**Players**

**Application store** – An object that holds all application state. Offers a dispatch method for executing actions and subscribe method for listening to state changes.

**Smart component** – A component that depends on application store. For example, ContactPageComponent displays all contacts stored inside appStore.getState().contacts

**Dumb component** – A component that does not access application store and instead receives the data from its parent component. For example, ContactListComponent has an input named contacts. Its parent, ContactPageComponent, takes the contact list from application state and sends it to the ContactListComponent via Angular data binding mechanism (@Input)

**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.

**Proxy –** An angular service that knows how to send HTTP request to 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 with proxies. The whole purpose of proxies is to simplify thunk implementation by factoring out HTTP details

**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 to whole process of updating application state

**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

**Notes**

* Most of our application code should reside inside action creators. Action creators should not access the DOM directly but rather use the store to update application state and then let Angular do its magic
* Once application grows, it is acceptable that one action creator uses the other. This is our primary way for factoring out common activities. In that case it is important that we arrange action creators in a strict hierarchy where no cycle exist
* Avoiding cycles between different layers is challenging. By default, all code reside inside the same project and therefore the programmer may accidently access an action creator from inside the proxy layer. To void that we may consider using different NPM package per layer or implement some kind or profiling mechanism to catch that king of errors

**Dos and Don’ts**

1. Don’t put complex logic into HTML templates
2. Use component’s properties to encapsulate complex template expressions
3. 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.
4. 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
5. A thunk should return a proxy. Thus, we can use async/await syntax to simplify management of multiple asynchronous invocations
6. 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.
7. Prefer <a> which executes component method over navigating directly to a new route
8. Transformation of HTTP response data might be too expensive. Consider moving to transformation logic to the server side.
9. Consider using web worker in case of complex/heavy transformation that must reside at client side
10. 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
11. 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 state
12. Always change the default “display” property of Angular component. The default “display” is inline which does not let you easily arrange child components. The display property should be changes to block/inline-block/flex
13. 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
14. Specify a Type for every field/parameter. Avoid any as much as possible. This way you will get a better refactor experience
15. 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
16. 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 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>