MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

JavaScript Web Storage: IndexedDB

158.258 Web Development

Kuda Dube
College of Sciences
Massey University

Revised: 2022-08-16

Reading Materials

NOTE: *The course textbook does not have materials for this topic.*

These slides are based on:

Mark J Collins (2017), *Pro HTML5 with CSS, JavaScript and Multimedia*, Apress.
ISBN: 978-1-4842-2463-2: Chapter 25, pp.467 - 494

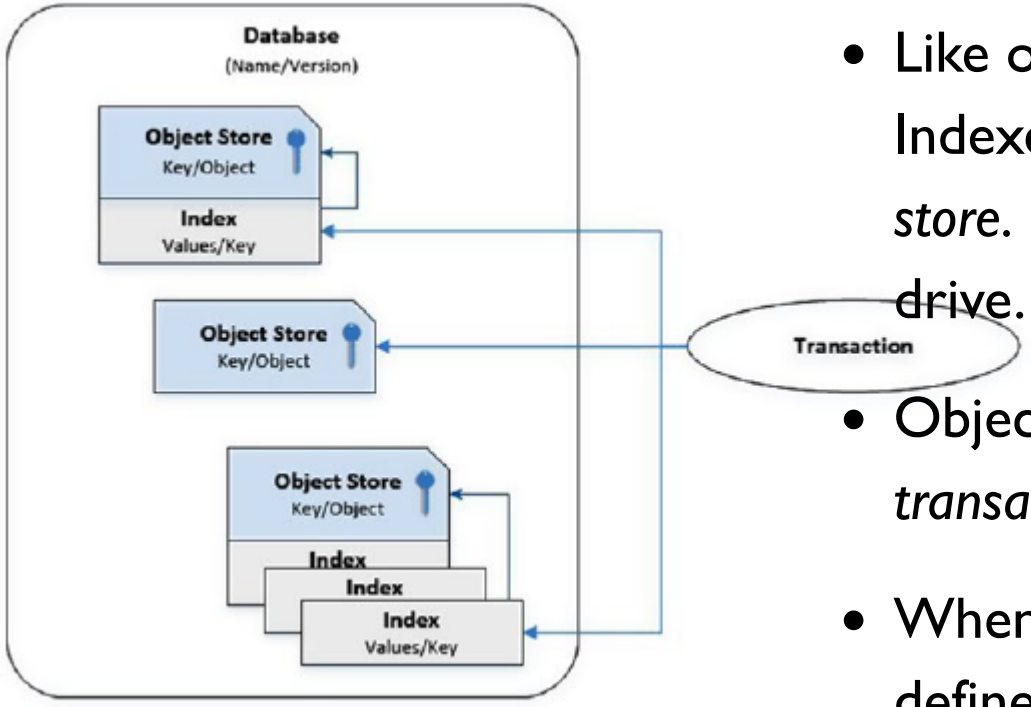
Learning Objectives

- To use Indexed DB to store and use data on the Web client-side;
- To explore capabilities of Indexed DB;
- To rewrite the Chessboard to use Indexed DB as a way to explore Web storage capabilities:
  - *Creating object stores to define the positions of each piece; and*
  - *Manipulating this data as the pieces are moved.*

Introduction: IndexedDB Web Storage

- Browsers are evolving to provide more functionality;
- Client-side storage and data manipulation is one modern functionality that is in demand;
- Indexed DB technology:
  - *Emerged to satisfy this demand;*
  - *A JavaScript application programming interface (API) provided by Web browsers for managing a NoSQL database of* ***JSON objects***
  - *A standard maintained by the World Wide Web Consortium (W3C).*

IndexedDB Entities



Database Entities of the Indexed DB

- Like other databases, the data in the Indexed DB is placed in a *persistent data store*. In this case, it's on the local hard drive. The data is permanent.
- Object stores are accessed through a *transaction object*.
- When creating the transaction, you must define its scope. This indicates which object stores it will reference and whether it will be reading or writing data to the database.

(Mark J Collins, 2017: Chapter 25, p.468)

Object Stores and Keys

- The **object store** (OS) is the *primary unit* of storage in Indexed DB;
- The OS is a collection of objects referenced by a **key**;
- The three types of keys in Indexed DB are: inline, out-of-line and system-generated keys.

