# Offline web applications

We will look at Web SQL, IndexedDB, the FileSystem API, and the HTTP cache

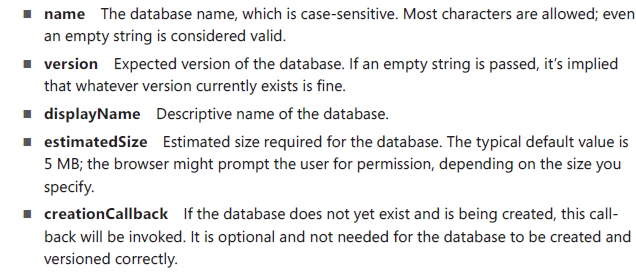
## Lesson 1: Working with Web SQL

**Serious Note**: W3C *no longer supports Web SQL*, but some browsers have continued the support of it. See http://caniuse.com/#feat=sql-storage for a full list of browsers (If you do plan on developing with Web SQL, consider the lack of browser support).

### Creating and opening the database

- Use the *openDatabase* method to start communication with a database (it returns a Database object). Note that if you attempt to open a database that doesn't exist, it will be automatically created for you.

The following are the *openDatabase* parameters:



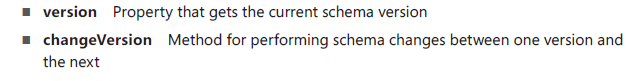
The following example creates a database named 'Library' with an estimated size of 5MB (it returns a Database object that supports transactional operations):

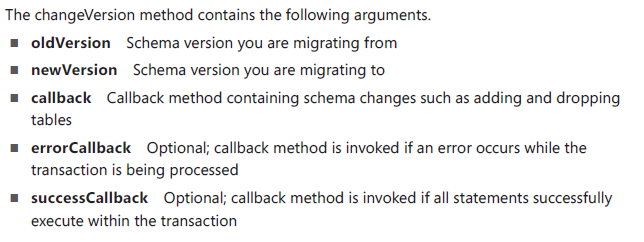
var db = openDatabase('Library', '1.0', 'My Library', 5 \* 1024 \* 1024);

**Closing a connection**: with Web SQL, closing a connection is automatically handled for you

### Performing schema updates

You may need to add new tables, drop existing ones, or even change particular columns. The *Database* object provides the following hooks.



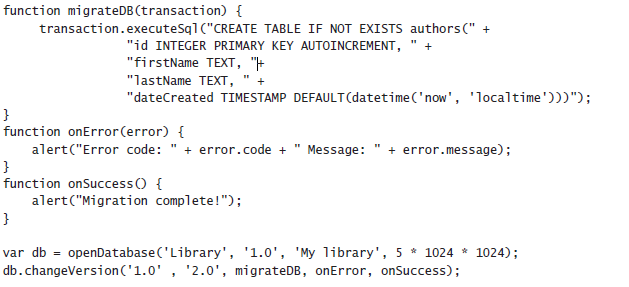


#### Adding a table

To add a table, need a callback method that accepts a *transaction* object, which executes the CREATE TABLE script.

**Note**: The *transaction* object allows multiple actions within it, and it automatically rolls back all changes if any fail.

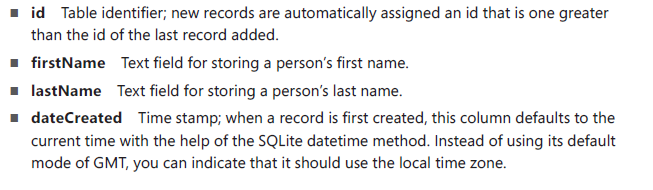
Javascript Example:



JSFiddle: https://jsfiddle.net/bs4r5ooa/ (**The code did not work on JsFiddle, or on my local computer - Forget this garbage**)

**To read the database version:** alert("Current schema: " + db.version);

**After the migration**, the current database has the following fields:



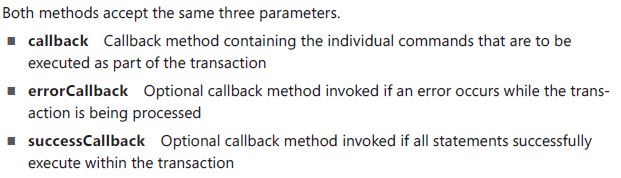
#### Using transactions

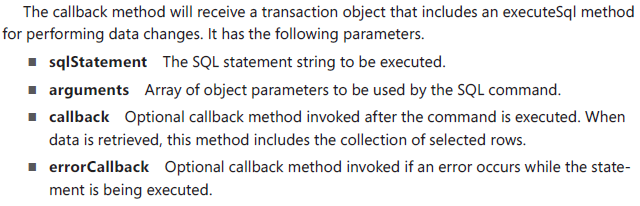
Use transactions to execute SQL statements.

