Programming 102

Contents

[WebAPI 4](#_Toc72347599)

[HTTP 4](#_Toc72347600)

[REST APIs 4](#_Toc72347601)

[IP Addressing 5](#_Toc72347602)

[WebAPI and C.R.A.P.e. 5](#_Toc72347603)

[HTTP and C.R.A.P.e. 5](#_Toc72347604)

[Designing REST APIs 5](#_Toc72347605)

[Designing Web API 6](#_Toc72347606)

[WebAPI Parameters 6](#_Toc72347607)

[URL Parameters 6](#_Toc72347608)

[Query String Parameters 6](#_Toc72347609)

[Request Body Parameters 7](#_Toc72347610)

[Databases 7](#_Toc72347611)

[Terminology 7](#_Toc72347612)

[Databases and C.R.A.P.e. 7](#_Toc72347613)

[Database Architecture 8](#_Toc72347614)

[Indexes 8](#_Toc72347615)

[SQL 9](#_Toc72347616)

[Select Statement 9](#_Toc72347617)

[Filtering Results 9](#_Toc72347618)

[Operators 9](file:///C:\Users\Character\Documents\Users\382847\Programming%20102%20-%20Handbook.docx#_Toc72347619)

[Operators 9](file:///C:\Users\Character\Documents\Users\382847\Programming%20102%20-%20Handbook.docx#_Toc72347620)

[Ordering and Limiting Results 9](#_Toc72347621)

[Aggregate Functions 10](#_Toc72347622)

[Grouping Data 10](#_Toc72347623)

[Joining Table Data 10](#_Toc72347624)

[NULL Values 11](#_Toc72347625)

[Subqueries 11](#_Toc72347626)

[Correlated Subqueries 11](#_Toc72347627)

[Adding Data – INSERT Statement 12](#_Toc72347628)

[Updating Data – UPDATE Statement 12](#_Toc72347629)

[Deleting Data – DELETE Statement 12](#_Toc72347630)

[Database Design 13](#_Toc72347631)

[Creating Tables 13](#_Toc72347632)

[Deleting Tables – DROP TABLE Statement 13](#_Toc72347633)

[Views 13](#_Toc72347634)

[Deleting Views – DROP VIEW Statement 14](#_Toc72347635)

[Indexes 14](#_Toc72347636)

[Deleting Indexes – DROP INDEX Statement 15](#_Toc72347637)

[Accessing Databases in .NET 15](#_Toc72347638)

[SQLiteConnection 15](#_Toc72347639)

[SQLiteCommand 15](#_Toc72347640)

[SQLiteCommand Methods 16](#_Toc72347641)

[SQLiteDataReader 16](#_Toc72347642)

[Retrieving a Single Value 16](#_Toc72347643)

[Change or Delete Values 16](#_Toc72347644)

[SQL Parameters 17](#_Toc72347645)

[SQL Injections 17](#_Toc72347646)

[HTML 18](#_Toc72347647)

[Page Format and Syntax 18](#_Toc72347648)

[Common Tags 18](#_Toc72347649)

[Tables 19](#_Toc72347650)

[Forms 19](#_Toc72347651)

[Cascading Style Sheets – CSS 20](#_Toc72347652)

[Style Sheet Format 20](#_Toc72347653)

[CSS Selectors 20](#_Toc72347654)

[CSS Values 21](#_Toc72347655)

[CSS Attributes 21](#_Toc72347656)

[Padding and Margin 21](#_Toc72347657)

[CSS Rules 22](#_Toc72347658)

[Bootstrap Framework 22](#_Toc72347659)

[Colors 22](#_Toc72347660)

[Grid System 22](#_Toc72347661)

[Padding and Margin 23](#_Toc72347662)

[Forms 23](#_Toc72347663)

[Components 23](#_Toc72347664)

[JavaScript 23](#_Toc72347665)

[C# vs JavaScript 24](#_Toc72347666)

[Debugging 24](#_Toc72347667)

[jQuery Framework 25](#_Toc72347668)

[Getting Started – HTML 25](#_Toc72347669)

[Getting Started – JS 25](#_Toc72347670)

[String Interpolation 25](#_Toc72347671)

[Accessing Elements 26](#_Toc72347672)

[jQuery Functions 26](#_Toc72347673)

[jQuery Events 26](#_Toc72347674)

[Add HTML Elements 27](#_Toc72347675)

[Web Applications 27](#_Toc72347676)

[Accessing WebAPI from JavaScript 27](#_Toc72347677)

[JavaScript Object Notation 27](#_Toc72347678)

[Complex Types and WebAPI 28](#_Toc72347679)

[Sending Complex Types to WebAPI 28](#_Toc72347680)

# WebAPI

## HTTP

* All traffic on a web site runs over HTTP
* HTTP runs over IP (Internet Protocol) networks
* Every request has the following parts
  + Verb
  + URL

GET https://www.google.com

* + Headers
  + Body (optional)
* HTTP Verbs
  + **GET** Retrieve Information
  + **POST** Send Information
  + **PUT** Upload Data
  + **DELETE** Delete Data

## REST APIs

Application Programming Interfaces (APIs) give developers programmatic ways of accessing data and functionality

