**Front End Architecture Guidelines**

This document describes the principles and design behind "generic" Elbit single page application. The guidelines described here should adhere to most single pages applications and should be considered before implementing a new app

This document covers the following areas

* Players
* Principles
* Layers
* Do's and don'ts
* Coding style

**Players**

* **Component** - A UI fragment that manages end user interaction and screen updates
* **Sensor –** A non-UI entity that raises events not related to end user activity. For example, a service which uses Web socket to implement server side events
* **Store** – A global container for UI/application state
* **Thunk –** A logical layer that handles commands from components. Encapsulates application flow
* **Proxy –** A thin wrapper around server REST api
* **Reducer –** A pure function that modifies application state

**Players (Deep dive)**

**Application store**

An object that holds all application state. Offers a dispatch method for executing actions and subscribe method for listening to state change events. Although feels like "anti pattern" the singe store principle is powerful. The store is read-only and should only be updated by dispatching action to reducers

**export interface** Contact {  
 **id**: **number**;  
 **name**: **string**;  
}  
  
**export interface** AppState {  
 **contacts**: Contact[];  
}

**export const** appStore: AppStore = createStore(*reducer*);

**Component**

A component is described by template, code and styles. A component is responsible for updating/synchronizing part of the screen against application state. We aim to develop slim components which are not responsible for application flow. Application flow is encapsulated by **thunks** and components should delegate the "hard" work to it. We distinguish between two types of components, smart and dumb. **Smart component** is a component that depends on application store and uses it by reading its state or by dispatching new actions. **Dumb component** is a component that does not access application store and instead receives the data from its parent component. Dumb component uses events to inform its parent about end user activity.

Below, is an example of smart component. The component uses the appStore service and reads some of its fields

<**ul**>  
 <**li \*ngFor="let *contact* of contacts"**>  
 <**span**>{{***contact***.**name**}}</**span**>  
 </**li**>  
</**ul**>

**import** { Component, OnInit } **from '@angular/core'**;  
**import** {AppStore} **from "../app.store"**;  
  
@Component({  
 **selector**: **'app-contact-page'**,  
 **templateUrl**: **'./contact-page.component.html'**,  
 **styleUrls**: [**'./contact-page.component.css'**]  
})  
**export class** ContactPageComponent {  
 **constructor**(**private** appStore: AppStore) { }  
  
 **get** contacts() {  
 **return this**.appStore.getState().**contacts**;  
 }  
}

**Action creator**

A method that returns an action. When the action returns a function, it is considered a **thunk**. The caller of an action creator should not really care whether an action creator returns an action or a function. A thunk handles asynchronous execution (for example HTTP) while plain action creator immediately returns an action that is handled by the reducer.

Below is an example of an action creator that return a plain action

@Injectable()  
**export class** ContactsActions {  
  
 **constructor**() { }  
  
 filter(filterStr: **string**) {  
 **return** {  
 **type**: **"FILTER"**,  
 filterStr,  
 }  
 }  
}

**Reducer**

A function that receives current application state + action and returns a new state object. A reducer should be pure and should not access contextual data. Its decision making should be based only on current application state and the requested action. A reducer cannot handle asynchronous execution (**thunks** can)

**function** *reducer*(state: AppState, action): AppState {  
 **if** (state == **null**) {  
 **return** *initialState*();  
 }  
  
 **if**(action.**type** == **"FILTER"**) {  
 **return** contactsReducers.*filter*(state, action.**filterStr**);  
 }  
  
 **return** state;  
}

**Proxy**

An angular service that knows how to send HTTP request to the server and returns back the server response. A proxy should not handle the response but rather return it back to its caller. Usually, **thunks** use proxies to fetch data from server. A proxy may perform some transformation/mutation of the response. Component should not deal directly with proxies. The whole purpose of proxies is to simplify thunk implementation by factoring out some HTTP details

@Injectable()  
**export class** ContactProxy {  
 **constructor**(**private** http: Http) {  
 }  
  
 getAll(): Observable<Contact> {  
 **return this**.http.get(**"/api/contact"**).**map**(res => res.json());  
 }  
}

**Sensor**

An object that is responsible for managing server commands. Server commands are usually pushed using web socket. A sensor is like a component, but, instead of receiving user actions it receives server commands. Just like component, a sensor delegates the server command to the appropriate **thunk** which manages the entire process of updating application state

**export class** AutoRefreshSensor {  
 **intervalId**;  
  
 **constructor**(**private** appStore: AppStore, **private** contactsActions: ContactsActions) {  
 }  
  
 start() {  
 **if**(**this**.**intervalId**) {  
 **return**;  
 }  
  
 **this**.**intervalId** = *setInterval*(()=> {  
 **this**.appStore.**dispatch**(**this**.contactsActions.reload())  
 }, 5000);  
 }  
  
 stop() {  
 **if**(**this**.**intervalId**) {  
 *clearInterval*(**this**.**intervalId**);  
 **this**.**intervalId** = **null**;  
 }  
 }  
}

**Principles**

Probably there are many principles that should be considered when implementing a single page application. Here, we try to describe only the most important ones

**Externalize component state**

A state that is held inside a component cannot be easily controlled by the outer world. Imagine a component with private fields. In case the component is required to be synchronized with another component you will have to use some kind of publisher/subscriber mechanism so one component can react to a change inside another one. Maintaining/understanding application flow which consists of many events is a hard task.   
A better solution is to externalize component state and hold it inside the app store. Thus, we can use reducers to keep different component's state synchronized with each other using functional/imperative techniques. Thus, no events are required to keep state synchronized between different components

**Arrange thunks in a tree hierarchy**

Thunks hold application flow. As such, it is important to organize them a tree structure where each thunk has children and single parent. The purpose is to prevent a case where cyclic dependencies exist between different thunks. A tree structure should allow us to easily understand application flow when handling a specific action.

**REST API should be defined according to the front-end needs**

The browser uses REST API to get/post information from server. A non-friendly backend is one that forces the client side to initiate multiple HTTP requests in order to get a specific data. Aggregating/transforming HTTP response at the client side is bad for performance. Browser machine is assumed to have weak hardware and in addition the browser enforces a single threaded execution model where both calculations and DOM updates are sharing the same thread and thus block each other. Prefer moving the transformation to the server side thus freeing the client side from CPU intensive operations

**Layers**

Above diagram describe the layers inside the application. The purpose of the diagram is to define the direction of interactions. A Component/Sensor is located higher in the layers diagram implies that Component uses Action creator but **not vice versa**

Our application reacts to user/server events. Therefore, we analyze any application sequence starting from the event itself.

For example, a user clicks a button. The click event is captured by a component. Components should be slim. Therefore, the component immediately delegates the work to an action creator. For most cases, the action creator interacts with the remote server using the Proxy layer and then updates application state using the dispatch method

**Dos and Don’ts**

**Don’t put complex logic into HTML templates.**

Prefer

<**button \*ngIf="allowDelete"**>Delete</**button**>

Over

<**button \*ngIf="authService.user.role == 'admin' || authService.user.role == 'manaeger'"**>Delete</**button**>

**Use component’s properties to encapsulate complex template expressions**

**export class** MyComponent {  
 **get** allowDelete(): **boolean** {  
 **return this**.authService.**user**.role == **'admin'** || **this**.authService.**user**.role == **'manaeger'**;  
 }  
}

**Don’t let component property do too much. A property should encapsulate an efficient JavaScript expression like appStore.getState().contacts. It should never runs loop and initiate asynchronous work**

**get** allowDelete(): **boolean** {  
 **const** user = **this**.authService.users.find(u=>u.**id**==**this**.authService.**userId**);  
 **return** user.**role** == **"admin"**;  
}

**Prefer component’s property that returns a field from application state over a field that holds a copy of application state and therefore need to be updated each time application state changes**

Prefer

**export class** ContactsPageComponent {  
 **constructor**(**private** appStore: AppStore) {  
 }  
  
 **get** contacts() {  
 **return this**.appStore.getState().**contacts**;  
 }  
}

Over

**export class** ContactsPageComponent {  
 **contacts**: Contact[];  
  
 **constructor**(**private** appStore: AppStore) {  
 }  
  
 ngOnInit() {  
 **this**.appStore.subscribe(()=> {  
 **this**.**contacts** = **this**.appStore.getState().**contacts**;  
 });  
 }  
}

**A thunk should return a promise. Thus, we can use async/await syntax to simplify management of multiple asynchronous invocations**

reload() {  
 **return async** (dispatch): Promise<**void**> => {  
 **const** contacts: Contact[] = **await this**.contactsProxy.getAll().**toPromise**();  
  
 dispatch(**this**.setContacts(contacts));  
 }  
}

**Don’t let Angular router control application flow. Routing is an implementation detail of the running thunk. A thunk may decide to navigate to a route once activity completes successfully.**

Prefer

**export class** ContactsPageComponent {  
 deleteContact() {  
 **this**.appStore.dispatch(**this**.contactsActions.deleteContact(**this**.contactId)]  
 }  
}

Over

**export class** ContactsPageComponent {  
 deleteContact() {  
 **this**.router.navigate([**"contact/delete"**, **this**.contactId]);  
 }  
}

**Prefer <a> which executes component method over navigating directly to a new route**

Prefer

<**button (click)="deleteContact()"**>Delete</**button**>

**export class** ContactsPageComponent {  
 deleteContact() {  
 **this**.appStore.dispatch(**this**.contactsActions.deleteContact(**this**.contactId)]  
 }  
}

Over

<**button routerLink="delete"**>Delete</**button**>

**Transformation of HTTP response data might be too expensive. Consider moving the transformation logic to the server side.**

**Each user action should initiate at maximum one HTTP request to the server. Multiple REST APIs request should be aggregated into one by creating a specific new REST API at the server side**

**Consider using web worker in case of complex/heavy transformation that must reside at client side**

**Don’t mutate application state inside ngOnInit. ngOnInit might be executed late during Angular change detection. Mutating application state at late stage might result with Angular error because of unstable UI state**

**Always change the default “display” property of an Angular component. The default “display” is inline which does not let you easily arrange child components. The display property should be changed to block/inline-block/flex**

**Enable Typescript compilation inside your editor. It is easier to get Typescript compilation errors inside IDE than to monitor Webpack outputs. Please note that at runtime only Webpack outputs are being used**

**Specify a Type for every field/parameter. Avoid any as much as possible. This way you will get a better refactoring experience**

**Application global look & feel (A.K.A theme) should be defined inside styles.css while specific component styling should reside inside each component’s css file**

**Don’t put both exported interface and exported implementation (for example, a function) inside the same file. It makes angular/cli nervous and eventually generates the following error “export ‘XXX’ was not found in ‘YYY’”**

**export interface** Contact {  
 **id**: **number**;  
 **name**: **string**;  
 **email**: **string**;  
}  
  
**export function** *xxx*() {  
}

**Style guide**

* See “Elbit Reference Application”
* Look at Angular style guide at <https://angular.io/guide/styleguide>