The Database Object provides the ff two methods:



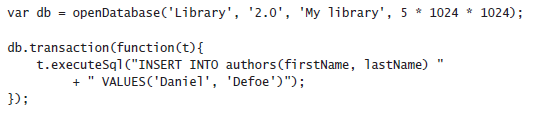




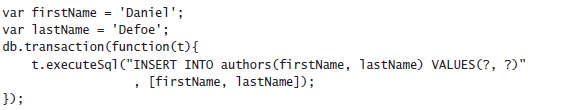


#### Inserting a new record

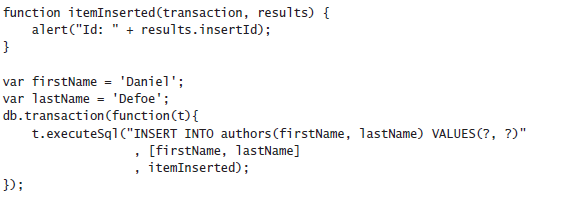
**Without using SQL parameters:**



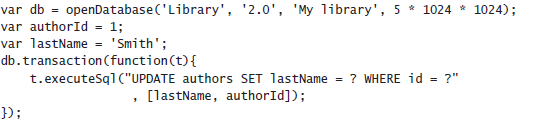
**Using SQL parameters:**

****

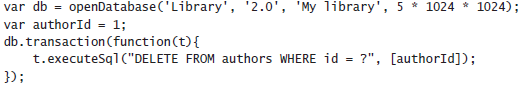
**Add a callback function to the executeSql method, which allows us capture the Id of the newly created row:**

****

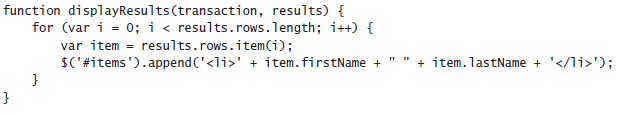
#### Updating an existing record

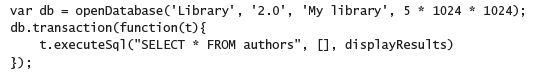


#### Deleting a record



#### Reading values from the database

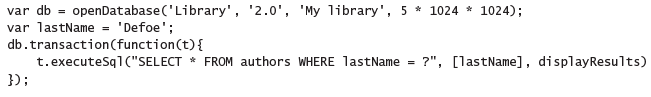




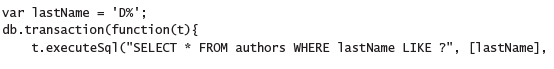
NOTE: Because you are only retrieving data, you just as easily could have used the *readTransaction* method instead of the transaction method



#### Filtering results (using WHERE)

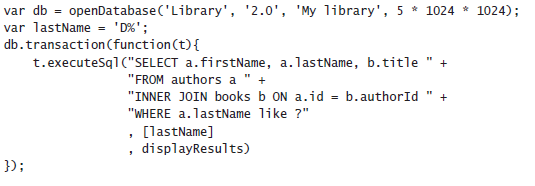


**Using Wildcard symbol** (%)

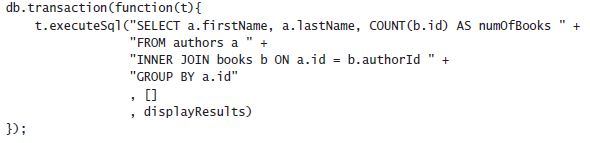




#### Using JOIN commands



#### using Aggregating functions



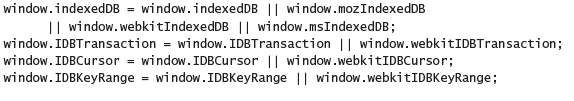
## Lesson 2: Working with IndexedDB

**What is IndexedDB**: A key/value database in which values can range from simple strings to complex object structures

### Using browser-specific code

IndexedDB is still under development hence we need to use browser-specific prefixes.

