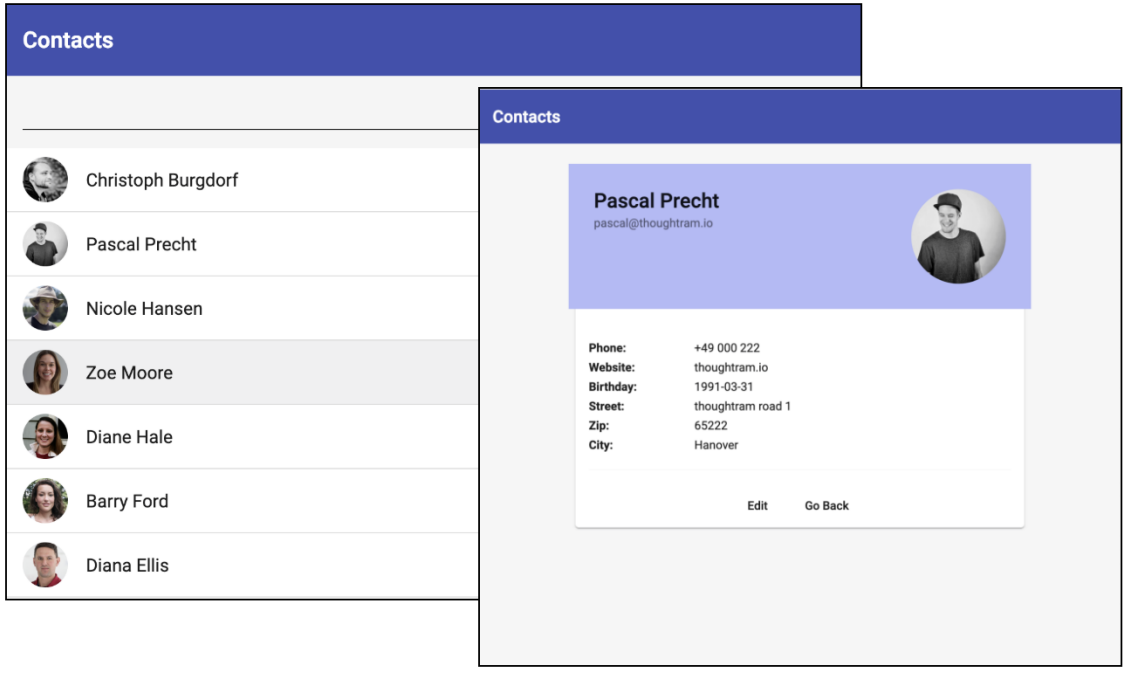
TradeMe Angular Master Class

+ NgRx



State Management with **NgRx**

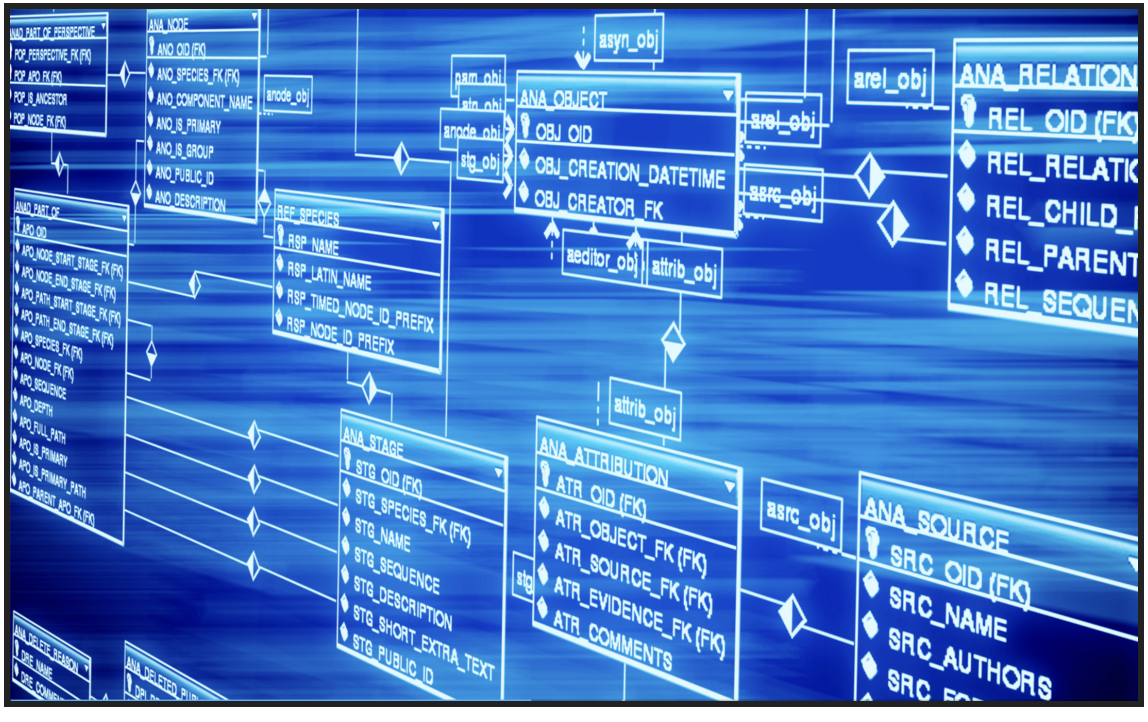
The key to scalable, maintainable enterprise Angular applications is **NgRx** and proper **state management**.

In this course, developers will learn about state management, **Redux**, and **NgRx** in Angular... critical patterns for testable, DRY applications.

NgRx State

Lists of data entities are analogous to SQL Tables

NgRx state are `lists of entities` AND more...



What is **State**?

- **Authentication** state
- **Router** state: Url, params
- Domain State
  - Server Responses or cached data
  - **Lists** of entities (eg Ticket, Product, Person)
- **Application** State
  - selected tickets
  - loading in-progress
  - search criteria,
- **UI** State
  - user input,
  - dropdowns,
  - sidebar is open
  - spinners,
  - pagination controls
  - etc.

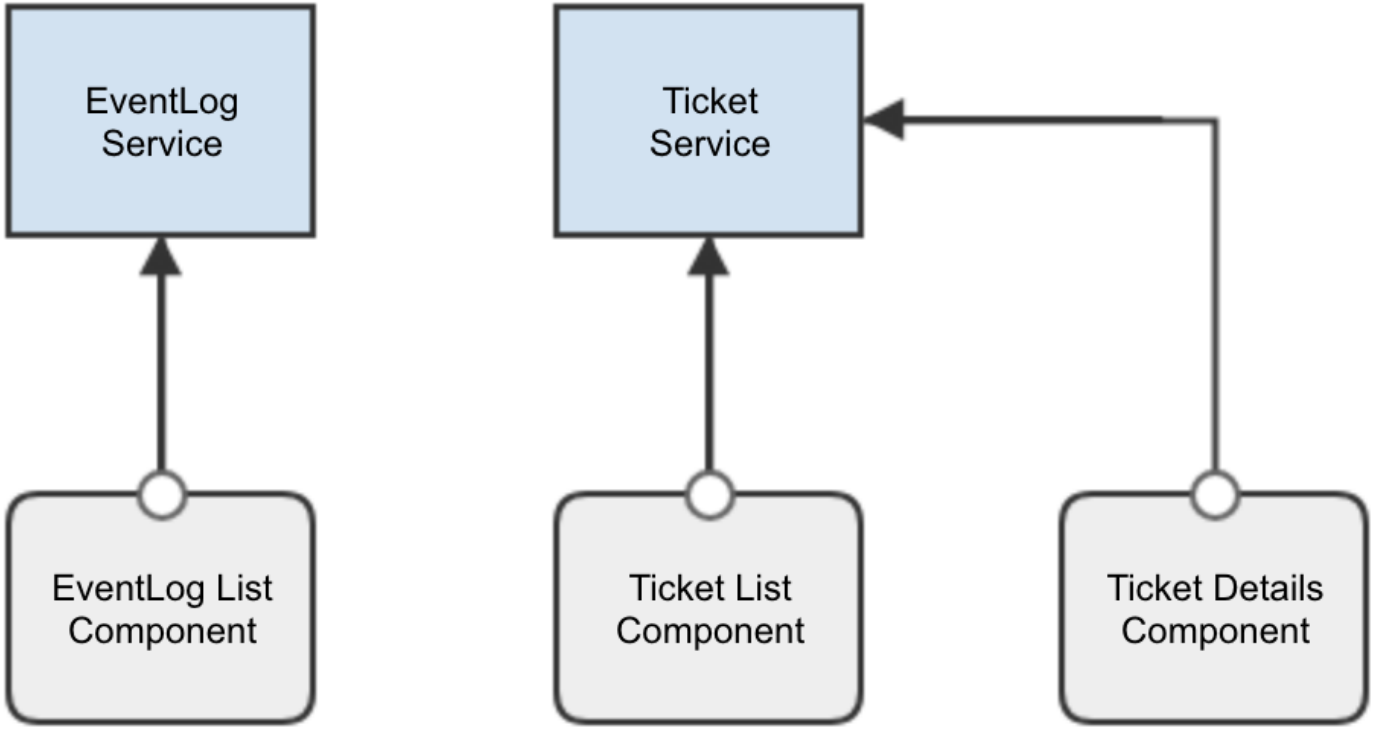
State management is hard!

- Who keeps track of state?
- How can state get changed?
- What state goes where?
- How do we effectively debug?

Speaker notes

Can components modify objects, should a service be responsible?
Do we store those objects on the component, in some data service, in some app storage?
Do we make services for different data types (logs, tickets, etc)?

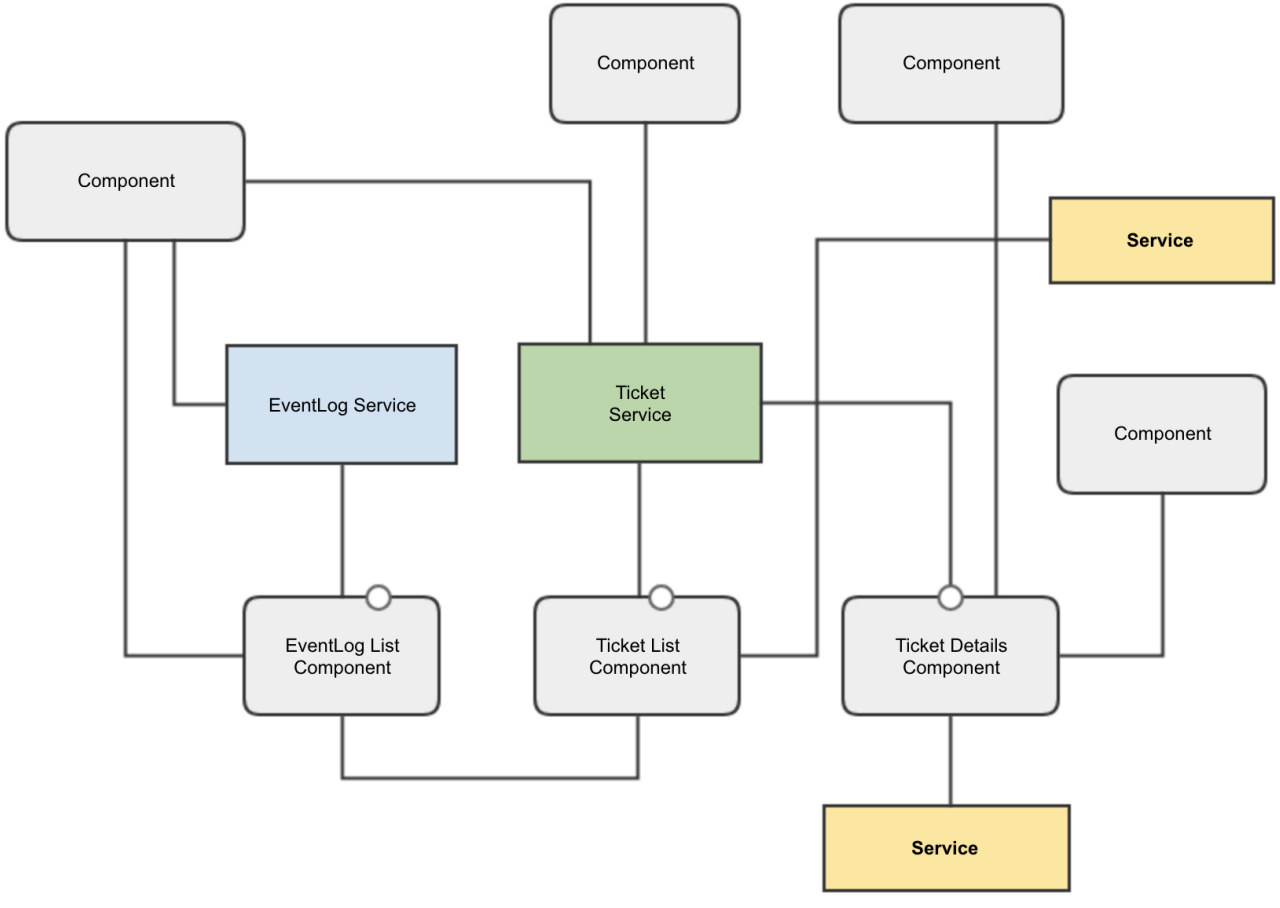
State management with **services**!



Speaker notes

This architecture seems reasonable, at first!
State management starts easy...

State management with **services**!

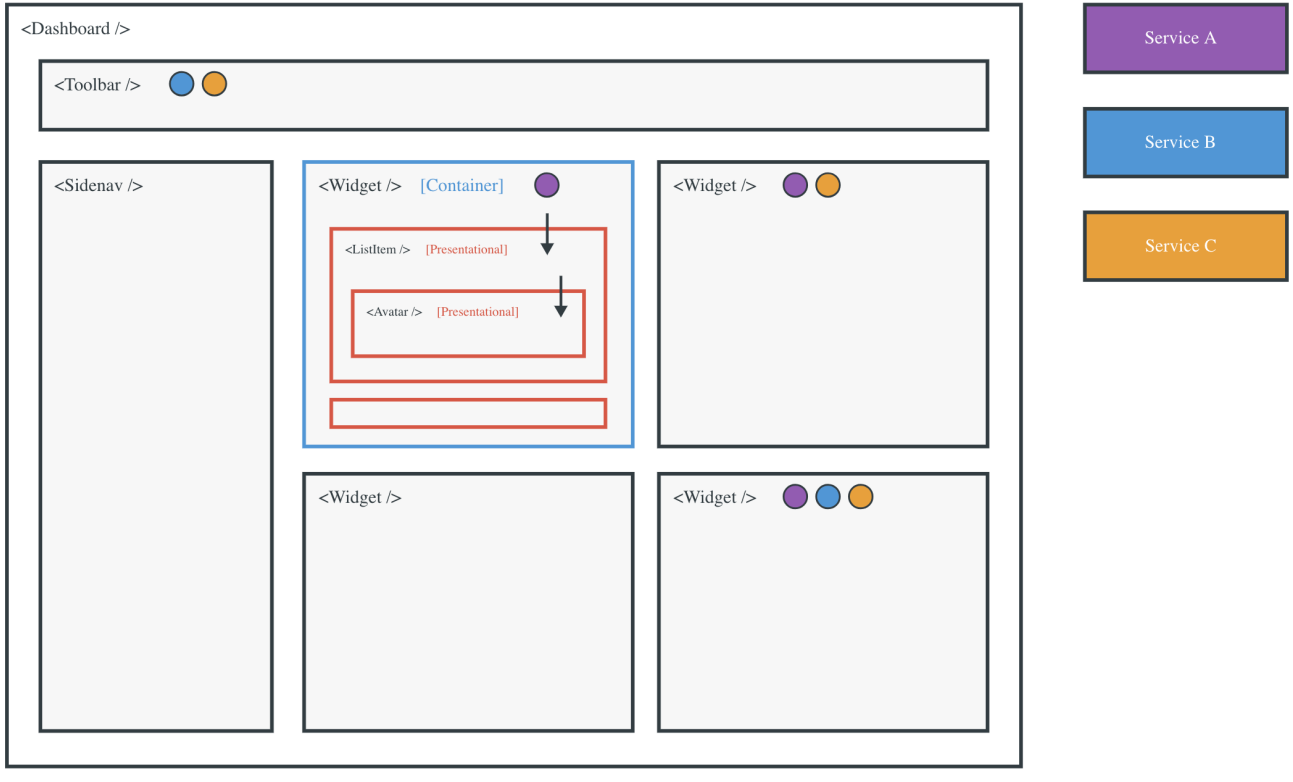


Speaker notes

Soon it gets much more complicated.

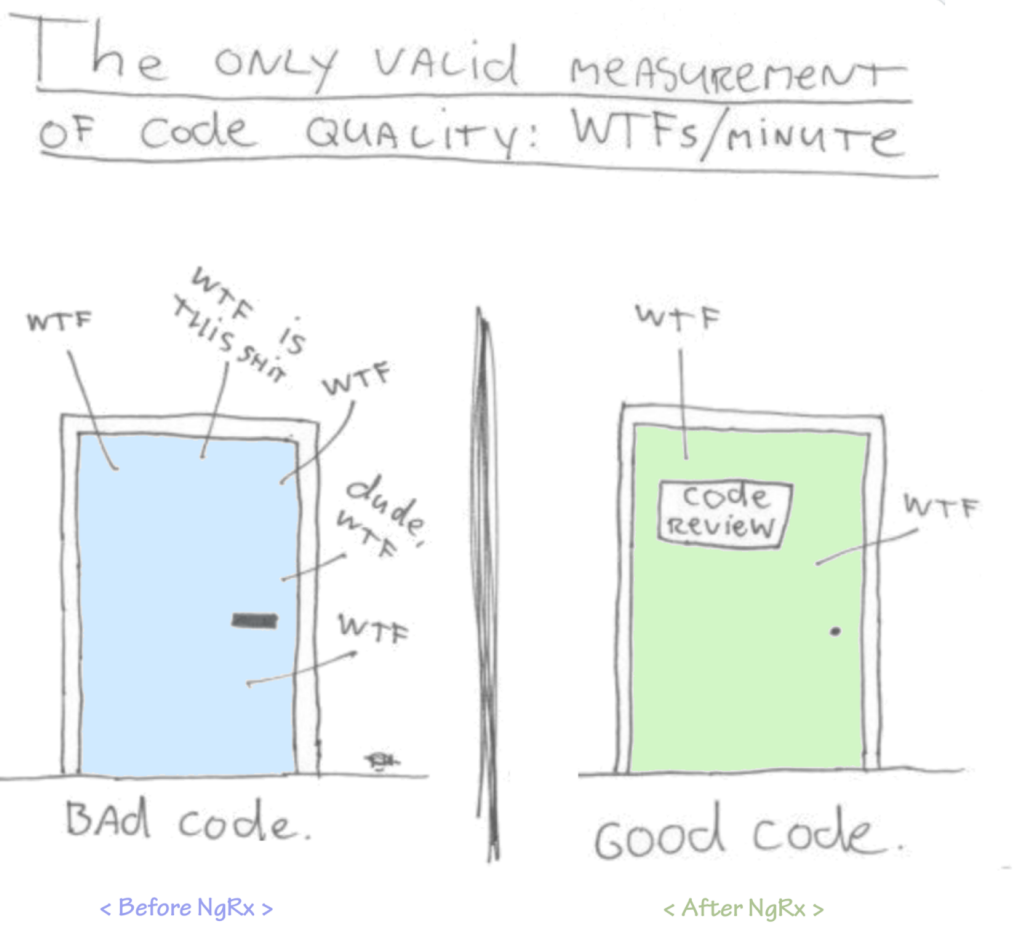
Can you figure out what changes when and where?
How do we track order of operations? How to we recreate a state scenario?

State, Services, and nested Views



How to deal with this complexity?