* REST is a way of running APIs over HTTP
* Most modern websites use REST APIs
* .NET WebAPI provides REST API functionality in .NET
* APIs are built with .NET classes that inherit from System.Web.Http.ApiController
* Methods inside the Controller class have Attributes attached that define the URL and Verb they will respond to
* Return values will be serialized appropriately depending on the Request Accept header

Add Libraries  
System.Web.Http contains WebAPI classes

Defines the Controller class - inherits from System.Web.Http.ApiController

Route Attribute defines URL for the method  
Defines the HelloWorld method returning string[]

Return a string[] with two values

using System;

using System.Web.Http;

public class DemoController : ApiController

{

[Route("hello\_world"), HttpGet]

public string[] HelloWorld()

{

return new string[] {

"Hello World!",

"Now: " + DateTime.Now

};

}

}

## IP Addressing

* All computers on an IP network have an IP Address
* IPv4 addresses are dotted quads (###.###.###.###)
* Each part of the address can range from 0 to 255
* Some addresses are defined as private
  + 10.#.#.#
  + 192.168.#.#
  + 172.16.#.#
* 127.0.0.1 is defined as the localhost or loopback address
* Every computer can be accessed via localhost

## WebAPI and C.R.A.P.e.

* Visual Studio is normally used when WebAPIs are developed
* Internet Information Services (IIS) is the Windows Web Server
* WebAPI can also be written using a self-hosted web server
* All our WebAPIs will be self-hosted running on localhost
* The WebServer.cs file in your project takes care of loading web pages and WebAPI
* Don’t make changes to this file or you may break your project
* Every request will show up in the Diagnostic Console

## HTTP and C.R.A.P.e.

* C.R.A.P.e. has support for executing HTTP requests
* Web Console allows you to make requests and see the results and headers
* HTTP Verb, Content Type, Accept and POST content can be set

## Designing REST APIs

* REST APIs should be easily understood
* Use the Verb and URL to describe what the API will do
* GET /inmate/{id}
* GET /inmate/{id}/photo
* PUT /inmate/{id}/photo
* POST /inmate/{id}/released/{date}
* DELETE /inmate/{id}
* GET /officer/{ssn}
* POST /institution/{name}/lockdown

## Designing Web API

* Group APIs together with ApiController classes
* Add RoutePrefix to Controller classes
* Add Route to each method / API

**[RoutePrefix("inmates/{id:int}")]**

public class InmateController : ApiController

{

**[Route("")]**

public string GetName(int id)

{

if(id == 382800)

return "Jack Castillo";

return "unknown";

}

}

## WebAPI Parameters

* Parameters can be passed to WebAPI methods
* Method parameters are handled just like any normal method
* WebAPI maps the input via HTTP into the method
* Parameters can be passed in three different ways
  + URL
  + Query String
  + Request Body

### URL Parameters

* URL Parameters are defined in the Route attributes
  + RoutePrefix All methods in Class have access to parameters
  + Route Parameters only available in associated method
* Request URLs must match the Route and Parameters
* Parameters can be defined with a required type

[Route("inmates/{id}")]

[Route("accounts/{acctNum}/transfer/{amt}/{toAcct}")]

[RoutePrefix("employees/{id:int}")] //id must be of type int

### Query String Parameters

* The Query String is a part of the URL where extra data can be sent to the web server
* The Query String is separated from the rest of the URL by a question mark **?**
* Each parameter in the Query String is separated by an ampersand **&**
* Each parameter has a name and a value separated by the equal sign **=**
* WebAPI will parse the Query String
* The first WebAPI Route that matches will execute
* The query string parameter name must match the method code parameter name

### Request Body Parameters

* Larger amounts of data can be passed in the body of the HTTP request
* Body data can contain .NET base types and also encoded data such as JSON
* Use the [FromBody] attribute to denote a method parameter that should be loaded from the body of the request
* There can only be one [FromBody] parameter per method

[Route("example/{x}"), HttpGet] GET http://localhost/example/5

public string urlExample(int x) { }

[Route("query"), HttpPost] POST http://localhost/query?id=5&name=John

public string queryExample(int id, string name) { }

[Route("body"), HttpPost] POST http://localhost/body Body: "stuff"

public bool bodyExample([FromBody] string notes) { }

# Databases

Applications that can store, retrieve and query data

## Terminology

* **Table** Basic unit of storage in a database
* **Row** Single record in a table
* **Column** Piece of information in a record
* **Datatype** The type of information stored in a column
* **Query** Request for information from or to the database
* **View** Prebuilt viewable query stored in the DB
* **Join** Combining the records for two or more tables
* **Primary Key** Column(s) that uniquely identify a record
* **Index** Makes queries more efficient
* **SQL** Structured Query Language

## Databases and C.R.A.P.e.

* Database Explorer
  + Tables
  + Views
  + Indexes
  + Right click on any item to get additional options
* SQL Editor
  + Write and Execute SQL queries
  + Press F9 to execute all queries
  + Select a specific query and then press F9 to just run that one
* Query Results – View results of queries

