



Day 37



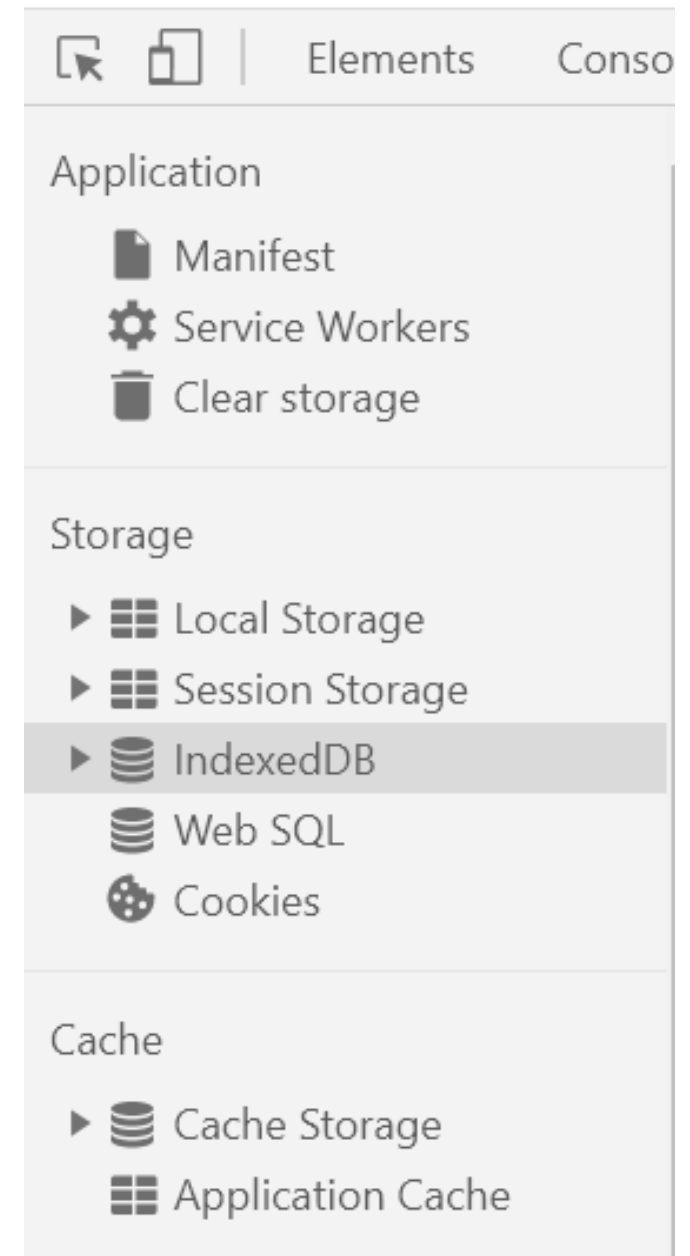
Client Side Storage

- 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



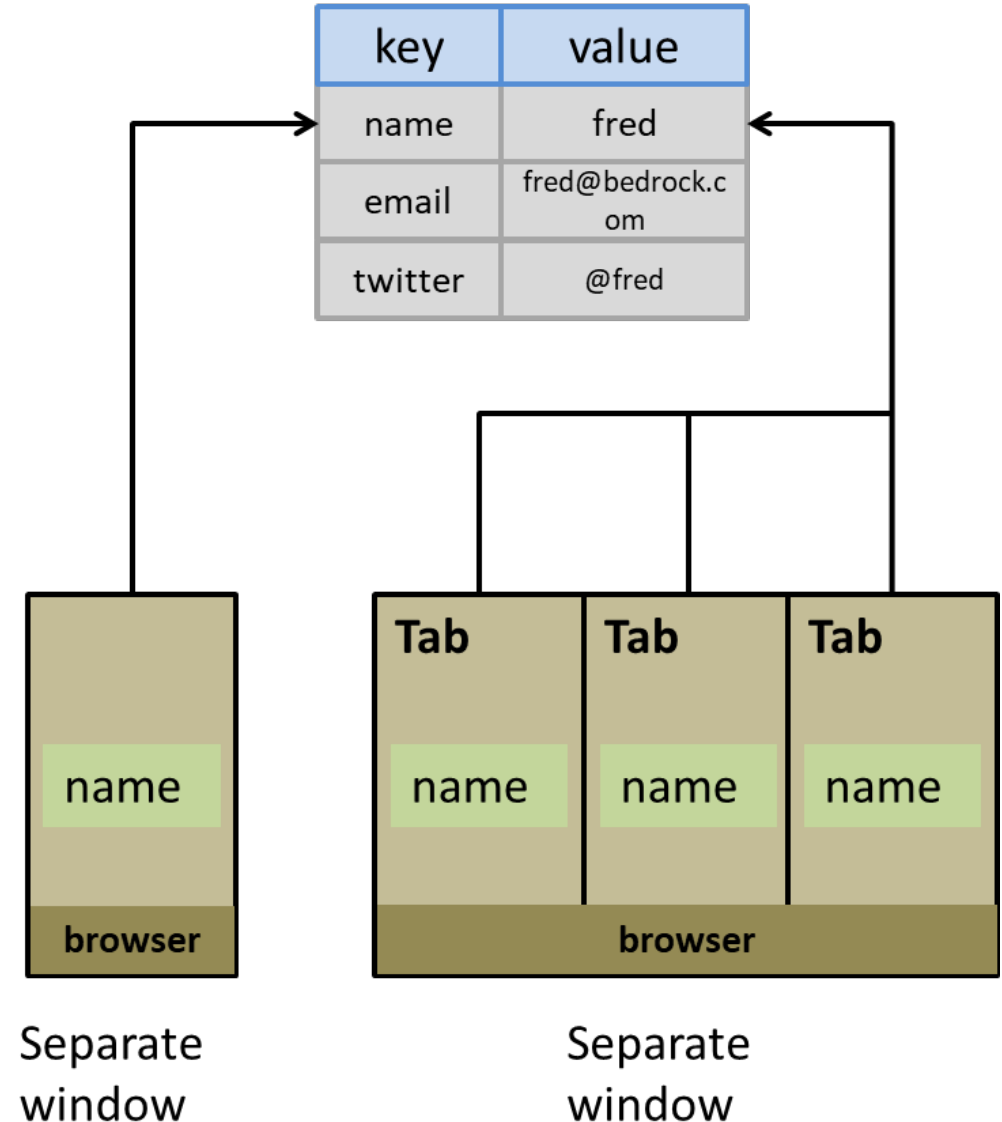
- Information saved as cookies are returned to the web application whenever the browser makes a request to the same server

- 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

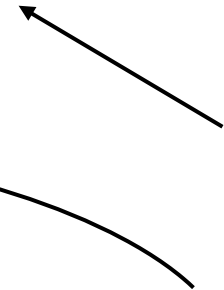


```
export interface Item {
  description: string
  price: number;
}
export interface Cart {
  cartId: number;
  username: string;
  date: number;
  contents: Item[];
}
```

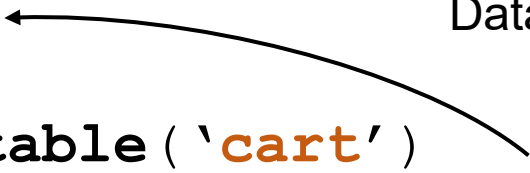
Schema of the document



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



Database name

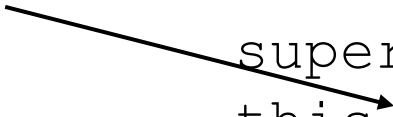


One or more attributes to be
indexed. Can annotate fields with
special characters to indicate the
type of index