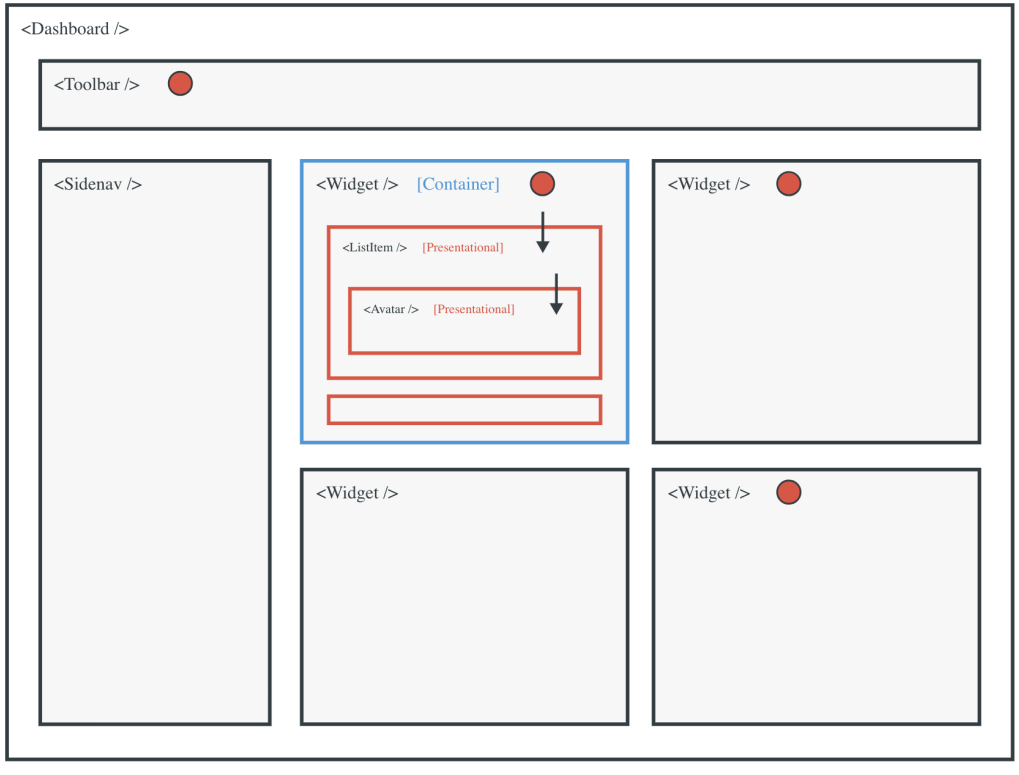
Application Code **WITHOUT** NgRx



< Before NgRx >

< After NgRx >

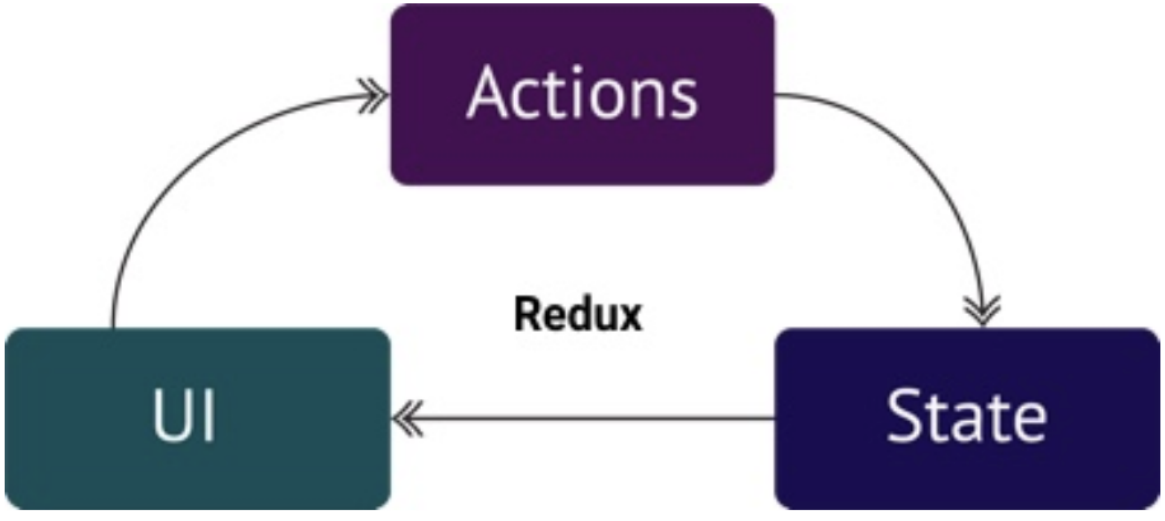
State, Services, and nested Views



NgRx

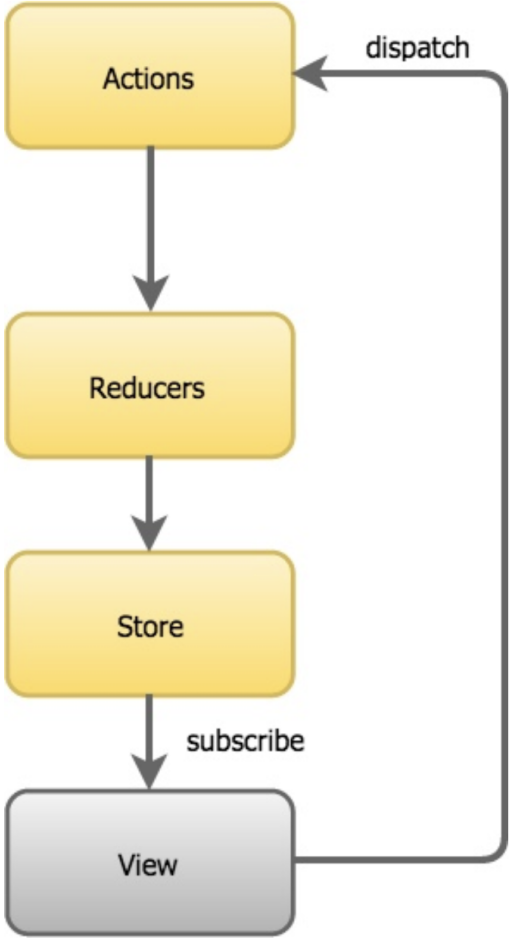
Use **NgRx**

Redux

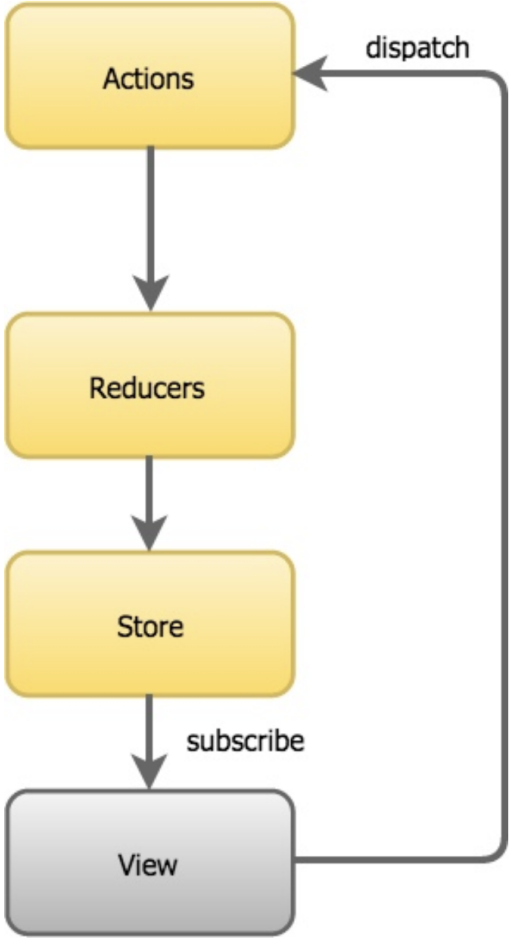


Redux is a **predictable state container** for JavaScript apps, that helps writing applications that behave consistently.

Redux Core Concepts

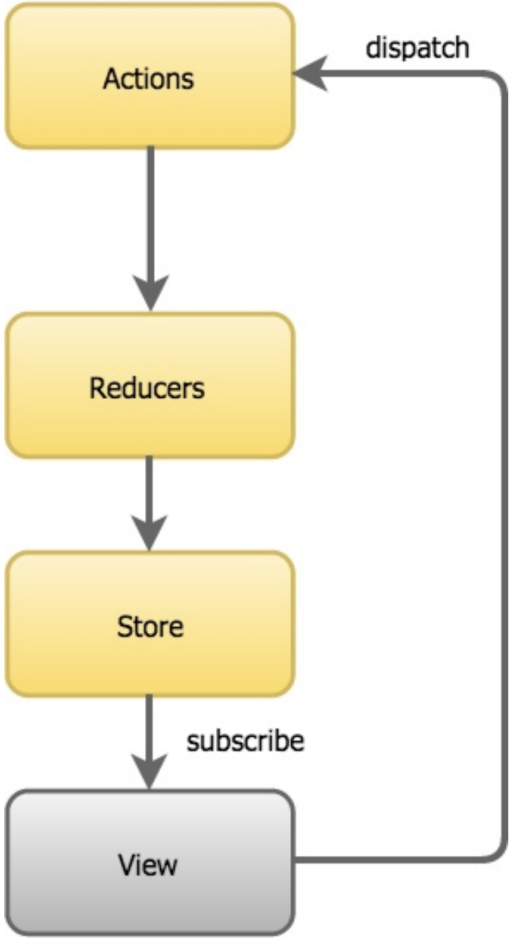


Redux Core Concepts



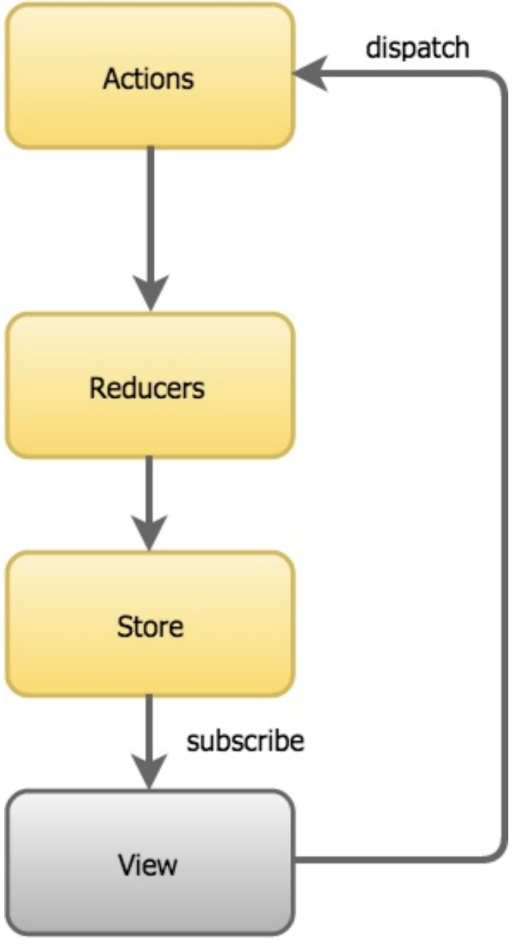
- **State** - State is described as plain object without setters

Redux Core Concepts



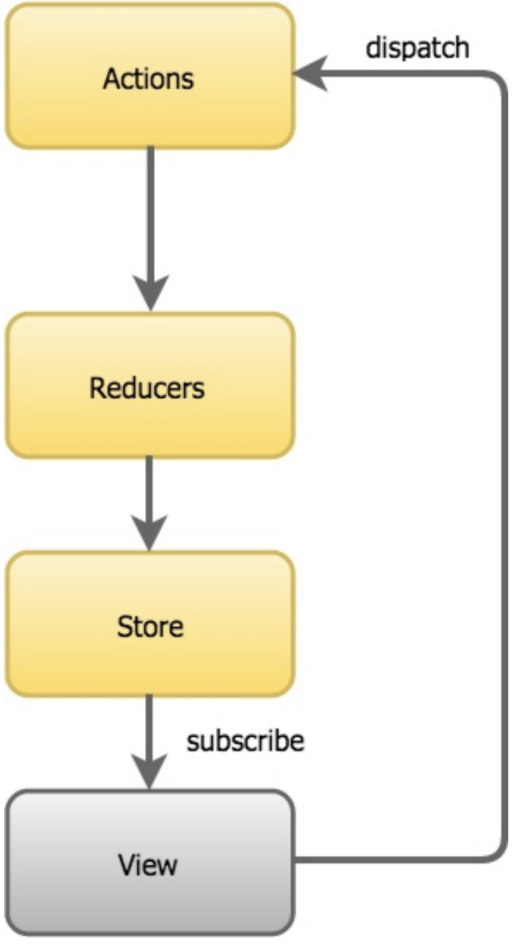
- **Store** - State container
- **State** - State is described as plain object without setters

Redux Core Concepts



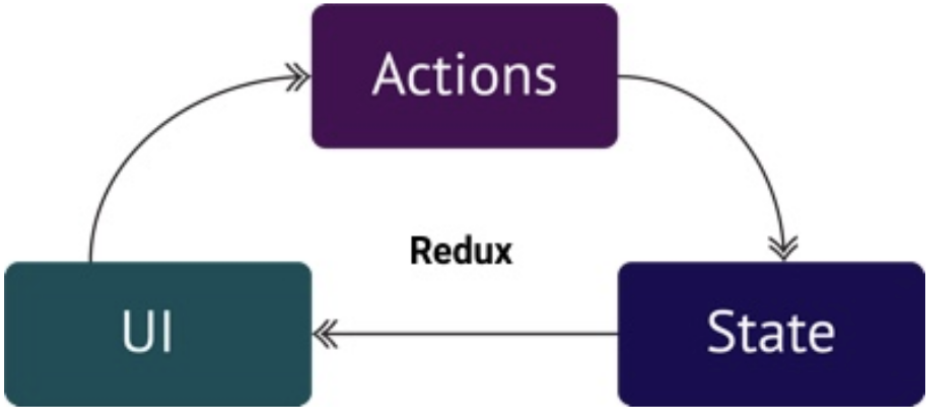
- **Store** - State container
- **State** - State is described as plain object without setters
- **Actions** - A plain object that describes events in our application

Redux Core Concepts



- **Store** - State container
- **State** - State is described as plain object without setters
- **Actions** - A plain object that describes events in our application
- **Reducers** - A pure function that takes state and action as arguments and returns next state

Angular Applications

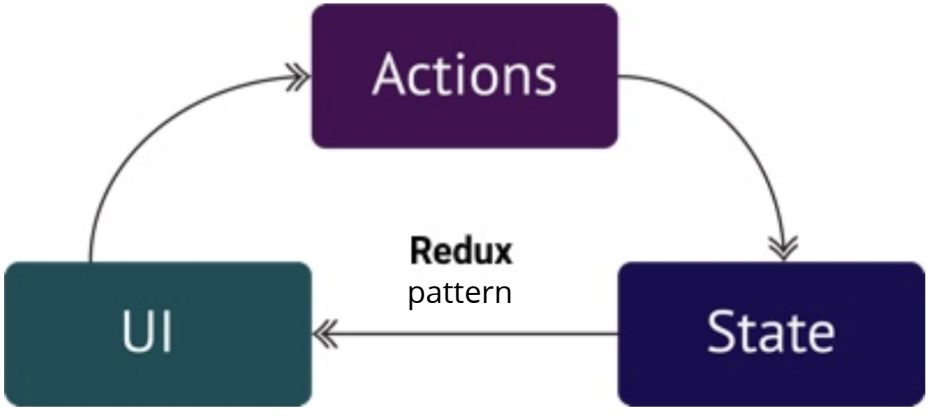


Redux + **RxJS** ==> **NgRx** for Angular

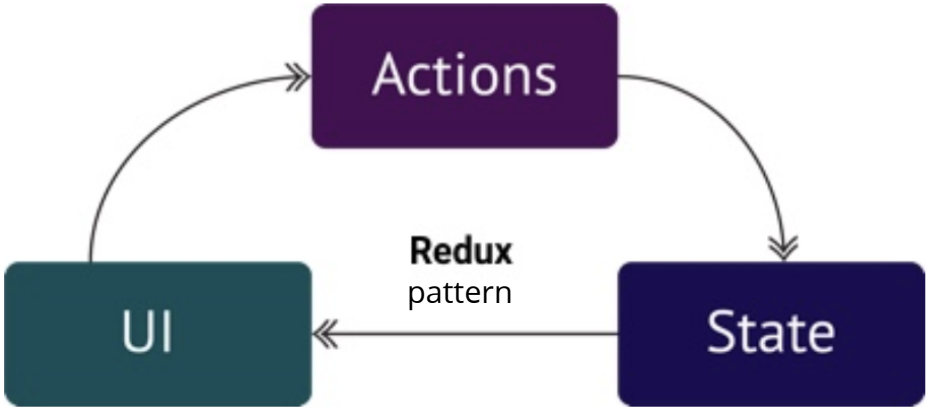
Speaker notes

NgRx to the rescue!
Separation of concerns makes it easier to understand parts of code, easier to track flow, easier to care about only what is important
Component doesn’t have to contain business logic
One (1) central point to track changes... 1-way data flows!

State Management in Angular

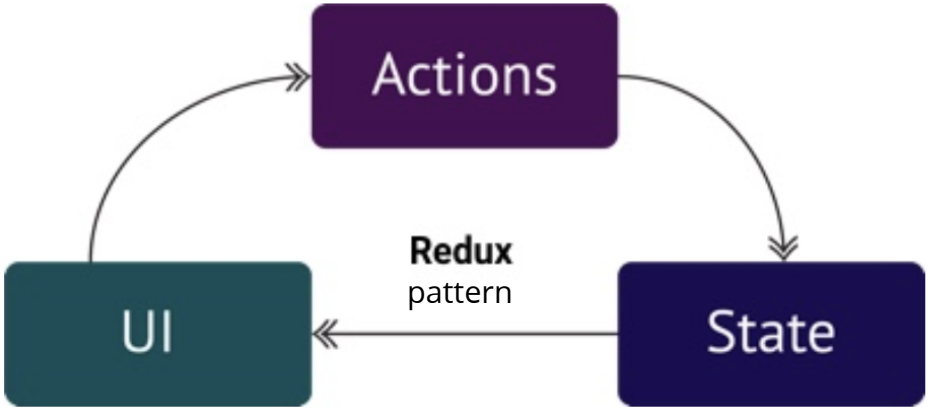


State Management in Angular



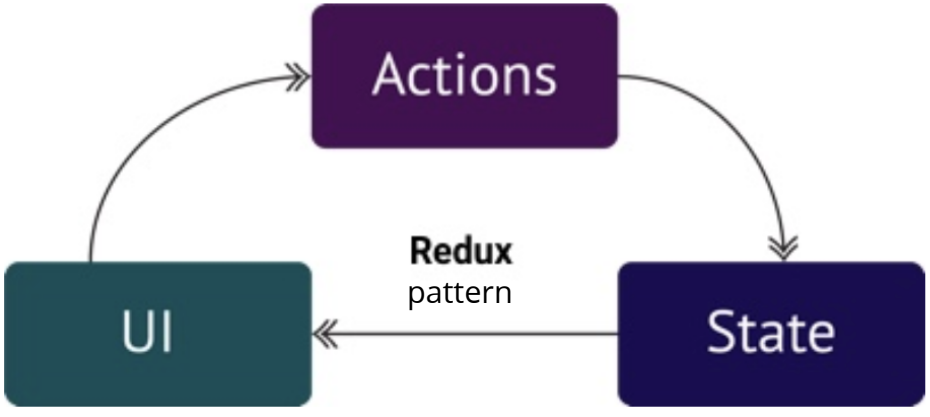
- Services + RxJS

State Management in Angular



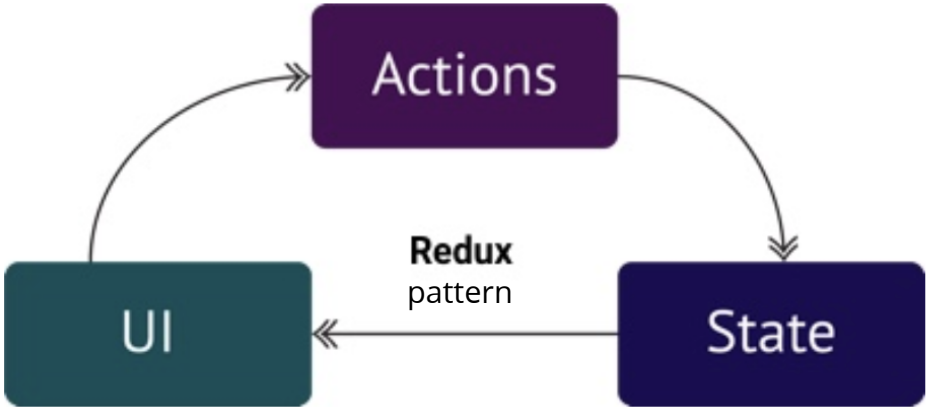
- Services + RxJS
- **@ngrx/platform**

State Management in Angular



- Services + RxJS
- **@ngrx/platform**
- @angular-redux/store

State Management in Angular

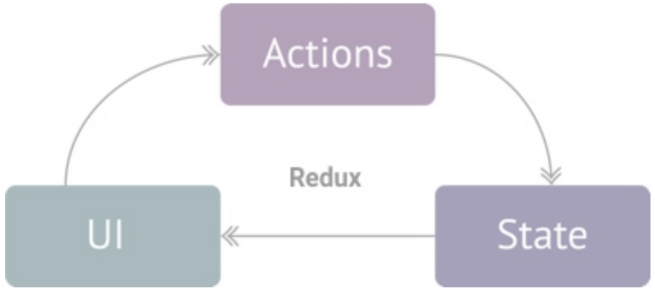


- Services + RxJS
- **@ngrx/platform**
- @angular-redux/store
- @ngxs/store

Speaker notes

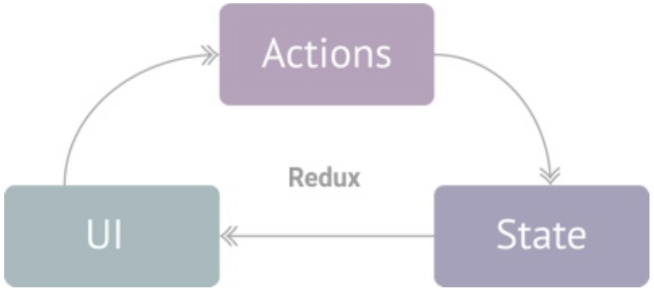
Many developers try to roll their own state management with services and RxJS
Eventually they reach a level of functionality and realize: "This is essentially ngrx!"

NgRx **SHARI**



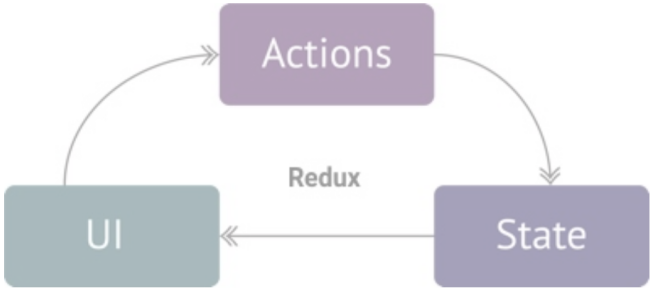
- **S**hared
- **H**ydrated
- **A**vailable
- **R**etrieved
- **I**mpacted

When to manage **State** ?

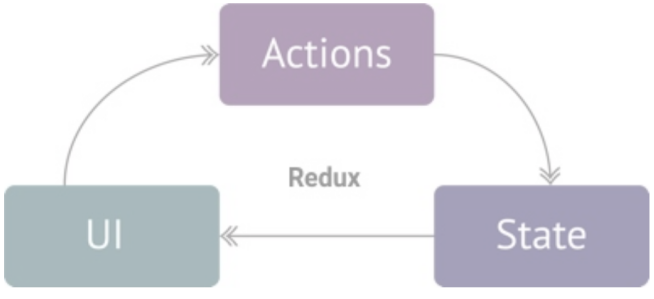


When to manage **State** ?

- State is **shared** between components

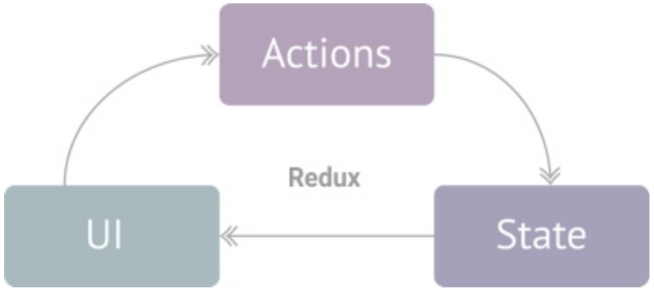


When to manage **State** ?