## Database Architecture

* Tables are the basic unit of storage in a database
* Every table has one or more columns that each store a specific type of data
* Individual records are stored as rows in the table
* Every table should have a column(s) that uniquely identifies a record - **Primary Key**
* Try to never store the same information in more than one table
* Databases allow developers to look up and join data between related tables
* Table rows are organized on disk by their Primary Key
* When querying the table by PK, retrievals are very fast
* Searching by other columns can be slow(er)

**OrderID**

CustomerID

OrderDate

ShipCity

ShipPostalCode

**Orders**

**CustomerID**

Name

Address

City

Phone

**Customers**

**OrderID**

**ProductID**

UnitPrice

Quantity

Discount

**OrderDetails**

**ProductID**

UnitPrice

UnitsInStock

UnitsOnOrder

**Products**

## Indexes

* Indexes make queries on non-PK columns more efficient
* Contain pointers from the columns in the index to the PK
* Take up additional storage but in most cases are worth it
* Ordered on disk by the indexed columns

# SQL

Structured Query Language – The programming language used to access most databases

## Select Statement

**\*** is not efficient and should be avoided in production

SELECT <columns> FROM <table>;

* SELECT Query Keyword
* <columns> Comma Separated list of columns or **\***
* FROM Keyword
* <table> The table you want to query

SELECT \* FROM tblUsers;

SELECT transactionID, timestamp FROM tblTransactions;

## Filtering Results

SELECT <columns> FROM <table>   
 WHERE <col> <op> <value> <AND/OR> <col> <op> <value>

* WHERE Query Filter Keyword
* <col> Column Name

### Operators

**=** = 'Crawford'  
**< <=** < 100  
**> >=** >= 10  
**IN** IN (1,2,5)  
**BETWEEN** BETWEEN 5 AND 10  
**LIKE** LIKE 'Dav%'

% is used as a wildcard in LIKE

* <op> Operator
* <value> Value to look for

SELECT \* FROM Products

WHERE ProductID = 5;

SELECT ProductName, QuantityPerUnit

FROM Products

WHERE ProductName LIKE 'Mozz%';

## Ordering and Limiting Results

SELECT <columns> FROM <table>   
 ORDER BY <col> <direction>, <col> <direction>  
 LIMIT <numResults>;

* ORDER BY Order Keyword
* <col> Column Name
* <direction> Ascending (ASC) or Descending (DESC)
* LIMIT LIMIT Keyword
* <numResults> Number of results to return

SELECT OrderID, Freight FROM Orders

ORDER BY Freight DESC

LIMIT 5;

## Aggregate Functions

Functions are available to aggregate (combine) data

* **AVG** Average
* **COUNT** Counts number of results
* **MAX** Returns maximum value
* **MIN** Returns minimum value
* **SUM** Calculates the sum of all returned values

## Grouping Data

SELECT <columns>, <aggFunc>(<col>) AS <aggName>   
 FROM <table>  
 GROUP BY <col>

HAVING <aggName> <op> <value>;

* <aggFunc> Aggregate function to use
* <col> Column to run function on
* AS Rename Keyword
* <aggName> Name of column to display function results
* GROUP BY Group Keyword
* <col> Column Name to group by
* HAVING Filter Keyword (like WHERE but with aggregates)
* <op> Operator
* <value> Value to look for

SELECT OrderID, SUM(Quantity) AS totItems FROM OrderDetails

GROUP BY OrderID

HAVING totItems > 1;

## Joining Table Data

* When two tables have related information a JOIN can be used
* JOINs merge rows from two tables into one row by matching values in key fields
* Key fields are often the Primary Key but not always
* Tables can be joined in two ways
  + INNER JOIN – Combine rows only when matching key values exists in **both** joined tables
  + OUTER JOIN – Combine rows if matching key values exists in **either** of the joined tables
* When joining rows in tables you must reference the columns to join the data on
* Normally the columns will be the Primary Key column(s) of the tables

**CustomerID**

Name

Address

City

Phone

**Customers**

**OrderID**

*CustomerID*

OrderDate

ShipCity

ShipPostalCode

**Orders**

SELECT \* FROM Orders

INNER JOIN OrderDetails ON Orders.OrderID = OrderDetails.OrderID;

SELECT o.OrderID, o.Freight, io.ExciseTax

FROM Orders o

LEFT OUTER JOIN InternationalOrders io ON o.OrderID = io.OrderID

WHERE o.OrderID = 10262;

## NULL Values

* When dealing with OUTER JOINs you may get back NULL values
* NULL is returned anytime SQL can’t find a matching record
* NULL values cannot be used in any operations
* Use IFNULL to convert a NULL value into something you can work with
* IFNULL takes two parameters
  + Value to test
  + Value to return if test value is NULL

SELECT o.OrderID, o.Freight, io.ExciseTax,

o.Freight + **IFNULL(**io.ExciseTax, 0**)** AS sumFET

FROM Orders AS o

LEFT OUTER JOIN InternationalOrders AS io

ON o.OrderID = io.OrderID

WHERE o.OrderID = 10262;

## Subqueries

A Query can be passed as the value to an operator

SELECT \* FROM Orders

WHERE CustomerID IN (SELECT CustomerID

FROM Customers

WHERE PostalCode = '98128');

## Correlated Subqueries

A Query can be passed as a column in a query and executed for every row in the result set

SELECT o.OrderID, o.Freight, io.ExciseTax AS ExciseTax,

(SELECT SUM(od.UnitPrice \* od.Quantity)

FROM OrderDetails AS od

WHERE od.OrderID = o.OrderID) AS odSum

FROM Orders o

LEFT OUTER JOIN InternationalOrders io

ON o.OrderID = io.OrderID

WHERE o.OrderID = 10262;

## Adding Data – INSERT Statement

When running an INSERT exclude any AUTOINCREMENT columns from the list of columns you pass

INSERT INTO <table> (<col>, <col>, ..., <col>)

VALUES (<val>, <val>, ..., <val>);

* INSERT INTO Insert Keyword
* <table> Table name to insert into
* <col> The columns you want to insert data into
* VALUES Keyword
* <val> The values to match each <col>

INSERT INTO Categories (CategoryName, Description)   
 VALUES ('Canteen', 'Expensive food');

## Updating Data – UPDATE Statement

UPDATE <table> SET <uCol> = <newVal>, <uCol> = <newVal>

WHERE <col> <op> <value>;

* UPDATE Update Keyword

UPDATE without WHERE will update <uCol> in **ALL ROWS**

* <table> Table name to update
* <uCol> The Column you want to update
* <op> Operator
* <uVal> The new value for <uCol>
* <col> Column Name
* <op> Operator
* <value> Value to look for

UPDATE Orders

SET RequiredDate = '2020-05-01 17:04:02',   
ShippedDate = CURRENT\_TIMESTAMP, Region = 'SC'

WHERE OrderID = 10248;

## Deleting Data – DELETE Statement

DELETE FROM <table> WHERE <col> <op> <value>;

* DELETE Delete Keyword

DELETE without WHERE will delete **ALL ROWS**

* <table> Table name to update
* <col> Column Name
* <op> Operator
* <value> Value to look for

DELETE FROM Orders WHERE OrderID = 10248;

## Database Design

* Efficient databases store the minimal amount of data required
* Duplication should be avoided unless absolutely necessary
* Joins allow developers to access related rows and combine data at query time
* Store important parts of the data in separate tables

## Creating Tables

* Most SQL engines can change the table layout after creation
* SQLite can’t, so think about what you are creating first
* Every table should have a Primary Key
* Very often the PK will be AUTOINCREMENT

Options

PRIMARY KEY  
AUTOINCREMENT   
NOT NULL

**Data Types**

varchar(maxSize)

text

datetime

money

integer

bit

CREATE TABLE [<tableName>] (

[<col>] <datatype> <option> <option>,

[<col>] <datatype> <option>,

[<col>] <datatype>

);

* CREATE TABLE Table Creation Keyword
* <col> Column Name
* <datatype> Datatype of the column
* <opt> Option

CREATE TABLE [Test] (

[TestID] integer PRIMARY KEY AUTOINCREMENT NOT NULL,

[FirstName] varchar(50) NOT NULL,

[LastName] varchar(50) NOT NULL,

[Phone] varchar(20),

[Email] varchar(200)

);

## Deleting Tables – DROP TABLE Statement

There will be no confirmation. The table and **all its data** will be **gone**.

DROP TABLE <tableName>;

* DROP TABLE Drop Keyword
* <tableName> Table name to update

DROP TABLE Orders;

## Views

* Views allow you to save queries in the database
* Queried just like a normal table
* Anytime you have a query that is used often consider creating a view
* Keep your application code cleaner because SQL code is in the database instead of your application

**CREATE VIEW [vTopCustomers] AS**

SELECT c.CustomerID, c.ContactName, COUNT(OrderID)

FROM Orders o

INNER JOIN Customers c ON o.CustomerID = c.CustomerID

WHERE Freight > 50

GROUP BY c.CustomerID

ORDER BY COUNT(o.CustomerID) DESC

LIMIT 5;

SELECT \* FROM vTopCustomers;

## Deleting Views – DROP VIEW Statement

There will be no confirmation. The view will be gone but the associated tables will not be touched.

DROP VIEW <viewName>;

* DROP VIEW Drop Keyword
* <viewName> View name to delete

DROP VIEW vTopCustomers;

## Indexes

* Indexes can be incredibly important in your queries
* When there is a lot of data in your table queries will begin to get very slow on non-indexed columns
* Indexes allow the database engine to more quickly run queries on non-primary key columns
* None of our tables will be large enough to really see any performance increases with indexes
* Wait until you understand what types of queries will be run on your data before creating indexes
* Determine what columns will need to be indexed and in what direction
* Create Indexes then test
* Too many indexes can hurt performance more than help

CREATE INDEX [IX\_<tableName>\_<description>] ON [<tableName>]   
(

[<col>] <direction>,

[<col>] <direction>  
);

* CREATE INDEX Index Creation Keyword
* <tableName> Table to index
* <col> Column to index
* <direction> Ascending (ASC) or Descending (DESC)

CREATE INDEX [IX\_Customers\_Region] ON [Customers]

(

[Region] ASC

);

## Deleting Indexes – DROP INDEX Statement

There will be no confirmation. The Index will be gone but the associated tables will not be touched.

