tintoa-data-service

Table of Contents *generated with DocToc*

News

Version 2.0.0 is out now! It comes with the new MongoStore.

Installation

```
> npm install tintoa-data-service --save
```

Basic Usage

To handle data with tintoa-data-service is very simple. You just need a StoreType and 3 Methods.

DataService.save : StoreResult

The Code below will create new data.

```
const service = new DataService("BaseData", DataService.StoreTypes.File);
            let result = await service.save({
                context: "User",
                data: {
                    name: "Tony Stark",
                    age: 45,
                    job: "Engineer"
                }
            });
```

For updating, you need to specify the id of the existing data.

```
            await service.save({
                context: "User",
                id: (<StoreData>result.first()).id
                data: {
                    name: "Tony Stark",
                    age: 45,
                    job: "Iron Man"
                }
            });
```

DataService.load : StoreResult

Specifying the context only, the service will do a `loadAll` and returns all availible entries.

```
            let data = await service.load({
                context: "User",
            });
```

Passing the id of an existing data-object as property, the data-service will load it.

```
            let ironMan = await service.load({
                context: "User",
                id: (<StoreData>result.first()).id
            });
```

Query an entry is easy the same way - just pass the `query` object. This will return all entries that match the given query.

```
            let ironMan = await service.load({
                context: "User",
                query: { age: 45, job: "Iron Man" }
            });
```

NOTE: If you pass an id **and** a query object, the query will always be ignored and the service will load the object with the given id.

DataService.delete : StoreResult

To delete data, call the delete method and pass the id of the object to delete. Passing an array of ids will delete multiple entries at once.

```
            let ironMan = await service.delete({
                context: "User",
                id: (<StoreData>result.first()).id
});
```

enum DataService.StoreTypes

The current version supports 2 types of stores. For more information read the chapter Stores

```
1. File-Storage:

All the data will be saved in local files. This is a great way if you have no
database connection or you are running some tests. You also can create log files
or something else.

2. MongoDB-Storage

This will allow you to connect to a mongo-server. Its up to you to setup this
server.
You just need to pass some connection options to the DataService instance.
```

interface DataService.DS_Settings

Config name	Description
host?: string;	The host where your mongo-server is running at.
port?: number;	The port the server listens to.
user?: string;	The name of the database user
password?: string;	The user password

Config name	Description
db_path?: string;	If you chose the FileStore, you can pass a root path for its rootDir (default:
encode?: boolean;	If you set this to false, all files and all content will not be encrypted.
jsonExtension?: boolean;	This option will let you add an json extension to the files (default: false - makes only sence if you disabled the encoding)<>

StoreResult

Every DataService method will resolve a StoreResult object. It contains information about the CRUD process, an error message if something went wrong and an array with result data. It also provides some methods to work with data.

Method / Property	Type / Return type	Description
success	boolean	`true` if the operation resolved, else `false`
message	string	Is empty if there is no error
data	StoreData.DataArray	Contains the data
`get` count	number	returns the number of data entries
toArray()	StoreData.DataObject[]	This will return an array with the plain data. No information about id or something else
first(query?KeyValues)	StoreData | undefined	This will query the data array until a matching entry is found. If no query is defined, it will return the first element of the data array. It returns undefined if data is empty.
toData()	IdData	Returns an object where each data is assigned to its id

Storable

Beside using an instance of a data-service directly, you can define `storable classes`. This works only if you use TypeScript with the decorator-factory to make things happen.

```
import { DataService, Storable } from 'tintoa-data-service';

const BaseData = new DataService("BaseData", DataService.StoreTypes.File);

@BaseData.Entity()
class User extends Storable{
    @BaseData.Property()
    public name?: string;
    @BaseData.Property()
    public age?: number;
}

let user = new User();
user.name = "Tony Stark";
user.age = 45;

await user.save();
```

For loading the objects, the Storable class has the static methods load, loadAll and find.

```
let allUsers : User[] = User.loadAll<User>(); // returns an array with all found users;
let Tony = await User.find<User>({ name: "Tony Stark" }); // returns a single User instance.
```

Credentials

Credentials are the way to make your saved data private. To implement this, you have to set the `owner` property, when you are saving the data.

```
            await service.save({
                context: "Secret",
                data: {
                    content: "I Love Beer."
                },
                owner: "data-service developer"
            });
```

When loading the data, you will find this entry only by passing the needed credentials.

```
import { Credentials } from 'tintoa-data-service'

            await service.load({
                context: "Secret",
                credentials: new Credentials("data-service developer")
            });
```

Stores

FileStore

All created data will be stored in a local files. With the `DS_Settings.db_path` option, you can setup a root directory for the dataservice. The default is your systems temporary directory. If you use the setting, the dataservice will create a folder `.dataservice` at the given location. Each dataservice instance can have its own location. On instantiaion, it will create a directory with the given dataservice-identifier in the db_path directory. Saving with a new contex will creating a new file in the instance-directory.

Imagine the following example

```
const settings: DataService.DS_Settings = {
    db_path: "~/dev",
    encode: false,
    jsonExtension: true
}

const BaseData = new DataService("BaseData", DataService.StoreTypes.File,
settings);
const UserData = new DataService("UserData", DataService.StoreTypes.File,
settings);
const AppData = new DataService("AppData", DataService.StoreTypes.File, settings);

await BaseData.save({context: "Config", data: {host: "localhost", port: 9872}});
await UserData.save({context: "Employee", data: {name: "MrRabbit",age: 77,job:
"Developer"}});
await UserData.save({context: "External", data: {name: "Schorsch",age: 11,job:
"Senior-Developer"}});
await AppData.save({context: "Usage", data: {registratedUsers: 2}});
```

The code above will lead to the following structure:

```
~/dev:
    |- .dataservice
        |- BaseData
            |- Config.json
        |- UserData
            |- Employee.json
            |- External.json
        |- AppData
            |- Usage.json
```

Now lets have a look at the settings, passed to the constructor:

- `db_path: "~/dev"`, as already mentioned, specifies where to create the root directory.
- `encode: false` prevents the encoding of the data with an `AES192` algorithm and all the data will be saved in plain text. This may be useful for debugging or writing logfiled.

- `jsonExtension: true` is helpful when you disabled the data-encoding. It adds the extension '.json' to all files that your editor can deal with it.

MongoStore

As the name says, it will save your data in a mongo database. For securing things, be sure to setup authentication for your database server! To setup things correctly, you have to provide the following settings:

```
let settings : DataService.DS_Settings = {
        "host": "HOST-TO-YOUR-MONGO-SERVER",
        "port": 27017,
        "user": "USER-WITH-READ-AND-WRITE-PRIVILEGES",
        "password": "TOPSECRET"
}
```

The MonogStore Will use the same structure as the FileStore. A Dataservice instance creates a db and each context is a mongo collection.