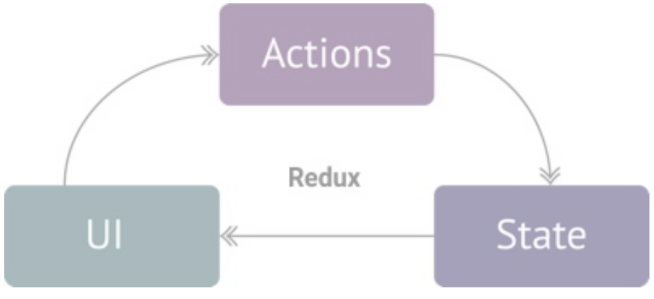
- State is **shared** between components
- State that needs to be persisted and **hydrated** across page reloads

When to manage **State** ?



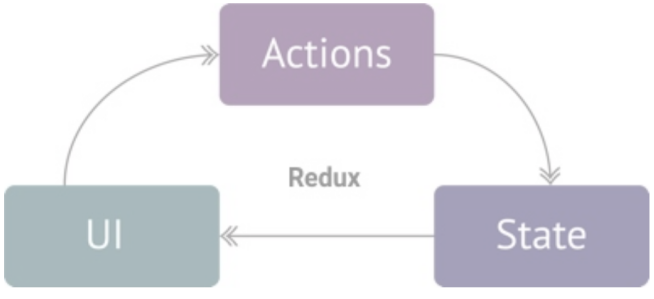
- State is **shared** between components
- State that needs to be persisted and **hydrated** across page reloads
  - State that needs to be **available** when re-entering routes

When to manage **State** ?



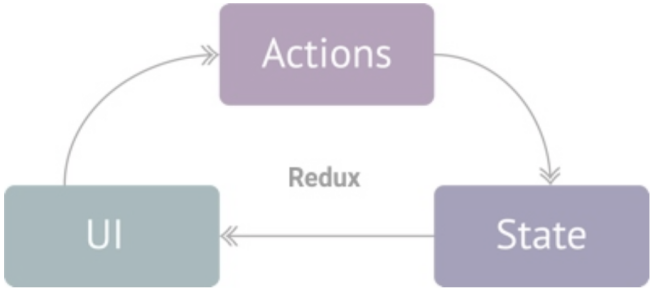
- State is **shared** between components
- State that needs to be persisted and **hydrated** across page reloads
  - State that needs to be **available** when re-entering routes
  - State that needs to be **retrieved** with a side-effect, separating the concerns

When to manage **State** ?



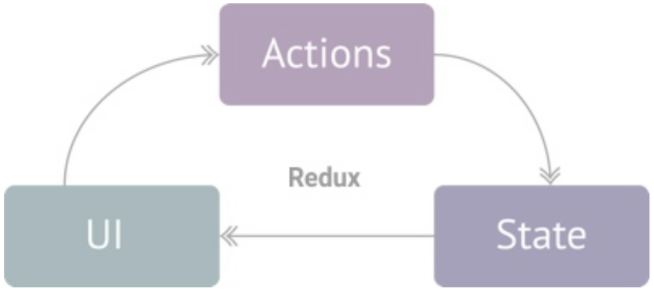
- State is **shared** between components
- State that needs to be persisted and **hydrated** across page reloads
  - State that needs to be **available** when re-entering routes
  - State that needs to be **retrieved** with a side-effect, separating the concerns
- State that is **impacted** by other components or services

When to manage **State** ?



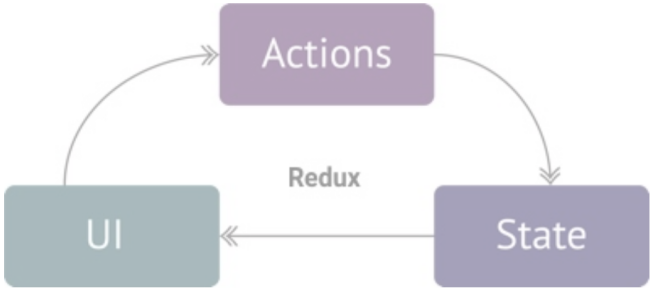
- State is **shared** between components
- State that needs to be persisted and **hydrated** across page reloads
  - State that needs to be **available** when re-entering routes
  - State that needs to be **retrieved** with a side-effect, separating the concerns
- State that is **impacted** by other components or services

When to manage **State** ?



- State is **shared** between components
- State that needs to be persisted and **hydrated** across page reloads
  - State that needs to be **available** when re-entering routes
  - State that needs to be **retrieved** with a side-effect, separating the concerns
- State that is **impacted** by other components or services

When to manage **State** ?



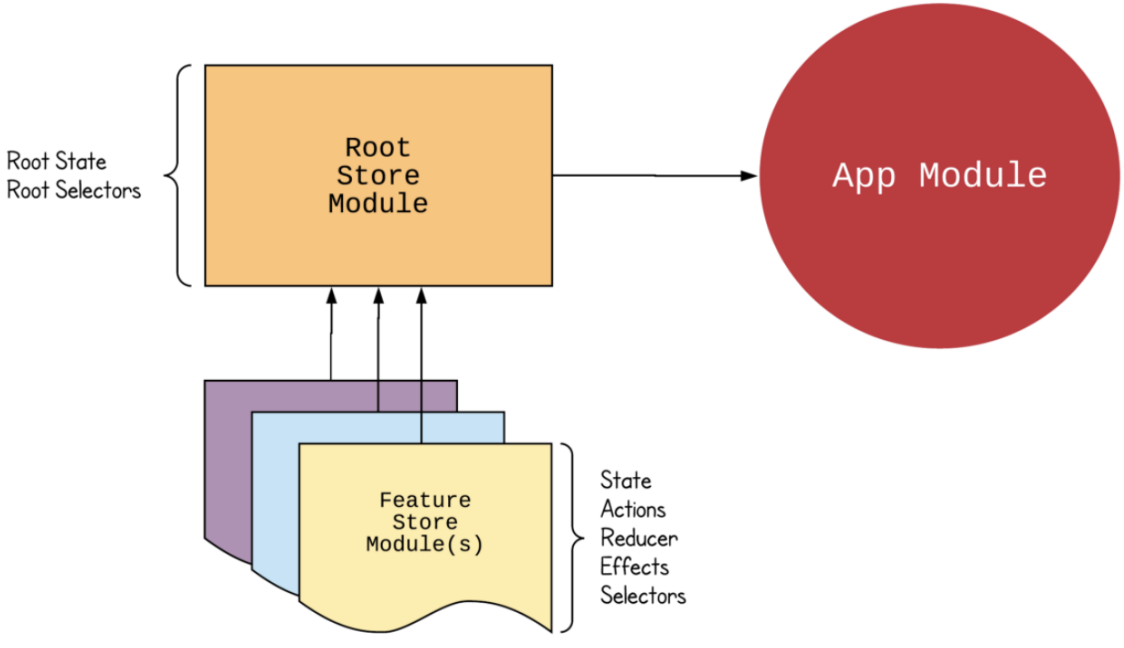
- State is **shared** between components
- State that needs to be persisted and **hydrated** across page reloads
  - State that needs to be **available** when re-entering routes
  - State that needs to be **retrieved** with a side-effect, separating the concerns
- State that is **impacted** by other components or services

Register to use **NgRx** ?

```
const metaReducers  = !environment.production ? [storeFreeze] : [];
const StoreDevTools = !environment.production ? StoreDevtoolsModule.instrument() : [];

@NgModule({
  imports: [
    BrowserModule,
    StoreModule.forRoot(
      {},                                 // Register root state and reducers
      { metaReducers }                    // Regsiter StoreFreeze; runs AFTER reducers
    ),
    EffectsModule.forRoot([ ]),           // Register root effects
    StoreRouterConnectingModule,          // Connect Router to NgRx
    StoreDevTools                         // Connect to Redux Dev Tools
  ]
})
export class AppModule {}
```

- App module is is for '**root**' features, setup, and tooling
  - Use **RootStoreModule** for separation of concerns
- Feature modules are used for **feature** state.

NgRx Best Practices



- Feature modules can be lazy loaded
- Feature modules can be independently registered

Why State Immutability ?

- Predictable *State Change*

  State Changes **ONLY** occur in the reducer(s)!

- Consistent *1-way Data Flows*

  Data flows into views. Views only render... never modify data!

- *Immutable Data for* Change Detection Performance

  Optimize which views update rendering, when.

StoreFreeze: Enforcing state Immutability

- Recursively freezes the **current state**, the dispatched **action payload** if provided and the **new state**.
- When mutation occurs, an **exception will be thrown**.

```
const metaReducers  = !environment.production ? [storeFreeze] : [];
const StoreDevTools = !environment.production ? StoreDevtoolsModule.instrument() : [];

@NgModule({
  imports: [
    StoreModule.forRoot(
      {},                   // no reducers here...
      {
        metaReducers        // no initialState here...
      }
    ),
    StoreDevTools
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

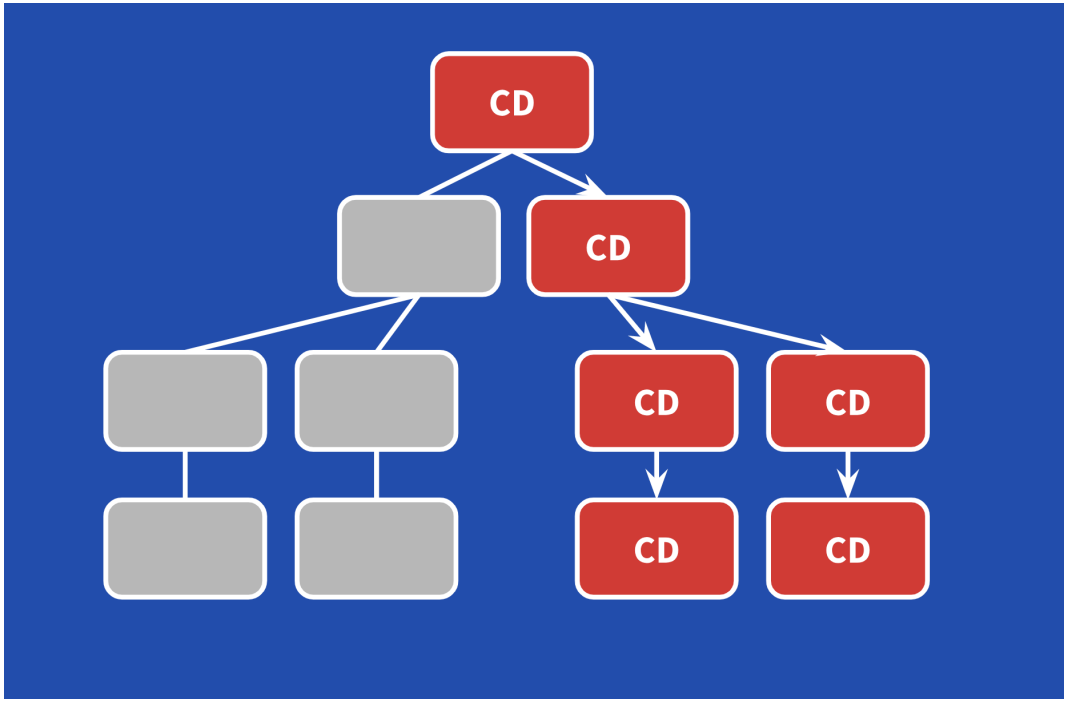
apps/customer-portal/src/app/app.module.ts

Speaker notes

Notice that StoreFreeze is only used in dev mode; to ensure that the state remains immutable.
After the reducers process actions and possibly create new state, StoreFreeze deep-freezes all properties in the state
! This is why you state should contain raw objects only !

Angular: Injection & Change Detection

Each view component has its own Change Detection



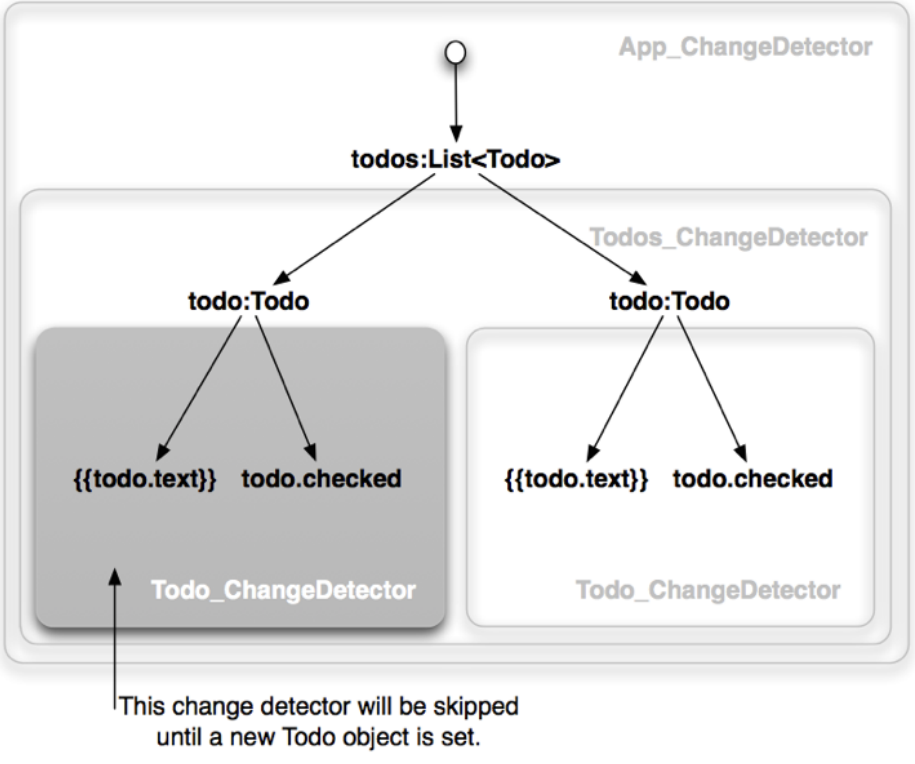

```
changeDetection: ChangeDetectionStrategy.OnPush
```

http://bit.ly/**2PRTL3i**

Speaker notes

Here is an illustration that shows Angular Top-Down Change Detection
The NgRx 1-way flow ensures predictable UI updates and changes

NgRx: 1-way Data Flows & Change Detection



- In Angular, changes flow from parents to children via **@Input**
- Components may also access data from services, like **Store**, as long as the uni-directional flow is preserved
- Having a single source of shared state helps enforce that changes flow the right direction

NgRx: Performance w/ Change Detection

- Change detection may trigger view rendering
- Use OnPush ChangeDetection Strategy

```
@Component({
  selector: 'app-ticket-details',
  templateUrl: './ticket-details.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class TicketDetailsComponent {

}
```

For this component, change detection is triggered when:

- Inputs change
- Async pipe detects new value(s) emitted

Speaker notes

Angular uses Zones to determine when views should be updated (re-rendered) after async changes occur.

Deep Dive: Zones & Change Detection

- Understand Zones
- Zones in Angular
- @angular/zones (rxjs)

In fact, the code that tells Angular to perform change detection whenever the VM turn is done, is as simple as this:

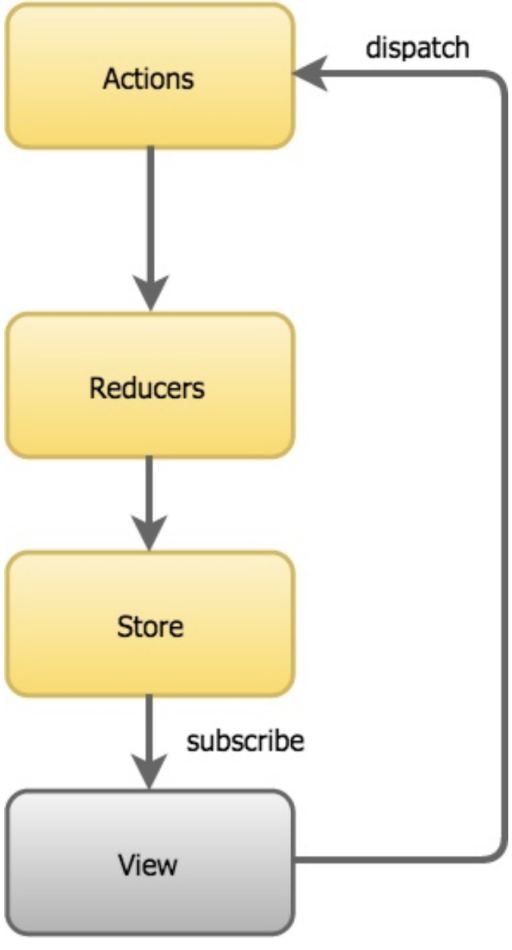
```
ObservableWrapper.subscribe(this.zone.onTurnDone, () => {
  this.zone.run(() => {
    this.tick();
  });
});

tick() {
  // perform change detection
  this.changeDetectorRefs.forEach((detector) => {
    detector.detectChanges();
  });
}
```

Whenever Angular's zone emits an `onTurnDone` event, it runs a task that performs change detection for the entire application.

Remember **5 Principles of NgRx**

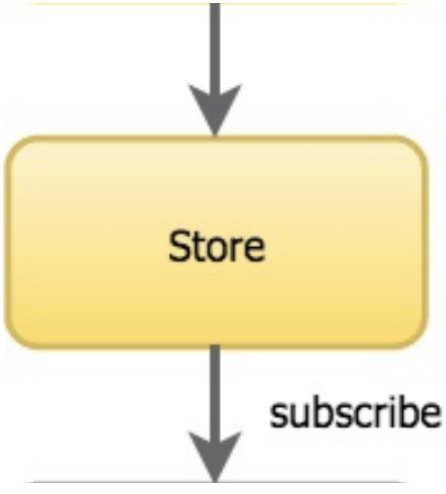
- **Single source of truth** - State of whole app is stored in a single state tree within a **single store**
- **State is read-only** - Only way to change state is to emit an **action**
- **Changes made only with reducers** - State tree is transformed or composed by pure **reducers**
- **Data is pushed** **-** Only way to read the state data is to use a selector and build a observable query
- **1-way Data Flows** **-** Read-only data flows into views only via observable queries.

NgRx Core Concepts



- **Store** - State container
- **State** - State is described as plain object without setters
- **Actions** - A plain object that describes events in our application
- **Reducers** - A pure function that takes state and action as arguments and returns next state
- **Selectors** - queries to extract data
- **Effects** - Asynchronous Activity

NgRx Store



- Store is a **singleton service**
- Manages **State** and state changes
- Store is **Observable**
  - **store.dispatch(action)**
  - Calls **reducers** when actions received to update state
  - **Pushes** updated state to all subscribers
  - **store.select(sliceName)**
- Does not mutate state

Defining the NgRx Application State

The application state, here, represents all data/state managed by NgRx

```
export interface ApplicationState {
  tickets   : TicketsState;
  comments  : TicketCommentsState;
  eventLogs : EventLogsState;
}

// Keys are used to match reducers to initialState

const allReducers = {
  tickets   : ticketsReducer,
  comments  : commentsReducer,
  eventLogs : eventLogsReducer
};
const initialState = {
  tickets   : ticketsInitialState,
  comments  : commentsInitialState,
  eventLogs : eventLogsInitialState
};

// Prepare Store instantiation with state registrations...

StoreModule.forRoot( allReducers, {
  initialState
});
```

Speaker notes

Here is an example application with three (3) slices of NgRx state in managed in the Store

Injecting the Store Instance

```
export interface ApplicationState {
  tickets    : TicketsState;
  comments   : TicketCommentsState;
  eventLogs  : EventLogsState;
}
```

```
@Component()
export class ContactsListComponent implements OnInit {

  constructor(
    private ticketssService: TicketsService,
    private store: Store<ApplicationState>
  ) { }

  ngOnInit() {
    ...
  }
}
```

Dependency Injection will inject the Store instance

The store has access to all state 'slices'/features

Speaker notes

We are now organizing our NgRx state by feature!
But when we inject the store, the store manages all slices of state...
So we have to define all slices in the 'ApplicationState' interface right?
Then what is the point of the Feature state modules?

NgRx State

SHARI can be used to determine what state belongs in a Store container.

- **S**hared
- **H**ydrated
- **A**vailable
- **R**etrieved
- **I**mpacted

Keep your store **DRY**.

No implicit or explicit state duplication.

NgRx State

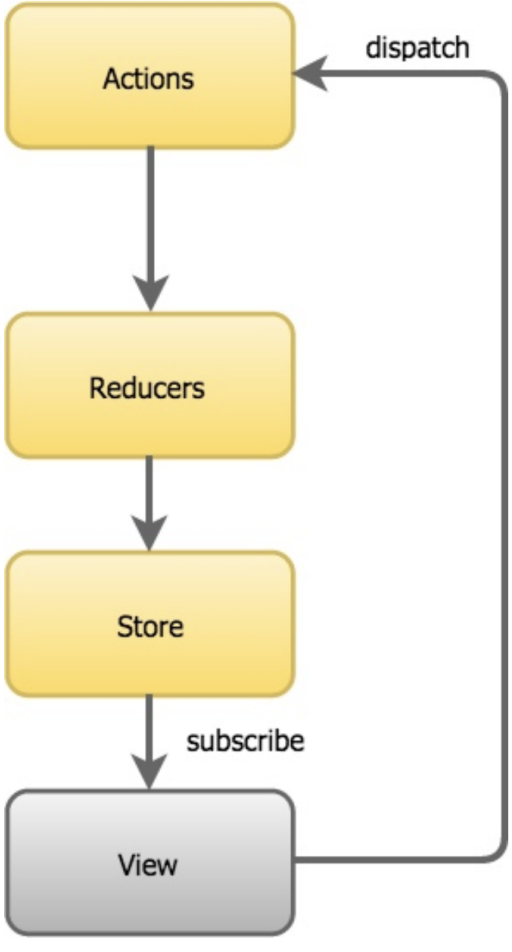
State is a plain JavaScript object without setter methods... think raw data!

```
export interface TicketsState {
  list      : Ticket[];
  selectedId: number;
  loading   : boolean;
  error     : any;
}


export const initialState: TicketsState = {
  list       : [],
  selectedId : -1,
  loading    : false,
  error      : ''
};
```

Speaker notes

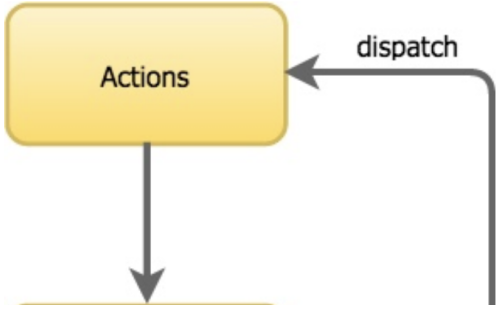
State contains nothing about how the data is loaded, persisted, or used.
It is raw data that will be used by consumers.

NgRx Core Concepts



- **Store** - State container
- **State** - State is described as plain object without setters
- **Actions** - A plain object that describes events in our application
- **Reducers** - A pure function that takes state and action as arguments and returns next state
- **Selectors** - queries to extract data
- **Effects** - Asynchronous Activity

NgRx Actions



A simple object that describes events in our application. This could be **user events**, **backend events**, **device events,** or **response events** for async activity.

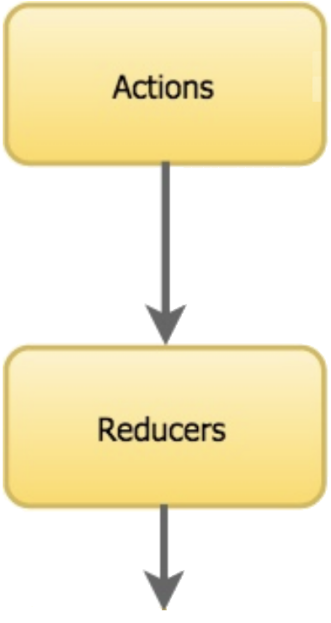
```
export class LoadTicket implements Action {
  type = TicketActionTypes.LOAD_TICKET;
  constructor(readonly payload: number) {}
}

const ticketId = 1;
this.store.dispatch(new LoadTicket(ticketId));
```

Speaker notes

Actions represent an intent to change state: Action == `This is what I'd like to do...`
Actions typically have two (2) properties:

```
type    : string,
payload ?: (any)
```

NgRx **Reducer**



Reducer is a pure **function** that

processes the **previous** state + an **action**

and returns **next** (new) state

- Typically there are **many reducers** in an application
- Each reducer is called for **each/every** dispatched action
- Reducers return **same state** if action is ignored

NgRx Reducer

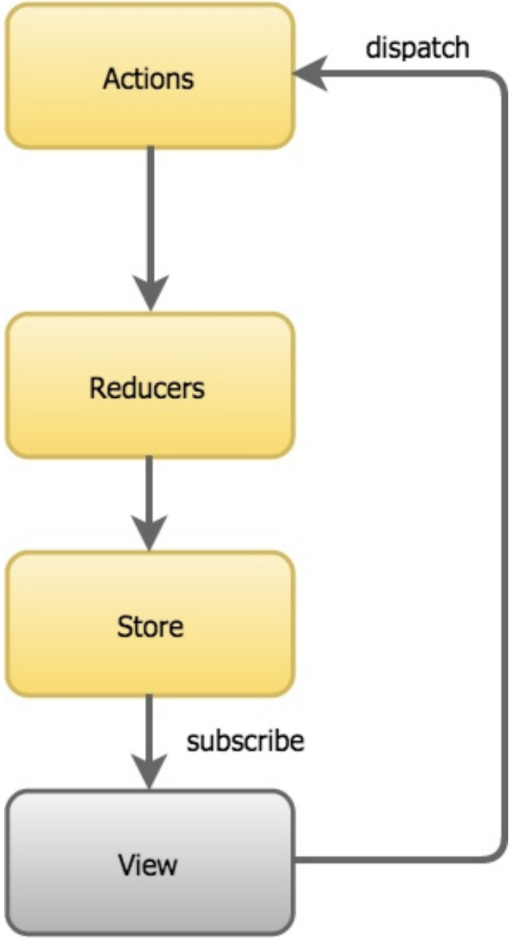
```
export function ticketsReducer(state: TicketsState, action: TicketAction): TicketsState {
  switch (action.type) {

    case TicketActionTypes.LOAD_TICKET_DONE: {
      const list = [...state.list];
      const ticket = (action as LoadTicketDone).ticket;

      if (!list.some(it => it.id === ticket.id)) {
        list.push(ticket);
      }

      return {
        ...state,
        list
      };
    }

  }
  return state;
}
```

Speaker notes

A reducer will process only desired actions.

Changes to state must involve the entire state object
If no changes, then return the original state.

NgRx Core Concepts



- **Store** - State container
- **State** - State is described as plain object without setters
- **Actions** - A plain object that describes events in our application
- **Reducers** - A pure function that takes state and action as arguments and returns next state

So far, we have **not used RxJS**...

NgRx: Selectors + RxJS

- Way to extract pieces of the state
- Use the **store.select()** method to build query “observable”

```
export class TicketListComponents {

  tickets$: Observable<Ticket[]>;

  constructor(store: Store<any>) {

    const selectorFn = (state):Ticket[] => state.tickets.list;
    const query$     = this.store.pipe( select(selectorFn) );

    this.tickets$    = query$;
  }

}
```

Speaker notes

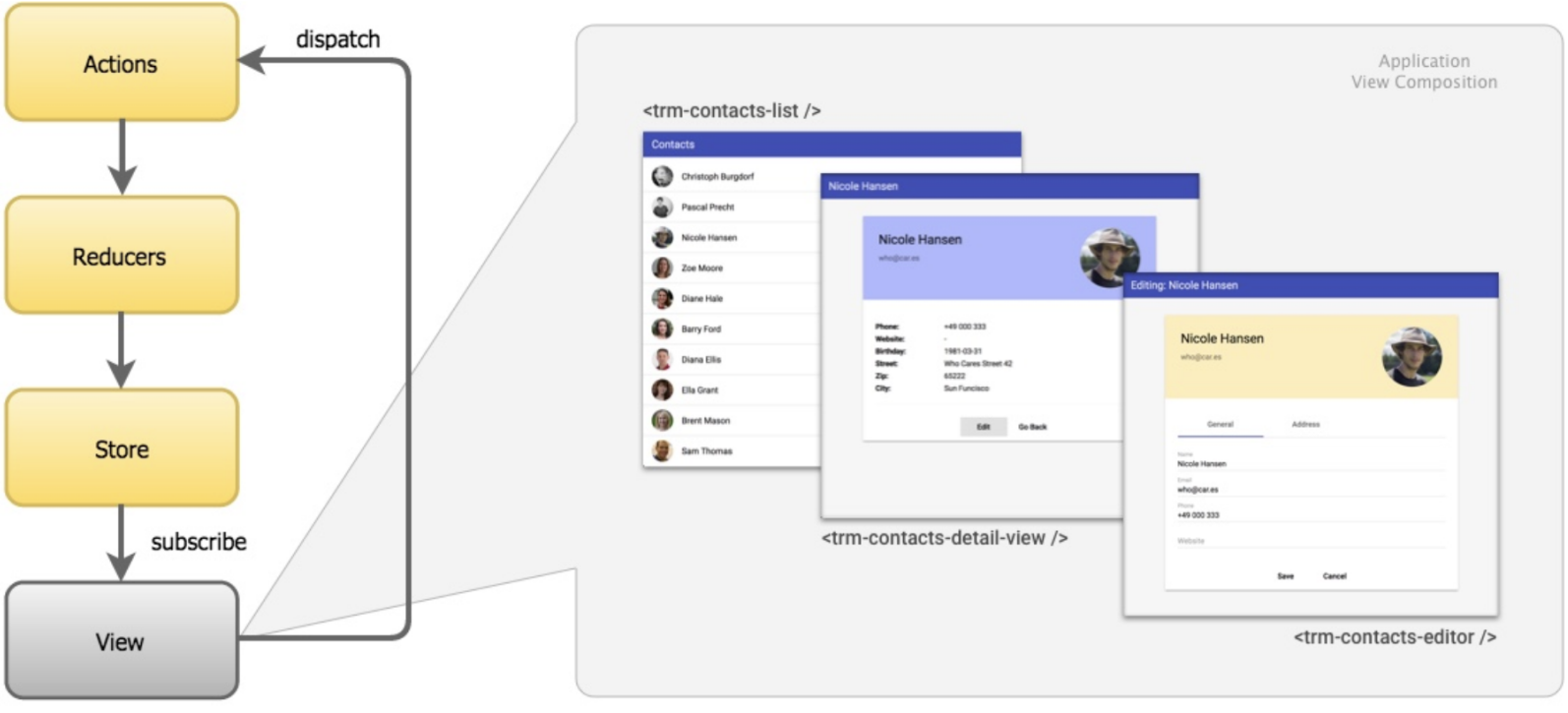
We use DI to inject a Store instance.
Here the Store means the store manages an unknown set of features; one of which is 'tickets';

NgRx

- **Store Model** (interfaces and initial object)
- **Actions** (command to request state change)
- **Reducers** (pure functions used to change state)
- **Selectors** (querying slices of the state)

We also need to register all of these ^ with NgRx as a **Feature**

Contacts + NgRx

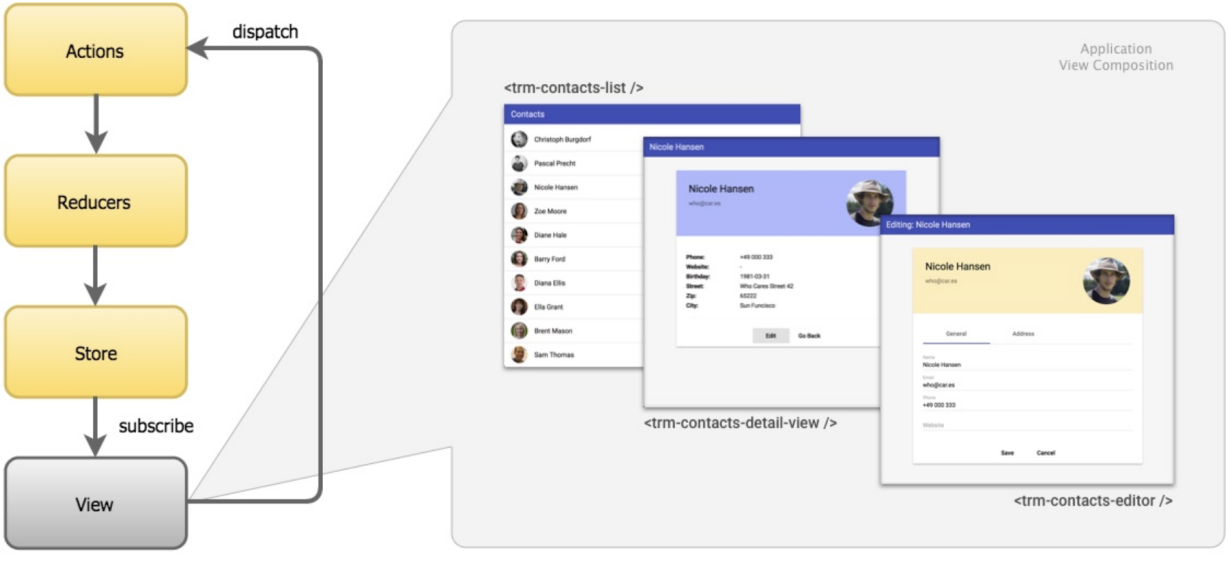


Speaker notes

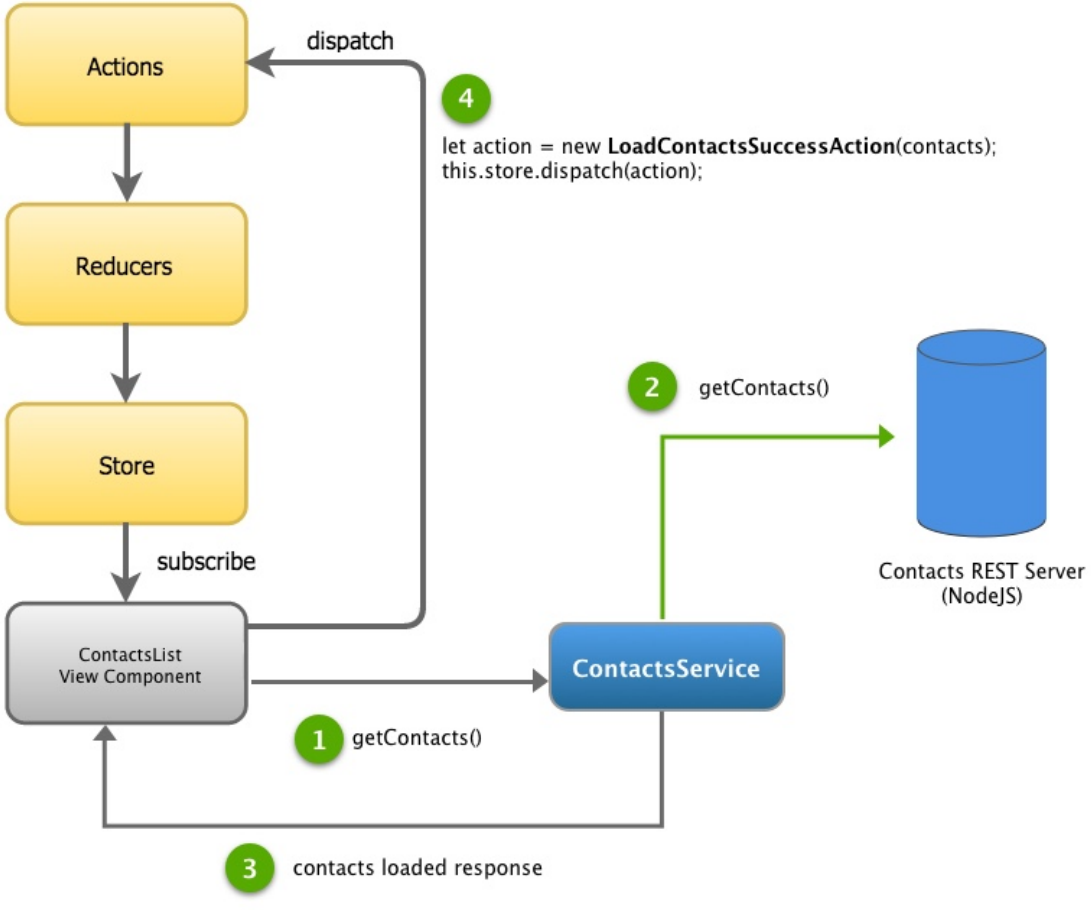
The Tickets NgRx feature is already present.
Let's explore the existing code and the NgRx artifacts...

Scenario

Configuring **ContactsListComponent** to dispatch an action after the contacts list has been loaded to hydrate store with contact data

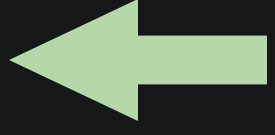


Contacts + NgRx



NgRx: ContactsState Models

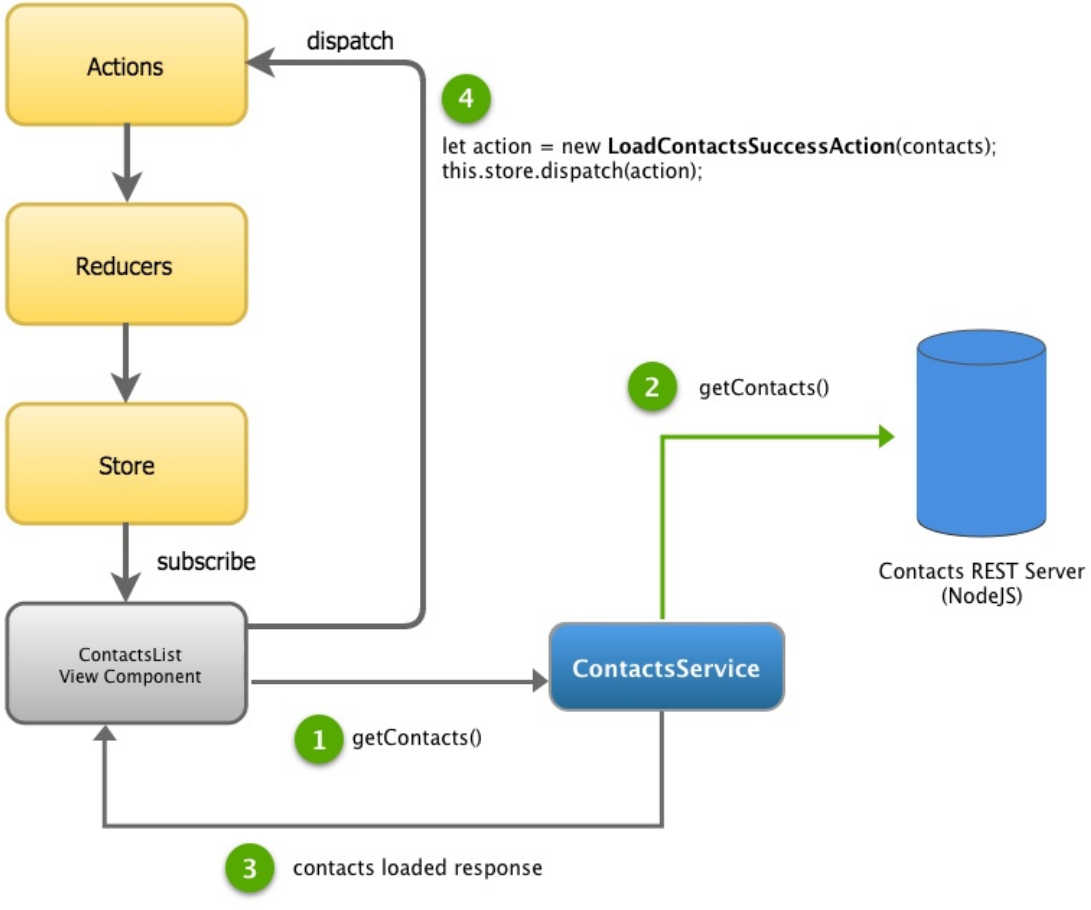
```
export interface ContactsState {
  list: Array<Contact>;
  selectedId: string;
  isLoading: boolean;
}

export const INITIAL_STATE: ContactsState = {
  list: [],
  selectedId: '',
  isLoading: false
};
```



NgRx: ContactsState Actions

```
import { Action } from '@ngrx/store';
import { Contact } from '../../models/contact';


export enum ContactsActionTypes {
  LOAD_CONTACTS_SUCCESS = '[Contacts] Contacts Loaded Success'
}

export class LoadContactsSuccessAction implements Action {
  readonly type = ContactsActionTypes.LOAD_CONTACTS_SUCCESS;
  constructor(public list: Array<Contact>) {}
}

export type ContactsActions = LoadContactsSuccessAction;
```

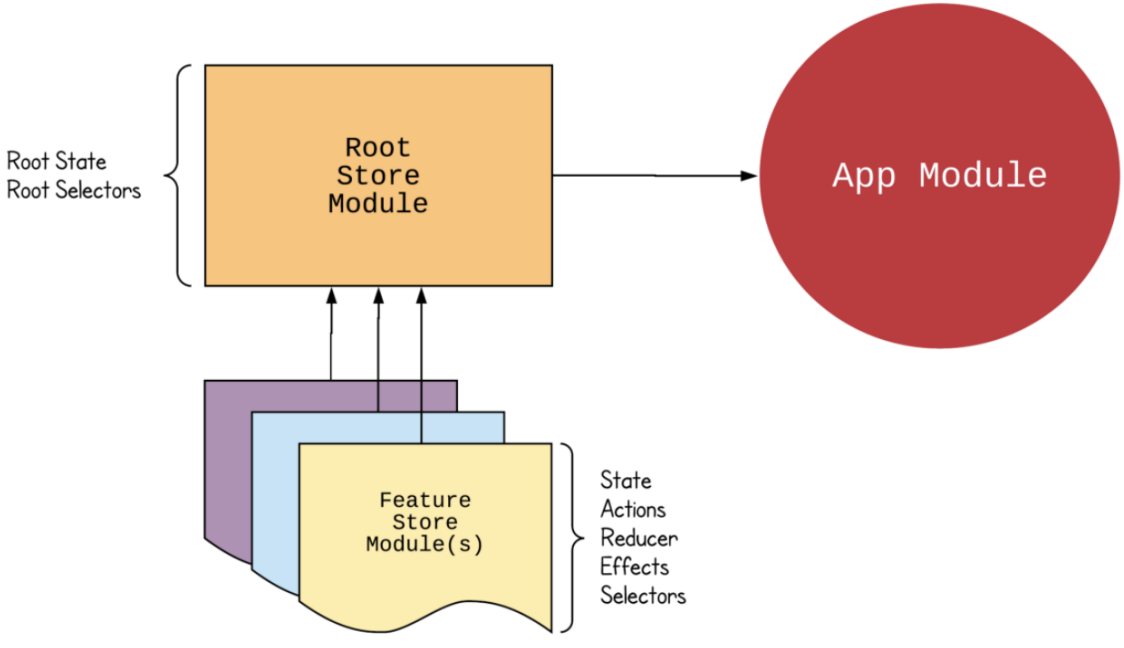
Contacts + NgRx



NgRx: ContactsState Reducer

```
export function contactsReducer(state: ContactsState = INITIAL_STATE,
                                action: ContactsActions) {
  switch (action.type) {

    case ContactsActionTypes.LOAD_CONTACTS_SUCCESS:
      const { list } = action;
      return { ...state, list };
  }

  return state
}
```

ContactsState Ngrx Feature



- Feature modules can be lazy loaded
- Feature modules can be independently registered

Speaker notes

We want to organize and register our TicketsState as an NgRx Feature.

Register your ContactsState Feature

```
import {FEATURE_KEY, contactsReducer, INITIAL_STATE} from './state';

@NgModule({
  imports: [
    CommonModule,
    StoreModule.forRoot({}, {metaReducers}),
    EffectsModule.forRoot([]),
    StoreModule.forFeature(
      FEATURE_KEY,
      contactsReducer,
      {
        initialState: INITIAL_STATE
      }
    ),
    StoreDevTools
  ],
  providers: [ ]
})
export class ContactsNgRxModule {
}
```

Data Access with **Selectors**

After injecting the store, how do you get your
ngrx-locked data into the view?

Data Access with **Selectors**

After injecting the store, how do you get your *ngrx-locked* data into the view?

- You cannot **PULL** it out of the state.
- You must configure a way to have it **PUSHED** to you

Data Access with **Selectors**

After injecting the store, how do you get your *ngrx-locked* data into the view?

- You cannot **PULL** it out of the state.
- You must configure a way to have it **PUSHED** to you

- Use **Selectors** to build your Observables

Data Access with **Selectors**

After injecting the store, how do you get your *ngrx-locked* data into the view?

- You cannot **PULL** it out of the state.
- You must configure a way to have it **PUSHED** to you

- Use **Selectors** to build your Observables
- Once the data is pushed, the view 'reacts' to that data change & update the views

Using your Contacts NgRx

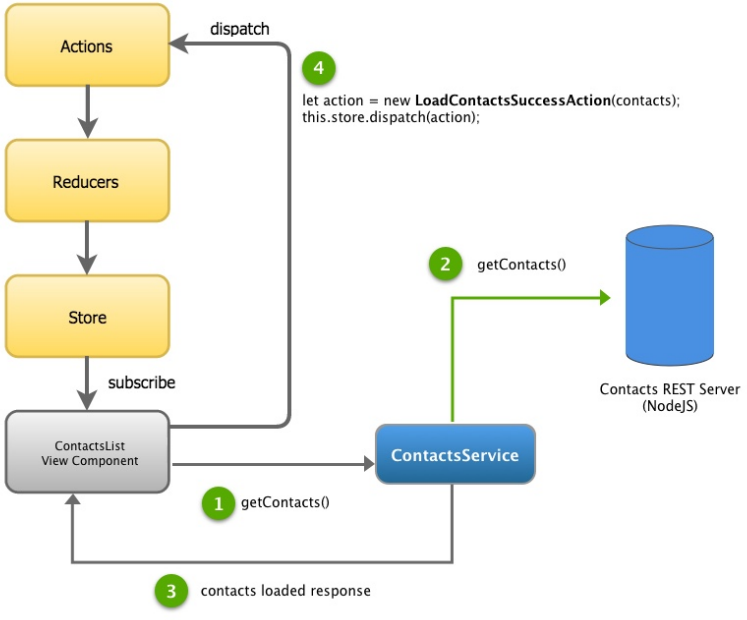
2) build query

1) injecting the Store

(3) dispatch action

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {
  contacts$: Observable<Contact[]> = this.store.pipe(
    select(state => state.contacts.list)
  );

  constructor(
      private store: Store<ApplicationState>,
      private contactsService: ContactsService) { }

  ngOnInit() {
    this.contactsService.getContacts().subscribe(contacts => {
      this.store.dispatch(
        new LoadContactsSuccessAction(contacts)
      );
    });
  }

  trackByContactId(index, contact) {
    return contact.id;
  }
}
```

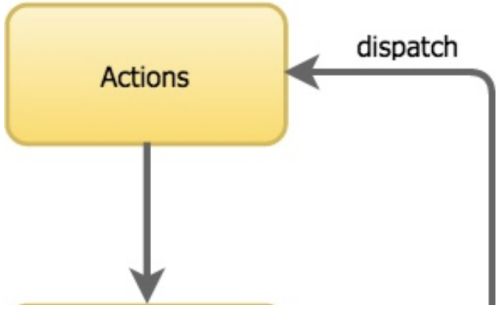
NgRx Lab 1: Actions, Reducer, & Selector



- Load Contacts (from View)
- Save Contacts list into the NgRx store
- Show Contacts (in View) using NgRx Selector

Lab Exercise

Adding more NgRx actions



New actions can be easily added using the same patterns

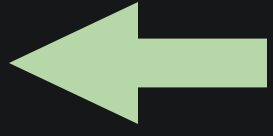
```
export enum ContactsActionTypes {
  ...
  UPDATE_CONTACT = '[Contacts] Update contact';
  SELECT_CONTACT = '[Contacts] Select contact';
}
```

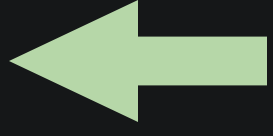
Adding more NgRx actions

```
export class LoadContactsSuccess implements Action {
  readonly type = ContactsActionTypes.LOAD_CONTACTS_SUCCESS;
  constructor(public list: Contact[]) {}
}

export class SelectContact implements Action {
  readonly type = ContactsActionTypes.SELECT_CONTACT;
  constructor(public selectedId: string) {}
}

export class UpdateContact implements Action {
  readonly type = ContactsActionTypes.UPDATE_CONTACT;
  constructor(public contact: Contact) {}
}

export type ContactsActions = LoadContactsSuccess | SelectContact | UpdateContact;
```

Update your NgRx reducer

```
export function contactsReducer(state: ContactsState = INITIAL_STATE, action: ContactsActions) {
  switch (action.type) {

    case ContactsActionTypes.LOAD_CONTACTS_SUCCESS:
      const { list } = action;
      return { ...state, loaded: true, list };

    case ContactsActionTypes.SELECT_CONTACT:
      const { selectedContactId } = action;
      return { ...state, selectedContactId };

    case ContactsActionTypes.UPDATE_CONTACT:
      const { contact } = action;
      const hasSameId = contact => contact.id === action.payload.id;
      const list = state.list.map(it => {
        return hasSameId(it) ? {...it, ...contact} : it;
      });
      return { ...state, list };
  }

  return state;
}
```



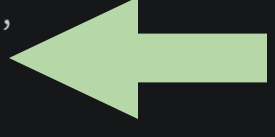


Warning! Cloning the Contact

For the ContactsEditorComponent it's important to **clone** the object returned by the NgRx selector.

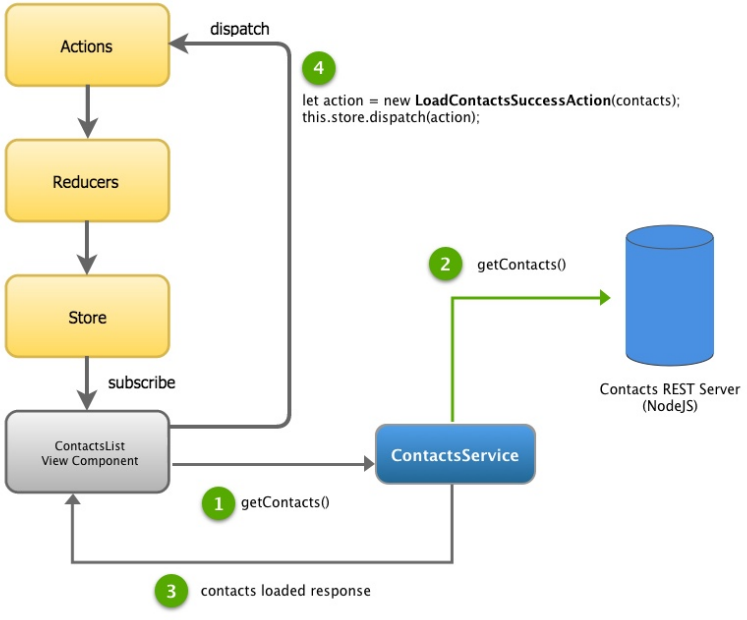
contacts-editor.component.ts

```
ngOnInit() {
  const contactId = this.route.snapshot.paramMap.get('id');
  const getSelectedContact = (state) => {
    return state.contacts.list.find(contact => {
      contact.id == contactId);
    };
  };
  const cloneContact = (contact) => !!contact ? { ...contact } : null;

  this.store.dispatch(new SelectContactAction(contactId));

  this.contact$ = this.store.pipe(
    select(getSelectedContact),
    map(cloneContact)
  );
}
```



This is required to ensure immutability. We must ensure that the FormControls and 2-way data bindings will not mutate our **Store** directly.

NgRx Lab 2: Selecting and Editing Contacts



- Select and Update Contact Actions
- Query selected contact in Editor and Details views.

Lab Exercise

Deep Linking with Router Guards

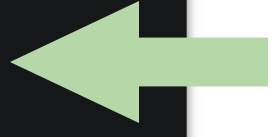
We use direct URL to display user details

In Angular we can apply **Guards** or **Resolvers** to the route configuration to handle these scenarios.

Guards or Resolvers **defer** the navigation while async activity works....

Router Guard

```
@Injectable()
export class ContactExistsGuard implements CanActivate {

  canActivate(route: ActivatedRouteSnapshot) {
    let contactId = route.paramMap.get('id');
    this.store.dispatch(new SelectContactAction(+contactId));

    return this.store.pipe(
        select(state => state.contacts.loaded)
        take(1),
        mergeMap((loaded: boolean) => {
          return loaded ? of(true) : this.loadContact(contactId);
        })
      );
  }

  // ********************************************************

  private loadContact(contactId) {
    const addContactToList = (contact: Contact) {
      this.store.dispatch(new AddContactAction(contact))
    };
    const contact$ = this.contactsService.getContact(contactId);

    return contact$.pipe(
      tap(addContactToList),
      map(contact => !!contact)
    );
  }
}
```



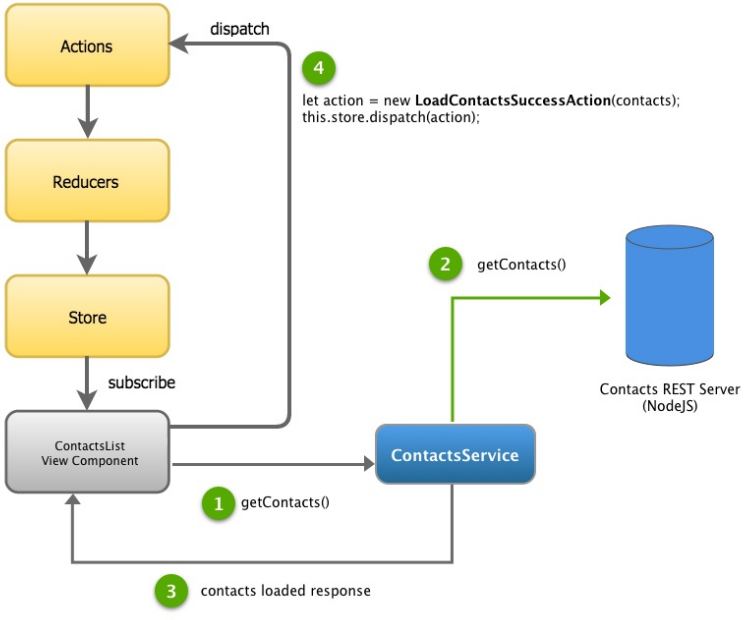
Adding more NgRx actions

```
export enum ContactsActionTypes {
  ADD_CONTACT = '[Contacts] Add Contact';
  ...
}

export class AddContactAction implements Action {
  readonly type = ContactsActionTypes.ADD_CONTACT;
  constructor(public contact: Contact) { }
}

export type ContactsActions = AddContactAction | ...
```

Update your NgRx reducer

```
case ContactsActionTypes.ADD_CONTACT:
  const { contact } = action;
  const inStore = state.list.find(it => {
    return it.id === action.payload.id;
  });

  return {
    ...state,
    list: !inStore ? [...state.list, contact] : state.list
  };
```

NgRx Lab 3: Deep Linking + Route Guards



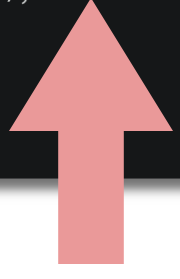
- Implement ContactExistsRouteGuard
- AddContact action/reducer

Lab Exercise

1-way Data Flows

- The Store **pushes state changes** to all subscribers (via queries).
- The ContactsListComponent uses **contacts$ | async** which triggers change detection
- Change detection triggers new updates to ContactsList UI

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {
  contacts$: Observable<Contact[]>;

  constructor(private store: Store<ApplicationState>) {
    const selectorFn = (state):Contact[] => state.contacts.list;
    const query$ = store.pipe( select(selectorFn) );

    this.contacts$ = query$
  }
}
```



Data Access with **Selectors**

```
// Define our selector
const getAllTickets = (state) => state.allTickets;

// Define our push-query
const allTickets$ = this.store.pipe( select(getAllTickets) );
```

Benefits of **Selectors**

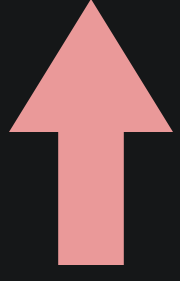
- Selectors are **functions** that extract data from the state argument
- Selectors can compute derived data, allowing NgRx to store the **minimal possible state**.
- Provide **Query API** for views

- Selectors are **composable** functions. They can be used as input to other selectors.
- Selectors are **efficient** when **memoized** *.
- **Simplify** reducers

* selector is not recomputed unless one of its arguments change

Problem: Simple Selectors

```
import {createFeatureSelector, createSelector} from '@ngrx/store';
import { ContactsState, FEATURE_CONTACTS } from './contacts.reducer';

const type PartialAppState = {          // Only defined to access contact data
  [FEATURE_CONTACTS]: ContactsState
}

export namespace ContactsQuery {
  export const getAllContacts  = (state:PartialAppState) => state[FEATURE_CONTACTS].list;
  export const getSelectedId   = (state:PartialAppState) => state[FEATURE_CONTACTS].selectedId;
  export const getIsLoading    = (state:PartialAppState) => state[FEATURE_CONTACTS].isLoading;

  export const getSelectedContact = (state:PartialAppState) => {
    const selectedId  = getSelectedId(state);
    const allContacts = getAllContacts(state);
    const matchingId  = (contact) => contact.id === selectedId;
    const contact     = allContacts.find(matchingId);

    return contact ? ({...contact} as Contact) : null
  };
}
```



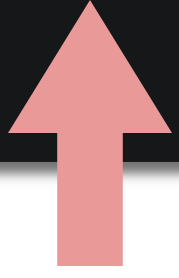
Here '**state**' is the all the NgRx state which includes the '**contacts**' state...

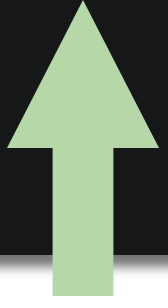
Speaker notes

Unfortunately, with these 'raw' selectors we are missing all the advantages of NgRx composed, memoized selectors.

Solution 1: Feature Selectors

```
import {createFeatureSelector, createSelector} from '@ngrx/store';
import { ContactsState, FEATURE_CONTACTS } from './contacts.reducer';

export namespace ContactsQuery {
  // Lookup the 'Contacts' feature state managed by NgRx
  export const getContactsState = createFeatureSelector<ContactsState>(FEATURE_CONTACTS);

  export const getContacts = (fullState) => {
    state: ContactsState = getContactsState(fullState);
    return state.list;
  }
}
```





Speaker notes

This extracts the state associated with the contactsReducer using the 'contacts' key.

Selector **Composition**

```
// Selector Chaining to extract state

const v1 = s1(state).s2(state).s3(state)

// Selector Composition

const v2 = s3(s2(s1(state)));

// ---------------------------------------------------------------
// With Ngrx. These prepare functions that LATER will
// be called with 'state'
// ---------------------------------------------------------------

const selector2 = createSelector(s1, s2);
const selector3 = createSelector(selector2, s3);

const v3 = selector3(state);

// or equivalently

const selector = createSelector(createSelector(s1, s2), s3);

const v4 = selector(state);
```

Solution 2: Composing Selectors

We compose selectors using **createSelector**().

```
import { createFeatureSelector, createSelector } from '@ngrx/store';
import { ContactsState, FEATURE_CONTACTS } from './contacts.reducer';

export namespace ContactsQuery {

  export const getContactsState = createFeatureSelector<ContactsState>(FEATURE_CONTACTS);

  export const getLoaded     = createSelector(getContactsState, (state:ContactsState) => state.loaded);
  export const getContacts   = createSelector(getContactsState, (state:ContactsState) => state.list);
  export const getSelectedId = createSelector(getContactsState, (state:ContactsState) => state.selectedId);

}
```

Now our selectors only 'know' about **ContactsState**...

What about **Memoization** ?

Memoization is an **optimization technique** used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Solution 3: Memoization & Computed State

```
import { createFeatureSelector, createSelector } from '@ngrx/store';
import { ContactsState, FEATURE_CONTACTS } from './contacts.reducer';

export namespace ContactsQuery {

  export const getContactsState = createFeatureSelector<ContactsState>(FEATURE_CONTACTS);

  export const getLoaded     = createSelector(getContactsState, (state:ContactsState) => state.loaded);
  export const getContacts   = createSelector(getContactsState, (state:ContactsState) => state.list);
  export const getSelectedId = createSelector(getContactsState, (state:ContactsState) => state.selectedId);

  export const getSelectedContact = createSelector(getContacts, getSelectedId,
    (list, id) => list.reduce(
      (result, it) => result || ((it.id === id) ? it : null),
      null
    )
  );

}
```

Speaker notes

So with
(a) composition used to leverage 'separation of concerns' and even compute state like 'selectedContact'
(b) memoization to cache results with specific input values
Our NgRx selectors become VERY powerful, reusable, and performant

Views: Using your Contacts NgRx

2) build query

1) injecting the Store instance

(3) dispatch action

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {
  contacts$: Observable<Array<Contact>> = this.store.pipe(
    select(ContactsQuery.getContacts)
  );

  constructor(
    private contactsService: ContactsService,
    private store: Store<ApplicationState>) { }

  ngOnInit() {
    this.contactsService.getContacts().subscribe(contacts => {
      this.store.dispatch(new LoadContactsSuccessAction(contacts));
    });
  }

}
```

Speaker notes

Later, we will fix the architecture so the views never load the data nor know about ContactsService

Entity-like Pattern

Current solution is **inefficient** for large collections.
Current solution(s) require lots of **CRUD logic** in the Reducers

With Entity patterns:

Entity-like Pattern

Current solution is **inefficient** for large collections.
Current solution(s) require lots of **CRUD logic** in the Reducers

With Entity patterns:

- Used to manage **collections** of similar objects

Entity-like Pattern

Current solution is **inefficient** for large collections.
Current solution(s) require lots of **CRUD logic** in the Reducers

With Entity patterns:

- Used to manage **collections** of similar objects
- Helps to reduce the complexity of our **reducers**

Entity-like Pattern

Current solution is **inefficient** for large collections.
Current solution(s) require lots of **CRUD logic** in the Reducers

With Entity patterns:

- Used to manage **collections** of similar objects
- Helps to reduce the complexity of our **reducers**
- Enables fast **CRUD** operations

Entity-like Pattern

Current solution is **inefficient** for large collections.
Current solution(s) require lots of **CRUD logic** in the Reducers

With Entity patterns:

- Used to manage **collections** of similar objects
- Helps to reduce the complexity of our **reducers**
- Enables fast **CRUD** operations
- You can use @ngrx/entity

Entity-like Pattern

Before using **@ngrx/entity**,

let's implement some of that functionality **manually**!

Speaker notes

This manual approach will not look easier.

But it builds on concepts... then when we use @ngrx/entity,
you will see the reduction in code and complexity.

NgRx TicketsState

tickets.interfaces.ts

```
export interface Ticket {
  id       : number;
  message  : string;
  ...
}


export interface TicketsState {
  list       : Ticket[];

  selectedId : number;
  loading    : boolean;
  error      : any;
}
```

our collection...

Entity-like Pattern

```
{ [id: number]: Ticket }
```

```ts
export interface Ticket {
  id      : number;
  message : string;
  ...
}


const entities = {

  '1' : { id : 1, message: 'Ticket #1' },
  '3' : { id : 3, message: 'Ticket #3' },
  '9' : { id : 9, message: 'Ticket #9' },

}
```

```
const ticket = entities['9'];
```

Speaker notes

With entity-like patterns, it is fast to get to one, easy to “change” the one
Normalized ticket data: only one reference location for ticket instances
Use ids array for lookup associations

Entity-like Pattern in a **Reducer**

```ts
export interface ContactsDictionary {
  [key: number]: Contact
}

export interface ContactsState {
  entities: ContactsDictionary;
  ids: number[];

  selectedContactId: number | null;
  loaded: boolean;
}
```

contacts.reducer.ts

contacts.reducer.ts

```ts
export function contactsReducer(state: ContactsState = INITIAL_STATE,
                                action: ContactsActions) {
  switch (action.type) {
    case ContactsActionTypes.LOAD_CONTACTS_SUCCESS:
      const { list } = action;
      const contactEntities = list.reduce(
          (entities, contact) => {
            return {...entities, [contact.id]: contact};
          },
          { ...state.entities }
      );

      return {
        ...state,
        loaded: true,
        entities: contactEntities
      };
  }

  return state;
}
```

Wait...

```
export interface ContactsDictionary {
  [key: number]: Contact
}

export interface ContactsState {
  entities: ContactsDictionary;
  ids: number[];

  selectedContactId: number | null;
  loaded: boolean;
}
```

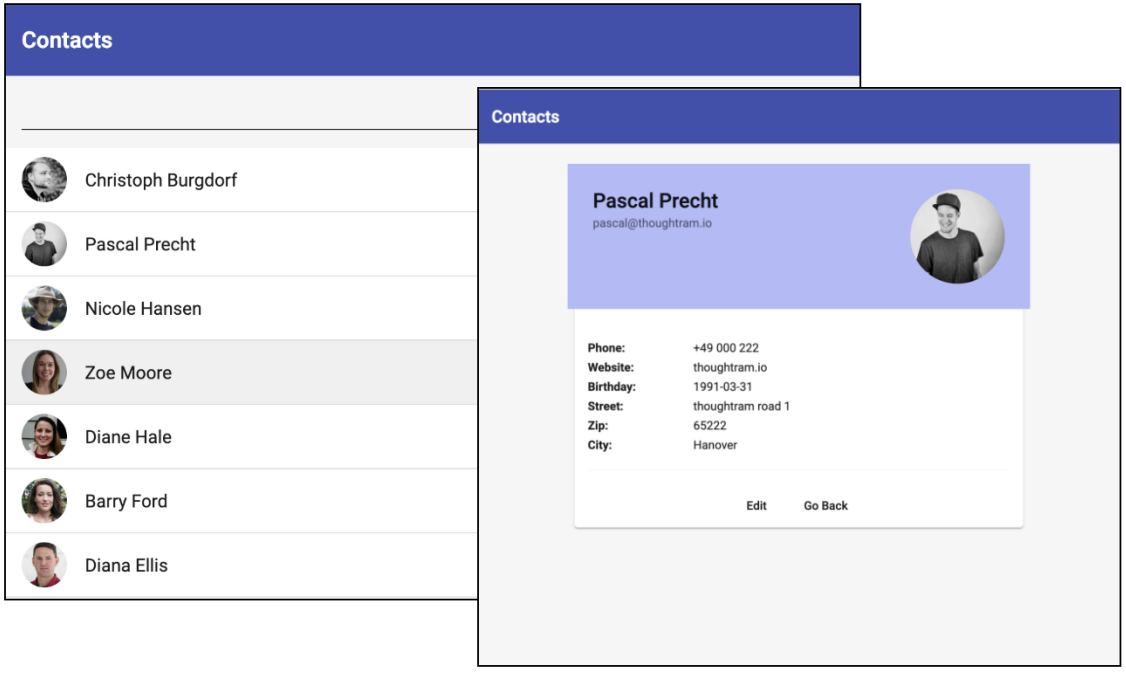
Using **entities** breaks everything... 😱

Update Selectors for **Entity-like Pattern**

contacts.selectors.ts

```
export namespace ContactsQuery {

  const getContactsState = createFeatureSelector<ContactsState>(FEATURE_CONTACTS);
  const getSelectedId = createSelector(getContactsState, (state:ContactsState) => state.selectedContactId);

  export const getEntities    = createSelector(getContactsState, (state:ContactsState) => state.entities);
  export const getLoaded      = createSelector(getContactsState, (state:ContactsState) => state.loaded);
  export const getContacts    = createSelector(getContactsEntities, entities => {
    return Object.keys(entities).map(id => entities[id])
  });
  export const getSelectedContact = createSelector(getContactsEntities, getSelectedContactId,
    (contactEntities, id) => contactEntities[id]
  );

}
```

NgRx Lab 4: Selectors + Entity-Like Pattern



- Cache the Contacts list as in a Dictionary, save ids as sorted list
- Update Selectors to use Entity properties

Lab Exercise

@Effects for Async Operations

- Allow to **centralize** all asynchronous work or side effects
- Extracting **side effects** from our components make them easier to test and reason about
- We can think of them as **background processes** within an application
- When each async activity completes a **synchronous action** is dispatched
- Available in the **@ngrx/effects** module.