**Note**: To **make your IndexedDB code cross-browser-friendly**, include the following code at the top of your page

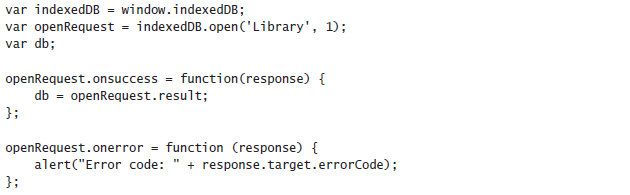


### Creating and opening the database

**To access the browser's indexedDB object,** use:

var indexedDB = window.indexedDB;

**To open an indexedDB object store**, use



**Live code**: Run "index.html". Look at the "testIndexedDBOpen.js" file to see working code.

**Note about using the 'open' indexedDB method**: This method returns an *IDBRequest* object and begins an *asynchronous process* of opening a connection. Hence we have to listen to the 'onsuccess' and 'onerror' methods to determine when the database has been opened or when an error occurs.

### Using object stores

IndexedDB uses 'Object stores', which are key/value storage areas.

#### Understanding versioning

Before creating a new 'object store', you need to understand how IndexedDB handles versioning.

E.g. Consider the code below



In the code, a **version number is passed as the second parameter**. The *'request object'* returned contains an *'onupgradeneeded'* event that will be **triggered if the version requested doesn't match the current version of the existing database (or if the database does not yet exist)**.

**Note**: the ' *onupgradeneeded'* event will be fired BEFORE the *'onsuccess'* event

**How to allocate a new storage area**

Within the *onupgradeneeded* event handler, use the *createObjectStore* method to allocate a new storage area.

This method requires an *object store name* and an *object* containing any extra parameters to use in configuring the store.

#### Using the keypath property

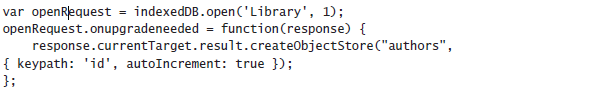
Used to specify which property on the *value* object should be used as the key.

(The key is used as the primary index for stored object instances)

**NOTE**: If the property specified by the keypath does not exist on the value object, you must use a key generator such as autoIncrement, which creates auto incrementing numeric keys

e.g.

**JS**:



**Live Code**: open 'keypathTest.html' in a browser. See 'testIndexedDBKeypath.js' for the code

**To use an existing field in the value object as the *key*** (e.g. using the 'email' property as a key)*,* use:

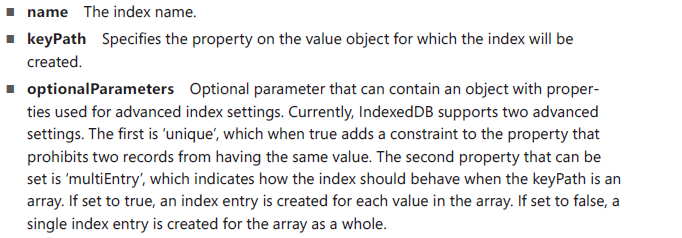
****

#### Adding indexes

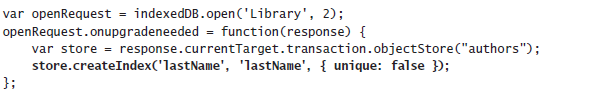
Although the key will be the primary index for object stores, you can specify other indexes.

**Use This When**: properties other than the key might be commonly used in sorting or filtering.

**To create an index**, use the 'createIndex' method on the object store, which has the following parameters:



e.g. using the *createIndex* method on the object store to create a new, non-unique index for the lastName property

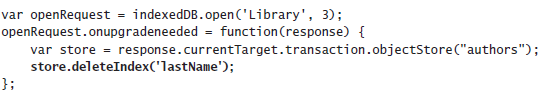


Note: The *createIndex* method is called during a database migration, within the *onupgradeneeded* event handler, to ensure that the index is created when the version is updated.

#### Removing indexes

Remove an index by creating a database migration that uses the object store's *deleteIndex*() method

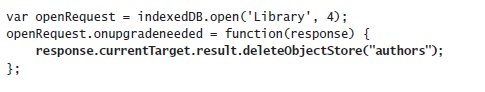
e.g.



