Schema

Schema()

Parameters

- [definition] «Object|Schema|Array» Can be one of: object describing schema paths, or schema to copy, or array of objects and schemas
- [options] «Object»

Inherits:

- «NodeJS EventEmitter http://nodejs.org/api/events.html#events_class_events_eventemitter»

Schema constructor.

Example:

```
const child = new Schema({ name: String });
const schema = new Schema({ name: String, age: Number, children: [child] });
const Tree = mongoose.model('Tree', schema);

// setting schema options
new Schema({ name: String }, { _id: false, autoIndex: false })
```

Options:

- autoIndex: bool - defaults to null (which means use the connection's autoIndex option)
- autoCreate: bool - defaults to null (which means use the connection's autoCreate option)
- bufferCommands: bool - defaults to true
- bufferTimeoutMS: number - defaults to 10000 (10 seconds). If `bufferCommands` is enabled, the amount of time Mongoose will wait for connectivity to be restablished before erroring out.
- capped: bool - defaults to false
- collection: string - no default
- discriminatorKey: string - defaults to `__t`
- id: bool - defaults to true
- _id: bool - defaults to true
- minimize: bool - controls document#toObject behavior when called manually - defaults to true
- read: string
- writeConcern: object - defaults to null, use to override the MongoDB server's default write concern settings
- shardKey: object - defaults to `null`
- strict: bool - defaults to true
- strictQuery: bool - defaults to false
- toJSON - object - no default
- toObject - object - no default
- typeKey - string - defaults to 'type'
- validateBeforeSave - bool - defaults to `true`
- versionKey: string or object - defaults to "__v"

- optimisticConcurrency: bool - defaults to false. Set to true to enable optimistic concurrency.
- collation: object - defaults to null (which means use no collation)
- selectPopulatedPaths: boolean - defaults to `true`
- skipVersioning: object - paths to exclude from versioning
- timestamps: object or boolean - defaults to `false`. If true, Mongoose adds `createdAt` and `updatedAt` properties to your schema and manages those properties for you.

Options for Nested Schemas:

- `excludeIndexes`: bool - defaults to `false`. If `true`, skip building indexes on this schema's paths.

Note:

When nesting schemas, (`children` in the example above), always declare the child schema first before passing it into its parent.

Schema.Types

Type:

- «property»

The various built-in Mongoose Schema Types.

Example:

```
const mongoose = require('mongoose');
const ObjectId = mongoose.Schema.Types.ObjectId;
```

Types:

- String
- Number
- Boolean | Bool
- Array
- Buffer
- Date
- ObjectId | Oid
- Mixed

Using this exposed access to the `Mixed` SchemaType, we can use them in our schema.

```
const Mixed = mongoose.Schema.Types.Mixed;
new mongoose.Schema({ _user: Mixed })
```

Schema.indexTypes

Type:

- «property»

The allowed index types

Schema.prototype.add()

Parameters

- obj «Object|Schema» plain object with paths to add, or another schema
- [prefix] «String» path to prefix the newly added paths with

Returns:

- «Schema» the Schema instance

Adds key path / schema type pairs to this schema.

Example:

```
const ToySchema = new Schema();
ToySchema.add({ name: 'string', color: 'string', price: 'number' });

const TurboManSchema = new Schema();
// You can also `add()` another schema and copy over all paths, virtuals,
// getters, setters, indexes, methods, and statics.
TurboManSchema.add(ToySchema).add({ year: Number });
```

Schema.prototype.childSchemas

Type:

- «property»

Array of child schemas (from document arrays and single nested subdocs) and their corresponding compiled models. Each element of the array is an object with 2 properties: `schema` and `model`.

This property is typically only useful for plugin authors and advanced users. You do not need to interact with this property at all to use mongoose.

Schema.prototype.clone()

Returns:

- «Schema» the cloned schema

Returns a deep copy of the schema

Example:

```
const schema = new Schema({ name: String });
const clone = schema.clone();
clone === schema; // false
clone.path('name'); // SchemaString { ... }
```

Schema.prototype.eachPath()

Parameters

- fn «Function» callback function

Returns:

- «Schema» this

Iterates the schemas paths similar to Array#forEach.

The callback is passed the pathname and the schemaType instance.

Example:

```
const userSchema = new Schema({ name: String, registeredAt: Date });
userSchema.eachPath((pathname, schematype) => {
  // Prints twice:
  // name SchemaString { ... }
  // registeredAt SchemaDate { ... }
  console.log(pathname, schematype);
});
```

Schema.prototype.get()

Parameters

- key «String» option name

Returns:

- «Any» the option's value

Gets a schema option.

Example:

```
schema.get('strict'); // true
schema.set('strict', false);
schema.get('strict'); // false
```

Schema.prototype.index()

Parameters

- fields «Object»
- [options] «Object» Options to pass to MongoDB driver's `createIndex()` function
- | «String» number} [options.expires=null] Mongoose-specific syntactic sugar, uses ms to convert `expires` option into seconds for the `expireAfterSeconds` in the above link.

Defines an index (most likely compound) for this schema.

Example

```
schema.index({ first: 1, last: -1 })
```

Schema.prototype.indexes()

Returns:

- «Array» list of indexes defined in the schema

Returns a list of indexes that this schema declares, via `schema.index()` or by `index: true` in a path's options. Indexes are expressed as an array `[spec, options]`.

Example:

```
const userSchema = new Schema({
  email: { type: String, required: true, unique: true },
  registeredAt: { type: Date, index: true }
});

// [ [ { email: 1 }, { unique: true, background: true } ],
//   [ { registeredAt: 1 }, { background: true } ] ]
userSchema.indexes();
```

Plugins can use the return value of this function to modify a schema's indexes. For example, the below plugin makes every index unique by default.

```
function myPlugin(schema) {
  for (const index of schema.indexes()) {
    if (index[1].unique === undefined) {
      index[1].unique = true;
    }
  }
}
```

Schema.prototype.loadClass()

Parameters

- model «Function»
- [virtualsOnly] «Boolean» if truthy, only pulls virtuals from the class, not methods or statics

Loads an ES6 class into a schema. Maps setters + getters, static methods, and instance methods to schema virtuals, statics, and methods.

Example:

```
const md5 = require('md5');
const userSchema = new Schema({ email: String });
class UserClass {
  // `gravatarImage` becomes a virtual
  get gravatarImage() {
    const hash = md5(this.email.toLowerCase());
    return `https://www.gravatar.com/avatar/${hash}`;
  }

  // `getProfileUrl()` becomes a document method
  getProfileUrl() {
    return `https://mysite.com/${this.email}`;
  }

  // `findByEmail()` becomes a static
  static findByEmail(email) {
    return this.findOne({ email });
  }
}

// `schema` will now have a `gravatarImage` virtual, a `getProfileUrl()` method,
// and a `findByEmail()` static
userSchema.loadClass(UserClass);
```

Schema.prototype.method()

Parameters

- method «String|Object» name
- [fn] «Function»

Adds an instance method to documents constructed from Models compiled from this schema.

Example

```
const schema = kittySchema = new Schema(..);

schema.method('meow', function () {
  console.log('meeeeeooooooooooooow');
})

const Kitty = mongoose.model('Kitty', schema);

const fizz = new Kitty;
fizz.meow(); // meeeeeooooooooooooow
```

If a hash of name/fn pairs is passed as the only argument, each name/fn pair will be added as methods.

```
schema.method({
    purr: function () {}
  , scratch: function () {}
});

// later
fizz.purr();
fizz.scratch();
```

NOTE: `Schema.method()` adds instance methods to the `Schema.methods` object. You can also add instance methods directly to the `Schema.methods` object as seen in the guide

Schema.prototype.obj

Type:

- «property»

The original object passed to the schema constructor

Example:

```
const schema = new Schema({ a: String }).add({ b: String });
schema.obj; // { a: String }
```

Schema.prototype.path()

Parameters

- path «String»
- constructor «Object»

Gets/sets schema paths.

Sets a path (if arity 2) Gets a path (if arity 1)

Example

```
schema.path('name') // returns a SchemaType
schema.path('name', Number) // changes the schemaType of `name` to Number
```

Schema.prototype.pathType()

Parameters

- path «String»

Returns:

- «String»

Returns the pathType of `path` for this schema.

Given a path, returns whether it is a real, virtual, nested, or ad-hoc/undefined path.

Example:

```
const s = new Schema({ name: String, nested: { foo: String } });
s.virtual('foo').get(() => 42);
s.pathType('name'); // "real"
s.pathType('nested'); // "nested"
s.pathType('foo'); // "virtual"
s.pathType('fail'); // "adhocOrUndefined"
```

Schema.prototype.paths

Type:

- «property»

The paths defined on this schema. The keys are the top-level paths in this schema, and the values are instances of the SchemaType class.

Example:

```
const schema = new Schema({ name: String }, { _id: false });
schema.paths; // { name: SchemaString { ... } }

schema.add({ age: Number });
schema.paths; // { name: SchemaString { ... }, age: SchemaNumber { ... } }
```

Schema.prototype.pick()

Parameters

- paths «Array» list of paths to pick
- [options] «Object» options to pass to the schema constructor. Defaults to `this.options` if not set.

Returns:

- «Schema»

Returns a new schema that has the picked `paths` from this schema.

This method is analagous to Lodash's `pick()` function for Mongoose schemas.

Example:

```
const schema = Schema({ name: String, age: Number });
// Creates a new schema with the same `name` path as `schema`,
// but no `age` path.
const newSchema = schema.pick(['name']);

newSchema.path('name'); // SchemaString { ... }
newSchema.path('age'); // undefined
```

Schema.prototype.plugin()

Parameters

- plugin «Function» callback
- [opts] «Object»

Registers a plugin for this schema.

Example:

```
const s = new Schema({ name: String });
s.plugin(schema => console.log(schema.path('name').path));
mongoose.model('Test', s); // Prints 'name'
```

Schema.prototype.post()

Parameters

- The «String|RegExp» method name or regular expression to match method name
- [options] «Object»
  - [options.document] «Boolean» If `name` is a hook for both document and query middleware, set to `true` to run on document middleware.
  - [options.query] «Boolean» If `name` is a hook for both document and query middleware, set to `true` to run on query middleware.
- fn «Function» callback

Defines a post hook for the document

```
const schema = new Schema(..);
schema.post('save', function (doc) {
  console.log('this fired after a document was saved');
});

schema.post('find', function(docs) {
  console.log('this fired after you ran a find query');
});

schema.post(/Many$/, function(res) {
  console.log('this fired after you ran `updateMany()` or `deleteMany()`');
});

const Model = mongoose.model('Model', schema);

const m = new Model(..);
m.save(function(err) {
  console.log('this fires after the `post` hook');
});

m.find(function(err, docs) {
  console.log('this fires after the post find hook');
});
```

Schema.prototype.pre()

Parameters

- The «String|RegExp» method name or regular expression to match method name
- [options] «Object»
  - [options.document] «Boolean» If `name` is a hook for both document and query middleware, set to `true` to run on document middleware. For example, set `options.document` to `true` to apply this hook to `Document#deleteOne()` rather than `Query#deleteOne()`.
  - [options.query] «Boolean» If `name` is a hook for both document and query middleware, set to `true` to run on query middleware.
- callback «Function»

Defines a pre hook for the model.

Example

```
const toySchema = new Schema({ name: String, created: Date });

toySchema.pre('save', function(next) {
  if (!this.created) this.created = new Date;
  next();
});

toySchema.pre('validate', function(next) {
  if (this.name !== 'Woody') this.name = 'Woody';
  next();
});

// Equivalent to calling `pre()` on `find`, `findOne`, `findOneAndUpdate`.
toySchema.pre(/^find/, function(next) {
  console.log(this.getFilter());
});

// Equivalent to calling `pre()` on `updateOne`, `findOneAndUpdate`.
toySchema.pre(['updateOne', 'findOneAndUpdate'], function(next) {
  console.log(this.getFilter());
});

toySchema.pre('deleteOne', function() {
  // Runs when you call `Toy.deleteOne()`
});

toySchema.pre('deleteOne', { document: true }, function() {
  // Runs when you call `doc.deleteOne()`
});
```

Schema.prototype.queue()

Parameters

- name «String» name of the document method to call later

- args «Array» arguments to pass to the method

Adds a method call to the queue.

Example:

```
schema.methods.print = function() { console.log(this); };
schema.queue('print', []); // Print the doc every one is instantiated

const Model = mongoose.model('Test', schema);
new Model({ name: 'test' }); // Prints '{"_id": ..., "name": "test" }'
```

Schema.prototype.remove()

Parameters

- path «String|Array»

Returns:

- «Schema» the Schema instance

Removes the given `path` (or [`paths`]).

Example:

```
const schema = new Schema({ name: String, age: Number });
schema.remove('name');
schema.path('name'); // Undefined
schema.path('age'); // SchemaNumber { ... }
```

Schema.prototype.requiredPaths()

Parameters

- invalidate «Boolean» refresh the cache

Returns:

- «Array»

Returns an Array of path strings that are required by this schema.

Example:

```
const s = new Schema({
  name: { type: String, required: true },
```

```
  age: { type: String, required: true },
  notes: String
});
s.requiredPaths(); // [ 'age', 'name' ]
```

Schema.prototype.set()

Parameters

- key «String» option name
- [value] «Object» if not passed, the current option value is returned

Sets a schema option.

Example

```
schema.set('strict'); // 'true' by default
schema.set('strict', false); // Sets 'strict' to false
schema.set('strict'); // 'false'
```

Schema.prototype.static()

Parameters

- name «String|Object»
- [fn] «Function»

Adds static "class" methods to Models compiled from this schema.

Example

```
const schema = new Schema(..);
// Equivalent to `schema.statics.findByName = function(name) {}`;
schema.static('findByName', function(name) {
  return this.find({ name: name });
});

const Drink = mongoose.model('Drink', schema);
await Drink.findByName('LaCroix');
```

If a hash of name/fn pairs is passed as the only argument, each name/fn pair will be added as statics.

Schema.prototype.virtual()

Parameters

- name «String»
- [options] «Object»
  - [options.ref] «String|Model» model name or model instance. Marks this as a populate virtual.
  - [options.localField] «String|Function» Required for populate virtuals. See populate virtual docs for more information.
  - [options.foreignField] «String|Function» Required for populate virtuals. See populate virtual docs for more information.
  - [options.justOne=false] «Boolean|Function» Only works with populate virtuals. If truthy, will be a single doc or `null`. Otherwise, the populate virtual will be an array.
  - [options.count=false] «Boolean» Only works with populate virtuals. If truthy, this populate virtual will contain the number of documents rather than the documents themselves when you `populate()`.
  - [options.get=null] «Function|null» Adds a getter to this virtual to transform the populated doc.

Returns:

- «VirtualType»

Creates a virtual type with the given name.

Schema.prototype.virtualpath()

Parameters

- name «String»

Returns:

- «VirtualType»

Returns the virtual type with the given `name`.

Schema.prototype.virtuals

Type:

- «property»

Object containing all virtuals defined on this schema. The objects' keys are the virtual paths and values are instances of `VirtualType`.

This property is typically only useful for plugin authors and advanced users. You do not need to interact with this property at all to use mongoose.

Example:

```
const schema = new Schema({});
schema.virtual('answer').get(() => 42);

console.log(schema.virtuals); // { answer: VirtualType { path: 'answer', ... } }
console.log(schema.virtuals['answer'].getters[0].call()); // 42
```

Schema.reserved

Type:

- «property»

Reserved document keys.

Keys in this object are names that are warned in schema declarations because they have the potential to break Mongoose/ Mongoose plugins functionality. If you create a schema using `new Schema()` with one of these property names, Mongoose will log a warning.

- _posts
- _pres
- collection
- emit
- errors
- get
- init
- isModified
- isNew
- listeners
- modelName
- on
- once
- populated
- prototype
- remove
- removeListener

- save
- schema
- toObject
- validate

NOTE: Use of these terms as method names is permitted, but play at your own risk, as they may be existing mongoose document methods you are stomping on.

```
const schema = new Schema(..);
 schema.methods.init = function () {} // potentially breaking
```