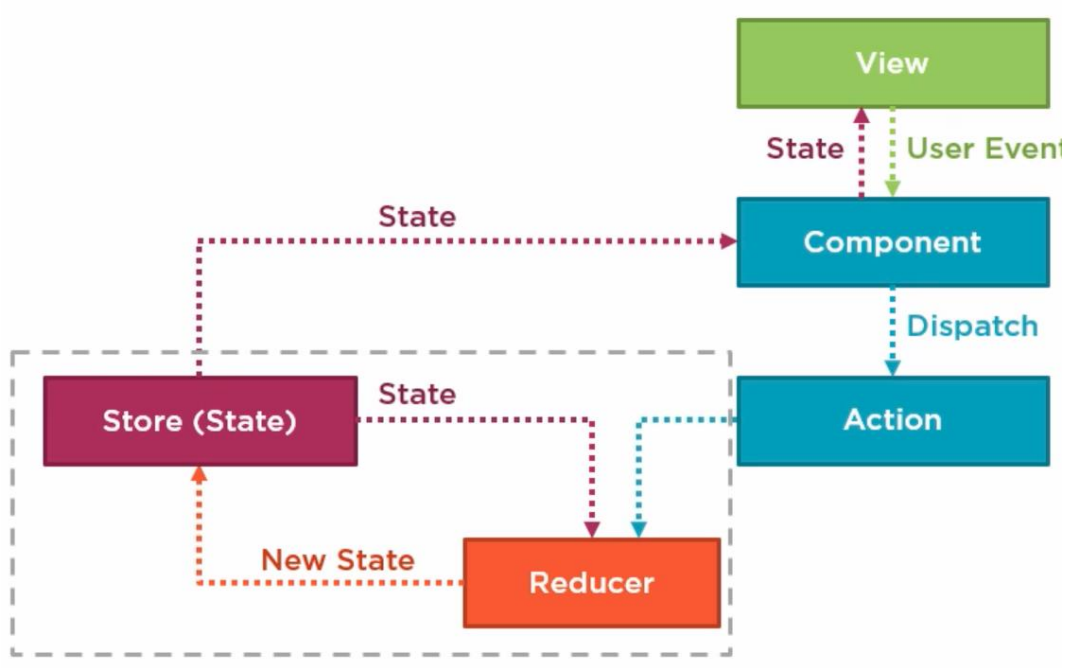
Primer vistazo a NgRx – Inicializando el Store

Cuando configuramos el Store en nuestra aplicación, inicializamos el Store con su Reducer:



Recordemos que el Reducer toma un *Action* y el estado existente, y actualiza el Store con ese nuevo estado. Así que tiene sentido asociar el Store con su Reducer que crea el estado para ese Store.

El código para inicializar el Store luce de la siguiente forma:

App Module

```
import { StoreModule }
          from '@ngrx/store';

@NgModule({
 imports: [
  BrowserModule,
  RouterModule.forRoot(appRoutes),
  ...
  StoreModule.forRoot(reducer)
 ],
 declarations: [ ... ],
 bootstrap: [ AppComponent ]
})
export class AppModule { }
```

En nuestro app.module, usamos la sentencia import para importar el StoreModule de la librería @ngrx/store que acabamos de instalar:

App Module

```
import { StoreModule }
          from '@ngrx/store';

@NgModule({
 imports: [
  BrowserModule,
  RouterModule.forRoot(appRoutes),
  ...
  StoreModule.forRoot(reducer)
 ],
 declarations: [ ... ],
 bootstrap: [ AppComponent ]
})
export class AppModule { }
```

A continuación, añadimos el módulo *StoreModule* al array de imports:

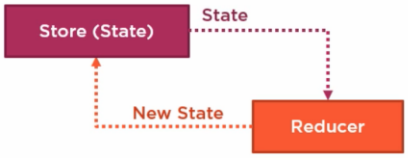
App Module

```
import { StoreModule }
          from '@ngrx/store';

@NgModule({
 imports: [
  BrowserModule,
  RouterModule.forRoot(appRoutes),
  ...
  StoreModule.forRoot(reducer)
 ],
 declarations: [ ... ],
 bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Como este es el módulo raíz de la aplicación, llamamos al método *forRoot* y le pasamos una referencia del Reducer. Todavía no hemos visto el código de nuestro Reducer, pero lo haremos en breve. Esta inicialización del Store asocia el Reducer con el Store y registra el contenedor de estados con la aplicación.

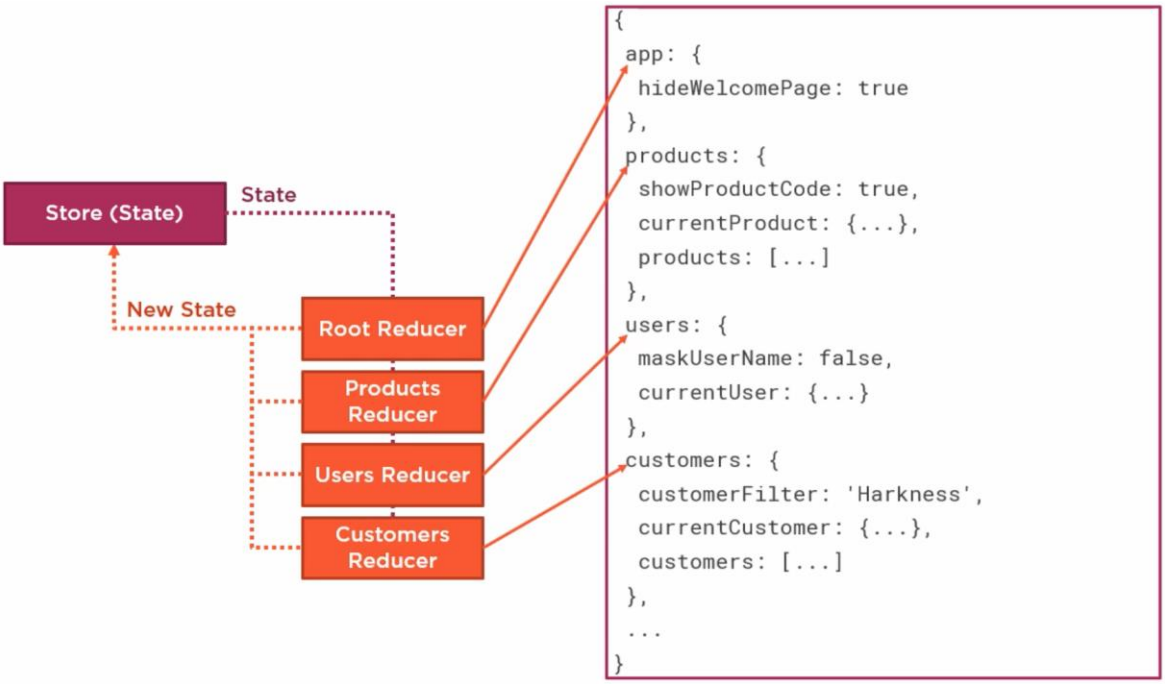
Con NgRx, tenemos un Store para todo el estado, por lo que podríamos construir un único Reducer para trabajar con el estado, pero no queremos un estado grande, plano y masivo:




```
{
  showProductCode: true,
  currentProduct: {...},
  products: [...],
  hideWelcomePage: true,
  maskUserName: false,
  currentUser: {...},
  customerFilter: 'Harkness',
  currentCustomer: {...},
  customers: [...],
  allowGuest: false,
  includeAds: true,
  displayCustomerDetail: false,
  allowEdit: false,
  listFilter: 'Hammer',
  displayAddress: false,
  orders: [...],
  cart: [...],
  ...
}
```

Tampoco queremos un Reducer enorme para gestionar todo el estado. Más bien, organizaremos nuestro estado por features para que coincida con nuestros módulos de tipo feature.

Para crear este estado, definimos varios Reducers, uno para cada *feature slice* del estado:



Cada Reducer es entonces más pequeño y más centrado y por lo tanto más fácil de construir, mantener y probar, además nunca se crea estado para un módulo que no está cargado. Es la combinación de estos Reducers la que representa el estado de la aplicación en un momento dado.

Para lograr este tipo de organización, utilizaremos una técnica denominada *Feature Module State Composition*. Esto nos permite componer el estado de nuestra aplicación a partir de nuestros *Feature Module Reducers*. Esto suena mucho más difícil de lo que es.

Para utilizar Feature Module State Composition necesitamos hacer lo siguiente:

1. Comenzamos inicializando el estado de nuestra aplicación en el archivo app.module:

App Module

