# Day 38

Client Side Storage

- Refers to storage on the client device - on the browser

- Accessible by JavaScript

- Uses cases for browser storage
  - Site preferences like language, colour, layout, etc
  - History like activities on Amazon
  - Store immutable/static data for faster access
  - For offline use

- Data is only available on the browser that it created it
  - Not replicated to other browser
  - Need to do this manually

- Relatively safe as each application can only read their own data
  - Should not store sensitive data in browser storage

Storage on the Browser

- Client side storage are dictated by
  - By the browser
  - By the application/user
  - By the web application

- By the browser
  - Service worker using the Cache API to cache request/response
    - https://developer.mozilla.org/en-US/docs/Web/API/Cache

- By the application
  - HTML 5 application
  - Web applications

- By the web application
  - For caching responses eg. `Cache-Control` header
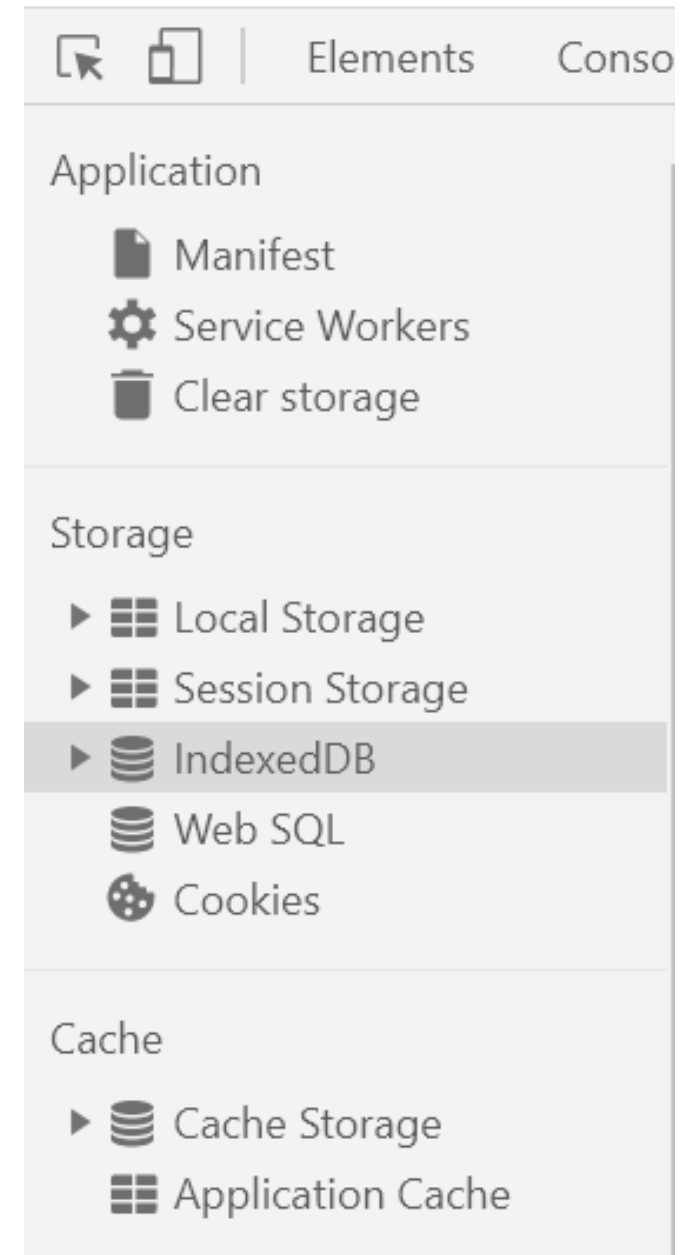
Storage on the Browser

- Cookies
  - Used by web application to save information on the client

```
GET http://www.example.com/ HTTP/1.1
```

```
HTTP/1.1 200 OK
Set-Cookie: session-id=12345;
```

```
GET http://www.example.com/ HTTP/1.1
Cookie: session-id=12345;
```

  - Information saved as cookies are returned to the web application whenever the browser makes a request to the same server

