redux-db Documentation

redux-db

Feb 16, 2019

Basics:

redux-db provides a normalized redux store and easy object management.

CHAPTER 1

Installation

npm:

```
npm install redux-db --save
```

yarn:

```
yarn add redux-db
```

bower:

```
bower install redux-db
```

CHAPTER 2

The schema

To make redux-db know how to tie your records together and how to normalize given data, you must define a schema. The schema is defined using a simple nested JSON object. Each property in the schema object defines a table name. Each table name must again define a new object for it's fields.

```
{
    "TableName": {
        "FieldName": { ... fieldProperties }
    }
}
```

The table name is up to you, but the field names **must match** your data model.

Note: It is not really required, but the table name should be written using pascal casing. This helps you seperate Table class instances later on.

2.1 Primary keys

If a table represents a single entity you should define a primary key.

```
{
    "Table" : {
        "id": { type: "PK" }
    }
}
```

You may define multiple fields as primary keys. Doing so will combine the key values to a unique id. This is useful for "join" tables.

2.2 Foreign keys

You connect your tables using foreign keys. Foreign keys are noted using the “references” property.

```
{
    "Table1" : {
        "ref": { references: "Table2", relationName: "refs" }
    },
    "Table2": {
        id: { type: "PK" }
    }
}
```

The “relationName” property is required if you wish to access the relationship on the Records of “Table2”. It is also used during normalization. An update/insert operation with the following data; will use the “relationName” to automatically normalize nested “refs”.

```
[
    { id: 1, refs: [
        { id: 10, someField: "with value" },
        { id: 11, someField: "with another value" }
    ]}
]
```

The “refs” will automatically “flow” to “Table2”. Upserting the related table.

2.3 Many to many relationships

To define a many to many relationship you can apply “join” tables. Tables that have multiple foreign keys to other tables.

```
{
    "Table1" : {
        id: { type: "PK" }
    },
    "Table2": {
        id: { type: "PK" }
    }
    "JoinTable": {
        "table1Id": { references: "Table1", relationName: "table2Refs" },
        "table2Id": { references: "Table2", relationName: "table1Refs" }
    }
}
```

2.4 One to one relationships

To specify a one to one relationship, you can set the “unique” flag on the field definition. This will enforce the unique constraint for updates and inserts.

```
{
    "Parent":{
        "id": { type: "PK" }
```

(continues on next page)

(continued from previous page)

```
    },
    "Child": {
        "id": { type: "PK" },
        "parentId": { references: "Parent", relationName: "child", unique: true }
    }
}
```

2.5 Self referencing tables

It's perfectly fine to add a self referencing table field

```
{
    "Table": {
        "id": { type: "PK" },
        "parentId": { references: "Table", relationName: "children" }
    }
}
```

2.6 Cascading deletes

You may define the "cascade" flag on a field definition. This will automatically delete the referencing records when the foreign table record is deleted.

```
{
    "Parent":{
        "id": { type: "PK" }
    },
    "Child": {
        "id": { type: "PK" },

        // when the related record of "Parent" is deleted, all it's children is also␣
↪deleted.
        "parentId": { references: "Parent", relationName: "children", cascade: true }
    }
}
```

2.7 Other field types

Aside for the "PK" type, you may also define fields of type "ATTR" and "MODIFIED".

The "ATTR" type is the default if not defined. This type promotes a field from the data model to the record model.

```
{
    "Table": {
        "id": { type: "PK" },
        "someValue": { type: "ATTR" }
    }
}
```

This will declare a "someValue" property on the RecordModel instance.

The "MODIFIED" type defines a field that will be used to compare the state record with a updated record. The default comparison is to do a object shallow equallity check.

```
{
    "Table": {
        "id": { type: "PK" },
        "modifiedDate": { type: "MODIFIED" } // Uses the "modifiedDate" value to␣
↪compare objects.
    }
}
```

Tip: Using the "MODIFIED" type could improve the performance for large updates.

2.8 Custom value factory

If you wish to transform the value of a given field, you may define a callback on the field definition.

```
{
    "Table": {
        "id": { type: "PK" },
        "fullName": { value: (r)=> r.firstName + " " + r.lastName },
        "modified": { type: "MODIFIED", value: r=> r.stamp1 + r.stamp2 } // Uses a␣
↪computed value to compare objects.
    }
}
```

2.9 Custom normalization

During data normalization you may have the need to transform the data. redux-db provides a basic normalization hook for each table.

```
ReduxDB.createDatabase( schema, {
    onNormalize: {
        "Table1": ( item, context ) => {
            const { id, name, ...rest } = item;

            // We split the given data and emits to "Table2" for normalization.
            context.emit( "Table2", rest );

            // returns the data for "Table1"
            return { id, name };
        }
    }
});
```

2.10 Schema reference

All supported definitions

```
{
    "Table" : {
        "Field": {
            type: "PK" | "MODIFIED" | "ATTR",

            // Defines a custom property name for the field. Defaults to the field␣
↪name.
            propName?: string;

            // Defines the foreign table this field references.
            references?: string;

            // Defines the relationship name, which'll be the property name on the␣
↪foreign table.
            relationName?: string;

            // If set, causes the record to be deleted if the foreign table row is␣
↪deleted.
            cascade?: boolean;

            // If set, declares that this relation is a one 2 one relationship.
            unique?: boolean;

            // Defines a custom value factory for each record.
            value?: (record: any, context?: ComputeContext) => any;
        }
    }
}
```

CHAPTER 3

The object model

To access and manipulate your normalized state redux-db provides you with a simple "Object-relational mapping" (ORM).

3.1 Database

Your schema definition must be provided to a Database class instance:

```
import { createDatabase } from "redux-db";

const schema = {
    ...
};

export const db = createDatabase( schema, /*options*/ );
```

3.2 Session

redux-db uses the concept of a session where each "table" is wrapped in a *TableModel* class. The *TableModel* class helps to query and perform CRUD operations easily.

To begin a new session and perform some action:

```
import { db } from "./schema";

const session = db.createSession(state /* not defined here */);

// your TableModels are properties in the session "tables" object.
const { BlogPost, Comment, User } = session.tables;

// the "get" method retrives a RecordModel by it's primary key.
```

(continues on next page)

(continued from previous page)

```
// the "update" method allows for partial updates of record data.
BlogPost.get("post1").update({ body: "new text" });

// commit the session
const newState = session.commit();
```

3.3 TableModel

The Table class provides several methods and properties for accessing the table records. Each method will return table rows wrapped in a *RecordModel* class.

Methods and propertes:

```
/// gets the number of records in the table.
length : number;

/// gets all raw record values
/// equivalent to all().map( r => r.value )
getValues() : any[];

/// returns all records in table.
all() : RecordModel[];

/// returns all records matching a given filter.
filter( predicate: ( record: RecordModel ) => boolean ) : RecordModel[];

/// returns a single record by id.
get( id:string|number ) : RecordModel;

/// returns a single record by id. null if not found.
getOrDefault(id: number | string) : RecordModel | null;

/// checks whether a record exists.
exists(id: number | string) : bool;

/// inserts one or more records.
/// returns the first inserted record.
insert( data: any ) : RecordModel;

/// inserts one or more records.
/// returns the inserted records.
insertMany( data: any ) : RecordModel[];

/// updates one or more records.
/// returns the first updated record.
update( data: any ) : RecordModel;

/// updates one or more records.
/// returns the updated records.
updateMany( data: any ) : RecordModel[];

/// upserts one or more records.
/// returns the first upserted record.
upsert( data: any ) : RecordModel;
```

(continues on next page)

(continued from previous page)

```
/// deletes a single record by it's primary key.
delete( id: string ) : boolean;

/// deletes all records in table.
deleteAll() : void;

/// get a single record value
value( id:number | string ) : any | undefined;
```

Note: The update/insert/upsert operations accepts nested data and will be normalized according to your schema definition. The normalized relations will also be updated/inserted to its respective tables.

