D:\应用程序案例\global-message\global-message.module.ts

@NgModule({

imports: [

GlobalMessageStoreModule,

EffectsModule.forFeature([GlobalMessageEffect]),

],

providers: [

provideDefaultConfigFactory(defaultGlobalMessageConfigFactory),

GlobalMessageService,

{ provide: GlobalMessageConfig, useExisting: Config },

],

})

export class GlobalMessageModule {

static forRoot(): ModuleWithProviders<GlobalMessageModule> {

return {

ngModule: GlobalMessageModule,

providers: [...errorHandlers, ...httpErrorInterceptors],

};

}

}

export function defaultGlobalMessageConfigFactory(): GlobalMessageConfig {

return {

globalMessages: {

[GlobalMessageType.MSG\_TYPE\_CONFIRMATION]: {

timeout: 3000,

},

[GlobalMessageType.MSG\_TYPE\_INFO]: {

timeout: 3000,

},

[GlobalMessageType.MSG\_TYPE\_ERROR]: {

timeout: 7000,

},

[GlobalMessageType.MSG\_TYPE\_WARNING]: {

timeout: 7000,

},

},

};

}

export type GlobalMessageTypeConfig = {

timeout?: number;

};

export abstract class GlobalMessageConfig {

globalMessages: {

[GlobalMessageType.MSG\_TYPE\_CONFIRMATION]?: GlobalMessageTypeConfig;

[GlobalMessageType.MSG\_TYPE\_INFO]?: GlobalMessageTypeConfig;

[GlobalMessageType.MSG\_TYPE\_ERROR]?: GlobalMessageTypeConfig;

[GlobalMessageType.MSG\_TYPE\_WARNING]?: GlobalMessageTypeConfig;

};

}

@Injectable()

export class GlobalMessageService {

constructor(protected store: Store<StateWithGlobalMessage>) {}

/\*\*

\* Get all global messages

\*/

get(): Observable<GlobalMessageEntities> {

return this.store.pipe(

select(GlobalMessageSelectors.getGlobalMessageEntities),

filter((data) => data !== undefined)

);

}

/\*\*

\* Add one message into store

\* @param text: string | Translatable

\* @param type: GlobalMessageType object

\* @param timeout: number

\*/

add(

text: string | Translatable,

type: GlobalMessageType,

timeout?: number

): void {

this.store.dispatch(

new GlobalMessageActions.AddMessage({

text: typeof text === 'string' ? { raw: text } : text,

type,

timeout,

})

);

}

/\*\*

\* Remove message(s) from store

\* @param type: GlobalMessageType

\* @param index:optional. Without it, messages will be removed by type; otherwise,

\* message will be removed from list by index.

\*/

remove(type: GlobalMessageType, index?: number): void {

this.store.dispatch(

index !== undefined

? new GlobalMessageActions.RemoveMessage({

type: type,

index: index,

})

: new GlobalMessageActions.RemoveMessagesByType(type)

);

}

}

@Injectable({ providedIn: 'root' })

export class HttpErrorInterceptor implements HttpInterceptor {

constructor(

@Inject(HttpErrorHandler) protected handlers: HttpErrorHandler[]

) {

// We reverse the handlers to allow for custom handlers

// that replace standard handlers

this.handlers.reverse();

}

intercept(

request: HttpRequest<any>,

next: HttpHandler

): Observable<HttpEvent<any>> {

return next.handle(request).pipe(

catchError((response: any) => {

if (response instanceof HttpErrorResponse) {

this.handleErrorResponse(request, response);

return throwError(response);

}

})

);

}

protected handleErrorResponse(

request: HttpRequest<any>,

response: HttpErrorResponse

): void {

const handler = this.getResponseHandler(response);

if (handler) {

handler.handleError(request, response);

}

}

/\*\*

\* return the error handler that matches the `HttpResponseStatus` code.

\* If no handler is available, the UNKNOWN handler is returned.

\*/

protected getResponseHandler(response: HttpErrorResponse): HttpErrorHandler {

const status = response.status;

let handler = this.handlers.find((h) => h.responseStatus === status);

if (!handler) {

handler = this.handlers.find(

(h) => h.responseStatus === HttpResponseStatus.UNKNOWN

);

}

return handler;

}

}

// store

@NgModule({

imports: [

StateModule,

StoreModule.forFeature(GLOBAL\_MESSAGE\_FEATURE, reducerToken),

],

providers: [reducerProvider],

})

export class GlobalMessageStoreModule {}

// state

export const GLOBAL\_MESSAGE\_FEATURE = 'global-message';

export interface StateWithGlobalMessage {

[GLOBAL\_MESSAGE\_FEATURE]: GlobalMessageState;

}

export interface GlobalMessageState {

entities: GlobalMessageEntities;

}

export interface GlobalMessageEntities {

[messageType: string]: Translatable[];

}

// action

export const ADD\_MESSAGE = '[Global-message] Add a Message';

export const REMOVE\_MESSAGE = '[Global-message] Remove a Message';

export const REMOVE\_MESSAGES\_BY\_TYPE =

'[Global-message] Remove messages by type';

export class AddMessage implements Action {

readonly type = ADD\_MESSAGE;

constructor(public payload: GlobalMessage) {}

}

export class RemoveMessage implements Action {

readonly type = REMOVE\_MESSAGE;

constructor(public payload: { type: GlobalMessageType; index: number }) {}

}

export class RemoveMessagesByType implements Action {

readonly type = REMOVE\_MESSAGES\_BY\_TYPE;

constructor(public payload: GlobalMessageType) {}

}

export type GlobalMessageAction =

| AddMessage

| RemoveMessage

| RemoveMessagesByType;

// reducers

export const initialState: GlobalMessageState = {

entities: {},

};

export function reducer(

state = initialState,

action: GlobalMessageActions.GlobalMessageAction

): GlobalMessageState {

switch (action.type) {

case GlobalMessageActions.ADD\_MESSAGE: {

const message: GlobalMessage = action.payload;

if (state.entities[message.type] === undefined) {

return {

...state,

entities: {

...state.entities,

[message.type]: [message.text],

},

};

} else {

const currentMessages: Translatable[] = state.entities[message.type];

return {

...state,

entities: {

...state.entities,

[message.type]: [...currentMessages, message.text],

},

};

}

}

case GlobalMessageActions.REMOVE\_MESSAGE: {

const msgType: GlobalMessageType = action.payload.type;

const msgIndex: number = action.payload.index;

if (

Object.keys(state.entities).length === 0 ||

!state.entities[msgType]

) {

return state;

}

const messages = [...state.entities[msgType]];

messages.splice(msgIndex, 1);

return {

...state,

entities: {

...state.entities,

[msgType]: messages,

},

};

}

case GlobalMessageActions.REMOVE\_MESSAGES\_BY\_TYPE: {

const entities = {

...state.entities,

[action.payload]: [],

};

return {

...state,

entities,

};

}

}

return state;

}

// effects

@Injectable()

export class GlobalMessageEffect {

@Effect()

removeDuplicated$: Observable<

GlobalMessageActions.RemoveMessage

> = this.actions$.pipe(

ofType(GlobalMessageActions.ADD\_MESSAGE),

pluck('payload'),

switchMap((message: GlobalMessage) =>

of(message.text).pipe(

withLatestFrom(

this.store.pipe(

select(

GlobalMessageSelectors.getGlobalMessageEntitiesByType(

message.type

)

)

)

),

filter(

([text, messages]: [Translatable, Translatable[]]) =>

countOfDeepEqualObjects(text, messages) > 1

),

map(

([text, messages]: [Translatable, Translatable[]]) =>

new GlobalMessageActions.RemoveMessage({

type: message.type,

index: indexOfFirstOccurrence(text, messages),

})

)

)

)

);

@Effect()

hideAfterDelay$: Observable<

GlobalMessageActions.RemoveMessage

> = isPlatformBrowser(this.platformId) // we don't want to run this logic when doing SSR

? this.actions$.pipe(

ofType(GlobalMessageActions.ADD\_MESSAGE),

pluck('payload'),

concatMap((message: GlobalMessage) => {

const config = this.config.globalMessages[message.type];

return this.store.pipe(

select(

GlobalMessageSelectors.getGlobalMessageCountByType(message.type)

),

take(1),

filter(

(count: number) =>

((config && config.timeout !== undefined) || message.timeout) &&

count &&

count > 0

),

delay(message.timeout || config.timeout),

switchMap(() =>

of(

new GlobalMessageActions.RemoveMessage({

type: message.type,

index: 0,

})

)

)

);

})

)

: EMPTY;

constructor(

private actions$: Actions,

private store: Store<StateWithGlobalMessage>,

private config: GlobalMessageConfig,

@Inject(PLATFORM\_ID) private platformId: any

) {}

}

// selectors

export const getGlobalMessageEntities: MemoizedSelector<

StateWithGlobalMessage,

GlobalMessageEntities

> = createSelector(

getGlobalMessageState,

(state: GlobalMessageState) => state.entities

);

export const getGlobalMessageEntitiesByType = (

type: GlobalMessageType

): MemoizedSelector<StateWithGlobalMessage, Translatable[]> => {

return createSelector(

getGlobalMessageEntities,

(entities) => entities && entities[type]

);

};

export const getGlobalMessageCountByType = (

type: GlobalMessageType

): MemoizedSelector<StateWithGlobalMessage, number> => {

return createSelector(

getGlobalMessageEntitiesByType(type),

(entities) => entities && entities.length

);

};

export const getGlobalMessageState: MemoizedSelector<

StateWithGlobalMessage,

GlobalMessageState

> = createFeatureSelector<GlobalMessageState>(GLOBAL\_MESSAGE\_FEATURE);