NgRx: Patterns and Techniques

Victor Savkin Follow
Jul 13, 2017 · 10 min read

Victor Savkin is a co-founder of nrwl.io, providing Angular consulting to enterprise teams. He was previously on the Angular core team at Google, and built the dependency injection, change detection, forms, and router modules.

. . .

Managing state is one of the hardest problems when building front-end applications. Angular supports multiple ways of doing it. For instance, we can do it in the AngularJS 1-like way and mutate the state in place. Over the last year a different approach has been getting a lot more traction—using NgRx.

NgRx is a library built around a few key primitives and it helps us manage state. NgRx is a very simple library and does not make a lot of

assumptions about how we use it. That is why knowing patterns and techniques is crucial when using NgRx, and this is what this article is about.

Agenda

This is a long and in-depth read, so let's look at what we are going to learn first.

- I will start with a short overview of NgRx—just to make sure we are on the same page.
- Then, I'll switch and talk about actions, in particular, the categories of actions we see in most applications.
- I'll also talk about how to implement a request-reply pattern using correlation ids.
- Finally, I'll cover the store: its building blocks, common types of deciders and transformers, and how we can compose them to handle advanced scenarios.

. . .

Short Overview of NgRx

Programming with NgRx is programming with messages. In NgRx they are called actions. To start an interaction a component or a service creates an action, populates it with data, and then dispatches it. The component does not mutate any state in place.

```
@Component({
  selector: 'todos',
  templateUrl: './todos.component.html'
})
class TodosComponent {
  constructor(private store: Store<any>) {}

  addTodo(data: {text: string}) {
    this.store.dispatch({
      type: 'ADD_TODO',
```

Then the action is received and processed, which can be done in two ways.

First, an action can be processed by a reducer—a synchronous pure function creating applications states. It does it by applying an action to the current state of the application.

```
function todosReducer(todos: Todo[] = [], action: any): Tod
  if (action.type === 'ADD_TODO') {
    return [...todos, action.payload];
  } else {
    return todos;
```

Second, an action can be processed by an effects class. An effects class taps into the *Actions* object (an observable of all the actions flowing through the app) to execute the needed side effects.

```
class TodosEffects {
  constructor(private actions: Actions, private http: Http)

  @Effect() addTodo = this.actions.ofType('ADD_TODO')
    .concatMap((todo) => this.http.post(...));
```

Many interactions require both an effects class and a reducer, as shown here.

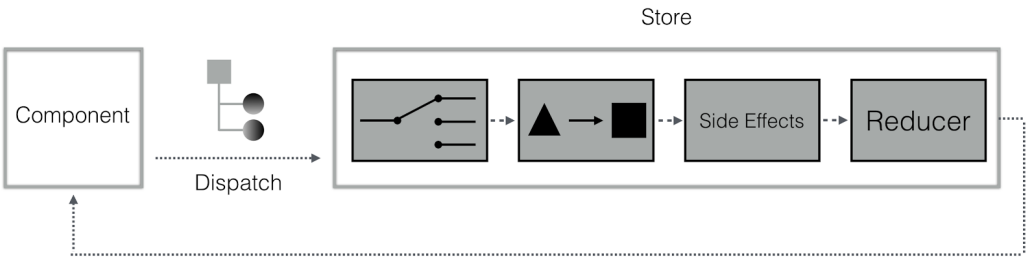
```
class TodosEffects {
  constructor(private actions: Actions, private http: Http

  @Effect() addTodo = this.actions.ofType('ADD_TODO').
    concatMap(todo => this.http.post(…).
    map(() => ({type: 'TODO_ADDED', payload: todo})));
}

function todosReducer(todos: Todo[] = [], action: any): To
  if (action.type === 'TODO_ADDED') {
```

In this example, we first make a post request in an effects class, and then update the app state in the reducer.

Finally, the component often reacts to the state changes by subscribing to the store. The following component gets an observable of todos. Any time a todo gets added, removed, or updated, that observable will emit a new array, which the component will display.

```
@Component({
  selector: 'todos',
  templateUrl: './todos.component.html'
})
class TodosComponent {
  public todos: Observable<Todo[]>;

  constructor(private store: Store<any>) {
    this.todos = store.select('todos');
  }

  addTodo(data: {text: string}) {
```

Graphically, it looks like this.

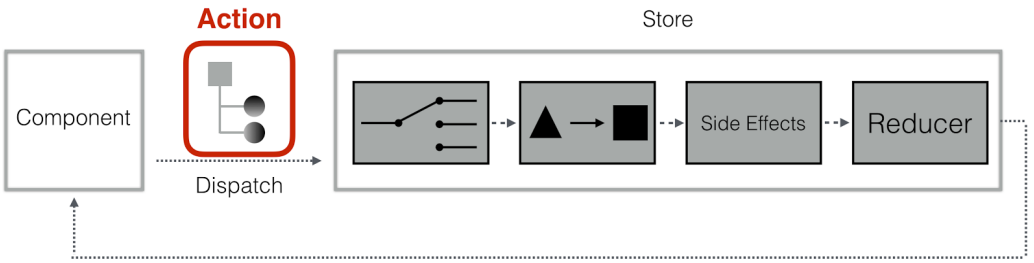


The component dispatches an action, which is processed in the store. This often involves deciding what should handle the action and how it should be transformed. This is done by the effects classes. Then we execute the needed side effects. Next, the reducers create a new state object. And, finally, the component displays the updated state. This is, in a nutshell, how NgRx works.