Hold a reference
of the collection



Schema
version



Collection
name





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 comma
- ```
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](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 50<sup>th</sup> 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 exist, based on the primary key

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

- Delete one or more documents

- Returns the number of documents deleted

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



# Example of Managing Data

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

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

 constructor() { ... }

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

Event to notify  
when a new cart  
is added



When a new cart is  
added notify subscribers





# State Management

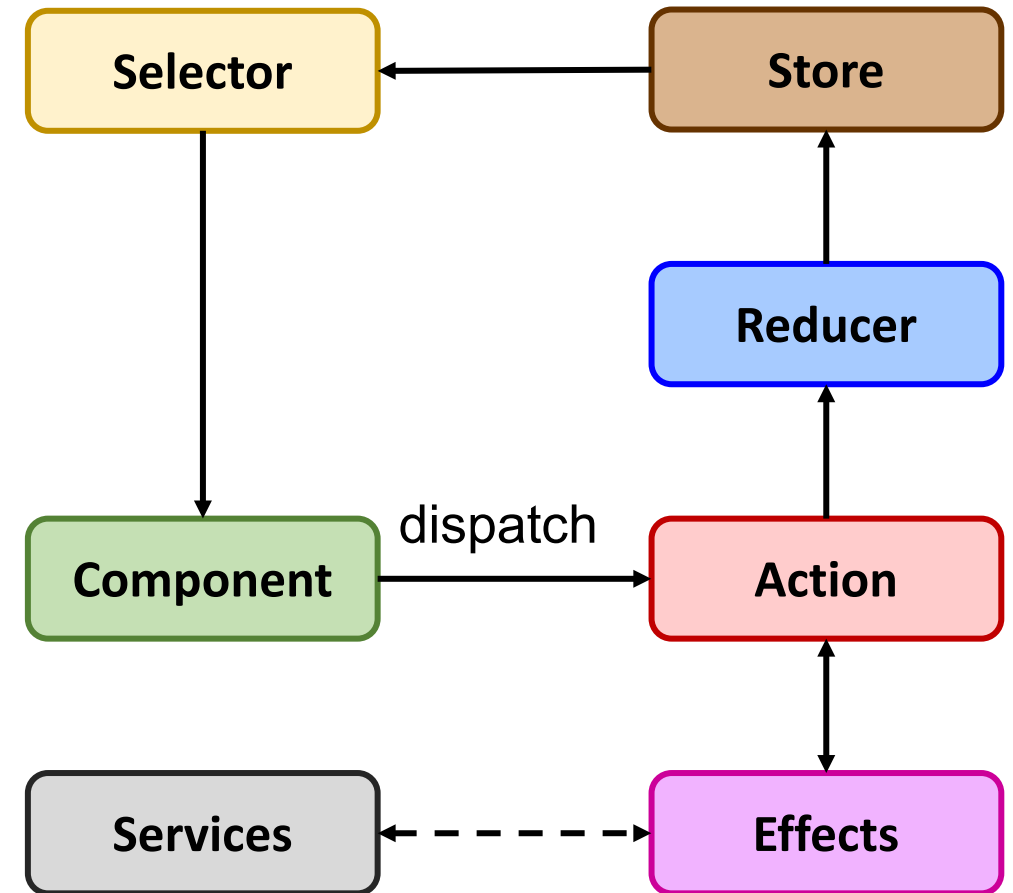
- Lots of data (state) in large Angular application. For efficiency, the states are held on the client and can be shared by many components
  - Create or mutate the data: insert, update, delete
  - Query these data
  - React to data changes
- Some libraries in the Angular ecosystem for state management
  - 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





# 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: []
}
```

Don't forget to register service

**@Injectable()**

```
export class TodoStore
```

```
 extends ComponentStore<TodoSlice> {
```

```
 constructor() {
```

```
 super(INIT_STATE)
```

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)
```

Assume a `TodoService`  
which loads `Todos` either  
remotely over HTTP or  
from IndexedDB

```
 constructor() { super(INIT_STATE) }
```

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

`Todoslice`

Lifecycle method to  
initialize the store

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
- Provide 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

Component Store passes the existing slice to the mutation function

The parameterized type of the update function. This is the input of the update function, the second parameter

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

The update function

The function that performs the store mutation. Has 2 parameters: the existing slice and the new element: Todo  
Tell Component Store HOW the slice should be updated

Creates a copy of the existing slice with the new element. Returns the new slice



# Store Mutation - Add New Todo

`updater` returns a function with 1 parameter which is assigned to `addToCart` with the following signature

```
addToCart (todo: Todo) : TodoSlice {
 ...
}
```

Use the function to add a new `Todo` to the slice by passing in a `Todo` object

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



# Store Mutation - Add New Todo