Speaker notes

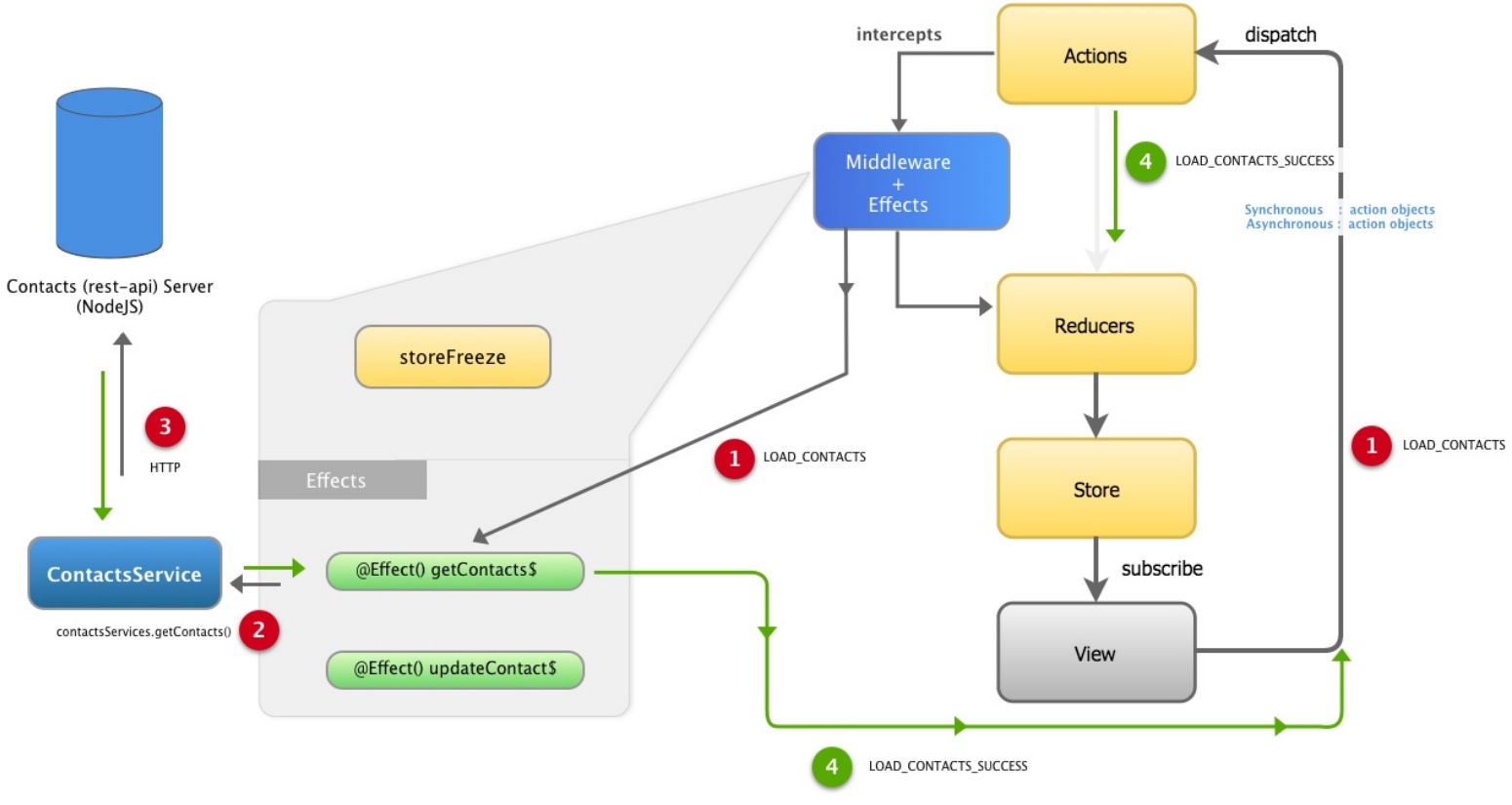
Effects can also be used for synchronous action deciders, action splitters, and more...
See the Victor Savkin's blog article on this.

Effects are Sources of Actions

- **@Effect()** decorator declares which observables on a service are action sources.
- Library **merges** your actions streams
- The **NgRx Store** **subscribes** to these observables; to listen for more actions that may be emitted.

- All Reducers are called for all actions...
- Effects can listen on stream of all actions...

Effects for Contacts



(1)	LOAD_CONTACTS	UPDATE_CONTACT	<async action>
(4)	LOAD_CONTACTS_SUCCESS LOAD_CONTACTS_ERROR	UPDATE_CONTACT_SUCCESS UPDATE_CONTACT_ERROR	<sync action>

Actions for Contacts

```
export namespace ContactsActionTypes {
  export const LOAD_CONTACTS          = '[Contacts] Load All';
  export const LOAD_CONTACTS_SUCCESS  = '[Contacts] Load All Done';
  export const LOAD_CONTACTS_ERROR    = '[Contacts] Load All Failed';

  export const UPDATE_CONTACT         = '[Contacts] Update Contact';
  export const UPDATE_CONTACT_SUCCESS = '[Contacts] Update Done';
  export const UPDATE_CONTACT_ERROR   = '[Contacts] Update Failed';
  ...
}

export class LoadContactsAction implements Action {
  readonly type = ContactsActionTypes.LOAD_CONTACTS;
}

export class UpdateContactAction implements Action {
  readonly type = ContactsActionTypes.UPDATE_CONTACT;
  constructor(public contact: Contact) { }
}
```

Speaker notes

Each action to fire async activity will have two associated synchronous actions: done or fail

Effects Classes

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { Actions, Effect, ofType } from '@ngrx/effects';

import { ContactsService } from '../../contacts.service';

@Injectable()
export class ContactsEffects {


  constructor(
    private actions$       : Actions,
    private router          : Router,
    private contactsService: ContactsService) {
  }

}
```

- Class gives us ability to inject services that we need to do the effect
- Class allows us to organize related workflow logic together
- **actions$** is a stream of all actions coming through **NgRx**

Creating **Effects**

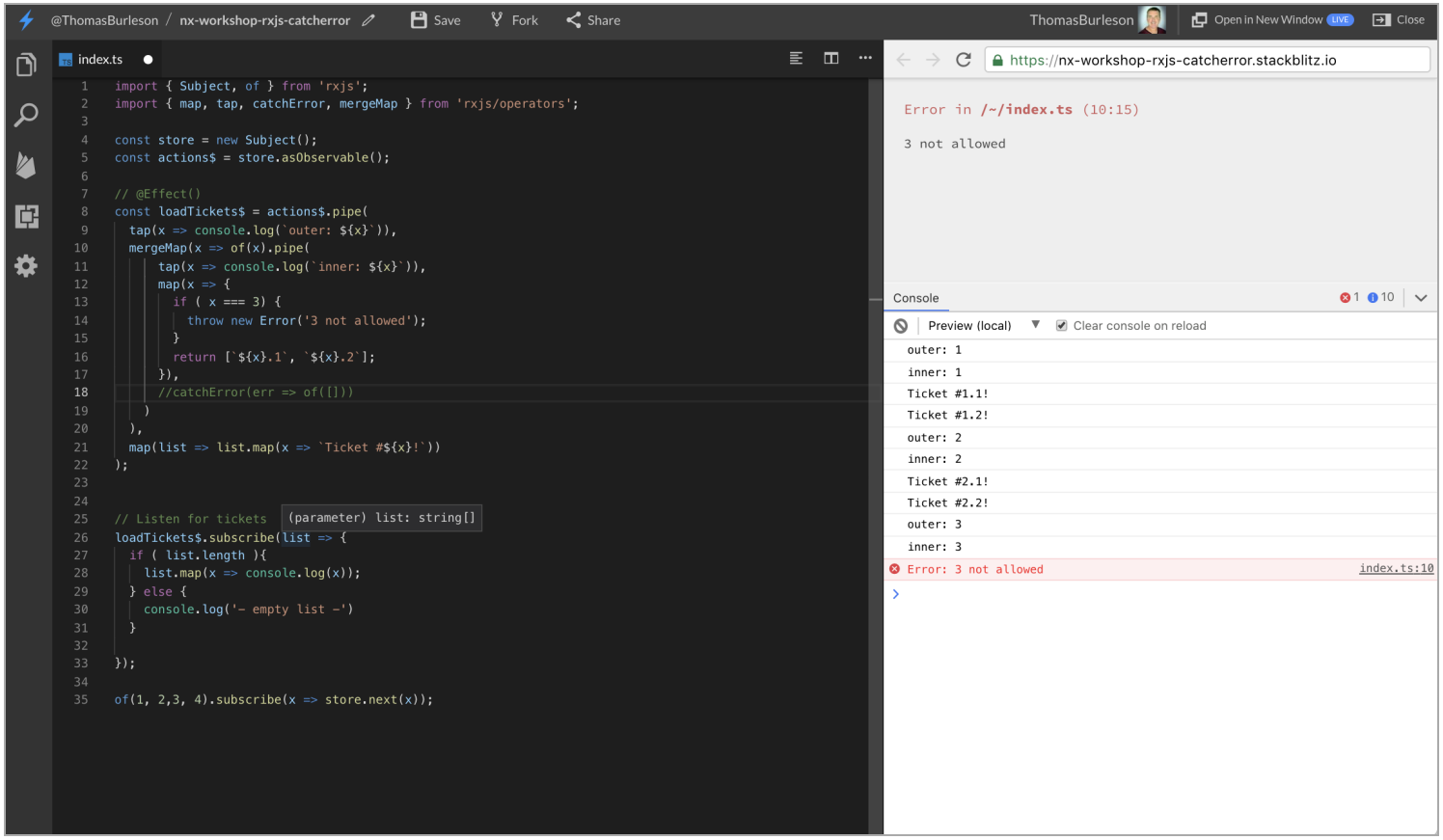
filter for specific action

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { Actions, Effect, ofType } from '@ngrx/effects';

import { map, exhaustMap, catchError } from 'rxjs/operators';

import { ContactsService } from '../../contacts.service';

@Injectable()
export class ContactsEffects {

  @Effect()
  allContacts$ = this.actions$.pipe(
    ofType(ContactsActionTypes.LOAD_CONTACTS),
    exhaustMap(_ => {
      return this.contactsService.getContacts().pipe(
        map(contacts => new LoadContactsSuccessAction(contacts)),
        catchError(error => new LoadContactsErrorAction(error))
      );
    })
  );


  constructor(
    private actions$       : Actions,
    private router          : Router,
    private contactsService: ContactsService) {
  }

}
```

Speaker notes

With { dispatch: true } this @Effect tells NgRx store that actions will be emitted in the future...
so the Store must internally subscribe to 'allContacts$' observable.

RxJS **Error Handling**



Errors stop subscriptions... for @Effects() this **permanently kills** the Effect

Using **catchError()** on inner traps error **without killing** outer subscription

http://bit.ly/**2B24Whr**

With **NgRx Effects**

Asynchronous code/logic can be removed from the View components...

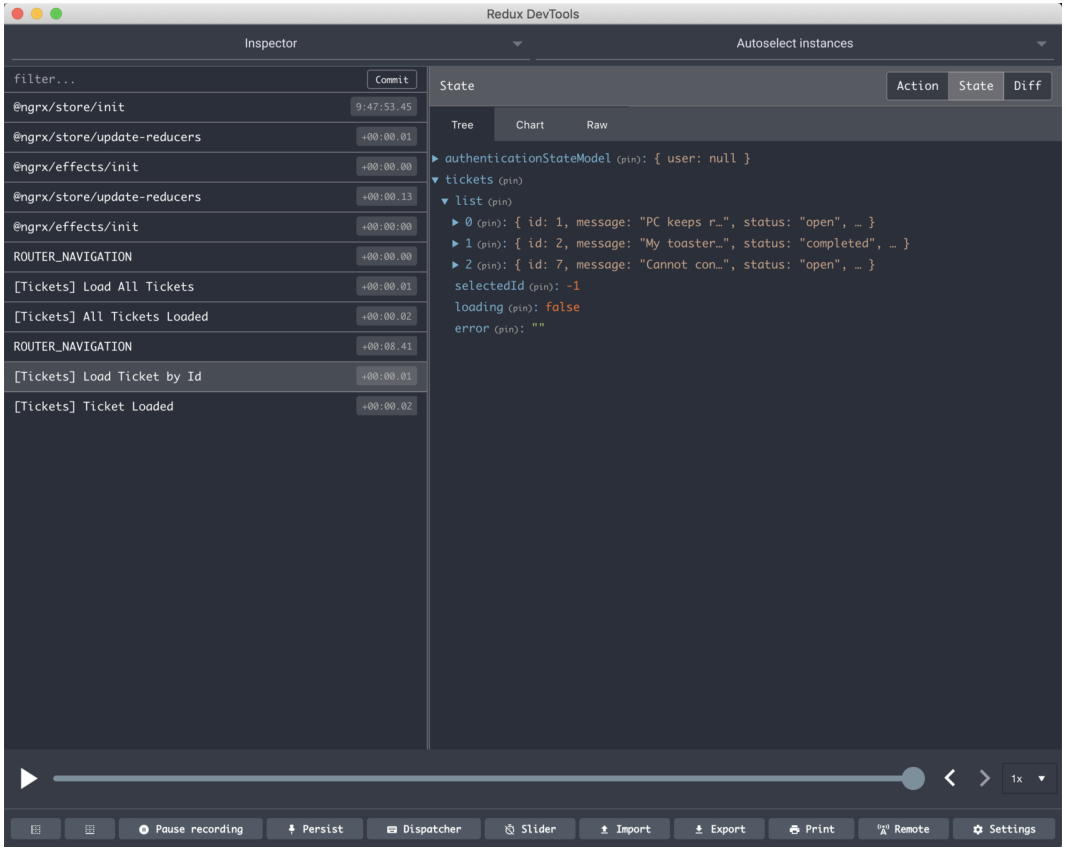
Since this logic is now in the **ContactsEffects** layer.

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {
  contacts$: Observable<Array<Contact>> = this.store.pipe(
      select(ContactsQuery.getContacts)
  );

  constructor(
      private store: Store<ApplicationState>,
      private contactsService: ContactsService) { }

  ngOnInit() {
    this.contactsService.getContacts().subscribe(contacts => {
      this.store.dispatch(new LoadContactsSuccessAction(contacts));
    });
  }

  trackByContactId(index, contact) {
    return contact.id;
  }
}
```

Register your **Effect** class with **NgRx**

```
@NgModule({
  imports: [
    CommonModule,

    StoreModule.forRoot({}, {metaReducers}),
    EffectsModule.forRoot([]),
    StoreDevTools

    StoreModule.forFeature(FEATURE_CONTACTS, contactsReducer, {
      initialState: INITIAL_STATE
    }),
    EffectsModule.forFeature([ContactsEffects]),

  ]
})
export class ContactsNgRxModule {
}
```

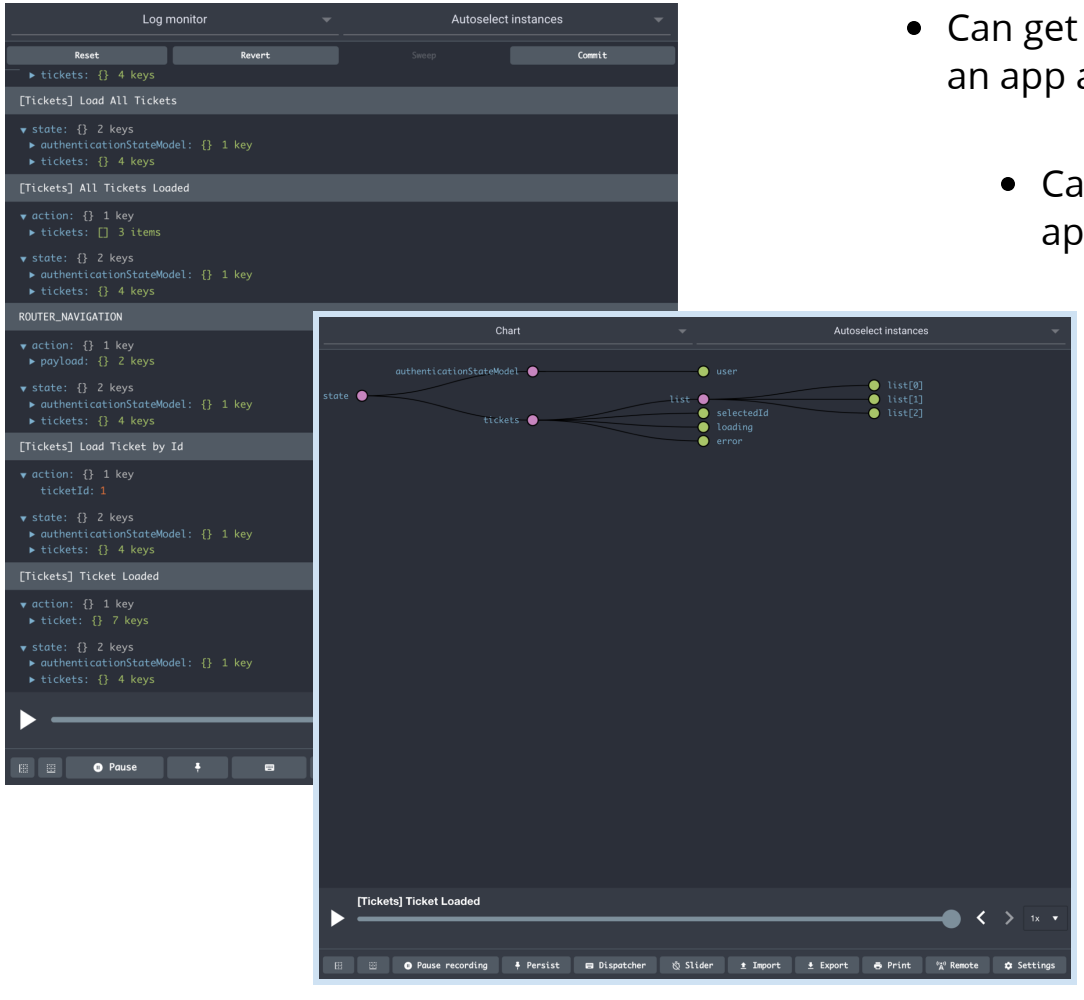
Redux DevTools: Debugging State Changes



Redux DevTools for Chrome

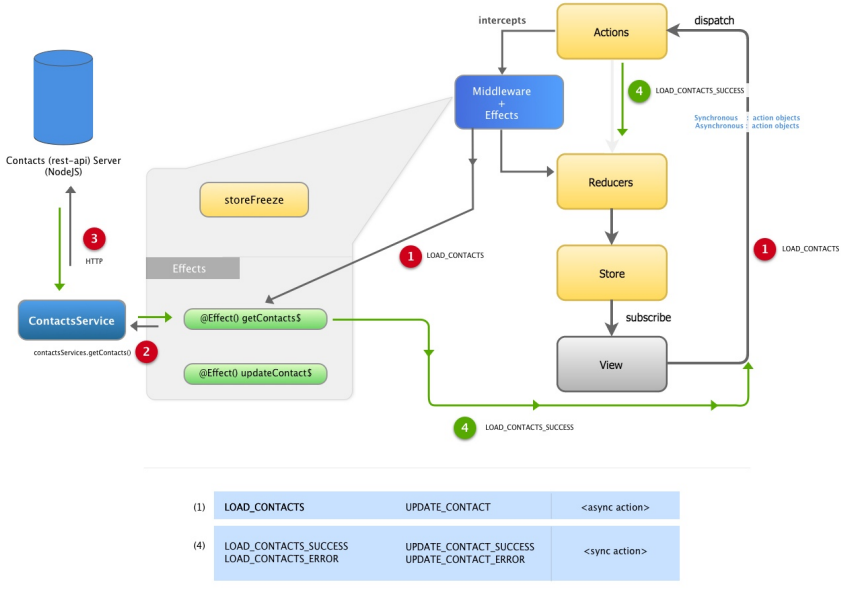
- Actions come through in a predictable order
- Payload on the action has the details of what was requested
- Can replay actions to see what happened over time

Redux DevTools: Debugging State Changes



- Can get the **current state** of an app at any time
    - Can use a **snapshot** to put the app back in that state
- Can use this to capture state during **errors or app failure**

NgRx Lab 5: Use @Effects & Redux DevTools



(1)	LOAD_CONTACTS	UPDATE_CONTACT	<async action>
(4)	LOAD_CONTACTS_SUCCESS LOAD_CONTACTS_ERROR	UPDATE_CONTACT_SUCCESS UPDATE_CONTACT_ERROR	<sync action>

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { Actions, Effect, ofType } from '@ngrx/effects';

import { map, exhaustMap, catchError } from 'rxjs/operators';

import { ContactsService } from '../../contacts.service';

@Injectable()
export class ContactsEffects {

  @Effect()
  allContacts$ = this.actions$.pipe(
    ofType(ContactsActionTypes.LOAD_CONTACTS),
    exhaustMap(_ => {
      return this.contactsService.getContacts().pipe(
        map(contacts => new LoadContactsSuccessAction(contacts)),
        catchError(error => new LoadContactsErrorAction(error))
      );
    })
  );


  constructor(
    private actions$       : Actions,
    private router         : Router,
    private contactsService: ContactsService) {
  }

}
```

- Use **Effects** for loading contacts(s)
- Open Redux DevTools to see actions and state changes

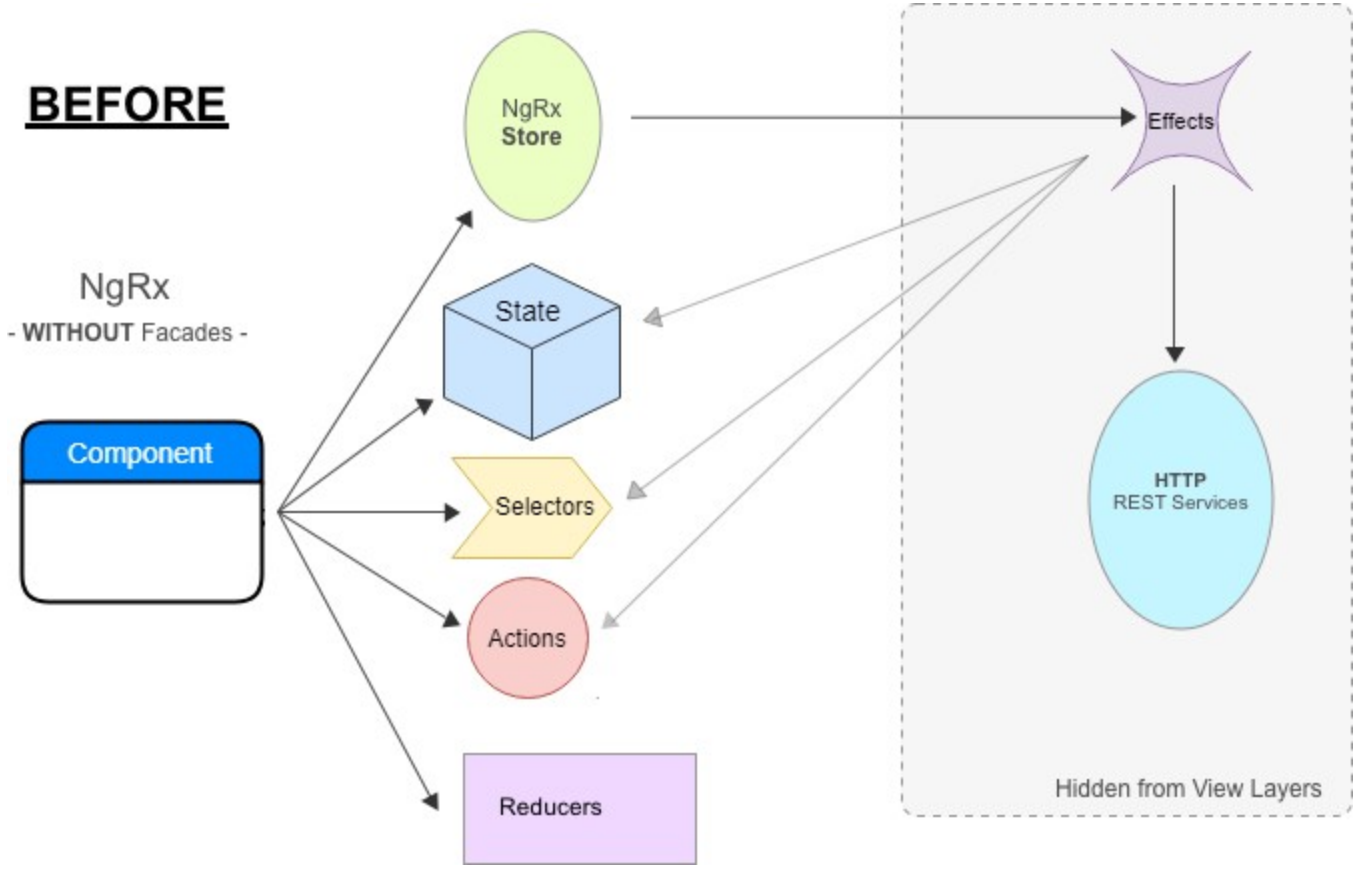
Lab Exercise

Using **NgRx Facades**

The Facade pattern makes a software library easier to use, understand, and test. It hides complexity and publishes a well-defined public API.

- Provides a simplified public API
- Black boxes the internals of the implementation

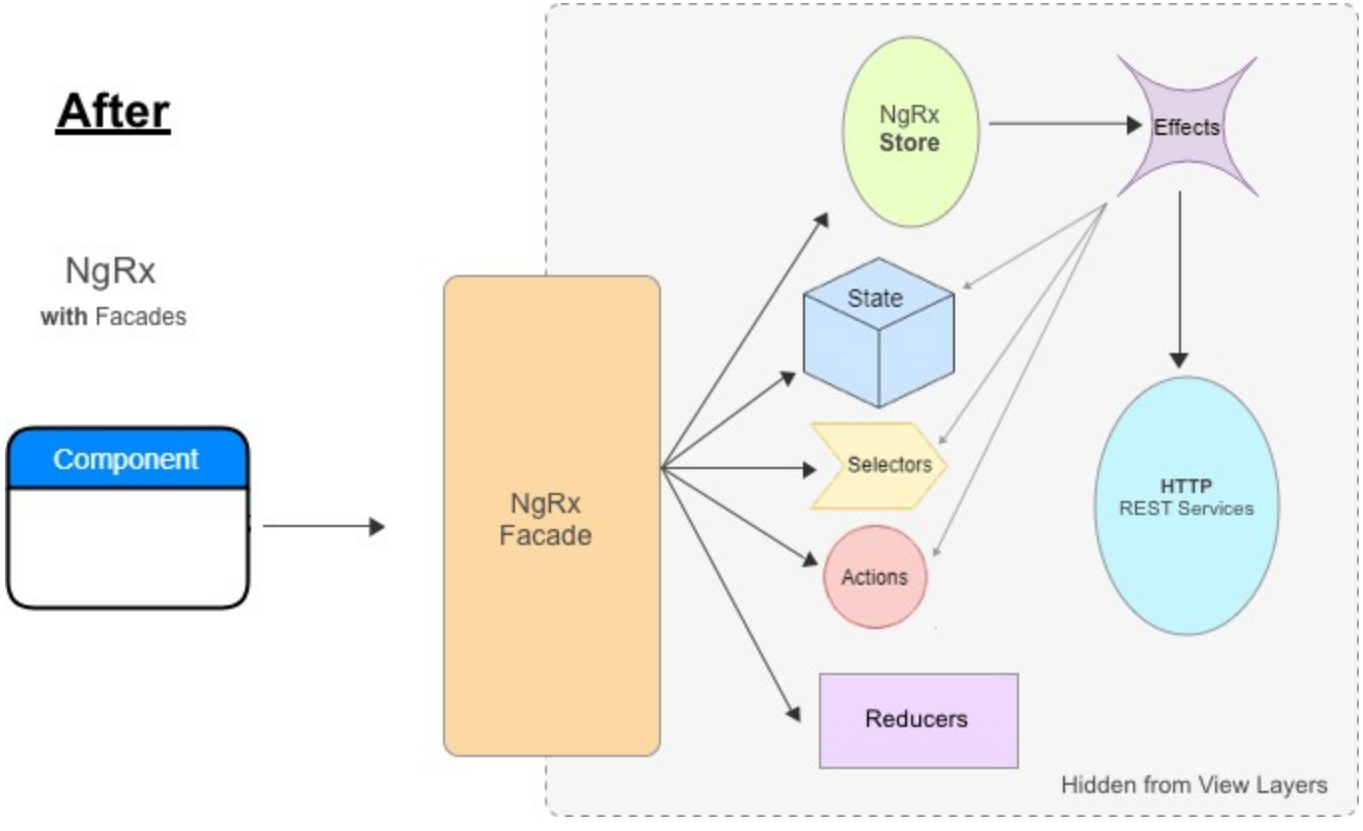
Problem: ...without NgRx Facades



Speaker notes

Our business logic is distributed
Views import and use Selectors
Views use and dispatch Actions
Views inject Store
Effect-like processes should work in the background

Solution: NgRx Facades



NgRx Facade Goals

The Facade pattern makes a software library easier to use, understand, and test. It hides complexity and publishes a well-defined public API.

- Hides Complexity
- Centralizes business logic

- Views only use Observables and Async Pipes
- Encourages views to be stateless, presentation-only

NgRx Facade

tickets.facade.ts

```
@Injectable()
export class ContactsFacade {

  loaded$          = this.store.pipe(select(ContactsQuery.getLoaded));
  contacts$        = this.store.pipe(select(ContactsQuery.getContacts));
  selectedContact$ = this.store.pipe(select(ContactsQuery.getSelectedContact));

  constructor(private store: Store<Schema>,
              private contactsService: ContactsService) {

    // Load all data as soon as the facade loads
    this.store.dispatch(new LoadContactsAction());
  }

   getContactById(contactId: number): Observable<Contact> {
     ...
   }

}
```

Facades handle all the details of **building observable queries**.

Facades can even **merge & blend** observables...

Speaker notes

Don't forget to add the Facade as a provider in contacts-ngrx.module.ts
And don't forget to export the ContactsFacade in the library's public barrel.

NgRx Facade

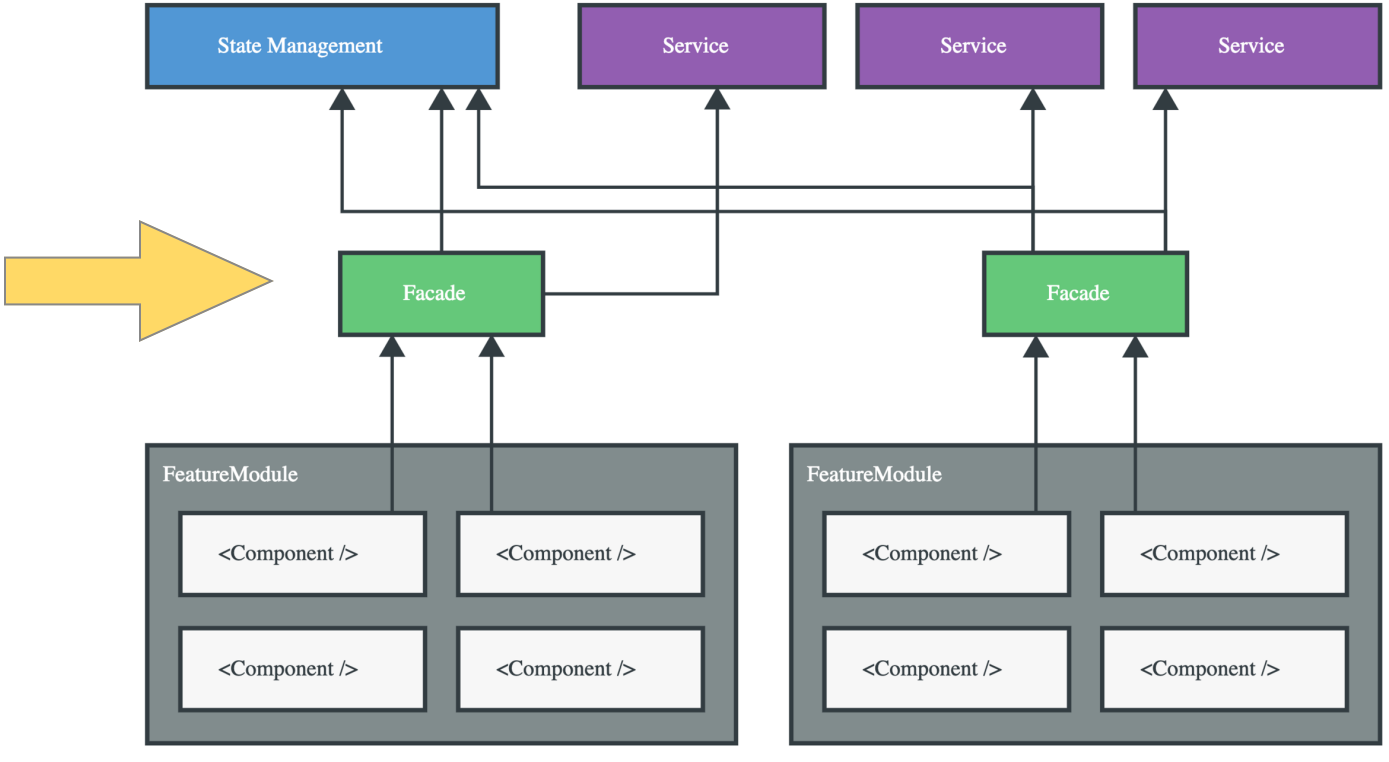
Before

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {
  contacts$: Observable<Array<Contact>> = this.store.pipe(
      select(ContactsQuery.getContacts)
  );

  constructor(
      private store: Store<ApplicationState>,
      private contactsService: ContactsService) { }

  ngOnInit() {
    this.contactsService.getContacts().subscribe(contacts => {
      this.store.dispatch(new LoadContactsSuccessAction(contacts));
    });
  }
}
```

After

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent {
  contacts$ = this.facade.contacts$

  constructor( private facade: ContactsFacade ) { }
}
```

NgRx Facade



Contacts-View FeatureModule

Comments-View FeatureModule

Speaker notes

Facades are the layer BETWEEN your view components and your NgRx State management

Applause from Nicholas Jamieson and 92 others

Thomas Burleson
Software architect with passion for Angular, UX, and great software products. Formerly Google Team Lead for AngularJS Material, @angular/flex-layout, and more..
Aug 11 · 4 min read

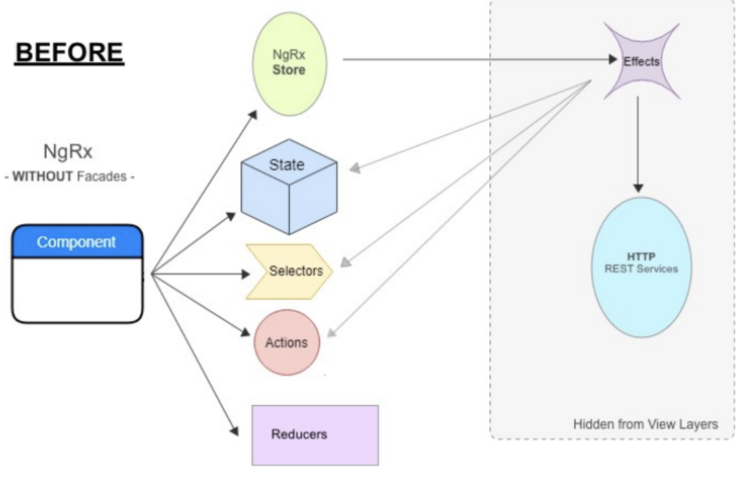
NgRx + Facades: Better State Management *

Prior to Nx 6.2, Nx already provided scalable state management with NgRx.

Now there is a new option when generating NgRx files to also generate a **facade** on top of your state management… to help you work even better at-scale.

Facades are a programming pattern in which a simpler public interface is provided to mask a composition of internal, more-complex, component usages.

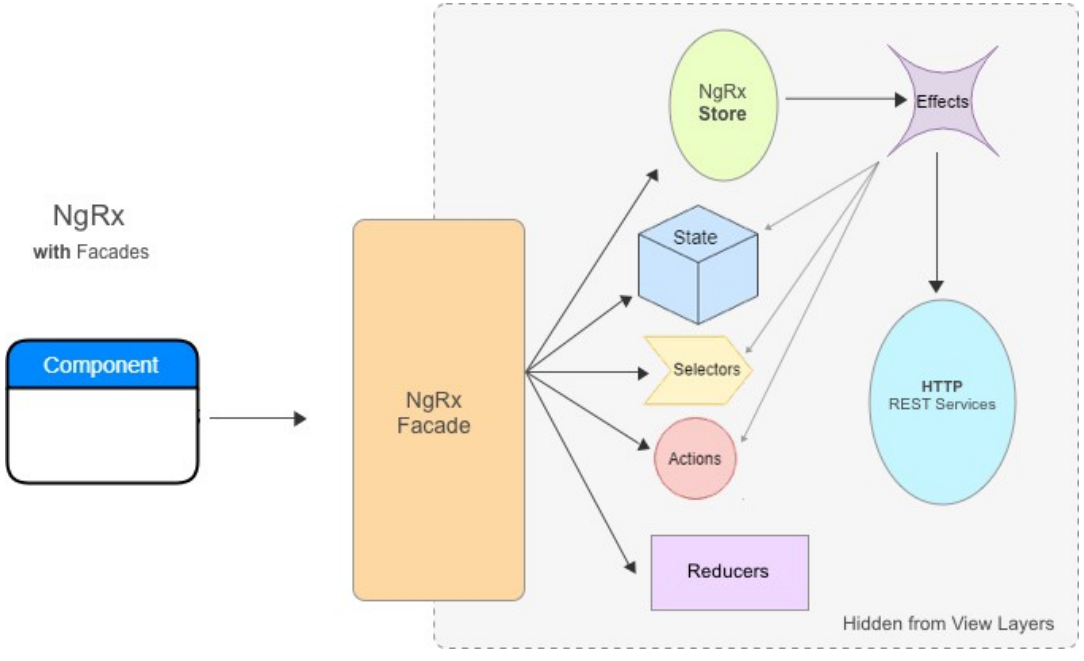
When writing a lot of NgRx code—as many enterprises do—developers quickly accumulate large collections of actions and selectors classes. These classes are used to [respectively] dispatch requests to- and query from- the NgRx Store.



Click to Read

http://bit.ly/2ALcrJx

NgRx Lab 6: NgRx **Facades**



- Create ContactsFacade
- Remove injected Store from ContactsList and ContactDetails view components

Lab Exercise

Angular: You may not need NgRx!

Thomas Burleson
Feb 6 · 7 min read

You may not need to use NgRx in your application. But you **should use Facades** in your application! Let's explore why, where, and how…

. . .

As part of my professional focus, I invest 10% of time training Angular developers around the world. And in every workshop, I tell the attending developers the following three (3) axioms:

- *RxJS* is the most **important** 'thing' to master in your career
- *NgRx* is the most **difficult** 'thing' to apply properly in applications
- *NgRx* is the most **fulfilling** 'thing' to use in real-world applications

Before you learn about NgRx, first make sure you have a solid RxJS foundation. And before you actually use NgRx, make sure to understand the impact and benefits to your code.

Now—with that said—you also may have heard from Aaron Frost, Mike Ryan, Brandon, or even Thomas Burleson (me) that "***You may not need NgRx!***"

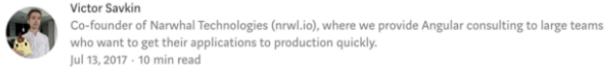
Using RxJS without State Management

To explore that statement, let's consider a simple application:

Click to Read

http://bit.ly/**2MUawHk**

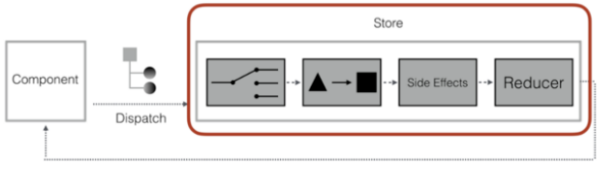
NgRx: Actions: Switchers & Deciders



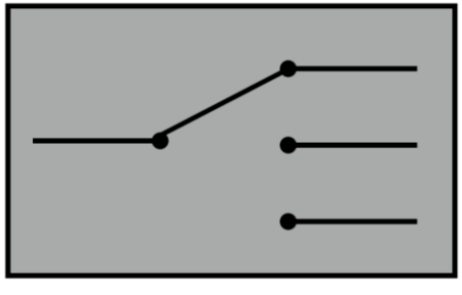
Victor Savkin
Co-founder of Narwhal Technologies (nrwl.io), where we provide Angular consulting to large teams who want to get their applications to production quickly.
Jul 13, 2017 · 10 min read

NgRx: Patterns and Techniques

Processing Actions



Content-Based Decider



[Read Blog Article]

http://bit.ly/**2DolTVI**

NgRx Effects: Routing

UpdateContact + Routing

contacts.effects.ts

```
@Injectable()
export class ContactsEffects {

  @Effect()
  updateContact$ = this.actions$.pipe(
    ofType(ContactsActionTypes.UPDATE_CONTACT),
    map((action: UpdateContactAction) => action.payload),
    concatMap((contact: Contact) => this.contactsService.updateContact(contact)),
    tap((contact: Contact) => this.router.navigate(['/contact', contact.id])),
    map((contact: Contact) => new UpdateContactSuccessAction(contact))
  );


}
```

Speaker notes

Here is an approach that uses `tapI)` to produce a routing side-effect AFTER we successfully update the contact on the server.

NgRx Effects: **Action Splitter**

Map incoming action to 1..n other actions

contacts.effects.ts

```
@Injectable()
export class ContactsEffects {

  // Action Decider (Splitter)
  @Effect()
  getContactDetails$ = this.actions$.pipe(
      ofType(ContactsActionTypes.LOAD_CONTACT_DETAILS),
      map(action => +action.selectedId),
      mergeMap(contactId => [
        new SelectContactAction(contactId),
        new LoadContactAction(contactId)
      ])
  );

}
```

A splitter is not concerned how SelectContact or LoadContact work... it only knows WHEN those actions should be triggered.

NgRx Effects: **Action Decider**

A content-based decider uses the action payload with OTHER data + `business logic` to decide on the resulting, outgoing action..

```
@Injectable()
export class ContactsEffects {
  selectedContact$ = this.store.pipe(
    select(ContactsQuery.getSelectedContact)
  );

  @Effect()
  loadIfMissing$ = this.actions$.pipe(
      ofType(ContactsActionTypes.LOAD_CONTACT),
      map((action: LoadContactAction) => action.payload),
      withLatestFrom( this.selectedContact$ ),
      switchMap(([contactId, selectedContact]) => {

      })
  );
}
```

withLatestFrom( ) is a super-powered operator!

Speaker notes

Here we are using a new RxJS operator 'withLatestFrom()' which will extract
the last value emitted from an observable and include it in the switchMap arguments.