```
import { StoreModule }
          from '@ngrx/store';

@NgModule({
 imports: [
  BrowserModule,
  RouterModule.forRoot(appRoutes),
  ...
  StoreModule.forRoot(reducer)
 ],
 declarations: [ ... ],
 bootstrap: [ AppComponent ]
})
export class AppModule { }
```

   Pasamos al método *forRoot* el Reducer que crea el estado raíz de nuestra aplicación. Esto podría incluir valores como el del flag hideWelcomePage.

2. A continuación, inicializamos el estado de cada feature utilizando el método *StoreModule.forFeature*:

Product Module

```
import { StoreModule }
          from '@ngrx/store';

@NgModule({
 imports: [
  SharedModule,
  RouterModule.forChild(productRoutes),
  ...
  StoreModule.forFeature('products', productReducer)
 ],
 declarations: [ ... ],
 providers: [ ... ]
})
export class ProductModule { }
```

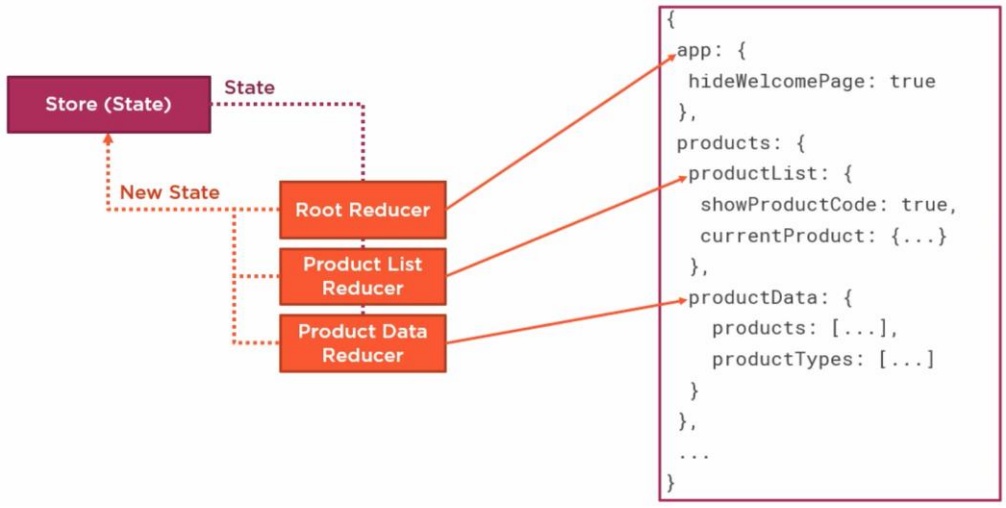
   El método *forFeature* toma el nombre del *feature slice*, que suele ser la forma plural de nuestro feature (products, en nuestro caso). El segundo argumento es una referencia al Reducer que gestiona el estado de nuestro feature slice (productReducer, en el caso de nuestro Reducer).

Podríamos desglosar aún más nuestro estado y construir varios Reducers para una porción de un feature:

```
{
 app: {
  hideWelcomePage: true
 },
 products: {
  productList: {
   showProductCode: true,
   currentProduct: {...}
  },
  productData: {
    products: [...],
    productTypes: [...]
  }
 },
 ...
}
```

En este ejemplo, dividimos el *slice* del estado de products en dos *sub-slices*, una para la configuración de la página de configuración de productList y la otra para productData.

A continuación, definimos un Reducer para cada *sub-slice*:



Estos Reducers se agregan para su feature asociado. Esto es especialmente útil cuando un solo feature tiene un estado muy grande, lo que nos permite dividir nuestros Reducers en piezas más pequeñas.

Al inicializar el Store de este escenario, volvemos a establecer el nombre del *feature slice* como un string en el primer argumento del método *forFeature*:

Product Module

```
StoreModule.forFeature('products',
  {
     productList: listReducer,
     productData: dataReducer
  }
)
```

En el segundo argumento, pasamos el conjunto de Reducers como pares clave valor:

Product Module

```
StoreModule.forFeature('products',
  {
     productList: listReducer,
     productData: dataReducer
  }
)
```

La clave es el nombre de cada *sub-clice* de estado, y el valor es una referencia a su Reducer asociado:

Para nuestro primer ejemplo, vamos a quedarnos con un Reducer para nuestro *feature slice* de productos: