site-context

@NgModule({

imports: [StateModule, SiteContextStoreModule],

})

export class SiteContextModule {

static forRoot(): ModuleWithProviders<SiteContextModule> {

return {

ngModule: SiteContextModule,

providers: [

provideDefaultConfigFactory(defaultSiteContextConfigFactory),

contextServiceMapProvider,

...contextServiceProviders,

...siteContextParamsProviders,

{ provide: SiteContextConfig, useExisting: Config },

provideConfigValidator(baseSiteConfigValidator),

],

};

}

}

/\*\*

\* config

\*/

export abstract class SiteContextConfig {

context?: { urlParameters?: string[];

[contextName: string]: string[];

};

}

import { SiteContextConfig } from './site-context-config';

import { CURRENCY\_CONTEXT\_ID, LANGUAGE\_CONTEXT\_ID,

} from '../providers/context-ids';

export function defaultSiteContextConfigFactory(): SiteContextConfig {

return {

context: {

[LANGUAGE\_CONTEXT\_ID]: [ 'en', 'pa', ],

[CURRENCY\_CONTEXT\_ID]: ['USD', 'ZAR', ],

},

};

}

import { SiteContextConfig } from './site-context-config';

export function getContextParameterValues( config: SiteContextConfig, parameter: string

): string[] {

return (config.context && config.context[parameter]) || [];

}

export function getContextParameterDefault( config: SiteContextConfig, parameter: string): string {

const param = getContextParameterValues(config, parameter);

return param && param.length ? param[0] : undefined;

}

export function baseSiteConfigValidator(config: SiteContextConfig) {

if (getContextParameterDefault(config, BASE\_SITE\_CONTEXT\_ID) === undefined) {

return 'Please configure context.parameters.baseSite before using storefront library!';

}

}

@Injectable({ providedIn: 'root',})

export class SiteConnector {

constructor(protected adapter: SiteAdapter) {}

getLanguages(): Observable<Language[]> {

return this.adapter.loadLanguages();

}

getCurrencies(): Observable<Currency[]> {

return this.adapter.loadCurrencies();

}

getCountries(type?: CountryType): Observable<Country[]> {

return this.adapter.loadCountries(type);

}

getRegions(countryIsoCode: string): Observable<Region[]> {

return this.adapter.loadRegions(countryIsoCode);

}

getBaseSite(): Observable<BaseSite> {

return this.adapter.loadBaseSite();

}

}

export abstract class SiteAdapter {

abstract loadLanguages(): Observable<Language[]>;

abstract loadCurrencies(): Observable<Currency[]>;

abstract loadCountries(type?: CountryType): Observable<Country[]>;

abstract loadRegions(countryIsoCode: string): Observable<Region[]>;

abstract loadBaseSite(): Observable<BaseSite>;

}

export const LANGUAGE\_NORMALIZER = new InjectionToken<Converter<any, Language>>(

'LanguageNormalizer'

);

export const CURRENCY\_NORMALIZER = new InjectionToken<Converter<any, Currency>>(

'CurrencyNormalizer'

);

export const COUNTRY\_NORMALIZER = new InjectionToken<Converter<any, Country>>(

'CountryNormalizer'

);

export const REGION\_NORMALIZER = new InjectionToken<Converter<any, Region>>(

'RegionNormalizer'

);

/\*

\* facade

\*/

export interface SiteContext<T> {

getAll(): Observable<T[]>;

getActive(): Observable<string>;

setActive(isocode: string);

}

/\*\*

\* Facade that provides easy access to language state, actions and selectors.

\*/

@Injectable()

export class LanguageService implements SiteContext<Language> {

private sessionStorage: Storage;

constructor(

protected store: Store<StateWithSiteContext>,

winRef: WindowRef,

protected config: SiteContextConfig

) {

this.sessionStorage = winRef.sessionStorage;

}

/\*\*

\* Represents all the languages supported by the current store.

\*/

getAll(): Observable<Language[]> {

return this.store.pipe(

select(SiteContextSelectors.getAllLanguages),

tap((languages) => {

if (!languages) {

this.store.dispatch(new SiteContextActions.LoadLanguages());

}

}),

filter((languages) => Boolean(languages))

);

}

/\*\*

\* Represents the isocode of the active language.

\*/

getActive(): Observable<string> {

return this.store.pipe(

select(SiteContextSelectors.getActiveLanguage),

filter((active) => Boolean(active))

);

}

/\*\*

\* Sets the active language.

\*/

setActive(isocode: string) {

return this.store

.pipe(select(SiteContextSelectors.getActiveLanguage), take(1))

.subscribe((activeLanguage) => {

if (activeLanguage !== isocode) {

this.store.dispatch(

new SiteContextActions.SetActiveLanguage(isocode)

);

}

});

}

initialize() {

const sessionLanguage =

this.sessionStorage && this.sessionStorage.getItem('language');

if (

sessionLanguage &&

getContextParameterValues(this.config, LANGUAGE\_CONTEXT\_ID).includes(

sessionLanguage

)

) {

this.setActive(sessionLanguage);

} else {

this.setActive(

getContextParameterDefault(this.config, LANGUAGE\_CONTEXT\_ID)

);

}

}

}

@Injectable()

export class CurrencyService implements SiteContext<Currency> {

private sessionStorage: Storage;

constructor(

protected store: Store<StateWithSiteContext>, winRef: WindowRef,

protected config: SiteContextConfig

) { this.sessionStorage = winRef.sessionStorage; }

getAll(): Observable<Currency[]> {

return this.store.pipe(

select(SiteContextSelectors.getAllCurrencies),

tap((currencies) => {

if (!currencies) {

this.store.dispatch(new SiteContextActions.LoadCurrencies());

}

}),

filter((currenies) => Boolean(currenies))

);

}

getActive(): Observable<string> {

return this.store.pipe(

select(SiteContextSelectors.getActiveCurrency),

filter((active) => Boolean(active))

);

}

setActive(isocode: string) {

return this.store

.pipe(select(SiteContextSelectors.getActiveCurrency), take(1))

.subscribe((activeCurrency) => {

if (activeCurrency !== isocode) {

this.store.dispatch(

new SiteContextActions.SetActiveCurrency(isocode)

);

}

});

}

initialize() {

const sessionCurrency =

this.sessionStorage && this.sessionStorage.getItem('currency');

if (

sessionCurrency &&

getContextParameterValues(this.config, CURRENCY\_CONTEXT\_ID).includes(

sessionCurrency

)

) { this.setActive(sessionCurrency); } else {

this.setActive( getContextParameterDefault(this.config, CURRENCY\_CONTEXT\_ID) );

}

}

}

@Injectable()

export class BaseSiteService implements SiteContext<string> {

constructor(

protected store: Store<StateWithSiteContext>,

protected config: SiteContextConfig

) {}

getActive(): Observable<string> {

return this.store.pipe(

select(SiteContextSelectors.getActiveBaseSite),

filter((active) => Boolean(active))

);

}

getAll(): Observable<string[]> {

return this.getActive().pipe(map((baseSite) => [baseSite]));

}

setActive(baseSite: string): Subscription {

return this.store

.pipe(select(SiteContextSelectors.getActiveBaseSite), take(1))

.subscribe((activeBaseSite) => {

if (baseSite && activeBaseSite !== baseSite) {

this.store.dispatch(

new SiteContextActions.SetActiveBaseSite(baseSite)

);

}

});

}

initialize(): void {

this.setActive(

getContextParameterDefault(this.config, BASE\_SITE\_CONTEXT\_ID)

);

}

getBaseSiteData(): Observable<BaseSite> {

return this.store.pipe(

select(SiteContextSelectors.getBaseSiteData),

tap((baseSite) => {

if (Object.keys(baseSite).length === 0) {

this.store.dispatch(new SiteContextActions.LoadBaseSite());

}

})

);

}

}

/\*

\* providers

\*/

export const LANGUAGE\_CONTEXT\_ID = 'language';

export const CURRENCY\_CONTEXT\_ID = 'currency';

export const BASE\_SITE\_CONTEXT\_ID = 'baseSite';

export abstract class ContextServiceMap {

[context: string]: Type<SiteContext<any>>;

}

export function serviceMapFactory() {

return {

[LANGUAGE\_CONTEXT\_ID]: LanguageService,

[CURRENCY\_CONTEXT\_ID]: CurrencyService,

[BASE\_SITE\_CONTEXT\_ID]: BaseSiteService,

};

}

export const contextServiceMapProvider: Provider = {

provide: ContextServiceMap,

useFactory: serviceMapFactory,

};

export function initializeContext(

baseSiteService: BaseSiteService,

langService: LanguageService,

currService: CurrencyService,

configInit: ConfigInitializerService

) {

return () => {

configInit.getStableConfig('context').then(() => {

baseSiteService.initialize();

langService.initialize();

currService.initialize();

});

};

}

export const contextServiceProviders: Provider[] = [

BaseSiteService,

LanguageService,

CurrencyService,

{

provide: APP\_INITIALIZER,

useFactory: initializeContext,

deps: [ BaseSiteService, LanguageService, CurrencyService, ConfigInitializerService, ],

multi: true,

},

];

// functions below should not be exposed in public API:

export function initSiteContextRoutesHandler(

siteContextRoutesHandler: SiteContextRoutesHandler,

configInit: ConfigInitializerService

) {

return () => {

configInit.getStableConfig('context').then(() => {

siteContextRoutesHandler.init();

});

};

}

export const siteContextParamsProviders: Provider[] = [

SiteContextParamsService,

SiteContextUrlSerializer,

{ provide: UrlSerializer, useExisting: SiteContextUrlSerializer },

{

provide: APP\_INITIALIZER,

useFactory: initSiteContextRoutesHandler,

deps: [SiteContextRoutesHandler, ConfigInitializerService],

multi: true,

},

];

/\*

\* services

\*/

@Injectable()

export class SiteContextParamsService {

constructor(

private config: SiteContextConfig, private injector: Injector, private serviceMap: ContextServiceMap

) {}

getContextParameters(): string[] {

if (this.config.context) {

return Object.keys(this.config.context).filter(

(param) => param !== 'urlParameters'

);

}

return [];

}

getUrlEncodingParameters(): string[] {

return (this.config.context && this.config.context.urlParameters) || [];

}

getParamValues(param: string): string[] {

return getContextParameterValues(this.config, param);

}

getParamDefaultValue(param: string): string {

return getContextParameterDefault(this.config, param);

}

getSiteContextService(param: string): SiteContext<any> {

if (this.serviceMap[param]) {

return this.injector.get<SiteContext<any>>(this.serviceMap[param], null);

}

}

getValue(param: string): string {

let value: string;

const service = this.getSiteContextService(param);

if (service) {

service .getActive() .subscribe((val) => (value = val)) .unsubscribe();

}

return value !== undefined ? value : this.getParamDefaultValue(param);

}

setValue(param: string, value: string) {

const service = this.getSiteContextService(param);

if (service) { service.setActive(value); }

}

getValues(params: string[]): Observable<Array<string>> {

if (params.length === 0) { return of([]); }

return combineLatest(

params.map((param) =>

this.getSiteContextService(param) .getActive() .pipe(distinctUntilChanged())

)

).pipe(filter((value) => value.every((param) => !!param)));

}

}

@Injectable({ providedIn: 'root',})

export class SiteContextRoutesHandler implements OnDestroy {

constructor(

private siteContextParams: SiteContextParamsService,

private serializer: SiteContextUrlSerializer,

private injector: Injector

) {}

private subscription = new Subscription();

private contextValues: {

[param: string]: string;

} = {};

private router: Router;

private location: Location;

private isNavigating = false;

init() {

this.router = this.injector.get<Router>(Router);

this.location = this.injector.get<Location>(Location);

const routingParams = this.siteContextParams.getUrlEncodingParameters();

if (routingParams.length) {

this.setContextParamsFromRoute(this.router.url);

this.subscribeChanges(routingParams);

this.subscribeRouting();

}

}

private subscribeChanges(params: string[]) {

params.forEach((param) => {

const service = this.siteContextParams.getSiteContextService(param);

if (service) {

this.subscription.add(

service.getActive().subscribe((value) => {

if (

!this.isNavigating && this.contextValues[param] && this.contextValues[param] !== value

) {

const parsed = this.router.parseUrl(this.router.url);

const serialized = this.router.serializeUrl(parsed);

this.location.replaceState(serialized);

}

this.contextValues[param] = value;

})

);

}

});

}

private subscribeRouting() {

this.subscription.add(

this.router.events

.pipe(

filter(

(event) =>

event instanceof NavigationStart || event instanceof NavigationEnd ||

event instanceof NavigationError || event instanceof NavigationCancel

)

)

.subscribe((event: RouterEvent) => {

this.isNavigating = event instanceof NavigationStart;

if (this.isNavigating) {

this.setContextParamsFromRoute(event.url);

}

})

);

}

private setContextParamsFromRoute(url: string) {

const { params } = this.serializer.urlExtractContextParameters(url);

Object.keys(params).forEach((param) =>

this.siteContextParams.setValue(param, params[param])

);

}

ngOnDestroy(): void { this.subscription.unsubscribe(); }

}

export interface ParamValuesMap { [name: string]: string;}

export interface UrlTreeWithSiteContext extends UrlTree {

siteContext?: ParamValuesMap;

}

const UrlSplit = /(^[^#?]\*)(.\*)/; // used to split url into path and query/fragment parts

@Injectable()

export class SiteContextUrlSerializer extends DefaultUrlSerializer {

private get urlEncodingParameters(): string[] {

return this.siteContextParams.getUrlEncodingParameters();

}

get hasContextInRoutes() {

return this.urlEncodingParameters.length > 0;

}

constructor(private siteContextParams: SiteContextParamsService) {

super();

}

parse(url: string): UrlTreeWithSiteContext {

if (this.hasContextInRoutes) {

const urlWithParams = this.urlExtractContextParameters(url);

const parsed = super.parse(urlWithParams.url) as UrlTreeWithSiteContext;

this.urlTreeIncludeContextParameters(parsed, urlWithParams.params);

return parsed;

} else {

return super.parse(url);

}

}

urlExtractContextParameters( url: string ): { url: string; params: ParamValuesMap } {

const [, urlPart, queryPart] = url.match(UrlSplit);

const segments = urlPart.split('/');

if (segments[0] === '') {

segments.shift();

}

const params = {};

let paramId = 0;

let segmentId = 0;

while (

paramId < this.urlEncodingParameters.length &&

segmentId < segments.length

) {

const paramName = this.urlEncodingParameters[paramId];

const paramValues = this.siteContextParams.getParamValues(paramName);

if (paramValues.includes(segments[segmentId])) {

params[paramName] = segments[segmentId];

segmentId++;

}

paramId++;

}

url = segments.slice(Object.keys(params).length).join('/') + queryPart;

return { url, params };

}

private urlTreeIncludeContextParameters(

urlTree: UrlTreeWithSiteContext,

params: ParamValuesMap

) {

urlTree.siteContext = params;

}

serialize(tree: UrlTreeWithSiteContext): string {

const params = this.urlTreeExtractContextParameters(tree);

const url = super.serialize(tree);

const serialized = this.urlIncludeContextParameters(url, params);

return serialized;

}

urlTreeExtractContextParameters(

urlTree: UrlTreeWithSiteContext

): ParamValuesMap {

return urlTree.siteContext ? urlTree.siteContext : {};

}

private urlIncludeContextParameters(url: string, params: ParamValuesMap) {

const contextRoutePart = this.urlEncodingParameters

.map((param) => {

return params[param]

? params[param]

: this.siteContextParams.getValue(param);

})

.join('/');

return contextRoutePart + url;

}

}

/\*

\* store

\*/

export function siteContextStoreConfigFactory(): StateConfig {

// if we want to reuse SITE\_CONTEXT\_FEATURE const in config, we have to use factory instead of plain object

const config: StateConfig = {

state: {

ssrTransfer: {

keys: { [SITE\_CONTEXT\_FEATURE]: StateTransferType.TRANSFER\_STATE },

},

},

};

return config;

}

@NgModule({

imports: [

CommonModule,

HttpClientModule,

StoreModule.forFeature(SITE\_CONTEXT\_FEATURE, reducerToken),

EffectsModule.forFeature(effects),

],

providers: [

provideDefaultConfigFactory(siteContextStoreConfigFactory),

reducerProvider,

],

})

export class SiteContextStoreModule {}

export const SITE\_CONTEXT\_FEATURE = 'siteContext';

export interface StateWithSiteContext {

[SITE\_CONTEXT\_FEATURE]: SiteContextState;

}

export interface SiteContextState {

languages: LanguagesState;

currencies: CurrenciesState;

baseSite: BaseSiteState;

}

export interface CurrencyEntities {

[isocode: string]: Currency;

}

export interface CurrenciesState {

entities: CurrencyEntities;

activeCurrency: string;

}

export interface LanguagesEntities {

[isocode: string]: Language;

}

export interface LanguagesState {

entities: LanguagesEntities;

activeLanguage: string;

}

export interface BaseSiteState {

activeSite: string;

details: BaseSite;

}

//action

export const LOAD\_BASE\_SITE = '[Site-context] Load BaseSite';

export const LOAD\_BASE\_SITE\_FAIL = '[Site-context] Load BaseSite Fail';

export const LOAD\_BASE\_SITE\_SUCCESS = '[Site-context] Load BaseSite Success';

export const SET\_ACTIVE\_BASE\_SITE = '[Site-context] Set Active BaseSite';

export const BASE\_SITE\_CHANGE = '[Site-context] BaseSite Change';

export class LoadBaseSite implements Action {

readonly type = LOAD\_BASE\_SITE;

}

export class LoadBaseSiteFail implements Action {

readonly type = LOAD\_BASE\_SITE\_FAIL;

constructor(public payload: any) {}

}

export class LoadBaseSiteSuccess implements Action {

readonly type = LOAD\_BASE\_SITE\_SUCCESS;

constructor(public payload: BaseSite) {}

}

export class SetActiveBaseSite implements Action {

readonly type = SET\_ACTIVE\_BASE\_SITE;

constructor(public payload: string) {}

}

export class BaseSiteChange implements Action {

readonly type = BASE\_SITE\_CHANGE;

}

// action types

export type BaseSiteAction = | LoadBaseSite | LoadBaseSiteFail | LoadBaseSiteSuccess

| SetActiveBaseSite | BaseSiteChange;

export const LOAD\_CURRENCIES = '[Site-context] Load Currencies';

export const LOAD\_CURRENCIES\_FAIL = '[Site-context] Load Currencies Fail';

export const LOAD\_CURRENCIES\_SUCCESS = '[Site-context] Load Currencies Success';

export const SET\_ACTIVE\_CURRENCY = '[Site-context] Set Active Currency';

export const CURRENCY\_CHANGE = '[Site-context] Currency Change';

export class LoadCurrencies implements Action {

readonly type = LOAD\_CURRENCIES;

}

export class LoadCurrenciesFail implements Action {

readonly type = LOAD\_CURRENCIES\_FAIL;

constructor(public payload: any) {}

}

export class LoadCurrenciesSuccess implements Action {

readonly type = LOAD\_CURRENCIES\_SUCCESS;

constructor(public payload: Currency[]) {}

}

export class SetActiveCurrency implements Action {

readonly type = SET\_ACTIVE\_CURRENCY;

constructor(public payload: string) {}

}

export class CurrencyChange implements Action {

readonly type = CURRENCY\_CHANGE;

}

// action types

export type CurrenciesAction = | LoadCurrencies

| LoadCurrenciesFail | LoadCurrenciesSuccess | SetActiveCurrency | CurrencyChange;

export const LOAD\_LANGUAGES = '[Site-context] Load Languages';

export const LOAD\_LANGUAGES\_FAIL = '[Site-context] Load Languages Fail';

export const LOAD\_LANGUAGES\_SUCCESS = '[Site-context] Load Languages Success';

export const SET\_ACTIVE\_LANGUAGE = '[Site-context] Set Active Language';

export const LANGUAGE\_CHANGE = '[Site-context] Language Change';

export class LoadLanguages implements Action {

readonly type = LOAD\_LANGUAGES;

}

export class LoadLanguagesFail implements Action {

readonly type = LOAD\_LANGUAGES\_FAIL;

constructor(public payload: any) {}

}

export class LoadLanguagesSuccess implements Action {

readonly type = LOAD\_LANGUAGES\_SUCCESS;

constructor(public payload: Language[]) {}

}

export class SetActiveLanguage implements Action {

readonly type = SET\_ACTIVE\_LANGUAGE;

constructor(public payload: string) {}

}

export class LanguageChange implements Action {

readonly type = LANGUAGE\_CHANGE;

}

// action types

export type LanguagesAction = | LoadLanguages

| LoadLanguagesFail | LoadLanguagesSuccess | SetActiveLanguage | LanguageChange;

export \* from './base-site.action';

export \* from './currencies.action';

export \* from './languages.action';

//reducers

export const initialState: BaseSiteState = {

details: {},

activeSite: '',

};

export function reducer(

state = initialState,

action: SiteContextActions.BaseSiteAction

): BaseSiteState {

switch (action.type) {

case SiteContextActions.LOAD\_BASE\_SITE\_SUCCESS: {

return { ...state, details: action.payload, };

}

case SiteContextActions.SET\_ACTIVE\_BASE\_SITE: {

return { ...state, activeSite: action.payload, };

}

}

return state;

}

export const initialState: CurrenciesState = {

entities: null,

activeCurrency: null,

};

export function reducer(

state = initialState,

action: SiteContextActions.CurrenciesAction

): CurrenciesState {

switch (action.type) {

case SiteContextActions.LOAD\_CURRENCIES\_SUCCESS: {

const currencies: Currency[] = action.payload;

const entities = currencies.reduce(

(currEntities: { [isocode: string]: Currency }, currency: Currency) => {

return {

...currEntities,

[currency.isocode]: currency,

};

},

{

...state.entities,

}

);

return { ...state, entities, };

}

case SiteContextActions.SET\_ACTIVE\_CURRENCY: {

const isocode: string = action.payload;

return { ...state, activeCurrency: isocode, };

}

}

return state;

}

export const initialState: LanguagesState = { entities: null, activeLanguage: null,};

export function reducer( state = initialState, action: SiteContextActions.LanguagesAction

): LanguagesState {

switch (action.type) {

case SiteContextActions.LOAD\_LANGUAGES\_SUCCESS: {

const languages: Language[] = action.payload;

const entities = languages.reduce(

(langEntities: { [isocode: string]: Language }, language: Language) => {

return {

...langEntities,

[language.isocode]: language,

};

},

{ ...state.entities, }

);

return { ...state, entities, };

}

case SiteContextActions.SET\_ACTIVE\_LANGUAGE: {

const isocode = action.payload;

return { ...state, activeLanguage: isocode, };

}

}

return state;

}

export function getReducers(): ActionReducerMap<SiteContextState> {

return {

languages: fromLanguages.reducer,

currencies: fromCurrencies.reducer,

baseSite: fromBaseSite.reducer,

};

}

export const reducerToken: InjectionToken<ActionReducerMap< SiteContextState>>

= new InjectionToken<ActionReducerMap<SiteContextState>>(

'SiteContextReducers'

);

export const reducerProvider: Provider = {

provide: reducerToken,

useFactory: getReducers,

};

//effects

@Injectable()

export class BaseSiteEffects {

@Effect()

loadBaseSite$: Observable<

SiteContextActions.LoadBaseSiteSuccess | SiteContextActions.LoadBaseSiteFail

> = this.actions$.pipe(

ofType(SiteContextActions.LOAD\_BASE\_SITE),

exhaustMap(() => {

return this.siteConnector.getBaseSite().pipe(

map((baseSite) => new SiteContextActions.LoadBaseSiteSuccess(baseSite)),

catchError((error) =>

of(

new SiteContextActions.LoadBaseSiteFail(

makeErrorSerializable(error)

) ) ) );

})

);

constructor(

private actions$: Actions,

private siteConnector: SiteConnector

) {}

}

@Injectable()

export class CurrenciesEffects {

@Effect()

loadCurrencies$: Observable<

| SiteContextActions.LoadCurrenciesSuccess | SiteContextActions.LoadCurrenciesFail

> = this.actions$.pipe(

ofType(SiteContextActions.LOAD\_CURRENCIES),

exhaustMap(() => {

return this.siteConnector.getCurrencies().pipe(

map(

(currencies) =>

new SiteContextActions.LoadCurrenciesSuccess(currencies)

),

catchError((error) =>

of(

new SiteContextActions.LoadCurrenciesFail(

makeErrorSerializable(error)

) ) ) );

})

);