DROP INDEX <indexName>;

* DROP INDEX Drop Keyword
* <indexName> Index name to drop

DROP INDEX IX\_Customers\_Region;

# Accessing Databases in .NET

## SQLiteConnection

.NET Class that handles database connections

* SQLiteConnection must be created in a using block
* The string passed into the SQLiteConnection defines where the SQLite database file resides
* We will always just use "app.db"
* Call SQLiteConnection.Open() before doing anything else

using(var conn = new SQLiteConnection("Data Source=app.db"))

{

conn.Open();

...

}

## SQLiteCommand

.NET Class that transmits a SQL command to the database

* SQLiteCommand must be created in a using block
* SQLiteCommand takes two arguments
  + A SQL Command string
  + The SQLiteConnection to use with the command
* Parameters can be added to the SQLiteCommand
* Multi-line queries can be used by adding an ampersand @ before the string

using(var conn = new SQLiteConnection("Data Source=app.db"))

{

conn.Open();

using(var cmd = new SQLiteCommand("SELECT lastName FROM Employees", conn);

{

...

}

}

### SQLiteCommand Methods

|  |  |  |
| --- | --- | --- |
| ExecuteReader() | Multiple Rows | using(SQLiteDataReader rdr = cmd.ExecuteReader()) |
| ExecuteScalar() | Single Value | object obj = cmd.ExecuteScalar(); |
| ExecuteNonQuery() | Change/Delete | int numRowsAffected = cmd.ExecuteNonQuery(); |

## SQLiteDataReader

.NET Class that can read multiple rows from a database query

* SQLiteDataReader must be created in a using block
* Almost always created with SQLiteCommand.ExecuteReader()
* Call SQLiteDataReader.Read() in a while loop to iterate through each row
* SQLiteDataReader acts like a Dictionary<string, object>

using(var conn = new SQLiteConnection("Data Source=app.db"))

{

conn.Open();

using(var cmd = new SQLiteCommand("SELECT lastName FROM Employees", conn))

using(var rdr = cmd.ExecuteReader())

{

while(rdr.Read())

{

var lastName = rdr["lastName"].ToString();

}

}

}

## Retrieving a Single Value

* SQLiteCommand.ExecuteScalar() returns a single value
* When just need one column from one record ExecuteScalar() is best
* ExecuteScalar() returns an object

using(var conn = new SQLiteConnection("Data Source=app.db"))

{

conn.Open();

using(var cmd = new SQLiteCommand(@"SELECT OrderID

FROM Orders

ORDER BY OrderID DESC

LIMIT 1", conn))

var mostRecentOrder = int.Parse(cmd.ExecuteScalar().ToString());

}

## Change or Delete Values

* SQLiteCommand.ExecuteNonQuery() returns the number of rows affected by the SQL command

## SQL Parameters

* Use Parameters to pass data into the query
* Define parameters in the SQL statement with @parameterName
* Each defined parameter in the SQL must have a matching SQLiteCommand.Parameters.AddWithValue statement in your code with the parameter name and value
* Parameters can be used with any type of query (SELECT, UPDATE, DELETE, etc.)

using(var conn = new SQLiteConnection("Data Source=app.db"))

{

conn.Open();

using(var cmd = new SQLiteCommand(@"UPDATE Employees

SET LastName = @name

WHERE employeeID = @empID", conn))

{

cmd.Parameters.AddWithValue("name", newName);

cmd.Parameters.AddWithValue("empID", employeeID);

int numRowsAffected = cmd.ExecuteNonQuery();

}

using(var cmd = new SQLiteCommand(@"SELECT LastName FROM Employees

WHERE employeeID = @empID", conn))

{

cmd.Parameters.AddWithValue("empID", employeeID);

return cmd.ExecuteScalar().ToString();

}

}

## SQL Injections

* Parameters sanitize the data coming into SQL
* If incoming data is not sanitized, SQL Injections can occur
* Before parameter use was common, SQL Injection attacks were widely seen on the Internet
* All kinds of nasty stuff can happen when input is not checked before getting sent to SQL
* Unless there is a **very** good reason always use SQL parameters in your code

# HTML

Hyper-Text Markup Language

* All elements on the page are Tags
* Tags must be opened and closed
* Content of the tag goes between the open and closing tag
* Tags can have attributes that store additional information
* Nesting / Indentation is very important for readability

<tag attribute1="value" attribute2="value">  
 content  
</tag>

## Page Format and Syntax

* Every HTML page has a specific set of tags at the beginning and end

<!doctype html>

<html lang="en">

<head></head>

<body>

<p id="myParagraph">

Paragraph One

</p>

<div class="shadow">

<p>Stuff</p>

</div>

<p style="color: blue">

Blue Paragraph

</p>

</body>

</html>

* These tags tell the browser where to look for resources and content
* All content goes inside the <body> tag
* Tags can be nested inside other tags
* Any nested tags will move and grow with   
  their parent element
* Nested tags should always be indented one   
  tab from their parent
* All tags can have attributes applied
* Attributes contain additional information   
  about the tag
* Some attributes can only be applied to specific tags