Now let's look at every part of this picture in detail.

. . .

Action



In NgRx, an action has two properties: type and payload. The type property is a string indicating what kind of action it is. The payload property is the extra information needed to process the action.

```
@Component({
  selector: 'todos',
  templateUrl: './todos.component.html'
})
class TodosComponent {
  constructor(private store: Store<any>) {}

  addTodo(data: {text: string}) {
    this.store.dispatch({
      type: 'ADD TODO',
```

As in most messaging systems, NgRx actions are reified, i.e., they are represented as concrete objects and they can be stored and passed around.

NgRx is a very simple library, so it does not make a lot of assumptions about your actions. NgRx does not prescribe one way to construct them, nor it tells us how to define types and payloads. But this does not mean that all actions are alike.

Three Categories of Actions

Actions can be divided into the following three categories: commands, documents, events.

> *Interestingly, the same categorization works well for most messaging systems.*

Commands

```
{
  type: 'ADD_TODO',
  payload: data
}
```

Dispatching a command is akin to invoking a method: we want to do something specific. This means we can only have one place, one handler, for every command. This also means that we may expect a reply: either a value or an error.

To indicate that an action is a command I name them starting with a verb.

Documents

```
{
  type: 'TODO',
  payload: data
}
```

When dispatching a document, we notify the application that some entity has been updated—we do not have a particular single handler in

mind. This means that we do not get any result and there might be more than one handler. Dispatching a document is less procedural.

Finally, I name my documents using nouns or noun phrases

Events

```
{
  type: 'APP_WENT_ONLINE',
  payload: data
}
```

When dispatching an event, we notify the application about an occured change. As with documents, we do not get a reply and there might be more than one handler.

Naming Conventions

I found using the naming convention to indicate the action category extremely useful, but we can go further than that and impose a certain schema on an action category. For instance, we can say that a document must have an ID, and an event must have a timestamp.

Using Several Actions to Implement a Single Interaction

We often use several actions to implement an interaction. Here, for instance, we use a command and an event to implement the todo addition. We handle the *ADD_TODO* command in an effects class, and then the *TODO_ADDED* event in the reducer.

```
class TodosEffects {
  constructor(private actions: Actions, private http: Http

  @Effect() addTodo = this.actions.ofType('ADD_TODO').
    concatMap(todo => this.http.post(…).
    map(() => ({type: 'TODO_ADDED', payload: todo})));
 }

function todosReducer(todos: Todo[] = [], action: any): To
  if (action.type === 'TODO_ADDED') {
```

Request—Reply

As I mentioned, when dispatching a command, we often expect a reply. But the dispatch method does not return anything, so how do we get it?

Let's look at the following component managing a todo. In its delete method we want to confirm that the user has not clicked on the delete button by accident.

```
@Component({
  selector: 'todo',
  templateUrl: './todo.component.html'
})
class TodoComponent {
  @Input() todo: Todo;

  constructor(private store: Store<any>) {}

  delete() {
    this.store.dispatch({
```

This operation will probably display some confirmation dialog, which may result in a router navigation and a URL change. These effects are not local to this component, and, as a result, must be handled by effects classes. This means that we have to dispatch an action.

Now imagine some effects class handling the confirmation. It will show the dialog to the user, get the result and store it as part of the application state. What the todo component needs to do is to query the state to get the result.

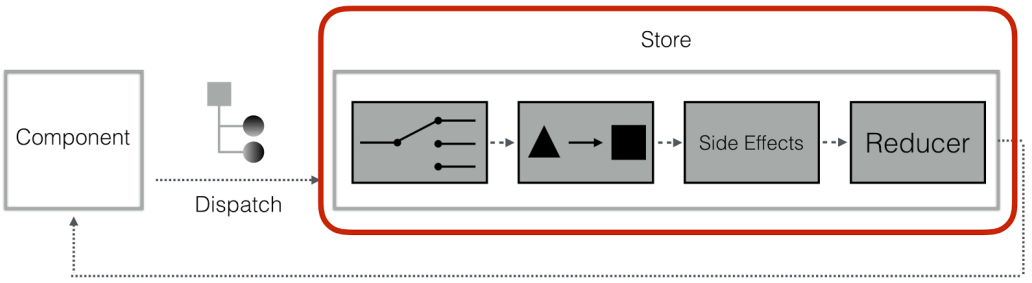
```
@Component({
  selector: 'todo',
  templateUrl: './todo.component.html'
})
class TodoComponent {
  @Input() todo: Todo;

  constructor(private store: Store<any>) {}

  delete() {
    this.store.dispatch({
      type: 'CONFIRM_TODO_DELETION',
      payload: {todoId: this.todo.id}
    });
```

Now how we are using the todo id to get the right reply. Ids used in this fashion are called correlation ids because we use them to correlate requests and replies. Entity ids tend to work well for this. But when they don't, we can always generate a synthetic correlation id.

```
@Component({
  selector: 'todo',
  templateUrl: './todo.component.html'
})
class TodoComponent {
  @Input() todo: Todo;

  constructor(private store: Store<any>) {}

  delete() {
    const correlationId = generateId();
    this.store.dispatch({
      type: 'CONFIRM_DELETION',
      payload: {todoId: this.todo.id, correlationId }
    });
```

. . .