#### Removing object stores

use *deleteObjectStore()* method in the *onupgradeneeded* event

e.g.



### Using transactions

**objectStoreNames**: use a single string when creating a transaction for one objectStore, and an array when creating a transaction for multiple objectStores

e.g. opening a transaction for a single object store

*var trans = db.transaction('authors');*

Here is an example of opening a transaction for multiple object stores.

*var trans = db.transaction(['authors', 'books']);*

**mode**: Optional parameter. Possible values are *readonly* and *readwrite*. Default value is *readonly.*

e.g. a transaction being opened in *readonly* mode.

*var trans = db.transaction('authors', 'readonly');*

Example of a transaction being opened in *readwrite* mode.

*var trans = db.transaction('authors', 'readwrite');*

#### Inserting a new record

After creating a transaction, we can use it to add a new record.

Steps to adding a new record using a transaction:

1. find the object store to which we want to add the record

2. Call the *add* method of the object store, which will insert the record asynchronously. (**Note**: the *add* method returns a request instance in which you can subscribe to an *onsuccess* event that provides notification when the operation is completed)

3. Use the *request.result* property to obtain the auto-generated id for the new record

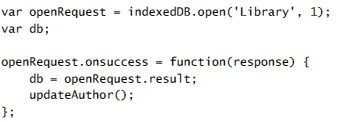
4. You can also subscribe to the *onerror* event if the operation fails

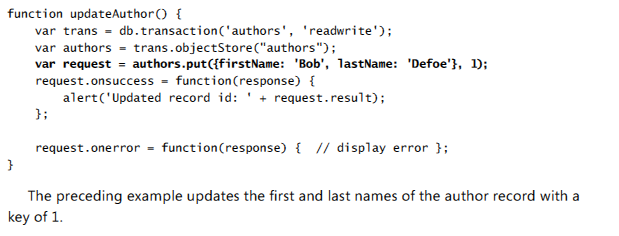
**Note**: An exception is thrown if the add method is called using a key that already exists  
e.g.  


#### Updating an existing record

Just like adding a new record except we use the *put* method of the object store.

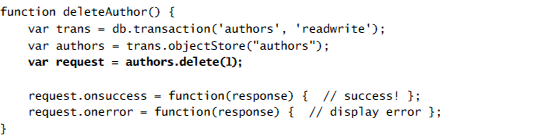
**Note**: We can use the *put()* method for both adding and updating records, However the *add()* method can only be used for adding new records.

e.g.  




#### Deleting a record

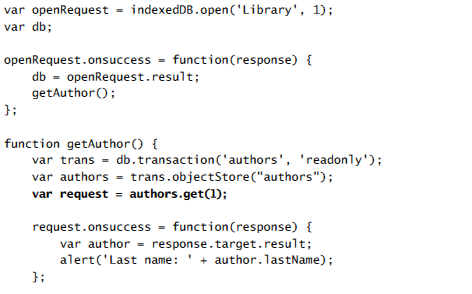
To remove a stored object, you need to pass its key value to the delete method of the object store

e.g.  


#### Retrieving a record

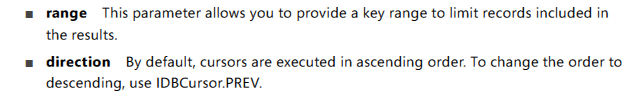
If you need to find a specific record use the *get()* method of the object store

**Note**: Like other operations, this needs to be done within a transaction

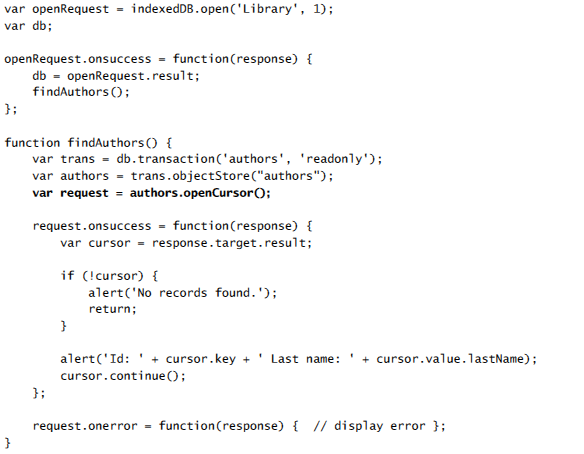
e.g.  
  