## Common Tags

|  |  |  |
| --- | --- | --- |
| **<a>** | Anchor / Link | <a href="https://www.google.com" name="g">Google</a>   * href The destination URL of the link * name The name of the anchor |
| **<p>** | Paragraph | <p>Programming 101 is really awesome!</p> |
| **<img>** | Image | <img src="/images/p101.jpg" alt="logo" />   * src The source URL for the image to load * alt A text description of the image |
| **<span>** | Inline Text | <span>Any text can go in here</span> |
| **<strong>** | Bold | <p>I really like <strong>HTML</strong></p> |
| **<em>** | Italics | <p>Programming is <em>really</em> fun!</p> |
| **<h#>** | Headings 1-6 | <h3>Heading 3 is about as big as you should go</h3> |
| **<ul>** | Unordered List | <ul>  <li>Item 1</li>  <li>Item 2</li> </ul>  <ol>  <li>1st Item</li>  <li>2nd Item</li> </ol> |
| **<ol>** | Ordered List |
| **<li>** | List Item |
| **<hr/>** | Horizontal Line | <p>Programming<hr/>101</p> |
| **<br/>** | Line Break | <p>Programming<br/>101</p> |
| **<blockquote>** | Quote | <blockquote>He said he didn’t do it!</blockquote> |
| **<div>** | Panel | <div>Divs are mainly used for layout</div> |
| Tables |  | <table>  <thead>  <tr>  <th>First</th>  <th>Last</th>  </tr>  </thead>  <tbody>  <tr>  <td>Jack</td>  <td>Castillo</td>  <tr>  </tbody>  </table> |
| **<table>** | Table |
| **<thead>** | Table Head |
| **<tbody>** | Body |
| **<tr>** | Row |
| **<td>** | Column |
| **<th>** | Column Header |
| Forms |  | <form>  <label>Name</label>  <input type="text"></input>  <label>Password</label>  <input type="password"></input>  <label>Remember Me</label>  <input type="checkbox"></input>  <br/>  <label>Account Type</label>  <input type="radio" name="acctT">Saving</input>  <input type="radio" name="acctT">Checking</input>  <select>  <option>First Option</option>  <option>Second Option</option>  </select>  <button type="button">Create Acct</button>  </form> |
| **<form>** | Form Wrapper |
| **<label>** | Label for input |
| **<input>** | Input Box |
| * type="text" * type="password" * type="email" * type="checkbox" * type="radio" | |
| **<select>** | Dropdown List |
| **<option>** | Select Option |
| **<button>** | Button |

## Cascading Style Sheets – CSS

Allows web developers to apply styles to HTML elements and pages in a standardized way

CSS can be added to a page in three ways

* <style> tag in <head>
* <link> tag in <head>
* Inline on elements with style and class attributes

<!doctype html>

<html lang="en">

<head>

<**style** type="text/css">

p, h3 {

color: green;

font-weight: bold;

}

.extraFluffy

{

padding: 20px;

}

</**style**>

<**link** href="/css/HW.css" rel="stylesheet" />

</head>

<body>

<ul **style="margin-left: 0"**>

<li>Welcome</li>

<li>To Programming 102</li>

</ul>

<p class="extraFluffy">More padding here</p>

</body>

</html>

## Style Sheet Format

#header {  
 padding-top: 5px;  
}

elementSelector {

css-property1: value;

css-property2: value;

}

/\* CSS Comment \*/

## CSS Selectors

* **tagName** Applies to all tags

/\* Set all <p> tags inside <li> tags inside   
<ul> with class="list" to be blue \*/

ul.list li p {  
 color: blue;  
}

* **#id** Specific tag with id
* **.class** Tag with class

Selectors can be chained to describe elements nested inside other elements

## CSS Values

|  |  |  |
| --- | --- | --- |
| **Color** | hex color code or name; | color: #5d5d5d; |
| **Sizing** | px or em or % | height: 5px; padding: 2em;  font-size: 90%; |
| **Padding / Margin** | all sides top/bottom left/right top right bottom left | padding: 5px; margin: 20px 0px; padding: 1px 2px 2px 1px; |

## CSS Attributes

|  |  |
| --- | --- |
| **background** | background: url(/images/bg.jpg); |
| **color** | color: #6bdcea; |
| **display** | display: none; |
| **height** | height: 30px; |
| **width** | width: 100%; |
| **margin** | margin: 5px; |
| **padding** | padding: 10px 50px; |
| **font-family** | font-family: Arial; |
| **font-size** | font-size: 15px; |
| **font-style** | font-style: italic; |
| **font-weight** | font-weight: bold; |
| **text-align** | text-align: left; /\* left center right \*/ |
| **text-decoration** | text-decoration: underline; /\* overline \*/ |
| **position** | position: absolute; /\* relative fixed \*/ |
| **top** | top: 5px; |
| **left** | left: 50%; |
| **border** | border: 1px solid #429fc9; |
| **list-style** | list-style: none; |