Processing Actions

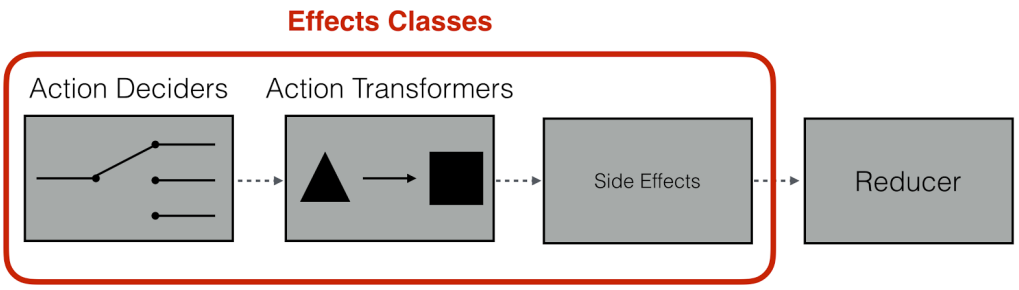


A dispatched action can be processed by effects classes and reducers.

Reducers are simple: they are synchronous pure functions creating new application states.

They are so simple because reducers don't deal with asynchrony, process coordination, talking to the server, which are the hard part. Effects classes deal with all of these. And that's where, it turns out, many patterns used for building message-based systems on the backend work really well.

Effects Classes



Effects classes have three roles:

- They decide on how to process actions.
- They transform actions into other actions.
- They perform side effects.

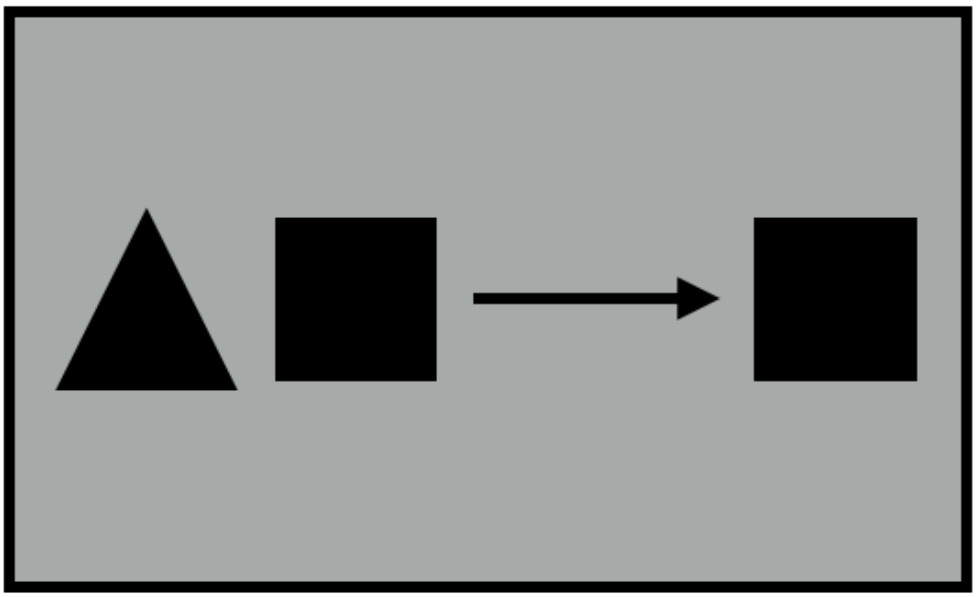
It's a good idea to keep these roles in mind when implementing effects classes. And, of course, it's even better to express them in the code.

Let's examine each of the roles in detail.

Action Deciders

An action decider determines if an effects class should process a particular action. A decider can also map an action to a different action.

