**Normalizer**

The idea behind [normalizr](https://github.com/gaearon/normalizr) is to take an API response that has nested resources and flatten them. It's a simple idea with a great upside - **it becomes much easier to query and manipulate data for your components**. And this is even better combined with reselect, which we'll talk about soon.

Some quick rules and reasoning for using Normalizr:

1. Use it directly within a callback from your API in your actions
2. After normalizing data convert entities to [Immutable.js records](https://facebook.github.io/immutable-js/docs/#/Record). We'll introduce these in the next chapter
3. Pass the normalized records to reducers. Reducers should only store these
4. When reducers only store normalized records it becomes super easy to query using reselect.

Essentially, Normalizr is a utility for making Reselect even easier.

### An example: building a blog

Let's say you're building a blog, and the API response for /posts/1 lists the post. The post embeds the post author and all comments, then each comment embeds an author. It might look like this:

{

"id": 1,

"title": "Blog Title",

"content": "Some really short blog content. Actually the least interesting post ever",

"created\_at": "2016-01-10T23:07:43.248Z",

"updated\_at": ""2016-01-10T23:07:43.248Z",

"author": {

"id": 81,

"name": "Mr Shelby"

}

"comments": [

{

"id": 352,

"content": "First!",

"author": {

"id": 41,

"name": "Foo Bar"

}

}

]

}

If we wanted to pull out all comment authors for a component we'd have to request the post, iterate through each comment, and pull out author.name. This gets tedious fast, and it's even worse with more deeply-nested resources.

Let's see how normalizr improves this.

##### Setting up normalizr

The first thing to do so we can normalize this API response is set up normalizr. This means we need to describe our resource relationships:

'use strict';

import { Schema, arrayOf } from 'normalizr';

// Here we create the normalizr schemas. These define the entity names

// (entities.posts, for example) and how we get the ID for each entity.

//

// We could define a function for idAttribute which returns a key (such as

// combining two fields).

const postSchema = new Schema('posts', { idAttribute: 'id' });

const postAuthorSchema = new Schema('postAuthors', { idAttribute: 'id' });

const commentSchema = new Schema('comments', { idAttribute: 'id' });

const CommentAuthorSchema = new Schema('commentAuthors', { idAttribute: 'id' });

// Here we define relationships between each resource/schema/entity

// (or whatever you feel like calling them these days)

// The post resource in our API response has an author and comments as children

postSchema.define({

author: postAuthorSchema,

comments: arrayOf(commentSchema)

});

// Each comment has an author

commentSchema.define({

author: commentAuthorSchema

});

Now this is set up Normalizr can do it's thing. When we call normalizr on a post it'll recursively extract all data. Calling normalizr like so:

normalize(response.body, postSchema);

Will produce this:

{

result: [1],

entities: {

posts: {

1: {

"id": 1,

"title": "Blog Title",

"content": "Some really short blog content. Actually the least interesting post ever",

"created\_at": "2016-01-10T23:07:43.248Z",

"updated\_at": ""2016-01-10T23:07:43.248Z",

"author": 81

"comments": [352]

}

},

postAuthors: {

81: {

"id": 81,

"name": "Mr Shelby"

}

},

comments: {

352: {

"id": 352,

"content": "First!",

"author": 41

}

},

commentAuthors: {

41: {

"id": 41,

"name": "Foo Bar"

}

}

}

}

Kinda magic! Let's see the three core features:

1. Each resource type is yanked into the entities object.
2. The 'Post' resource no longer embeds each comment. Instead it's replaced with an array of comment IDs. This makes it easy and fast to look up comments from entities.comments.
3. results stores an array of IDs of posts in the order of the API response (objects do not guarantee orders)

##### Benefits

Let's say we now wanted to list all comment authors. Instead of recursing through each comment and pulling the authors we already have a list defined. It's super easy to do. It's super easy to figure out just how many comments we also have.

##### Caveats

###### Maintaining API ordering

Normalizr places your data inside an object within entities. Unfortunately JS objects **do not guarantee any specific order**. If you're normalizing a search result, for example, this might completely break the search results you display.

To maintain the order of data from the API you must **iterate through the result property of the normalized object**. This is an array which lists the IDs of entities from the search result. And arrays safely guarantee the order of its values.

If you're iterating through an embedded resource normalizr replaces these with an array of IDs also in the same order as the API — see below.

For example:

{

// This is an array of the top-level schema passed to normalize()

result: [1],

entities: {

posts: {

1: {

...

// This is an array of comment IDs in the same order

// as the API. Iterate through entities.comments to pull

// the resources out.

"comments": [352]

}

},

comments: {...}

...

}

}

And here's some code to pull this out of the posts reducer:

// Returns an array of page entities in order from the API response

const getPages = (state) => {

const result = state.posts.get('result');

return result.map(id => state.entities.getIn(['posts', id]));

}