NgRx Facade Pattern

The NgRx Facade Pattern was first introduced by Thomas Burleson in 2018 and has drawn a lot of attention in recent years. In this article, we will discuss the pattern, how to implement it in Angular and discuss whether or not we *should* implement it.

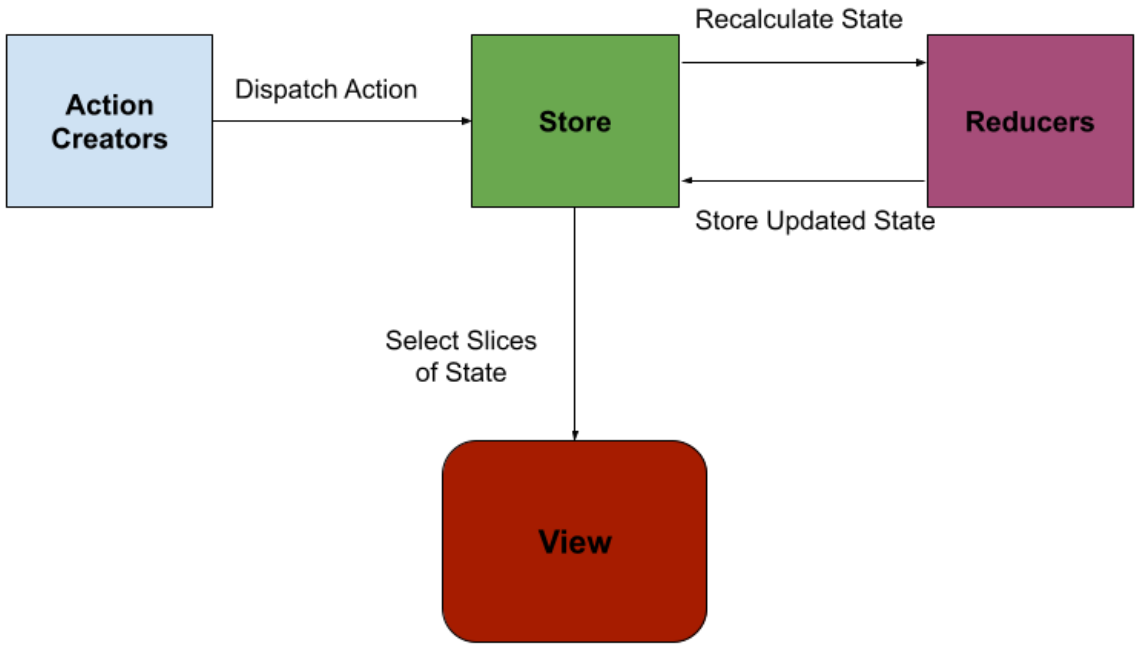
What is NgRx?

First, what is NgRx?
NgRx is a state management solution for Angular built on top of RxJS which adheres to the redux pattern.

It contains an immutable centralized store where the state of our application gets stored.

- We select slices of state from the `Store` using `Selectors`, which we can then render in our components.
- We dispatch `Actions` to our `Store`.
- Our `Store` redirects our `Action` to our `Reducers` to recalculate our state and replaces the state within our `Store`.

See the diagram below for an illustrated example:



This provides us with a tried and tested pattern for managing the state of our application.

What is the Facade Pattern?

Now that we know what NgRx is, what is the Facade Pattern?

Well, what *are* Facades?
Facades are a pattern that provides a simple public interface to mask more complex usage.

As we use NgRx more and more in our application, we add more actions and more selectors that our components must use and track. This increases the coupling between our component and the actions and selectors themselves.

The Facade pattern wants to simplify this approach by wrapping the NgRx interactions in one place, allowing the Component to only ever interact with the Facade. This means you are free to refactor the NgRx artefacts without worrying about breaking your Components.

In Angular, NgRx Facades are simply services. They inject the NgRx Store allowing you to contain your interactions with the Store in the service.

How do we implement it?

To begin with, let's show a Component that uses NgRx directly:

```
export class TodoListComponent implements OnInit {
  todoList$: Observable<Todo[]>;

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

  ngOnInit() {
    this.todoList$ = this.store.select(getLoadedTodoList);

    this.loadTodoList();
  }

  loadTodoList() {
    this.store.dispatch(new LoadTodoList());
```

```
  }

  addNewTodo(todo: string) {
    this.store.dispatch(new AddTodo(todo));
  }

  editTodo(id: string, todo: string) {
    this.store.dispatch(new EditTodo({ id, todo }));
  }

  deleteTodo(id: string) {
    this.store.dispatch(new DeleteTodo(id));
  }
}
```

As we can see, this depends a lot on interactions with the Store and has made our component fairly complex and coupled to NgRx.

Let's create a Facade that will encapsulate this interaction with NgRx:

```
@Injectable({
  providedIn: 'root',
})
export class TodoListFacade {
  todoList$ = this.store.select(getLoadedTodoList);

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

  loadTodoList() {
    this.store.dispatch(new LoadTodoList());
  }

  addNewTodo(todo: string) {
    this.store.dispatch(new AddTodo(todo));
  }

  editTodo(id: string, todo: string) {
    this.store.dispatch(new EditTodo({ id, todo }));
  }

  deleteTodo(id: string) {
    this.store.dispatch(new DeleteTodo(id));
```

```
  }
}
```

It's essentially everything we had in the component, except now in a service.

We then inject this service into our Component:

```
export class TodoListComponent implements OnInit {
  todoList$: Observable<Todo[]>;

  constructor(private todoListFacade: TodoListFacade) {}

  ngOnInit() {
    this.todoList$ = this.todoListFacade.todoList$;

    this.todoListFacade.loadTodoList();
  }

  addNewTodo(todo: string) {
    this.todoListFacade.addNewTodo(todo);
  }

  editTodo(id: string, todo: string) {
    this.todoListFacade.editTodo({ id, todo }));
  }

  deleteTodo(id: string) {
    this.todoListFacade.deleteTodo(id);
  }
}
```

By implementing the Facade and using it in our Component, our component no longer depends on NgRx and we do not have to import all actions and selectors.

The Facade hides those implementation details, keeping our Component cleaner and easier tested.

Pros

What are some advantages of using Facades?

- **It adds a single abstraction of a section of the Store.**

    - This service can be used by any component that needs to interact with this section of the store. For example, if another component needs to access the `TodoListState` from our example above, they do not have to reimplement the action dispatch or state selector code. It's all readily available in the Facade.
- **Facades are scalable**
    - As Facades are just services, we can compose them within other Facades allowing us to maintain the encapsulation and hide complex logic that interacts directly with NgRx, leaving us with an API that our developers can consume.

Cons

- **Facades lead to reusing Actions.**
    - Mike Ryan gave a talk at ng-conf 2018 on Good Action Hygiene which promotes creating as many actions as possible that dictate how your user is using your app and allowing NgRx to update the state of the application from your user's interactions.
    - Facades force actions to be reused. This becomes a problem as we no longer update state based on the user's interactions. Instead, we create a coupling between our actions and various component areas within our application.
    - Therefore, by changing one action and one accompanying reducer, we could be impacting a significant portion of our application.
- **We lose indirection**
    - Indirection is when a portion of our app is responsible for certain logic, and the other pieces of our app (the view layer etc.) communicate with it via messages.
    - In NgRx, this means that our Effects or Reducers do not know what told them to work; they just know they have to.
    - With Facades, we hide this indirection as only the service knows about how the state is being updated.
- **Knowledge Cost**
    - It becomes more difficult for junior developers to understand how to interact, update and work with NgRx if their only interactions with the state management solution are through Facades.

- It also becomes more difficult for them to write new Actions, Reducers and Selectors as they may not have been exposed to them before.

Conclusion

Hopefully, this gives you an introduction to NgRx Facades and the pros and cons of using them. This should help you evaluate whether to use them or not.