- Local and session storage
  - Key/value data bases
  - Values can only be strings
  - Session storage clears all data when you exit the browser
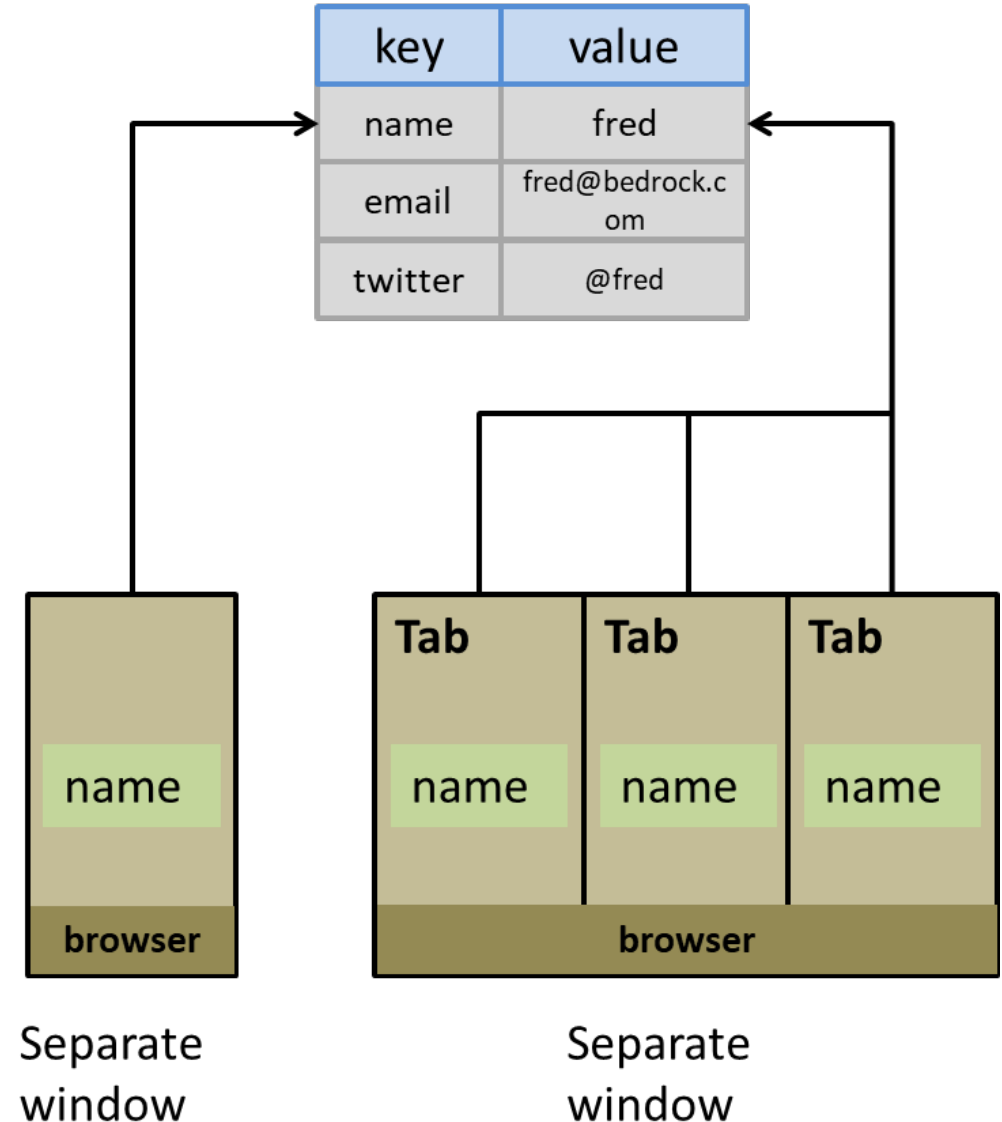 - Local storage will persist data across browser restarts
- IndexedDB
  - Document/object store
  - Richer data format and data type
  - Can store more data and higher performance

