the values, and insert a question mark in the sql, which will be replaced by the value array:
INSERT INTO customers (name, address) VALUES ?

Example

Fill the "customers" table with data:

```
var mysql = require('mysql');

var con = mysql.createConnection({
 host:  "localhost",      user:
"yourusername",  password:
"yourpassword",
 database: "mydb"
});

con.connect(function(err) {
 if             (err)             throw             err;
console.log("Connected!");
 var sql = "INSERT INTO customers (name, address) VALUES ?";
var values = [    ['John', 'Highway 71'],
   ['Peter', 'Lowstreet 4'],
   ['Amy', 'Apple st 652'],
   ['Hannah', 'Mountain 21'],
   ['Michael', 'Valley 345'],
   ['Sandy', 'Ocean blvd 2'],
   ['Betty', 'Green Grass 1'],
   ['Richard', 'Sky st 331'],
```

```
    ['Susan', 'One way 98'],
    ['Vicky', 'Yellow Garden 2'],
    ['Ben', 'Park Lane 38'],
    ['William', 'Central st 954'],
    ['Chuck', 'Main Road 989'],
    ['Viola', 'Sideway 1633']
  ];
  con.query(sql, [values], function (err, result) {
    if (err) throw err;
    console.log("Number of records inserted: " + result.affectedRows);
  });
});
```

Executing a query

- **Check if records are inserted**

When executing a query, a result object is returned.

The result object contains information about how the query affected the table.

The result object returned from the example above looks like this:

```
{      fieldCount:  0,
affectedRows:   14,
insertId:              0,
serverStatus:       2,
warningCount: 0,
  message: '\'Records:14  Duplicated: 0  Warnings: 0',
protocol41: true,   changedRows: 0
}
```

Get Inserted ID

For tables with an auto increment id field, you can get the id of the row you just inserted by asking the result object.

Note: To be able to get the inserted id, **only one row** can be inserted.

Example

Insert a record in the "customers" table, and return the ID:

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host:  "localhost",      user:
"yourusername",  password:
"yourpassword",    database:
"mydb"
});

con.connect(function(err)    {
if (err) throw err;
  var sql = "INSERT INTO customers (name, address) VALUES ('Michelle', 'Blue Village 1')";  con.query(sql,
function (err, result) {
   if (err) throw err;
   console.log("1 record inserted, ID: " + result.insertId);  });
});
```

2. Retrieve and display data from database in the most appropriate control according to the information requirements

- Preparing a query string
  - ✓ **Adding conditions**

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

WHERE Syntax

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

  - ✓ **Using different SQL Clauses**

SQL WHERE Clause

The WHERE clause is used to filter records.

The WHERE clause is used to extract only those records that fulfill a specified condition.

WHERE Syntax

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

SQL ORDER BY

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

ORDER BY Syntax

SELECT *column1, column2, ...*
FROM *table_name*
ORDER BY *column1, column2, ...* ASC|DESC;
SQL TOP, LIMIT or ROWNUM Clause The SQL SELECT TOP Clause

The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause is useful on large tables with thousands of records. Returning a large number of records can impact on performance.

Note: Not all database systems support the SELECT TOP clause. MySQL supports the LIMIT clause to select a limited number of records, while Oracle uses ROWNUM.

SQL Server / MS Access Syntax:

SELECT TOP *number|percent column_name(s)*
FROM *table_name*
WHERE *condition*;

MySQL Syntax:

SELECT *column_name(s)*
FROM *table_name*
WHERE *condition* LIMIT
number;

Oracle Syntax:

SELECT *column_name(s)*
FROM *table_name*
WHERE ROWNUM <= *number*;
SQL TOP, LIMIT and ROWNUM Examples

The following SQL statement selects the first three records from the "Customers" table:

Example

```
SELECT TOP 3 * FROM Customers;
```

The following SQL statement shows the equivalent example using the LIMIT clause:

Example

```
SELECT * FROM Customers LIMIT
3;
```

The following SQL statement shows the equivalent example using ROWNUM:

Example

```
SELECT * FROM Customers WHERE
ROWNUM <= 3;
```

SQL TOP PERCENT Example

The following SQL statement selects the first 50% of the records from the "Customers" table:

Example

```
SELECT TOP 50 PERCENT * FROM Customers;
```

ADD a WHERE CLAUSE

The following SQL statement selects the first three records from the "Customers" table, where the country is "Germany":

Example

```
SELECT TOP 3 * FROM Customers WHERE
Country='Germany';
```

The following SQL statement shows the equivalent example using the LIMIT clause:

Example

```
SELECT * FROM Customers
WHERE Country='Germany' LIMIT
3;
```

The following SQL statement shows the equivalent example using ROWNUM:

Example

```
SELECT * FROM Customers
WHERE Country='Germany' AND ROWNUM <= 3;
```

Using functions and operators

SQL MIN() and MAX() Functions

The MIN() function returns the smallest value of the selected column.

The MAX() function returns the largest value of the selected column.

MIN() Syntax

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

MAX() Syntax

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

SQL COUNT(), AVG() and SUM() Functions

The COUNT() function returns the number of rows that matches a specified criteria.

The AVG() function returns the average value of a numeric column.

The SUM() function returns the total sum of a numeric column.

COUNT() Syntax

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

AVG() Syntax

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

SUM() Syntax

SELECT SUM(*column_name*)
FROM *table_name*
WHERE *condition*;
SQL AND, OR and NOT Operators

The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

 The AND operator displays a record if all the conditions separated by AND is TRUE. The OR operator displays a record if any of the conditions separated by OR is TRUE.

The NOT operator displays a record if the condition(s) is NOT TRUE.

AND Syntax

SELECT *column1*, *column2*, ...
FROM *table_name*
WHERE *condition1* AND *condition2* AND *condition3* ...;

OR Syntax

SELECT *column1*, *column2*, ...
FROM *table_name*
WHERE *condition1* OR *condition2* OR *condition3* ...;

NOT Syntax

SELECT *column1*, *column2*, ...
FROM *table_name*
WHERE NOT *condition*;

SQL LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

- % - The percent sign represents zero, one, or multiple characters
- _ - The underscore represents a single character

Note: MS Access uses a question mark (?) instead of the underscore (_).

The percent sign and the underscore can also be used in combinations!

LIKE Syntax

SELECT *column1, column2, ...*
FROM *table_name*
WHERE *columnN* LIKE *pattern*;

Tip: You can also combine any number of conditions using AND or OR operators.

Here are some examples showing different LIKE operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

SQL IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

IN Syntax

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name* IN (*value1*, *value2*, ...);

or:

SELECT *column_name(s)*
FROM *table_name*

WHERE *column_name* IN (*SELECT STATEMENT*);

SQL BETWEEN Operator

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

BETWEEN Syntax

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name* BETWEEN *value1* AND *value2;*

- **Executing a query**
  - **Check if records are found**

```
app.get('/',(req,res)=>{

        mysqlConnection.query('SELECT * FROM students WHERE RegN < 10 ORDER BY FisrtName ASC',(err, rows, fields)=>{

                if(!err)

                        res.send(rows); //This send results to the browser.

                //console.log(rows); //This send results to the console.

                else

                console.log(err);

        })

})
```

- Creating an array of results app.get('/',(req,res)=>{
  mysqlConnection.query('SELECT * FROM students WHERE RegN < 10 ORDER BY FisrtName ASC',(err, rows, fields)=>{
  if(!err) res.send(rows); //This send results to the browser.
  //console.log(rows); //This send results to the console.
  else
  console.log(err);

```
        })
    })
```

3. **Update data with user-supplied input according to the information changes.**
   You can update existing records in a table by using the "UPDATE" statement.

- **Preparing a query string**

✓ **Adding conditions**

Condition can be added using **WHERE clause.**

✓ Using different SQL Clauses

Seen with SELECT.

✓ Using functions and operators

Seen with SELECT.

✓ **Executing a query**

```
con.connect(function(err)    {
if (err) throw err;
 var sql = "UPDATE customers SET address = 'Canyon 123' WHERE address = 'Valley 345'";
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log(result.affectedRows + " record(s) updated");
  });
});
```

✓ **Check if records are updated** `if (err) throw err; console.log(result.affectedRows + " record(s) updated");`

Notice the WHERE clause in the UPDATE syntax: The WHERE clause specifies which record or records that should be updated. If you omit the WHERE clause, all records will be updated!

4. **Delete data from database according to the information requirements.**

You can delete records from an existing table by using the "DELETE FROM" statement.

- Preparing a query string

✓ **Adding conditions**

Condition can be added using **WHERE clause.**

✓ **Using different SQL Clauses** Seen with SELECT.

✓ **Using functions and operators** Seen with SELECT.

- Executing a query

```
con.connect(function(err)    {
if (err) throw err;
 var sql = "DELETE FROM customers WHERE address = 'Mountain 21'";  con.query(sql,
function (err, result) {
   if (err) throw err;
   console.log("Number of records deleted: " + result.affectedRows);
 });
});
```

✓ **Check if records are deleted**

```
   if (err) throw err;      console.log("Number of records deleted: " +
result.affectedRows);
```

Notice the WHERE clause in the DELETE syntax: The WHERE clause specifies which record or records that should be deleted. If you omit the WHERE clause, all records will be deleted!

Learning Unit3: Create reports to present summary information

15 Hours

Learning Outcomes:

1. Display general data from database according to the information requirements.

- Loops in Arrays
  - While loop

The While Loop

The while loop loops through a block of code as long as a specified condition is true. **Syntax**

```
while (condition) {      code
block to be executed }
```

Example

In the following example, the code in the loop will run, over and over again, as long as a variable (i) is less than 10:

Example

```
while (i < 10) {
```

```
   text += "The number is " + i;
i++; }
```

If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {
   code block to be executed
}              while
(condition);
```

Example

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested: **Example**

```
do {
   text += "The number is " + i;
i++; }
while (i < 10);
```

- **for each loop**

The forEach() method calls a function once for each element in an array, in order.

```
var fruits = ["apple", "orange", "cherry"];
fruits.forEach(myFunction);

function myFunction(item, index) {
  document.getElementById("demo").innerHTML += index + ":" + item + "<br>";  }
```

Syntax *array*.forEach(*function(currentValue, index, arr), thisValue*)

Get the sum of all the values in the array:

```
var sum = 0; var numbers =
[65, 44, 12, 4];
numbers.forEach(myFunction);

function    myFunction(item)    {
sum += item;
 document.getElementById("demo").innerHTML = sum;
}
```

Example

For each element in the array: update the value with 10 times the original value:

```
var numbers = [65, 44, 12, 4];
numbers.forEach(myFunction)

function myFunction(item, index, arr) {
 arr[index] = item * 10; }
```

memory release data presentation in HTML tables

You're asked to build an HTML table with JavaScript. Starting from an **array** of "mountains" your task is to generate the table assigning **every key to a column** and **one row per object**.

Every object has the following shape:

```
{ name: "Monte Falco", height: 1658, place: "Parco Foreste Casentinesi" }
```

We have a name, an height and a place in which the peak is located in. But what makes an HTML table? An HTML table is an element containing tabular data, presented in rows and columns. That means given the following array:

```
let mountains = [
 { name: "Monte Falco", height: 1658, place: "Parco Foreste Casentinesi" },   { name: "Monte Falterona", height: 1654, place: "Parco Foreste Casentinesi"
}
];
```

We are expecting to generate the following table:

```
<table>
  <thead>
  <tr>
```

```
      <th>name</th>
      <th>height</th>
      <th>place</th>
    </tr>
    </thead>
    <tbody>
    <tr>
      <td>Monte Falco</td>
      <td>1658</td>
      <td>Parco Foreste Casentinesi</td>
    </tr>
    <tr>
      <td>Monte Falterona</td>
      <td>1654</td>
      <td>Parco Foreste Casentinesi</td>
    </tr>
    </tbody>
</table>
```

As you can see the table has a **thead (table head)** containing a **tr (table row)** which in turn contains three **th (table header)**.

Then there's the **tbody (table body)** containing a bunch of **tr (table rows)**. Each table row contains a certain number of **td elements (table cells)**.

With these requirements in place we can start coding our JavaScript file. Our starting point can be the following HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta  charset="UTF-8">        <title>Build  a
table</title>
</head> <body>
<table>
<!-- here goes our data! -->
</table>
</body>
<script src="build-table.js"></script> </html>
```

Save the file as **build-table.html** and go ahead to the next section!

How to generate a table with JavaScript: generating the table head

Create a new file named **build-table.js** in the same folder as build-table.html and start the file with the following array:

```
let mountains = [
  { name: "Monte Falco", height: 1658, place: "Parco Foreste Casentinesi" },  { name: "Monte Falterona", height: 1654, place: "Parco Foreste Casentinesi" },
  { name: "Poggio Scali", height: 1520, place: "Parco Foreste Casentinesi" },  { name: "Pratomagno", height: 1592, place: "Parco Foreste Casentinesi" },
  { name: "Monte Amiata", height: 1738, place: "Siena" } ];
```

Our first goal is to **generate the table head**. But let's think a moment about it. We know that the native method createElement() creates whatever element we pass to it. Say we want to create a table head, we can do document.createElement('thead'). But do we have a better alternative?

Let's head over to MDN, at the element table reference. You can see that the DOM interface for table is **HTMLTableElement**.

The interesting thing for HTMLTableElement is the methods it exposes. Among the methods there is **createTHead()**. Bingo! createTHead returns the table head element associated with a given table, but better, if no header exists in the table, createTHead creates one for us.

Armed with this knowledge let's create a function in our file, taking the table as a parameter. Given the table we can create a new thead inside it:

```
function generateTableHead(table) {  let thead = table.createTHead(); }
```

Now let's grab our table (remember we have one in build-table.html) and pass that to our function:

```
function generateTableHead(table) {  let thead = table.createTHead();
}  let table = document.querySelector("table");
generateTableHead(table);
```

If you call build-table.html in a browser you'll see nothing on the screen but the developer console will show you a thead right inside the table. We're half way down to populating the table head. We saw that the table head contains a row filled with a bunch of th (table headers). Every table header must map to a key describing what our data is made of.

The information is already there, inside the first object in the mountain array. We can iterate over the keys of the first object:

```
let mountains = [
  { name: "Monte Falco", height: 1658, place: "Parco Foreste Casentinesi" },
```

```
// ];
```

and generate three table headers with said keys. But first we need to add a row to our thead! How? document.createElement('tr')? No no. Our HTMLTableRowElement is kind enough to offer an **insertRow()** method, to be called on our table header. Let's refactor a bit our generateTableHead function:

```
function generateTableHead(table) {  let thead = table.createTHead();  let row = thead.insertRow(); }
```

And while we're there let's think of populating the table head. The new row should contain three th (table headers). We need to create these th elements manually and for each th (table header) we will append a text node. Our function can take another parameter to iterate over:

```
function generateTableHead(table, data) {  let thead = table.createTHead();  let row = thead.insertRow();  for (let key of data) {
  let th = document.createElement("th");  let text = document.createTextNode(key);  th.appendChild(text); row.appendChild(th);
 }
}  let table = document.querySelector("table"); let data = Object.keys(mountains[0]); generateTableHead(table, data);
```

Save the file and refresh build-table.html: you should see your table head being populated with name, height and place as table headers. Congratulations! **Sometimes it feels so good to take a break from React and Vue just for the sake of recalling how hard and cumbersome direct DOM manipulation is**. But stay here! We're not done yet.

Time to populate the table ...

How to generate a table with JavaScript: generating rows and cells

For populating the table we will follow a similar approach but this time we need to iterate over every object in the array of mountains. And while we're inside the **for...of** loop we will create **a new row for every item**.

For creating rows you will use **insertRow()**.

But we cannot stop here. Inside the main loop we need an **inner loop**, this time a **for...in**. The inner loop iterates over every key of the current object and in the same time it:

- creates a new cell
- creates a new text node

- appends the text node to the cell

The cells are created with another useful method of HTMLTableRowElement, **insertCell()**.

That is, with the logic above we can populate our table. Open up **build-table.js** and create a new function named generateTable. The signature can be the same as our existing function:

```
function generateTable(table, data) {  for (let
element of data) {  let row =
table.insertRow();  for (key in element) {
let cell = row.insertCell();
    let text = document.createTextNode(element[key]);  cell.appendChild(text);
  }
 } }
```

To run this function you will call it like so:

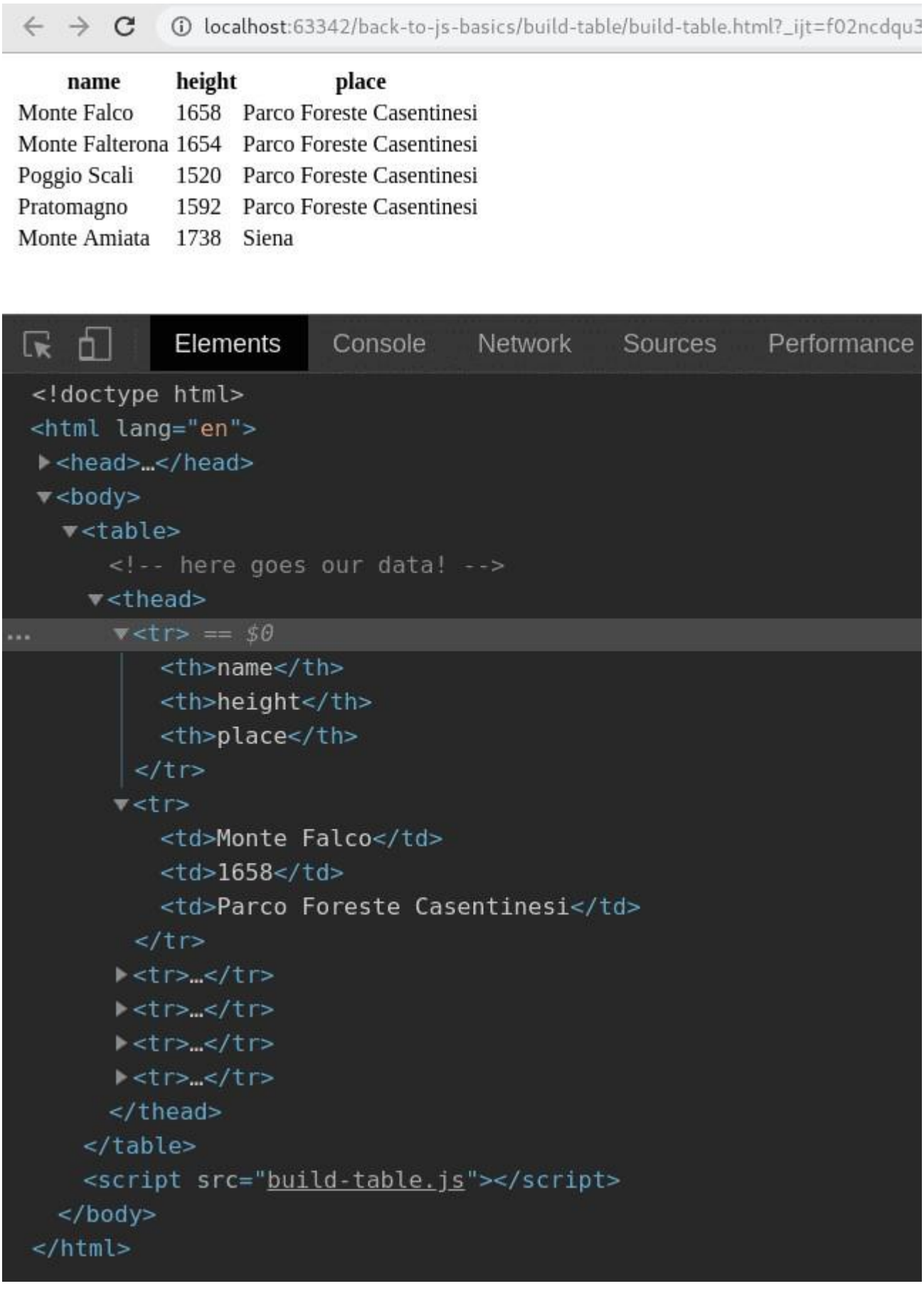
```
generateTable(table, mountains);
```

Let's take a look at the complete code:

```
let mountains = [
 { name: "Monte Falco", height: 1658, place: "Parco Foreste Casentinesi" },  { name: "Monte
Falterona", height: 1654, place: "Parco Foreste Casentinesi"
},
 { name: "Poggio Scali", height: 1520, place: "Parco Foreste Casentinesi" },  { name:
"Pratomagno", height: 1592, place: "Parco Foreste Casentinesi" },
 { name: "Monte Amiata", height: 1738, place: "Siena" }
];
function generateTableHead(table, data) {  let
thead = table.createTHead();  let row =
thead.insertRow();  for (let key of data) {
  let th = document.createElement("th");
  let text = document.createTextNode(key);
th.appendChild(text);  row.appendChild(th);
 }
}  function generateTable(table, data) {  for
(let element of data) {  let row =
table.insertRow();  for (key in element) {
let cell = row.insertCell();
    let text = document.createTextNode(element[key]);  cell.appendChild(text);
  }
 }
}  let table = document.querySelector("table"); let data
=  Object.keys(mountains[0]);
```

```
generateTableHead(table, data); generateTable(table,
mountains);
```

Do you think it works? Let's give it a shot:

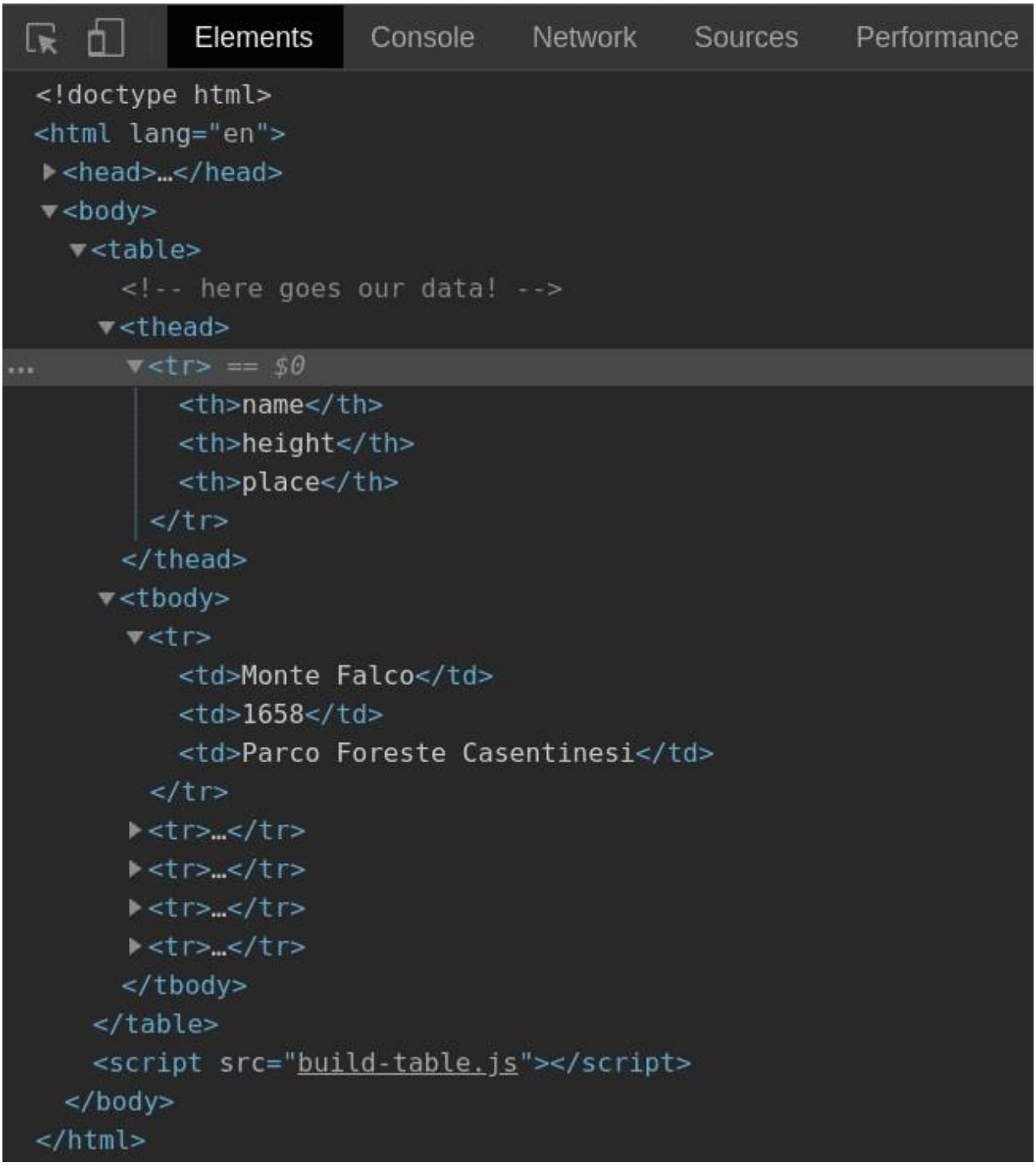

localhost:63342/back-to-js-basics/build-table/build-table.html?_ijt=f02ncdqu3
name height place
Monte Falco 1658 Parco Foreste Casentinesi
Monte Falterona 1654 Parco Foreste Casentinesi
Poggio Scali 1520 Parco Foreste Casentinesi
Pratomagno 1592 Parco Foreste Casentinesi
Monte Amiata 1738 Siena
Elements
Console
Network
Sources
Performance
<!doctype html>
<html lang="en">
<head>…</head>
<body>
<table>
<!-- here goes our data! -->
<thead>
<tr> == $0
<th>name</th>
<th>height</th>
<th>place</th>
</tr>
<tr>
<td>Monte Falco</td>
<td>1658</td>
<td>Parco Foreste Casentinesi</td>
</tr>
<tr>…</tr>
<tr>…</tr>
<tr>…</tr>
<tr>…</tr>
</thead>
</table>
<script src="build-table.js"></script>
</body>
</html>

Wow. It looks like our rows are being appended to the table head rather than to the table body. Also, **there is no table body**!

But what happens if we switch the functions order? Let's try:

```
// omitted for brevity
 let table = document.querySelector("table"); let data =
Object.keys(mountains[0]);
generateTable(table, mountains); // generate the table first
generateTableHead(table, data); // then the head and refresh the browser
again:
```

name	height	place
Monte Falco	1658	Parco Foreste Casentinesi
Monte Falterona	1654	Parco Foreste Casentinesi
Poggio Scali	1520	Parco Foreste Casentinesi
Pratomagno	1592	Parco Foreste Casentinesi
Monte Amiata	1738	Siena



It works! Plus we'got a tbody (table body) for free. How so? When you call insertRow() on an empty table the methods takes care of creating a tbody for you (if none is present).

Well done! Our code may not be well organized (too many global bindings) but we'll come to that in one of the next posts.

By now **you should be able to manipulate HTML tables without any external library**.

Pagination and paragraphing

Pagination, also known as **paging**, is the process of dividing a document into discrete pages, either electronic pages or printed pages.

Paragraphing (to make *paragraph*-ending decisions), and automated *pagination* (to make pagebreaking decisions).

HTML CODES

```
<script type="text/javascript">

 function previous(){

       new_page = parseInt($('#current_page').val()) - 1;

   //if there is an item before the current active link run the function

if($('.active_page').prev('.page_link').length==true){

go_to_page(new_page);

   }

    }

 function next(){     new_page = parseInt($('#current_page').val()) + 1;     //if there is an item after the current active link run the function          if($('.active_page').next('.page_link').length==true){ go_to_page(new_page);

   }

}

function go_to_page(page_num){

   //get the number of items shown per page          var

show_per_page = parseInt($('#show_per_page').val());
```

```
   //get the element number where to start the slice from
start_from = page_num * show_per_page;
   //get the element number where to end the slice
end_on = start_from + show_per_page;
   //hide all children elements of content div, get specific items and show them
   $('#content').children().css('display', 'none').slice(start_from, end_on).css('display', 'block');
   /*get the page link that has longdesc attribute of the current page and add active_page class to it
   and remove that class from previously active page link*/
   $('.page_link[longdesc=' + page_num +']').addClass('active_page').siblings('.active_page').removeClass('active_page');
//update the current page input field
   $('#current_page').val(page_num);
}
</script>
   <!-- the input fields that will hold the variables we will use -->
   <input type='hidden' id='current_page' />
   <input type='hidden' id='show_per_page' />
   <!-- Content div. The child elements will be used for paginating(they don't have to be all the same, you can use divs, paragraphs, spans, or whatever you like mixed together). '-->
   <div id='content'>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Lorem ipsum dolor sit amet, consectetur adipiscing elit. </p>
      <p>Vestibulum consectetur ipsum sit amet urna euismod imperdiet aliquam urna laoreet. Lorem ipsum dolor sit amet, consectetur adipiscing elit.</p>
```

** Curabitur a ipsum ut elit porttitor egestas non vitae libero. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Lorem ipsum dolor sit amet, consectetur adipiscing elit. **

** Pellentesque ac sem ac sem tincidunt euismod. Lorem ipsum dolor sit amet, consectetur adipiscing elit. **

** Duis hendrerit purus vitae nibh tincidunt bibendum. Lorem ipsum dolor sit amet, consectetur adipiscing elit. **

** Nullam in nisi sit amet velit placerat laoreet. Lorem ipsum dolor sit amet, consectetur adipiscing elit. **

** Vestibulum posuere ligula non dolor semper vel facilisis orci ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Lorem ipsum dolor sit amet, consectetur adipiscing elit. **

** Donec tincidunt lorem et dolor fringilla ut bibendum lacus fringilla. Lorem ipsum dolor sit amet, consectetur adipiscing elit. **

** In non eros eu lacus vestibulum sodales. **

** Duis ultrices metus sit amet sem adipiscing sit amet blandit orci convallis. **

** Proin ullamcorper est vitae lorem mollis bibendum. **

** Maecenas congue fringilla enim, tristique laoreet tortor adipiscing eget. **

** Duis imperdiet metus et lorem venenatis nec porta libero porttitor. **

** Maecenas lacinia lectus ac nulla commodo lacinia. **

** Maecenas quis massa nisl, sed aliquet tortor. **

** Quisque porttitor tellus ut ligula mattis luctus. **

** In at mi dolor, at consectetur risus. **

** Etiam id erat ut lorem fringilla dictum. **

** Curabitur sagittis dolor ac nisi interdum sed posuere tellus commodo. **

** Pellentesque quis magna vitae quam malesuada aliquet. **

** Curabitur tempus tellus quis orci egestas condimentum. **

** Maecenas laoreet eros ac orci adipiscing pharetra. **

** Nunc non mauris eu nibh tincidunt iaculis. **

** Ut semper leo lacinia purus hendrerit facilisis. **

** Praesent et eros lacinia massa sollicitudin consequat. **

** Proin non mauris in sem iaculis iaculis vel sed diam. **

** Nunc quis quam pulvinar nibh volutpat aliquet eget in ante. **

** In ultricies dui id libero pretium ullamcorper. **

** Morbi laoreet metus vitae ipsum lobortis ultrices. **

** Donec venenatis egestas arcu, quis eleifend erat tempus ullamcorper. **

** Morbi nec leo non enim mollis adipiscing sed et dolor. **

** Cras non tellus enim, vel mollis diam. **

** Phasellus luctus quam id ligula commodo eu fringilla est cursus. **

** Ut luctus augue tortor, in volutpat enim. **

** Cras bibendum ante sed erat pharetra sodales. **

** Donec sollicitudin enim eu mi suscipit luctus posuere eros imperdiet. **

** Vestibulum mollis tortor quis ipsum suscipit in venenatis nulla fermentum. **

** Proin vehicula suscipit felis, vitae facilisis nulla bibendum ac. **

** Cras iaculis neque et orci suscipit id porta risus feugiat. **

** Suspendisse eget tellus purus, ac pulvinar enim. **

** Morbi hendrerit ultrices enim, ac rutrum felis commodo in. **

** Suspendisse sagittis mattis sem, sit amet faucibus nisl fermentum vitae. **

** Nulla sed purus et tellus convallis scelerisque. **

** Nam at justo ut ante consectetur faucibus. **

** Proin dapibus nisi a quam interdum lobortis. **

** Nunc ornare nisi sed mi vehicula eu luctus mauris interdum. **

** Mauris auctor suscipit tellus, at sodales nisi blandit sed. **

```
</div>

<!-- An empty div which will be populated using jQuery -->
<div id='page_navigation'></div>
```

CSS CODES

```
#content{      min-height:
250px;         border:  1px
solid                  #ccc;
background:      #f9f9f9;
padding:              20px;
margin: 20px;    border-
radius: 10px;
}
#content       p       {
padding: 15px;
}
#page_navigation            {
margin: 0 20px;
}

#page_navigation          a{
padding:3px;    border:1px
solid gray;      margin:2px;
```

```
color:black;                text-
decoration:none

}
.active_page{
background:darkblue;    color:white
!important;
}
```

JS CODES

```
$(document).ready(function(){

   //how much items per page to show
var show_per_page = 5;
   //getting the amount of elements inside content div          var
number_of_items = $('#content').children().size();          //calculate the
number of pages we are going to have       var number_of_pages =
Math.ceil(number_of_items/show_per_page);

   //set the value of our hidden input fields
   $('#current_page').val(0);
   $('#show_per_page').val(show_per_page);

   //now when we got all we need for the navigation let's make it '

   /*
   what are we going to have in the navigation?
```

```
- link to previous page
- links to specific pages
- link to next page
   */
   var navigation_html = '<a class="previous_link" href="javascript:previous();">Prev</a>';  var current_link = 0;  while(number_of_pages > current_link){
      navigation_html += '<a class="page_link" href="javascript:go_to_page(' + current_link +')" longdesc="' + current_link +'">'+ (current_link + 1) +'</a>'; current_link++;
   }
   navigation_html += '<a class="next_link" href="javascript:next();">Next</a>';

   $('#page_navigation').html(navigation_html);

   //add active_page class to the first page link
   $('#page_navigation .page_link:first').addClass('active_page');

   //hide all the elements inside content div
   $('#content').children().css('display', 'none');

   //and show the first n (show_per_page) elements
   $('#content').children().slice(0, show_per_page).css('display', 'block');

});
```

data in form elements

Form elements may include ***textareas, selects, radio buttons and checkboxes*.**

<form action="index.php" id="form_name" method="post" >

Use below code to get all form element by JS :-

document.forms["form_name"].getElementsByTagName("input");

Note:- Above Code will work only if you don't have selects or textareas in your form.

If you have assigned id in DOM element like below,

<input type="text" name="name" id="uniqueID" value="value" />

Then you can access it via below code:- Javascript:-

var nameValue = document.getElementById("uniqueID").value;

If you have Radio button in your form, then use below code:-

<input type="radio" name="radio_name" value="1" > 1
<input type="radio" name="radio_name" value="0" > 0

Javascript:-

var radios = document.getElementsByName('radio_name');

for (var i = 0, length = radios.length; i < length; i++) { if (radios[i].checked)
{
// do whatever you want with the checked radio alert(radios[i].value);

// only one radio can be logically checked, don't check the rest break;
} }

2. Create specific report based on user-supplied input data.
User input data handling

Data from a form can be processed by a script in the same page, or they may be sent to another page when the values are used to define the content of this page.

In a previous article we saw how to pass data from one HTML page to another. This article will explain in more detail the use of form to submit data to a page or a script on the server.

Form data are used in the page, or another one

When the values are processed by a script in the same page that contains the form (or an included file), the *action* attribute of the object *form* has no value.

<form action="" >

To button that sends the data, is associated in this case an *onClick* event that calls the JavaScript function defined to process the values of the form.

<script type="text/javascript"> function processForm() { ... } </script><input type="button" onClick="processForm()" > The form may

also be sent bu an image field.

To access the elements locally, they are identified by a string made of the word *document*, the name of the form, name of the object, and the attribute value.

If a form has for *name* attribute "myform", a text field named "mytext", the value is accessed as follows:

var x = document.myform.mytext.value; **Action is**

assigned the name of a file to send data

When, on the contrary we want to transmit values to another page or a script, we assign the file name to the attribute *action*:

<form method="GET" action="myfiler.php">

That file is an HTML or PHP page or a script in another language, it makes no difference at sending, it is only the processing of the values that is different.
However it remains to define how values are extracted from elements of the form according to the type of object.

Name and values of form objects are transmitted

The transmission of values has a single general form, but the operating principles can depend on the object.

In most cases, the attributes *name* and *value* are used to build a string that is transmitted ar parameter to the target.

<input type="text" name="mytext" value="some texte"> gives the

string:

mytext=some+texte

The variables are separated by the & symbol and the whole is separated from the file name by the ? symbol.

Example: myfile.php?mytext=some+text&checkbox=cb1

This string of parameters is built automatically, you have just to assign the file name to the *action* attribute of the form.

Special case are:

1. Text boxes.
   It is the content of the tag which is the value

   <input type="textarea">Content</>

2. Checkboxes.
3. The name and the value are transmitted when the box is checked, or if *checked* was added to the definition and that the user does not unchecks the case. Otherwise the item is ignored, nothing is transmitted.
4. Radio buttons groups.
   Radio buttons in a group have the same name, such as "Radio" and a specific value, such as "Radio1", "Radio2", etc..
   The name "Radio" is transmitted with the value of the selected button, for example:

   radio=radio2

   if the second button is selected.

5. Menus and lists.
   They contain several *option* tags. It is the content of the selected option tag which forms the value associated with the name of the list.
6. Image field.
   When you click on a image field, it sends form data. For the the image field, two values x and y are created for the the position where you clicked in the image.
   For example, if the name is "image", and that you click into position x = 10 and y = 20, the following string will be transmitted:

   image.x=10&image.y=20

The demonstration allows for interactive tests on all case of transmission of values.

Case of checkboxes

The *checkbox* is an issue when it is not checked, if the script that receives the values needs for all the elements of the form, since nothing is transmitted in this case. One can easily circumvent this problem by assigning the *value* field with the *onClick* event.

We want the checkbox "cb" has a value "yes" when it is checked and "no" when it is not. We initialize the value which corresponds to the initial state, if it is *checked*, with "yes".

<input type="checkbox" name="cb" value="yes" checked onClick="cbChange(this)" >

A function updates the value when the state of the checkbox changes:

```
function cbChange(element)
{               if(element.checked)
element.value="yes";     else
    element.value="no"; }
```

We'll see now how the values sent are retrieved by the page that receives them.

Receiving the values

The parameters associated with the URL of a page is assigned to the *search* attribute of the *location* object. The string may be viewed with the following JavaScript script, which is used in our demonstration.

```
function receive()
{
   var parameters = location.search;
   document.getElementById("string").innerHTML     =     parameters;     }
window.onload=receive;
```

The "string" name is the ID of the tag where we will store the parameter string to view it.

To isolate the elements of this string of parameterd, we proceed in 3 steps:

1. The symbol ? is skipped by a call to *substring* method. location.search.substring(1)

2. The string is cutted in elements on the & separator and the *split* method.

   location.search.substring(1).split("&");

3. Each name and value of component are separated in the same way.

The complete script becomes:

```
var parameters = location.search.substring(1).split("&"); var data = "";
for (x in parameters)
{
   var temp = parameters[x].split("="); thevar =
unescape(temp[0]); thevalue = unescape(temp[1]);
thevalue = thevalue.replace("+", " "); data += thevar +
"=" + thevalue + "<br>";
} document.getElementById("data").innerHTML = data;
```

The elements of the parameters string become elements of where the index is *x*. This gives the names and values, in the example, they are concatenated in the *data* variable to be displayed.

If the data are sent to a PHP script on the server, or a page with PHP code, they will be available in the global variables $_GET or $_POST depending on the method used. There are associative arrays.

Data are recovered by the keys. For example, from this HTML code:

```
<form method="POST" action="xxx.php">
<input type="text" name="login" value="xxx" /> </form>
```

The PHP code is:

```
$login = $_POST['login'];
```

- **Input processing using form methods**
  - **GET method**

