Serializr

Serialize and deserialize complex object graphs to JSON

Serializr is feature complete, and easily extendable. Since there are no active maintainers the project is frozen feature wise. Bug reports and well designed pull requests are welcome and will be addressed.

Want to maintain a small open source project or having great ideas for this project? We are looking for maintainers, so apply!

Introduction

Serializr is a utility library that helps converting json structures into complex object graphs and the other way around. For a quick overview, read the introduction blog post

Features:

- (De)serialize objects created with a constructor / class
- (De)serialize primitive values
- (De)serialize nested objects, maps and arrays
- Resolve references asynchronously (during deserialization)
- Supports inheritance
- Works on any ES5 environment (if ES3 is needed file a feature request)
- Convenience decorators for ESNext / Typescript
- Ships with typescript / flow typings
- Generic solution that works well with for example MobX out of the box

Non-features:

- Serializr is not an ORM or data management library. It doesn't manage object instances, provided api's like fetch, search etc. If you are building such a thing though, serializr might definitely take care of the serialization part for you 😃.
- Serializr is not a MobX specific (de)serialization mechanism, it is generic and should fit work with any type of model objects

Installation

From npm: `npm install serializr --save`

From CDN: https://unpkg.com/serializr which declares the global `serializr` object.

Quick example:

```
import {
    createModelSchema,
    primitive,
    reference,
    list,
    object,
    identifier,
    serialize,
    deserialize,
} from 'serializr';

// Example model classes
class User {
    uuid = Math.floor(Math.random() * 10000);
    displayName = 'John Doe';
}

class Message {
    message = 'Test';
    author = null;
    comments = [];
}

function fetchUserSomewhere(uuid) {
    // Lets pretend to actually fetch a user; but not.
    // In a real app this might be a database query
    const user = new User();
    user.uuid = uuid;
    user.displayName = `John Doe ${uuid}`;
    return user;
}

function findUserById(uuid, callback, context) {
    // This is a lookup function
    // uuid is the identifier being resolved
    // callback is a node style callback function to be invoked with the found
object (as second arg) or an error (first arg)
    // context is an object detailing the execution context of the serializer now
    callback(null, fetchUserSomewhere(uuid));
}

// Create model schemas
createModelSchema(Message, {
    message: primitive(),
    author: reference(User, findUserById),
    comments: list(object(Message)),
});

createModelSchema(User, {
    uuid: identifier(),
    displayName: primitive(),
});
```

```
// can now deserialize and serialize!
const message = deserialize(Message, {
    message: 'Hello world',
    author: 17,
    comments: [
        {
            message: 'Welcome!',
            author: 23,
        },
    ],
});

const json = serialize(message);

console.dir(message, { colors: true, depth: 10 });
```

Using decorators (optional)

With decorators (TypeScript or ESNext) building model schemas is even more trivial:

```
import {
    createModelSchema,
    primitive,
    reference,
    list,
    object,
    identifier,
    serialize,
    deserialize,
    getDefaultModelSchema,
    serializable,
} from 'serializr';

class User {
    @serializable(identifier())
    uuid = Math.random();

    @serializable displayName = 'John Doe';
}

class Message {
    @serializable message = 'Test';

    @serializable(object(User))
    author = null;

    // Self referencing decorators work in Babel 5.x and Typescript. See below for
more.
    @serializable(list(object(Message)))
    comments = [];
}
```

```
// You can now deserialize and serialize!
const message = deserialize(Message, {
    message: 'Hello world',
    author: { uuid: 1, displayName: 'Alice' },
    comments: [
        {
            message: 'Welcome!',
            author: { uuid: 1, displayName: 'Bob' },
        },
    ],
});

console.dir(message, { colors: true, depth: 10 });

// We can call serialize without the first argument here
//because the schema can be inferred from the decorated classes

const json = serialize(message);
```

Decorator: Caveats

Babel 6.x does not allow decorators to self-reference during their creation, so the above code would not work for the Message class. Instead write:

```
class Message {
    @serializable message = 'Test';

    @serializable(object(User))
    author = null;

    comments = [];

    constructor() {
        getDefaultModelSchema(Message).props['comments'] = list(
            object(Message)
        );
    }
}
```

Enabling decorators (optional)

TypeScript

Enable the compiler option `experimentalDecorators` in `tsconfig.json` or pass it as flag `--experimentalDecorators` to the compiler.

Babel 7.x:

Install support for decorators: `npm i --save-dev @babel/plugin-proposal-class-properties @babel/plugin-proposal-decorators`. And enable it in your `.babelrc` file:

```
{
    "presets": ["@babel/preset-env"],
    "plugins": [
        ["@babel/plugin-proposal-decorators", { "legacy": true}],
        ["@babel/plugin-proposal-class-properties", { "loose": true}]
    ]
}
```

Babel 6.x:

Install support for decorators: `npm i --save-dev babel-plugin-transform-decorators-legacy`. And enable it in your `.babelrc` file:

```
{
    "presets": ["es2015", "stage-1"],
    "plugins": ["transform-decorators-legacy"]
}
```

Babel 5.x

```
 {
   "stage": 1
 }
```

Probably you have more plugins and presets in your `.babelrc` already, note that the order is important and `transform-decorators-legacy` should come as first.

Concepts

The two most important functions exposed by serializr are `serialize(modelschema?, object) -> json tree` and `deserialize(modelschema, json tree) -> object graph`. What are those model schemas?

ModelSchema

The driving concept behind (de)serialization is a ModelSchema. It describes how model object instances can be (de)serialize to json.

A simple model schema looks like this:

```
const todoSchema = {
    factory: context => new Todo(),
```

```
    extends: ModelSchema,
    props: {
        modelfield: PropSchema,
    },
};
```

The `factory` tells how to construct new instances during deserialization. The optional `extends` property denotes that this model schema inherits its props from another model schema. The props section describes how individual model properties are to be (de)serialized. Their names match the model field names. The combination `fieldname: true` is simply a shorthand for `fieldname: primitive()`

For convenience, model schemas can be stored on the constructor function of a class. This allows you to pass in a class reference wherever a model schema is required. See the examples below.

PropSchema

PropSchemas contain the strategy on how individual fields should be serialized. It denotes whether a field is a primitive, list, whether it needs to be aliased, refers to other model objects etc. PropSchemas are composable. See the API section below for the details, but these are the built-in property schemas:

- `primitive()`: Serialize a field as primitive value
- `identifier()`: Serialize a field as primitive value, use it as identifier when serializing references (see `reference`)
- `date()`: Serializes dates (as epoch number)
- `alias(name, propSchema)`: Serializes a field under a different name
- `list(propSchema)`: Serializes an array based collection
- `map(propSchema)`: Serializes an Map or string key based collection
- `mapAsArray(propSchema, keyPropertyName)`: Serializes a map to an array of elements
- `object(modelSchema)`: Serializes an child model element
- `reference(modelSchema, lookupFunction?)`: Serializes a reference to another model element
- `custom(serializeFunction, deserializeFunction)`: Create your own property serializer by providing two functions, one that converts modelValue to jsonValue, and one that does the inverse
- There is a special prop schema: `"*": true` that serializes all enumerable, non mentioned values as primitive

It is possible to define your own prop schemas. You can define your own propSchema by creating a function that returns an object with the following signature:

```
{
    serializer: (sourcePropertyValue: any) => jsonValue,
    deserializer: (jsonValue: any, callback: (err, targetPropertyValue: any) =>
void, context?, currentPropertyValue?) => void
}
```

For inspiration, take a look at the source code of the existing ones on how they work, it is pretty straightforward.

Deserialization context

The context object is an advanced feature and can be used to obtain additional context-related information about the deserialization process. `context` is available as:

1. first argument of factory functions
2. third argument of the lookup callback of `ref` prop schema's (see below)
3. third argument of the `deserializer` of a custom propSchema

When deserializing a model elememt / property, the following fields are available on the context object:

- `json`: Returns the complete current json object that is being deserialized
- `target`: The object currently being deserialized. This is the object that is returned from the factory function.
- `parentContext`: Returns the parent context of the current context. For example if a child element is being deserialized, the `context.target` refers to the current model object, and `context.parentContext.target` refers to the parent model object that owns the current model object.
- `args`: If custom arguments were passed to the `deserialize` / `update` function, they are available as `context.args`.

AdditionalPropArgs

A PropSchema can be further parameterized using AdditionalPropArgs. Currently, they can be used to specify lifecycle functions. During deserialization they can be useful, e.g. in case you want to

- react to errors in the deserialization on a value level and retry with corrected value,
- remove invalid items e.g. in arrays or maps,
- react to changes in field names, e.g. due to schema migration (i.e. only one-directional changes that cannot be dealt with by alias operators).

It is possible to define those functions by passing them as additional property arguments to the propSchema during its creation.

```
const myHandler = {
  beforeDeserialize: function (callback, jsonValue, jsonParentValue,
propNameOrIndex, context, propDef) {
     if (typeof jsonValue === 'string') {
       callback(null, jsonValue)
     } else if (typeof jsonValue === 'number') {
       callback(null, jsonValue.toString())
     } else {
       callback(new Error('something went wrong before deserialization'))
     }
  },
  afterDeserialize: function (callback, error, newValue, jsonValue,
jsonParentValue, propNameOrIndex, context,
                                                                propDef,
numRetry) {
     if (!error && newValue !== 'needs change') {
```

```
        callback(null, newValue)
      } else if (!error && newValue === 'needs change') {
        callback(new Error(), 'changed value')
      } else {
        callback(error)
      }
    }
  }

  class MyData {
    @serializable(primitive(myHandler)) mySimpleField
  }
```

A more detailed example can be found in test/typescript/ts.ts.

API

Table of Contents

ModelSchema

src/serializr.js:52-52

JSDOC type defintions for usage w/o typescript.

Type: object

Parameters

- `value` **any**
- `writeable` **boolean**
- `get` **(Function | undefined)**

- `set` (**Function** | **undefined**)
- `configurable` **boolean**
- `enumerable` **boolean**
- `sourcePropertyValue` **any**
- `jsonValue` **any**
- `callback` **cpsCallback**
- `context` **Context**
- `currentPropertyValue` **any**
- `id` **any**
- `target` **object**
- `context` **Context**
- `result` **any**
- `error` **any**
- `id` **string**
- `callback` **cpsCallback**
- `factory`
- `props`
- `targetClass`

Properties

- `serializer` **serializerFunction**
- `deserializer` **deserializerFunction**
- `identifier` **boolean**

Returns **any** any - serialized object

Returns **any** void

Returns **any** void

Returns **any** void

createSimpleSchema

src/api/createSimpleSchema.js:17-24

Creates a model schema that (de)serializes from / to plain javascript objects. Its factory method is: `() => ({})`

Parameters

- `props` **object** property mapping,

Examples

```
var todoSchema = createSimpleSchema({
    title: true,
    done: true,
```

```
});

var json = serialize(todoSchema, { title: 'Test', done: false });
var todo = deserialize(todoSchema, json);
```

Returns **object** model schema

createModelSchema

src/api/createModelSchema.js:29-47

Creates a model schema that (de)serializes an object created by a constructor function (class). The created model schema is associated by the targeted type as default model schema, see setDefaultModelSchema. Its factory method is `() => new clazz()` (unless overriden, see third arg).

Parameters

- `clazz` **(constructor | class)** class or constructor function
- `props` **object** property mapping
- `factory` **function** optional custom factory. Receives context as first arg

Examples

```
function Todo(title, done) {
    this.title = title;
    this.done = done;
}

createModelSchema(Todo, {
    title: true,
    done: true,
});

var json = serialize(new Todo('Test', false));
var todo = deserialize(Todo, json);
```

Returns **object** model schema

getDefaultModelSchema

src/api/getDefaultModelSchema.js:9-18

Returns the standard model schema associated with a class / constructor function

Parameters

- `thing` **object**

Returns **ModelSchema** model schema

setDefaultModelSchema

src/api/setDefaultModelSchema.js:15-18

Sets the default model schema for class / constructor function. Everywhere where a model schema is required as argument, this class / constructor function can be passed in as well (for example when using `object` or `ref`.

When passing an instance of this class to `serialize`, it is not required to pass the model schema as first argument anymore, because the default schema will be inferred from the instance type.

Parameters

- `clazz` **(constructor | class)** class or constructor function
- `modelSchema` **ModelSchema** a model schema

Returns **ModelSchema** model schema

serializable

src/api/serializable.js:93-103

Decorator that defines a new property mapping on the default model schema for the class it is used in.

When using typescript, the decorator can also be used on fields declared as constructor arguments (using the `private` / `protected` / `public` keywords). The default factory will then invoke the constructor with the correct arguments as well.

Parameters

- `arg1`
- `arg2`
- `arg3`

Examples

```
class Todo {
```

Returns **PropertyDescriptor**

serialize

src/core/serialize.js:14-36

Serializes an object (graph) into json using the provided model schema. The model schema can be omitted if the object type has a default model schema associated with it. If a list of objects is provided, they should have an uniform type.