@Effect()

activateCurrency$: Observable<

SiteContextActions.CurrencyChange

> = this.actions$.pipe(

ofType(SiteContextActions.SET\_ACTIVE\_CURRENCY),

tap((action: SiteContextActions.SetActiveCurrency) => {

if (this.winRef.sessionStorage) {

this.winRef.sessionStorage.setItem('currency', action.payload);

}

}),

map(() => new SiteContextActions.CurrencyChange())

);

constructor(

private actions$: Actions, private siteConnector: SiteConnector, private winRef: WindowRef

) {}

}

@Injectable()

export class LanguagesEffects {

@Effect()

loadLanguages$: Observable<

| SiteContextActions.LoadLanguagesSuccess | SiteContextActions.LoadLanguagesFail

> = this.actions$.pipe(

ofType(SiteContextActions.LOAD\_LANGUAGES),

exhaustMap(() => {

return this.siteConnector.getLanguages().pipe(

map(

(languages) => new SiteContextActions.LoadLanguagesSuccess(languages)

),

catchError((error) =>

of(

new SiteContextActions.LoadLanguagesFail(

makeErrorSerializable(error)

) ) ) );

})

);

@Effect()

activateLanguage$: Observable<

SiteContextActions.LanguageChange

> = this.actions$.pipe(

ofType(SiteContextActions.SET\_ACTIVE\_LANGUAGE),

tap((action: SiteContextActions.SetActiveLanguage) => {

if (this.winRef.sessionStorage) {

this.winRef.sessionStorage.setItem('language', action.payload);

}

}),

map(() => new SiteContextActions.LanguageChange())

);

constructor(

private actions$: Actions, private siteConnector: SiteConnector, private winRef: WindowRef

) {}

}

export const effects: any[] = [ LanguagesEffects, CurrenciesEffects, BaseSiteEffects,];

export const getActiveBaseSite: MemoizedSelector< StateWithSiteContext, string> = createSelector(

getSiteContextState,

(state: SiteContextState) =>

state && state.baseSite && state.baseSite.activeSite

);

export const getBaseSiteData: MemoizedSelector< StateWithSiteContext, BaseSite> = createSelector(

getSiteContextState,

(state: SiteContextState) => state && state.baseSite && state.baseSite.details

);

const currenciesEntitiesSelector = (state: CurrenciesState) => state.entities;

const activeCurrencySelector = (state: CurrenciesState) => state.activeCurrency;

export const getCurrenciesState: MemoizedSelector< StateWithSiteContext, CurrenciesState

> = createSelector(

getSiteContextState,

(state: SiteContextState) => state.currencies

);

export const getCurrenciesEntities: MemoizedSelector< StateWithSiteContext, CurrencyEntities

> = createSelector(getCurrenciesState, currenciesEntitiesSelector);

export const getActiveCurrency: MemoizedSelector< StateWithSiteContext, string

> = createSelector(getCurrenciesState, activeCurrencySelector);

export const getAllCurrencies: MemoizedSelector< StateWithSiteContext, Currency[]

> = createSelector(getCurrenciesEntities, (entities) => {

return entities

? Object.keys(entities).map((isocode) => entities[isocode])

: null;

});

const activeLanguageSelector = (state: LanguagesState) => state.activeLanguage;

const languagesEntitiesSelector = (state: LanguagesState) => state.entities;

export const getLanguagesState: MemoizedSelector< StateWithSiteContext, LanguagesState

> = createSelector(

getSiteContextState,

(state: SiteContextState) => state.languages

);

export const getLanguagesEntities: MemoizedSelector< StateWithSiteContext, LanguagesEntities

> = createSelector(getLanguagesState, languagesEntitiesSelector);

export const getActiveLanguage: MemoizedSelector< StateWithSiteContext, string

> = createSelector(getLanguagesState, activeLanguageSelector);

export const getAllLanguages: MemoizedSelector< StateWithSiteContext, Language[]

> = createSelector(getLanguagesEntities, (entities) => {

return entities

? Object.keys(entities).map((isocode) => entities[isocode])

: null;

});

export const getSiteContextState: MemoizedSelector< StateWithSiteContext, SiteContextState

> = createFeatureSelector<SiteContextState>(SITE\_CONTEXT\_FEATURE);

import \* as SiteContextSelectors from './site-context-group.selectors';

export { SiteContextSelectors };