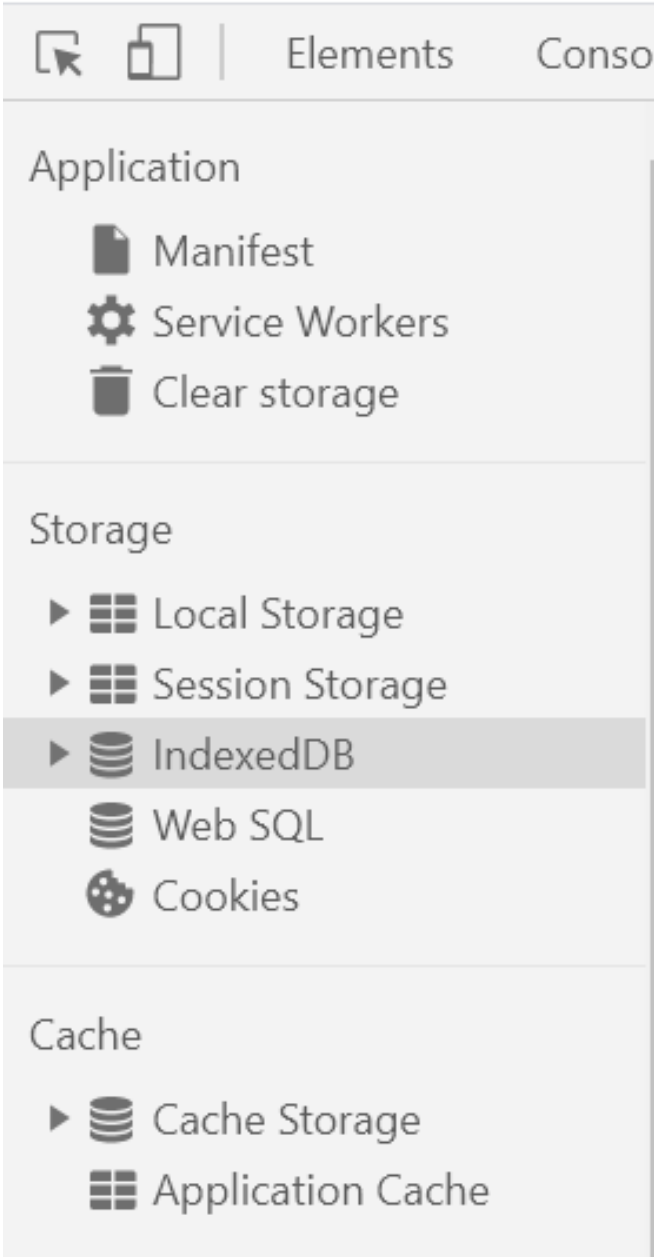
Day 37

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

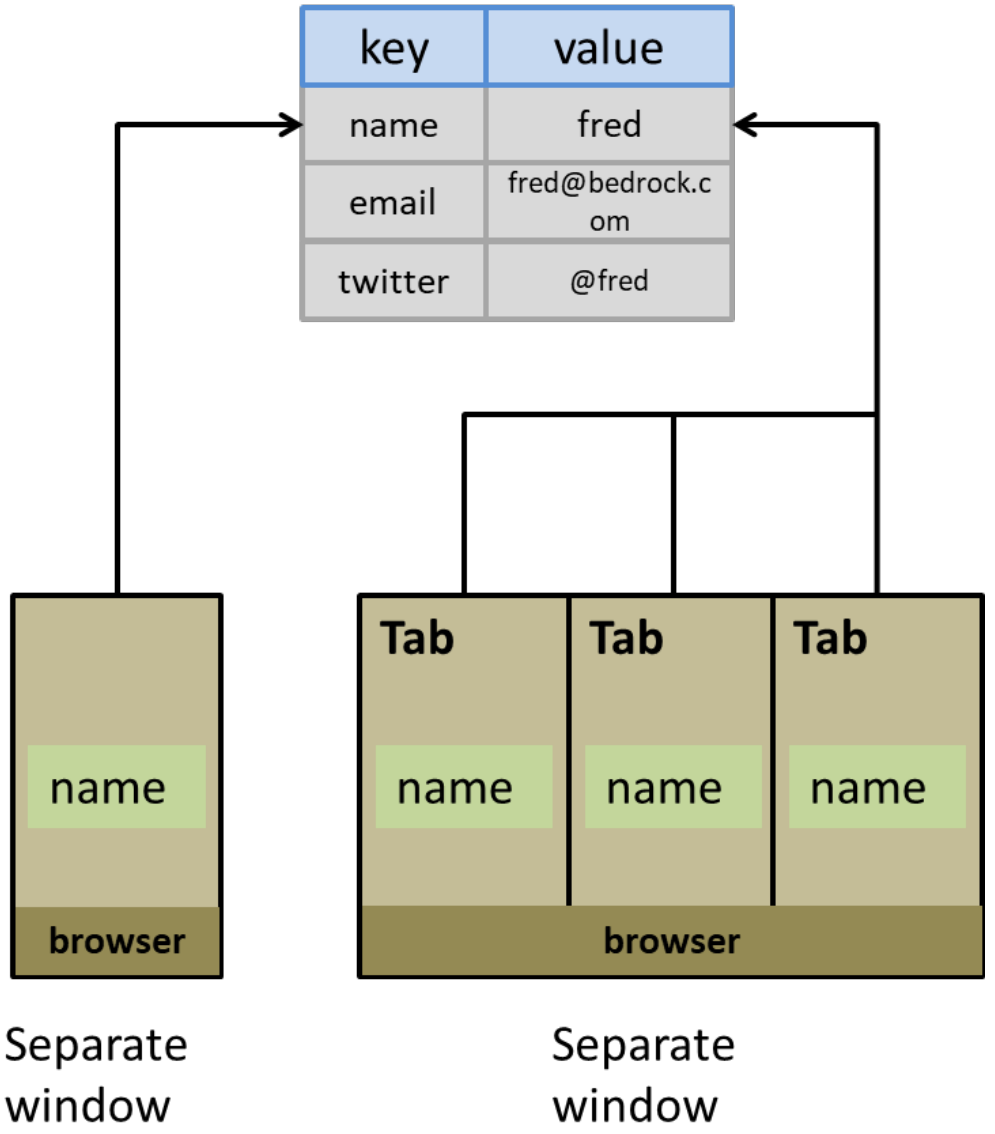
- Cookies
  - Used by web application to save information on the client

  - 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



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

```
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')
  }
}
```

Extend Dexie

Schema version

Collection name

Database name

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

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

```
export interface Item {
  description: string
  price: number;
}
export interface Cart {
  cartId: number;
  username: string;
  date: numner;
  contents: Item[];
}
```

Schema of the document

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'
```

- Composite 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 => { ... })
```

CRUD

- Add a new cart

```
await this.cart.add(newCart)
```

- Bulk add

```
const newCarts: Cart[] = [ ... ]
await this.cart.bulkAdd(newCarts)
```

CRUD

- 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()
```

Example of Managing Data

Event to notify when a new cart is added

```
@Injectable()
export class CartSerivce extends Dexie {

  private cart: Dexie.Table<Cart, String>
  onNew$ = new Subject<Cart>()

  constructor() { ... }

  addCart(cart: Cart) {
    this.cart.add(cart)
      .then(() => onNew$.next(cart)
  }
}
```

When a new cart is added notify subscribers

State Management

- Lots of data (state) in large Angular application. For efficiency, the states are held on the client and can be shared by many components
  - Create or mutate the data: insert, update, delete
  - Query these data
  - React to data changes
- Some libraries in the Angular ecosystem for state management
  - Ngrx - https://ngrx.io/
  - Ngxs - https://www.ngxs.io/
  - Akita - https://opensource.salesforce.com/akita/

State Management Core Concepts

- Store - the data store. Can hold more or more logical states
  - Eg. inventory, customer
  - Each of these logical states is call a slice
- Action - instructions are dispatched to perform CUD operations
  - Eg. create a new customer
  - Action holds the value
- Reducer - operations that changes the state in the store
  - Eg. the create customer reducer inserts a new customer into the store
  - Components do not directly modify the store
- Selector - state queries
- Effects - actions that result in side effect
  - Eg. a delete action may cause a HTTP request to the server to delete the data

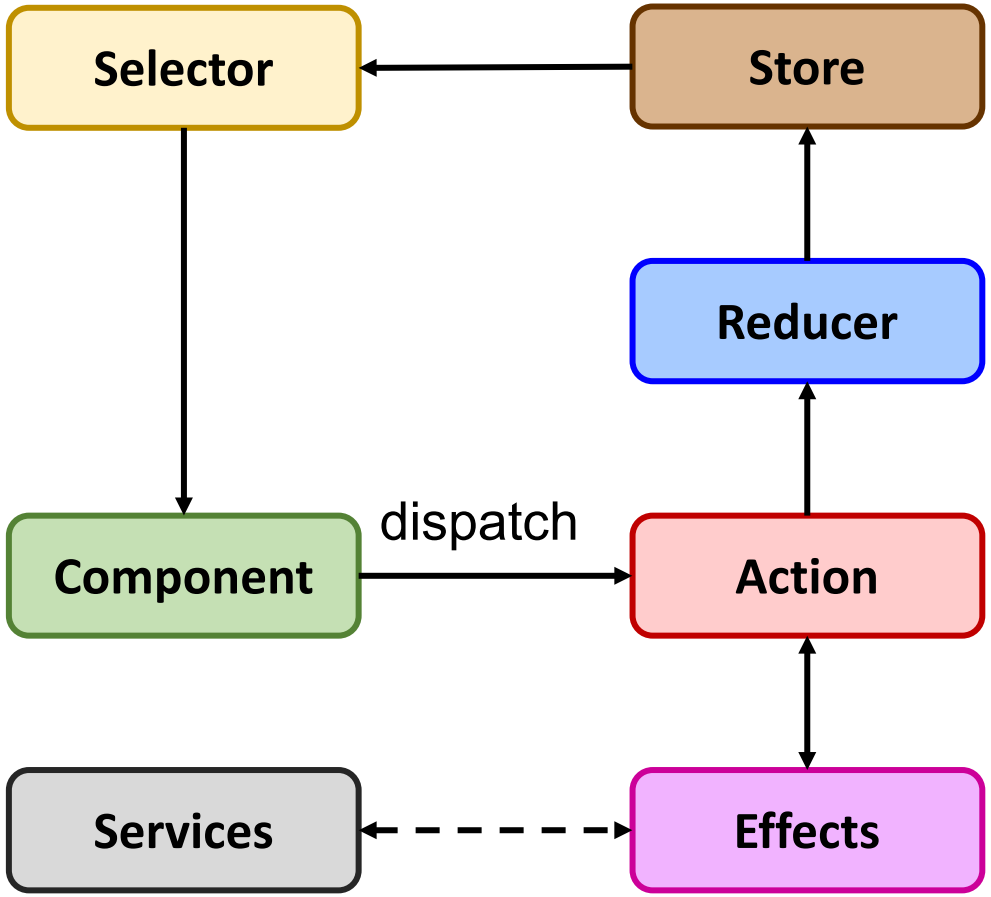


Image adapted from https://ngrx.io/generated/images/guide/store/state-management-lifecycle.png

NgRX Component Store

- A standalone, lightweight state management library
  - Use when you are writing "Service with subjects" to manage your application state
  - The entire store is implemented in a single class, does not use actions
- Use case is to manage data from a module in a large Angular application
  - We will use it to manage the state of the entire application
  - Goal is to introduce the state management concepts
- Use `@ngrx/store` for more control but more complicated

```
ng add @ngrx/component-store
```

Store Initialization

```
const INIT_STATE: TodoSlice = {
  todos: []
}

@Injectable()
export class TodoStore
   extends ComponentStore<TodoSlice> {

  constructor() {
    super(INIT_STATE)
  }

  ...
}
```

Don't forget to register service

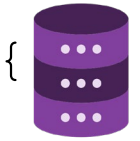
Create a store for `TodoSlice`

Initialize the store

```
export interface Task {
  name: string
  priority: string
  duration: number
}
```

```
export interface Todo {
  id: string
  name: string
  createdOn: number
  tasks: Task[]
}
```

```
export interface TodoSlice {
  todos: Todo[]
}
```



The slice (table) for all the `Todo` data

Store Initialization with `OnStoreInit`

```
@Injectable()
export class TodoStore extends ComponentStore<TodoSlice>
    implements OnStoreInit {

  private todoSvc = inject(TodoService)

  constructor() { super(INIT_STATE)}

  ngrxOnInitStore(): void {
    this.todoSvc.getAllTasks()
      .then(todos => this.setState({ todos }))
  }
}
```

Assume a `TodoService` which loads `Todos` either removely over HTTP or from IndexDB

Lifecycle method to initialize the store

`TodoSlice`

Update the entire state of the store with `setState({...})`

Use case: populate the component store from an external data source

Updating the Store

- Use the `updater<T>(update)` method to update the store
  - `T` is the type to be passed to the update function
- Takes an update function to update the store
  - Update function signature contains has 2 parameters: slice and `T`
  - slice is the current store; `T` is the value to update the slice
- Returns a function with 1 parameter of the type `T`
- The update function must create a new instance of the slice with the mutations
  - Not change the slice that is passed into the update function
  - Return the mutated slice

Store Mutation - Add New `Todo`

```
@Injectable()
export class TodoStore extends ComponentStore<TodoSlice> {

  readonly addNewTodo = this.updater<Todo>(
    (slice: TodoSlice, todo: Todo) => {
      const newSlice: TodoSlice = {
        todos: [ ...slice.todos, todo ]
      }
      return newSlice
    }
  )
}
```

Update takes a `Todo` object
Type of the second parameter

First parameter
is the slice

Update the slice by
creating a new instance

Return the updated slice

```
const todo: Todo = this.form.value
this.todoStore.addNewTodo(todo)
```

The return updater function has 1 parameter;
the type is the parameterized type of `updater`

Store Mutation - Delete Todo

```
readonly deleteTodoById = this.updater<string>(
  (slice: TodoSlice, id: string) => (
    {
      todos: slice.todos.filter(todo => todo.id !== id)
    } as TodoSlice
  )
)


var todoId: string = // TodoId
this.todoStore.deleteTodoById(todoId)
```

Store Query

- The `select` method creates a query, returns an `Observable`
- Pass a query function with the store as the parameter

```
readonly getTodoSummaries = this.select(
  (slice: TodoSlice) => slice.todos.map(
    todo => ({
      id: todo.id,
      name: todo.name,
      count: todo.tasks.length
    } as TodoSummar)
  )
)
```

```
export interface TodoSummary {
  id: string
  name: string
  count: number
}
```

```
this.todoSummaries$ = this.todoStore.getTodoSummaries
```

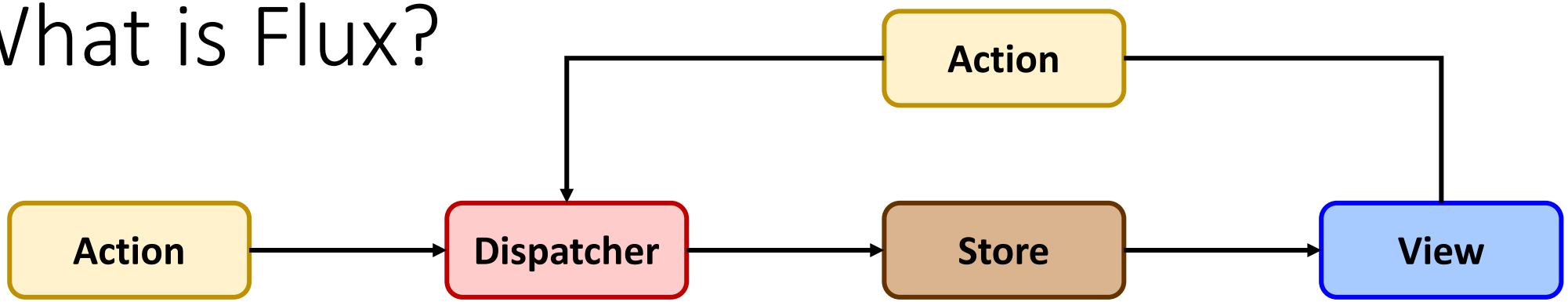
Store Query

```
<span (click)="showDetails(todo.id)"> ... </span>
<div *ngIf="todo$ | async as t"> {{ t | json }}</div>


showDetails(id: string): void {
  this.todo$ = this.todoStore.getTodoById(id)
}


this.getTodoById = (id: string) =>
   this.select(
     (slice: TodoSlice) =>
       slice.todos.find(todo => todo.id === id)
   )
```

Pass parameters into queries by using a function to create the query

Unused

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

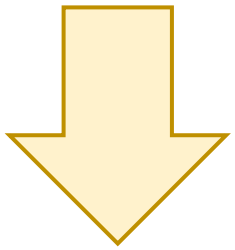
Example - Flux

```
@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)
}
```



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

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