Parameters

- `arg1` class or modelschema to use. Optional
- `arg2` object(s) to serialize

Returns **object** serialized representation of the object

serializeAll

src/core/serializeAll.js:38-67

The `serializeAll` decorator can may used on a class to signal that all primitive properties, or complex properties with a name matching a `pattern`, should be serialized automatically.

Parameters

- `targetOrPattern`
- `clazzOrSchema`

Examples

```
class DataType {
```

cancelDeserialize

src/core/cancelDeserialize.js:12-18

Cancels an asynchronous deserialization or update operation for the specified target object.

Parameters

- `instance` object that was previously returned from deserialize or update method

deserialize

src/core/deserialize.js:69-87

Deserializes a json structure into an object graph.

This process might be asynchronous (for example if there are references with an asynchronous lookup function). The function returns an object (or array of objects), but the returned object might be incomplete until the callback has fired as well (which might happen immediately)

Parameters

- `schema` **(object | array)** to use for deserialization
- `json` **json** data to deserialize
- `callback` **function** node style callback that is invoked once the deserialization has finished. First argument is the optional error, second argument is the deserialized object (same as the return value)
- `customArgs` **any** custom arguments that are available as `context.args` during the deserialization process. This can be used as dependency injection mechanism to pass in, for example, stores.

Returns **(object | array)** deserialized object, possibly incomplete.

update

src/core/update.js:22-44

Similar to deserialize, but updates an existing object instance. Properties will always updated entirely, but properties not present in the json will be kept as is. Further this method behaves similar to deserialize.

Parameters

- `modelSchema` **object** , optional if it can be inferred from the instance type
- `target` **object** target instance to update
- `json` **object** the json to deserialize
- `callback` **function** the callback to invoke once deserialization has completed.
- `customArgs` **any** custom arguments that are available as `context.args` during the deserialization process. This can be used as dependency injection mechanism to pass in, for example, stores.

Returns **(object | array)** deserialized object, possibly incomplete.

primitive

src/types/primitive.js:18-32

Indicates that this field contains a primitive value (or Date) which should be serialized literally to json.

Parameters

- `additionalArgs` **AdditionalPropArgs** optional object that contains beforeDeserialize and/or afterDeserialize handlers

Examples

```
createModelSchema(Todo, {
    title: primitive(),
});

console.dir(serialize(new Todo('test')));
// outputs: { title : "test" }
```

Returns **ModelSchema**

identifier

src/types/identifier.js:43-66

Similar to primitive, but this field will be marked as the identifier for the given Model type. This is used by for example `reference()` to serialize the reference

Identifier accepts an optional `registerFn` with the signature: `(id, target, context) => void` that can be used to register this object in some store. note that not all fields of this object might have been deserialized yet.

Parameters

- `arg1` **(RegisterFunction | AdditionalPropArgs)** optional registerFn: function to register this object during creation.
- `arg2` **AdditionalPropArgs** optional object that contains beforeDeserialize and/or afterDeserialize handlers

Examples

```
var todos = {};

var s = _.createSimpleSchema({
    id: _.identifier((id, object) => (todos[id] = object)),
    title: true,
});

_.deserialize(s, {
    id: 1,
    title: 'test0',
});
_.deserialize(s, [{ id: 2, title: 'test2' }, { id: 1, title: 'test1' }]);

t.deepEqual(todos, {
    1: { id: 1, title: 'test1' },
    2: { id: 2, title: 'test2' },
});
```

Returns **PropSchema**

date

src/types/date.js:9-26

Similar to primitive, serializes instances of Date objects

Parameters

- `additionalArgs` **AdditionalPropArgs** optional object that contains beforeDeserialize and/or afterDeserialize handlers

Returns **PropSchema**

alias

src/types/alias.js:20-33

Alias indicates that this model property should be named differently in the generated json. Alias should be the outermost propschema.

Parameters

- `name` **string** name of the json field to be used for this property
- `propSchema` **PropSchema** propSchema to (de)serialize the contents of this field

Examples

```
createModelSchema(Todo, {
    title: alias('task', primitive()),
});

console.dir(serialize(new Todo('test')));
// { task : "test" }
```

Returns **PropSchema**

custom

src/types/custom.js:60-75

Can be used to create simple custom propSchema. Multiple things can be done inside of a custom propSchema, like deserializing and serializing other (polymorphic) objects, skipping the serialization of something or checking the context of the obj being (de)serialized.

The `custom` function takes two parameters, the `serializer` function and the `deserializer` function.

The `serializer` function has the signature: `(value, key, obj) => void`

When serializing the object `{a: 1}` the `serializer` function will be called with `serializer(1, 'a', {a: 1})`.

The `deserializer` function has the following signature for synchronous processing `(value, context, oldValue) => void`

For asynchronous processing the function expects the following signature `(value, context, oldValue, callback) => void`

When deserializing the object `{b: 2}` the `deserializer` function will be called with `deserializer(2, contextObj)` (contextObj reference).

Parameters

- `serializer` **function** function that takes a model value and turns it into a json value
- `deserializer` **function** function that takes a json value and turns it into a model value. It also takes context argument, which can allow you to deserialize based on the context of other parameters.
- `additionalArgs` **AdditionalPropArgs** optional object that contains beforeDeserialize and/or afterDeserialize handlers

Examples

```
var schemaDefault = _.createSimpleSchema({
    a: _.custom(
        function(v) {
            return v + 2;
        },
        function(v) {
            return v - 2;
        }
    ),
});
t.deepEqual(_.serialize(schemaDefault, { a: 4 }), { a: 6 });
t.deepEqual(_.deserialize(schemaDefault, { a: 6 }), { a: 4 });

var schemaWithAsyncProps = _.createSimpleSchema({
    a: _.customAsync(
        function(v) {
            return v + 2;
        },
        function(v, context, oldValue, callback) {
            somePromise(v, context, oldValue).then((result) => {
                callback(null, result - 2)
            }.catch((err) => {
                callback(err)
            }
        }
    ),
});
t.deepEqual(_.serialize(schemaWithAsyncProps, { a: 4 }), { a: 6 });
_.deserialize(schemaWithAsyncProps, { a: 6 }, (err, res) => {
  t.deepEqual(res.a, 4)
};
```

Returns **PropSchema**

object

src/types/object.js:35-55

`object` indicates that this property contains an object that needs to be (de)serialized using its own model schema.

N.B. mind issues with circular dependencies when importing model schema's from other files! The module resolve algorithm might expose classes before `createModelSchema` is executed for the target class.

Parameters

- `modelSchema` **ModelSchema** to be used to (de)serialize the object
- `additionalArgs` **AdditionalPropArgs** optional object that contains beforeDeserialize and/or afterDeserialize handlers

Examples

```
class SubTask {}
class Todo {}

createModelSchema(SubTask, {
    title: true,
});
createModelSchema(Todo, {
    title: true,
    subTask: object(SubTask),
});

const todo = deserialize(Todo, {
    title: 'Task',
    subTask: {
        title: 'Sub task',
    },
});
```

Returns **PropSchema**

optional

src/types/optional.js:18-31

Optional indicates that this model property shouldn't be serialized if it isn't present.

Parameters

- `name`
- `propSchema` **PropSchema** propSchema to (de)serialize the contents of this field

Examples

```
createModelSchema(Todo, {
    title: optional(primitive()),
});

console.dir(serialize(new Todo()));
// {}
```

Returns **PropSchema**

reference

src/types/reference.js:66-105

`reference` can be used to (de)serialize references that point to other models.

The first parameter should be either a ModelSchema that has an `identifier()` property (see identifier) or a string that represents which attribute in the target object represents the identifier of the object.

The second parameter is a lookup function that is invoked during deserialization to resolve an identifier to an object. Its signature should be as follows:

`lookupFunction(identifier, callback, context)` where: 1. `identifier` is the identifier being resolved 2. `callback` is a node style calblack function to be invoked with the found object (as second arg) or an error (first arg) 3. `context` see context.

The lookupFunction is optional. If it is not provided, it will try to find an object of the expected type and required identifier within the same JSON document

N.B. mind issues with circular dependencies when importing model schemas from other files! The module resolve algorithm might expose classes before `createModelSchema` is executed for the target class.

Parameters

- `target` : ModelSchema or string
- `lookupFn` **(RefLookupFunction | AdditionalPropArgs)** optional function or additionalArgs object
- `additionalArgs` **AdditionalPropArgs** optional object that contains beforeDeserialize and/or afterDeserialize handlers

Examples

```
class User {}
class Post {}

createModelSchema(User, {
    uuid: identifier(),
    displayname: primitive(),
});

createModelSchema(Post, {
    author: reference(User, findUserById),
    message: primitive(),
});

function findUserById(uuid, callback) {
    fetch('http://host/user/' + uuid)
        .then(userData => {
            deserialize(User, userData, callback);
        })
        .catch(callback);
}

deserialize(
    Post,
    {
        message: 'Hello World',
        author: 234,
```

```
    },
    (err, post) => {
        console.log(post);
    }
);
```

Returns **PropSchema**

list

src/types/list.js:43-105

List indicates that this property contains a list of things. Accepts a sub model schema to serialize the contents

Parameters

- `propSchema` **PropSchema** to be used to (de)serialize the contents of the array
- `additionalArgs` **AdditionalPropArgs** optional object that contains beforeDeserialize and/or afterDeserialize handlers

Examples

```
class SubTask {}
class Task {}
class Todo {}

createModelSchema(SubTask, {
    title: true,
});
createModelSchema(Todo, {
    title: true,
    subTask: list(object(SubTask)),
});

const todo = deserialize(Todo, {
    title: 'Task',
    subTask: [
        {
            title: 'Sub task 1',
        },
    ],
});
```

Returns **PropSchema**

map

src/types/map.js:14-65

Similar to list, but map represents a string keyed dynamic collection. This can be both plain objects (default) or ES6 Map like structures. This will be inferred from the initial value of the targetted attribute.

Parameters

- `propSchema` **any**
- `additionalArgs` **AdditionalPropArgs** optional object that contains beforeDeserialize and/or afterDeserialize handlers

Returns **PropSchema**

mapAsArray

src/types/mapAsArray.js:19-66

Similar to map, mapAsArray can be used to serialize a map-like collection where the key is contained in the 'value object'. Example: consider Map<id: number, customer: Customer> where the Customer object has the id stored on itself. mapAsArray stores all values from the map into an array which is serialized. Deserialization returns a ES6 Map or plain object object where the `keyPropertyName` of each object is used for keys. For ES6 maps this has the benefit of being allowed to have non-string keys in the map. The serialized json also may be slightly more compact.

Parameters

- `propSchema` **any**
- `keyPropertyName` **string** the property of stored objects used as key in the map
- `additionalArgs` **AdditionalPropArgs** optional object that contains beforeDeserialize and/or afterDeserialize handlers

Returns **PropSchema**

raw

src/types/raw.js:18-29

Indicates that this field is only need to putted in the serialized json or deserialized instance, without any transformations. Stay with its original value

Parameters

- `additionalArgs` **AdditionalPropArgs** optional object that contains beforeDeserialize and/or afterDeserialize handlers

Examples

```
createModelSchema(Model, {
    rawData: raw(),
});
```

```
console.dir(serialize(new Model({ rawData: { a: 1, b: [], c: {} } } })));
// outputs: { rawData: { a: 1, b: [], c: {} } } }
```

Returns **ModelSchema**

SKIP

src/constants.js:20-20

In the event that a property needs to be deserialized, but not serialized, you can use the SKIP symbol to omit the property. This has to be used with the custom serializer.

Examples

```
var schema = _.createSimpleSchema({
    a: _.custom(
        function(v) {
            return _.SKIP
        },
        function(v) {
            return v;
        }
    ),
});
t.deepEqual(_.serialize(s, { a: 4 }), { });
t.deepEqual(_.deserialize(s, { a: 4 }), { a: 4 });
```

Recipes and examples

1. Plain schema with plain objects

```
const todoSchema = {
    factory: () => {},
    props: {
        task: primitive(),
        owner: reference('_userId', UserStore.findUserById), // attribute of the
owner attribute of  a todo + lookup function
        subTasks: alias('children', list(object(todoSchema))),
    },
};

const todo = deserialize(
    todoSchema,
    { task: 'grab coffee', owner: 17, children: [] },
    (err, todo) => {
        console.log('finished loading todos');
    }
```

```
);

const todoJson = serialize(todoSchema, todo);
```

2. Create schema and store it on constructor

```
function Todo(parentTodo) {
    this.parent = parentTodo; // available in subTasks
}

const todoSchema = {
    factory: context => new Todo(context.parent),
    props: {
        task: primitive(),
        owner: reference('_userId', UserStore.findUserById), // attribute of the
owner attribute of  a todo + lookup function
        subTasks: alias('children', list(object(todoSchema))),
    },
};
setDefaultModelSchema(Todo, todoSchema);

const todo = deserialize(
    Todo, // just pass the constructor name, schema will be picked up
    { task: 'grab coffee', owner: 17, children: [] },
    (err, todos) => {
        console.log('finished loading todos');
    }
);

const todoJson = serialize(todo); // no need to pass schema explicitly
```