### Understanding cursors

Cursors are another way to find records.

A cursor can be opened by calling the *openCursor()* method on the object store, which returns a request object and accepts the following parameters:  


e.g. The following is a simple example that iterates through all records held in the authors object store



**Notes on using cursors**:

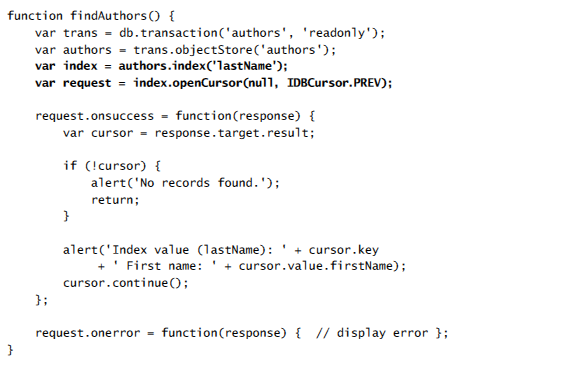
* The **cursor itself is on the result property** of the response of the *onsuccess* event handler.
* If records are found, the **cursor’s value property will contain the current record**.
* **To continue iterating, invoke the cursor’s continue method**, which will trigger the *onsuccess* eventhandler again, this time with the next record in the results. **When it reaches the end of the collection, the *onsuccess* event will have a null cursor**.

#### Indexing cursors

Cursors can also be created by using an index of an object store.

The cursor opened in the previous example using the *openCursor()* method will return the entire object associated to the index value.

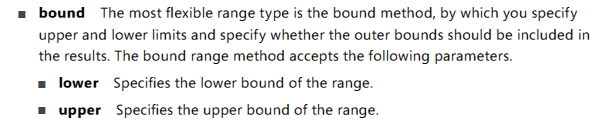
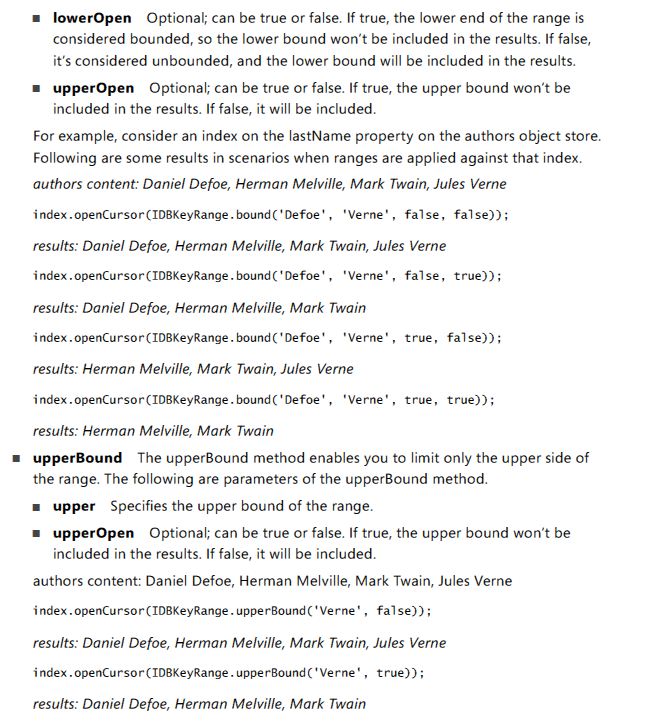
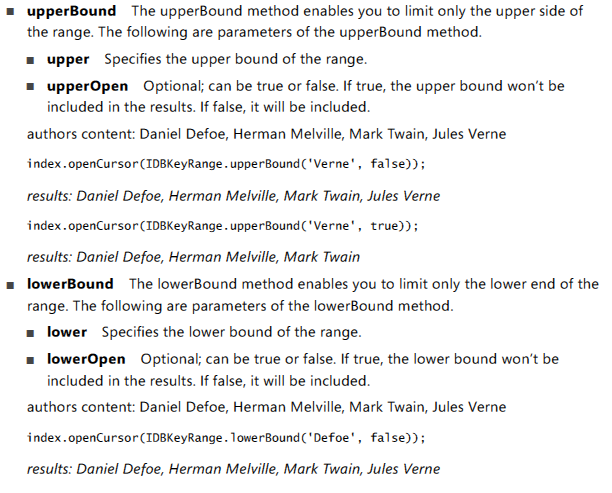
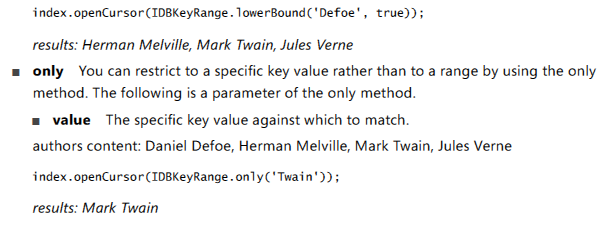
If you **only need the object keys** and not the full object, use the *openKeyCursor()* method