3.4 RecordModel

The RecordModel class wraps an table row and provides methods and propertes for the given row/entity.

Methods and propertes:

```
/// gets the record primary key value
id: string;

/// gets the raw record value
value : any;

/// updates the record with new data.
update( data: any ) : this;

/// deletes the record
delete() : void;
```

In addition to the methods and propertes defined above, the RecordModel class will also contain properties for accessing foreign relations. Given the following schema:

```
{
    "Table1" : {
        "id": { type: "PK" }
    },
    "Table2" : {
        "id": { type: "PK" },
        "ref": { references: "Table1", relationName: "rels" }
    }
};
```

The Record class for “Table1” will contain a property “rels” of type *RecordSetModel*. The *RecordSetModel* wraps all records in “Table2” relating to “Table1” by its PK. The Record class for “Table2” will contain a property named “ref” which holds a Record of “Table1”.

```
Table1.insert( {
    id: 1,
    body: "some text"
});
```

(continues on next page)

(continued from previous page)

```
Table2.insert( [
    { id: 1, ref: 1 },
    { id: 2, ref: 1 },
    { id: 3, ref: 2 }
]);

const table1Record = Table1.get( 1 );

// table1Record will have the property "rels"
table1Record.rels;

// the rels property is a RecordSet of Records from "Table2"
/* table1Record.rels.value => [
    { id: 1, ref: 1 },
    { id: 2, ref: 1 }
]; */

// Table2 records has the "ref" property.
Table2.get(1).ref.value.body === "some text";

---

// Since we have defined the relationName "rels" on the "Table2.ref" field, the␣
↪following insert is equivalent to the first two.
Table1.insert({
    id: 1,
    body: "some text",
    rels: [
        { id: 1, ref: 1 },
        { id: 2, ref: 1 },
        { id: 3, ref: 2 }
    ]
});
```

3.5 RecordSetModel

The RecordSetModel class wraps a relation between two tables.

Methods and propertes:

```
/// gets the primary keys of the records in the set.
ids : string[];

/// gets the raw array of record values.
value : any[];

/// gets the number of records in the set.
length : number;

/// returns all records in the set.
all() : RecordModel[];

/// updates the records with new data.
update( data: any ) : this;
```

(continues on next page)

(continued from previous page)

```

/// deletes all records in the set.
delete() : void;
```

CHAPTER 4

Typescript

redux-db comes with a set of interfaces that you may use to define a strongly typed session.

```
import * as ReduxDB from "redux-db";

// Given the schema
export const schema = {
    "Project": {
        "id": { type: "PK" }
    },
    "Task": {
        "id": { type: "PK" },
        "projectId": { propName: "project", references: "Project", relationName:
↪"tasks" }
    }
}

// Data models
export interface Project {
    id: number,
    title: string;
}
export interface Task {
    id: number;
    projectId: number;
}

// Schema models
export interface ProjectRecord extends ReduxDB.TableRecord<Project> {
    tasks: ReduxDB.TableRecordSet<TaskRecord>;
}

export interface TaskRecord extends ReduxDB.TableRecord<Task> {
    project: ProjectRecord;
}
```

(continues on next page)

(continued from previous page)

```
export interface Session {
    Project: ReduxDB.Table<Project, ProjectRecord>;
    Task: ReduxDB.Table<Task, TaskRecord>;
}

// Reducer
export const dbReducer = (session: Session, action: { type: string, payload: any }) =>
↪ {
    const { Project, Task } = session;

    switch (action.type) {
        case "SOME_ACTION": {
            const project = Project.get(action.payload);
            // project.tasks === typeof RecordSet<TaskRecord>
            // project.tasks.map( t=> t.value === typeof Task )
            break;
        }
    }
}
```

CHAPTER 5

Advanced features

5.1 Custom model factory

You may provide your own implementation of the model factory used by redux-db. This allows you to extend the functionalliy each model provides.

Here is an example of a model factory extending the default models.

```
import { DefaultModelFactory, TableModel, RecordModel } from "redux-db";
import { schema } from "./schema";

class CustomTableModel extends TableModel {
    constructor(session, state, schema) {
        super(session, state, schema);
    }

    // Override delete operation to add logging to console.
    delete(id) {
        const deleted = super.delete(id);
        if (deleted)
            console.info(`Record "${this.schema.name}[${id}]" deleted.`);
        else
            console.warn(`Record "${this.schema.name}[${id}]" not deleted.`);
        return deleted;
    }
}

class CustomRecordModel extends RecordModel {
    constructor(id, table) {
        super(id, table);
    }

    // Override toString to give a JSON representation of record value.
    toString() {
```

(continues on next page)

(continued from previous page)

```
        return JSON.stringify(this.value);
    }
}

class CustomModelFactory extends DefaultModelFactory {
    getRecordBaseClass(schema) {
        return CustomRecordModel;
    }

    newTableModel(session, state, schema) {
        return new CustomTableModel(session, state, schema);
    }
}

export const db = createDatabase(schema, {
    factory: new CustomModelFactory()
});
```

CHAPTER 6

Defining your schema

To make redux-db normalize your nested data, you must define a schema. Given the following data:

```
const blogPosts = [
    {
        id: "post1",
        author: { username: "user1", name: "User 1" },
        body: "......",
        comments: [
            {
                id: "comment1",
                author: { username: "user2", name: "User 2" },
                comment: "....."
            },
            {
                id: "comment2",
                author: { username: "user3", name: "User 3" },
                comment: "....."
            }
        ]
    },
    {
        id: "post2",
        author: { username: "user2", name: "User 2" },
        body: "......",
        comments: [
            {
                id: "comment3",
                author: { username: "user1", name: "User 1" },
                comment: "....."
            }
        ]
    }
]
```

You would define a schema like so:

```
/// schema.js

import * as ReduxDB from "redux-db";

const schema = {
    User: {
        username: { type: "PK" }
    },
    BlogPost: {
        id: { type: "PK" },
        author: { references: "User", relationName: "posts" }
    },
    Comment: {
        id: { type: "PK" },
        post: { references: "BlogPost", relationName: "comments" },
        author: { references: "User", relationName: "comments" }
    }
};

export const db = ReduxDB.createDatabase(schema);
```

Note you may only define foreign and primary keys. The data fields like "User.name" and "Comment.comment" are not needed in the schema.

Using this schema definition, the example data would be normalized out in the following manner:

```
const normalizedData = {
    User: {
        ids: ["user1", "user2", "user3"],
        byId: {
            "user1": {
                username: "user1",
                name: "User 1"
            },
            "user2": {
                username: "user2",
                name: "User 2"
            }
        }
    },
    BlogPost: {
        ids: ["post1", "post2"],
        byId: {
            "post1": {
                id: "post1",
                author: "user1",
                body: "....."
            },
            "post2": {
                id: "post2",
                author: "user2",
                body: "....."
            }
        }
    },
    Comment: {
        ids: ["comment1", "comment3"],
        byId: {
```

(continues on next page)

(continued from previous page)

```
            "comment1": {
                id: "comment1",
                post: "post1",
                author: "user2",
                comment: "....."
            },
            "comment3": {
                id: "comment3",
                post: "post2",
                author: "user1",
                comment: "....."
            }
        }
    }
}
```

CHAPTER 7

Applying reducers

When you have your schema set up it's time to write your reducers. redux-db provides a simple function to combine multiple reducers to work on the normalized state.

```
/// reducer.js

import ReduxDB from "redux-db";
import db from "./schema";

export const dbReducer = db.combineReducers(
    (session, action) => {
        const { BlogPost, Comment, User } = session;

        switch (action.type) {
            case "POSTS_FETCH_DONE":
            case "POST_FETCH_DONE":
            case "POST_UPDATE":
            case "POST_INSERT":
                // all these actions may be handled using just one statement.
                // the upsert method accepts both single objects and arrays.

                BlogPost.upsert(action.payload);
                break;

            case "POST_DELETE": {
                const { id } = action.payload;
                const post = BlogPost.get(id);

                post.comments.delete();
                post.delete();
                break;
            }
            case "COMMENT_UPDATE":
            case "COMMENT_INSERT": {
                // assuming payload contains {id,post,author}
```