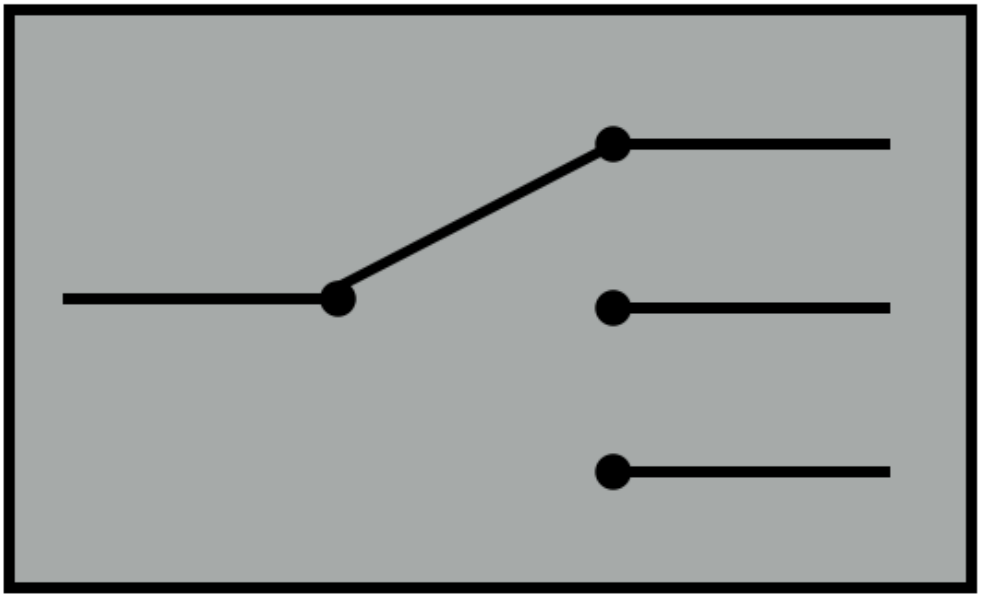
Filtering Decider



The most basic decider we all familiar with is the filtering decider. It is so common that NgRx comes with an operator implementing it: *ofType*.

```
class TodosEffects {
  constructor(private actions: Actions, private http: Http)

  @Effect() addTodo = this.actions.ofType('ADD_TODO').
    concatMap(todo => this.http.post(…).
```

Content-Based Decider



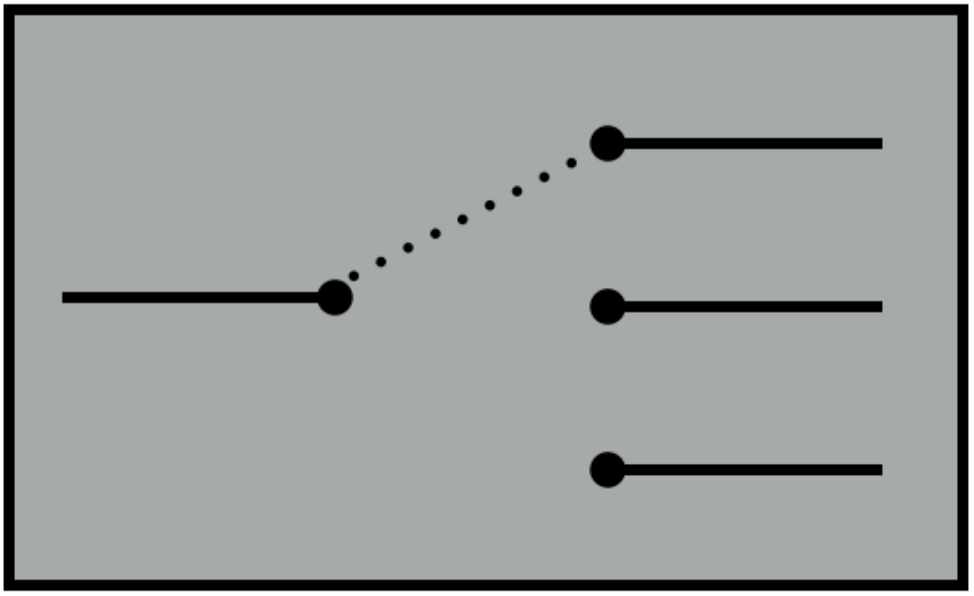
A content-based decider uses the payload of an action to map it to a different action.

In the following example, we are mapping *ADD_TODO* to ether *APPEND_TODO* or *INSERT_TODO*, depending on the content of the payload.

```
class TodosEffects {
  constructor(private actions: Actions) {}

  @Effect() addTodo = this.actions.typeOf('ADD_TODO').map(
    if (add.payload.addLast) {
      return ({type: 'APPEND_TODO', payload: add.payload})
    } else {
      return ({type: 'INSERT_TODO', payload: add.payload})
```

By using content-based deciders we introduce another level of indirection, which can be useful for several reasons. For instance, it allows us to change how certain actions are handled and what data they need, without affecting the component dispatching them.

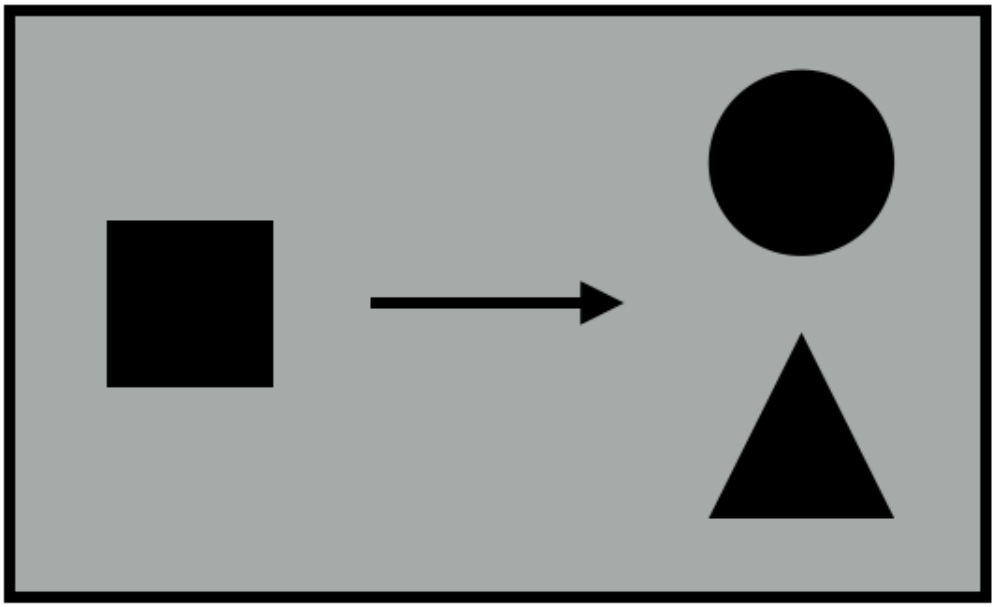
Context-Based Decider



A context-based decider uses some information from the environment to map an action to a different action. Using it allows us to have distinct workflows the component dispatching the action is not aware of.

```
class TodosEffects {
  constructor(private actions: Actions, private env: Env)

  @Effect() addTodo = this.actions.typeOf('ADD_TODO').map(
    if (this.env.confirmationIsOn) {
      return ({type: ‘ADD_TODO_WITH_CONFIRMATION’, payload
    } else {
      return ({type: ‘ADD_TODO_WITHOUT_CONFIRMATION', payl
```

Splitter

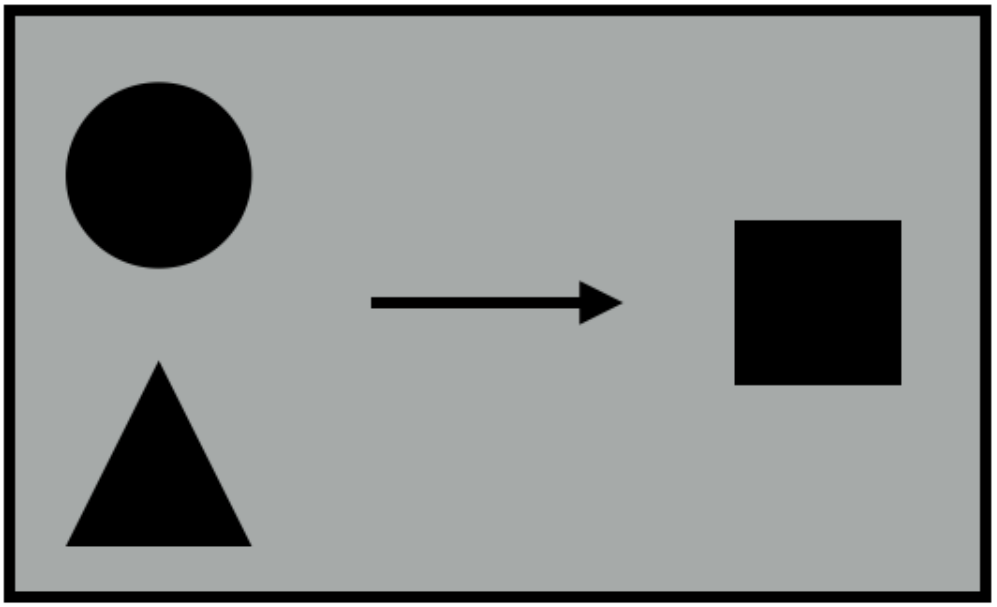


A splitter maps one action to an array of actions, i.e., it splits an action.

```
class TodosEffects {
  constructor(private actions: Actions) {}

  @Effect() addTodo = this.actions.typeOf('REQUEST_ADD_TODO
    {type: 'ADD_TODO', payload: add.payload},
    {type: 'LOG_OPERATION', payload: {loggedAction: 'ADD_TO
```

This is useful for exactly the same reasons as splitting a method into multiple methods: we can test, decorate, monitor every action independently.

Aggregator



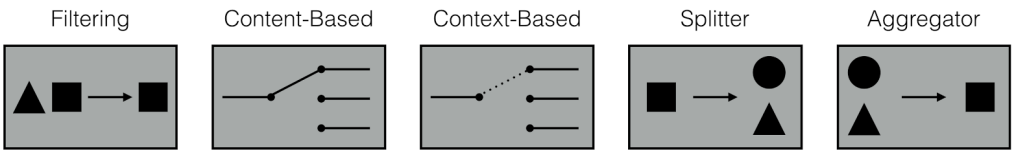
An aggregator maps an array of actions to a single action.

```
class TodosEffects {
  constructor(private actions: Actions) {}

  @Effect() aggregator = this.actions.typeOf('ADD_TODO').f
    zip(
      // note how we use a correlation id to select the ri
      this.actions.filter(t => t.type == 'TODO_ADDED' && t
      this.actions.filter(t => t.type == 'LOGGED' && t.pay
    )
  ).map(pair => ({
```

Aggregator are not as common as say splitters, so RxJs does not come with an operator implementing it. That's why we had to add some boilerplate to do it ourselves. But could always introduce a custom RxJS operator to help with that.

```
class TodosEffects {
  constructor(private actions: Actions) {}

  @Effects() a = this.actions.typeOf('ADD_TODO').
    aggregate(['TODO_ADDED', 'OPERATION_ADDED'], (a, t) =>
```

Overview Deciders

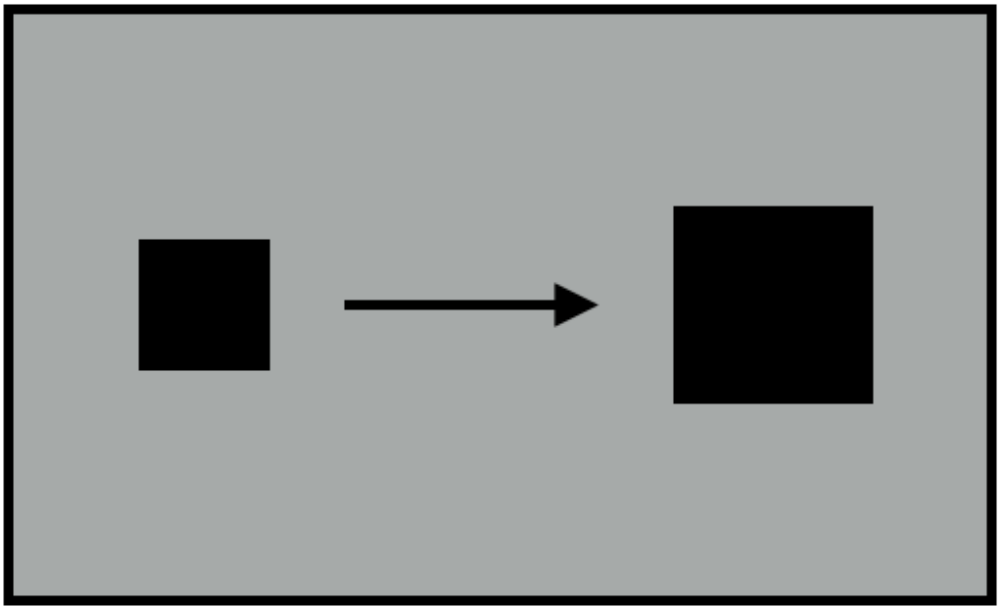


These are the most common deciders we just looked at:

- A filtering decider uses the action type to filter actions.
- A content-based decider uses the action payload to map an action to a different action.
- A context-based decider uses some injected object to map an action to another one.
- A splitter maps an action to an array of actions.
- A aggregator maps an array of actions to a single action.

Action Transformers

Content Enricher

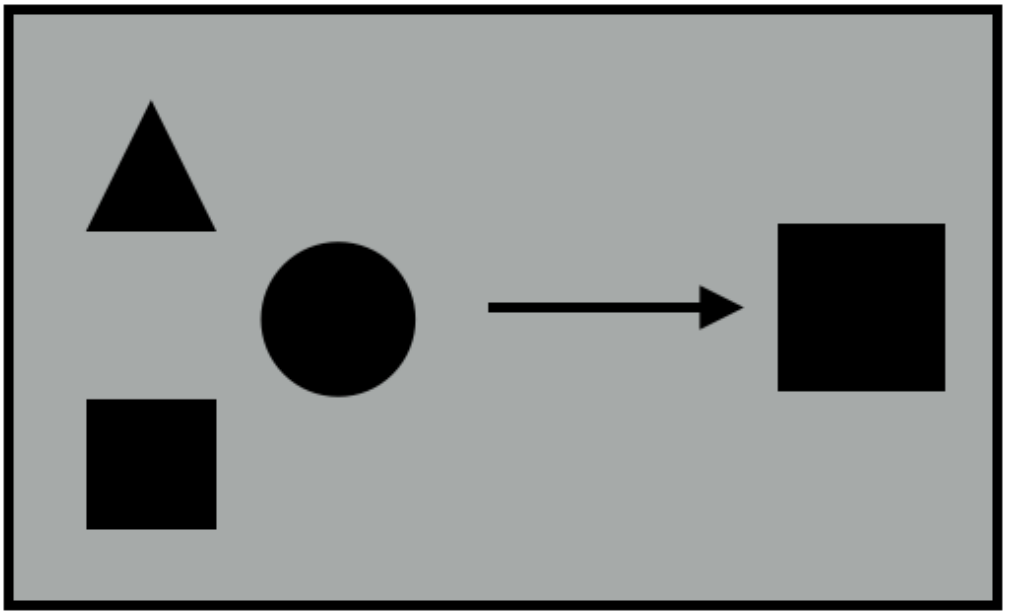


A content enricher adds some information to an action's payload.

```
class TodosEffects {
  constructor(private actions: Actions, private currentUser

  @Effect() addTodo = this.actions.ofType('ADD_TODO').
    map(add => ({
       action: 'ADD_TODO_BY_USER',
       payload: {...add.payload, user: this.currentUser}
```

This example is very basic: we merely add the already available current user to the payload. In a more interesting example we would fetch data from the backend and add it to the payload.

Normalizer & Canonical Actions

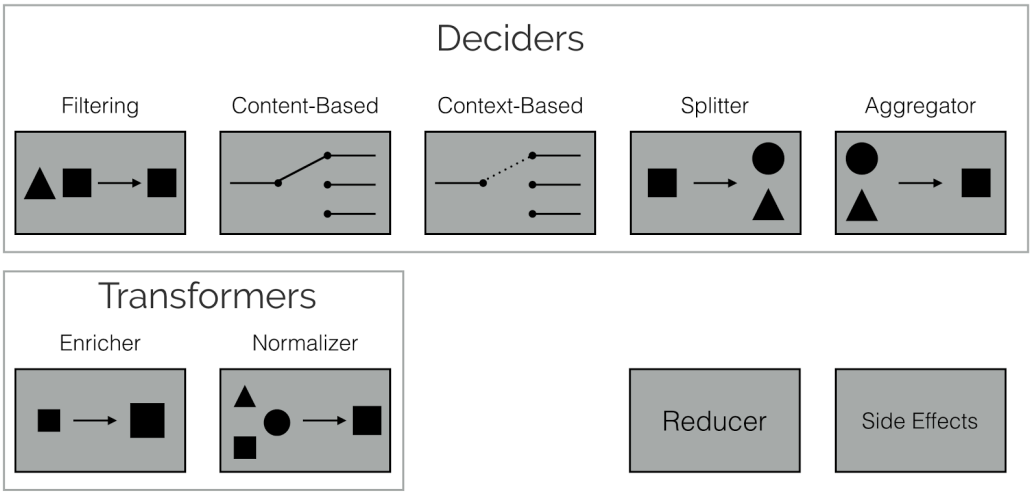


A normalizer maps a few similar actions to the same (canonical) action.

```
class TodosEffects {
  constructor(private actions: Actions) {}

  @Effect() insertTodo = this.actions.ofType('INSERT_TODO'
    map(insert => ({
      action: 'ADD_TODO',
      payload: {...insert.payload, append: false}
    }));

  @Effect() appendTodo = this.actions.ofType('APPEND_TODO'
```

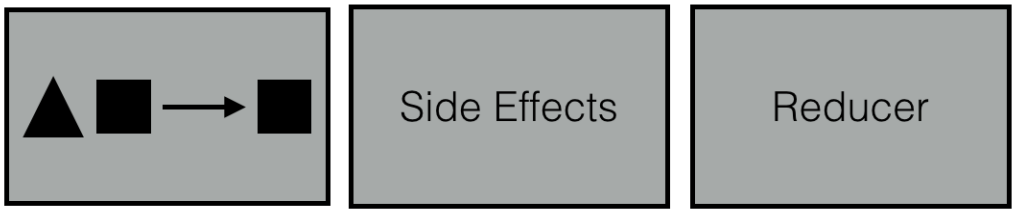
Building Blocks

These are the common building blocks used to implement application logic using NgRx:

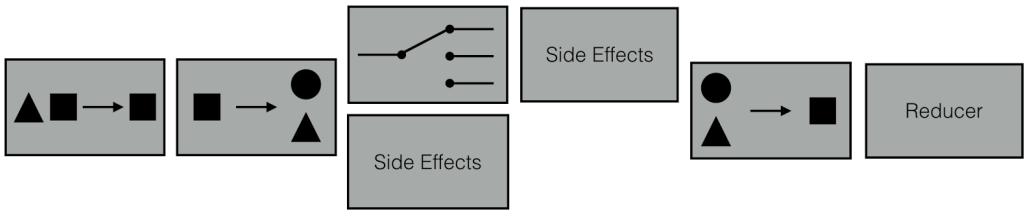


The best thing about them is how well they compose.

Let's look at a simple scenario first, where we select one type of action, execute needed side effects, and then update the state.



Often that's not enough, and we need to implement at a more complex scenario.



In this scenario we start with a filtering decider, next we use a splitter. We then use a content-based decider for the top action. The bottom one

is simpler. After executing the side effects, we aggregate the results and pass them to the reducer.

To make one thing clear, I'm not advocating splitting every interaction into ten separate classes. This is the last thing we should do. What I'm advocating is having a clear mental model and a language we can use to talk about these things with our fellow developers.

The next example, for instance, implements a complex interaction, but everything is done in one class.

```
class TodosEffects {
  constructor(private actions: Actions, private currentUse

  @Effect() addTodo =
     this.actions.ofType('ADD_TODO'). // filtering decider
     map(t => ({type: t.type, payload: {...payload, user:
     map(t => t.append ? // content-based decider
        ({type: 'APPEND_TODO', payload: t.payload}) :
        ({type: 'INSERT_TODO', payload: t.payload})).
     flatMap(t => [t, {type: 'LOG_OPERATION', payload: t}]

  @Effect() appendTodo =
    this.actions.ofType('APPEND_TODO').
```

But even though everything is implemented in a single class, we can still talk about every aspect of it using the language and the patterns we learned in this article.

Using the labels forces us to be more intentional about the design of our effects classes. Here, for instance, we can see that the *addTodo* effect does not execute any side effects. And the *appendTodo* and *insertTodo*

effects only execute side effects. This alone has huge implications on how these things should be tested, monitored, etc..

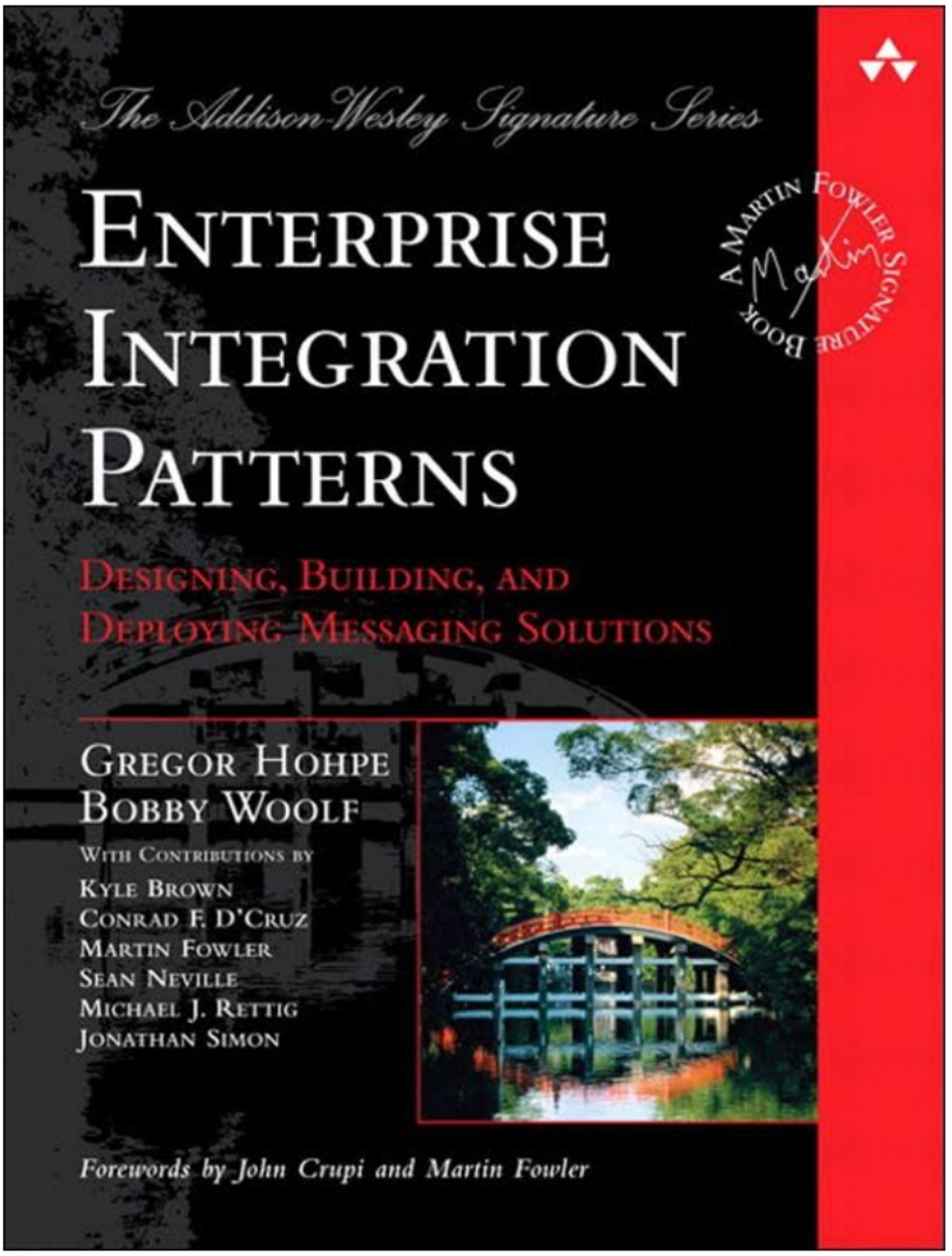
. . .

Summary

- Programming with NgRx is programming with messages. In NgRx they are called ‘actions’.
- Components create and dispatch actions.
- Actions can be categorized as commands, events, and documents.
- A command has a single handler, and we often expect a reply after dispatching it.
- Events and documents can have multiple handlers, and we do not expect replies.
- Deciders, transformers, reducers, and side effects are the building blocks we use to express our application logic. They compose well.

Enterprise Integration Patterns

This article is based on this book. The title “Enterprise Integration Patterns” may sound a bit scary, but this is the best book on messaging I know of. So I highly recommend you to check it out.



. . .

Victor Savkin is a co-founder of Nrwl—Enterprise Angular Consulting.

. . .

If you liked this, click the ♡ below so other people will see this here on Medium. Follow @victorsavkin to read more about Angular.