Three Types of Keys in IndexedDB

1. **Inline Yey** — one of the object properties serving as the key:

```
// Using an inline key
let typeStore = db.createObjectStore("pieceType", { keyPath: "id" });
typeStore.add(pieceType);
```

   In this case **id** property with unique values act as the *inline key*.

2. **Out-of-line Key** — a user-specified key added when the object is being stored:

```
// Using an out-of-line key
let sampleStore = db.createObjectStore("sample", { });
sampleStore.add(sample, 5);
```

   In this case, the user-specified key is **5**.

3. **System-generated Key** — the OS automatically asign incremental key

values to objects being stored:

```
// Using a key generator
let pieceStore = db.createObjectStore("piece", { autoIncrement: true });
pieceStore.add(piece);
```

In this case, user just stores objects without specifying a key and the system automatially assign the key to each object.

(Mark J Collins, 2017: Chapter 25, pp.468-469)

Indices on Objects in IndexedDB

- In Indexed DB, one could create more than one index on the object store;
- Indices are used to find specific object or a collection of them very fast;
- Each index is a collection of name-value pairs in which the value is the key into the object store;
- The following code fragent illustrates an index created over the **lastname** property of an object — the index entry will be the *lastname* and corresponding key to that object:

```
// Create an index on the lastName property
customerStore.createIndex("lastName", "id", { unique: true });

// Get the index
let index = customerStore.index("lastName");
index.get(lastName).onsuccess = function(); // get the object
index.getKey(lastName).onsuccess = function(); // get the key
```

In this case, the parameter **id** of the **createIndex()** function specifies the **key path**. Same as for **inline keys.**

(Mark J Collins, 2017: Chapter 25, p.469)

Relationships between Object Stores

- No support for object store relationships in Indexed DB;
- E.g., **foreign key constraints**, though possible through a hack, are not enforced by the database system;
- Join operations are also not supported

Defining/Creating/Openning Database (1)

When you open a database, you need to implement the following three ***event handlers****:*

1. **onsuccess:** The database is opened; do something with it;
2. **onerror:** An error occurred, likely an access issue; and
3. **onupgradeneeded:** The database needs to be created or upgraded.

(Mark J Collins, 2017: Chapter 25, p.469)

Defining/Creating/Openning Database (2)

It is important to note the following:

- When opening a database, if it doesn't exist, it will be created automatically. The **onupgradeneeded** event will be raised.
- You must implement an **event handler** for the **onupgradeneeded** event, which will create object stores and populate them with any default data.
- This is the only time when you are allowed to alter the database structure.
- The important thing to remember is that the **onupgradeneeded** event is fired before the **onsuccess event.**

(Mark J Collins, 2017: Chapter 25, p.469)

Asynchronous Processing in IndexedDB

- All Indexed DB operations are done asynchronously;
- The general pattern of operation of Indexed DB occurs in the following order:

    1. *Call database operation, e.g., open database of retrieve objects;*
    2. *Request object is returned;*
    3. *If request was successful, then **onsuccess handler** is invoked. Otherwise, **onerror handler** is invoked;*
    4. *Result of operation or method call is passed through event object.*

Example: Making three IndexedDB requests

For complex processing that requires several database calls, you'll need to be careful to nest the event handlers and consider when they are executed.

For example, if you needed to make three database requests, your code might look like this:

```
let request = dbCall1()
request.onsuccess = function (e1) {
    f1(e1.target.result);
    dbCall2().onsuccess = function (e2) {
            f3(e2.target.result, e1.target.result);
            dbCall3().onsuccess = function (e3) {
                f5(e3.target.result, e2.target.result, e1.targe
            }
            f4(e2.target.result);
        }
        f2(e1.target.result);
}
request.onerror = function(e) {
```

Explanation of the execution process:

1. This code calls dbCall1(), dbCall2(), and dbCall3(), in that order — processed sequentially.
2. dbCall2() will not start until dbCall1() has completed, and only if it was successful.
3. Each call provides an

```
alert("The call failed");
}
```

onsuccess event handler, which makes the next call.

