Managing State in Angular with NGXS

Author : Gauthier PEEL

Angular

- https://github.com/ngxs/store
- https://ngxs.gitbook.io/ngxs/getting-started
- https://alligator.io/angular/ngxs/
- Comparison :
  https://ordina-jworks.github.io/angular/2018/10/08/angular-state-management-comparison.html

Angular
NGXS

Features	NGRX	NGXS	Akita	Plain RxJS
Async actions	Yes, through effects	**Yes**	No	No
(Memoized) selectors	Yes	**Yes**	Yes, as queries	No
Cross-state selectors	Yes	**Yes**	No	No
Offline persistence	3rd party package	**1st party package**	Main package	No
Snapshot selection without *first()*	No	**Yes**	Yes	No
Forms synchronization	3rd party packages	**1st party package**	Main package	No
Router synchronization	1st party package	**1st party package**	No	No
WebSocket	3rd party package	**1st party package**	No	No
Angular ErrorHandler	No	**Yes**	No	No
Meta Reducers	Yes	**Yes**	No	No
Lazy loading	Yes	**Yes**	Yes	No
Cancellation	No	**Yes**	No	No
Side effects	Yes	**Yes**	No	No
Web workers	No	**No**	Yes	No
Transactions	No	**No**	Yes	No

State Framework showdown

Boilerplate

SUMMARY

Boilerplate	Files generated	Total files (*)	Boilerplate code
NGRX	9	12	Heavy
NGXS	3	7	Medium
Akita	**4**	**6**	**Low**
Plain RxJS	0	6	Medium

State Framework showdown

Boilerplate for TODO-CRUD

- NgRx
  - crud.action.ts 53 lines
  - crud.effect.ts 35 lines
  - crud.reducer.ts (includes crud.state.ts) 52
  - crud.selector.ts 19 lines
  - TaskService.ts 65

224 lines

- NGXS
  - crud.actions.ts 42 lines
  - crud.reducer.ts 75
  - crud.state.ts 12
  - TaskService.ts 34

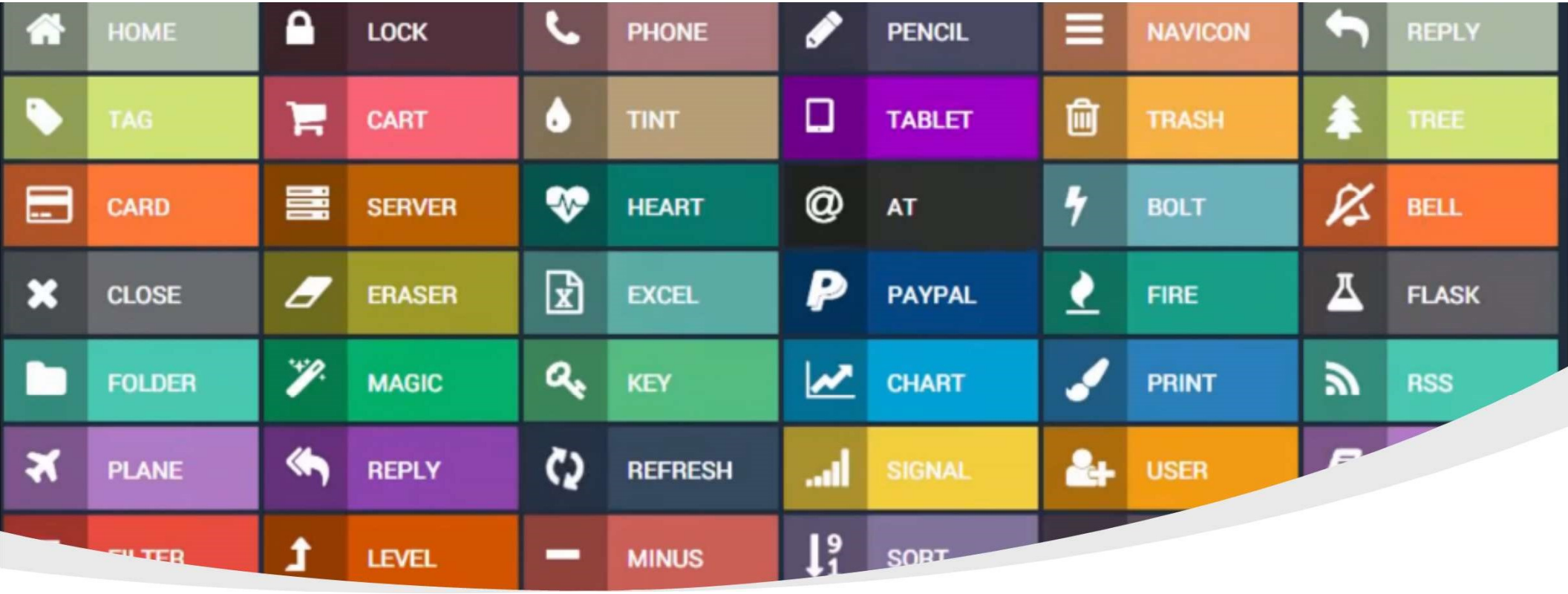
163 lines

- AKITA
  - crud.store.ts 43 lines
  - crud.query.ts 19
  - TaskService.ts 41

103 lines

- BehaviorSubject
  - TaskService.ts 50 lines

50 lines



Installation

Angular

NGXS setup

- February 2019 : V3.3.4
- May 2019 : V3.4.3
- Nov 2020 : V3.7.0

```
npm install @ngxs/store --save
```

```
npm install @ngxs/store@dev --save
npm install @ngxs/logger-plugin@dev --save
```

```
npm install @ngxs/devtools-plugin --save-dev
```


Angular

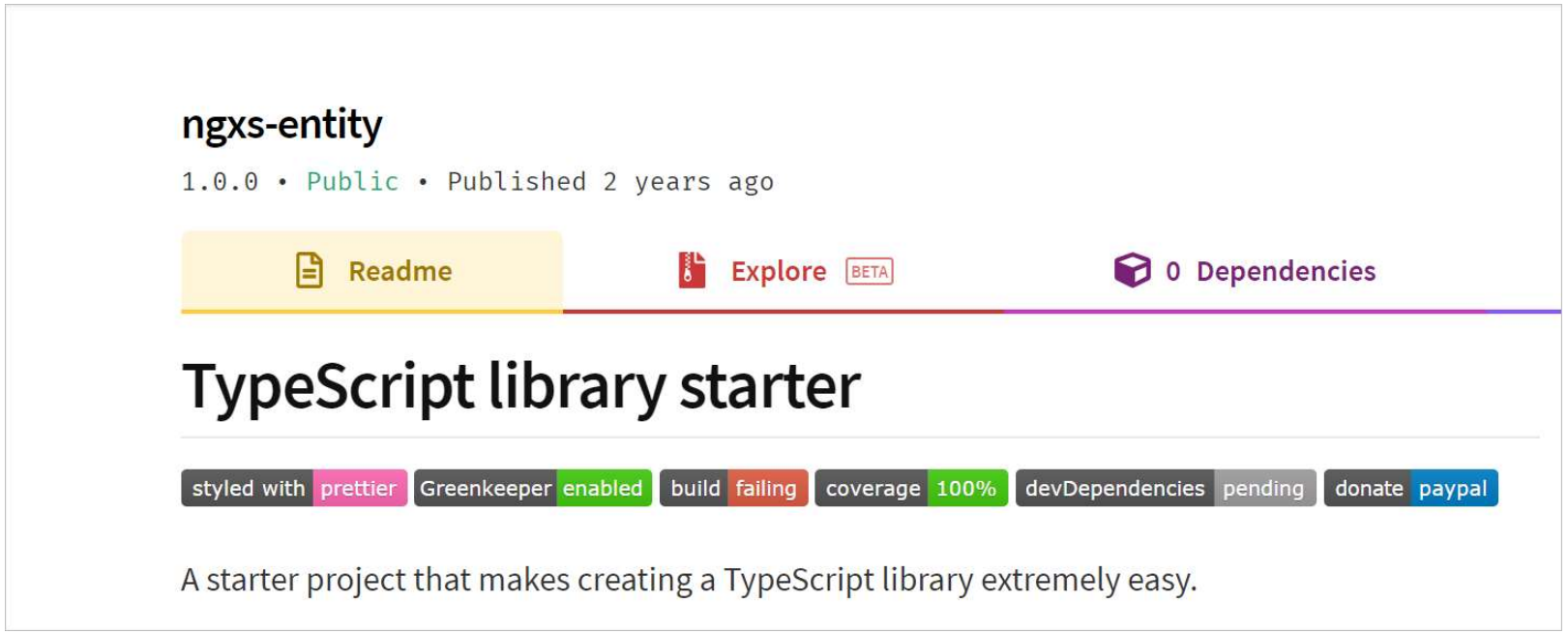
NGXS setup

- To get @SelectSnapshot()
  - https://github.com/ngxs-labs/select-snapshot

```
npm install @ngxs-labs/select-snapshot
```


