Session 7. Storage & Promises

Programació Multiplataforma i Distribuïda

Grau d'informàtica. EPSEVG

Octubre 2021

Jordi Esteve jesteve@cs.upc.edu

Why we need browser storage?

The web clients and servers are REST compliant systems.

REST (REpresentational State Transfer) is an architectural style providing standards to communicate computer systems:

- The client and server implementations can be done **independently** of each other.
- Are **stateless**: Server does not know about the state of the client and vice versa.

So, when a web server has sent a web page to a browser, the connection is shut down and the server forgets everything about the client.

- How to remember information about the client?
- How to store local information to avoid requesting it again several times?
- How to store local information to be able to work offline?

Browser storage is the solution.

Browser storage

We have different options to get browser storage (before HTML5 only cookies):

- **Cookies**: Small piece of data (<4kB) stored in name-value pairs and optional parameters like expiring date and path. They are include in every server request.
- **Web Storage**: Large storage (5MB) for name-value pairs. The storage could be local (no expiration date) o session (deleted when closing browser or tab). It is more secure and fast because the information is not sent in the server requests.
- **IndexedDB**: Allows to store large amount of data and provides indexing, transactions, and robust querying capabilities. It is more efficient and faster.

~~**Web SQL Database**~~, a storage based in a SQL database like SQLite, was a working specification but now W3C has deprecated it. So some browsers like Firefox, Safari o IE do not support it. IndexedDB is the alternative.

See the evolution of browser support of these technologies in https://caniuse.com/

Browser storage: Cookies

A **Cookie** is a small piece of data (<4kB) stored in name-value pairs and optional parameters like expiring date and path.

When a browser requests a web page from a server, cookies belonging to the page are added to the request to give this information to server. It slows down the connection.

JavaScript can create, read, and delete cookies with the **document.cookie** property.

If there is not an expiring date, the cookie is deleted when the browser/tab is closed.

It there is not a path, the cookie belongs to the current page.

```
document.cookie = "user=Joan Coma"; // Cookie created with a name=value.
document.cookie = "user=Joan Coma; expires=Thu, 25 Dec 2020 12:00:00 UTC"; // With date
document.cookie = "user=Joan Coma; expires=Thu, 25 Dec 2020 12:00:00 UTC; path=/";
// Modify a cookie overwriting it
document.cookie = "user=Marc Sala; expires=Thu, 25 Dec 2020 12:00:00 UTC; path=/";
// Read a cookie. Now we must process its parts, for example with c.split(";")
let c = document.cookie;
// Delete a cookie giving a date in the past
document.cookie = "user=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/";
```

Browser storage: Cookies exercise

We want to store the user preferences in our task list web app. For example, we could store the last options about searching, ordering and filtering by active tasks.

1. Download from Atenea the *cookie.js* library that offers 3 functions to *get*, *set* and *delete* cookies. Save it in your task list app folder (I suggest you to create a *lib* subfolder where to put all your additional libraries).

2. Load this library in task.html: <script src="lib/cookie.js"></script>

3. Initialize the variables (the string of searching, the type of order and the boolean of filtering by active tasks) with the cookie values if the cookies exist:

```
active = Cookie.get("active") ? JSON.parse(Cookie.get("active")) : false;
search = Cookie.get("search") ? JSON.parse(Cookie.get("search")) : "";
order  = Cookie.get("order")  ? JSON.parse(Cookie.get("order"))  : {};
```

4. When the previous variables change, save their values to their cookies:

```
Cookie.set("active", JSON.stringify(active), 7);
Cookie.set("search", JSON.stringify(search), 7);
Cookie.set("order",  JSON.stringify(order),  7);
```

Browser storage: Web Storage

Web Storage allows a large storage (minimum of 5MB) for name-value pairs. It is more secure and fast because the information is not sent in the server requests.

Web browsers offers separated Web Storages for each origin (domain+protocol).

Two different window objects can be used with Web Storage:

- **localStorage**: The stored data does not have expiration date
- **sessionStorage**: The stored data is deleted when closing browser or tab.

The values are stored as string. Use JSON.parse() to recover original types.

```
// 2 ways to store a value on browser for duration of the session
sessionStorage.setItem('key', 'value');
sessionStorage.key = 'value';
// 2 ways to get the value
sessionStorage.getItem('key');
sessionStorage.key;
// 2 ways to store a value on the browser beyond the duration of the session
localStorage.setItem('key', 'value');
localStorage.key = 'value';
```

Browser storage: Web Storage exercise

We want to get a long term storage to keep our task list over time. So we could use **localStorage** to keep the task array.

1. Change where the task list is stored: Instead of using the *tasks* attribute, save the task list in a *localStorage* (use the *name* attribute as the key to prevent that different *TaskModel* objects use the same localStorage):
```
localStorage[this.name] = localStorage[this.name] || JSON.stringify(this.initial_tasks);
```
2. Recover the task list array from the localStorage when you need it:
```
let tasks = JSON.parse(localStorage[this.name]);
```
3. Update the localStorage if the task list is changed (create, update, delete, reset):
```
localStorage[this.name] = JSON.stringify(tasks);
```
4. Test if the task list is conserved after closing the tab or the browser.

Tip: Inside the developer tools of our Web browser there is a Storing tab/section where we can observe the data in localStorage, sessionStorage, IndexedDB and cookies. If we have problems with their values, we can change or remove them.

Task list exercise: Pagination + TaskVC constructor

Now we will add pagination in the task list to prevent display large lists when there are a lot of tasks. We can select the number of items per page and to go to the next/previous page or to a specific page.

The View and Controller code can be placed inside a constructor (TaskVC), so we can create several task view+controller objects to manage several task lists at once.

1. Download the *task_all.zip* file from Atena that contains a basic HTML web page, a script (*task_view_controller.js*) with the **views** and **controllers** inside a constructor and a script (*task_model.js)* with the **model** to manipulate lists of tasks stored in localStorage.

2. Unzip the file. Load *task.html* in a web browser. Test all the operations in both task lists. Add new tasks to grow up the list to test the new pagination feature.

3. Fill the missing code in the `switchController`, `deleteController` and `resetController` functions. Then test the deletion, reset and switch work fine.

Browser storage: IndexedDB

Web Storage is much better than Cookies but has some limitations:

- Can not search/sort efficiently inside the values.
- Can get race conditions (two functions that modify the same data wins the last).
- Read/Write operations blocks the browser (they are not asynchronous).

IndexedDB allows to store large amount of data and provides indexing, transactions, and robust querying capabilities:

- It is more efficient and faster than Web Storage.
- It stores and gets objects indexed by a "key", but other indexes can be created.
- The changes to the database (CRUD operations) happen within transactions.
- It follows a same-origin policy: You cannot access data across different domains.
- It makes extensive use of an asynchronous API: most processing will be done in callback functions.

Relational database definitions

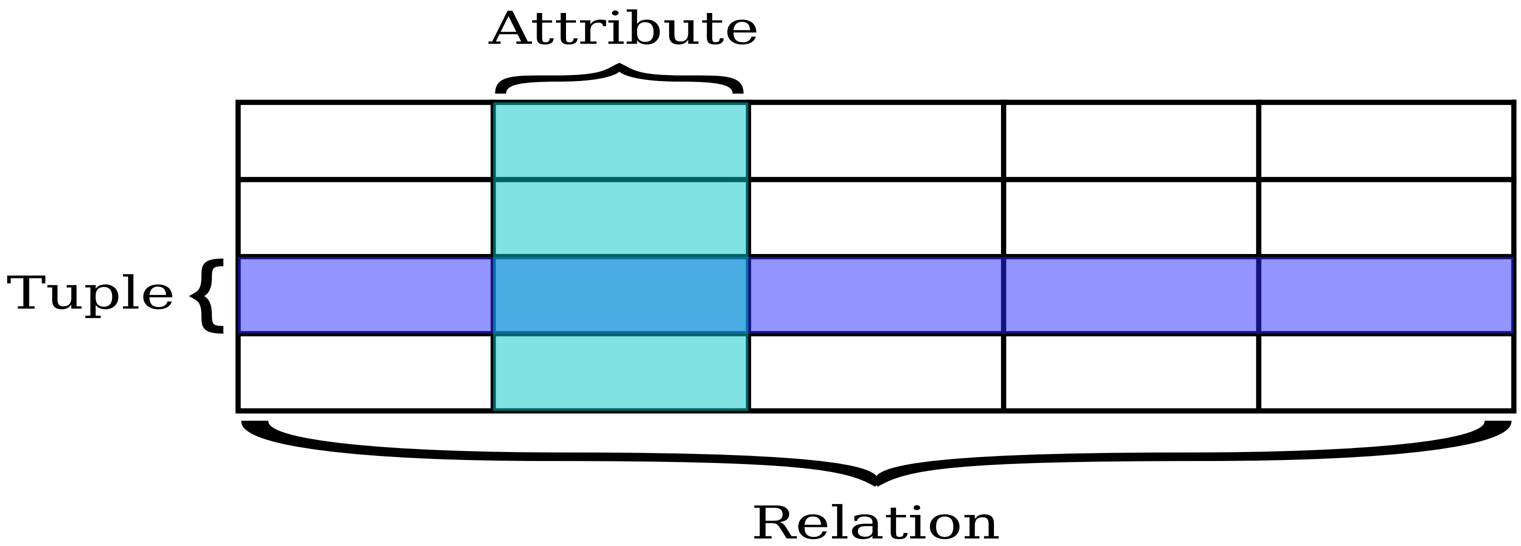
In relational databases we have the concepts of:

Tuple/Row/Record: A data set representing a single item (e.g. "A student")

Attribute/Column/Field: A labeled element of a tuple, (e.g. "The student's name")

Relation/Table: A set of tuples sharing the same attributes; a set of columns and rows (e.g. "The students of the PMUD course")

Database: Set of relations/tables that contains related information.



Browser storage: IndexedDB definitions

Each origin (domain+port) has an associated set of IndexedDB databases.

An **IndexedDB database** comprises

- a collection of **Object Stores** (similar to relations or tables) which comprises
  - a collection of **Indexes** (similar to attributes or columns).

Database

- Every database has a **name** that identifies it within a specific origin.
- Each database has a current **version** starting with 0. We can do a version change to upgrade the database.
- The act of opening a database creates a **connection**. There may be multiple connections to a given database at any given time.

Browser storage: IndexedDB definitions (II)

Object Store

- Every object store has a unique **name**.
- The object store persistently holds records (JavaScript objects), which are **key-value pairs** (the key is like a "primary key" in a relational database).
- Records within an object store are sorted according to the keys in ascending order.

Key. The key identifies the record. Several mechanisms to derive the key:

- **A key generator.** It generates a monotonically increasing number every time a key is needed (similar to auto-incremented primary key in a relational database).
- Keys can be derived via a **key path**.
- Keys can also be **explicitly specified** when a value is stored.

Browser storage: IndexedDB definitions (III)

Value

- Each record has a value, it can be any JavaScript value: boolean, number, string, date, object, array, regexp, undefined, and null.

Transaction

- A transaction is used to interact with the data in a database.
- Is an **atomic, isolate** and **durable** set of data access & modification operations.
- The transaction has a **mode** (read, readwrite or versionchange) that determines which types of interactions can be performed upon that transaction.
- The transaction ends with **success** or **fail**.

Browser storage: IndexedDB definitions (IV)

Index

- Useful to retrieve records in an object store through other means than their key.
- An index allows to look up records using the properties of the values.
- Indexes can **speed up** object retrieval and allow **multi-criteria searches**.
- There can be several indexes referencing the same object store. Changes to the object store cause all such indexes to get updated.
- An index contains a **unique** flag. When it is true, the index enforces that no two records in the index have the same key.

Key range

- It is a **continuous interval** over some data type used for keys.
- We can get records from object stores and indexes using **keys** or a **range of keys**.
- The key range can be limited using **lower** and **upper bounds**.

Browser storage: IndexedDB. Example of opening a database

```
let db; // the database connection we want to create
let requestdb = indexedDB.open('StudentDB', 1);
requestdb.onerror = function(event) {  // Handle errors.
  console.error("Error opening the database, errcode=" + event.target.error.name);
};
requestdb.onupgradeneeded = function(event) {
   console.log("Creating a new version of the database");
   db = event.target.result;
   // Create an objectStore to hold students using dni as key path.
   var objectStore = db.createObjectStore("students", { keyPath: "dni" });
   // Create an index to search students by name. The name is not unique.
   objectStore.createIndex("name", "name", { unique: false });
   // Create an index to search students by email. The email is unique.
   objectStore.createIndex("email", "email", { unique: true });
};
requestdb.onsuccess = function(event) {
  console.log("Database is opened");
  db = event.target.result;
};
```

Browser storage: IndexedDB. Example of transaction and request

```
let transaction = db.transaction(["students"], "readwrite"); // Created a readwrite transaction
transaction.oncomplete = function(event) {
   console.log("Transaction done");
};
transaction.onerror = function(event) {
  console.error("Error on transaction, errcode=" + event.target.error.name);
};

let request = transaction.objectStore("students").add(  // Request to add a student
  {dni: "12345678A", name: "Joan Coma", email: "jcoma@upc.edu" }
);
request.onsuccess = function(event) {
   console.log("Student with dni= " + event.target.result + "added.");
};
request.onerror = function(event) {
   console.error("Could not insert student, errcode = " + event.target.error.name);
};
```

Browser storage: IndexedDB + YDN library

You can find more information and examples about IndexedDB in this article.

The YDN-DB library can help us writing code that uses IndexedDB. It offers:

- Multi-browser support: If a browser does not support IndexedDB, it changes to WebStorage automatically.
- Powerful queries (compound indexes, multi-entry indexes, …)
- Full-text searches
- Database syncs (synchronization with RESTFul web services)

To be able to use IndexedDB and YDN-DB can be useful to learn about asynchronous programming using promises (**Promise** is a new class added in JavaScript ES6).

JavaScript Promises

A **promise** is a task that promises to generate a value in the future, if it can.

A promise has three states:

- **pending**: before executing the associated task
- **resolved**: the task is successful and generates the promised value
- **rejected**: the task fails and generates a rejection code, not the promised value

Promises simplify **asynchronous** programming.

They retain the efficiency of execution of asynchronous **callback**.

They clearly separate the **normal code** from the **error attention code**.

JavaScript Promises: Creation of a promise

A promise is an object of the class **Promise** built with

new Promise ((resolve, reject) => <sentences>)

resolve and **reject** are functions that must be invoked to resolve or reject the promise.

Parameter: **resolve (<value>)**
You must call resolve to indicate that the promise has been completed successfully.
<value> is the value generated by the promise.
<value> can be of any type: string, number, array, object or even another promise.

Parameter: **reject (<reason>)**
You must call reject to indicate that the promise is rejected.
<reason> is the rejection code generated by the promise, which usually describes the cause of the failure.

```
let p = new Promise ((resolve, reject) => {
 if(ok) resolve(data); else reject(err);
});
```

JavaScript Promises: Creation of special promises

Class method: **Promise.resolve (<value>)**

- Creates a promise that always ends successfully and generates <value>, unless an error occurs in the generation of <value>.
- It is used to generate the first value of several chained promises.
- This promise calculates and returns <value> immediately.

```
let p1 = Promise.resolve({id: 1, info: 'Ok'}); // Promises p1 and p2 are equivalent
let p2 = new Promise((resolve, reject) => resolve({id: 1, info: 'Ok'}) );
```

Class method: **Promise.reject (<reason>)**

- Creates a promise that is always rejected with <reason>.

```
let p3 = Promise.reject("Refused promise"); // Promises p3 and p4 are equivalent
let p4 = new Promise((resolve, reject) => reject("Refused promise") );
```

JavaScript Promises: Processing the result of a promise

<promise>.then(<success_callback>, <rejection_callback>)

This method receives the success or rejection value generated by <promise>.

It will invoke the **success** or **rejection callback** (handler) that corresponds to the result of <promise> when it is resolved.

This method returns a promise, so it can be chained with another then(...) method.

The success callback **(data => <sentences>)** is invoked if the previous promise is resolved. The callback will receive in the data parameter the value generated by the previous promise.

The rejection callback **(err => <sentences>)** is invoked if the promise is rejected. The callback will receive in the err parameter the reason for the rejection by the previous promise.

```
let p = new Promise ((resolve, reject) => {
  if(ok) resolve(data); else reject(err);
}).then(
  data => console.log("Result: " + data),
  err => console.error("Error: " + err)
);
```

JavaScript Promises: Processing the result of a promise (II)

then(...) can be invoked only with the success callback:

<promise>.then(<success_callback>)

In this case he attends only to success and if he receives a rejection, he propagates it to the next promise.

catch(...) is a complementary method of then(...), which only attends rejections and propagates successes. catch (...) is invoked only with the rejection callback:

<promise>.catch(<rejection_callback>)

then(...) and catch(...) create promises that can be chained together:

```
// Same promise than previous page with the same success and rejection processing
let p = new Promise ((resolve, reject) => {
  if(ok) resolve(data); else reject(err);
})
.then(data => console.log("Result: " + data))
.catch(err => console.log("Error: " + err));
```

Task list exercise: IndexedDB + Promises

Now we will change the *task_model.js* to work with IndexedDB instead of localStorage.

1. Download the *task_all_indexedDB.zip* file from Atena that contains a basic HTML web page, a script (*task_view_controller.js*) with the **views** and **controllers** inside a constructor and a script (*task_model.js)* with the **model** to manipulate lists of tasks stored in indexedDB using YDN-DB library.
2. Unzip the file. Load *task.html* in a web browser and test the operations.
3. Look at *task_model.js* how the db schema is defined, the db is opened and all the CRUD operations are programmed creating promises.
4. Look at *task_view_controller.js* how the promises are processed with **.then()** and **.catch()**, and several promises are chained. **throw** can trigger a failure.
5. Fill the missing code in the `switchController`, `deleteController` and `resetController` functions. Then test the deletion, reset and switch.