```
this.todoStore.addNewTodo (todo)
```

Internal copy of `TodoSlice`  
from `TodoStore`

Argument to `addNewTodo` is  
bound to the second parameter  
of the update function

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

The update function that  
you supplied to `updater`  
when creating  
`addNewTodo`

New slice stored internally by `TodoStore`





# 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)
```



# Querying the Store

- `select<T>` method creates a query returning an `Observable`
  - `T` is the type return by the query
- Provide a query function to tell Component Store how to query the slice
  - 1 parameter which is the current state of the slice
  - Returns the result as a type of `T`
- Select function must not alter the slice
  - No side effects



# Store Query - Retrieving Todo Summary

Component Store passes the existing slice to the query function

The query function

The parameterized type of select defines the type returned by the query

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

The function that performs the query. Takes a single parameter which is the slice. The slice represents the current state of the store

The result of the query function. The result must match the parameterized type



# Store Query - Retrieving Todo Summary

`select` returns a function with which is assigned to `getTodoSummaries` with the following signature

```
getTodoSummaries () : Observable<TodoSummary[]> {
 ...
}
```

Use the function to get notification whenever `TodoSlice` changes

```
summary$: Observable<TodoSummary[]> =
 this.todoStore.getTodoSummaries()
```



# Store Query with Parameters

- Create queries that return a subset of the store rather than the entire store
  - Eg. `Todo` for user `fred`
- Use closure to capture the parameter by returning the select function in an enclosing function

```
readonly getTodoByName = (name: string) =>
 this.select<Todo[]>(
 (slice: TodoSlice) => slice.todos.filter(
 todo => name.toLowerCase() === todo.name.toLowerCase()
)
)
```



# Store Query - Retrieving Todos by Name

```
todos$: Observable<Todo[]> = this.getTodoByName('fred')
```

```
(name: string) =>
 this.select<Todo[]>(
 (slice: TodoSlice) => slice.todos.filter(
 todo => name.toLowerCase() === todo.name.toLowerCase()
)
)
)
```

Call the function to create the selector. The parameter is bound to the returned function

select returns a query function that closes over the free variable name when the select function is returned to the caller

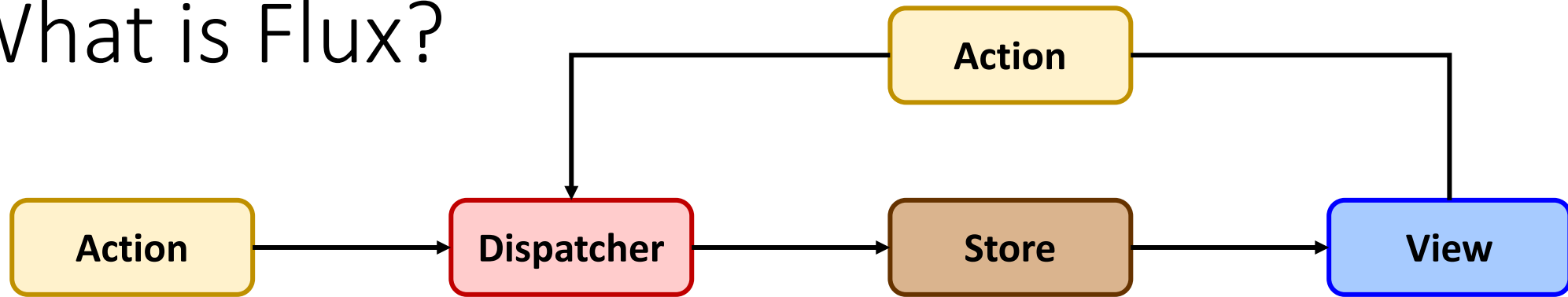
```
() => (slice: TodoSlice) => slice.todos.filter(
 todo => 'fred'.toLowerCase() === todo.name.toLowerCase()
)
```



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 { }
```

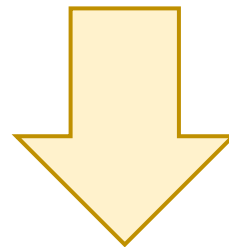
## Provide the store as a Service

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':
 ...
 }
 }
)
}
```





# 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
 }
)
}
```

First parameter is the slice

Update takes a Todo object  
Type of the second parameter

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 Query

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

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

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

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



# Store Query

```
 ...
<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