(continues on next page)

(continued from previous page)

```
                const { post } = action.payload;

                BlogPost.get(post).comments.add(action.payload);
                // or just, Comment.upsert(action.payload);
                break;
            }
            case "COMMENT_DELETE": {
                const { id } = action.payload;
                Comment.delete(id);
                break;
            }
        }
    }
);
```

CHAPTER 8

Configuring the store

After defining your schema and setting up your reducer(s). You'll need to configure the redux store.

```
/// store.js

import { createStore, combineReducers } from "redux";
import dbReducer from "./reducer";

export const store = createStore(
    combineReducers({
        db: dbReducer
        /* other reducers */
    })
);
```

The name for the redux-db state tree is up to you, but "db" suits well.

CHAPTER 9

Connect your components

As the state tree is now normalized you are likely to denormalize your data for your views. Example given in react:

```
import { Component } from "react";
import { connect } from "react-redux";
import { db } from "./schema";

class PostListComponent extends Component {
    render() {
        return <table>
            <tbody>
                {this.props.posts.map(post => (
                    <tr>
                        <td>{post.title}</td>
                        <td>{post.author}</td>
                        <td>{post.numComments}</td>
                    </tr>
                ) }
            </tbody>
        </table>;
    }
}

const mapStateToProps = (state, ownProps) => {
    const { BlogPost } = db.selectTables(state.db);

    return {
        posts: BlogPost.all().map(post => ({
            ...post.value,
            numComments: post.comments.length,
            author: post.author.value.name
        }))
    };
};

export const PostList = connect(mapStateToProps)(PostListComponent);
```

Warning: This is all well and good, but as your state tree and application grows you should definitely switch to using memoized selectors (eg. reselect).

CHAPTER 10

Improving performance with selectors

Example using reselect:

```
/// selectors.js

import { db as myDb } from './schema';
import { createSelector, createStructuredSelector } from 'reselect';

// Single table, all records, no relations
export const selectAllPosts = createSelector(
    ({ db }) => db.BlogPost,
    (table) => {
        return myDb.selectTable(table).values; // NB. should be converted to a view␣
↪model.
    }
);

// using a structured selector, makes us select only the tables we want
const blogPostAndRels = createStructuredSelector({
    posts: ({ db }) => db.BlogPost,
    comments: ({ db }) => db.Comment,
    users: ({ db }) => db.User
});

// multiple complex models
export const selectAllPosts = createSelector(
    blogPostAndRels,
    (tables) => {
        // get the TableModel
        const { BlogPost } = myDb.selectTables(tables);

        return BlogPost.all().map(_serializePost);
    }
);
```

(continues on next page)

(continued from previous page)

```
// single complex model by id
export const selectPost = createSelector(
    blogPostAndRels,
    (state, props) => props.postId,
    (tables, id) => {
        // get the TableModel
        const { BlogPost } = myDb.selectTables(tables);

        // get the Record by id
        const postModel = BlogPost.getOrDefault(id);

        return postModel && _serializePost(postModel);
    }
);

const _serializePost = (post) => {
    return {
        ...post.value,
        numComments: post.comments.length,
        author: post.author.value.name
    };
};
```

Using selectors we can improve the previous example:

```
import { Component } from "react";
import { connect } from "react-redux";
import { db } from "./schema";
import { selectAllPosts } from "./selectors";

class PostListComponent extends Component {
    render() {
        return <table>
            <tbody>
                {this.props.posts.map(post => (
                    <tr>
                        <td>{post.title}</td>
                        <td>{post.author}</td>
                        <td>{post.numComments}</td>
                    </tr>
                ) }
            </tbody>
        </table>;
    }
}

const mapStateToProps = (state, ownProps) => {
    return {
        posts: selectAllPosts( state )
    };
};

export const PostList = connect(mapStateToProps)(PostListComponent);
```