Angular
NGXS setup

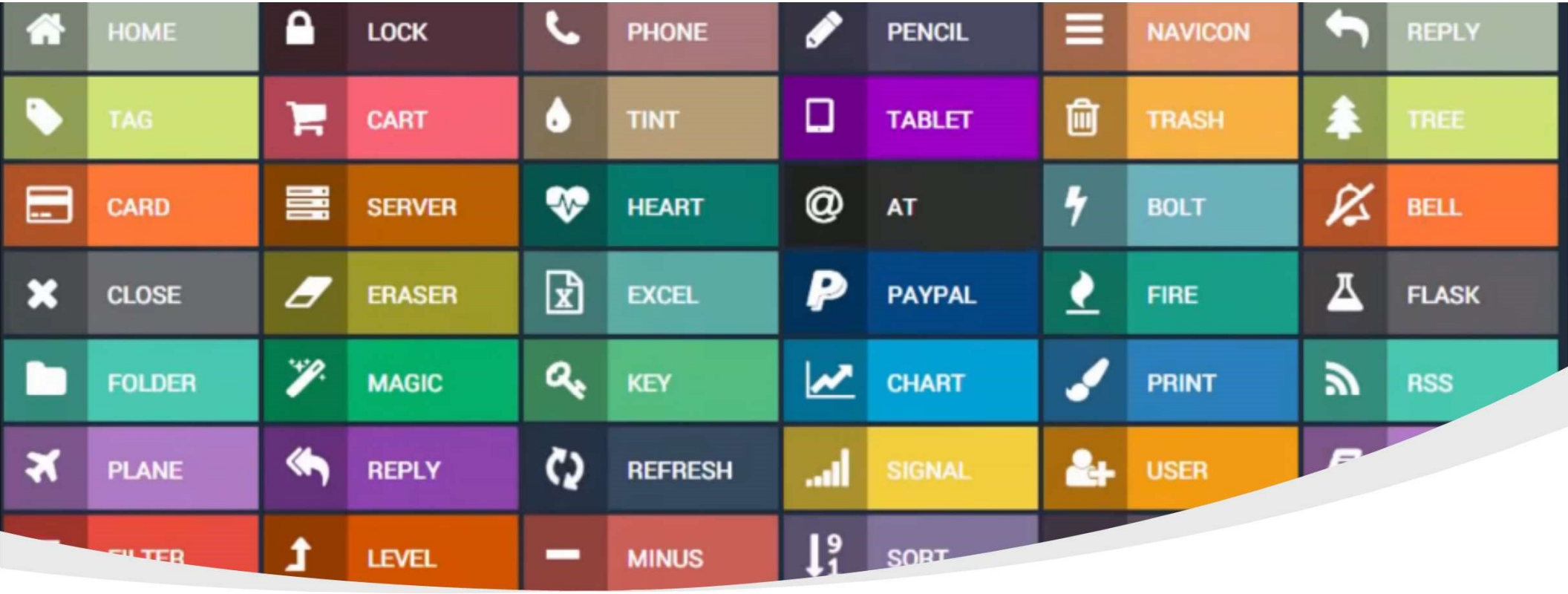
- Extension to test
  - https://www.npmjs.com/package/ngxs-entity



Angular

NGXS setup

- Switching to development mode will freeze your store using deep-freeze-strict module.
- NO install needed, its' all included, AND IT WORKS ! You can't modify a received state.