Local/Session Storage

- Local storage is a key/value pair storage
- Both key and value are string
  - So need to convert String to appropriate data type
  - Eg parseInt() to convert to integer
- Local storage data are shared by all open tabs/windows from the same domain
- The data are stored locally in the browser
  - It is not encrypted so its not secure
  - Only store non sensitive data

key	value
name	fred
email	fred@bedrock.com
twitter	@fred

name

browser

Separate window

Tab   Tab   Tab

name   name   name

browser

Separate window

Using Local/Session Storage

- Browsers that support local storage provides a global object called **`localStorage`** or **`sessionStorage`**
- To save and retrieve data from **`localStorage`**
  - `localStorage.setItem("key", "value")`
  - `localStorage.getItem("key")`
- **`localStorage`** also behaves like an object
  - `localStorage["key"] = "value"`
  - `localStorage["key"]`
- To remove an item
  - `localStorage.removeItem("key")`
  - `localStorage["key"] = null` will not remove the item
    - Just sets the value of the key to `null`
- Remove all data for the current domain
  - `localStorage.clear()`

IndexedDB

- Has larger storage capacity than local storage
- Can store structured data type and rich data types
  - Number, string, boolean, objects, array
- Native IndexedDB API is very low level and cumbersome
  - Uses callback
- Dexie.js is a wrapper for the IndexedDB native API
  - API is very easy to use
- Install with

```
npm install dexie
```

Creating a Database

```typescript
import Dexie from 'dexie';

export class MyStore extends Dexie {

  cart: Dexie.Table<Cart, number>;


  constructor() {
  super('MyStoreDB')
  this.version(1).stores({
    cart: '++cartId'
  });
  this.cart = this.table('cart')
  }
}
```
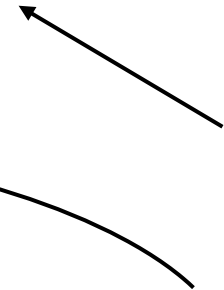
```typescript
export interface Item {
  description: string
  price: number;
}
export interface Cart {
  cartId: number;
  username: string;
  date: numner;
  contents: Item[];
}
```
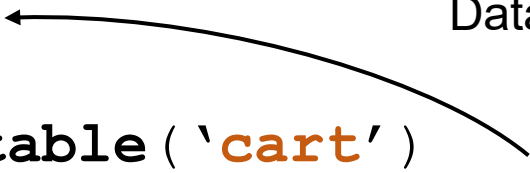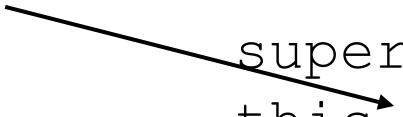
Extend Dexie

Schema of the document

Table with Cart as the schema and the primary key is number

Database name

Schema version

Collection name

Hold a reference of the collection

One or more attributes to be indexed. Can annotate fields with special characters to indicate the type of index

STACKUP

Database Version

- `version()` is used to set the version of the database
- Once created, schema cannot be changed unless the version number is increased

```
this.version(1).store({
  cart: '++cartId'
})
```

```
this.version(1).store({
  cart: '++cartId',
  user: 'userId'
})
```
❌

```
this.version(2).store({
  cart: '++cartId',
  user: 'userId'
})
```
✅

- See https://dexie.org/docs/Version/Version for more details

Indexes

- Indexes are specified when database is created
  - Can indexed multiple attributes, separated by comman

```
this.version(1).store({
  cart: '++cartId,username'
})
```

  - Do not indexed 'blobs' eg. images, MP3
- First field is the primary key, the rest are indexed attributes

- Non auto-incremented primary key

```
cart: 'username'
```

- Auto incremented primary key
  - Attribute type must be number

```
cart: '++cartId'
```

- Compound primary key

```
cart: '[cartId+username]'
```

- See https://dexie.org/docs/Version/Version.stores()#indexable-types

Dexie Examples

- Return the entire collection
  - `toArray()` returns then entire collection as an array

    ```
    const carts: Cart[] = await this.cart.toArray()
    ```

- Return 50 documents starting from the 50th document
  ```
  const carts: Cart[] = this.cart
      .offset(50).limit(50)
      .toArray()
  ```

- Processing one document at a time
  ```
  this.cart
      .orderBy('date').reverse()
      .each(c => { ... })
  ```

Dexie Examples

- **Find a document by primary key**

```
const cart: Cart = await this.cart.get(12345)
```

- **Find documents**

```
this.cart.filter(c => c.date > yesterday).toArray()

this.cart.where('username').equals('fred').toArray()

this.cart.where('username').equals('fred')
    .and(c => c.date > yesterday).
    .each(c => { ... })
```