3. Create schema for simple argumentless constructors

```
function Todo() {}

// creates a default factory, () => new Todo(), stores the schema as default model
schema
createModelSchema(Todo, {
    task: primitive(),
});

const todo = deserialize(
    Todo, // just pass the constructor name, schema will be picked up
    { task: 'grab coffee', owner: 17, children: [] },
    (err, todos) => {
        console.log('finished loading todos');
    }
);
```

```
const todoJson = serialize(todo); // no need to pass schema explicitly
```

4. Create schema for simple argumentless constructors using decorators

```
class Todo {
    @serializable(primitive())
    task = 'Grab coffee';

    @serializable(reference('_userId', UserStore.findUserById))
    owner = null;

    @serializable(alias('children', list(object(todoSchema))))
    subTasks = [];
}

// note that (de)serialize also accepts lists
const todos = deserialize(
    Todo,
    [
        {
            task: 'grab coffee',
            owner: 17,
            children: [],
        },
    ],
    (err, todos) => {
        console.log('finished loading todos');
    }
);

const todoJson = serialize(todos);
```

5. use custom factory methods to reuse model object instances

```
const someTodoStoreById = {};

getDefaultModelSchema(Todo).factory = context => {
    const json = context.json;
    if (someTodoStoreById[json.id]) return someTodoStoreById[json.id]; // reuse
instance
    return (someTodoStoreById[json.id] = new Todo());
};
```

6. use custom arguments to inject stores to models

This pattern is useful to avoid singletons but allow to pass context specific data to constructors. This can be done by passing custom data to `deserialize` / `update` as last argument, which will be available as `context.args` on all places where context is available:

```
class User {
    constructor(someStore) {
        // User needs access to someStore, for whatever reason
    }
}

// create model schema with custom factory
createModelSchema(User, { username: true }, context => {
    return new User(context.args.someStore);
});

// don't want singletons!
const someStore = new SomeStore();
// provide somestore through context of the deserialization process
const user = deserialize(
    User,
    someJson,
    (err, user) => {
        console.log('done');
    },
    {
        someStore: someStore,
    }
);
```

7. Putting it together: MobX store with plain objects, classes and internal references

```
// models.js:
import { observable, computed } from 'mobx';
import { serializable, identifier } from 'serializr';

function randomId() {
    return Math.floor(Math.random() * 100000);
}

export class Box {
    @serializable(identifier()) id = randomId();
    @serializable @observable x = 0;
    @serializable @observable y = 0;
    @serializable @observable location = 0;

    constructor(location, x, y) {
        this.location = location;
        this.x = x;
        this.y = y;
```

```
    }

    @serializable @computed get area() {
        return this.x * this.y;
    }
}

export class Arrow {
    @serializable(identifier()) id = randomId();
    @serializable(reference(Box)) from;
    @serializable(reference(Box)) to;
}

// store.js:
import { observable, transaction } from 'mobx';
import {
    createSimpleSchema,
    identifier,
    list,
    serialize,
    deserialize,
    update,
} from 'serializr';
import { Box, Arrow } from './models';

// The store that holds our domain: boxes and arrows
const store = observable({
    boxes: [],
    arrows: [],
    selection: null,
});

// Model of the store itself
const storeModel = createSimpleSchema({
    boxes: list(object(Box)),
    arrows: list(object(Arrow)),
    selection: reference(Box),
});

// Example Data
// You can push data in as a class
store.boxes.push(new Box('Rotterdam', 100, 100), new Box('Vienna', 650, 300));

// Or it can be an raw javascript object with the right properties
store.arrows.push({
    id: randomId(),
    from: store.boxes[0],
    to: store.boxes[1],
});

// (de) serialize functions
function serializeState(store) {
    return serialize(storeModel, store);
}
```

```
function deserializeState(store, json) {
    transaction(() => {
        update(storeModel, store, json);
    });
}

// Print ... out for debugging
console.dir(serializeState(store), { depth: 10, colors: true });
```

Future ideas

- [ ] If MobX, optimize by leveraging createTransformer and transactions
- [ ] Support async serialization (future)
- [ ] Support ImmutableJS out of the box
- [ ] Make `"*": true` respect extends clauses