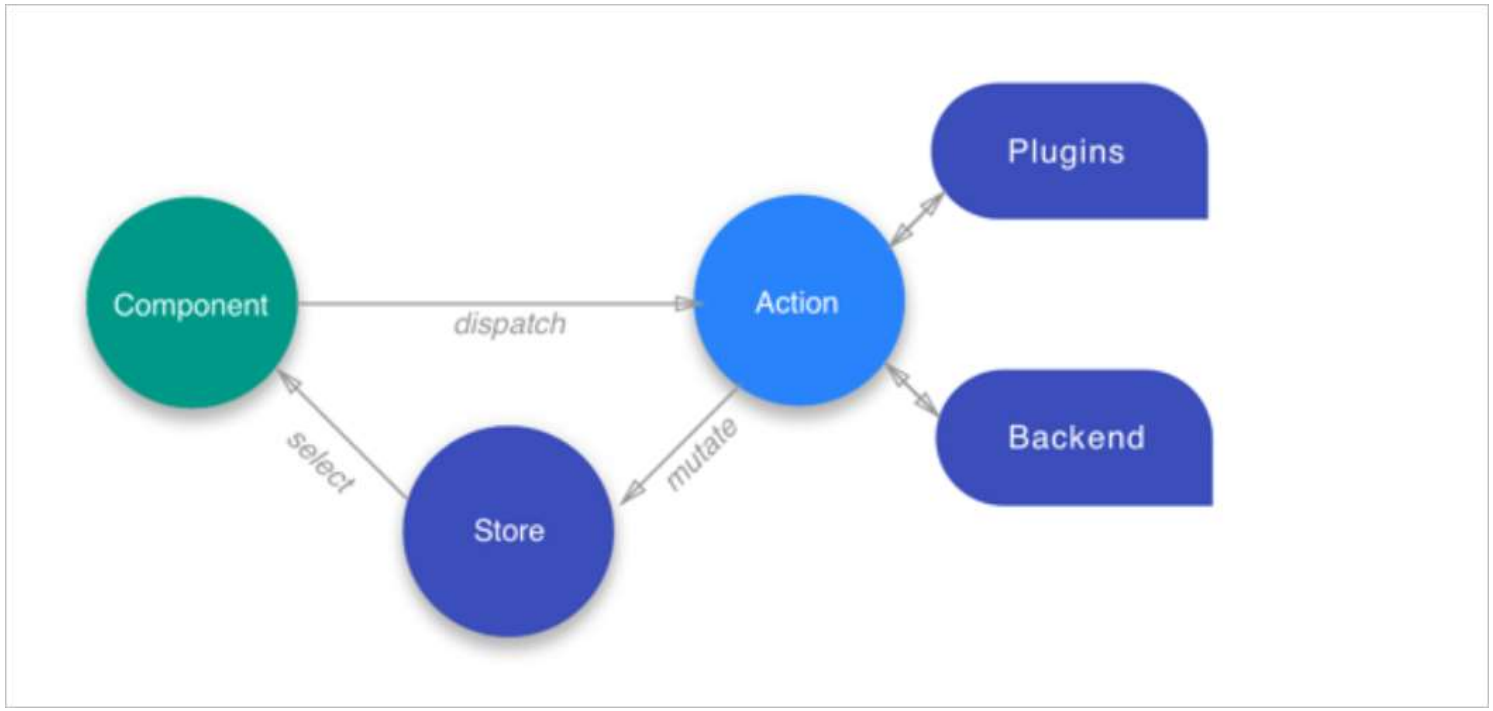


Architecture

Angular NGXS

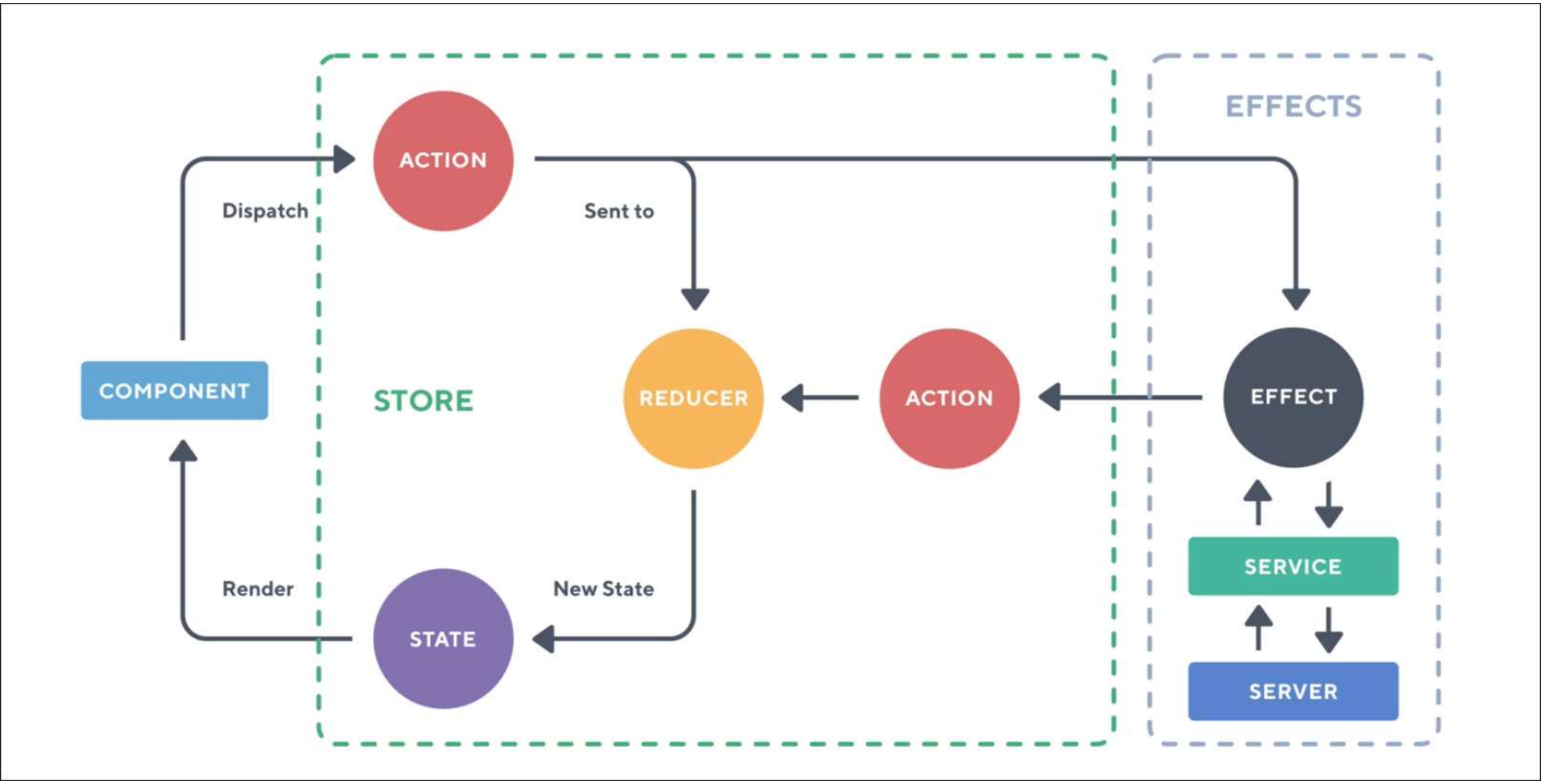
Architecture

- CQRS => Command Query Responsibility Segregation

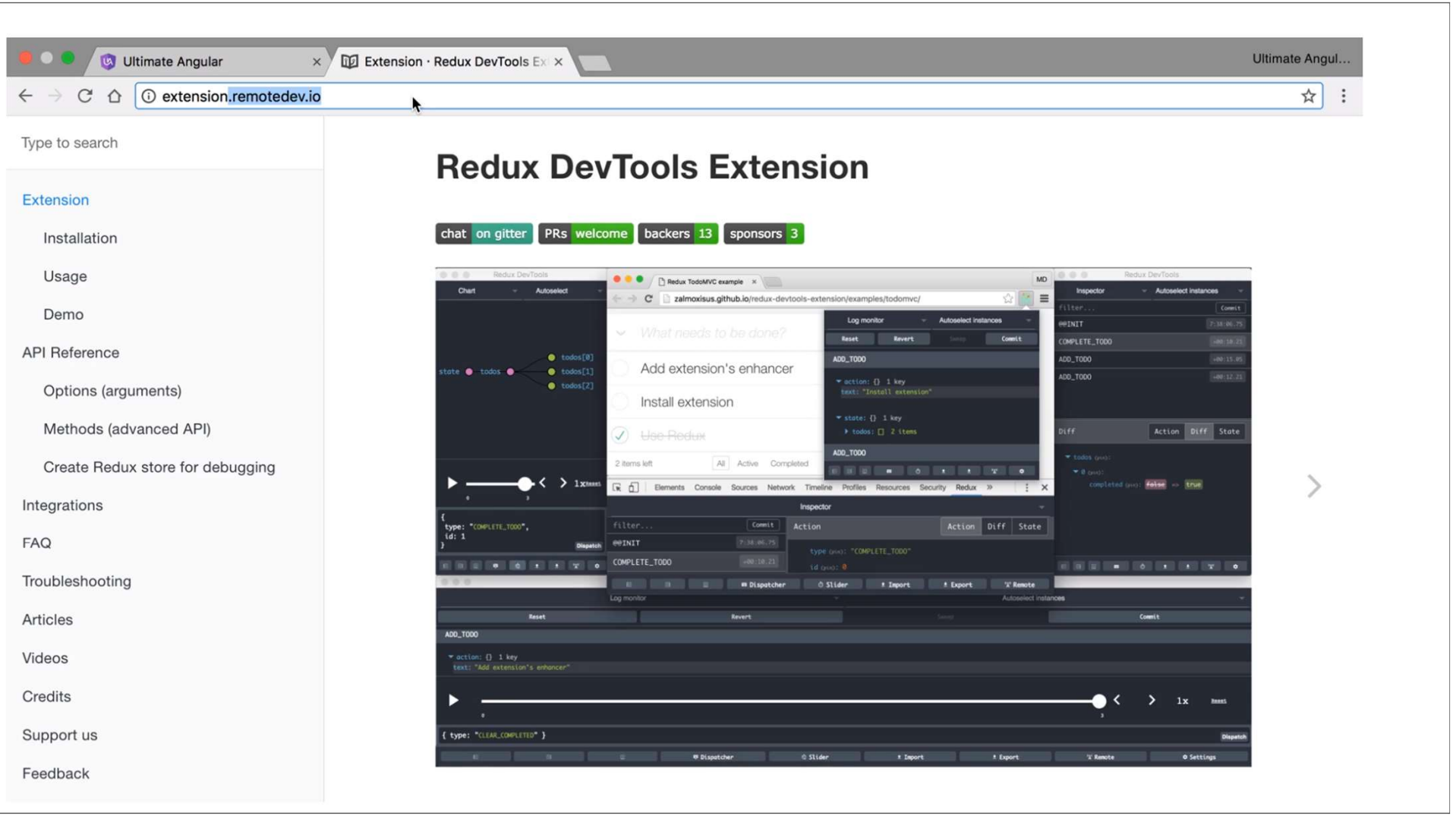


Angular NgRx

Comparison NgRx vs NGXS



Angular NGXS



Redux DevTools explained

https://codeburst.io/redux-devtools-for-dummies-74566c597d7

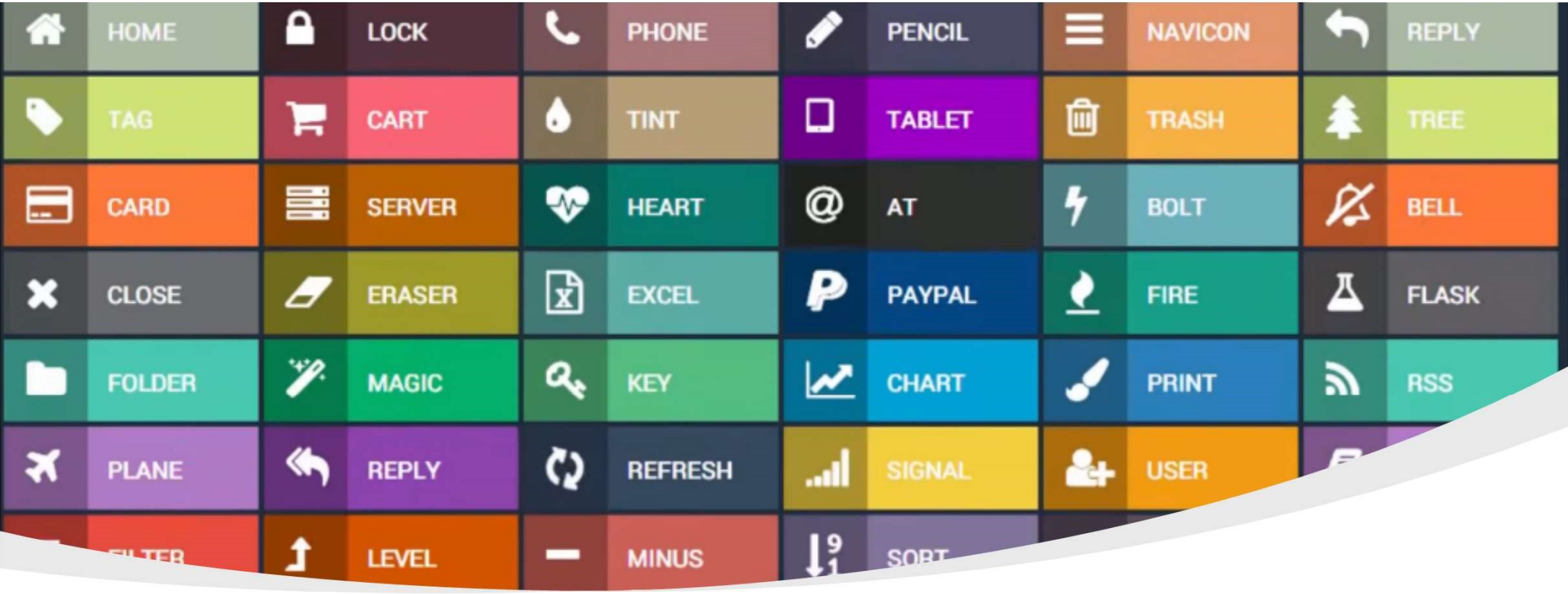
https://egghead.io/lessons/javascript-getting-started-with-redux-dev-tools

When you click on an individual action it displays two options (`Jump` and `Skip`). This is the first introduction to time traveling and changing the application view. `Jump` will take your application to the state of the app at the time this action fired. The skip will cross out the action and show you your app without that action.

DEVTOOLS IN NGXS

Although NGXS is also modeled after CQRS, it behaves a bit differently. It provides *@ngxs/devtools-plugin* for DevTools. It does, however, not support all functionalities. The latest actions can be viewed with their impact and resulting state. But while it's possible to jump to specific actions, it's not possible to skip actions or dispatch new ones using the DevTools. Implementing the tools is just as easy as with NGRX, importing the following line to the **AppModule**:

```
NgxsReduxDevtoolsPluginModule.forRoot()
```

Configuration as Module

Angular NGXS

Angular Router in the NGXS State

```
npm install @ngxs/router-plugin --save
```

- https://ngxs.gitbook.io/ngxs/plugins/router

```
1  import { NgxsModule } from '@ngxs/store';
2  import { NgxsRouterPluginModule } from '@ngxs/router-plugin';
3
4  @NgModule({
5    imports: [
6      NgxsModule.forRoot([]),
7      NgxsRouterPluginModule.forRoot()
8    ]
9  })
10 export class AppModule {}
```

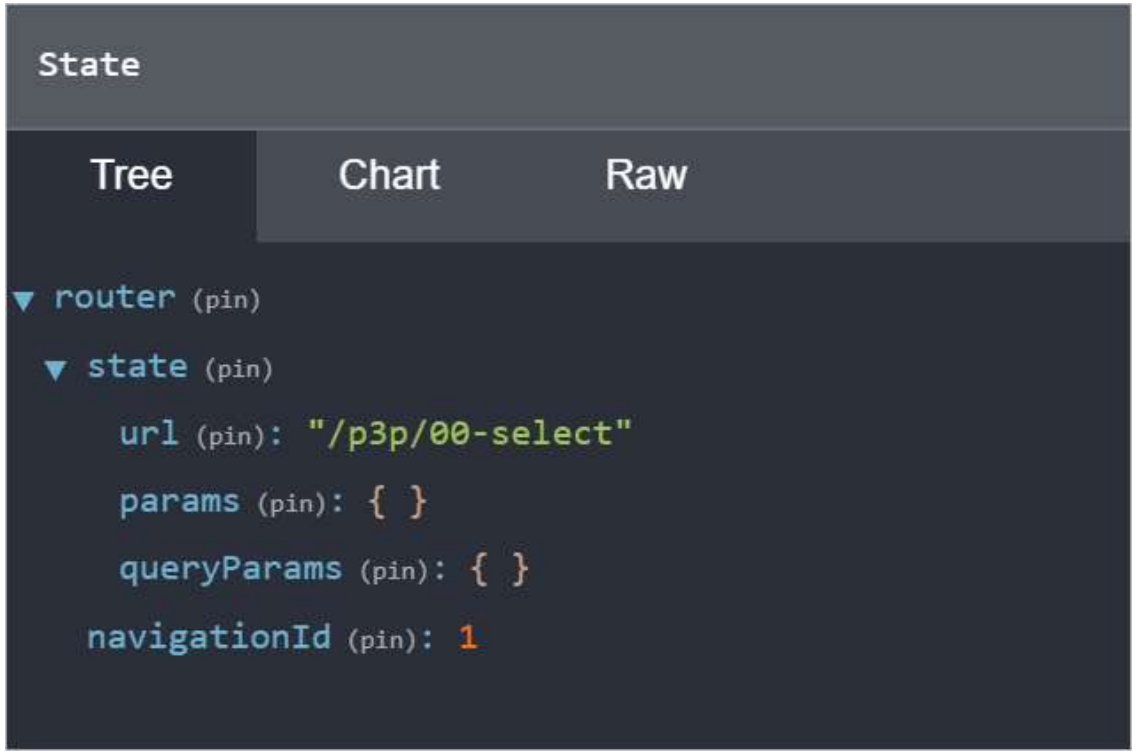

Angular NGXS : BETTER CONFIG

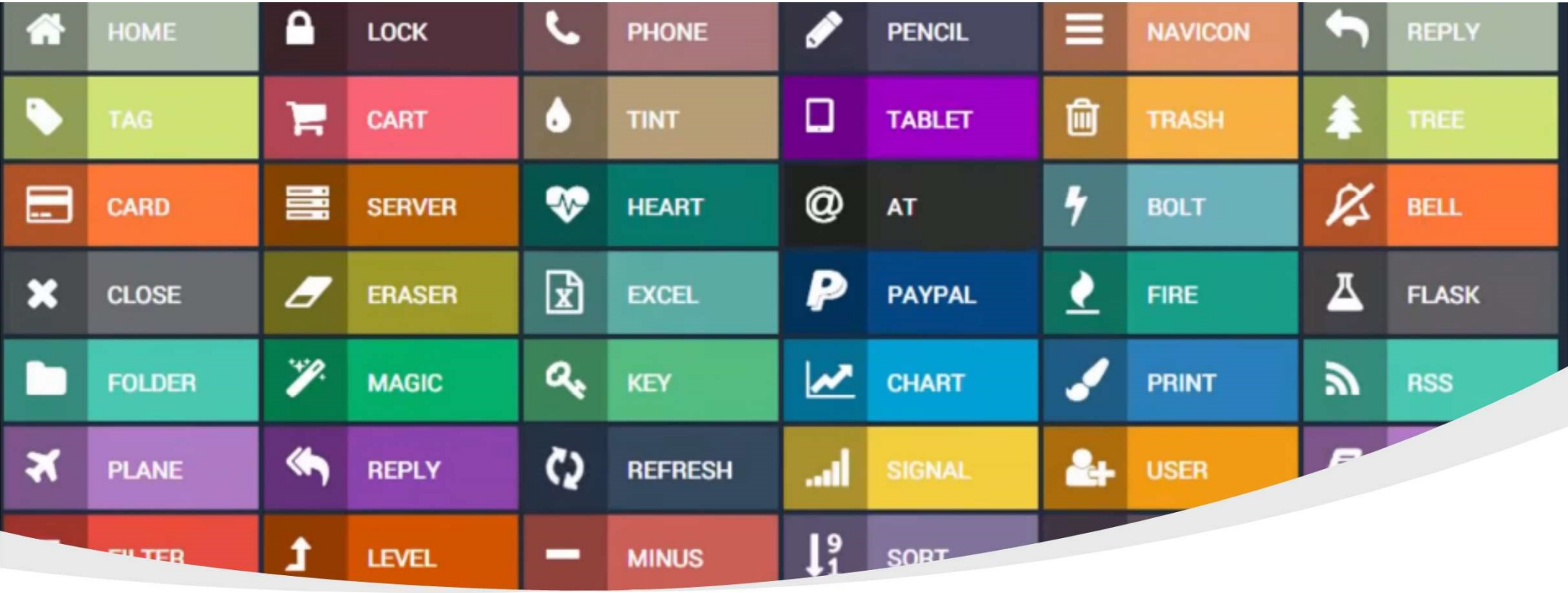
```
@NgModule({
  imports: [
    // tek one shot
    BrowserModule,
    // tek partagé
    SharedModule,

    // NGXS
    // NgxsModule.forRoot([], {developmentMode: false}),
    // NgxsModule.forRoot([], {developmentMode: true}),
    NgxsModule.forRoot([], {developmentMode: !environment.production}),
    // WARNING in prod mode SHOULD NOT BE INSTALLED

    // Hard-coded import of Module
    NgxsReduxDevtoolsPluginModule.forRoot(),
    NgxsLoggerPluginModule.forRoot(),

    // better config importing the Module ONLY if in DEV Mode
    environment.production ? [] :
      [NgxsReduxDevtoolsPluginModule.forRoot(), NgxsLoggerPluginModule.forRoot()],

    //
    // fonctionnel
    CrudModule
  ],
  declarations: [
    AppComponent,
```


Angular NGXS

Angular Router in the NGXS State

- Redux tab




HOME
LOCK
PHONE
PENCIL
NAVICON
REPLY
TAG
CART
TINT
TABLET
TRASH
TREE
CARD
SERVER
HEART
AT
BOLT
BELL
CLOSE
ERASER
EXCEL
PAYPAL
FIRE
FLASK
FOLDER
MAGIC
KEY
CHART
PRINT
RSS
PLANE
REPLY
REFRESH
SIGNAL
USER
LEVEL
MINUS

Coding ..

Angular NGXS

Getting Started

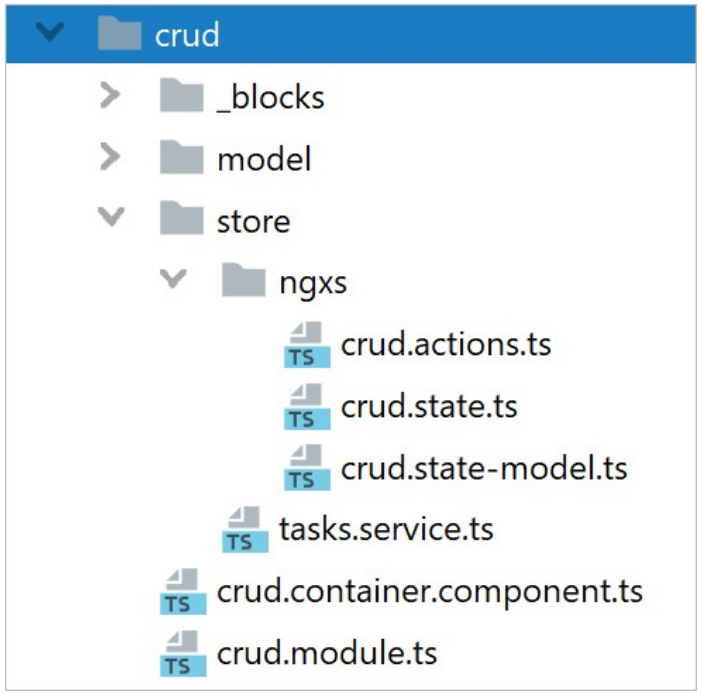
- Step1: Define you Model for the State *(file crud.state.ts)*

```
export interface CrudStateModel {
  tasks: Task[];
}

export const initialCrudStateModel: CrudStateModel = {
  tasks: [
    new Task('Aller boire des bières'),
    new Task('Dormir', true),
    new Task('Faire du sport (non je rigole)')]
};
```


Angular NGXS

Getting Started

- Step1 bis: folder structure



Angular NGXS

Getting Started

- The goal is to dispatch actions to the store:

```
addTask(task: Task) {
  return this.store.dispatch(new CreateTask(task));
}
```

- So we define specific classes to encapsulate the data to execute this action

```
export class CreateTask {
  static readonly type = '[Crud] Task Create';
  constructor(public task: Task) {}
}
```


Angular NGXS

Getting Started

- Step2: Create and organize Actions to make access easy.
  There is NO standard way, everyone tries to find a good way of doing it ….
  (file crud.action.ts)

```
enum CRUD_ACTION_NAMES {
  CREATE_TASK = '[Crud] Task Create',
  UPDATE_TASK = '[Crud] Task Update',
  DELETE_TASK = '[Crud] Task Delete',
  TOGGLE_ALL_TASK = '[Crud] Task Toggle All',
}

export class CreateTask {
  static readonly type = CRUD_ACTION_NAMES.CREATE_TASK;
  constructor(public task: Task) {}
}
```

```
export const CRUD_ACTIONS = {
  CreateTask,
  UpdateTask,
  DeleteTask,
  ToggleAllTask,
};
```


Angular NGXS

Getting Started

- Step3: the Reducer is where the job of modifying the state is done. NGXS calls the class <Something>State

```
@State<CrudStateModel>({
  name: 'crud',
  defaults: initialCrudStateModel
})
export class CrudState {

  @Action(CRUD_ACTIONS.CreateTask)
  createTask(ctx: StateContext<CrudStateModel>, action: CRUD_ACTIONS.CreateTask) {
    Plog.state('ACTION!!!! CREATE_TASK', action);
    const task: Task = action.task;
    const crudState: CrudStateModel = ctx.getState();
    ctx.setState({...crudState, tasks: [...crudState.tasks, task]});
  }
```


Angular NGXS

Getting Started

- Step5: Module declaration

```
@NgModule({
  imports: [
    SharedModule,
    // NGXS
    NgxsModule.forFeature([CrudState]),
  ],
  declarations: [...],
  exports: [...],
  providers: [...]
})
export class CrudModule {
```


Angular NGXS

Getting Started

- Step: Dispatching Actions

```
import * as CRUD_ACTIONS from './crud/crud.actions';
import {CrudState} from './crud/crud.state';
import {Store} from '@ngxs/store';

@Injectable()
export class TasksService {

  constructor(private store: Store) {
  }

  addTask(task: Task): Observable<any> {
    return this.store.dispatch(new CRUD_ACTIONS.CreateTask(task));
  }
```


Angular NGXS

Getting Started

- Step: Getting data by directly receiving the FULL State

```
@Component({selector: 'todo-crud-container'...})
export class CrudContainerComponent {

  @Select(state => state.crud.tasks) tasks$: Observable<Task[]>;
```


Angular NGXS

Getting Started

- Step: Getting data by directly receiving the FULL root State

```
import {Store} from '@ngxs/store';

@Component({selector: 'todo-crud-container'...})
export class CrudContainerComponent {

  tasks$: Observable<Task[]>;

  constructor(store: Store) {
    this.tasks$ = store.select(state => state.crud.tasks);
  }
```


Angular NGXS

Getting Started

- Step: Getting data by directly receiving the FULL root State using @Select()

```
@Component({selector: 'todo-crud-container'...})
export class CrudContainerComponent {

  @Select(state => state.crud.tasks) tasks$: Observable<Task[]>;
```

- For memo, previous page was:

```
constructor(store: Store) {
  this.tasks$ = store.select(state => state.crud.tasks);
}
```


Angular NGXS

Getting Started

- Step: Getting data with factorized Selectors

```
@State<CrudStateModel>({
  name: 'crud',
  defaults: initialCrudStateModel
})
export class CrudState {

  constructor(private store: Store) {}

  @Selector()
  static tasks(crudStateModel: CrudStateModel) {
    return crudStateModel.tasks;
  }
```

```
export class CrudContainerComponent {

  @Select(CrudState.tasks) tasks$: Observable<Task[]>;
```

Angular NGXS

Getting Started

- Step Summary

```
@Component({selector: 'todo-crud-container'...})
export class CrudContainerComponent {

  @Select(state => state.crud.tasks) tasks$: Observable<Task[]>;
```

```
constructor(store: Store) {
  this.tasks$ = store.select(state => state.crud.tasks);
}
```

```
export class CrudContainerComponent {

  @Select(CrudReducer.tasks) tasks$: Observable<Task[]>;
```

memoized

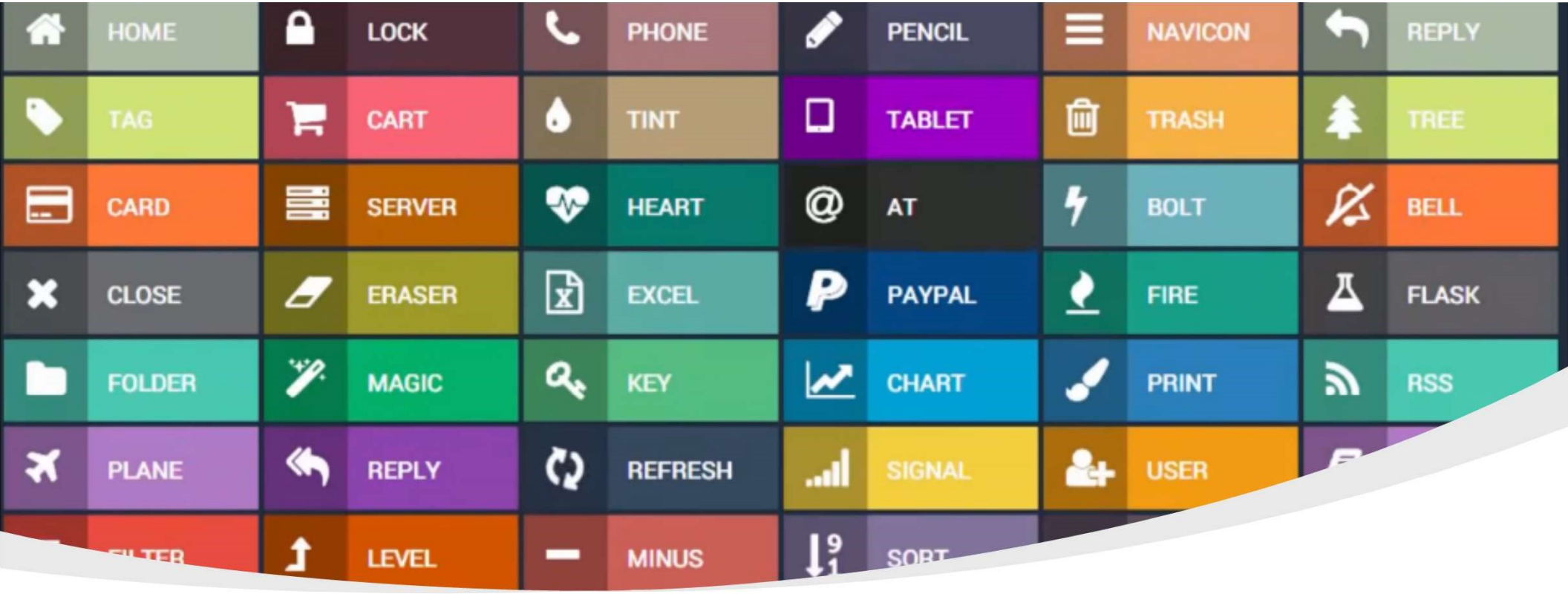
Angular NGXS

Getting Started

- Step6 details : select, selectOnce, selectSnapshot

```
@Injectable()
export class TasksService {

  constructor(private store: Store) {}

  getTask$(): Observable<Task[]> {
    return this.store.select(state => state.crud.tasks);
  }
```

```
select<T>(selector: (state: any, ...states: any[]) => T)        Observable<T>
selectOnce<T>(selector: (state: any, ...states: any[]) => T)    Observable<T>
selectSnapshot<T>(selector: (state: any, ...states: any[]) => T)          T
Press Enter to insert, Tab to replace
```

Event of Action Stream

Angular NGXS

Actions stream

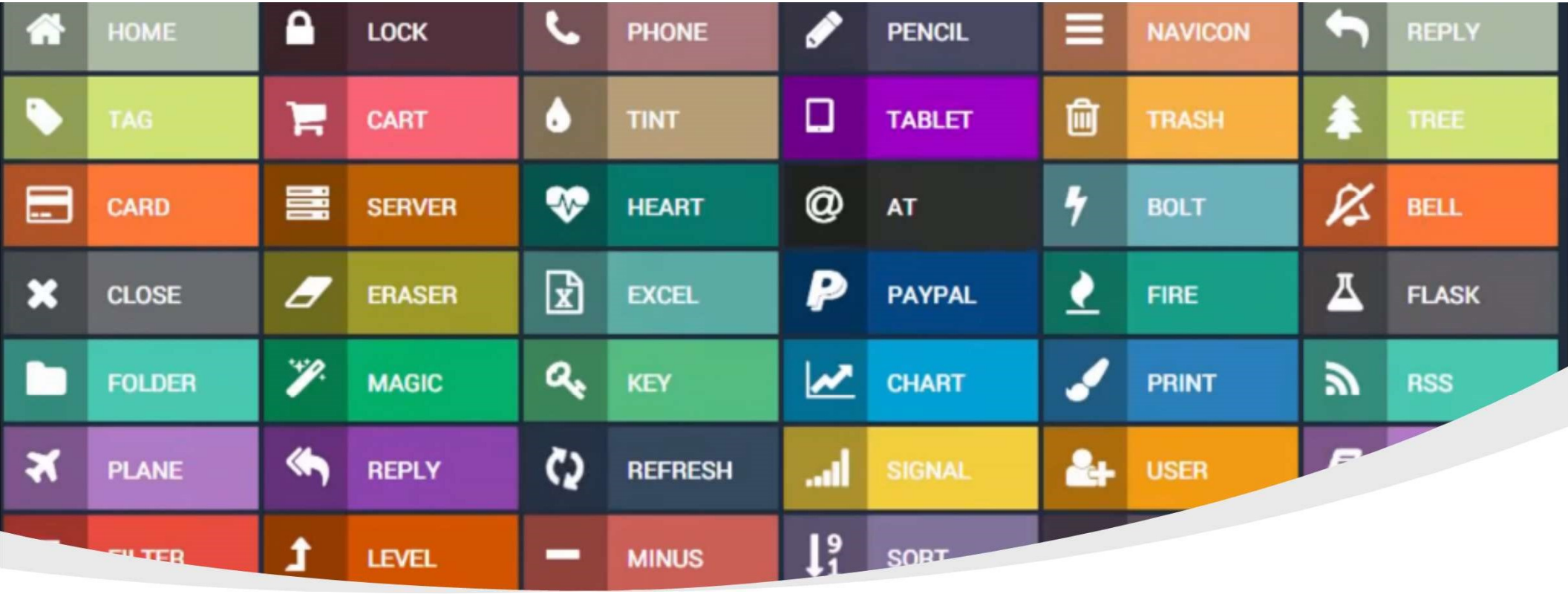
```
import { Actions, ofActionSuccessful } from '@ngxs/store';
import { RouterDataResolved } from '@ngxs/router-plugin';

import { Subject } from 'rxjs';
import { takeUntil } from 'rxjs/operators';

@Component({ ... })
export class AppComponent {

  private destroy$ = new Subject<void>();

  constructor(actions$: Actions) {
    actions$.pipe(
      ofActionSuccessful(RouterDataResolved),
      takeUntil(this.destroy$)
    ).subscribe((action: RouterDataResolved) => {
      console.log(action.routerState.root.firstChild.data);
    });
  }

  ngOnDestroy(): void {
    this.destroy$.next();
    this.destroy$.complete();
  }

}
```


Angular NGXS

Actions stream

```
@Component({ ... })
export class CartComponent {
  constructor(private actions$: Actions) {}

  ngOnInit() {
    this.actions$.pipe(ofActionSuccessful(CartDelete)).subscribe(() => alert('Item
  }
}
```

```
import { Actions, ofActionDispatched } from '@ngxs/store';

@Injectable({ providedIn: 'root' })
export class RouteHandler {
  constructor(private router: Router, private actions$: Actions) {
    this.actions$
      .pipe(ofActionDispatched(RouteNavigate))
      .subscribe(({ payload }) => this.router.navigate([payload]));
  }
}
```

How to code good state reducers

Writing Good Reducers

{ ...state, loading: true }

- And Anti-Pattern to code good spread/rest/destructuring code

  https://redux.js.org/recipes/structuring-reducers/immutable-update-patterns

Common Mistake #1: New variables that point to the same objects

Defining a new variable does *not* create a new actual object - it only creates another reference to the same object. An example of this error would be:

```
function updateNestedState(state, action) {
  const nestedState = state.nestedState;
  // ERROR: this directly modifies the existing object reference - don't do this!
  nestedState.nestedField = action.data;

  return {
    ...state,
    nestedState
  };
}
```


Writing Good Reducers

{ ...state, loading: true }

- NGXS introduced State Operators to manipulate the State
- https://ngxs.gitbook.io/ngxs/advanced/operators
- https://medium.com/ngxs/ngxs-state-operators-8b339641b220

NGXS State Operators

Better reducer code

- NGXS State Operators example

```
// Initial Version
const state = ctx.getState();
ctx.setState({ ...state, loading: true });

// Refactoring 1 - use the operator syntax
ctx.setState((state) => ({ ...state, loading: true }));

// Refactoring 2 - extract the state operator code into a function:
function setLoading(newValue: boolean) {
  return (state) => ({ ...state, loading: newValue });
}
// The code would now be expressed as:
ctx.setState(setLoading(true));

// Refactoring 3 - the extracted function could then be used to represent a higher level concept
function startLoading() { return setLoading(true); }
// The code would now be expressed as:
ctx.setState(startLoading());
```


NGXS State Operators

Better reducer code

- NGXS provided State Operators are :
  patch,
- For arrays:
  updateItem, removeItem, insertItem, append
- Others:
  compose, iif

addCity

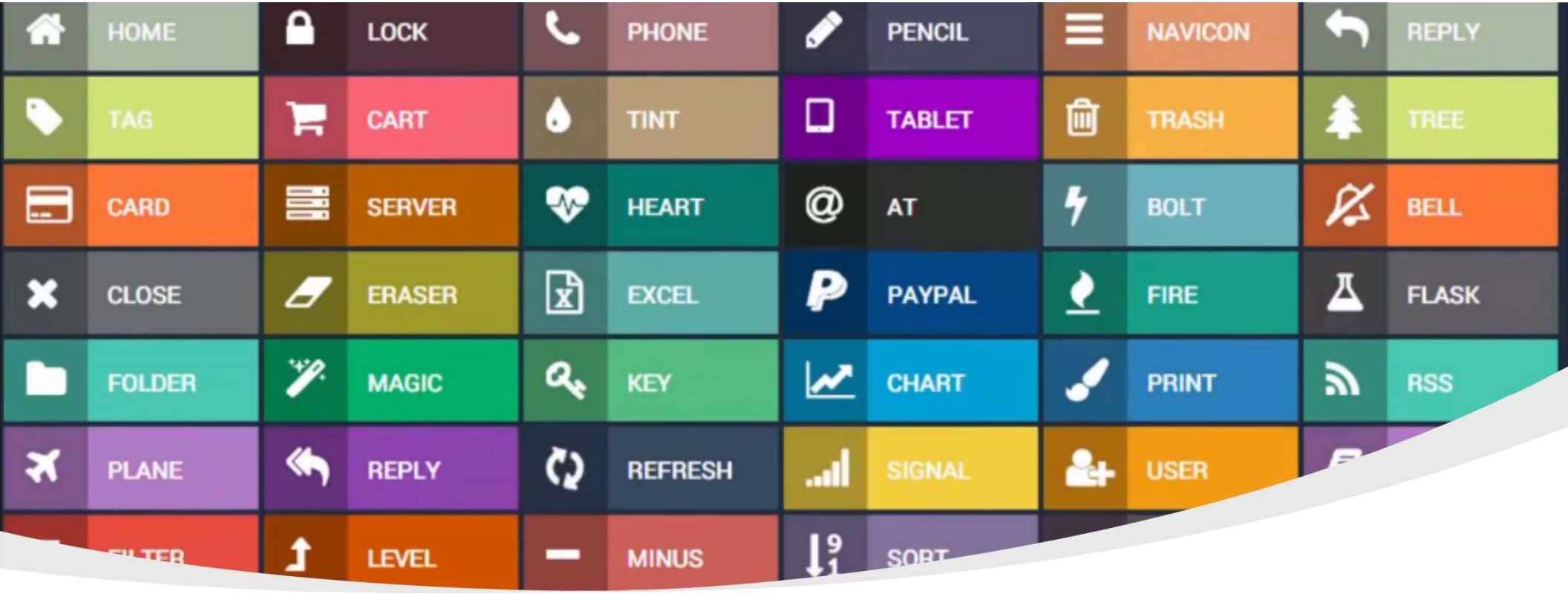
```
// Before
const city = payload.city;
const state = ctx.getState();
ctx.setState({
  ...state,
  entities: { ...state.entities, [city.id]: city },
  ids: [ ...state.ids, city.id ]
});

// Refactoring 1 - leverage the provided state operators
const city = payload.city;
ctx.setState( patch<CitiesStateModel>{
  entities: patch( { [city.id]: city } ),
  ids: append( [ city.id ] )
} );

// Refactoring 2 - extract a general purpose addEntity State Operator
interface EntitiesStateModel<T extends { id: number }> {
  entities: { [id: number]: T };
  ids: number[];
}
function addEntity<T extends { id: number }>(entity: T) {
  return patch<EntitiesStateModel<T>>({
    entities: patch( { [entity.id]: entity } ),
    ids: append( [ entity.id ] )
  });
}
// Our code would now be expressed as:
const city = payload.city;
ctx.setState( addEntity(city) );
```

NG
Bett

```
// Initial Version
const state = ctx.getState();
ctx.setState({ ...state, loading: true });

// Refactoring 1 - use the operator syntax
ctx.setState((state) => ({ ...state, loading: true }));

// Refactoring 2 - extract the state operator code into a function:
function setLoading(newValue: boolean) {
  return (state) => ({ ...state, loading: newValue });
}
// The code would now be expressed as:
ctx.setState(setLoading(true));

// Refactoring 3 - the extracted function could then be used to represent a higher level concept
function startLoading() { return setLoading(true); }
// The code would now be expressed as:
ctx.setState(startLoading());
```

- NGXS State Operators new options : patchState. Meaning only the properties appearing in the Partial<T> are to be updated.
  Here <T> is WizardState = { processStarted: …, step: …, stepInfos: …}

```
@Action([WIZARD_ACTIONS.StartProcess])
startProcess(ctx: StateContext<WizardStateModel>, action: WIZARD_ACTIONS.StartProcess) {
  ctx.patchState({processStarted: true});
}
```

Deciding what to put in the Store

Angular NGXS

Which data in the Store ?

- 2 main strategies
  - Strictly Data in the Store
    - And strictly data handling Actions.
  - Almost everything in the Store
    - Typically splitting ACTIONs in 3 parts:
      GET_HERO_LOADING
      GET_HERO_ERROR
      GET_HERO_SUCCESS
    - Adding more transient information like
      - loading boolean when an HTTP request is ongoing,
      - UI local state
      - Form transient state
      - Navigation Action

NGXS

Storing only Stable DATA

- Storing Only Stable Data
  - Less work
  - Entity State
  - Routing State
  - Some UI State
  - Action : CREATE / UPDATE / DELETE / FIND
  - You should consider that the HTTP methods calls are outside the State/Action process. They could have been executed before the @Action() method, but for simpler code, coding inside an @Action() method is much cleaner thanks to NGXS concepts.
  - No Routing Actions (no Spring-Flow like Action/Nav configuration)

Angular NGXS

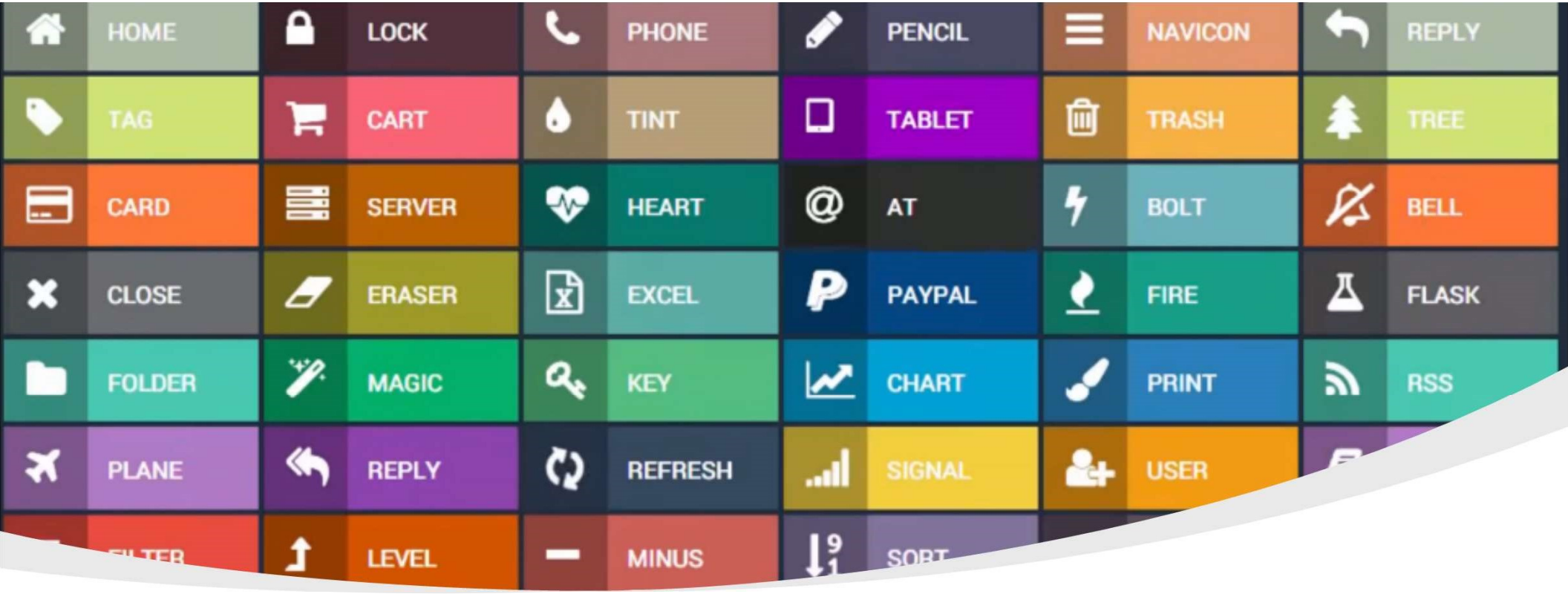
Storing a wide range of information in the Store

- Almost everything in the Store
  - Data
  - UI State
  - UI local state

  - Transient information like
    - loading State when an HTTP request is ongoing,

  - ACTIONs are like :
    GET_HERO (meaning we start an HTTP Request)
    GET_HERO_ERROR
    GET_HERO_SUCCESS

  - (Optional) All Navigation in Action (Spring-Flow like)



Only Stable Data in Store

NGXS

Storing only Stable DATA

- Coding async work inside an @Action()

```
@Action(CRUD_ACTIONS.UpdateTask)
updateTask(ctx: StateContext<CrudStateModel>, action: CRUD_ACTIONS.UpdateTask) {
  Plog.state('ACTION!!!! UPDATE_TASK', action);
  return this.taskHttpService.update(action.task)
    .pipe(
      mergeMap(t => {
          Plog.state('CallBack from HTTP Update, cascading refresh FindAll');
          return ctx.dispatch(CRUD_ACTIONS.FindAllTask); // notice the return
          // because we need an Observable in the stream
        }
      )
    );
}
```

- The caller gets feedback on error and on success of the last Action.

NGXS Storing only Stable DATA

- Coding async work inside an @Action()

```
@Action(CRUD_ACTIONS.UpdateTask)
updateTask(ctx: StateContext<CrudStateModel>, action: CRUD_ACTIONS.UpdateTask) {
  Plog.state('ACTION!!!! UPDATE_TASK', action);
  return this.taskHttpService.update(action.task)
    .pipe(
      mergeMap(t => {
          Plog.state('CallBack from HTTP Update, cascading refresh FindAll');
          return ctx.dispatch(CRUD_ACTIONS.FindAllTask); // notice the return
          // because we need an Observable in the stream
        }
      )
    );
}
```

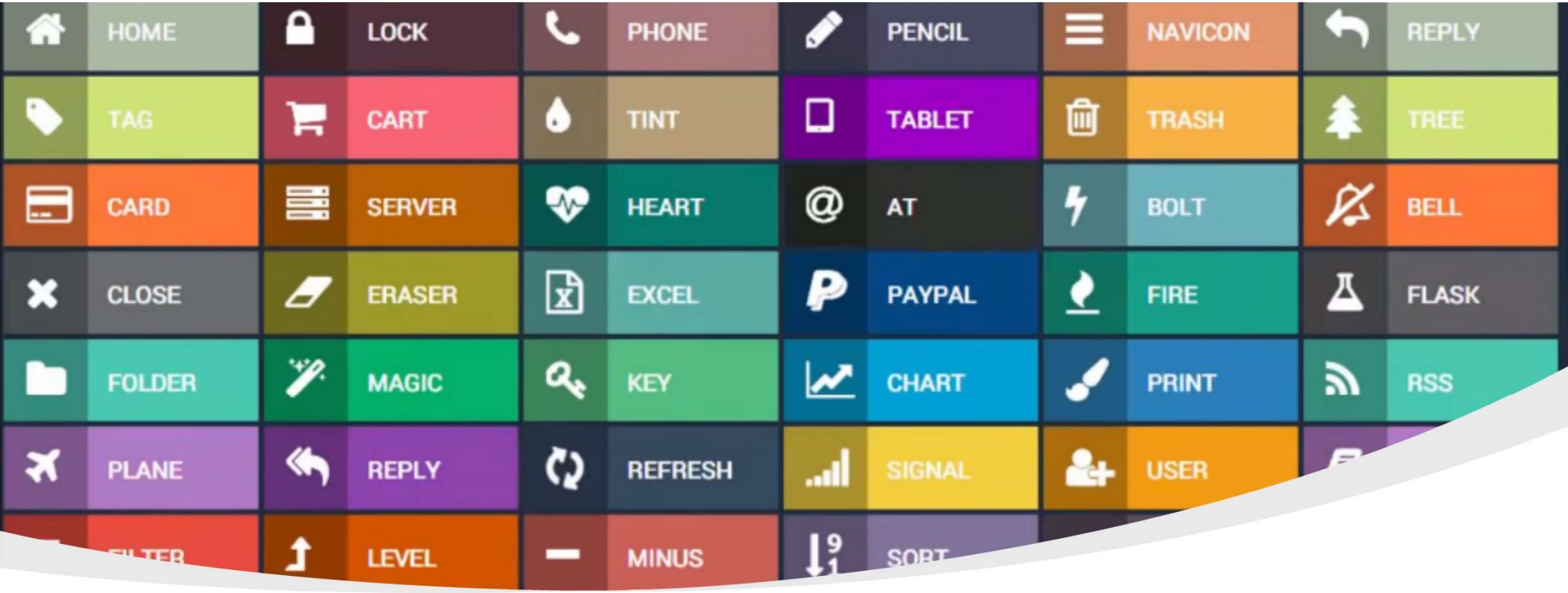
- The caller gets feedback on error and on success of the last Action.

```
updateTask(taskToBeUpdated: Task) {
  return this.store.dispatch(new CRUD_ACTIONS.UpdateTask(taskToBeUpdated));
}
```

tasks.service.ts

```
onUpdate(task: Task) {
  this.tasksService.updateTask(task)
    .subscribe(
      () => Plog.debug('RECEIVING ASYNC event on SUCCESS'),
      e => Plog.state('RECEIVING ASYNC event on ERROR ', e)
      // I WILL receive any Error coming from Async Actions :
      // first or re-dispatched action if it is the focused data in the Observable xxxMap

    );
}
```

crud-container.component.ts



FULL Data ACTION (HTTP, SUCCESS, ERROR) in Store

NGXS

FULL Data in Store

- Action CREATE_TASK could be split into 3 @Action(s)
  - CREATE_TASK => kicks in an HTTP Request
  - => on success dispatch CREATE_TASK_SUCCESS
  - => on error dispatch CREATE_TASK_ERROR

```
CREATE_TASK = '[Crud] Task Create Http',
CREATE_TASK_SUCCESS = '[Crud] Task Create Http SUCCESS',
CREATE_TASK_ERROR = '[Crud] Task Create Http ERROR',
```