```
<FORM NAME="myform" ACTION="" METHOD="GET">

Enter something in the box: <BR>

<INPUT TYPE="text" NAME="inputbox" VALUE=""><P>

<INPUT TYPE="button" NAME="button" Value="Click" onClick="testResults(this.form)">

</FORM>
```

- *FORM NAME="myform"* defines and names the form. Elsewhere in the JavaScript you can reference this form by the name *myform*. The name you give your form is up to you, but it should comply with JavaScript's standard variable/function naming rules (no spaces, no weird characters except the underscore, etc.).

- *ACTION=""* defines how you want the browser to handle the form when it is submitted to a CGI program running on the server. As this example is not designed to submit anything, the URL for the CGI program is omitted.
- *METHOD="GET"* defines the method data is passed to the server when the form is submitted. In this case the atttibute is puffer as the example form does not submit anything.
- *INPUT TYPE="text"* defines the text box object. This is standard HTML markup.
- *INPUT TYPE="button"* defines the button object. This is standard HTML markup except for the onClick handler.
- *onClick="testResults(this.form)"* is an event handler -- it handles an event, in this case clicking the button. When the button is clicked, JavaScript executes the expression within the quotes. The expression says to call the testResults function elsewhere on the page, and pass to it the current form object.

Listing 1. testform.html

```
<HTML>
<HEAD>
<TITLE>Test Input</TITLE> <SCRIPT
LANGUAGE="JavaScript">
function testResults (form) {       var
TestVar     =     form.inputbox.value;
alert ("You typed: " + TestVar);
}
</SCRIPT>
</HEAD>
<BODY>
<FORM NAME="myform" ACTION="" METHOD="GET">Enter something in the box: <BR>
<INPUT TYPE="text" NAME="inputbox" VALUE=""><P>
<INPUT TYPE="button" NAME="button" Value="Click" onClick="testResults(this.form)">
</FORM>
</BODY>
</HTML>
```

Listing 2. set_formval.html

```
<HTML>

<HEAD>

<TITLE>Test Input </TITLE>

<SCRIPT LANGUAGE="JavaScript">
```

```
function readText (form) {
TestVar =form.inputbox.value;
alert ("You typed: " + TestVar);
} function writeText (form) {
form.inputbox.value = "Have a nice day!"
}
</SCRIPT>
</HEAD>
<BODY>
<FORM NAME="myform" ACTION="" METHOD="GET">
Enter something in the box: <BR>
<INPUT TYPE="text" NAME="inputbox" VALUE=""><P>
<INPUT TYPE="button" NAME="button1" Value="Read" onClick="readText(this.form)">
<INPUT TYPE="button" NAME="button2" Value="Write" onClick="writeText(this.form)">
</FORM>
</BODY>
</HTML>
```

- **POST method**

- Information format and display

3. Prepare customized and periodic according to information requirements.

- Custom parameters
  - ✓ Start parameter
  - ✓ End parameter
- **Methods for sending Data**
  - ✓ GET method

The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.

- ✓ POST method

The POST method requests that the server accept the data enclosed in the request as a new object/entity of the resource identified by the URI.

- **Form attributes**
  - ✓ Action: This attribute specify the file to which the form data are sent for being processed/handled.
  - ✓ Method
  - ✓ Enctype
- Report format according to the application specifications.

This is an example of an HTML form:

```
<form method="POST" action="/submit-form">
  <input type="text" name="username" />
  <input type="submit" />
</form>
```

When the user presses the submit button, the browser will automatically make a POST request to the /submit-form URL on the same origin of the page. The browser sends the data contained, encoded as application/x-www-form-urlencoded. In this particular example, the form data contains the username input field value.

Forms can also send data using the GET method, but the vast majority of the forms you'll build will use POST.

The form data will be sent in the POST request body.

To extract it, you will need to use the express.urlencoded() middleware, provided by Express:

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

app.use(express.urlencoded())
```

Now, you need to create a POST endpoint on the /submit-form route, and any data will be available on Request.body:

```
app.post('/submit-form', (req, res) => {
const username = req.body.username
  //...
```

```
  res.end() })
```

Don’t forget to validate the data before using it, using express-validator.

Forms are an integral part of the web. Almost every website we visit offers us forms that submit or fetch some information for us. To get started with forms, we will first install the *bodyparser*(for parsing JSON and url-encoded data) and multer(for parsing multipart/form data) middleware.

To install the *body-parser* and *multer*, go to your terminal and use − npm install --save body-parser multer

Replace your **index.js** file contents with the following code −

```
var express = require('express'); var
bodyParser = require('body-parser'); var
multer = require('multer'); var upload =
multer();
var app = express();

app.get('/', function(req, res){
res.render('form');
});

app.set('view engine', 'pug');

app.set('views', './views');

// for parsing application/json
app.use(bodyParser.json());

// for parsing application/xwww-
app.use(bodyParser.urlencoded({ extended: true }));
//form-urlencoded

// for parsing multipart/form-data app.use(upload.array());
app.use(express.static('public'));

app.post('/', function(req, res){
console.log(req.body);
  res.send("recieved your request!");
}); app.listen(3000);
```

After importing the body parser and multer, we will use the **body-parser** for parsing json and xwww-form-urlencoded header requests, while we will use **multer** for parsing multipart/formdata.

Let us create an html form to test this out. Create a new view called **form.pug** with the following code −

```
html html
head
    title    Form    Tester
body
    form(action = "/", method = "POST")
div          label(for = "say") Say:
        input(name = "say" value = "Hi")
br        div          label(for = "to") To:
        input(name = "to" value = "Express forms")
br
      button(type = "submit") Send my greetings
```

Run your server using the following.

```
nodemon index.js
```

Now go to localhost:3000/ and fill the form as you like, and submit it.

NPM (**node package manager**)is a package manager for the JavaScript programming language. It is the default package manager for the JavaScript runtime environment Node.js. It consists of a command line client, also called npm, and an online database of public and paid-for private packages, called the npm registry.