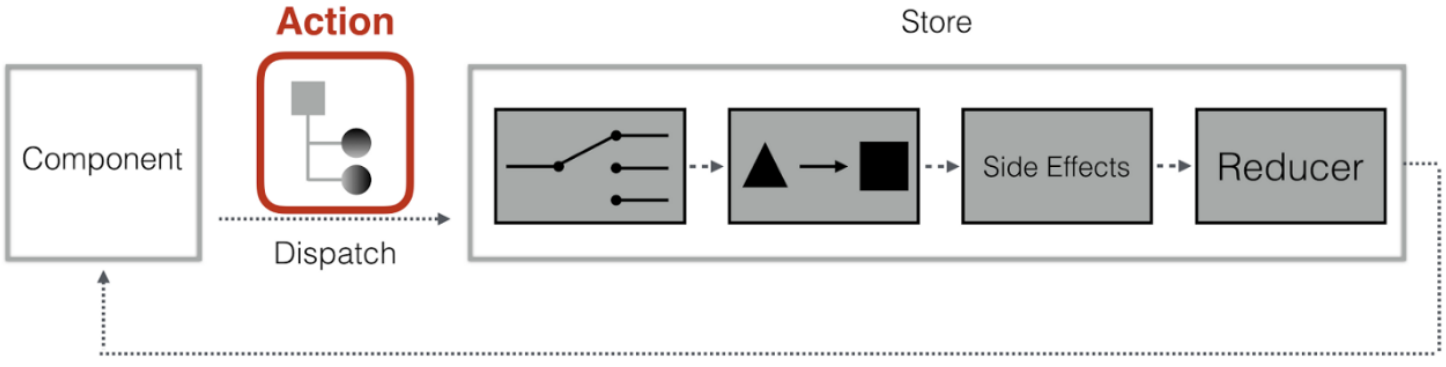
NgRx Effects: Action Decider

Based on current state, decide what to do next...

contacts.effects.ts

```
@Injectable()
export class ContactsEffects {
  selectedContact$ = this.store.pipe(
    select(ContactsQuery.getSelectedContact)
  );

  @Effect()
  loadIfMissing$ = this.actions$.pipe(
      ofType(ContactsActionTypes.LOAD_CONTACT),
      map((action: LoadContactAction) => action.payload),
      withLatestFrom( this.selectedContact$ ),
      switchMap(([contactId, selectedContact]) => {
        const contactLoaded = selectedContact && selectedContact.id === +contactId;
        const loadFromServer = () => this.contactsService.getContact(`${contactId}`)

        return !contactLoaded ? loadFromServer() : of(null);
      }),
      map((contact: Contact | null) => {
        return contact
            ? new AddContactAction(contact)
            : new NoopAction();
      })
  );
}
```

NgRx Lab 7: Effects with Deciders/Splitters



- Split 1 action into multiple actions; to spawn 1..n other actions
- Decide on *output action(s)* based on input content or biz logic

Lab Exercise

Using **@ngrx/entity**

- Provides **API** for working with collections
- Handles the **key/value** modeling for you
- Easy management of **collections** of similar objects.
- Simplify Reducer
- Reducer boilerplate
- Performant API for **CRUD** operations
- Protects against ***reducer mutation issues***
  because you don't write that code by hand

The **EntityState** Interface

The 'Hand-crafted' one...

```
export interface ContactsDictionary {
  [key: number]: Contact
}

export interface ContactsState {
  entities: ContactsDictionary;
  ids: number[];

  selectedContactId: number | null;
  loaded: boolean;
}
```

is replaced with one that **extends EntityState< >**

```
import {EntityState} from '@ngrx/entity';
import {Contact} from '../../models/contact';

export interface ContactsState extends EntityState<Contact> {
  selectedContactId: string;
  loaded: boolean;
  error?: any;
}
```

Speaker notes

Entity provides a predefined generic interface for an entity collection
Can include your own additional entity state properties on here as well

The **EntityAdapter**

The adapter is used to perform operations against the collection type.

```
export const adapter: EntityAdapter<Contact> = createEntityAdapter<Contact>();
```

or

contacts.reducer.ts

```
import {createEntityAdapter, EntityState, EntityAdapter} from '@ngrx/entity';
import {Contact} from '../../models/contact';

export const adapter: EntityAdapter<Contact> = createEntityAdapter<Contact>({
  selectId: (item: Contact) => String(item.id),
  sortComparer: false
});
```

Speaker notes

The 'sortComparer' only needed if the collection needs to be sorted when the state is updated... before to push to subscribers.

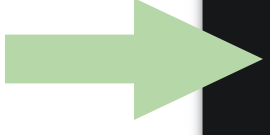
Initializing the State with the Adaptor

Use the **getInitialState** method on the adapter to set the initial state

contacts.reducer.ts

```
export const adapter: EntityAdapter<Contact> = createEntityAdapter<Contact>();

export function getContactsInitialState(): ContactsState {
  return adapter.getInitialState({
    selectedContactId: '',
    loaded: false,
    error: null
  });
}
```

Use **getInitialState** during registration



contacts-ngrx.module.ts

```
@NgModule({
  imports: [
    StoreModule.forFeature(
      FEATURE_CONTACTS,
      contactsReducer, {
      initialState: getContactsInitialState
      }
    ),
    EffectsModule.forFeature([ContactsEffects]),
    StoreDevTools
  ],
  providers: [
    ContactsFacade
  ]
})
export class ContactsNgRxModule { }
```

Speaker notes

getInitialState is a method for returning initial values for the Entity state
+ your CUSTOM feature state

Using @ngrx/entity **Adapter** methods

- **addOne**: Add one entity to the collection
- **addMany**: Add multiple entities to the collection
- **addAll**: Replace current collection with provided collection
- **removeOne**: Remove one entity from the collection
- **removeMany**: Remove multiple entities from the collection
- **removeAll**: Clear entity collection
- **updateOne**: Update one entity in the collection
- **updateMany**: Update multiple entities in the collection
- **upsertOne**: Update one entity in the collection
- **upsertMany**: Update multiple entities in the collection

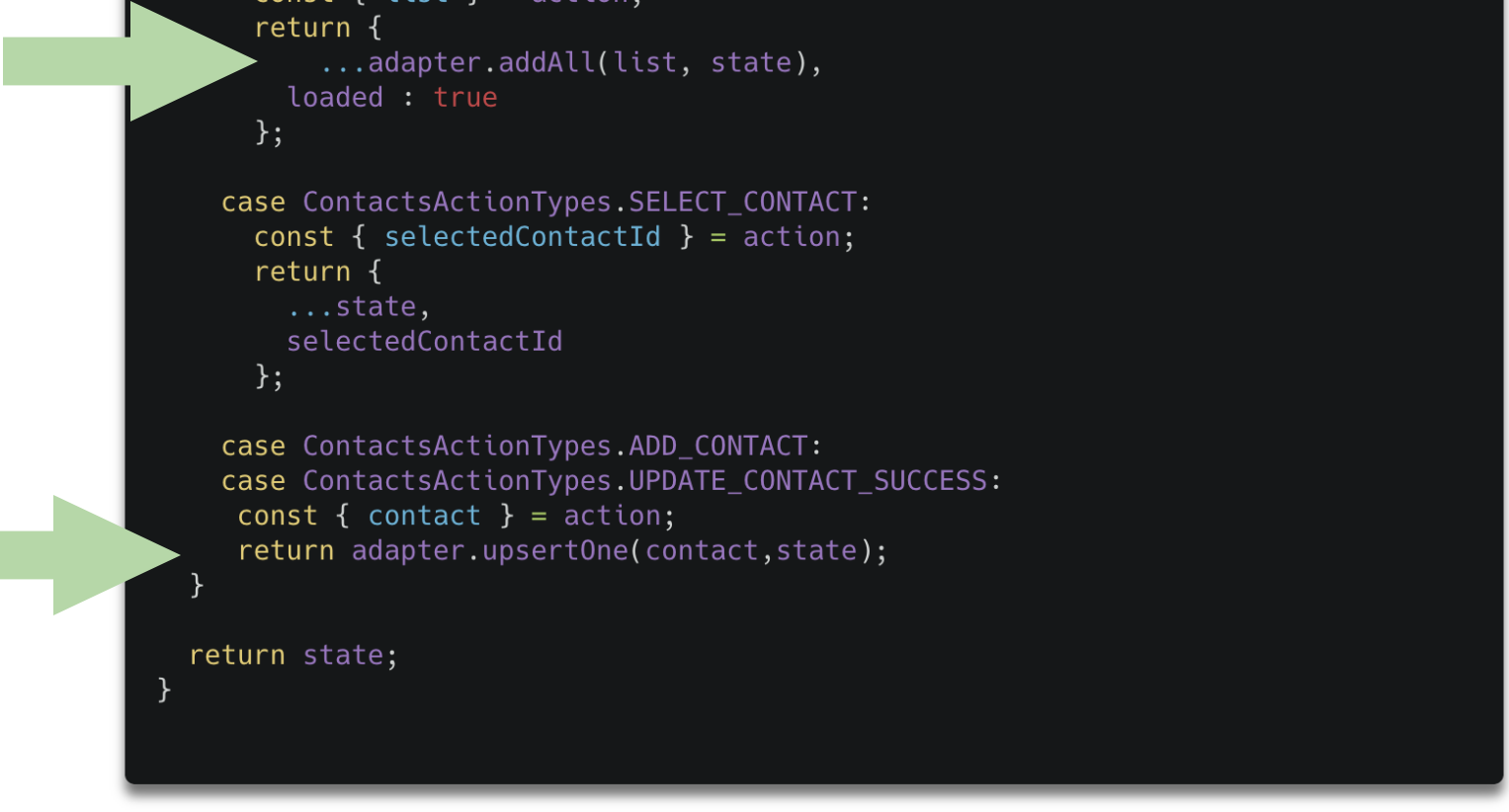
Speaker notes

The entire purpose of @ngrx/entity is to perform and manage CRUD operations.

These features are in the Adapter methods.

Using the **Adapter** in the Reducer

contacts.reducer.ts



```
export function contactsReducer(state: ContactsState, action: ContactsActions) {
  switch (action.type) {
    case ContactsActionTypes.LOAD_CONTACTS_SUCCESS:
      const { list } = action;
      return {
          ...adapter.addAll(list, state),
        loaded : true
      };

    case ContactsActionTypes.SELECT_CONTACT:
      const { selectedContactId } = action;
      return {
        ...state,
        selectedContactId
      };

    case ContactsActionTypes.ADD_CONTACT:
    case ContactsActionTypes.UPDATE_CONTACT_SUCCESS:
     const { contact } = action;
     return adapter.upsertOne(contact,state);
  }

  return state;
}
```

Speaker notes

Reducer logic becomes simpler

Just use adapter to change entity data; the Adapter ensures you are not mutating data!

Using **@ngrx/entity**

- Changed our state to extend EntityState<Ticket>
- Use an Entity Adapter with **createEntityAdapter**
- Use the adapter in our Reducer for CRUD operations.

Need to update our query selectors
to use **Entity Selectors**

Selectors with **@ngrx/entity**

The **getSelectors** method on the adapter returns an object of functions; used to extract (aka select) info from the entity collection

contacts.selectors.ts

```
const {
  selectAll,              // Array of Contact Items
  selectEntities,         // Dictionary: fast-lookup of Contact items
  selectIds,              // Contact IDs useful for sorting
  selectTotal             // Total count of available items in collection
} = adapter.getSelectors();
```

Selectors with **@ngrx/entity**


```
export namespace ContactsQuery {

  // Lookup the 'Contacts' feature state managed by NgRx
  const getContactsState     = createFeatureSelector<ContactsState>(FEATURE_CONTACTS);
  const getContactsEntities  = createSelector(getContactsState, selectEntities);

  export const getContacts   = createSelector(getContactsState, selectAll);
  export const getLoaded     = createSelector(getContactsState, (state:ContactsState) => state.loaded);
  export const getSelectedId = createSelector(getContactsState, (state:ContactsState) => {
    return state.selectedContactId
  });

  export const getSelectedContact = createSelector(
      getContactsEntities,
      getSelectedContactId,
      (contactEntities, id) => {
        return contactEntities[id];
      }
  );
}
```

contacts.selectors.ts

NgRx Lab 8: Use @ngrx/entity

```
import {createEntityAdapter, EntityState, EntityAdapter} from '@ngrx/entity';
import {Contact} from '../../models/contact';

export const adapter: EntityAdapter<Contact> = createEntityAdapter<Contact>({
  selectId: (item: Contact) => String(item.id),
  sortComparer: false
});
```

- Changed our state to **extend EntityState<Ticket>**
- Use an Entity Adapter with **createEntityAdapter**
- Simplify the ticketsReducer using **ticketsAdapter** for CRUD operations.
- Update our Selectors by composing **EntityAdapter Selectors**

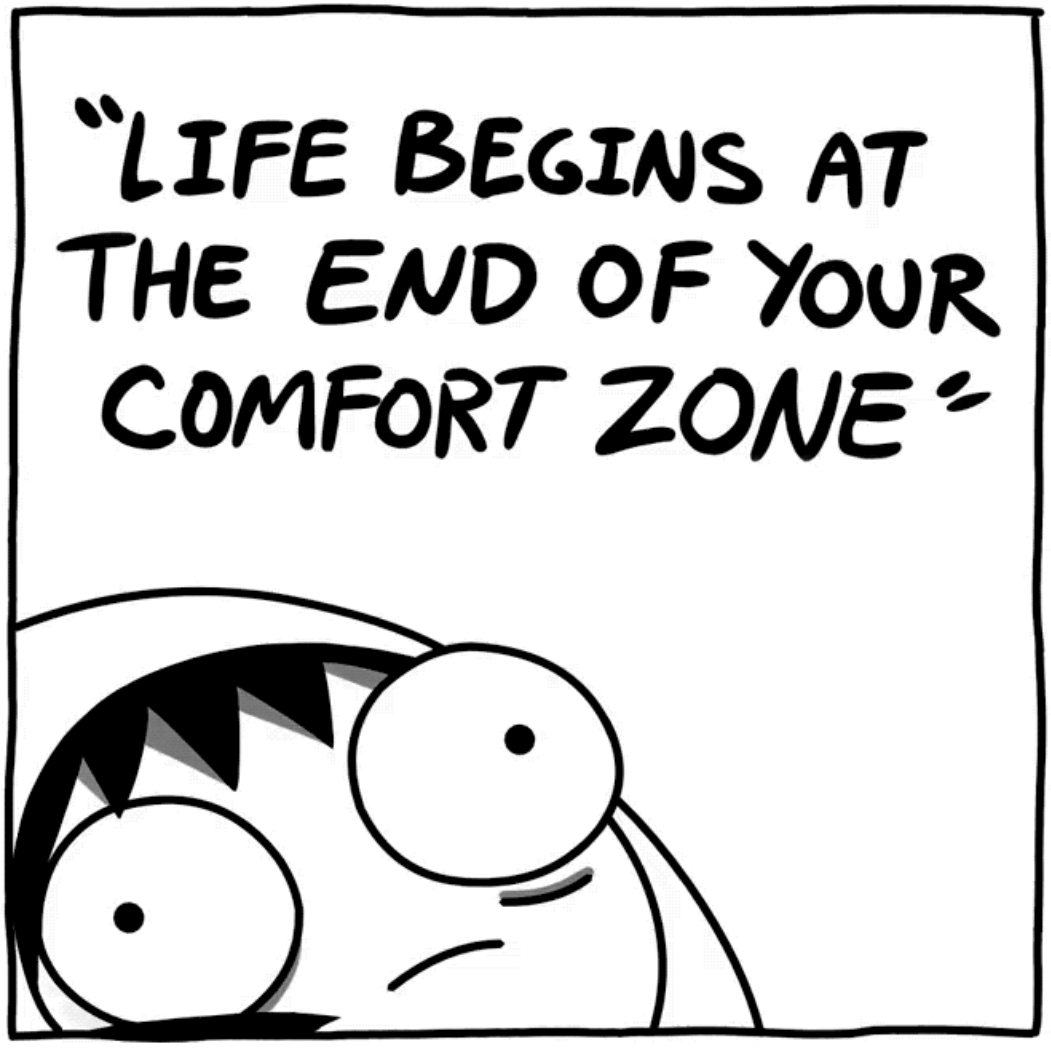
Lab Exercise

NgRx: Contact Searches with Routing

With all this work, the application still has some issues:

- Search criteria should be preserved between routes
- Filter in-memory or new server-API call ?
- Contacts list should be filtered based on search criteria

NgRx Lab 9: Managing **Search** State



NgRx Store as a Database & Immutability

- Think in terms of database design
- Put things that can be serialized in there
- Process state changes in NgRx **Reducers**
  - Avoid mutating state in reducers
  - Create objects and arrays with 'spread' operator
- Management of collections with NgRx **Entity**
- Public API from NgRx **Facades**
- Async work in NgRx **Effects**
- Leverage power of Nx **DataPersistence**

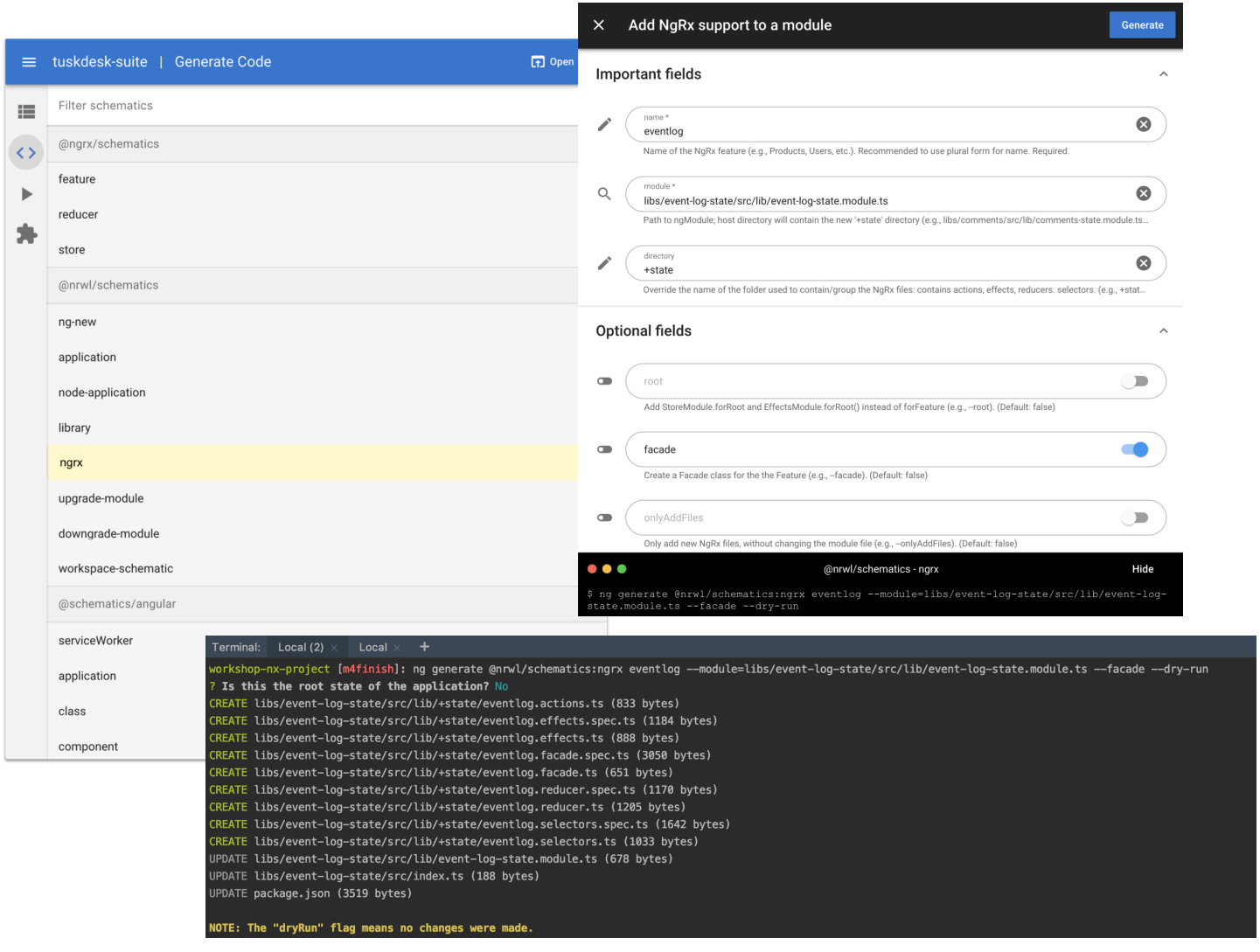
Remember **5 Principles of NgRx**

- **Single source of truth** - State of whole app is stored in a single state tree within a **single store**
- **State is read-only** - Only way to change state is to emit an **action**
- **Changes made only with reducers** - State tree is transformed or composed by pure **reducers**
- **Data is pushed** **-** Only way to read the state data is to use a selector and build a observable query
- **1-way Data Flows** **-** Read-only data flows into views only via observable queries.

NgRx: Other features you will need...

- **Search Criteria** (multiple UI-related data fields)
- **Pagination**
- Saves/Updates
- Better Router Integration
- Ghost Elements while loading
- etc.

Angular Console with Nx **ngrx** Schematics

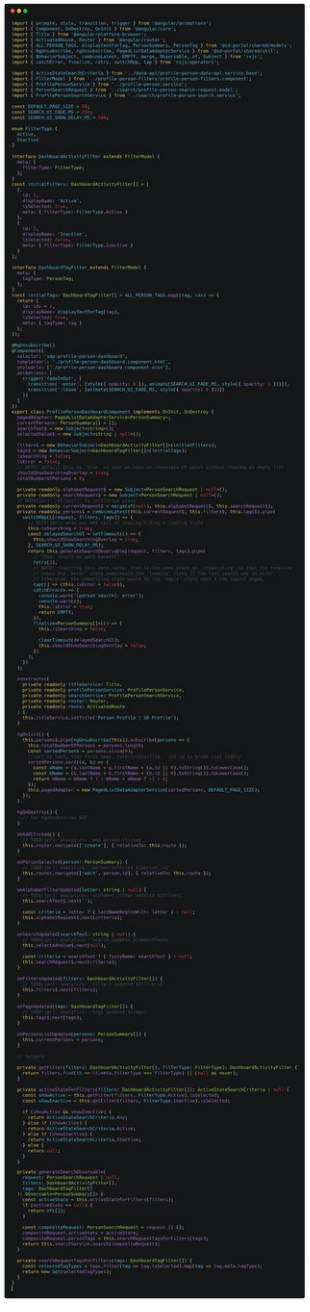


Speaker notes

Here we are generating NgRx state files to be part of the 'event-log-state' library.
The ngrx schematics will generate actions, reducer, selectors, effects, and facades... as well as associated test files.

These will all be registered with StoreModule.forFeature() and EffectsModule.forFeature()

NgRx has major impacts on Views & Testing



300 Lines of Code
Reduced to
30 Lines of Code