Dexie Examples

- Creating or updating a document
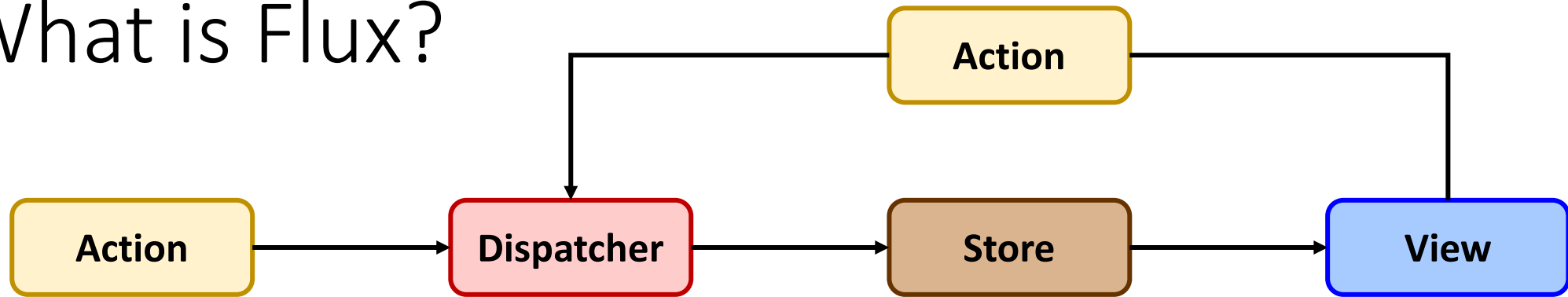 - Document will be inserted if it does not exists, based on the primary key

```
const cart: Cart = await this.cart.get(12345)
// changes cart, write update back
...
await this.cart.put(cart)
```

- Delete on or more documents
  - Returns the number of document deleted

```
const deleteCount = await.this.cart
    .where('name').anyOf('fred', 'barney')
    .and(c => c.date < someDate)
    .delete()
```

What is Flux?

- Flux is an architectural pattern which runs a unidirectional data flow
  - A dispatcher to perform updates to the data store
  - Alternative unidirectional data flow architecture is Redux
- Major components
  - Actions describes the modification to be performed on the store
  - Dispatcher coordinates operations (actions) on the store
  - Store holds data and application state for the entire application
  - Views subscribes to store events like value changes; receive notification on store events

Example - Flux

```
@Injectable
public class CartStore extends Dixie {
  store!: Dixie<Cart, number>
  constructor(private logger: Logger) {
    super('carts')
    this.version(1).stores({ ... })
  }
  on: Subject<Operation<Cart>>()
  async create(cart: Cart) {
    await this.store.put(cart)
    logger.info(`create: ${cart.id}=${JSON.stringify(cart)}`)
    this.on.next({ action: 'create', newData: cart } as Operation<Cart>)
  }
  async update(cart: Cart) {
    const oldData = await this.cart.get(cart.id)
    await this.cart.put(cart)
    logger.info(`update: ${cart.Id}=${JSON.stringify(oldData)}=>${JSON.stringify(cart)}`)
    this.on.next({ action: 'update', key, oldData, newData: cart } as Operation<Cart>)
  }
  ...
}
```

```
export interface Operation<T> {
  action: string
  newData?: T
  oldData?: T
}
```

Create and interface for the store operations

Fire an event whenever the an operation is performed on the store

Example - Flux
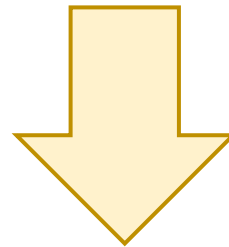
@NgModule({
  provider: [CartStore],
  ...
})
export AppModule { }

Provide the store as a Service

Whenever an operation happens in the store,
subscribers will be notified of the operation

```
constructor(private custStore: CartStore) { }

process() {
  const newCart = ... //Create new cart
  this.custStore.create(cart)
}
```

ACTION

```
constructor(private custStore: Store<Customer>) { }

ngOnInit() {
  this.custStore$ = this.custStore.on(
    event => {
      switch event.action {
        case 'create':
          ...
      }
    }
  )
}
```