4. If the first call fails, an alert is raised.
5. What may be unexpected is the order that the non-database calls are executed.
6. Database calls return immediately, and the event handler is called later, when the operation has completed.
7. As soon as the call to

dbCall2() is made, the function returns, and f2() will be executed.

8. Later, the dbCall2() completes, its event handler is called, and f3() is executed.

(Mark J Collins, 2017: Chapter 25, p.470)

Using Transaction with IndexedDB

- All data access, both reading and writing, is done within a transaction, so you must first create a transaction object.
- When the transaction is created, you specify its scope, which is defined by the object stores it will access.
- You also specify the mode (read-only or read-write).
- You can then obtain an object store from the transaction and get or put data from/into the store like this:

```
let xact = db.transaction(["piece", "pieceType"], "readwrite");
let pieceStore = xact.objectStore("piece");
```

- For read-write transactions, the data changes are not committed until the transaction completes. "*When does a transaction complete?*" — when there are

no more outstanding requests for it.

Transactions: Request-based Modus Operandi

- Everything is request based — you make a request and then implement an event handler to do something when it finishes.
- If that event handler issues another request on that transaction, then the transaction stays alive.
- This is another important reason for nesting the event handlers.
- If you end an event handler without issuing another request, the transaction will complete, and all changes are committed.
- If you try to use the transaction after that, you will get a TRANSACTION_INACTIVE_ERR error.

Transaction Scope Issues

- Important thing to remember — read-write transactions cannot have overlapping scopes.
- If you create a read-write transaction, you can create a second one as long as they don't both include some of the same object stores.
- If they have overlapping scopes, you must wait for the first transaction to complete before creating the second one.
- Read-only transactions, however, can have overlapping scopes.

Example: Using Indexed DB in Creating the Chessboard

This example will also be used as a tutorial exercise during which the challenge is to study the code in pairs and explain how the Indexed DB is being used.

Download and study the Example Code: **10-local-storage-example-chessboard.zip**

This archive contains the following files:

1. **chess.html** — the Web page;
2. **chess.js** — the JavaScript program that draws the chessboard with pieces and then animates or simulates the chess moves that lead to checkmate;
3. **data.js** — the chessboard static data to be loaded into the Indexed DB

Step 1: Create Web Project: chess.html & chess.js

Create a new web page called **chess.html**, using the basic markup:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Chapter 25 - Chess</title>
        <script src="Data.js" defer></script>
        <script src="Chess.js" defer></script>
    </head>
    <body>
    </body>
</html>
```

Then create the **chess.js** script file using the following code:

```
"use strict";
```

Add a div element in the empty body element of the **chess.html** file, using the following markup:

```
<div>
    <canvas id="board" width ="600" height ="600
        Not supported
    </canvas>
</div>
```

Add an Images folder in the web project. In this folder, place the chess piece images from 10-local-storage-chess-pieces.zip

Step 2: Drawing the Canvas

- Now you'll design the canvas element using JavaScript.
- The initial design will just draw an empty board.
- You'll then add the chess pieces later.
- Refer to the Lecture on *HTML5 Canvas* for more explanation about working with a canvas element.
- Add the code from the listing here to the **chess.js** script file.

```
function drawBoard() {
    chessContext.clearRect(0, 0, 600, 600);

    let gradient = chessContext.createLinearGrad
    gradient.addColorStop(0, "#D50005");
    gradient.addColorStop(0.5, "#E27883");
    gradient.addColorStop(1, "#FFDDDD");

    chessContext.fillStyle = gradient;
    chessContext.strokeStyle = "red";

    // Draw the alternating squares
    for (let x = 0; x < 8; x++) {
        for (let y = 0; y < 8; y++) {
            if ((x + y) % 2) {
                chessContext.fillRect(75 * x, 75
            }
        }
    }
    // Add a border around the entire board
    chessContext.strokeRect(0, 0, 600, 600);
}
```

```
// Get the canvas context
let chessCanvas = document.getElementById("board");
let chessContext = chessCanvas.getContext("2d");

drawBoard();
```

Step 2: The Initial Blank Chessboard

Display the **chess.html** file in a browser, which should look like the following screenshot:

Step 3: Configuring the Chess Pieces

Image objects and array

- You will be using image files to represent the chess pieces.
- Before adding them to the canvas, you'll need to create an **image object** for each one and specify its **src** attribute.
- You'll also put these into an **array** to make it easier to programmatically select the desired image.

Object Store Data

- You'll need to define the data that will be loaded into the object stores.
- The **pieceTypes[]** array defines the properties needed to display each piece

such as **height** and **width**.

- It also specifies the corresponding index in the **images[]** array for both the black and *white* images.
- The **pieces[]** array contains the same details used in the previous lecture such as *row* and *column* and defines the starting position for each of the 32 pieces.

Adding code to the data.js script

- Add the code in 10-local-storage-chessboard-data.txt to a new **data.js** script file.
- This text file is provided so you can simply copy the data code to your Web project instead of re-typing this data.

Step 4.1: Openning the Database

- Now you're ready to create and use a local Indexed DB database to configure and display the chess pieces. Initially, the data will be loaded from static data, and you will simply display the starting position. Later you will animate the pieces by updating their location in the object store.
- You will need to populate the database with some data. For this application, you will just use the

```
let dbEng = window.indexedDB ||
            window.webkitIndexedDB || // Chrome
            window.mozIndexedDB || // Firefox
            window.msIndexedDB; // IE

let db; // This is a handle to the database

if (!dbEng)
    alert("IndexedDB is not supported on this br
else {
    let request = dbEng.open("Chess", 1);

    request.onsuccess = function (event) {
        db = event.target.result;
    }
    request.onerror = function (event) {
        alert("Please allow the browser to open
    }
    request.onupgradeneeded = function (event) {
        configureDatabase(event);
    }
}
```

data declared in the **data.js** script and copy it to the object store. For other applications, this could be downloaded from a server or entered from user input.

- Add the code shown on this slide to the **chess.js** script, just after the call to **drawBoard()** and before the implementation of the **drawBoard()** function itself.
- If you can't access the **indexedDB object**, then the browser does not support it. For this demo you can simply use **alert()** to notify the user and stop further processing.

Step 4.2: Explanation of the Code to open database

- The code in the previous slide uses the **indexedDB object** to open the *Chess database*, specifying that **version** should be used.
- The **open()** method returns an **IDBOpenDBRequest object**, as described earlier in this lecture. You will attach **three event handlers** for this request. These event handlers are as follows:

    1. ***onsuccess:*** *This event handler simply saves the reference to the database. You will add more logic here later. Notice that the database is obtained from the* ***event.target.result*** *property, which is how all results are returned.*
    2. ***onerror:*** *The primary reason that the browser fails to open a database is that the browser has the* ***IndexedDB feature*** *blocked. This can be disabled for security reasons. In this case, the user is prompted to allow access. Alternatively, you could choose to display the error message instead.*

3. ***onupgradeneeded:*** *This is raised if the database does not exist or if the specified version is not the current version. This calls the* ***configureDatabase()*** *function, which you'll implement in the next step.*

Step 5: Defining the Database Structure

- Add the code shown on this slide to the **chess.js** script to implement the **configureDatabase()** function.
- The **objectStoreNames** property of the database object contains a list of the names of all the object stores that have been created.
- To remove all the existing object stores, each of the names in this

```
function configureDatabase(e) {
    alert("Configuring database - current versic
        e.oldVersion + ", requested version is "
            e.newVersion);

    db = e.currentTarget.result;

    // Remove all existing data stores
    let storeList = db.objectStoreNames;
    for (let i = 0; i < storeList.length; i++) {
        db.deleteObjectStore(storeList[i]);
    }
    // Store the piece types
    let typeStore = db.createObjectStore(
        "pieceType", { keyPath: "name" });
    for (let i in pieceTypes){
        typeStore.add(pieceTypes[i]);
    }
    // Create the piece data store (you'll add
    // the data later)
    let pieceStore = db.createObjectStore(
            "piece", { autoIncrement: true });
    pieceStore.createIndex(
        "piecePosition", "pos", { unique: true }
}
```

- To simplify things in this project,

list is passed to the **deleteObjectStore()** method.

- Initially, you'll create two data stores using the **createObjectStore()** method.

    1. ***pieceType:*** *Contains an object for each type of piece such as pawn, rook, or king*
    2. ***piece:*** *Contains an object for each piece, 16 black and 16 white*

- The **configureDatabase()** function will be called if the database does not exist or if it is not the current version. For version changes, you can get the list is passed to the **deleteObjectStore()** method. you'll simply remove all object stores and rebuild the database from scratch.

- *This will wipe out all existing data. That is fine for this example, but in most cases, you'll need to preserve the data where possible.*

current version by using the **db.version** property and then make the necessary adjustments.

- Also, the *event object* passed to the **onupgradeneeded** event handler will have the **e.oldVersion** and **e.newVersion** properties.

(Mark J Collins, 2017: Chapter 25, p.477-8)

Step 6: Specifying the Object Key - I

- You must specify a name for the store when calling the **createObjectStore()** method.

- You can also specify one or more optional parameters.
- Only two are supported:

    1. ***keypath*** *— specified as a collection of property names. If using a single property, this can be specified as a string rather than a collection of strings. This defines the object property (or properties) that will be used as the key. If no keypath is specified, the key must be defined out-of-line using a key generator or providing the key as explained later in this section.*

    2. ***autoIncrement*** *— If true, this indicates that the keys are sequentially assigned by the object store.*

- Every object in a store must have a unique key. There are three ways to specify the key, as previously explained:

    1. *Use the keypath parameter to specify one or more properties that define a unique key. As objects are added, the keypath is used to generate a key based on the object's properties.*

    2. *Use a key generator. If autoIncrement is specified, the object store will assign a key based on an internal counter.*

3. *Provide the key value when adding the object. If you don't specify a key path or use a key generator, you must supply a key when adding an object to a store.*

- **Note:** While in the **onupgradeneeded** event handler, data can be added to an object store without explicitly creating a transaction. There is an implicit transaction created in response to the **onupgradeneeded** event.

Step 6: Create Index - II

- For the piece store there is no natural key available in the pieces data, so you'll use a key generator.
- Generated keys are just a synthetic key used to satisfy the unique constraint.
- Initially when you're drawing the board, you'll retrieve all of the objects, so you don't need to know what the key is.
- Later you'll need to retrieve a piece

- **Warning:** Since the pos property is unique, you might be tempted to use it as the key. However, since you will be moving pieces, their position will change, and it's considered a poor design pattern to use a key that changes often. For Indexed DB, this is especially problematic since you can't actually change a key; you must delete the current object and then add it with the new key.

so you can move it.

- You will find the desired piece based on its position on the board. To facilitate that, you'll add an index to the store based on the pos property.
- Since no two pieces can occupy the same space, the pos property can be used as a unique index. By specifying this as a unique index, you will get an error if you try to insert an object with the same position as an existing object.
- When creating an index, you must specify a keypath like this:

```
pieceStore.createIndex("piecePosition", "pos"
    { unique: true });
```

- In this case, the pos property is the keypath for this index. The keypath may include more than one property, in which case the index will be based on the combination of the selected properties. When an object store has an index, the index is automatically populated when an object is added to the store.

Step 7: Resetting the Board - I

You'll now add a **resetBoard()** function to the **chess.js** script using the following code. This will be called not when the database is created but when the page is loaded.

```
function resetBoard() {
    let xact = db.transaction(["piece"], "readwrite");
    let pieceStore = xact.objectStore("piece");
    let request = pieceStore.clear();
    request.onsuccess = function(event) {
        for (let i in pieces) {
            pieceStore.put(pieces[i]);
        }
    }
}
```

This code creates a transaction using the read-write mode and specifies only the piece object store since that is the only one you'll need to access. Then the piece store is obtained from the transaction. The **clear()** method is used to delete all the objects from the store. Finally, all the objects in the **pieces[]** array are copied to the object store.

Step 7: Resetting the Board - II

Now add the following code shown in bold to the onsuccess event handler. This will call the resetBoard() function after the database has been opened.

```
let request = dbEng.open("Chess", 1);

request.onsuccess = function (event) {
    db = event.target.result;

    // Add the pieces to the board
    resetBoard();
}
```

Note: The *onupgradeneeded* event is raised, and its event handler must complete before the *onsuccess* event is raised. This ensures that the database has been properly configured before it is used.

Tip: In the resetBoard() function, you called the put() method (repeatedly, 32 times). However, you did not get any response objects or implement any event handlers. This code appears to be working synchronously. Actually, these calls

are processed asynchronously, and a response object is returned in both cases, but the return value was ignored. You could implement both onsuccess and onerror event handlers for these requests. In this case, you cheated a little. Since you don't need the result value like you would when retrieving data, you don't have to handle the onsuccess event. Because these calls are within a transaction, subsequent use of these object stores by a different transaction will be blocked until the updates are complete.

Step 8: Drawing the Pieces - I

So far you have opened the database, configuring the object stores, if necessary. You have also populated the piece store with the initial positions. Now you're ready to draw the pieces. To do that you'll implement a drawAllPieces() function to iterate through all of the pieces and a drawPiece() function to display a single image. These functions will be similar to the functions you created in previous lecture with the same names. However, the data for these functions will be retrieved from the new database.

The drawAllPieces() function will use a cursor to process all the objects in the piece object store. For each piece, this will extract the necessary properties and pass them to the drawPiece() function. The drawPiece() function must then access the pieceType store to obtain the type properties such as height and width and display the image in the appropriate location.

Step 8: Drawing the Pieces - II

Using a Cursor

When retrieving data from an object store, if you want to retrieve a single record using its key, use the get() method, which I will describe next. You can also select one or more objects using an index, and I will explain that later in this chapter. To get all the pieces, you'll need to access the entire object store, which you'll do using a cursor.

After creating the transaction and

```
function drawAllPieces() {

    let xact = db.transaction(["piece", "pieceTy
    let pieceStore = xact.objectStore("piece");
    let cursor = pieceStore.openCursor();

    cursor.onsuccess = function (event) {
        let item = event.target.result;
        if (item) {
            if (!item.value.killed) {
                drawPiece(item.value.type,
                          item.value.color,
                          item.value.row,
                          item.value.column,
                          xact);
            }
            item.continue();
        }
    }
}
```

obtaining the object store, you'll call its openCursor() method. This returns an IDBRequest object, and you'll need to provide an onsuccess event handler for it. When the event fires, it provides the first object only. You can obtain the next object by calling the continue() method. To demonstrate this, add the function shown on this slide to the chess.js script.

Step 9: Retrieving a Single Object - I

Now you'll implement the drawPiece() function that will draw a single piece on the board. It must first access the pieceType object store to get the image details. In this case, you'll retrieve a single object using its key. The key to the

pieceType object store is the type property. Add the function shown on this slide to the chess.js script.

```
function drawPiece(type, color, row, column, xact) {
    let typeStore = xact.objectStore("pieceType");
    let request = typeStore.get(type);
    request.onsuccess = function (event) {
        let img;

        if (color === "black") {
            img = images[event.target.result.blackImage];
        }
        else {
            img = images[event.target.result.whiteImage];
        }

        chessContext.drawImage(img,
                        (75 - event.target.result.width) / 2 + (75 * column),
                        73 - event.target.result.height + (75 * row),
                        event.target.result.width,
                        event.target.result.height);
    }
}
```

Step 9: Retrieving a Single Object - II

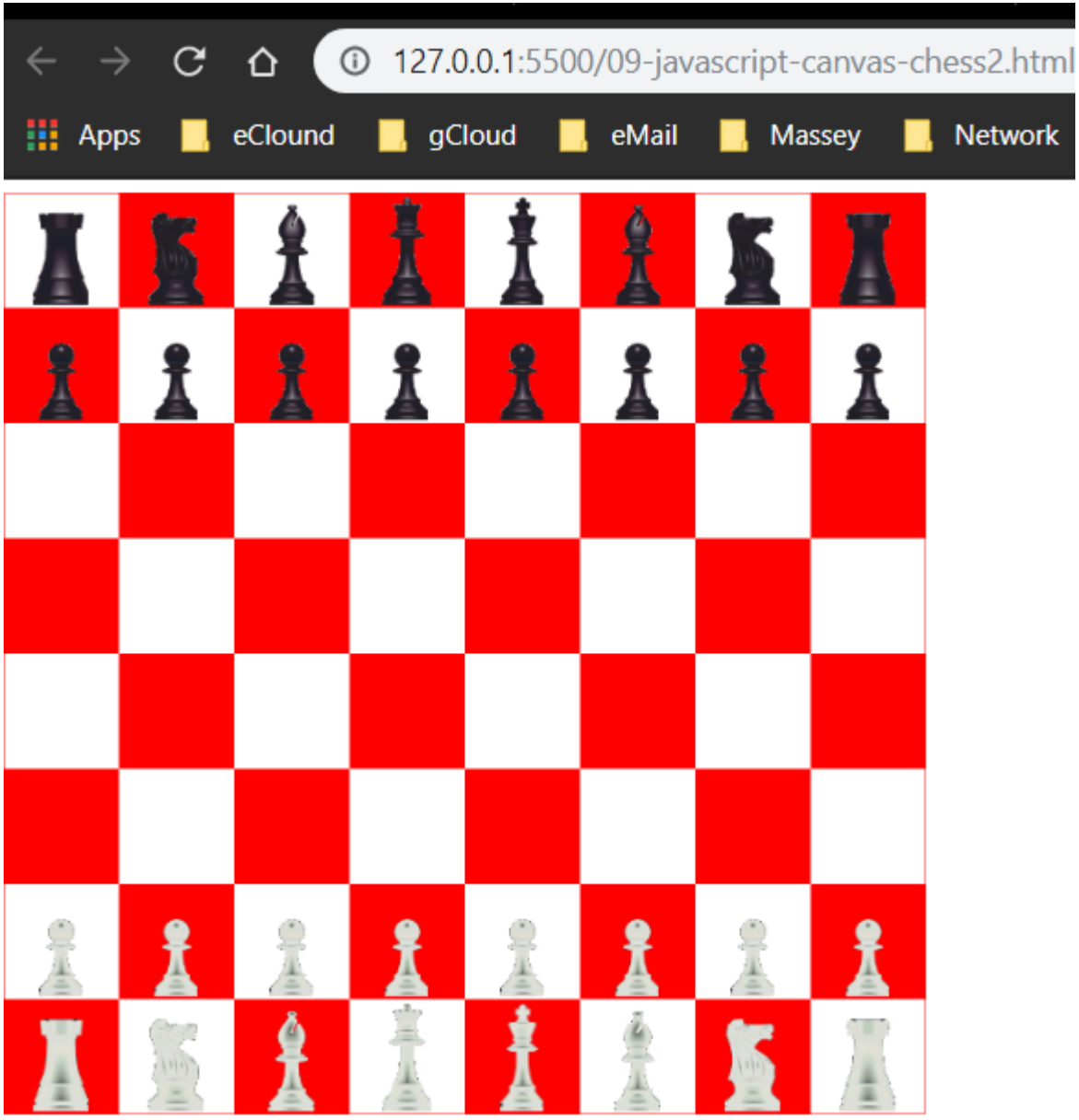
The code in the previous slide uses the same transaction object, which is passed in. It obtains the pieceType object store and then calls its get() method. The onsuccess event handler gets the necessary properties and calls the canvas drawImage() method. Refer to previous lecture slides for more information about drawing images on a canvas. Now add the call to drawAllPieces() in the onsuccess event handler for the open() call by adding the code shown in bold.

```
request.onsuccess = function (event) {
    db = event.target.result;

    // Add the pieces to the board
    resetBoard();

    // Draw the pieces in their initial positions
    drawAllPieces();
}
```

Step 10: Testing the Application

The alert showing the database is being configured

The completed chess board with static pieces



Step 10: Moving the Pieces

- Now you're ready to animate the board by moving the pieces.
- A piece can be moved by simply updating its position and then redrawing the board.
- There is one complication, however; if a move captures a piece, you need to remove it from the board. For now, you'll simply delete the object from the store.
- Study the code in the archive, 10-local-storage-example-chessboard.zip, and identify how

The completed chess board with animations. Visit the live version.

moving the pieces is implemented.
Discuss with your fellow student.