e.g. the example below is a modified version of the *findAuthors()* method, which creates the cursor against the lastName index instead of going directly against the object store.  


#### Applying key range limits

To limit the results of a cursor to a subset of the data store, pass an IDBKeyRange value as the first parameter to the openCursor method.

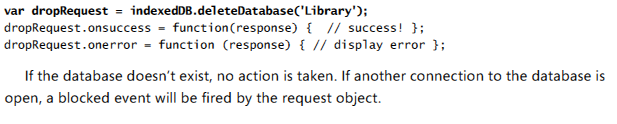
The IDBKeyRange object can accept the following range methods:

### Dropping a database

The IDBFactory object that is referenced by the indexedDB object contains a *deleteDatabase()* method that removes an existing database.

**deleteDatabase()** : takes a name parameter and returns a request object immediately while asynchronously attempting to drop the database

e.g.  


## Lesson 3: Working with the FileSystem API

Using the FileSystem API, we can create directories in a sandboxed location on the user’s system where we can store images, text files, and other large bits of data

Lesson objectives

* Describe the use of the FileSystem API
* Implement the FileSystem API

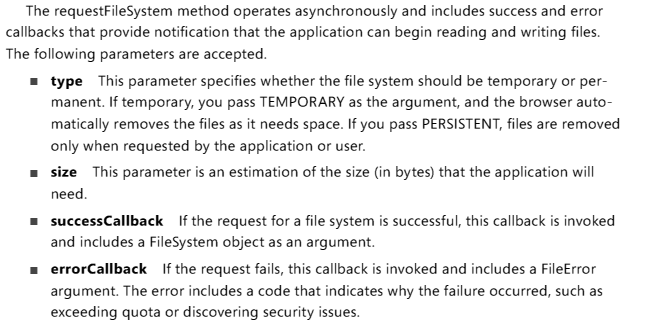
### Assessing browser support

The only major browser that supports the FileSystem API is Chrome, therefore the FileSystem API is most commonly used in creating Chrome extensions.

### Opening the file system

Use the following code to open the FileSystem:

window.requestFileSystem = window.requestFileSystem || window.webkitRequestFileSystem;

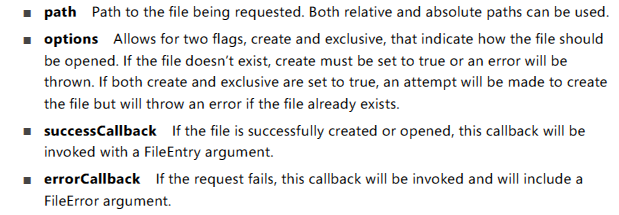


### Creating and opening a file

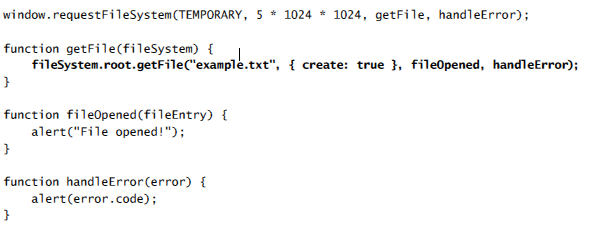
To create a file we need to have a DirectoryEntry object so we have an allocated place to put the file.

The FileSystem argument passed to the *successCallback()* method includes a special DirectoryEntry as a property named root, which points to the root of the file system reserved for the application.

A DirectoryEntry object has a getFile() method that can both create new files and read those that already exist.

The following are the parameters of the getFile() method:  
  


e.g. The example below shows us how to create a new file in the root directory of the DirectoryEntry



### Writing to a file

When you have access to a FileEntry object, you can create a FileWriter, which persists data to the opened file.