## Padding and Margin

**Padding** The space between the content of an element and its borders

**Margin** The space between an elements borders and its neighbors

## CSS Rules

* Styles are processed from the top of the CSS file to the bottom
* Effects cascade from one matching definition to the next matching definition
* Later definitions can override previous ones

# Bootstrap Framework

* Many developers rely on CSS Frameworks to help design their sites
* Most frameworks provide prebuilt styles, controls and layout functionality
* Focus on the design and functionality instead of complex CSS
* Bootstrap Framework developed by Twitter
* Mobile first / Responsive
* Bootstrap uses CSS class names to apply styling
* Most components will have two or more classes applied

## Colors

* Bootstrap uses color classes for many items

|  |  |
| --- | --- |
| .primary .secondary .success .danger .warning | .info .light .dark .white |

* Class names are based on context not actual color
* Normally the CSS class name has a prefix

<div class="alert alert-primary">

A simple primary alert

</div>

## Grid System

* Bootstrap utilizes a 12 column grid system for layout
* Columns widths can set to render differently   
  on different size devices

<div class="row">

<div class="col">Col 1</div>

<div class="col">Col 2</div>

</div>

* Depending on the screen size the developer   
  can set columns to stack vertically   
  instead of sitting side by side
* Columns can be offset from one another
* Every set of columns is wrapped in <div class="container">
* Each row is wrapped in <div class="row">
* Each column is wrapped in <div class="col">

|  |  |
| --- | --- |
| **.col** | Equal width columns on all screen sizes |
| **.col-{size}** | Equal width columns at {size} and larger |
| **.col-{size}-{#}** | # columns wide at {size} and larger |
| **.offset-{size}-{#}** | Offset # columns at {size} and larger |
| **.order-{size}-{#}** | Change render order at {size} and larger |

## Padding and Margin

* Bootstrap has predefined classes to adjust margin and padding
* The classes are named using the format
  + {property}{sides}-{0,1,2,3,4,5}
  + {property}{sides}-{breakpoint}-{0,1,2,3,4,5}
* Where property is one of:
  + m – Margin

.pt-5 Padding Top 5

.mr-0 Remove Margin Right

.pb-sm-2 Padding Bottom 2 on   
small and larger

* + p – Padding
* Where sides is one of:
  + l – Left r – Right x – Left AND Right
  + t – Top b – Bottom y – Top AND Bottom

## Forms

<form>

<div class="form-group">

<label>Username</label>

<input type="text" class="form-control" />

</div>

<div class="form-group">

<label>Username</label>

<input type="password" class="form-control" />

</div>

<button type="button" class="btn btn-info">Submit</button>

</form>

## Components

* Bootstrap includes templates for components
* Pre-styled HTML tags and classes
* Can completely change the default look and feel
* Utility classes are also available

# JavaScript

* JavaScript (JS) is an interpreted language; no compiler needed
* JS is a derivative of the Java language which is very similar to C#
* Web Browsers have built-in interpreters that optimize and execute JS
* Website JS can access and manipulate the Document Object Model (DOM), which is a programmatic representation of all the elements on the HTML page

## C# vs JavaScript

**C#**

var myStr = "here is my string";

var myArr = new string[] {"Ja","Kb"};

public void Add(int x, int y)

{

return x + y;

}

for(var i = 0; i < myArr.Length; i++)

{

Console.WriteLine(myArr[i]);

}

**JavaScript**

var myStr = "here is my string";

var myAarr = ["Ja","Kb"];

function Add(x, y)

{

return x + y;

}

for(var i = 0; i < myArr.length; i++)

{

console.log(myArr[i]);

}

* JS is **not** a strongly typed language unlike C#

var a = [ "asdf", 13, 52.2 ]; VALID  
a[0] + 50 asdf50  
a[1] + 50 63  
a[1] + "50" "1350"

* Nameless functions

$(document).ready(function() { alert("hi!"); });

* “Magic” Arguments

$("#btnSubmit").click(handleSubmit);  
function handleSubmit(event) {  
 alert(event);  
}

* LOTS of JSON

$.ajax({  
 url: "/api/values",  
 success: function (data, status) {  
 alert("data");

}  
});

## Debugging

* console.log(message, color);
  + Logs a message to the debugging window with an optional Bootstrap color
  + You can log any type of object
* alert(message);
  + Opens a message box in the browser
* JS syntax errors will also be logged in the debugging window
* Add the <!—Logging--> comment to your HTML so this stuff works

## jQuery Framework

* jQuery has become popular because it is easy to use and has a lot of functionality
* Developers can select specific HTML elements by ID, class and other attribute values much like CSS selectors
* Utility methods are also provided to run code on specific user actions like button clicks and hovering over elements

## Getting Started – HTML

1. Create HTML file
2. Add JS file for your directory
3. Add <script> tag pointing to jQuery
4. Add <!--Logging--> comment to enable C.R.A.P.e debugging
5. Add <script> tag pointing to your JS file
6. Add HTML elements

</head>

<body>

<p>Hello World!</p>

<script src="/js/jquery.js" type="text/javascript"></script>

<!--Logging-->

<script src="app.js" type="text/javascript"></script>

</body>

</html>

## Getting Started – JS

1. Add $(document).ready(function() { });
2. Add events to HTML elements
3. Add functions as needed
4. Refresh your page and pray everything worked