  - CREATE_TASK_SUCCESS
    - Modifies the State to add the createdTask into the local Store when the server says ok.
    - We could add SnackBack calls here (with still a new dispatched Action, or by code)
  - CREATE_TASK_ERROR
    - We could add SnackBack calls here (with still a new dispatched Action, or by code)

NGXS

FULL Data in Store

- Code example

```
@Action(CRUD_ACTIONS.CreateTask)
createTask(ctx: StateContext<CrudStateModel>, action: CRUD_ACTIONS.CreateTask) {
  Plog.state('ACTION!!!! starting  CREATE_TASK', action);
  return this.taskHttpService.add(action.taskToCreated)
    .pipe(
      mergeMap(t => {
          Plog.state('CallBack from HTTP Create, cascading refresh to CreateTaskSuccess');
          return ctx.dispatch(new CRUD_ACTIONS.CreateTaskSuccess(t));
        }
      ),
      catchError(err => ctx.dispatch(new CRUD_ACTIONS.CreateTaskError(err)))
    );
}

@Action(CRUD_ACTIONS.CreateTaskSuccess)
createTaskSuccess(ctx: StateContext<CrudStateModel>, action: CRUD_ACTIONS.CreateTaskSuccess) {
  Plog.state('ACTION!!!! starting  CREATE_TASK_SUCCESS', action);
  ctx.setState(patch({tasks: [...ctx.getState().tasks, action.createdTask]}));
}

@Action(CRUD_ACTIONS.CreateTaskError)
createTaskError(ctx: StateContext<CrudStateModel>, action: CRUD_ACTIONS.CreateTaskError) {
  Plog.state('ACTION!!!! starting  CREATE_TASK_ERROR', action);
}
```

Here →

Here →

- We could insert UI code to show a SnackBar …

NGXS

More Data in Store, too much ?

- Where to decide to show a SnackBar ?
  - State-Strict Approach
    - In the @Component Callback

```
onCreate(task: Task) {
  this.tasksService.updateTask(task)
    .subscribe(
      () => Plog.state('RECEIVING ASYNC event on SUCCESS'),
      e => Plog.state('RECEIVING ASYNC event on ERROR ', e)
    );
}
```

crud-container.component.ts

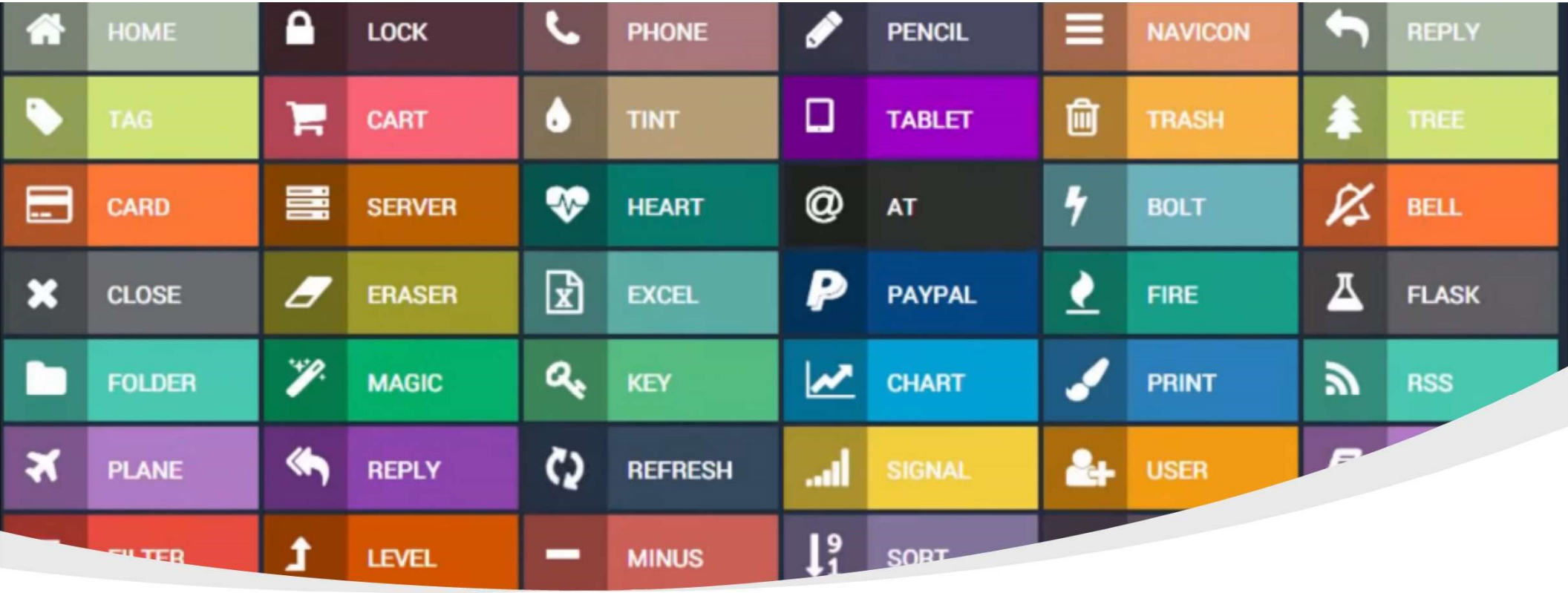
  - More-Data Approach
    - In the @Action() SUCCESS and ERROR

```
@Action(CRUD_ACTIONS.CreateTaskSuccess)
createTaskSuccess(ctx: StateContext<CrudStateModel>, action: CRUD_ACTIONS.CreateTaskSuccess) {
  Plog.state('ACTION!!!! starting  CREATE_TASK_SUCCESS', action);
  ctx.setState(patch({tasks: [...ctx.getState().tasks, action.createdTask]}));
}

@Action(CRUD_ACTIONS.CreateTaskError)
createTaskError(ctx: StateContext<CrudStateModel>, action: CRUD_ACTIONS.CreateTaskError) {
  Plog.state('ACTION!!!! starting  CREATE_TASK_ERROR', action);
}
```

NGXS

More Data in Store, too much ?

- Where to decide to show a SnackBar ?
  - State-Strict Approach
    - In the @Component Callbacks
  - Then you decide if you Navigate somewhere depending on Success/Error. That is typical UI decision.
  - More-Data Approach
    - In the @Action() SUCCESS and ERROR
    - Then you could add Navigation decision if you want.
      With that strategy all app flow is in the Store.



Inner knowledge:

in the end NGXS Actions are strings !

NGXS events

Action class vs Action String Type

- NGXS seems to introduce good tying to identify the Action

```
export interface ZooStateModel {
  animals: string[];
}

export class AddAnimal {
  static readonly type = '[Zoo] Add Animal';
  constructor(public name: string) {}
}

@State<ZooStateModel>({
  name: 'zoo',
  defaults: {
    animals: []
  }
})
export class ZooState {
  @Action(AddAnimal)
  addAnimals(ctx: StateContext<ZooStateModel>, action: AddAnimal) {
    const state = ctx.getState();
    ctx.setState({
      ...state,
      animals: [...state.animals, action.name]
    });
  }
}
```


NGXS events

Action class vs Action String Type

- When we dispatch an action we use the Class AddAnimal,
  but the in the Reducer the class type is un-important, only the String Action.type
  will be used to execute the right methods, ex : `'[Zoo] Add Animal'`

```
addAnimal(name: string) {
  this.store.dispatch(new AddAnimal(name));
}
```

Identification is done only on the type String, not because the Class is AddAnimal

```
@State<ZooStateModel>({name: 'zoo'...})
export class ZooState {
  @Action(AddAnimal)
  addAnimals(ctx: StateContext<ZooStateModel>, action: AddAnimal) {
    const state = ctx.getState();
    ctx.setState({
      ...state,
      animals: [...state.animals, action.name]
    });
  }
}
```

So if you define twice the same String both Actions would be executed on same dispatch

NGXS events

Action class vs Action String Type

- NGXS seems to introduce good tying to identify the Action, but only string-type matters

```
export class Start {
  static readonly type = '[P4P] Start';
}

export class Start2 {
  static readonly type = '[P4P] Start';
}
```

```
@Action(Start2)
start2(ctx: StateContext<P4pState>, action: Start2) {
  MyLogger.violet('PPPPPPP Start2 action even though Start was emit');
}

@Action(Start)
start(ctx: StateContext<P4pState>, action: Start) {
  ctx.setState(P4P_INITIAL_STATE);
}
```

```
startService() {
  this.store.dispatch(new Start());
}
```

Both action will be fired

- So it is similar to real Redux or NgRx in that respect, the class API is just relooking