$(document).ready(function() {

console.log("Loaded!");

$("#btnAddItem").click(function() {

console.log("button clicked");

});

});

## String Interpolation

* String interpolation is very similar to string formatting in C#
* Interpolation allows you to build a string and include code inside the string
* Use the format **`${**code**}`** to create an interpolated string

var name = "Jack";

console.log(**`**Hello World ${3 + 5} I’m ${name}**`**);

## Accessing Elements

* jQuery can access almost any element in the DOM
* Use CSS element selectors to select one or more elements
* Use the $ function to access an element
* Elements can be accessed by

var byTagName = $("a");

var byID = $("#wrapper");

var byClass = $(".col");

* + Tag Name
  + ID
  + CSS Class
* Once you have a reference to one or more elements you can call jQuery functions to use them

## jQuery Functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Change CSS** | addClass(className) | **Get/Set Contents** | html() html(content) |
|  | removeClass(className) |  | text() text(content) |
|  | toggleClass(className) |  | val() val(newValue) |
|  | css(propName, value) |  | prop(propName) |
| **Add Contents** | append(content) | **Show / Hide Elements** | hide() |
|  | prepend(content) |  | fadeIn() slideIn() |
|  | after(content) |  | fadeOut() slideOut() |
|  | before(content) |  | fadeToggle() slideToggle() |

## jQuery Events

* Functions to handle events on the page are also included in jQuery
* Developers can “listen” for event to occur and then execute code when they do
* Inside the event code the element that triggered the event can be accessed using $(this)

|  |  |  |
| --- | --- | --- |
| **Mouse Click** | click() | $(".btnEvents").click(function() { }); |
| **Mouse Move** | mousemove() | $("div").mousemove(function(e) {  console.log(**`**${e.clientX},${e.clientY}**`**);  }); |
| **Mouse Button Down / Up** | mousedown() mouseup() | $("div").mousedown(function(e) {  console.log(**`**${e.clientX},${e.clientY}**`**);  }); |
| **Mouse Move** | mousemove() | $("div").mousedown(function(e) {  console.log(**`**${e.clientX},${e.clientY}**`**);  }); |
| **Input value changed** | change()  change() only fires when the element looses focus | $("input").change(function(e) {  var newValue = $(this).val();  alert(newValue);  }); |
| **Key Press / Down / Up** | keypress() keyup() keydown() | $("div").keydown(function(e) {  console.log(e.keyCode);  }); |

## Add HTML Elements

* jQuery can very easily add elements to the page
* By using jQuery methods we can add HTML to whatever element we selected
* String Interpolation makes this much cleaner

var items = ["Jack", "James", "Sam", "Bob"];

for(var i = 0; i < items.length; i++)

$("myList").append(**`**<li>${items[i]}</li>**`**);

# Web Applications

By combining the tools we have, we can create interactive, data-driven web applications

## Accessing WebAPI from JavaScript

* jQuery has functions for calling REST APIs
* To make a simple GET or POST request you can call
* $.get(url, function(data) { });
* $.post(url, function(data) { });

$("#btnAddItem").click(function() {

var val = $("#txtInput").val();

$.get("/api?content=" + val, function(data) {

$("#myList").append(`<li>${data}</li>`);

$("#txtInput").val("");

});

});

## JavaScript Object Notation

var dob = "1980-04-02";

var obj = {

firstName: "Jack",

lastName: "Castillo",

id: 382000,

birthDate: dob

};

obj["room"] = "254";

obj["firstName"] = "Julio";

var str = JSON.stringify(obj);

var json = JSON.parse(str);

* JSON stores complex types
* Variables and hardcoded values can be used
* Accessed like a Dictionary<string, object> to Get, Set and Add properties and values
* JSON.stringify() Serialize
* JSON.parse() Deserialize

## Complex Types and WebAPI

* WebAPI allows us to send and receive complex types
* We can create Plain Old CLR Objects (POCOs) in our WebAPI

public class CalculatorPOCO

{

public int X;

public int Y;

public string Operation;

public int Result {

get {

if(Operation == "+")

return X + Y;

...

}

}

}

* POCOs can be shared between
  + Server (WebAPI)
  + Client (Browser/jQuery)
* Most of the time POCOs won’t have any functionality but it may make sense to add some
* Try to keep logic related to the POCO inside the POCO itself if possible
* Logic only executes on the server
* Executes when the POCO is serialized back to the client

## Sending Complex Types to WebAPI

$("#btnCalculate").click(function() {

var json\_poco = {

X: $("#txtX").val(),

Y: $("#txtY").val(),

Operation: $("#drpOp").val()

};

$.ajax({

url: "/api",

type: "POST",

contentType: "application/json",

dataType: "json",

data: JSON.stringify(json\_poco),

success: function (data, status) {

$("#divResult").text(data.Result);

}

});

});

1. Create JSON object
2. Call $.ajax
3. Set URL
4. Set HTTP Verb (Type)
5. Set contentType header
6. Set dataType
7. Serialize JSON
8. Handle success()